Package ‘sparklyr’

April 29, 2024

Type Package

Title R Interface to Apache Spark

Version 1.8.6

Maintainer Edgar Ruiz <edgar@rstudio.com>

Description R interface to Apache Spark, a fast and general engine for big data processing, see <https://spark.apache.org/>. This package supports connecting to local and remote Apache Spark clusters, provides a ‘dplyr’ compatible back-end, and provides an interface to Spark’s built-in machine learning algorithms.

License Apache License 2.0 | file LICENSE

URL https://spark.posit.co/

BugReports https://github.com/sparklyr/sparklyr/issues

Depends R (>= 3.2)

Imports config (>= 0.2), DBI (>= 1.0.0), dbplyr (>= 2.5.0), dplyr (>= 1.0.9), generics, globals, glue, httr (>= 1.2.1), jsonlite (>= 1.4), methods, openssl (>= 0.8), purr, rlang (>= 0.1.4), rstudioapi (>= 0.10), tidyselect, uuid, vctrs, withr, xml2

Suggests arrow (>= 0.17.0), broom, diffobj, foreach, ggplot2, iterators, janeaustenr, Lahman, mlbench, nnet, nycflights13, R6, r2d3, RCurl, reshape2, shiny (>= 1.0.1), parsnip, testthat, rprojroot

Encoding UTF-8

RoxygenNote 7.3.1

SystemRequirements Spark: 2.x, or 3.x

'ml_feature_stop_words_remover.R' 'ml_feature_string_indexer.R'
'ml_feature_string_indexer_model.R' 'ml_feature_tokenizer.R'
'ml_feature_vector_assembler.R' 'ml_feature_vector_indexer.R'
'ml_feature_vector_slicer.R' 'ml_feature_word2vec.R'
'ml_fpm_fpgrowth.R' 'ml_fpm_prefixspan.R' 'ml_helpers.R'
'ml_mapping_tables.R' 'ml_metrics.R' 'ml_model_als.R'
'ml_model_bisecting_kmeans.R' 'ml_model_constructors.R'
'ml_model_decision_tree.R' 'ml_model_gaussian_mixture.R'
'ml_model_generalized_linear_regression.R'
'ml_model_gradient_boosted_trees.R'
'ml_model_isotonic_regression.R' 'ml_model_kmeans.R'
'ml_model_lda.R' 'ml_model_linear_regression.R'
'ml_model_linear_svc.R' 'ml_model_logistic_regression.R'
'ml_model_naive_bayes.R' 'ml_model_one_vs_rest.R'
'ml_model_random_forest.R' 'ml_model_utils.R'
'ml_param_utils.R' 'ml_persistence.R' 'ml_pipeline.R'
'ml_pipeline_utils.R' 'ml_print_utils.R'
'ml_recommendation_als.R'
'ml_regression_aft_survival_regression.R'
'ml_regression_decision_tree_regressor.R'
'ml_regression_gbt_regressor.R'
'ml_regression_generalized_linear_regression.R'
'ml_regression_isotonic_regression.R'
'ml_regression_linear_regression.R'
'ml_regression_random_forest_regressor.R' 'ml_stat.R'
'ml_summary.R' 'ml_transformation_methods.R'
'ml_transformer_and_estimator.R' 'ml_tuning.R'
'ml_tuning_cross_validator.R'
'ml_tuning_train_validation_split.R' 'ml_utils.R'
'ml_validator_utils.R' 'mutation.R' 'na_actions.R'
'new_model_multilayer_perceptron.R' 'params_validator.R'
'precondition.R' 'project_template.R' 'qubole_connection.R'
'reexports.R' 'sdf_dim.R' 'sdf_distinct.R' 'sdf_ml.R'
'sdf_saveload.R' 'sdf_sequence.R' 'sdf_stat.R'
'sdf_streaming.R' 'tidyr_utils.R' 'sdf_unnest_longer.R'
'sdf_wrapper.R' 'sdf_unnest_wider.R' 'sdf_utils.R'
'spark_compile.R' 'spark_context_config.R' 'spark_extensions.R'
'spark_gateway.R' 'spark_gen_embedded_sources.R'
'spark_globs.R' 'spark_hive.R' 'spark_home.R' 'spark_ide.R'
'spark_submix.R' 'spark_update_embedded_sources.R'
'spark_utils.R' 'spark_verify_embedded_sources.R'
'stream_data.R' 'stream_job.R' 'stream_operations.R'
'stream_shiny.R' 'stream_view.R' 'synapse_connection.R'
'test_connection.R' 'tidiers_ml_aft_survival_regression.R'
'tidiers_ml_als.R' 'tidiers_ml_isotonic_regression.R'
'tidiers_ml_lda.R' 'tidiers_ml_linear_models.R'
'tidiers_ml_logistic_regression.R'
'tidiers_ml_multilayer_perceptron.R' 'tidiers_ml_naive_bayes.R'
R topics documented:

'tidiers_ml_svc_models.R' 'tidiers_ml_tree_models.R'
'tidiers_ml_unsupervised_models.R' 'tidiers_pca.R'
'tidiers_utils.R' 'tidyr_fill.R' 'tidyr_nest.R'
'tidy_pivot_utils.R' 'tidy_pivot_longer.R'
'tidy_pivot_wider.R' 'tidy_separate.R' 'tidy_unite.R'
'tidy_unnest.R' 'worker_apply.R' 'worker_connect.R'
'worker_connection.R' 'worker_invoke.R' 'worker_log.R'
'worker_main.R' 'yarn_cluster.R' 'yarn_config.R' 'yarn_ui.R'
'zzz.R'

NeedsCompilation no

Author  Javier Luraschi [aut],
         Kevin Kuo [aut] (<https://orcid.org/0000-0001-7803-7901>),
         Kevin Ushey [aut],
         JJ Allaire [aut],
         Samuel Macedo [ctb],
         Hossein Falaki [aut],
         Lu Wang [aut],
         Andy Zhang [aut],
         Yitao Li [aut] (<https://orcid.org/0000-0002-1261-905X>),
         Jozef Hajnala [ctb],
         Maciej Szymbkiewicz [ctb] (<https://orcid.org/0000-0003-1469-9396>),
         Wil Davis [ctb],
         Edgar Ruiz [aut, cre],
         RStudio [cph],
         The Apache Software Foundation [aut, cph]

Repository CRAN

Date/Publication  2024-04-29 17:30:02 UTC

R topics documented:

  checkpoint_directory ............................................. 10
  collect_from_rds .................................................... 11
  compile_package_jars ............................................. 11
  connection_config .................................................. 12
  copy_to.spark_connection ......................................... 12
  distinct ......................................................... 13
  download_scalac .................................................... 14
  dplyr_hof .......................................................... 14
  ensure ........................................................... 14
  fill ............................................................. 15
  filter ............................................................ 15
  find_scalac ......................................................... 15
  ft_binarizer ......................................................... 16
  ft_bucketizer ........................................................ 17
  ft_chisq_selector .................................................. 19
  ft_count_vectorizer ............................................... 20
R topics documented:

- `ft_dct` ................................................. 22
- `ft_elementwise_product` .......................... 23
- `ft_feature_hasher` ................................. 24
- `ft_hashing_tf` ..................................... 26
- `ft_idf` ............................................. 27
- `ft_imputer` .......................................... 28
- `ft_index_to_string` ............................... 29
- `ft_interaction` ...................................... 30
- `ft_lsh` .............................................. 31
- `ft_lsh_utils` ....................................... 33
- `ft_max_abs_scaler` ............................... 34
- `ft_min_max_scaler` ............................... 35
- `ft_ngram` ......................................... 37
- `ft_normalizer` ...................................... 38
- `ft_one_hot_encoder` .............................. 39
- `ft_one_hot_encoder_estimator` .................. 40
- `ft_pca` ............................................ 41
- `ft_polynomial_expansion` ....................... 43
- `ft_quantile_discretizer` ......................... 44
- `ft_regex_tokenizer` .............................. 46
- `ft_robust_scaler` .................................. 47
- `ft_r_formula` ....................................... 48
- `ft_sql_transformer` ............................... 50
- `ft_standard_scaler` ............................... 51
- `ft_stop_words_remover` ........................... 53
- `ft_string_indexer` .................................. 54
- `ft_tokenizer` ....................................... 56
- `ft_vector_assembler` ............................. 57
- `ft_vector_indexer` .................................. 58
- `ft_vector_slicer` ................................... 59
- `ft_word2vec` ...................................... 60
- `full_join` .......................................... 61
- `generic_call_interface` ......................... 62
- `get_spark_sql_catalog_implementation` ........ 62
- `hive_context_config` ............................. 63
- `hof_aggregate` ..................................... 63
- `hof_array_sort` .................................... 64
- `hof_exists` ........................................ 65
- `hof_filter` ........................................ 65
- `hof_forall` ....................................... 66
- `hof_map_filter` .................................... 67
- `hof_map_zip_with` .................................. 68
- `hof_transform` ..................................... 69
- `hof_transform_keys` .............................. 70
- `hof_transform_values` ............................ 71
- `hof_zip_with` ...................................... 71
- `inner_join` ....................................... 72
- `invoke` ............................................ 73
jarray ................................................................. 73
jfloat ................................................................. 74
jfloat_array ......................................................... 75
join.tbl.spark ....................................................... 75
j_invoke ............................................................... 77
left_join ............................................................. 78
list.sparklyr.jars ................................................. 78
livy_config .......................................................... 78
livy.service.start .................................................. 80
ml-params ........................................................... 80
ml-persistence ....................................................... 81
ml-transform-methods .............................................. 82
ml-tuning ............................................................ 83
ml.aft.survival.regression ........................................ 85
ml.als ............................................................... 88
ml.als.tidiers ....................................................... 91
ml.bisecting.kmeans .............................................. 91
ml.chisquare.test .................................................. 93
ml.clustering.evaluator .......................................... 94
ml.corr .............................................................. 95
ml.decision.tree.classifier ....................................... 96
ml.default.stop.words ............................................ 100
ml.evaluate ........................................................ 101
ml.evaluator ........................................................ 102
ml.feature.importances .......................................... 104
ml.fpgrowth ........................................................ 105
ml.gaussian.mixture .............................................. 106
ml.gbt.classifier .................................................. 107
ml.generalized.linear.regression ................................ 112
ml.glm.tidiers ....................................................... 114
ml.isotonic.regression ........................................... 116
ml.isotonic.regression.tidiers .................................. 117
ml.kmeans .......................................................... 118
ml.kmeans.cluster.eval .......................................... 120
ml.lda .............................................................. 120
ml.lda.tidiers ...................................................... 124
ml.linear.regression .............................................. 125
ml.linear.svc ....................................................... 127
ml.linear.svc.tidiers ............................................. 129
ml.logistic.regression ............................................ 129
ml.logistic.regression.tidiers ................................... 132
ml.metrics.binary .................................................. 133
ml.metrics.multiclass ............................................ 134
ml.metrics.regression ............................................. 135
ml.model.data ...................................................... 136
ml.multilayer.perceptron.classifier ............................. 137
ml.multilayer.perceptron.tidiers ................................ 140
ml.naive.bayes ..................................................... 140
R topics documented:

- ml_naive_bayes_tidiers .................................................. 142
- ml_one_vs_rest ........................................................... 143
- ml_pca_tidiers ............................................................. 144
- ml_pipeline ................................................................. 145
- ml_power_iteration ....................................................... 145
- ml_prefixspan ............................................................. 148
- ml_random_forest_classifier ........................................... 149
- ml_stage ................................................................. 153
- ml_summary .............................................................. 154
- ml_survival_regression_tidiers ........................................ 154
- ml_tree_tidiers ......................................................... 155
- ml_uid ................................................................. 157
- ml_unsupervised_tidiers ................................................. 157
- mutate ................................................................. 158
- na.replace .............................................................. 158
- nest ................................................................. 158
- pivot_longer ............................................................. 159
- pivot_wider .............................................................. 159
- random_string ........................................................... 159
- reactiveSpark ............................................................. 160
- registerDoSpark .......................................................... 160
- register_extension ........................................................ 161
- replace_na .............................................................. 161
- right_join .............................................................. 161
- sdf-saveload ............................................................. 162
- sdf-transform-methods ................................................... 162
- sdf_along ............................................................... 163
- sdf_bind ................................................................ 163
- sdf_broadcast ........................................................... 164
- sdf_checkpoint .......................................................... 165
- sdf_coalesce ............................................................ 165
- sdf_collect ............................................................. 166
- sdf_copy_to ............................................................. 166
- sdf_crosstab ............................................................ 167
- sdf_debug_string ......................................................... 168
- sdf_describe ............................................................ 168
- sdf_dim ................................................................. 169
- sdf_distinct ............................................................. 169
- sdf_drop_duplicates ..................................................... 170
- sdf_expand_grid .......................................................... 170
- sdf_from_avro ........................................................... 171
- sdf_is_streaming ......................................................... 171
- sdf_last_index .......................................................... 172
- sdf_len ................................................................. 172
- sdf_num_partitions ....................................................... 173
- sdf_partition_sizes ..................................................... 173
- sdfPersist .............................................................. 174
- sdf_pivot .............................................................. 174
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdf_project</td>
<td>175</td>
</tr>
<tr>
<td>sdf_quartile</td>
<td>176</td>
</tr>
<tr>
<td>sdf_random_split</td>
<td>177</td>
</tr>
<tr>
<td>sdf_rbeta</td>
<td>178</td>
</tr>
<tr>
<td>sdf_rbinom</td>
<td>179</td>
</tr>
<tr>
<td>sdf_rcauchy</td>
<td>180</td>
</tr>
<tr>
<td>sdf_rchisq</td>
<td>181</td>
</tr>
<tr>
<td>sdf_read_column</td>
<td>181</td>
</tr>
<tr>
<td>sdf_register</td>
<td>182</td>
</tr>
<tr>
<td>sdf_repartition</td>
<td>182</td>
</tr>
<tr>
<td>sdf_residuals.ml_model_generalized_linear_regression</td>
<td>183</td>
</tr>
<tr>
<td>sdf_rexp</td>
<td>183</td>
</tr>
<tr>
<td>sdf_rgamma</td>
<td>184</td>
</tr>
<tr>
<td>sdf_rgeom</td>
<td>185</td>
</tr>
<tr>
<td>sdf_rhyper</td>
<td>186</td>
</tr>
<tr>
<td>sdf_rlnorm</td>
<td>187</td>
</tr>
<tr>
<td>sdf_rnorm</td>
<td>188</td>
</tr>
<tr>
<td>sdf_rpois</td>
<td>189</td>
</tr>
<tr>
<td>sdf_rt</td>
<td>189</td>
</tr>
<tr>
<td>sdf_runif</td>
<td>190</td>
</tr>
<tr>
<td>sdf_rweibull</td>
<td>191</td>
</tr>
<tr>
<td>sdf_sample</td>
<td>192</td>
</tr>
<tr>
<td>sdf_schema</td>
<td>192</td>
</tr>
<tr>
<td>sdf_separate_column</td>
<td>193</td>
</tr>
<tr>
<td>sdf_seq</td>
<td>193</td>
</tr>
<tr>
<td>sdf_sort</td>
<td>194</td>
</tr>
<tr>
<td>sdf_sql</td>
<td>194</td>
</tr>
<tr>
<td>sdf_to_avro</td>
<td>195</td>
</tr>
<tr>
<td>sdf_unnest_longer</td>
<td>195</td>
</tr>
<tr>
<td>sdf_unnest_wider</td>
<td>196</td>
</tr>
<tr>
<td>sdf_weighted_sample</td>
<td>198</td>
</tr>
<tr>
<td>sdf_with_sequential_id</td>
<td>198</td>
</tr>
<tr>
<td>sdf_with_unique_id</td>
<td>199</td>
</tr>
<tr>
<td>select</td>
<td>199</td>
</tr>
<tr>
<td>separate</td>
<td>199</td>
</tr>
<tr>
<td>spark-api</td>
<td>200</td>
</tr>
<tr>
<td>spark-connections</td>
<td>201</td>
</tr>
<tr>
<td>sparklyr_get_backend_port</td>
<td>203</td>
</tr>
<tr>
<td>spark_adaptive_query_execution</td>
<td>203</td>
</tr>
<tr>
<td>spark_advisory_shuffle_partition_size</td>
<td>204</td>
</tr>
<tr>
<td>spark_apply</td>
<td>204</td>
</tr>
<tr>
<td>spark_apply_bundle</td>
<td>207</td>
</tr>
<tr>
<td>spark_apply_log</td>
<td>207</td>
</tr>
<tr>
<td>spark_auto_broadcast_join_threshold</td>
<td>208</td>
</tr>
<tr>
<td>spark_coalesce_initial_num_partitions</td>
<td>208</td>
</tr>
<tr>
<td>spark_coalesce_min_num_partitions</td>
<td>209</td>
</tr>
<tr>
<td>spark_coalesce_shuffle_partitions</td>
<td>209</td>
</tr>
<tr>
<td>spark_compilation_spec</td>
<td>210</td>
</tr>
</tbody>
</table>
topics documented:

spark_config ........................................... 211
spark_config_kubernetes ................................. 212
spark_config_settings ................................. 213
spark_connection ........................................ 213
spark_connection-class ................................. 213
spark_connection_find ................................. 214
spark_connect_method ................................. 214
spark_context_config .................................. 215
spark_dataframe ......................................... 216
spark_default_compilation_spec ..................... 216
spark_dependency ....................................... 217
spark_dependency_fallback ......................... 218
spark_extension ........................................ 218
spark_home_set .......................................... 219
spark_ide_connection_open ............................. 219
spark_insert_table ..................................... 221
spark_install ............................................ 222
spark_integ_test_skip .................................. 223
spark_job .................................................. 223
spark_job-class .......................................... 224
spark_last_error ......................................... 224
spark_load_table ........................................ 224
spark_log ................................................. 225
spark_read ............................................... 226
spark_read_avro ......................................... 227
spark_read_binary ....................................... 228
spark_read_csv .......................................... 229
spark_read_delta ....................................... 231
spark_read_image ........................................ 232
spark_read_jdbc .......................................... 233
spark_read_json .......................................... 234
spark_read_libsvm ....................................... 235
spark_read_orc .......................................... 236
spark_read_parquet ..................................... 238
spark_read_source ...................................... 239
spark_read_table ....................................... 240
spark_read_text ......................................... 241
spark_save_table ....................................... 242
spark_session_config .................................. 243
spark_statistical_routines ......................... 243
spark_table_name ....................................... 244
spark_version ............................................ 244
spark_version_from_home ............................... 245
spark_web ................................................ 245
spark_write ............................................. 246
spark_write_avro ....................................... 247
spark_write_csv ....................................... 248
spark_write_delta ..................................... 249
checkpoint_directory

Set/Get Spark checkpoint directory

Description

Set/Get Spark checkpoint directory

Usage

spark_set_checkpoint_dir(sc, dir)

spark_get_checkpoint_dir(sc)
collect_from_rds

Arguments

sc  A spark_connection.
dir  checkpoint directory, must be HDFS path of running on cluster

Description

Deserialize Spark data that is serialized using 'spark_write_rds()' into a R dataframe.

Usage

collect_from_rds(path)

Arguments

path  Path to a local RDS file that is produced by 'spark_write_rds()' (RDS files stored in HDFS will need to be downloaded to local filesystem first (e.g., by running 'hadoop fs -copyToLocal ...' or similar)

See Also

Other Spark serialization routines: spark_insert_table(), spark_load_table(), spark_read(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(), spark_read_image(), spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(), spark_read_parquet(), spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(), spark_write_avro(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(), spark_write_orc(), spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()

compile_package_jars  Compile Scala sources into a Java Archive (jar)

Description

Compile the scala source files contained within an R package into a Java Archive (jar) file that can be loaded and used within a Spark environment.

Usage

compile_package_jars(..., spec = NULL)
copy_to.spark_connection

Arguments

... Optional compilation specifications, as generated by spark_compilation_spec. When no arguments are passed, spark_default_compilation_spec is used instead.

spec An optional list of compilation specifications. When set, this option takes precedence over arguments passed to ....

copy_to.spark_connection  Copy an R Data Frame to Spark

Description

Copy an R data.frame to Spark, and return a reference to the generated Spark DataFrame as a tbl_spark. The returned object will act as a dplyr-compatible interface to the underlying Spark table.

connection_config  Read configuration values for a connection

Description

Read configuration values for a connection

Usage

connection_config(sc, prefix, not_prefix = list())

Arguments

sc  spark_connection

prefix  Prefix to read parameters for (e.g. spark.context., spark.sql., etc.)

not_prefix  Prefix to not include.

Value

Named list of config parameters (note that if a prefix was specified then the names will not include the prefix)
Usage

```r
## S3 method for class 'spark_connection'

copy_to(
  dest,
  df,
  name = spark_table_name(substitute(df)),
  overwrite = FALSE,
  memory = TRUE,
  repartition = 0L,
  ...
)
```

Arguments

- **dest**: A `spark_connection`.
- **df**: An R `data.frame`.
- **name**: The name to assign to the copied table in Spark.
- **overwrite**: Boolean; overwrite a pre-existing table with the name `name` if one already exists?
- **memory**: Boolean; should the table be cached into memory?
- **repartition**: The number of partitions to use when distributing the table across the Spark cluster. The default (0) can be used to avoid partitioning.
- **...**: Optional arguments; currently unused.

Value

A `tbl_spark`, representing a `dplyr`-compatible interface to a Spark DataFrame.

---

**distinct**

**Distinct**

Description

See `distinct` for more details.
download_scalac | Downloads default Scala Compilers

**Description**

compile_package_jars requires several versions of the scala compiler to work, this is to match Spark scala versions. To help setup your environment, this function will download the required compilers under the default search path.

**Usage**

download_scalac(dest_path = NULL)

**Arguments**

dest_path | The destination path where scalac will be downloaded to.

**Details**

See find_scalac for a list of paths searched and used by this function to install the required compilers.

dplyr_hof | dplyr wrappers for Apache Spark higher order functions

**Description**

These methods implement dplyr grammars for Apache Spark higher order functions.

ensure | Enforce Specific Structure for R Objects

**Description**

These routines are useful when preparing to pass objects to a Spark routine, as it is often necessary to ensure certain parameters are scalar integers, or scalar doubles, and so on.

**Arguments**

object | An R object.
allow.na | Are NA values permitted for this object?
allow.null | Are NULL values permitted for this object?
default | If object is NULL, what value should be used in its place? If default is specified, allow.null is ignored (and assumed to be TRUE).
**find**

**Fill**

**filter**

**Filter**

**Description**

See **filter** for more details.

**find_scalac**

**Discover the Scala Compiler**

**Description**

Find the scalac compiler for a particular version of scala, by scanning some common directories containing scala installations.

**Usage**

```r
find_scalac(version, locations = NULL)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>The scala version to search for. Versions of the form major.minor will be matched against the scalac installation with version major.minor.patch; if multiple compilers are discovered the most recent one will be used.</td>
</tr>
<tr>
<td>locations</td>
<td>Additional locations to scan. By default, the directories /opt/scala and /usr/local/scala will be scanned.</td>
</tr>
</tbody>
</table>
Feature Transformation – Binarizer (Transformer)

Description

Apply thresholding to a column, such that values less than or equal to the threshold are assigned the value 0.0, and values greater than the threshold are assigned the value 1.0. Column output is numeric for compatibility with other modeling functions.

Usage

```r
ft_binarizer(
  x,
  input_col,
  output_col,
  threshold = 0,
  uid = random_string("binarizer_"),
  ...
)
```

Arguments

- `x`: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `input_col`: The name of the input column.
- `output_col`: The name of the output column.
- `threshold`: Threshold used to binarize continuous features.
- `uid`: A character string used to uniquely identify the feature transformer.
- `...`: Optional arguments; currently unused.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`
Examples

```r
## Not run:
library(dplyr)

sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

iris_tbl %>%
  ft_binarizer(
    input_col = "Sepal_Length",
    output_col = "Sepal_Length_bin",
    threshold = 5
  ) %>%
  select(Sepal_Length, Sepal_Length_bin, Species)

## End(Not run)
```

---

### ft_bucketizer

**Feature Transformation – Bucketizer (Transformer)**

**Description**

Similar to R's `cut` function, this transforms a numeric column into a discretized column, with breaks specified through the `splits` parameter.

**Usage**

```r
ft_bucketizer(
  x,
  input_col = NULL,
  output_col = NULL,
  splits = NULL,
  input_cols = NULL,
  output_cols = NULL,
  splits_array = NULL,
  handle_invalid = "error",
  uid = random_string("bucketizer_"),
  ...
)
```

**Arguments**

- `x`: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `input_col`: The name of the input column.
- `output_col`: The name of the output column.
- `splits`: A numeric vector of cutpoints, indicating the bucket boundaries.
input_cols  Names of input columns.
output_cols  Names of output columns.
splits_array Parameter for specifying multiple splits parameters. Each element in this array can be used to map continuous features into buckets.
handle_invalid (Spark 2.1.0+) Param for how to handle invalid entries. Options are 'skip' (filter out rows with invalid values), 'error' (throw an error), or 'keep' (keep invalid values in a special additional bucket). Default: "error"
uid  A character string used to uniquely identify the feature transformer.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

Examples

## Not run:
library(dplyr)

sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

iris_tbl %>%
  ft_bucketizer(
    input_col = "Sepal_Length",
    output_col = "Sepal_Length_bucket",
    splits = c(0, 4.5, 5, 8)
  ) %>%
  select(Sepal_Length, Sepal_Length_bucket, Species)

## End(Not run)
ft_chisq_selector  
Feature Transformation – ChiSqSelector (Estimator)

Description

Chi-Squared feature selection, which selects categorical features to use for predicting a categorical label.

Usage

`ft_chisq_selector(
  x,
  features_col = "features",
  output_col = NULL,
  label_col = "label",
  selector_type = "numTopFeatures",
  fdr = 0.05,
  fpr = 0.05,
  fwe = 0.05,
  num_top_features = 50,
  percentile = 0.1,
  uid = random_string("chisq_selector_"),
  ...
)`

Arguments

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **features_col**: Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.
- **output_col**: The name of the output column.
- **label_col**: Label column name. The column should be a numeric column. Usually this column is output by `ft_r_formula`.
- **selector_type** (Spark 2.1.0+): The selector type of the ChiSqSelector. Supported options: "numTopFeatures" (default), "percentile", "fpr", "fdr", "fwe".
- **fdr** (Spark 2.2.0+): The upper bound of the expected false discovery rate. Only applicable when selector_type = "fdr". Default value is 0.05.
- **fpr** (Spark 2.1.0+): The highest p-value for features to be kept. Only applicable when selector_type = "fpr". Default value is 0.05.
- **fwe** (Spark 2.2.0+): The upper bound of the expected family-wise error rate. Only applicable when selector_type = "fwe". Default value is 0.05.
num_top_features

Number of features that selector will select, ordered by ascending p-value. If the number of features is less than num_top_features, then this will select all features. Only applicable when selector_type = "numTopFeatures". The default value of num_top_features is 50.

percentile

(Spark 2.1.0+) Percentile of features that selector will select, ordered by statistics value descending. Only applicable when selector_type = "percentile". Default value is 0.1.

uid

A character string used to uniquely identify the feature transformer.

Details

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

---

**ft_count_vectorizer**  
*Feature Transformation – CountVectorizer (Estimator)*

Description

Extracts a vocabulary from document collections.
ft_count_vectorizer

Usage

ft_count_vectorizer(
    x,
    input_col = NULL,
    output_col = NULL,
    binary = FALSE,
    min_df = 1,
    min_tf = 1,
    vocab_size = 2^18,
    uid = random_string("count_vectorizer_"),
    ...
)

ml_vocabulary(model)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.
input_col The name of the input column.
output_col The name of the output column.
binary Binary toggle to control the output vector values. If TRUE, all nonzero counts (after min_tf filter applied) are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts. Default: FALSE
min_df Specifies the minimum number of different documents a term must appear in to be included in the vocabulary. If this is an integer greater than or equal to 1, this specifies the number of documents the term must appear in; if this is a double in [0,1), then this specifies the fraction of documents. Default: 1.
min_tf Filter to ignore rare words in a document. For each document, terms with frequency/count less than the given threshold are ignored. If this is an integer greater than or equal to 1, then this specifies a count (of times the term must appear in the document); if this is a double in [0,1), then this specifies a fraction (out of the document’s token count). Default: 1.
vocab_size Build a vocabulary that only considers the top vocab_size terms ordered by term frequency across the corpus. Default: 2^18.
uid A character string used to uniquely identify the feature transformer.
... Optional arguments; currently unused.
model A ml_count_vectorizer_model.

Details

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.
Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

ml_vocabulary() returns a vector of vocabulary built.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh(), ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimater(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

---

**ft_dct**

*Feature Transformation – Discrete Cosine Transform (DCT) (Transformer)*

**Description**

A feature transformer that takes the 1D discrete cosine transform of a real vector. No zero padding is performed on the input vector. It returns a real vector of the same length representing the DCT. The return vector is scaled such that the transform matrix is unitary (aka scaled DCT-II).

**Usage**

```r
ft_dct(
  x,
  input_col = NULL,
  output_col = NULL,
  inverse = FALSE,
  uid = random_string("dct_"),
  ...
)
```

```r
ft_discrete.cosine.transform(
  x,
  input_col,
  output_col,
  inverse = FALSE,
  uid = random_string("dct_"),
  ...
)
```
ft_elementwise_product

Arguments

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **inverse**: Indicates whether to perform the inverse DCT (TRUE) or forward DCT (FALSE).
- **uid**: A character string used to uniquely identify the feature transformer.
- ... Optional arguments; currently unused.

Details

- **ft_discrete_cosine_transform()** is an alias for **ft_dct** for backwards compatibility.

Value

The object returned depends on the class of **x**. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()
Usage

```
ft_elementwise_product(
  x,
  input_col = NULL,
  output_col = NULL,
  scaling_vec = NULL,
  uid = random_string("elementwise_product_"),
  ...
)
```

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **scaling_vec**: The vector to multiply with input vectors.
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh()`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`
Usage

    ft_feature_hasher(
        x,
        input_cols = NULL,
        output_col = NULL,
        num_features = 2^18,
        categorical_cols = NULL,
        uid = random_string("feature_hasher"),
        ...
    )

Arguments

    x                   A spark_connection, ml_pipeline, or a tbl_spark.
    input_cols          Names of input columns.
    output_col          Name of output column.
    num_features        Number of features. Defaults to $2^{18}$.
    categorical_cols   Numeric columns to treat as categorical features. By default only string and
                        boolean columns are treated as categorical, so this param can be used to explicitly
                        specify the numerical columns to treat as categorical.
    uid                 A character string used to uniquely identify the feature transformer.
    ...                 Optional arguments; currently unused.

Details

Feature hashing projects a set of categorical or numerical features into a feature vector of specified
dimension (typically substantially smaller than that of the original feature space). This is done
using the hashing trick https://en.wikipedia.org/wiki/Feature_hashing to map features to
indices in the feature vector.

The FeatureHasher transformer operates on multiple columns. Each column may contain either nu-
meric or categorical features. Behavior and handling of column data types is as follows: -Numeric
columns: For numeric features, the hash value of the column name is used to map the feature value
to its index in the feature vector. By default, numeric features are not treated as categorical (even
when they are integers). To treat them as categorical, specify the relevant columns in categorical-
Cols. -String columns: For categorical features, the hash value of the string "column_name=value"
is used to map to the vector index, with an indicator value of 1.0. Thus, categorical features are
"one-hot" encoded (similarly to using OneHotEncoder with drop_last=FALSE). -Boolean columns:
Boolean values are treated in the same way as string columns. That is, boolean features are repre-
sented as "column_name=true" or "column_name=false", with an indicator value of 1.0.

Null (missing) values are ignored (implicitly zero in the resulting feature vector).

The hash function used here is also the MurmurHash 3 used in HashingTF. Since a simple modulo
on the hashed value is used to determine the vector index, it is advisable to use a power of two as
the num_features parameter; otherwise the features will not be mapped evenly to the vector indices.
Value

The object returned depends on the class of \( x \). If it is a spark_connection, the function returns a \ml_estimator\ or a \ml_estimator\ object. If it is a \ml_pipeline\, it will return a pipeline with the transformer or estimator appended to it. If a \tbl_spark\, it will return a \tbl_spark\ with the transformation applied to it.

See Also

Other feature transformers: \ft_binarizer()\, \ft_bucketizer()\, \ft_chisq_selector()\, \ft_countVectorizer()\, \ft_dct()\, \ft_elementwise_product()\, \ft_hashing_tf()\, \ft_idf()\, \ft_imputer()\, \ft_index_to_string()\, \ft_interaction()\, \ft_lsh()\, \ft_max_abs_scaler()\, \ft_min_max_scaler()\, \ft_ngram()\, \ft_normalizer()\, \ft_one_hot_encoder()\, \ft_one_hot_encoder_estimator()\, \ft_pca()\, \ft_polynomial_expansion()\, \ft_quantile_discretizer()\, \ft_r_formula()\, \ft_regex_tokenizer()\, \ft_robustScaler()\, \ft_sql_transformer()\, \ft_standard_scaler()\, \ft_stopWordsRemover()\, \ft_stringIndexer()\, \ft_tokenizer()\, \ft_vectorAssembler()\, \ft_vectorIndexer()\, \ft_vectorSlicer()\, \ft_word2vec()
**Value**

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

**See Also**

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

---

**ft_idf**

*Feature Transformation – IDF (Estimator)*

**Description**

Compute the Inverse Document Frequency (IDF) given a collection of documents.

**Usage**

```r
ft_idf(
  x,
  input_col = NULL,
  output_col = NULL,
  min_doc_freq = 0,
  uid = random_string("idf_"),
  ...)
```

**Arguments**

- `x`: A spark_connection, ml_pipeline, or a tbl_spark.
- `input_col`: The name of the input column.
- `output_col`: The name of the output column.
- `min_doc_freq`: The minimum number of documents in which a term should appear. Default: 0
- `uid`: A character string used to uniquely identify the feature transformer.
- `...`: Optional arguments; currently unused.

**Details**

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.
Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh(), ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

---

**ft_imputer**

*Feature Transformation – Imputer (Estimator)*

**Description**

Imputation estimator for completing missing values, either using the mean or the median of the columns in which the missing values are located. The input columns should be of numeric type. This function requires Spark 2.2.0+.

**Usage**

```r
ft_imputer(
  x,
  input_cols = NULL,
  output_cols = NULL,
  missing_value = NULL,
  strategy = "mean",
  uid = random_string("imputer_"),
  ...
)
```

**Arguments**

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **input_cols**: The names of the input columns.
- **output_cols**: The names of the output columns.
- **missing_value**: The placeholder for the missing values. All occurrences of missing_value will be imputed. Note that null values are always treated as missing.
strategy  The imputation strategy. Currently only "mean" and "median" are supported. If "mean", then replace missing values using the mean value of the feature. If "median", then replace missing values using the approximate median value of the feature. Default: mean

uid  A character string used to uniquely identify the feature transformer.

Details

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()
uid = random_string("index_to_string_"),
...
)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.
input_col The name of the input column.
output_col The name of the output column.
labels Optional param for array of labels specifying index-string mapping.
uid A character string used to uniquely identify the feature transformer.
... Optional arguments; currently unused.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

ft_string_indexer

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(),
ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(),
ft_imputer(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(),
ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion,
ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(),
ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(),
ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()
ft_lsh

Feature Transformation – LSH (Estimator)

Description
Locality Sensitive Hashing functions for Euclidean distance (Bucketed Random Projection) and Jaccard distance (MinHash).
Usage

```r
ft_bucketed_random_projection_lsh(
  x,
  input_col = NULL,
  output_col = NULL,
  bucket_length = NULL,
  num_hash_tables = 1,
  seed = NULL,
  uid = random_string("bucketed_random_projection_lsh_"),
  ...
)
```

```r
ft_minhash_lsh(
  x,
  input_col = NULL,
  output_col = NULL,
  num_hash_tables = 1L,
  seed = NULL,
  uid = random_string("minhash_lsh_"),
  ...
)
```

Arguments

- `x` A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `input_col` The name of the input column.
- `output_col` The name of the output column.
- `bucket_length` The length of each hash bucket, a larger bucket lowers the false negative rate. The number of buckets will be (max L2 norm of input vectors) / bucketLength.
- `num_hash_tables` Number of hash tables used in LSH OR-amplification. LSH OR-amplification can be used to reduce the false negative rate. Higher values for this param lead to a reduced false negative rate, at the expense of added computational complexity.
- `seed` A random seed. Set this value if you need your results to be reproducible across repeated calls.
- `uid` A character string used to uniquely identify the feature transformer.
- `...` Optional arguments; currently unused.

Details

In the case where `x` is a `tbl_spark`, the estimator fits against `x` to obtain a transformer, returning a `tbl_spark`.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with
the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

ft_lsh_utils

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()
ft_max_abs_scaler

dist_col | Output column for storing the distance between each result row and the key.
dataset_a | One of the datasets to join.
dataset_b | Another dataset to join.
threshold | The threshold for the distance of row pairs.

ft_max_abs_scaler | Feature Transformation – MaxAbsScaler (Estimator)

Description
Rescale each feature individually to range [-1, 1] by dividing through the largest maximum absolute value in each feature. It does not shift centerX the data, and thus does not destroy any sparsity.

Usage
ft_max_abs_scaler(
  x,
  input_col = NULL,
  output_col = NULL,
  uid = random_string("max_abs_scaler_"),
  ...
)

Arguments
x | A spark_connection, ml_pipeline, or a tbl_spark.
input_col | The name of the input column.
output_col | The name of the output column.
uid | A character string used to uniquely identify the feature transformer.
... | Optional arguments; currently unused.

Details
In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

Value
The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimot object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.
**ft_min_max_scaler**  

**Feature Transformation – MinMaxScaler (Estimator)**

**Description**

Rescale each feature individually to a common range $[\text{min}, \text{max}]$ linearly using column summary statistics, which is also known as min-max normalization or Rescaling.

**Usage**

```r
ft_min_max_scaler(
  x,
  input_col = NULL,
  output_col = NULL,
  min = 0,
  max = 1,
  uid = random_string("min_max_scaler_"),
)```

See Also

Other feature transformers:  
- `ft_binarizer()`  
- `ft_bucketizer()`  
- `ft_chisq_selector()`  
- `ft_count_vectorizer()`  
- `ft_dct()`  
- `ft_elementwise_product()`  
- `ft_feature_hasher()`  
- `ft_hashing_tf()`  
- `ft_idf()`  
- `ft_imputer()`  
- `ft_index_to_string()`  
- `ft_interaction()`  
- `ft_lsh()`  
- `ft_min_max_scaler()`  
- `ft_ngram()`  
- `ft_normalizer()`  
- `ft_one_hot_encoder()`  
- `ft_one_hot_encoder_estimator()`  
- `ft_pca()`  
- `ft_polynomial_expansion()`  
- `ft_quantile_discretizer()`  
- `ft_r_formula()`  
- `ft_regex_tokenizer()`  
- `ft_robust_scaler()`  
- `ft_sql_transformer()`  
- `ft_standard_scaler()`  
- `ft_stop_words_remover()`  
- `ft_string_indexer()`  
- `ft_tokenizer()`  
- `ft_vector_assembler()`  
- `ft_vector_indexer()`  
- `ft_vector_slicer()`  
- `ft_word2vec()`

Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

features <- c("Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width")

iris_tbl %>%
  ft_vector_assembler(
    input_col = features,
    output_col = "features_temp"
  ) %>%
  ft_max_abs_scaler(
    input_col = "features_temp",
    output_col = "features"
  )

## End(Not run)
```
Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **min**: Lower bound after transformation, shared by all features. Default: 0.0
- **max**: Upper bound after transformation, shared by all features. Default: 1.0
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

Details

In the case where `x` is a `tbl_spark`, the estimator fits against `x` to obtain a transformer, returning a `tbl_spark`.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns an `ml_estimator` or an `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slice()`, `ft_word2vec()`

Examples

```r
## Not run:
sd <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sdc, iris, name = "iris_tbl", overwrite = TRUE)
features <- c("Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width")
iris_tbl %>%
  ft_vector_assembler(
    input_col = features,
    output_col = "features_temp"
  ) %>%
```
### ft_ngram

#### ft_min_max_scaler(
    input_col = "features_temp",
    output_col = "features"
)

## End(Not run)

---

**ft_ngram** | Feature Transformation – NGram (Transformer)

---

**Description**

A feature transformer that converts the input array of strings into an array of n-grams. Null values in the input array are ignored. It returns an array of n-grams where each n-gram is represented by a space-separated string of words.

**Usage**

```r
ft_ngram(
    x,
    input_col = NULL,
    output_col = NULL,
    n = 2,
    uid = random_string("ngram_"),
    ...
)
```

**Arguments**

- `x`: A spark_connection, ml_pipeline, or a tbl_spark.
- `input_col`: The name of the input column.
- `output_col`: The name of the output column.
- `n`: Minimum n-gram length, greater than or equal to 1. Default: 2, bigram features
- `uid`: A character string used to uniquely identify the feature transformer.
- `...`: Optional arguments; currently unused.

**Details**

When the input is empty, an empty array is returned. When the input array length is less than n (number of elements per n-gram), no n-grams are returned.

**Value**

The object returned depends on the class of `x`. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.
See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`

---

**ft_normalizer**  
*Feature Transformation – Normalizer (Transformer)*

**Description**

Normalize a vector to have unit norm using the given p-norm.

**Usage**

```r
ft_normalizer(
  x,
  input_col = NULL,
  output_col = NULL,
  p = 2,
  uid = random_string("normalizer_"),
  ...
)
```

**Arguments**

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **p**: Normalization in L^p space. Must be >= 1. Defaults to 2.
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

**Value**

The object returned depends on the class of x. If it is a spark_connection, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.
See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tfidf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`

---

**ft_one_hot_encoder**  
*Feature Transformation – OneHotEncoder (Transformer)*

**Description**

One-hot encoding maps a column of label indices to a column of binary vectors, with at most a single one-value. This encoding allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features. Typically, used with `ft_string_indexer()` to index a column first.

**Usage**

```r
ft_one_hot_encoder(
  x,  
  input_cols = NULL,  
  output_cols = NULL,  
  handle_invalid = NULL,  
  drop_last = TRUE,  
  uid = random_string("one_hot_encoder_"),  
  ...  
)
```

**Arguments**

- **x**  
  A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_cols**  
  The name of the input columns.
- **output_cols**  
  The name of the output columns.
- **handle_invalid**  
  (Spark 2.1.0+) Param for how to handle invalid entries. Options are 'skip' (filter out rows with invalid values), 'error' (throw an error), or 'keep' (keep invalid values in a special additional bucket). Default: "error"
- **drop_last**  
  Whether to drop the last category. Defaults to TRUE.
- **uid**  
  A character string used to uniquely identify the feature transformer.
- **...**  
  Optional arguments; currently unused.
Value

The object returned depends on the class of \( x \). If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ftidf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quintile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

Usage

```r
ft_one_hot_encoder_estimator(
  x,
  input_cols = NULL,
  output_cols = NULL,
  handle_invalid = "error",
  drop_last = TRUE,
  uid = random_string("one_hot_encoder_estimator_"),
  ...
)
```

Arguments

- \( x \) A spark_connection, ml_pipeline, or a tbl_spark.
- input_cols Names of input columns.
- output_cols Names of output columns.
handle_invalid  (Spark 2.1.0+) Param for how to handle invalid entries. Options are 'skip' (filter out rows with invalid values), 'error' (throw an error), or 'keep' (keep invalid values in a special additional bucket). Default: "error"

drop_last  Whether to drop the last category. Defaults to TRUE.

uid  A character string used to uniquely identify the feature transformer.

...  Optional arguments; currently unused.

Details

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

Description

PCA trains a model to project vectors to a lower dimensional space of the top k principal components.

Usage

```r
ft_pca(  
  x,  
  input_col = NULL,  
  output_col = NULL,  
  k = NULL,  
  uid = random_string("pca_"),  
  ...  
)
```

```r
ml_pca(x, features = tbl_vars(x), k = length(features), pc_prefix = "PC", ...)```
Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **k**: The number of principal components.
- **uid**: A character string used to uniquely identify the feature transformer.
- **features**: The columns to use in the principal components analysis. Defaults to all columns in `x`.
- **pc_prefix**: Length-one character vector used to prepend names of components.

Details

In the case where `x` is a `tbl_spark`, the estimator fits against `x` to obtain a transformer, returning a `tbl_spark`.

`ml_pca()` is a wrapper around `ft_pca()` that returns a `ml_model`.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`

Examples

```r
# Not run:
library(dplyr)
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)
iris_tbl %>%
  select(-Species) %>%
  ml_pca(k = 2)
```
ft_polynomial_expansion

Feature Transformation – PolynomialExpansion (Transformer)

Description

Perform feature expansion in a polynomial space. E.g. take a 2-variable feature vector as an example: (x, y), if we want to expand it with degree 2, then we get (x, x * x, y, x * y, y * y).

Usage

ft_polynomial_expansion(
  x,
  input_col = NULL,
  output_col = NULL,
  degree = 2,
  uid = random_string("polynomial_expansion_"),
  ...
)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.
input_col The name of the input column.
output_col The name of the output column.
degree The polynomial degree to expand, which should be greater than equal to 1. A value of 1 means no expansion. Default: 2
uid A character string used to uniquely identify the feature transformer.
... Optional arguments; currently unused.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.
ft_quantile_discretizer

Feature Transformation – QuantileDiscretizer (Estimator)

Description

ft_quantile_discretizer takes a column with continuous features and outputs a column with binned categorical features. The number of bins can be set using the num_buckets parameter. It is possible that the number of buckets used will be smaller than this value, for example, if there are too few distinct values of the input to create enough distinct quantiles.

Usage

```r
ft_quantile_discretizer(
    x,
    input_col = NULL,
    output_col = NULL,
    num_buckets = 2,
    input_cols = NULL,
    output_cols = NULL,
    num_buckets_array = NULL,
    handle_invalid = "error",
    relative_error = 0.001,
    uid = random_string("quantile_discretizer_"),
    weight_column = NULL,
    ...
)
```

Arguments

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **num_buckets**: Number of buckets (quantiles, or categories) into which data points are grouped. Must be greater than or equal to 2.
- **input_cols**: Names of input columns.
output_cols  Names of output columns.
num_buckets_array  Array of number of buckets (quantiles, or categories) into which data points are grouped. Each value must be greater than or equal to 2.
handle_invalid  (Spark 2.1.0+) Param for how to handle invalid entries. Options are 'skip' (filter out rows with invalid values), 'error' (throw an error), or 'keep' (keep invalid values in a special additional bucket). Default: "error"
relative_error  (Spark 2.0.0+) Relative error (see documentation for org.apache.spark.sql.DataFrameStatFunctions.approxQuantile here for description). Must be in the range [0, 1]. default: 0.001
uid  A character string used to uniquely identify the feature transformer.
weight_column  If not NULL, then a generalized version of the Greenwald-Khanna algorithm will be run to compute weighted percentiles, with each input having a relative weight specified by the corresponding value in 'weight_column'. The weights can be considered as relative frequencies of sample inputs.
...  Optional arguments; currently unused.

Details

NaN handling: null and NaN values will be ignored from the column during QuantileDiscretizer fitting. This will produce a Bucketizer model for making predictions. During the transformation, Bucketizer will raise an error when it finds NaN values in the dataset, but the user can also choose to either keep or remove NaN values within the dataset by setting handle_invalid If the user chooses to keep NaN values, they will be handled specially and placed into their own bucket, for example, if 4 buckets are used, then non-NaN data will be put into buckets[0-3], but NaNs will be counted in a special bucket[4].

Algorithm: The bin ranges are chosen using an approximate algorithm (see the documentation for org.apache.spark.sql.DataFrameStatFunctions.approxQuantile here for a detailed description). The precision of the approximation can be controlled with the relative_error parameter. The lower and upper bin bounds will be -Infinity and +Infinity, covering all real values.

Note that the result may be different every time you run it, since the sample strategy behind it is non-deterministic.

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

ft.bucketizer

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf()
**ft_regex_tokenizer**

A regex based tokenizer that extracts tokens either by using the provided regex pattern to split the text (default) or repeatedly matching the regex (if gaps is false). Optional parameters also allow filtering tokens using a minimal length. It returns an array of strings that can be empty.

**Usage**

```r
ft_regex_tokenizer(
  x,
  input_col = NULL,
  output_col = NULL,
  gaps = TRUE,
  min_token_length = 1,
  pattern = "\s+",
  to_lower_case = TRUE,
  uid = random_string("regex_tokenizer_"),
  ...
)
```

**Arguments**

- `x` A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `input_col` The name of the input column.
- `output_col` The name of the output column.
- `gaps` Indicates whether regex splits on gaps (TRUE) or matches tokens (FALSE).
- `min_token_length` Minimum token length, greater than or equal to 0.
- `pattern` The regular expression pattern to be used.
- `to_lower_case` Indicates whether to convert all characters to lowercase before tokenizing.
- `uid` A character string used to uniquely identify the feature transformer.
- `...` Optional arguments; currently unused.
ft_robust_scaler

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_countVectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()

---

ft_robust_scaler Feature Transformation – RobustScaler (Estimator)

Description

RobustScaler removes the median and scales the data according to the quantile range. The quantile range is by default IQR (Interquartile Range, quantile range between the 1st quartile = 25th quantile and the 3rd quartile = 75th quantile) but can be configured. Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Median and quantile range are then stored to be used on later data using the transform method. Note that missing values are ignored in the computation of medians and ranges.

Usage

```r
ft_robust_scaler(
  x,
  input_col = NULL,
  output_col = NULL,
  lower = 0.25,
  upper = 0.75,
  with_centering = TRUE,
  with_scaling = TRUE,
  relative_error = 0.001,
  uid = random_string("ft_robust_scaler_"),
  ...
)
```
Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **lower**: Lower quantile to calculate quantile range.
- **upper**: Upper quantile to calculate quantile range.
- **with_centering**: Whether to center data with median.
- **with_scaling**: Whether to scale the data to quantile range.
- **relative_error**: The target relative error for quantile computation.
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

Details

In the case where `x` is a `tbl_spark`, the estimator fits against `x` to obtain a transformer, returning a `tbl_spark`.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`
Usage

```r
default_ft_r_formula(
  x,  
  formula = NULL, 
  features_col = "features", 
  label_col = "label", 
  force_index_label = FALSE, 
  uid = random_string("r_formula_"), 
  ... 
)
```

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **formula**: R formula as a character string or a formula. Formula objects are converted to character strings directly and the environment is not captured.
- **features_col**: Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.
- **label_col**: Label column name. The column should be a numeric column. Usually this column is output by `ft_r_formula`.
- **force_index_label**: (Spark 2.1.0+) Force to index label whether it is numeric or string type. Usually we index label only when it is string type. If the formula was used by classification algorithms, we can force to index label even it is numeric type by setting this param with true. Default: FALSE.
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

Details

The basic operators in the formula are:

- `~` separate target and terms
- `+` concat terms, "+ 0" means removing intercept
- `-` remove a term, "- 1" means removing intercept
- `:` interaction (multiplication for numeric values, or binarized categorical values)
- `.` all columns except target

Suppose `a` and `b` are double columns, we use the following simple examples to illustrate the effect of RFormula:

- `y ~ a + b` means model `y ~ w0 + w1 * a + w2 * b` where `w0` is the intercept and `w1`, `w2` are coefficients.
- `y ~ a + b + a:b - 1` means model `y ~ w1 * a + w2 * b + w3 * a * b` where `w1`, `w2`, `w3` are coefficients.
RFormula produces a vector column of features and a double or string column of label. Like when formulas are used in R for linear regression, string input columns will be one-hot encoded, and numeric columns will be cast to doubles. If the label column is of type string, it will be first transformed to double with StringIndexer. If the label column does not exist in the DataFrame, the output label column will be created from the specified response variable in the formula.

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

**Value**

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

**See Also**

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()
Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.

statement A SQL statement.

uid A character string used to uniquely identify the feature transformer.

... Optional arguments; currently unused.

tbl A tbl_spark generated using dplyr transformations.

Details

`ft_dplyr_transformer()` is mostly a wrapper around `ft_sql_transformer()` that takes a tbl_spark instead of a SQL statement. Internally, the `ft_dplyr_transformer()` extracts the dplyr transformations used to generate tbl as a SQL statement or a sampling operation. Note that only single-table dplyr verbs are supported and that the sdf_ family of functions are not.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`
Usage

```r
ft_standard_scaler(
  x,
  input_col = NULL,
  output_col = NULL,
  with_mean = FALSE,
  with_std = TRUE,
  uid = random_string("standard_scaler_"),
  ...
)
```

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **with_mean**: Whether to center the data with mean before scaling. It will build a dense output, so take care when applying to sparse input. Default: FALSE
- **with_std**: Whether to scale the data to unit standard deviation. Default: TRUE
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

Details

In the case where `x` is a `tbl_spark`, the estimator fits against `x` to obtain a transformer, returning a `tbl_spark`.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`
Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

features <- c("Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width")

iris_tbl %>%
  ft_vector_assembler(
    input_col = features,
    output_col = "features_temp"
  ) %>%
  ft_standard_scaler(
    input_col = "features_temp",
    output_col = "features",
    with_mean = TRUE
  )

## End(Not run)
```

---

**ft_stop_words_remover**  
**Feature Transformation – StopWordsRemover (Transformer)**

**Description**

A feature transformer that filters out stop words from input.

**Usage**

```r
ft_stop_words_remover(
  x,  
  input_col = NULL,  
  output_col = NULL,  
  case_sensitive = FALSE,  
  stop_words = ml_default_stop_words(spark_connection(x), "english"),  
  uid = random_string("stop_words_remover_"),  
  ...  
)
```

**Arguments**

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **case_sensitive**: Whether to do a case sensitive comparison over the stop words.
- **stop_words**: The words to be filtered out.
uid

A character string used to uniquely identify the feature transformer.

...

Optional arguments; currently unused.

Value

The object returned depends on the class of \(x\). If it is a spark_connection, the function returns a ml_estimator or a ml_pipeline object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

See Also

ml_default_stop_words

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer(), ft_word2vec()
input_col = NULL,
output_col = NULL,
labels,
handle_invalid = "error",
uid = random_string("string_indexer_model_"),
...
)

Arguments

x: A spark_connection, ml_pipeline, or a tbl_spark.
input_col: The name of the input column.
output_col: The name of the output column.
handle_invalid: (Spark 2.1.0+) Param for how to handle invalid entries. Options are 'skip' (filter out rows with invalid values), 'error' (throw an error), or 'keep' (keep invalid values in a special additional bucket). Default: "error"
string_order_type: (Spark 2.3+) How to order labels of string column. The first label after ordering is assigned an index of 0. Options are "frequencyDesc", "frequencyAsc", "alphabetDesc", and "alphabetAsc". Defaults to "frequencyDesc".
uid: A character string used to uniquely identify the feature transformer.
...: Optional arguments; currently unused.
model: A fitted StringIndexer model returned by ft_string_indexer()
labels: Vector of labels, corresponding to indices to be assigned.

Details

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

ml_labels() returns a vector of labels, corresponding to indices to be assigned.

See Also

ft_index_to_string

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft Polynomial expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_sort_auc(), ft_sort_index(), ft_sort_key(), ft_sort_values(), ft_svd()
ft_tokenizer

Feature Transformation – Tokenizer (Transformer)

Description

A tokenizer that converts the input string to lowercase and then splits it by white spaces.

Usage

```r
ft_tokenizer(
  x,
  input_col = NULL,
  output_col = NULL,
  uid = random_string("tokenizer_"),
  ...)
```

Arguments

- `x` A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `input_col` The name of the input column.
- `output_col` The name of the output column.
- `uid` A character string used to uniquely identify the feature transformer.
- `...` Optional arguments; currently unused.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_vector_assembler()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`
ft_vector_assembler

Feature Transformation – VectorAssembler (Transformer)

Description

Combine multiple vectors into a single row-vector; that is, where each row element of the newly
generated column is a vector formed by concatenating each row element from the specified input
columns.

Usage

```r
ft_vector_assembler(
  x,
  input_cols = NULL,
  output_col = NULL,
  uid = random_string("vector_assembler_"),
  ...
)
```

Arguments

- `x`: A spark_connection, ml_pipeline, or a tbl_spark.
- `input_cols`: The names of the input columns
- `output_col`: The name of the output column.
- `uid`: A character string used to uniquely identify the feature transformer.
- `...`: Optional arguments; currently unused.

Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns
a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with
the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the
transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`,
`ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`,
`ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`,
`ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`,
`ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`,
`ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`,
`ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_indexer()`, `ft_vector_slicer()`, `ft_word2vec()`
**ft_vector_indexer**  
*Feature Transformation – VectorIndexer (Estimator)*

**Description**
Indexing categorical feature columns in a dataset of Vector.

**Usage**
```r
ft_vector_indexer(
  x,
  input_col = NULL,
  output_col = NULL,
  handle_invalid = "error",
  max_categories = 20,
  uid = random_string("vector_indexer_"),
  ...
)
```

**Arguments**
- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **handle_invalid**: (Spark 2.1.0+) Param for how to handle invalid entries. Options are 'skip' (filter out rows with invalid values), 'error' (throw an error), or 'keep' (keep invalid values in a special additional bucket). Default: "error"
- **max_categories**: Threshold for the number of values a categorical feature can take. If a feature is found to have > max_categories values, then it is declared continuous. Must be greater than or equal to 2. Defaults to 20.
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

**Details**
In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

**Value**
The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.
**ft_vector_slicer**

*Feature Transformation – VectorSlicer (Transformer)*

**Description**

Takes a feature vector and outputs a new feature vector with a subarray of the original features.

**Usage**

```r
ft_vector_slicer(
  x,
  input_col = NULL,
  output_col = NULL,
  indices = NULL,
  uid = random_string("vector_slicer_"),
  ...)
```

**Arguments**

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **input_col**: The name of the input column.
- **output_col**: The name of the output column.
- **indices**: An vector of indices to select features from a vector column. Note that the indices are 0-based.
- **uid**: A character string used to uniquely identify the feature transformer.
- **...**: Optional arguments; currently unused.

**Value**

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` or a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the transformer or estimator appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the transformation applied to it.

See Also

Other feature transformers: `ft_binarizer()`, `ft_bucketizer()`, `ft_chisq_selector()`, `ft_count_vectorizer()`, `ft_dct()`, `ft_elementwise_product()`, `ft_feature_hasher()`, `ft_hashing_tf()`, `ft_idf()`, `ft_imputer()`, `ft_index_to_string()`, `ft_interaction()`, `ft_lsh`, `ft_max_abs_scaler()`, `ft_min_max_scaler()`, `ft_ngram()`, `ft_normalizer()`, `ft_one_hot_encoder()`, `ft_one_hot_encoder_estimator()`, `ft_pca()`, `ft_polynomial_expansion()`, `ft_quantile_discretizer()`, `ft_r_formula()`, `ft_regex_tokenizer()`, `ft_robust_scaler()`, `ft_sql_transformer()`, `ft_standard_scaler()`, `ft_stop_words_remover()`, `ft_string_indexer()`, `ft_tokenizer()`, `ft_vector_assembler()`, `ft_vector_slicer()`, `ft_word2vec()`
See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vectorAssembler(), ft_vector_indexer(), ft_word2vec()

---

**ft_word2vec**

**Feature Transformation – Word2Vec (Estimator)**

**Description**

Word2Vec transforms a word into a code for further natural language processing or machine learning process.

**Usage**

```r
ft_word2vec(
  x,
  input_col = NULL,
  output_col = NULL,
  vector_size = 100,
  min_count = 5,
  max_sentence_length = 1000,
  num_partitions = 1,
  step_size = 0.025,
  max_iter = 1,
  seed = NULL,
  uid = random_string("word2vec_"),
  ...
)
```

```
ml_find_synonyms(model, word, num)
```

**Arguments**

- `x` A spark_connection, ml_pipeline, or a tbl_spark.
- `input_col` The name of the input column.
- `output_col` The name of the output column.
- `vector_size` The dimension of the code that you want to transform from words. Default: 100
- `min_count` The minimum number of times a token must appear to be included in the word2vec model’s vocabulary. Default: 5
full_join

max_sentence_length

(Spark 2.0.0+) Sets the maximum length (in words) of each sentence in the input data. Any sentence longer than this threshold will be divided into chunks of up to max_sentence_length size. Default: 1000

num_partitions

Number of partitions for sentences of words. Default: 1

step_size

Param for Step size to be used for each iteration of optimization (> 0).

max_iter

The maximum number of iterations to use.

seed

A random seed. Set this value if you need your results to be reproducible across repeated calls.

uid

A character string used to uniquely identify the feature transformer.

... Optional arguments; currently unused.

model

A fitted Word2Vec model, returned by ft_word2vec().

word

A word, as a length-one character vector.

num

Number of words closest in similarity to the given word to find.

Details

In the case where x is a tbl_spark, the estimator fits against x to obtain a transformer, returning a tbl_spark.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator or a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the transformer or estimator appended to it. If a tbl_spark, it will return a tbl_spark with the transformation applied to it.

ml_find_synonyms() returns a DataFrame of synonyms and cosine similarities

See Also

Other feature transformers: ft_binarizer(), ft_bucketizer(), ft_chisq_selector(), ft_count_vectorizer(), ft_dct(), ft_elementwise_product(), ft_feature_hasher(), ft_hashing_tf(), ft_idf(), ft_imputer(), ft_index_to_string(), ft_interaction(), ft_lsh, ft_max_abs_scaler(), ft_min_max_scaler(), ft_ngram(), ft_normalizer(), ft_one_hot_encoder(), ft_one_hot_encoder_estimator(), ft_pca(), ft_polynomial_expansion(), ft_quantile_discretizer(), ft_r_formula(), ft_regex_tokenizer(), ft_robust_scaler(), ft_sql_transformer(), ft_standard_scaler(), ft_stop_words_remover(), ft_string_indexer(), ft_tokenizer(), ft_vector_assembler(), ft_vector_indexer(), ft_vector_slicer()
get_spark_sql_catalog_implementation

---

**generic_call_interface**

*Generic Call Interface*

---

**Description**

Generic Call Interface

**Arguments**

- `sc` : spark_connection
- `static` : Is this a static method call (including a constructor). If so then the `object` parameter should be the name of a class (otherwise it should be a `spark_jobj` instance).
- `object` : Object instance or name of class (for static method)
- `method` : Name of method
- `...` : Call parameters

---

**get_spark_sql_catalog_implementation**

*Retrieve the Spark connection’s SQL catalog implementation property*

---

**Description**

Retrieve the Spark connection’s SQL catalog implementation property

**Usage**

`get_spark_sql_catalog_implementation(sc)`

**Arguments**

- `sc` : spark_connection

**Value**

spark.sql.catalogImplementation property from the connection’s runtime configuration
hive_context_config  

Runtime configuration interface for Hive

Description

Retrieves the runtime configuration interface for Hive.

Usage

```r
hive_context_config(sc)
```

Arguments

- `sc`  
  A `spark_connection`.

hof_aggregate  

Apply Aggregate Function to Array Column

Description

Apply an element-wise aggregation function to an array column (this is essentially a `dplyr` wrapper for the `aggregate(array<T>, A, function<A, T, A>[, function<A, R>]): R built-in Spark SQL functions)

Usage

```r
hof_aggregate(
  x,
  start,
  merge,
  finish = NULL,
  expr = NULL,
  dest_col = NULL,
  ...
)
```

Arguments

- `x`  
  The Spark data frame to run aggregation on
- `start`  
  The starting value of the aggregation
- `merge`  
  The aggregation function
- `finish`  
  Optional param specifying a transformation to apply on the final value of the aggregation
- `expr`  
  The array being aggregated, could be any SQL expression evaluating to an array (default: the last column of the Spark data frame)
- `dest_col`  
  Column to store the aggregated result (default: expr)
- `...`  
  Additional params to `dplyr::mutate`
hof_array_sort

Sorts array using a custom comparator

Description

Applies a custom comparator function to sort an array (this is essentially a dplyr wrapper to the 'array_sort(expr, func)' higher-order function, which is supported since Spark 3.0)

Usage

hof_array_sort(x, func, expr = NULL, dest_col = NULL, ...)

Arguments

x The Spark data frame to be processed
func The comparator function to apply (it should take 2 array elements as arguments and return an integer, with a return value of -1 indicating the first element is less than the second, 0 indicating equality, or 1 indicating the first element is greater than the second)
expr The array being sorted, could be any SQL expression evaluating to an array (default: the last column of the Spark data frame)
dest_col Column to store the sorted result (default: expr)
... Additional params to dplyr::mutate

Examples

## Not run:

library(sparklyr)
sc <- spark_connect(master = "local")
# concatenates all numbers of each array in `array_column` and add parentheses around the resulting string
copy_to(sc, dplyr::tibble(array_column = list(1:5, 21:25))) %>%
hof_aggregate(
  start = "",
  merge = ~ CONCAT(.y, .x),
  finish = ~ CONCAT("(", .x, ")")
)

## End(Not run)
hof_exists

Determine Whether Some Element Exists in an Array Column

Description

Determines whether an element satisfying the given predicate exists in each array from an array column (this is essentially a dplyr wrapper for the exists(array<T>, function<T, Boolean>): Boolean built-in Spark SQL function)

Usage

hof_exists(x, pred, expr = NULL, dest_col = NULL, ...)

Arguments

- **x**: The Spark data frame to search
- **pred**: A boolean predicate
- **expr**: The array being searched (could be any SQL expression evaluating to an array)
- **dest_col**: Column to store the search result
- **...**: Additional params to dplyr::mutate

hof_filter

Filter Array Column

Description

Apply an element-wise filtering function to an array column (this is essentially a dplyr wrapper for the filter(array<T>, function<T, Boolean>): array<T> built-in Spark SQL functions)

Usage

hof_filter(x, func, expr = NULL, dest_col = NULL, ...)
Arguments

x The Spark data frame to filter
func The filtering function
expr The array being filtered, could be any SQL expression evaluating to an array (default: the last column of the Spark data frame)
dest_col Column to store the filtered result (default: expr)
... Additional params to dplyr::mutate

Examples

## Not run:
library(sparklyr)
sc <- spark_connect(master = "local")
# only keep odd elements in each array in `array_column`
copy_to(sc, dplyr::tibble(array_column = list(1:5, 21:25))) %>%
  hof_filter(~ .x %% 2 == 1)

## End(Not run)

---

**hof_forall** Checks whether all elements in an array satisfy a predicate

Description

Checks whether the predicate specified holds for all elements in an array (this is essentially a dplyr wrapper to the ‘forall(expr, pred)’ higher-order function, which is supported since Spark 3.0)

Usage

hof_forall(x, pred, expr = NULL, dest_col = NULL, ...)

Arguments

x The Spark data frame to be processed
pred The predicate to test (it should take an array element as argument and return a boolean value)
expr The array being tested, could be any SQL expression evaluating to an array (default: the last column of the Spark data frame)
dest_col Column to store the boolean result (default: expr)
... Additional params to dplyr::mutate
**Examples**

```r
## Not run:
sc <- spark_connect(master = "local", version = "3.0.0")
df <- dplyr::tibble(  
  x = list(c(1, 2, 3, 4, 5), c(6, 7, 8, 9, 10)),  
  y = list(c(1, 4, 2, 8, 5), c(7, 1, 4, 2, 8)),  
)
sdf <- sdf_copy_to(sc, df, overwrite = TRUE)

all_positive_tbl <- sdf %>%  
  hof_forall(pred = ~ .x > 0, expr = y, dest_col = all_positive) %>%  
  dplyr::select(all_positive)

## End(Not run)
```

---

**hof_map_filter**

Filters a map

**Description**

Filters entries in a map using the function specified (this is essentially a dplyr wrapper to the `map_filter(expr, func)` higher-order function, which is supported since Spark 3.0)

**Usage**

```r
hof_map_filter(x, func, expr = NULL, dest_col = NULL, ...)
```

**Arguments**

- **x**: The Spark data frame to be processed
- **func**: The filter function to apply (it should take (key, value) as arguments and return a boolean value, with FALSE indicating the key-value pair should be discarded and TRUE otherwise)
- **expr**: The map being filtered, could be any SQL expression evaluating to a map (default: the last column of the Spark data frame)
- **dest_col**: Column to store the filtered result (default: expr)
- **...**: Additional params to dplyr::mutate

**Examples**

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local", version = "3.0.0")
sdf <- sdf_len(sc, 1) %>% dplyr::mutate(m = map(1, 0, 2, 2, 3, -1))
```
filtered_sdf <- sdf %>% hof_map_filter(~ .x > .y)

### End(Not run)

---

**Merges two maps into one**

### Description

Merges two maps into a single map by applying the function specified to pairs of values with the same key (this is essentially a dplyr wrapper to the `map_zip_with(map1, map2, func)` higher-order function, which is supported since Spark 3.0)

### Usage

`hof_map_zip_with(x, func, dest_col = NULL, map1 = NULL, map2 = NULL, ...)`

### Arguments

- **x**: The Spark data frame to be processed
- **func**: The function to apply (it should take (key, value1, value2) as arguments, where (key, value1) is a key-value pair present in map1, (key, value2) is a key-value pair present in map2, and return a transformed value associated with key in the resulting map
- **dest_col**: Column to store the query result (default: the last column of the Spark data frame)
- **map1**: The first map being merged, could be any SQL expression evaluating to a map (default: the first column of the Spark data frame)
- **map2**: The second map being merged, could be any SQL expression evaluating to a map (default: the second column of the Spark data frame)
- **...**: Additional params to dplyr::mutate

### Examples

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local", version = "3.0.0")

# create a Spark dataframe with 2 columns of type MAP<STRING, INT>
two_maps_tbl <- sdf_copy_to(
  sc,
  dplyr::tibble(
    m1 = c("{1\":2,\"3\":4,\"5\":6}\", "{2\":1,\"4\":3,\"6\":5}\"),
    m2 = c("{1\":1,\"3\":3,\"5\":5}\", "{2\":2,\"4\":4,\"6\":6}\")
  ),
)`
overwrite = TRUE
)

# create a 3rd column containing MAP<STRING, INT> values derived from the
# first 2 columns

transformed_two_maps_tbl <- two_maps_tbl %>%
  hof_map_zip_with(
    func = .(k, v1, v2) %>% (CONCAT(k, "\_", v1, "\_", v2)),
    dest_col = m3
  )

## End(Not run)

---

**hof_transform**  
*Transform Array Column*

**Description**

Apply an element-wise transformation function to an array column (this is essentially a dplyr wrapper for the `transform(array<T>, function<T, U>): array<U>` and the `transform(array<T>, function<T, Int, U>): array<U>` built-in Spark SQL functions)

**Usage**

`hof_transform(x, func, expr = NULL, dest_col = NULL, ...)`

**Arguments**

- `x`: The Spark data frame to transform
- `func`: The transformation to apply
- `expr`: The array being transformed, could be any SQL expression evaluating to an array (default: the last column of the Spark data frame)
- `dest_col`: Column to store the transformed result (default: `expr`)
- `...`: Additional params to `dplyr::mutate`

**Examples**

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local")
# applies the (x -> x * x) transformation to elements of all arrays
copy_to(sc, dplyr::tibble(arr = list(1:5, 21:25))) %>%
  hof_transform(~ .x * .x)
```
**hof_transform_keys**   
*Transforms keys of a map*

**Description**
Applies the transformation function specified to all keys of a map (this is essentially a dplyr wrapper to the `transform_keys(expr, func)` higher-order function, which is supported since Spark 3.0)

**Usage**

```r
hof_transform_keys(x, func, expr = NULL, dest_col = NULL, ...)
```

**Arguments**
- **x**: The Spark data frame to be processed
- **func**: The transformation function to apply (it should take (key, value) as arguments and return a transformed key)
- **expr**: The map being transformed, could be any SQL expression evaluating to a map (default: the last column of the Spark data frame)
- **dest_col**: Column to store the transformed result (default: expr)
- **...**: Additional params to dplyr::mutate

**Examples**

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local", version = "3.0.0")
sdf <- sdf_len(sc) %>% dplyr::mutate(m = map("a", 0L, "b", 2L, "c", -1L))
transformed_sdf <- sdf %>% hof_transform_keys(~ CONCAT(.x, " == ", .y))
## End(Not run)
```
**hof_transform_values**  
*Transforms values of a map*

**Description**
Applies the transformation function specified to all values of a map (this is essentially a dplyr wrapper to the ‘transform_values(expr, func)’ higher-order function, which is supported since Spark 3.0)

**Usage**
hof_transform_values(x, func, expr = NULL, dest_col = NULL, ...)

**Arguments**
- `x`: The Spark data frame to be processed
- `func`: The transformation function to apply (it should take (key, value) as arguments and return a transformed value)
- `expr`: The map being transformed, could be any SQL expression evaluating to a map (default: the last column of the Spark data frame)
- `dest_col`: Column to store the transformed result (default: expr)
- `...`: Additional params to dplyr::mutate

**Examples**
```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local", version = "3.0.0")
sdf <- sdf_len(sc, 1) %>% dplyr::mutate(m = map("a", 0L, "b", 2L, "c", -1L))
transformed_sdf <- sdf %>% hof_transform_values(~ CONCAT(.x, " == ", .y))
## End(Not run)
```

**hof_zip_with**  
*Combines 2 Array Columns*

**Description**
Applies an element-wise function to combine elements from 2 array columns (this is essentially a dplyr wrapper for the zip_with(array<T>, array<U>, function<T, U, R>): array<R> built-in function in Spark SQL)
Usage

```r
hof_zip_with(x, func, dest_col = NULL, left = NULL, right = NULL, ...)
```

Arguments

- **x**: The Spark data frame to process
- **func**: Element-wise combining function to be applied
- **dest_col**: Column to store the query result (default: the last column of the Spark data frame)
- **left**: Any expression evaluating to an array (default: the first column of the Spark data frame)
- **right**: Any expression evaluating to an array (default: the second column of the Spark data frame)
- **...**: Additional params to `dplyr::mutate`

Examples

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local")
# compute element-wise products of 2 arrays from each row of `left` and `right`
# and store the resulting array in `res`
sc
copy_to(
  sc,
dplyr::tibble(
    left = list(1:5, 21:25),
    right = list(6:10, 16:20),
    res = c(0, 0)
  )
)
## End(Not run)
```

**inner_join**

### Inner join

Description

See `inner_join` for more details.
**invoke**

*Invoke a Method on a JVM Object*

**Description**
Invoke methods on Java object references. These functions provide a mechanism for invoking various Java object methods directly from R.

**Usage**

```r
invoke(jobj, method, ...)

invoke_static(sc, class, method, ...)

invoke_new(sc, class, ...)
```

**Arguments**

- **jobj**: An R object acting as a Java object reference (typically, a `spark_jobj`).
- **method**: The name of the method to be invoked.
- **...**: Optional arguments, currently unused.
- **sc**: A `spark_connection`.
- **class**: The name of the Java class whose methods should be invoked.

**Details**
Use each of these functions in the following scenarios:

- **invoke**: Execute a method on a Java object reference (typically, a `spark_jobj`).
- **invoke_static**: Execute a static method associated with a Java class.
- **invoke_new**: Invoke a constructor associated with a Java class.

**Examples**

```r
sc <- spark_connect(master = "spark://HOST:PORT")
spark_context(sc) %>%
  invoke("textFile", "file.csv", 1L) %>%
  invoke("count")
```

---

**jarray**

*Instantiate a Java array with a specific element type.*
**Description**

Given a list of Java object references, instantiate an `Array[T]` containing the same list of references, where `T` is a non-primitive type that is more specific than `java.lang.Object`.

**Usage**

`jarray(sc, x, element_type)`

**Arguments**

- `sc`: A `spark_connection`.
- `x`: A list of Java object references.
- `element_type`: A valid Java class name representing the generic type parameter of the Java array to be instantiated. Each element of `x` must refer to a Java object that is assignable to `element_type`.

**Examples**

```r
sc <- spark_connect(master = "spark://HOST:PORT")

string_arr <- jarray(sc, letters, element_type = "java.lang.String")
# string_arr is now a reference to an array of type String[]
```

---

**jfloat**

*Instantiate a Java float type.*

**Description**

Instantiate a `java.lang.Float` object with the value specified. NOTE: this method is useful when one has to invoke a Java/Scala method requiring a float (instead of double) type for at least one of its parameters.

**Usage**

`jfloat(sc, x)`

**Arguments**

- `sc`: A `spark_connection`.
- `x`: A numeric value in R.

**Examples**

```r
sc <- spark_connect(master = "spark://HOST:PORT")

jflt <- jfloat(sc, 1.23e-8)
# jflt is now a reference to a java.lang.Float object
```
### jfloat_array

**Instantiante an Array[Float].**

**Description**

Instantiate an Array[Float] object with the value specified. NOTE: this method is useful when one has to invoke a Java/Scala method requiring an Array[Float] as one of its parameters.

**Usage**

`jfloat_array(sc, x)`

**Arguments**

- `sc` A spark_connection.
- `x` A numeric vector in R.

**Examples**

```r
sc <- spark_connect(master = "spark://HOST:PORT")
jflt_arr <- jfloat_array(sc, c(-1.23e-8, 0, -1.23e-8))
# jflt_arr is now a reference an array of java.lang.Float
```

---

### join.tbl.spark

**Join Spark tbls.**

**Description**

These functions are wrappers around their 'dplyr' equivalents that set Spark SQL-compliant values for the 'suffix' argument by replacing dots ('.') with underscores ('_'). See [join] for a description of the general purpose of the functions.

**Usage**

```r
## S3 method for class 'tbl.spark'
inner_join(
x,
y,
by = NULL,
copy = FALSE,
suffix = c("_x", "_y"),
auto_index = FALSE,
..., sql_on = NULL
```

---
## S3 method for class 'tbl_spark'
left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c("_x", "_y"),
  auto_index = FALSE,
  ...,
  sql_on = NULL
)

## S3 method for class 'tbl_spark'
right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c("_x", "_y"),
  auto_index = FALSE,
  ...,
  sql_on = NULL
)

## S3 method for class 'tbl_spark'
full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c("_x", "_y"),
  auto_index = FALSE,
  ...,
  sql_on = NULL
)

### Arguments

- **x, y**  
  A pair of lazy data frames backed by database queries.

- **by**  
  A join specification created with \texttt{join_by()}, or a character vector of variables to join by.

  If \texttt{NULL}, the default, \texttt{*_join()} will perform a natural join, using all variables in common across \texttt{x} and \texttt{y}. A message lists the variables so that you can check they're correct; suppress the message by supplying \texttt{by} explicitly.

  To join on different variables between \texttt{x} and \texttt{y}, use a \texttt{join_by()} specification. For example, \texttt{join_by(a == b)} will match \texttt{x$a} to \texttt{y$b}. 


To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match $x$a to $y$b and $x$c to $y$d. If the column names are the same between $x$ and $y$, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at `?join_by` for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins $x$a to $y$a and $x$b to $y$b. If variable names differ between $x$ and $y$, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of $x$ and $y$, see `cross_join()`.

**copy**  
If $x$ and $y$ are not from the same data source, and `copy` is `TRUE`, then $y$ will be copied into a temporary table in same database as $x$. `_join()` will automatically run `ANALYZE` on the created table in the hope that this will make you queries as efficient as possible by giving more data to the query planner.

This allows you to join tables across srcs, but it’s potentially expensive operation so you must opt into it.

**suffix**  
If there are non-joined duplicate variables in $x$ and $y$, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

**auto_index**  
if `copy` is `TRUE`, automatically create indices for the variables in `by`. This may speed up the join if there are matching indexes in $x$.

...  
Other parameters passed onto methods.

**sql_on**  
A custom join predicate as an SQL expression. Usually joins use column equality, but you can perform more complex queries by supply `sql_on` which should be a SQL expression that uses `LHS` and `RHS` aliases to refer to the left-hand side or right-hand side of the join respectively.

---

### j_invoke

**Invoke a Java function.**

Invoke a Java function and force return value of the call to be retrieved as a Java object reference.

### Usage

```
j_invoke(obj, method, ...)  
j_invoke_static(sc, class, method, ...)  
j_invoke_new(sc, class, ...)  
```
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>job</code></td>
<td>An R object acting as a Java object reference (typically, a <code>spark_jobj</code>).</td>
</tr>
<tr>
<td><code>method</code></td>
<td>The name of the method to be invoked.</td>
</tr>
<tr>
<td><code>...</code></td>
<td>Optional arguments, currently unused.</td>
</tr>
<tr>
<td><code>sc</code></td>
<td>A <code>spark_connection</code>.</td>
</tr>
<tr>
<td><code>class</code></td>
<td>The name of the Java class whose methods should be invoked.</td>
</tr>
</tbody>
</table>

left_join

Left join

Description

See `left_join` for more details.

list_sparklyr_jars

list all sparklyr-*.jar files that have been built

Description

list all sparklyr-*.jar files that have been built

Usage

`list_sparklyr_jars()`

livy_config

Create a Spark Configuration for Livy

Description

Create a Spark Configuration for Livy

Usage

```r
livy_config(
  config = spark_config(),
  username = NULL,
  password = NULL,
  negotiate = FALSE,
  custom_headers = list(`X-Requested-By` = "sparklyr"),
  proxy = NULL,
  curl_opts = NULL,
  ...
)
```
Arguments

- `config` Optional base configuration
- `username` The username to use in the Authorization header
- `password` The password to use in the Authorization header
- `negotiate` Whether to use gssnegotiate method or not
- `custom_headers` List of custom headers to append to http requests. Defaults to `list("X-Requested-By" = "sparklyr")`.
- `proxy` Either NULL or a proxy specified by `httr::use_proxy()`. Defaults to NULL.
- `curl_opts` List of CURL options (e.g., `verbose`, `connecttimeout`, `dns_cache_timeout`, etc, see `httr::httr_options()` for a list of valid options) – NOTE: these configurations are for libcurl only and separate from HTTP headers or Livy session parameters.
- `...` additional Livy session parameters

Details

Extends a Spark `spark_config()` configuration with settings for Livy. For instance, `username` and `password` define the basic authentication settings for a Livy session.

The default value of "custom_headers" is set to `list("X-Requested-By" = "sparklyr")` in order to facilitate connection to Livy servers with CSRF protection enabled.

Additional parameters for Livy sessions are:

- `proxy_user` User to impersonate when starting the session
- `jars` Jars to be used in this session
- `py_files` Python files to be used in this session
- `files` Files to be used in this session
- `driver_memory` Amount of memory to use for the driver process
- `driver_cores` Number of cores to use for the driver process
- `executor_memory` Amount of memory to use per executor process
- `executor_cores` Number of cores to use for each executor
- `num_executors` Number of executors to launch for this session
- `archives` Archives to be used in this session
- `queue` The name of the YARN queue to which submitted
- `name` The name of this session
- `heartbeat_timeout` Timeout in seconds to which session be orphaned
- `conf` Spark configuration properties (Map of key=value)

Note that `queue` is supported only by version 0.4.0 of Livy or newer. If you are using the older one, specify `queue` via `config` (e.g. `config = spark_config(spark.yarn.queue = "my_queue")`).

Value

Named list with configuration data
livy_service_start  Start Livy

Description
Starts the livy service.
Stops the running instances of the livy service.

Usage
livy_service_start(
  version = NULL,
  spark_version = NULL,
  stdout = "",
  stderr = "",
  ...
)
livy_service_stop()

Arguments
version               The version of 'livy' to use.
spark_version         The version of 'spark' to connect to.
stdout, stderr        where output to 'stdout' or 'stderr' should be sent. Same options as system2.
...                    Optional arguments; currently unused.

ml-params  Spark ML – ML Params

Description
Helper methods for working with parameters for ML objects.

Usage
ml_is_set(x, param, ...)
ml_param_map(x, ...)
ml_param(x, param, allow_null = FALSE, ...)
ml_params(x, params = NULL, allow_null = FALSE, ...)
Arguments

- `x` A Spark ML object, either a pipeline stage or an evaluator.
- `param` The parameter to extract or set.
- `...` Optional arguments; currently unused.
- `allow_null` Whether to allow NULL results when extracting parameters. If FALSE, an error will be thrown if the specified parameter is not found. Defaults to FALSE.
- `params` A vector of parameters to extract.

Description

Save/load Spark ML objects

Usage

```r
ml_save(x, path, overwrite = FALSE, ...)
```

```r
## S3 method for class 'ml_model'
ml_save(
  x,
  path,
  overwrite = FALSE,
  type = c("pipeline_model", "pipeline"),
  ...
)
```

```r
ml_load(sc, path)
```

Arguments

- `x` A ML object, which could be a `ml_pipeline_stage` or a `ml_model`.
- `path` The path where the object is to be serialized/deserialized.
- `overwrite` Whether to overwrite the existing path, defaults to FALSE.
- `...` Optional arguments; currently unused.
- `type` Whether to save the pipeline model or the pipeline.
- `sc` A Spark connection.
**Value**

`ml_save()` serializes a Spark object into a format that can be read back into `sparklyr` or by the Scala or PySpark APIs. When called on `ml_model` objects, i.e. those that were created via the `tbl_spark - formula` signature, the associated pipeline model is serialized. In other words, the saved model contains both the data processing (`RFormulaModel`) stage and the machine learning stage.

`ml_load()` reads a saved Spark object into `sparklyr`. It calls the correct Scala load method based on parsing the saved metadata. Note that a `PipelineModel` object saved from a sparklyr `ml_model` via `ml_save()` will be read back in as an `ml_pipeline_model`, rather than the `ml_model` object.

---

**Description**

Methods for transformation, fit, and prediction. These are mirrors of the corresponding `sdf-transform-methods`.

**Usage**

`is_ml_transformer(x)`

`is_ml_estimator(x)`

`ml_fit(x, dataset, ...)`

```
## Default S3 method:
ml_fit(x, dataset, ...)
```

`ml_transform(x, dataset, ...)`

`ml_fit_and_transform(x, dataset, ...)`

`ml_predict(x, dataset, ...)`

```
## S3 method for class 'ml_model_classification'
ml_predict(x, dataset, probability_prefix = "probability_", ...)
```

**Arguments**

- `x` A `ml_estimator`, `ml_transformer` (or a list thereof), or `ml_model` object.
- `dataset` A `tbl_spark`.
- `...` Optional arguments; currently unused.
- `probability_prefix` String used to prepend the class probability output columns.
**Details**

These methods are

**Value**

When \( x \) is an estimator, `ml_fit()` returns a transformer whereas `ml_fit_and_transform()` returns a transformed dataset. When \( x \) is a transformer, `ml_transform()` and `ml_predict()` return a transformed dataset. When `ml_predict()` is called on a `ml_model` object, additional columns (e.g. probabilities in case of classification models) are appended to the transformed output for the user’s convenience.

---

**Spark ML – Tuning**

**Description**

Perform hyper-parameter tuning using either K-fold cross validation or train-validation split.

**Usage**

```r
ml_sub_models(model)

ml_validation_metrics(model)

ml_cross_validator(
  x,
  estimator = NULL,
  estimator_param_maps = NULL,
  evaluator = NULL,
  num_folds = 3,
  collect_sub_models = FALSE,
  parallelism = 1,
  seed = NULL,
  uid = random_string("cross_validator_"),
  ...
)

ml_train_validation_split(
  x,
  estimator = NULL,
  estimator_param_maps = NULL,
  evaluator = NULL,
  train_ratio = 0.75,
  collect_sub_models = FALSE,
  parallelism = 1,
  seed = NULL,
  uid = random_string("train_validation_split_"),
)```
Arguments

- **model**: A cross validation or train-validation-split model.
- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **estimator**: A ml_estimator object.
- **estimator_param_maps**: A named list of stages and hyper-parameter sets to tune. See details.
- **evaluator**: A ml_evaluator object, see ml_evaluator.
- **num_folds**: Number of folds for cross validation. Must be >= 2. Default: 3
- **collect_sub_models**: Whether to collect a list of sub-models trained during tuning. If set to FALSE, then only the single best sub-model will be available after fitting. If set to true, then all sub-models will be available. Warning: For large models, collecting all sub-models can cause OOMs on the Spark driver.
- **parallelism**: The number of threads to use when running parallel algorithms. Default is 1 for serial execution.
- **seed**: A random seed. Set this value if you need your results to be reproducible across repeated calls.
- **uid**: A character string used to uniquely identify the ML estimator.
- **...**: Optional arguments; currently unused.
- **train_ratio**: Ratio between train and validation data. Must be between 0 and 1. Default: 0.75

Details

- **ml_cross_validator()** performs k-fold cross validation while **ml_train_validation_split()** performs tuning on one pair of train and validation datasets.

Value

The object returned depends on the class of x.

- **spark_connection**: When x is a spark_connection, the function returns an instance of a ml_cross_validator or ml_train_validation_split object.
- **ml_pipeline**: When x is a ml_pipeline, the function returns a ml_pipeline with the tuning estimator appended to the pipeline.
- **tbl_spark**: When x is a tbl_spark, a tuning estimator is constructed then immediately fit with the input tbl_spark, returning a ml_cross_validation_model or a ml_train_validation_split_model object.

For cross validation, **ml_sub_models()** returns a nested list of models, where the first layer represents fold indices and the second layer represents param maps. For train-validation split, **ml_sub_models()** returns a list of models, corresponding to the order of the estimator param maps.

**ml_validation_metrics()** returns a data frame of performance metrics and hyperparameter combinations.
ml_aft_survival_regression

**Spark ML – Survival Regression**

**Description**

Fit a parametric survival regression model named accelerated failure time (AFT) model (see Accelerated failure time model (Wikipedia)) based on the Weibull distribution of the survival time.

**Usage**

```r
ml_aft_survival_regression()
```
ml_aft_survival_regression

```r
ml_aft_survival_regression(
  x,
  formula = NULL,
  censor_col = "censor",
  quantile_probabilities = c(0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99),
  fit_intercept = TRUE,
  max_iter = 100L,
  tol = 1e-06,
  aggregation_depth = 2,
  quantiles_col = NULL,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  uid = random_string("aft_survival_regression_"),
  ...)
```

```r
ml_survival_regression(
  x,
  formula = NULL,
  censor_col = "censor",
  quantile_probabilities = c(0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.99),
  fit_intercept = TRUE,
  max_iter = 100L,
  tol = 1e-06,
  aggregation_depth = 2,
  quantiles_col = NULL,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  uid = random_string("aft_survival_regression_"),
  response = NULL,
  features = NULL,
  ...)
```

**Arguments**

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **formula**: Used when `x` is a `tbl_spark`. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see `ft_r_formula` for details.
- **censor_col**: Censor column name. The value of this column could be 0 or 1. If the value is 1, it means the event has occurred i.e. uncensored; otherwise censored.
- **quantile_probabilities**: Quantile probabilities array. Values of the quantile probabilities array should be in the range (0, 1) and the array should be non-empty.
- **fit_intercept**: Boolean; should the model be fit with an intercept term?
- **max_iter**: The maximum number of iterations to use.
tol  \hspace{1cm} \text{Param for the convergence tolerance for iterative algorithms.}
aggregation_depth  \hspace{1cm} (Spark 2.1.0+) Suggested depth for treeAggregate (>= 2).
quantiles_col  \hspace{1cm} \text{Quantiles column name. This column will output quantiles of corresponding quantileProbabilities if it is set.}
features_col  \hspace{1cm} \text{Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by ft_r_formula.}
label_col  \hspace{1cm} \text{Label column name. The column should be a numeric column. Usually this column is output by ft_r_formula.}
prediction_col  \hspace{1cm} \text{Prediction column name.}
uid  \hspace{1cm} \text{A character string used to uniquely identify the ML estimator.}
...  \hspace{1cm} \text{Optional arguments; see Details.}
response  \hspace{1cm} \text{(Deprecated) The name of the response column (as a length-one character vector.)}
features  \hspace{1cm} \text{(Deprecated) The name of features (terms) to use for the model fit.}

\textbf{Details}

\texttt{ml_survival_regression()} is an alias for \texttt{ml_aft_survival_regression()} for backwards compatibility.

\textbf{Value}

The object returned depends on the class of \texttt{x}. If it is a \texttt{spark_connection}, the function returns a \texttt{ml_estimator} object. If it is a \texttt{ml_pipeline}, it will return a pipeline with the predictor appended to it. If a \texttt{tbl_spark}, it will return a \texttt{tbl_spark} with the predictions added to it.

\textbf{See Also}

Other \texttt{ml} algorithms: \texttt{ml_decision_tree_classifier()}, \texttt{ml_gbt_classifier()}, \texttt{ml_generalized_linear_regression()}, \texttt{ml_isotonic_regression()}, \texttt{ml_linear_regression()}, \texttt{ml_linear_svc()}, \texttt{ml_logistic_regression()}, \texttt{ml_multilayer_perceptron_classifier()}, \texttt{ml_naive_bayes()}, \texttt{ml_one_vs_rest()}, \texttt{ml_random_forest_classifier}.

\textbf{Examples}

```r
## Not run:
library(survival)
library(sparklyr)
sc <- spark_connect(master = "local")
ovarian_tbl <- sdf_copy_to(sc, ovarian, name = "ovarian_tbl", overwrite = TRUE)
partitions <- ovarian_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

ovarian_training <- partitions$training
```
ovarian_test <- partitions$test

sur_reg <- ovarian_training %>%
  ml_aft_survival_regression(futime ~ ecog_ps + rx + age + resid_ds, censor_col = "fustat")

pred <- ml_predict(sur_reg, ovarian_test)
pred

## End(Not run)

---

**ml_als**  
*Spark ML – ALS*

**Description**

Perform recommendation using Alternating Least Squares (ALS) matrix factorization.

**Usage**

```r
ml_als(
  x,
  formula = NULL,
  rating_col = "rating",
  user_col = "user",
  item_col = "item",
  rank = 10,
  reg_param = 0.1,
  implicit_prefs = FALSE,
  alpha = 1,
  nonnegative = FALSE,
  max_iter = 10,
  num_user_blocks = 10,
  num_item_blocks = 10,
  checkpoint_interval = 10,
  cold_start_strategy = "nan",
  intermediate_storage_level = "MEMORY_AND_DISK",
  final_storage_level = "MEMORY_AND_DISK",
  uid = random_string("als_"),
  ...
)
```

ml_recommend(model, type = c("items", "users"), n = 1)

**Arguments**

- **x**  
  A `spark_connection`, `ml_pipeline`, or `tbl_spark`. 

---
formula Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see `ft_r_formula` for details. The ALS model requires a specific formula format, please use `rating_col ~ user_col + item_col`.

ing_col Column name for ratings. Default: "rating"

g_col Column name for user ids. Ids must be integers. Other numeric types are supported for this column, but will be cast to integers as long as they fall within the integer value range. Default: "user"

item_col Column name for item ids. Ids must be integers. Other numeric types are supported for this column, but will be cast to integers as long as they fall within the integer value range. Default: "item"

rank Rank of the matrix factorization (positive). Default: 10

reg_param Regularization parameter.

implicit_prefs Whether to use implicit preference. Default: FALSE.

alpha Alpha parameter in the implicit preference formulation (nonnegative).

nonnegative Whether to apply nonnegativity constraints. Default: FALSE.

max_iter Maximum number of iterations.

num_user_blocks Number of user blocks (positive). Default: 10

num_item_blocks Number of item blocks (positive). Default: 10

checkpoint_interval Set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations, defaults to 10.

cold_start_strategy (Spark 2.2.0+) Strategy for dealing with unknown or new users/items at prediction time. This may be useful in cross-validation or production scenarios, for handling user/item ids the model has not seen in the training data. Supported values: - "nan": predicted value for unknown ids will be NaN. - "drop": rows in the input DataFram containing unknown ids will be dropped from the output DataFram containing predictions. Default: "nan".

intermediate_storage_level (Spark 2.0.0+) StorageLevel for intermediate datasets. Pass in a string representation of StorageLevel. Cannot be "NONE". Default: "MEMORY_AND_DISK".

final_storage_level (Spark 2.0.0+) StorageLevel for ALS model factors. Pass in a string representation of StorageLevel. Default: "MEMORY_AND_DISK".

uid A character string used to uniquely identify the ML estimator.

... Optional arguments; currently unused.

model An ALS model object

type What to recommend, one of items or users

n Maximum number of recommendations to return
Details

`ml_recommend()` returns the top `n` users/items recommended for each item/user, for all items/users. The output has been transformed (exploded and separated) from the default Spark outputs to be more user friendly.

Value

ALS attempts to estimate the ratings matrix `R` as the product of two lower-rank matrices, `X` and `Y`, i.e. $X \times Y^T = R$. Typically these approximations are called "factor" matrices. The general approach is iterative. During each iteration, one of the factor matrices is held constant, while the other is solved for using least squares. The newly-solved factor matrix is then held constant while solving for the other factor matrix.

This is a blocked implementation of the ALS factorization algorithm that groups the two sets of factors (referred to as "users" and "products") into blocks and reduces communication by only sending one copy of each user vector to each product block on each iteration, and only for the product blocks that need that user's feature vector. This is achieved by pre-computing some information about the ratings matrix to determine the "out-links" of each user (which blocks of products it will contribute to) and "in-link" information for each product (which of the feature vectors it receives from each user block it will depend on). This allows us to send only an array of feature vectors between each user block and product block, and have the product block find the users' ratings and update the products based on these messages.

For implicit preference data, the algorithm used is based on "Collaborative Filtering for Implicit Feedback Datasets", available at doi:10.1109/ICDM.2008.22, adapted for the blocked approach used here.

Essentially instead of finding the low-rank approximations to the rating matrix `R`, this finds the approximations for a preference matrix `P` where the elements of `P` are 1 if `r` is greater than 0 and 0 if `r` is less than or equal to 0. The ratings then act as 'confidence' values related to strength of indicated user preferences rather than explicit ratings given to items.

The object returned depends on the class of `x`.

- `spark_connection`: When `x` is a `spark_connection`, the function returns an instance of a `ml_als` recommender object, which is an Estimator.
- `ml_pipeline`: When `x` is a `ml_pipeline`, the function returns a `ml_pipeline` with the recommender appended to the pipeline.
- `tbl_spark`: When `x` is a `tbl_spark`, a recommender estimator is constructed then immediately fit with the input `tbl_spark`, returning a recommendation model, i.e. `ml_als_model`.

Examples

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local")

movies <- data.frame(
  user = c(1, 2, 0, 1, 2, 0),
  item = c(1, 1, 1, 2, 2, 0),
)
```
ml_als_tidiers

Tidying methods for Spark ML ALS

Description

These methods summarize the results of Spark ML models into tidy forms.

Usage

```r
## S3 method for class 'ml_model_als'
tidy(x, ...)
```

```r
## S3 method for class 'ml_model_als'
augment(x, newdata = NULL, ...)
```

```r
## S3 method for class 'ml_model_als'
glance(x, ...)
```

Arguments

- `x` a Spark ML model.
- `...` extra arguments (not used.)
- `newdata` a tbl_spark of new data to use for prediction.

ml_bisecting_kmeans

Spark ML – Bisecting K-Means Clustering

Description

A bisecting k-means algorithm based on the paper "A comparison of document clustering techniques" by Steinbach, Karypis, and Kumar, with modification to fit Spark. The algorithm starts from a single cluster that contains all points. Iteratively it finds divisible clusters on the bottom level and bisects each of them using k-means, until there are k leaf clusters in total or no leaf clusters are divisible. The bisecting steps of clusters on the same level are grouped together to increase parallelism. If bisecting all divisible clusters on the bottom level would result more than k leaf clusters, larger clusters get higher priority.
Usage

ml_bisecting_kmeans(
  x,
  formula = NULL,
  k = 4,
  max_iter = 20,
  seed = NULL,
  min_divisible_cluster_size = 1,
  features_col = "features",
  prediction_col = "prediction",
  uid = random_string("bisecting_bisecting_kmeans_"),
  ...)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.
formula Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see ft_r_formula for details.
k The number of clusters to create
max_iter The maximum number of iterations to use.
seed A random seed. Set this value if you need your results to be reproducible across repeated calls.
min_divisible_cluster_size The minimum number of points (if greater than or equal to 1.0) or the minimum proportion of points (if less than 1.0) of a divisible cluster (default: 1.0).
features_col Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by ft_r_formula.
prediction_col Prediction column name.
uid A character string used to uniquely identify the ML estimator.
...

Examples

## Not run:
library(dplyr)
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)
iris_tbl %>%
  select(-Species) %>%
  ml_bisecting_kmeans(k = 4, Species ~ .)

## End(Not run)

---

### ml_chisquare_test

Chi-square hypothesis testing for categorical data.

#### Description

Conduct Pearson’s independence test for every feature against the label. For each feature, the (feature, label) pairs are converted into a contingency matrix for which the Chi-squared statistic is computed. All label and feature values must be categorical.

#### Usage

```
ml_chisquare_test(x, features, label)
```

#### Arguments

- **x**: A tbl_spark.
- **features**: The name(s) of the feature columns. This can also be the name of a single vector column created using `ft_vector_assembler()`.
- **label**: The name of the label column.

#### Value

A data frame with one row for each (feature, label) pair with p-values, degrees of freedom, and test statistics.

#### Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)
features <- c("Petal_Width", "Petal_Length", "Sepal_Length", "Sepal_Width")
ml_chisquare_test(iris_tbl, features = features, label = "Species")
## End(Not run)
```
ml_clustering_evaluator

Spark ML - Clustering Evaluator

Description
Evaluator for clustering results. The metric computes the Silhouette measure using the squared Euclidean distance. The Silhouette is a measure for the validation of the consistency within clusters. It ranges between 1 and -1, where a value close to 1 means that the points in a cluster are close to the other points in the same cluster and far from the points of the other clusters.

Usage
ml_clustering_evaluator(
  x,
  features_col = "features",
  prediction_col = "prediction",
  metric_name = "silhouette",
  uid = random_string("clustering_evaluator_"),
  ...)

Arguments
  x A spark_connection object or a tbl_spark containing label and prediction columns. The latter should be the output of sdf_predict.
  features_col Name of features column.
  prediction_col Name of the prediction column.
  metric_name The performance metric. Currently supports "silhouette".
  uid A character string used to uniquely identify the ML estimator.
  ... Optional arguments; currently unused.

Value
The calculated performance metric

Examples
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

partitions <- iris_ttbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

iris_training <- partitions$training
iris_test <- partitions$test

formula <- Species ~ .

# Train the models
kmeans_model <- ml_kmeans(iris_training, formula = formula)
b_kmeans_model <- ml_bisecting_kmeans(iris_training, formula = formula)
gmm_model <- ml_gaussian_mixture(iris_training, formula = formula)

# Predict
pred_kmeans <- ml_predict(kmeans_model, iris_test)
pred_b_kmeans <- ml_predict(b_kmeans_model, iris_test)
pred_gmm <- ml_predict(gmm_model, iris_test)

# Evaluate
ml_clustering_evaluator(pred_kmeans)
ml_clustering_evaluator(pred_b_kmeans)
ml_clustering_evaluator(pred_gmm)

## End(Not run)

---

**ml_corr**

*Compute correlation matrix*

**Description**

Compute correlation matrix

**Usage**

```
ml_corr(x, columns = NULL, method = c("pearson", "spearman"))
```

**Arguments**

- **x**
  - A tbl_spark.

- **columns**
  - The names of the columns to calculate correlations of. If only one column is specified, it must be a vector column (for example, assembled using `ft_vector_assembler()`).

- **method**
  - The method to use, either "pearson" or "spearman".

**Value**

A correlation matrix organized as a data frame.
Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

features <- c("Petal_Width", "Petal_Length", "Sepal_Length", "Sepal_Width")

ml_corr(iris_tbl, columns = features, method = "pearson")

## End(Not run)
```

### Description

Perform classification and regression using decision trees.

### Usage

```r
ml_decision_tree_classifier(
  x,
  formula = NULL,
  max_depth = 5,
  max_bins = 32,
  min_instances_per_node = 1,
  min_info_gain = 0,
  impurity = "gini",
  seed = NULL,
  thresholds = NULL,
  cache_node_ids = FALSE,
  checkpoint_interval = 10,
  max_memory_in_mb = 256,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  probability_col = "probability",
  raw_prediction_col = "rawPrediction",
  uid = random_string("decision_tree_classifier_{")
)
```

```r
ml_decision_tree(
  x,
  formula = NULL,
  type = c("auto", "regression", "classification")
)
\texttt{ml_decision_tree_classifier}

\begin{verbatim}
features_col = "features",
label_col = "label",
prediction_col = "prediction",
variance_col = NULL,
probability_col = "probability",
raw_prediction_col = "rawPrediction",
checkpoint_interval = 10L,
impurity = "auto",
max_bins = 32L,
max_depth = 5L,
min_info_gain = 0,
min_instances_per_node = 1L,
seed = NULL,
thresholds = NULL,
cache_node_ids = FALSE,
max_memory_in_mb = 256L,
uid = random_string("decision_tree_"),
response = NULL,
features = NULL,
...
)

\texttt{ml_decision_tree_regressor(}
  \texttt{x},
  \texttt{formula = NULL},
  \texttt{max_depth = 5,}
  \texttt{max_bins = 32,}
  \texttt{min_instances_per_node = 1,}
  \texttt{min_info_gain = 0,}
  \texttt{impurity = "variance",}
  \texttt{seed = NULL,}
  \texttt{cache_node_ids = FALSE,}
  \texttt{checkpoint_interval = 10,}
  \texttt{max_memory_in_mb = 256,}
  \texttt{variance_col = NULL,}
  \texttt{features_col = "features",}
  \texttt{label_col = "label",}
  \texttt{prediction_col = "prediction",}
  \texttt{uid = random_string("decision_tree_regressor_"),}
  \texttt{...}
)
\end{verbatim}

**Arguments**

- **x**
  A \texttt{spark_connection}, \texttt{ml_pipeline}, or a \texttt{tbl_spark}.

- **formula**
  Used when \texttt{x} is a \texttt{tbl_spark}. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see \texttt{ft_r_formula} for details.
max_depth | Maximum depth of the tree (≥ 0); that is, the maximum number of nodes separating any leaves from the root of the tree.

max_bins | The maximum number of bins used for discretizing continuous features and for choosing how to split on features at each node. More bins give higher granularity.

min_instances_per_node | Minimum number of instances each child must have after split.

min_info_gain | Minimum information gain for a split to be considered at a tree node. Should be ≥ 0, defaults to 0.

impurity | Criterion used for information gain calculation. Supported: "entropy" and "gini" (default) for classification and "variance" (default) for regression. For ml_decision_tree, setting "auto" will default to the appropriate criterion based on model type.

seed | Seed for random numbers.

thresholds | Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0 excepting that at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the class’s threshold.

cache_node_ids | If FALSE, the algorithm will pass trees to executors to match instances with nodes. If TRUE, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Defaults to FALSE.

checkpoint_interval | Set checkpoint interval (≥ 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations, defaults to 10.

max_memory_in_mb | Maximum memory in MB allocated to histogram aggregation. If too small, then 1 node will be split per iteration, and its aggregates may exceed this size. Defaults to 256.

features_col | Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by ft_r_formula.

label_col | Label column name. The column should be a numeric column. Usually this column is output by ft_r_formula.

prediction_col | Prediction column name.

probability_col | Column name for predicted class conditional probabilities.

raw_prediction_col | Raw prediction (a.k.a. confidence) column name.

uid | A character string used to uniquely identify the ML estimator.

... | Optional arguments; see Details.

type | The type of model to fit. "regression" treats the response as a continuous variable, while "classification" treats the response as a categorical variable. When "auto" is used, the model type is inferred based on the response variable type – if it is a numeric type, then regression is used; classification otherwise.
**ml_decision_tree_classifier**

variance_col  (Optional) Column name for the biased sample variance of prediction.
response  (Deprecated) The name of the response column (as a length-one character vector.)
features  (Deprecated) The name of features (terms) to use for the model fit.

**Details**

`ml_decision_tree` is a wrapper around `ml_decision_tree_regressor.tbl_spark` and `ml_decision_tree_classifier.tbl_spark` and calls the appropriate method based on model type.

**Value**

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the predictor appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the predictions added to it.

**See Also**

Other ml algorithms: `ml_aft_survival_regression()`, `ml_gbt_classifier()`, `ml_generalized_linear_regression()`, `ml_isotonic_regression()`, `ml_linear_regression()`, `ml_linear_svc()`, `ml_logistic_regression()`, `ml_multilayer_perceptron_classifier()`, `ml_naive_bayes()`, `ml_one_vs_rest()`, `ml_random_forest_classifier()`

**Examples**

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

partitions <- iris_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

iris_training <- partitions$training
iris_test <- partitions$test

dt_model <- iris_training %>%
  ml_decision_tree(Species ~ .)
pred <- ml_predict(dt_model, iris_test)
ml_multiclass_classification_evaluator(pred)

## End(Not run)
```
Description

Loads the default stop words for the given language.

Usage

```r
ml_default_stop_words(
  sc,
  language = c("english", "danish", "dutch", "finnish", "french", "german", "hungarian",
              "italian", "norwegian", "portuguese", "russian", "spanish", "swedish", "turkish"),
  ...
)
```

Arguments

- `sc`: A `spark_connection`
- `language`: A character string.
- `...`: Optional arguments; currently unused.

Details


Value

A list of stop words.

See Also

- `ft_stop_words_remover`
ml_evaluate

Evaluate the Model on a Validation Set

Description
Compute performance metrics.

Usage

```r
ml_evaluate(x, dataset)

## S3 method for class 'ml_model_logistic_regression'
ml_evaluate(x, dataset)

## S3 method for class 'ml_logistic_regression_model'
ml_evaluate(x, dataset)

## S3 method for class 'ml_model_linear_regression'
ml_evaluate(x, dataset)

## S3 method for class 'ml_linear_regression_model'
ml_evaluate(x, dataset)

## S3 method for class 'ml_model_generalized_linear_regression'
ml_evaluate(x, dataset)

## S3 method for class 'ml_generalized_linear_regression_model'
ml_evaluate(x, dataset)

## S3 method for class 'ml_model_clustering'
ml_evaluate(x, dataset)

## S3 method for class 'ml_model_classification'
ml_evaluate(x, dataset)

## S3 method for class 'ml_evaluator'
ml_evaluate(x, dataset)
```

Arguments

- `x`: An ML model object or an evaluator object.
- `dataset`: The dataset to be validate the model on.

Examples

```r
## Not run:
sc <- spark_connect(master = "local")
```
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

ml_gaussian_mixture(iris_tbl, Species ~ .) %>%
  ml_evaluate(iris_tbl)

ml_kmeans(iris_tbl, Species ~ .) %>%
  ml_evaluate(iris_tbl)

ml_bisecting_kmeans(iris_tbl, Species ~ .) %>%
  ml_evaluate(iris_tbl)

## End(Not run)

---

**ml_evaluator**  
*Spark ML - Evaluators*

**Description**

A set of functions to calculate performance metrics for prediction models. Also see the Spark ML Documentation [https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.evaluation.package](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.evaluation.package)

**Usage**

```r
ml_binary_classification_evaluator(
  x,
  label_col = "label",
  raw_prediction_col = "rawPrediction",
  metric_name = "areaUnderROC",
  uid = random_string("binary_classification_evaluator_"),
  ...
)

ml_binary_classification_eval(
  x,
  label_col = "label",
  prediction_col = "prediction",
  metric_name = "areaUnderROC"
)

ml_multiclass_classification_evaluator(
  x,
  label_col = "label",
  prediction_col = "prediction",
  metric_name = "f1",
  uid = random_string("multiclass_classification_evaluator_"),
  ...
)
```
ml_classification_eval(
    x,
    label_col = "label",
    prediction_col = "prediction",
    metric_name = "f1"
)

ml_regression_evaluator(
    x,
    label_col = "label",
    prediction_col = "prediction",
    metric_name = "rmse",
    uid = random_string("regression_evaluator_"),
    ...
)

**Arguments**

- **x**
  A spark_connection object or a tbl_spark containing label and prediction columns. The latter should be the output of `sdf_predict`.

- **label_col**
  Name of column string specifying which column contains the true labels or values.

- **raw_prediction_col**
  Raw prediction (a.k.a. confidence) column name.

- **metric_name**
  The performance metric. See details.

- **uid**
  A character string used to uniquely identify the ML estimator.

- **...**
  Optional arguments; currently unused.

- **prediction_col**
  Name of the column that contains the predicted label or value NOT the scored probability. Column should be of type `Double`.

**Details**

The following metrics are supported

- **Binary Classification**: areaUnderROC (default) or areaUnderPR (not available in Spark 2.X.)
- **Multiclass Classification**: f1 (default), precision, recall, weightedPrecision, weightedRecall or accuracy; for Spark 2.X: f1 (default), weightedPrecision, weightedRecall or accuracy.
- **Regression**: rmse (root mean squared error, default), mse (mean squared error), r2, or mae (mean absolute error.)

`ml_binary_classification_eval()` is an alias for `ml_binary_classification_evaluator()` for backwards compatibility.

`ml_classification_eval()` is an alias for `ml_multiclass_classification_evaluator()` for backwards compatibility.
Value

The calculated performance metric

Examples

```r
## Not run:
sd <- spark_connect(master = "local")
mtcars_tbl <- sdf_copy_to(sc, mtcars, name = "mtcars_tbl", overwrite = TRUE)

partitions <- mtcars_tbl %>%
sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

mtcars_training <- partitions$training
mtcars_test <- partitions$test

# for multiclass classification
rf_model <- mtcars_training %>%
  ml_random_forest(cyl ~ ., type = "classification")
pred <- ml_predict(rf_model, mtcars_test)
ml_multiclass_classification_evaluator(pred)

# for regression
rf_model <- mtcars_training %>%
  ml_random_forest(cyl ~ ., type = "regression")
pred <- ml_predict(rf_model, mtcars_test)
ml_regression_evaluator(pred, label_col = "cyl")

# for binary classification
rf_model <- mtcars_training %>%
  ml_random_forest(am ~ gear + carb, type = "classification")
pred <- ml_predict(rf_model, mtcars_test)
ml_binary_classification_evaluator(pred)

## End(Not run)
```

---

**Spark ML - Feature Importance for Tree Models**

Description

Spark ML - Feature Importance for Tree Models
**ml_fpgrowth**

*Usage*

```r
ml_feature_importances(model, ...)
ml_tree_feature_importance(model, ...)
```

*Arguments*

- `model`: A decision tree-based model.
- `...`: Optional arguments; currently unused.

*Value*

For `ml_model`, a sorted data frame with feature labels and their relative importance. For `ml_prediction_model`, a vector of relative importances.

---

**ml_fpgrowth**

*Frequent Pattern Mining – FPgrowth*

*Description*

A parallel FP-growth algorithm to mine frequent itemsets.

*Usage*

```r
ml_fpgrowth(
  x,
  items_col = "items",
  min_confidence = 0.8,
  min_support = 0.3,
  prediction_col = "prediction",
  uid = random_string("fpgrowth_"),
  ...
)
```

- `x`: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `items_col`: Items column name. Default: "items".
- `min_confidence`: Minimal confidence for generating Association Rule. `min_confidence` will not affect the mining for frequent itemsets, but will affect the association rules generation. Default: 0.8.
**min_support**  Minimal support level of the frequent pattern. [0.0, 1.0]. Any pattern that appears more than (min_support * size-of-the-dataset) times will be output in the frequent itemsets. Default: 0.3

**prediction_col**  Prediction column name.

**uid**  A character string used to uniquely identify the ML estimator.

**...**  Optional arguments; currently unused.

**model**  A fitted FPGrowth model returned by `ml_fpgrowth()`

---

**Description**

This class performs expectation maximization for multivariate Gaussian Mixture Models (GMMs). A GMM represents a composite distribution of independent Gaussian distributions with associated "mixing" weights specifying each’s contribution to the composite. Given a set of sample points, this class will maximize the log-likelihood for a mixture of k Gaussians, iterating until the log-likelihood changes by less than `tol`, or until it has reached the max number of iterations. While this process is generally guaranteed to converge, it is not guaranteed to find a global optimum.

**Usage**

```r
ml_gaussian_mixture(
  x,
  formula = NULL,
  k = 2,
  max_iter = 100,
  tol = 0.01,
  seed = NULL,
  features_col = "features",
  prediction_col = "prediction",
  probability_col = "probability",
  uid = random_string("gaussian_mixture_"),
  ...
)
```

**Arguments**

- **x**  A spark_connection, ml_pipeline, or a tbl_spark.
- **formula**  Used when `x` is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see `ft_r_formula` for details.
- **k**  The number of clusters to create
- **max_iter**  The maximum number of iterations to use.
- **tol**  Param for the convergence tolerance for iterative algorithms.

---

**Spark ML – Gaussian Mixture clustering.**

This class performs expectation maximization for multivariate Gaussian Mixture Models (GMMs). A GMM represents a composite distribution of independent Gaussian distributions with associated "mixing" weights specifying each’s contribution to the composite. Given a set of sample points, this class will maximize the log-likelihood for a mixture of k Gaussians, iterating until the log-likelihood changes by less than `tol`, or until it has reached the max number of iterations. While this process is generally guaranteed to converge, it is not guaranteed to find a global optimum.
**seed** A random seed. Set this value if you need your results to be reproducible across repeated calls.

**features_col** Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.

**prediction_col** Prediction column name.

**probability_col** Column name for predicted class conditional probabilities. Note: Not all models output well-calibrated probability estimates! These probabilities should be treated as confidences, not precise probabilities.

**uid** A character string used to uniquely identify the ML estimator.

**...** Optional arguments, see Details. #`@return` The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the predictor appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the predictions added to it.

### Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

gmm_model <- ml_gaussian_mixture(iris_tbl, Species ~ .)
pred <- sdf_predict(iris_tbl, gmm_model)
ml_clustering_evaluator(pred)
## End(Not run)
```

---

### Description

Perform binary classification and regression using gradient boosted trees. Multiclass classification is not supported yet.

### Usage

```r
ml_gbt_classifier(
  x,
  formula = NULL,
  max_iter = 20,
  max_depth = 5,
  step_size = 0.1,
  subsampling_rate = 1,
```
feature_subset_strategy = "auto",
min_instances_per_node = 1L,
max_bins = 32,
min_info_gain = 0,
loss_type = "logistic",
seed = NULL,
thresholds = NULL,
checkpoint_interval = 10,
cache_node_ids = FALSE,
max_memory_in_mb = 256,
features_col = "features",
label_col = "label",
prediction_col = "prediction",
probability_col = "probability",
raw_prediction_col = "rawPrediction",
uid = random_string("gbt_classifier_"),
...)

ml_gradient_boosted_trees(
x,
formula = NULL,
type = c("auto", "regression", "classification"),
features_col = "features",
label_col = "label",
prediction_col = "prediction",
probability_col = "probability",
raw_prediction_col = "rawPrediction",
checkpoint_interval = 10,
loss_type = c("auto", "logistic", "squared", "absolute"),
max_bins = 32,
max_depth = 5,
max_iter = 20L,
min_info_gain = 0,
min_instances_per_node = 1,
step_size = 0.1,
subsampling_rate = 1,
feature_subset_strategy = "auto",
seed = NULL,
thresholds = NULL,
cache_node_ids = FALSE,
max_memory_in_mb = 256,
uid = random_string("gradient_boosted_trees_"),
response = NULL,
features = NULL,
...)}
ml_gbt_regressor(
  x,
  formula = NULL,
  max_iter = 20,
  max_depth = 5,
  step_size = 0.1,
  subsampling_rate = 1,
  feature_subset_strategy = "auto",
  min_instances_per_node = 1,
  max_bins = 32,
  min_info_gain = 0,
  loss_type = "squared",
  seed = NULL,
  checkpoint_interval = 10,
  cache_node_ids = FALSE,
  max_memory_in_mb = 256,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  uid = random_string("gbt_regressor_"),
  ...
)

Arguments

  x  A spark_connection, ml_pipeline, or a tbl_spark.
  formula  Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see ft_r_formula for details.
  max_iter  Maximum number of iterations.
  max_depth  Maximum depth of the tree (>= 0); that is, the maximum number of nodes separating any leaves from the root of the tree.
  step_size  Step size (a.k.a. learning rate) in interval (0, 1] for shrinking the contribution of each estimator. (default = 0.1)
  subsampling_rate  Fraction of the training data used for learning each decision tree, in range (0, 1]. (default = 1.0)
  feature_subset_strategy  The number of features to consider for splits at each tree node. See details for options.
  min_instances_per_node  Minimum number of instances each child must have after split.
  max_bins  The maximum number of bins used for discretizing continuous features and for choosing how to split on features at each node. More bins give higher granularity.
  min_info_gain  Minimum information gain for a split to be considered at a tree node. Should be >= 0, defaults to 0.
loss_type  Loss function which GBT tries to minimize. Supported: "squared" (L2) and "absolute" (L1) (default = squared) for regression and "logistic" (default) for classification. For ml_gradient_boosted_trees, setting "auto" will default to the appropriate loss type based on model type.

seed  Seed for random numbers.

thresholds  Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0 excepting that at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the class’s threshold.

checkpoint_interval  Set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations, defaults to 10.

cache_node_ids  If FALSE, the algorithm will pass trees to executors to match instances with nodes. If TRUE, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Defaults to FALSE.

max_memory_in_mb  Maximum memory in MB allocated to histogram aggregation. If too small, then 1 node will be split per iteration, and its aggregates may exceed this size. Defaults to 256.

features_col  Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by ft_r_formula.

label_col  Label column name. The column should be a numeric column. Usually this column is output by ft_r_formula.

prediction_col  Prediction column name.

probability_col  Column name for predicted class conditional probabilities.

raw_prediction_col  Raw prediction (a.k.a. confidence) column name.

uid  A character string used to uniquely identify the ML estimator.

...  Optional arguments; see Details.

type  The type of model to fit. "regression" treats the response as a continuous variable, while "classification" treats the response as a categorical variable. When "auto" is used, the model type is inferred based on the response variable type – if it is a numeric type, then regression is used; classification otherwise.

response  (Deprecated) The name of the response column (as a length-one character vector.)

features  (Deprecated) The name of features (terms) to use for the model fit.

Details

The supported options for feature_subset_strategy are

- "auto": Choose automatically for task: If num_trees == 1, set to "all". If num_trees > 1 (forest), set to "sqrt" for classification and to "onethird" for regression.
ml_gbt_classifier

- "all": use all features
- "onethird": use 1/3 of the features
- "sqrt": use use sqrt(number of features)
- "log2": use log2(number of features)
- "n": when n is in the range (0, 1.0], use n * number of features. When n is in the range (1, number of features), use n features. (default = "auto")

ml_gradient_boosted_trees is a wrapper around ml_gbt_regressor.tbl_spark and ml_gbt_classifier.tbl_spark and calls the appropriate method based on model type.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.

See Also

Other ml algorithms: ml_aft_survival_regression(), ml_decision_tree_classifier(), ml_generalized_linear_regression(), ml_isotonic_regression(), ml_linear_regression(), ml_linear_svc(), ml_logistic_regression(), ml_multilayer_perceptron_classifier(), ml_naive_bayes(), ml_one_vs_rest(), ml_random_forest_classifier()

Examples

```r
## Not run:
sd <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

partitions <- iris_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)
iris_training <- partitions$training
iris_test <- partitions$test
gbt_model <- iris_training %>%
  ml_gradient_boosted_trees(Sepal_Length ~ Petal_Length + Petal_Width)
pred <- ml_predict(gbt_model, iris_test)
ml_regression_evaluator(pred, label_col = "Sepal_Length")

## End(Not run)
```
**ml_generalized_linear_regression**

*Spark ML – Generalized Linear Regression*

**Description**

Perform regression using Generalized Linear Model (GLM).

**Usage**

```r
ml_generalized_linear_regression(
  x,
  formula = NULL,
  family = "gaussian",
  link = NULL,
  fit_intercept = TRUE,
  offset_col = NULL,
  link_power = NULL,
  link_prediction_col = NULL,
  reg_param = 0,
  max_iter = 25,
  weight_col = NULL,
  solver = "irls",
  tol = 1e-06,
  variance_power = 0,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  uid = random_string("generalized_linear_regression_"),
  ...)
```

**Arguments**

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **formula**: Used when `x` is a `tbl_spark`. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see `ft_r_formula` for details.
- **family**: Name of family which is a description of the error distribution to be used in the model. Supported options: "gaussian", "binomial", "poisson", "gamma" and "tweedie". Default is "gaussian".
- **link**: Name of link function which provides the relationship between the linear predictor and the mean of the distribution function. See for supported link functions.
- **fit_intercept**: Boolean; should the model be fit with an intercept term?
- **offset_col**: Offset column name. If this is not set, we treat all instance offsets as 0.0. The feature specified as offset has a constant coefficient of 1.0.
link_power  Index in the power link function. Only applicable to the Tweedie family. Note that link power 0, 1, -1 or 0.5 corresponds to the Log, Identity, Inverse or Sqrt link, respectively. When not set, this value defaults to 1 - variancePower, which matches the R "statmod" package.

link_prediction_col  Link prediction (linear predictor) column name. Default is not set, which means we do not output link prediction.

reg_param  Regularization parameter (aka lambda)

max_iter  The maximum number of iterations to use.

weight_col  The name of the column to use as weights for the model fit.

solver  Solver algorithm for optimization.

tol  Param for the convergence tolerance for iterative algorithms.

variance_power  Power in the variance function of the Tweedie distribution which provides the relationship between the variance and mean of the distribution. Only applicable to the Tweedie family. (see Tweedie Distribution (Wikipedia)) Supported values: 0 and [1, Inf). Note that variance power 0, 1, or 2 corresponds to the Gaussian, Poisson or Gamma family, respectively.

features_col  Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.

label_col  Label column name. The column should be a numeric column. Usually this column is output by `ft_r_formula`.

prediction_col  Prediction column name.

uid  A character string used to uniquely identify the ML estimator.

...  Optional arguments; see Details.

**Details**

Valid link functions for each family is listed below. The first link function of each family is the default one.

- gaussian: "identity", "log", "inverse"
- binomial: "logit", "probit", "loglog"
- poisson: "log", "identity", "sqrt"
- gamma: "inverse", "identity", "log"
- tweedie: power link function specified through link_power. The default link power in the tweedie family is 1 - variance_power.

**Value**

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.
ml_glm_tidiers

Tidying methods for Spark ML linear models

Description

These methods summarize the results of Spark ML models into tidy forms.

See Also

Other ml algorithms: ml_aft_survival_regression(), ml_decision_tree_classifier(), ml_gbt_classifier(), ml_isotonic_regression(), ml_linear_regression(), ml_linear_svc(), ml_logistic_regression(), ml_multilayer_perceptron_classifier(), ml_naive_bayes(), ml_one_vs_rest(), ml_random_forest_classifier()

Examples

```r
## Not run:
library(sparklyr)

sc <- spark_connect(master = "local")
mtcars_tbl <- sdf_copy_to(sc, mtcars, name = "mtcars_tbl", overwrite = TRUE)

partitions <- mtcars_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

mtcars_training <- partitions$training
mtcars_test <- partitions$test

# Specify the grid
family <- c("gaussian", "gamma", "poisson")
link <- c("identity", "log")
family_link <- expand.grid(family = family, link = link, stringsAsFactors = FALSE)
family_link <- data.frame(family_link, rmse = 0)

# Train the models
for (i in seq_len(nrow(family_link))) {
  glm_model <- mtcars_training %>%
    ml_generalized_linear_regression(mpg ~ .,
    family = family_link[i, 1],
    link = family_link[i, 2]
  )
  pred <- ml_predict(glm_model, mtcars_test)
  family_link[i, 3] <- ml_regression_evaluator(pred, label_col = "mpg")
}

family_link

## End(Not run)
```
Usage

```r
## S3 method for class 'ml_model_generalized_linear_regression'
tidy(x, exponentiate = FALSE, ...)

## S3 method for class 'ml_model_linear_regression'
tidy(x, ...)

## S3 method for class 'ml_model_generalized_linear_regression'
augment(
  x,
  newdata = NULL,
  type.residuals = c("working", "deviance", "pearson", "response"),
  ...)

## S3 method for class '_ml_model_linear_regression'
augment(
  x,
  newdata = NULL,
  type.residuals = c("working", "deviance", "pearson", "response"),
  ...)

## S3 method for class 'ml_model_linear_regression'
augment(
  x,
  newdata = NULL,
  type.residuals = c("working", "deviance", "pearson", "response"),
  ...)

## S3 method for class 'ml_model_generalized_linear_regression'
glance(x, ...)

## S3 method for class 'ml_model_linear_regression'
glance(x, ...)
```

Arguments

- `x` a Spark ML model.
- `exponentiate` For GLM, whether to exponentiate the coefficient estimates (typical for logistic regression.)
- `...` extra arguments (not used.)
- `newdata` a tbl_spark of new data to use for prediction.
- `type.residuals` type of residuals, defaults to "working". Must be set to "working" when `newdata` is supplied.
- `new_data` a tbl_spark of new data to use for prediction.
Details

The residuals attached by augment are of type "working" by default, which is different from the default of "deviance" for residuals() or sdf_residuals().

ml_isotonic_regression

`Spark ML – Isotonic Regression`

Description

Currently implemented using parallelized pool adjacent violators algorithm. Only univariate (single feature) algorithm supported.

Usage

```r
ml_isotonic_regression(
  x,
  formula = NULL,
  feature_index = 0,
  isotonic = TRUE,
  weight_col = NULL,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  uid = random_string("isotonic_regression_"),
  ...
)
```

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **formula**: Used when `x` is a `tbl_spark`. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see `ft_r_formula` for details.
- **feature_index**: Index of the feature if `features_col` is a vector column (default: 0), no effect otherwise.
- **isotonic**: Whether the output sequence should be isotonic/increasing (true) or antitonic/decreasing (false). Default: true
- **weight_col**: The name of the column to use as weights for the model fit.
- **features_col**: Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.
- **label_col**: Label column name. The column should be a numeric column. Usually this column is output by `ft_r_formula`. 
prediction_col  Prediction column name.
uid          A character string used to uniquely identify the ML estimator.
...          Optional arguments; see Details.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a
ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended
to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.

See Also

Other ml algorithms: ml_aft_survival_regression(), ml_decision_tree_classifier(), ml_gbt_classifier(),
ml_generalized_linear_regression(), ml_linear_regression(), ml_linear_svc(), ml_logistic_regression(),
ml_multilayer_perceptron_classifier(), ml_naive_bayes(), ml_one_vs_rest(), ml_random_forest_classifier

Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

partitions <- iris_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

iris_training <- partitions$training
iris_test <- partitions$test

iso_res <- iris_tbl %>%
  ml_isotonic_regression(Petal_Length ~ Petal_Width)
pred <- ml_predict(iso_res, iris_test)
pred

## End(Not run)
```
ml_kmeans

Usage

```r
## S3 method for class 'ml_model_isotonic_regression'
tidy(x, ...)

## S3 method for class 'ml_model_isotonic_regression'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model_isotonic_regression'
glance(x, ...)
```

Arguments

- `x` a Spark ML model.
- `...` extra arguments (not used.)
- `newdata` a tbl_spark of new data to use for prediction.

Description

K-means clustering with support for k-means|| initialization proposed by Bahmani et al. Using `ml_kmeans()` with the formula interface requires Spark 2.0+.

Usage

```r
ml_kmeans(
x,
formula = NULL,
k = 2,
max_iter = 20,
tol = 1e-04,
init_steps = 2,
init_mode = "k-means||",
seed = NULL,
features_col = "features",
prediction_col = "prediction",
uid = random_string("kmeans_"),
...)
```

```r
ml_compute_cost(model, dataset)
```

```r
ml_compute_silhouette_measure(model, dataset, distance_measure = c("squaredEuclidean", "cosine")
```
ml_kmeans

Arguments

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **formula**: Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see \texttt{ft_r_formula} for details.
- **k**: The number of clusters to create.
- **max_iter**: The maximum number of iterations to use.
- **tol**: Param for the convergence tolerance for iterative algorithms.
- **init_steps**: Number of steps for the k-means|| initialization mode. This is an advanced setting – the default of 2 is almost always enough. Must be > 0. Default: 2.
- **init_mode**: Initialization algorithm. This can be either "random" to choose random points as initial cluster centers, or "k-means||" to use a parallel variant of k-means++ (Bahmani et al., Scalable K-Means++, VLDB 2012). Default: k-means||.
- **seed**: A random seed. Set this value if you need your results to be reproducible across repeated calls.
- **features_col**: Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by \texttt{ft_r_formula}.
- **prediction_col**: Prediction column name.
- **uid**: A character string used to uniquely identify the ML estimator.
- **...**: Optional arguments, see Details. #’@return The object returned depends on the class of \texttt{x}. If it is a spark_connection, the function returns a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.
- **model**: A fitted K-means model returned by \texttt{ml_kmeans()}.
- **dataset**: Dataset on which to calculate K-means cost
- **distance_measure**: Distance measure to apply when computing the Silhouette measure.

Value

- \texttt{ml_compute_cost()} returns the K-means cost (sum of squared distances of points to their nearest center) for the model on the given data.
- \texttt{ml_compute_silhouette_measure()} returns the Silhouette measure of the clustering on the given data.

Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)
ml_kmeans(iris_tbl, Species ~ .)
## End(Not run)
```
**ml_kmeans_cluster_eval**

Evaluates a K-mean clustering

**Description**

Evaluate a K-mean clustering

**Arguments**

- `model`: A fitted K-means model returned by `ml_kmeans()`
- `dataset`: Dataset on which to calculate K-means cost

---

**ml_lda**

*Spark ML – Latent Dirichlet Allocation*

**Description**

Latent Dirichlet Allocation (LDA), a topic model designed for text documents.

**Usage**

```r
ml_lda(
  x,
  formula = NULL,
  k = 10,
  max_iter = 20,
  doc_concentration = NULL,
  topic_concentration = NULL,
  subsampling_rate = 0.05,
  optimizer = "online",
  checkpoint_interval = 10,
  keep_last_checkpoint = TRUE,
  learning_decay = 0.51,
  learning_offset = 1024,
  optimize_doc_concentration = TRUE,
  seed = NULL,
  features_col = "features",
  topic_distribution_col = "topicDistribution",
  uid = random_string("lda_"),
  ...
)
```

```r
ml_describe_topics(model, max_terms_per_topic = 10)
```
ml_log_likelihood(model, dataset)

ml_log_perplexity(model, dataset)

ml_topics_matrix(model)

Arguments

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **formula**: Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see `ft_r_formula` for details.
- **k**: The number of clusters to create
- **max_iter**: The maximum number of iterations to use.
- **doc_concentration**: Concentration parameter (commonly named "alpha") for the prior placed on documents' distributions over topics ("theta"). See details.
- **topic_concentration**: Concentration parameter (commonly named "beta" or "eta") for the prior placed on topics' distributions over terms.
- **subsampling_rate**: (For Online optimizer only) Fraction of the corpus to be sampled and used in each iteration of mini-batch gradient descent, in range (0, 1]. Note that this should be adjusted in sync with `max_iter` so the entire corpus is used. Specifically, set both so that `maxIterations * miniBatchFraction` greater than or equal to 1.
- **optimizer**: Optimizer or inference algorithm used to estimate the LDA model. Supported: "online" for Online Variational Bayes (default) and "em" for Expectation-Maximization.
- **checkpoint_interval**: Set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations, defaults to 10.
- **keep_last_checkpoint**: (Spark 2.0.0+) (For EM optimizer only) If using checkpointing, this indicates whether to keep the last checkpoint. If FALSE, then the checkpoint will be deleted. Deleting the checkpoint can cause failures if a data partition is lost, so set this bit with care. Note that checkpoints will be cleaned up via reference counting, regardless.
- **learning_decay**: (For Online optimizer only) Learning rate, set as an exponential decay rate. This should be between (0.5, 1.0] to guarantee asymptotic convergence. This is called "kappa" in the Online LDA paper (Hoffman et al., 2010). Default: 0.51, based on Hoffman et al.
- **learning_offset**: (For Online optimizer only) A (positive) learning parameter that downweights early iterations. Larger values make early iterations count less. This is called "tau0" in the Online LDA paper (Hoffman et al., 2010) Default: 1024, following Hoffman et al.
optimize_doc_concentration
(For Online optimizer only) Indicates whether the doc_concentration (Dirichlet parameter for document-topic distribution) will be optimized during training. Setting this to true will make the model more expressive and fit the training data better. Default: FALSE

seed
A random seed. Set this value if you need your results to be reproducible across repeated calls.

features_col
Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.

topic_distribution_col
Output column with estimates of the topic mixture distribution for each document (often called "theta" in the literature). Returns a vector of zeros for an empty document.

uid
A character string used to uniquely identify the ML estimator.

... Optional arguments, see Details. # @return The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.

model
A fitted LDA model returned by `ml_lda()`.

max_terms_per_topic
Maximum number of terms to collect for each topic. Default value of 10.

dataset
test corpus to use for calculating log likelihood or log perplexity

Details

For `ml_lda.tbl_spark` with the formula interface, you can specify named arguments in `...` that will be passed `ft_regex_tokenizer()`, `ft_stop_words_remover()`, and `ft_count_vectorizer()`. For example, to increase the default `min_token_length`, you can use `ml_lda(dataset, ~ text, min_token_length = 4)`.

Terminology for LDA:

- "term" = "word": an element of the vocabulary
- "token": instance of a term appearing in a document
- "topic": multinomial distribution over terms representing some concept
- "document": one piece of text, corresponding to one row in the input data


Input data (features_col): LDA is given a collection of documents as input data, via the features_col parameter. Each document is specified as a Vector of length vocab_size, where each entry is the count for the corresponding term (word) in the document. Feature transformers such as `ft_tokenizer` and `ft_count_vectorizer` can be useful for converting text to word count vectors
Value

ml_describe_topics returns a DataFrame with topics and their top-weighted terms.
ml_log_likelihood calculates a lower bound on the log likelihood of the entire corpus

Parameter details

doc_concentration: This is the parameter to a Dirichlet distribution, where larger values mean more smoothing (more regularization). If not set by the user, then doc_concentration is set automatically. If set to singleton vector [alpha], then alpha is replicated to a vector of length k in fitting. Otherwise, the doc_concentration vector must be length k. (default = automatic)

Optimizer-specific parameter settings:

EM
- Currently only supports symmetric distributions, so all values in the vector should be the same.
- Values should be greater than 1.0
- default = uniformly (50 / k) + 1, where 50/k is common in LDA libraries and +1 follows from Asuncion et al. (2009), who recommend a +1 adjustment for EM.

Online
- Values should be greater than or equal to 0
- default = uniformly (1.0 / k), following the implementation from here

topic_concentration:
This is the parameter to a symmetric Dirichlet distribution.
Note: The topics’ distributions over terms are called "beta" in the original LDA paper by Blei et al., but are called "phi" in many later papers such as Asuncion et al., 2009.
If not set by the user, then topic_concentration is set automatically. (default = automatic)

Optimizer-specific parameter settings:

EM
- Value should be greater than 1.0
- default = 0.1 + 1, where 0.1 gives a small amount of smoothing and +1 follows Asuncion et al. (2009), who recommend a +1 adjustment for EM.

Online
- Value should be greater than or equal to 0
- default = (1.0 / k), following the implementation from here.

topic_distribution_col: This uses a variational approximation following Hoffman et al. (2010), where the approximate distribution is called "gamma." Technically, this method returns this approximation "gamma" for each document.

Examples

## Not run:
library(janeaustenr)
library(dplyr)
sc <- spark_connect(master = "local")
lines_tbl <- sdf_copy_to(sc, 
austen_books()[c(1:30), ],
  name = "lines_tbl",
  overwrite = TRUE  
)

# transform the data in a tidy form
lines_tbl_tidy <- lines_tbl %>%
  ft_tokenizer(
    input_col = "text",
    output_col = "word_list"
  ) %>%
  ft_stop_words_remover(
    input_col = "word_list",
    output_col = "wo_stop_words"
  ) %>%
  mutate(text = explode(wo_stop_words)) %>%
  filter(text != "") %>%
  select(text, book)

lda_model <- lines_tbl_tidy %>%
  ml_lda(~text, k = 4)

# vocabulary and topics
tidy(lda_model)

## End(Not run)

---

**ml_lda_tidiers**

Tidying methods for Spark ML LDA models

**Description**

These methods summarize the results of Spark ML models into tidy forms.

**Usage**

### S3 method for class 'ml_model lda'

tidy(x, ...)

### S3 method for class 'ml_model lda'

augment(x, newdata = NULL, ...)

### S3 method for class 'ml_model lda'

glance(x, ...)
Arguments

- **x**: A Spark ML model.
- **...**: Extra arguments (not used.)
- **newdata**: A tbl_spark of new data to use for prediction.

### Description

Perform regression using linear regression.

### Usage

```r
ml_linear_regression(
  x,
  formula = NULL,
  fit_intercept = TRUE,
  elastic_net_param = 0,
  reg_param = 0,
  max_iter = 100,
  weight_col = NULL,
  loss = "squaredError",
  solver = "auto",
  standardization = TRUE,
  tol = 1e-06,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  uid = random_string("linear_regression_"),
  ...
)
```

### Arguments

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
- **formula**: Used when `x` is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see `ft_r_formula` for details.
- **fit_intercept**: Boolean; should the model be fit with an intercept term?
- **elastic_net_param**: ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty.
- **reg_param**: Regularization parameter (aka lambda)
- **max_iter**: The maximum number of iterations to use.
weight_col  The name of the column to use as weights for the model fit.
loss        The loss function to be optimized. Supported options: "squaredError" and "huber". Default: "squaredError"
solver      Solver algorithm for optimization.
standardization  Whether to standardize the training features before fitting the model.
tol         Param for the convergence tolerance for iterative algorithms.
features_col Features column name, as a length-one character vector. The column should be a single vector column of numeric values. Usually this column is output by \texttt{ft_r_formula}.
label_col   Label column name. The column should be a numeric column. Usually this column is output by \texttt{ft_r_formula}.
prediction_col Prediction column name.
uid         A character string used to uniquely identify the ML estimator.
...          Optional arguments; see Details.

Value

The object returned depends on the class of \texttt{x}. If it is a \texttt{spark_connection}, the function returns a \texttt{ml_estimator} object. If it is a \texttt{ml_pipeline}, it will return a pipeline with the predictor appended to it. If a \texttt{tbl_spark}, it will return a \texttt{tbl_spark} with the predictions added to it.

See Also

Other ml algorithms: \texttt{ml_aft_survival_regression()}, \texttt{ml_decision_tree_classifier()}, \texttt{ml_gbt_classifier()}, \texttt{ml_generalized_linear_regression()}, \texttt{ml_isotonic_regression()}, \texttt{ml_linear_svc()}, \texttt{ml_logistic_regression()}, \texttt{ml_multilayer_perceptron_classifier()}, \texttt{ml_naive_bayes()}, \texttt{ml_one_vs_rest()}, \texttt{ml_random_forest_classifier()}

Examples

```r
## Not run:
spark_connection <- spark_connect(master = "local")
mtcars_tbl <- sdf_copy_to(sc, mtcars, name = "mtcars_tbl", overwrite = TRUE)
partitions <- mtcars_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)
mtcars_training <- partitions$training
mtcars_test <- partitions$test
lm_model <- mtcars_training %>%
  ml_linear_regression(mpg ~ .)
pred <- ml_predict(lm_model, mtcars_test)
ml_regression_evaluator(pred, label_col = "mpg")
## End(Not run)
```
ml_linear_svc

Spark ML – LinearSVC

Description

Perform classification using linear support vector machines (SVM). This binary classifier optimizes
the Hinge Loss using the OWLQN optimizer. Only supports L2 regularization currently.

Usage

ml_linear_svc(
  x,
  formula = NULL,
  fit_intercept = TRUE,
  reg_param = 0,
  max_iter = 100,
  standardization = TRUE,
  weight_col = NULL,
  tol = 1e-06,
  threshold = 0,
  aggregation_depth = 2,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  raw_prediction_col = "rawPrediction",
  uid = random_string("linear_svc_"),
  ...
)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.
formula Used when x is a tbl_spark. R formula as a character string or a formula.
This is used to transform the input dataframe before fitting, see ft_r_formula for
details.
fit_intercept Boolean; should the model be fit with an intercept term?
reg_param Regularization parameter (aka lambda)
max_iter The maximum number of iterations to use.
standardization Whether to standardize the training features before fitting the model.
weight_col The name of the column to use as weights for the model fit.
tol Param for the convergence tolerance for iterative algorithms.
threshold in binary classification prediction, in range [0, 1].
aggregation_depth (Spark 2.1.0+) Suggested depth for treeAggregate (>= 2).
features_col  Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.

label_col  Label column name. The column should be a numeric column. Usually this column is output by `ft_r_formula`.

prediction_col  Prediction column name.

raw_prediction_col  Raw prediction (a.k.a. confidence) column name.

uid  A character string used to uniquely identify the ML estimator.

...  Optional arguments; see Details.

**Value**

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the predictor appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the predictions added to it.

**See Also**

Other ml algorithms: `ml_aft_survival_regression()`, `ml_decision_tree_classifier()`, `ml_gbt_classifier()`, `ml_generalized_linear_regression()`, `ml_isotonic_regression()`, `ml_linear_regression()`, `ml_logistic_regression()`, `ml_multilayer_perceptron_classifier()`, `ml_naive_bayes()`, `ml_one_vs_rest()`, `ml_random_forest_classifier()`

**Examples**

```r
## Not run:
library(dplyr)

sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

partitions <- iris_tbl %>%
  filter(Species != "setosa")
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

iris_training <- partitions$training
iris_test <- partitions$test

svc_model <- iris_training %>%
  ml_linear_svc(Species ~ .)

pred <- ml_predict(svc_model, iris_test)

ml_binary_classification_evaluator(pred)
## End(Not run)
```
**ml_linear_svc_tidiers**  *Tidying methods for Spark ML linear svc*

---

**Description**
These methods summarize the results of Spark ML models into tidy forms.

**Usage**

```r
## S3 method for class 'ml_model_linear_svc'
tidy(x, ...)

## S3 method for class 'ml_model_linear_svc'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model_linear_svc'
glance(x, ...)
```

**Arguments**

- `x`: a Spark ML model.
- `...`: extra arguments (not used.)
- `newdata`: a tbl_spark of new data to use for prediction.

---

**ml_logistic_regression**  *Spark ML – Logistic Regression*

---

**Description**
Perform classification using logistic regression.

**Usage**

```r
ml_logistic_regression(
x,
formula = NULL,
fit_intercept = TRUE,
elastic_net_param = 0,
reg_param = 0,
max_iter = 100,
threshold = 0.5,
thresholds = NULL,
tol = 1e-06,
weight.col = NULL,
```
aggregation_depth = 2,
lower_bounds_on_coefficients = NULL,
lower_bounds_on_intercepts = NULL,
upper_bounds_on_coefficients = NULL,
upper_bounds_on_intercepts = NULL,
features_col = "features",
label_col = "label",
family = "auto",
prediction_col = "prediction",
probability_col = "probability",
raw_prediction_col = "rawPrediction",
uid = random_string("logistic_regression_"),
...

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.

formula Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see fit_r_formula for details.

fit_intercept Boolean; should the model be fit with an intercept term?

elastic_net_param ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty.

reg_param Regularization parameter (aka lambda)

max_iter The maximum number of iterations to use.

threshold in binary classification prediction, in range [0, 1].

thresholds Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0 excepting that at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the class’s threshold.

tol Param for the convergence tolerance for iterative algorithms.

weight_col The name of the column to use as weights for the model fit.

aggregation_depth (Spark 2.1.0+) Suggested depth for treeAggregate (>= 2).

lower_bounds_on_coefficients (Spark 2.2.0+) Lower bounds on coefficients if fitting under bound constrained optimization. The bound matrix must be compatible with the shape (1, number of features) for binomial regression, or (number of classes, number of features) for multinomial regression.

lower_bounds_on_intercepts (Spark 2.2.0+) Lower bounds on intercepts if fitting under bound constrained optimization. The bounds vector size must be equal with 1 for binomial regression, or the number of classes for multinomial regression.
upper_bounds_on_coefficients
(Spark 2.2.0+) Upper bounds on coefficients if fitting under bound constrained optimization. The bound matrix must be compatible with the shape (1, number of features) for binomial regression, or (number of classes, number of features) for multinomial regression.

upper_bounds_on_intercepts
(Spark 2.2.0+) Upper bounds on intercepts if fitting under bound constrained optimization. The bounds vector size must be equal with 1 for binomial regression, or the number of classes for multinomial regression.

features_col Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by `ft_r_formula`.

label_col Label column name. The column should be a numeric column. Usually this column is output by `ft_r_formula`.

family (Spark 2.1.0+) Param for the name of family which is a description of the label distribution to be used in the model. Supported options: "auto", "binomial", and "multinomial."

prediction_col Prediction column name.

probability_col Column name for predicted class conditional probabilities.

raw_prediction_col Raw prediction (a.k.a. confidence) column name.

uid A character string used to uniquely identify the ML estimator.

... Optional arguments; see Details.

Value
The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.

See Also
Other ml algorithms: `ml_aft_survival_regression()`, `ml_decision_tree_classifier()`, `ml_gbt_classifier()`, `ml_generalized_linear_regression()`, `ml_isotonic_regression()`, `ml_linear_regression()`, `ml_linear_svc()`, `ml_multilayer_perceptron_classifier()`, `ml_naive_bayes()`, `ml_one_vs_rest()`, `ml_random_forest_classifier()`

Examples
## Not run:
sc <- spark_connect(master = "local")
mtcars_tbl <- sdf_copy_to(sc, mtcars, name = "mtcars_tbl", overwrite = TRUE)
partitions <- mtcars_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

mtcars_training <- partitions$training
mtcars_test <- partitions$test

lr_model <- mtcars_training %>%
  ml_logistic_regression(am ~ gear + carb)
pred <- ml_predict(lr_model, mtcars_test)

ml_binary_classification_evaluator(pred)

## End(Not run)

# Tidying methods for Spark ML Logistic Regression

## S3 method for class 'ml_model_logistic_regression'
tidy(x, ...)

## S3 method for class 'ml_model_logistic_regression'
augment(x, newdata = NULL, ...)

## S3 method for class `ml_model_logistic_regression``
augment(x, new_data = NULL, ...)

## S3 method for class 'ml_model_logistic_regression'
glance(x, ...)

### Description

These methods summarize the results of Spark ML models into tidy forms.

### Usage

```r
## S3 method for class 'ml_model_logistic_regression'
tidy(x, ...)

## S3 method for class 'ml_model_logistic_regression'
augment(x, newdata = NULL, ...)

## S3 method for class `ml_model_logistic_regression``
augment(x, new_data = NULL, ...)

## S3 method for class 'ml_model_logistic_regression'
glance(x, ...)
```

### Arguments

- `x` a Spark ML model.
- `...` extra arguments (not used.)
- `newdata` a tbl_spark of new data to use for prediction.
- `new_data` a tbl_spark of new data to use for prediction.
ml_metrics_binary

Extracts metrics from a fitted table

Description

The function works best when passed a ‘tbl_spark’ created by ‘ml_predict()’. The output ‘tbl_spark’ will contain the correct variable types and format that the given Spark model "evaluator" expects.

Usage

```r
ml_metrics_binary(
  x,
  truth = label,
  estimate = rawPrediction,
  metrics = c("roc_auc", "pr_auc"),
  ...
)
```

Arguments

- **x**: A ‘tbl_spark’ containing the estimate (prediction) and the truth (value of what actually happened)
- **truth**: The name of the column from ‘x’ with an integer field containing the binary response (0 or 1). The ‘ml_predict()’ function will create a new field named ‘label’ which contains the expected type and values. ‘truth’ defaults to ‘label’.
- **estimate**: The name of the column from ‘x’ that contains the prediction. Defaults to ‘rawPrediction’, since its type and expected values will match ‘truth’.
- **metrics**: A character vector with the metrics to calculate. For binary models the possible values are: ‘roc_auc’ (Area under the Receiver Operator curve), ‘pr_auc’ (Area under the Precesion Recall curve). Defaults to: ‘roc_auc’, ‘pr_auc’
- **...**: Optional arguments; currently unused.

Details

The ‘ml_metrics’ family of functions implement Spark’s ‘evaluate’ closer to how the ‘yardstick’ package works. The functions expect a table containing the truth and estimate, and return a ‘tibble’ with the results. The ‘tibble’ has the same format and variable names as the output of the ‘yardstick’ functions.

Examples

```r
## Not run:
sc <- spark_connect("local")
tbl_iris <- copy_to(sc, iris)
prep_iris <- tbl_iris %>%
  mutate(is_setosa = ifelse(Species == "setosa", 1, 0))
iris_split <- sdf_random_split(prep_iris, training = 0.5, test = 0.5)
```
model <- ml_logistic_regression(iris_split$training, "is_setosa ~ Sepal_Length")
tbl_predictions <- ml_predict(model, iris_split$test)
ml_metrics_binary(tbl_predictions)

## End(Not run)

### ml_metrics_multiclass

Extracts metrics from a fitted table

#### Description

The function works best when passed a ‘tbl_spark’ created by ‘ml_predict()’. The output ‘tbl_spark’ will contain the correct variable types and format that the given Spark model “evaluator” expects.

#### Usage

```r
ml_metrics_multiclass(
  x,
  truth = label,
  estimate = prediction,
  metrics = c("accuracy"),
  beta = NULL,
  ...
)
```

#### Arguments

- **x**: A ‘tbl_spark’ containing the estimate (prediction) and the truth (value of what actually happened)
- **truth**: The name of the column from ‘x’ with an integer field containing an the indexed value for each outcome. The ‘ml_predict()’ function will create a new field named ‘label’ which contains the expected type and values. ‘truth’ defaults to ‘label’.
- **estimate**: The name of the column from ‘x’ that contains the prediction. Defaults to ‘prediction’, since its type and indexed values will match ‘truth’.
- **metrics**: A character vector with the metrics to calculate. For multiclass models the possible values are: ‘accuracy’, ‘f_meas’ (F-score), ‘recall’ and ‘precision’. This function translates the argument into an acceptable Spark parameter. If no translation is found, then the raw value of the argument is passed to Spark. This makes it possible to request a metric that is not listed here but, depending on version, it is available in Spark. Other metrics form multi-class models are: ‘weightedTruePositiveRate’, ‘weightedFalsePositiveRate’, ‘weightedFMeasure’, ‘truePositiveRateByLabel’, ‘falsePositiveRateByLabel’, ‘precisionByLabel’, ‘recallByLabel’, ‘fMeasureByLabel’, ‘logLoss’, ‘hammingLoss’
- **beta**: Numerical value used for precision and recall. Defaults to NULL, but if the Spark session’s version is 3.0 and above, then NULL is changed to 1, unless something different is supplied in this argument.
- **...**: Optional arguments; currently unused.
Details

The ‘ml_metrics’ family of functions implement Spark’s ‘evaluate’ closer to how the ‘yardstick’ package works. The functions expect a table containing the truth and estimate, and return a ‘tibble’ with the results. The ‘tibble’ has the same format and variable names as the output of the ‘yardstick’ functions.

Examples

```r
## Not run:
sc <- spark_connect("local")
tbl_iris <- copy_to(sc, iris)
iris_split <- sdf_random_split(tbl_iris, training = 0.5, test = 0.5)
model <- ml_random_forest(iris_split$training, "Species ~ .")
tbl_predictions <- ml_predict(model, iris_split$test)

ml_metrics_multiclass(tbl_predictions)

# Request different metrics
ml_metrics_multiclass(tbl_predictions, metrics = c("recall", "precision"))

# Request metrics not translated by the function, but valid in Spark
ml_metrics_multiclass(tbl_predictions, metrics = c("logLoss", "hammingLoss"))

## End(Not run)
```

ml_metrics_regression

Extracts metrics from a fitted table

Description

The function works best when passed a ‘tbl_spark’ created by ‘ml_predict()’. The output ‘tbl_spark’ will contain the correct variable types and format that the given Spark model “evaluator” expects.

Usage

```r
ml_metrics_regression(
  x,
  truth,
  estimate = prediction,
  metrics = c("rmse", "rsq", "mae"),
  ...
)
```

Arguments

- `x`: A ‘tbl_spark’ containing the estimate (prediction) and the truth (value of what actually happened)
truth The name of the column from ‘x’ that contains the value of what actually happened

estimate The name of the column from ‘x’ that contains the prediction. Defaults to ‘prediction’, since it is the default that ‘ml_predict()’ uses.

metrics A character vector with the metrics to calculate. For regression models the possible values are: ‘rmse’ (Root mean squared error), ‘mse’ (Mean squared error), ‘rsq’ (R squared), ‘mae’ (Mean absolute error), and ‘var’ (Explained variance). Defaults to: ‘rmse’, ‘rsq’, ‘mae’

... Optional arguments; currently unused.

Details

The ‘ml_metrics’ family of functions implement Spark’s ‘evaluate’ closer to how the ‘yardstick’ package works. The functions expect a table containing the truth and estimate, and return a ‘tibble’ with the results. The ‘tibble’ has the same format and variable names as the output of the ‘yardstick’ functions.

Examples

```r
## Not run:
sc <- spark_connect("local")
tbl_iris <- copy_to(sc, iris)
iris_split <- sdf_random_split(tbl_iris, training = 0.5, test = 0.5)
training <- iris_split$training
reg_formula <- "Sepal_Length ~ Sepal_Width + Petal_Length + Petal_Width"
model <- ml_generalized_linear_regression(training, reg_formula)
tbl_predictions <- ml_predict(model, iris_split$test)
tbl_predictions %>%
  ml_metrics_regression(Sepal_Length)
## End(Not run)
```

---

**ml_model_data**  
Extracts data associated with a Spark ML model

**Description**

Extracts data associated with a Spark ML model

**Usage**

```r
ml_model_data(object)
```

**Arguments**

- **object**  
  a Spark ML model
ml_multilayer_perceptron_classifier

Spark ML – Multilayer Perceptron

Value

A tbl_spark

Description

Classification model based on the Multilayer Perceptron. Each layer has sigmoid activation function, output layer has softmax.

Usage

```r
ml_multilayer_perceptron_classifier(
  x,
  formula = NULL,
  layers = NULL,
  max_iter = 100,
  step_size = 0.03,
  tol = 1e-06,
  block_size = 128,
  solver = "l-bfgs",
  seed = NULL,
  initial_weights = NULL,
  thresholds = NULL,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  probability_col = "probability",
  raw_prediction_col = "rawPrediction",
  uid = random_string("multilayer_perceptron_classifier_")
)
```

```r
ml_multilayer_perceptron(
  x,
  formula = NULL,
  layers,
  max_iter = 100,
  step_size = 0.03,
  tol = 1e-06,
  block_size = 128,
  solver = "l-bfgs",
  seed = NULL,
  initial_weights = NULL,
)```
features_col = "features",
label_col = "label",
thresholds = NULL,
prediction_col = "prediction",
probability_col = "probability",
raw_prediction_col = "rawPrediction",
uid = random_string("multilayer_perceptron_classifier_"),
response = NULL,
features = NULL,
... )

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.
formula Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see ft_r_formula for details.
layers A numeric vector describing the layers – each element in the vector gives the size of a layer. For example, c(4, 5, 2) would imply three layers, with an input (feature) layer of size 4, an intermediate layer of size 5, and an output (class) layer of size 2.
max_iter The maximum number of iterations to use.
step_size Step size to be used for each iteration of optimization (> 0).
tol Param for the convergence tolerance for iterative algorithms.
block_size Block size for stacking input data in matrices to speed up the computation. Data is stacked within partitions. If block size is more than remaining data in a partition then it is adjusted to the size of this data. Recommended size is between 10 and 1000. Default: 128
solver The solver algorithm for optimization. Supported options: "gd" (minibatch gradient descent) or "l-bfgs". Default: "l-bfgs"
seed A random seed. Set this value if you need your results to be reproducible across repeated calls.
initial_weights The initial weights of the model.
thresholds Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0 excepting that at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the class’s threshold.
features_col Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by ft_r_formula.
label_col Label column name. The column should be a numeric column. Usually this column is output by ft_r_formula.
**prediction_col**  Prediction column name.

**probability_col**  Column name for predicted class conditional probabilities.

**raw_prediction_col**  Raw prediction (a.k.a. confidence) column name.

**uid**  A character string used to uniquely identify the ML estimator.

...  Optional arguments; see Details.

**response**  (Deprecated) The name of the response column (as a length-one character vector.)

**features**  (Deprecated) The name of features (terms) to use for the model fit.

### Details

`ml_multilayer_perceptron()` is an alias for `ml_multilayer_perceptron_classifier()` for backwards compatibility.

### Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the predictor appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the predictions added to it.

### See Also

Other ml algorithms: `ml_aft_survival_regression()`, `ml_decision_tree_classifier()`, `ml_gbt_classifier()`, `ml_generalized_linear_regression()`, `ml_isotonic_regression()`, `ml_linear_regression()`, `ml_linear_svc()`, `ml_logistic_regression()`, `ml_naive_bayes()`, `ml_one_vs_rest()`, `ml_random_forest_classifier()`.

### Examples

```r
## Not run:
sd <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)
partitions <- iris_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)
iris_training <- partitions$training
iris_test <- partitionsitest
mlp_model <- iris_training %>%
  ml_multilayer_perceptron_classifier(Species ~ ., layers = c(4, 3, 3))
pred <- ml_predict(mlp_model, iris_test)
ml_multiclass_classification_evaluator(pred)
## End(Not run)
```
ml_multilayer_perceptron_tidiers

Tidying methods for Spark ML MLP

Description

These methods summarize the results of Spark ML models into tidy forms.

Usage

```r
## S3 method for class 'ml_model_multilayer_perceptron_classification'
tidy(x, ...)

## S3 method for class 'ml_model_multilayer_perceptron_classification'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model_multilayer_perceptron_classification'
glance(x, ...)
```

Arguments

- `x`: a Spark ML model.
- `...`: extra arguments (not used.)
- `newdata`: a tbl_spark of new data to use for prediction.

ml_naive_bayes

Spark ML – Naive-Bayes

Description

Naive Bayes Classifiers. It supports Multinomial NB (see here) which can handle finitely supported discrete data. For example, by converting documents into TF-IDF vectors, it can be used for document classification. By making every vector a binary (0/1) data, it can also be used as Bernoulli NB (see here). The input feature values must be nonnegative.

Usage

```r
ml_naive_bayes(
x,
  formula = NULL,
  model_type = "multinomial",
  smoothing = 1,
  thresholds = NULL,
  weight_col = NULL,
  features_col = "features",
```
ml_naive_bayes

label_col = "label",
prediction_col = "prediction",
probability_col = "probability",
raw_prediction_col = "rawPrediction",
uid = random_string("naive_bayes_"),
...
)

Arguments

x                  A spark_connection, ml_pipeline, or a tbl_spark.
formula           Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see ft_r_formula for details.
model_type        The model type. Supported options: "multinomial" and "bernoulli". (default = multinomial)
smoothing         The (Laplace) smoothing parameter. Defaults to 1.
thresholds        Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0 excepting that at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the class’s threshold.
weight_col        (Spark 2.1.0+) Weight column name. If this is not set or empty, we treat all instance weights as 1.0.
features_col      Features column name, as a length-one character vector. The column should be a single vector column of numeric values. Usually this column is output by ft_r_formula.
label_col         Label column name. The column should be a numeric column. Usually this column is output by ft_r_formula.
prediction_col    Prediction column name.
probability_col   Column name for predicted class conditional probabilities.
raw_prediction_col Raw prediction (a.k.a. confidence) column name.
uid                A character string used to uniquely identify the ML estimator.
...                Optional arguments; see Details.

Value

The object returned depends on the class of x. If it is a spark_connection, the function returns a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.
See Also

Other ml algorithms: `ml_aft_survival_regression()`, `ml_decision_tree_classifier()`, `ml_gbt_classifier()`, `ml_generalized_linear_regression()`, `ml_isotonic_regression()`, `ml_linear_regression()`, `ml_linear_svc()`, `ml_logistic_regression()`, `ml_multilayer_perceptron_classifier()`, `ml_one_vs_rest()`, `ml_random_forest_classifier()`

Examples

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

partitions <- iris_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)

iris_training <- partitions$training
iris_test <- partitions$test

nb_model <- iris_training %>%
  ml_naive_bayes(Species ~ .)
pred <- ml_predict(nb_model, iris_test)
ml_multiclass_classification_evaluator(pred)
## End(Not run)
```

---

**ml_naive_bayes_tidiers**

*Tidying methods for Spark ML Naive Bayes*

### Description

These methods summarize the results of Spark ML models into tidy forms.

### Usage

```r
## S3 method for class 'ml_model_naive_bayes'
tidy(x, ...)
```

```r
## S3 method for class 'ml_model_naive_bayes'
augment(x, newdata = NULL, ...)
```

```r
## S3 method for class 'ml_model_naive_bayes'
glance(x, ...)
```
Arguments

x       a Spark ML model.
...
newdata a tbl_spark of new data to use for prediction.

Description

Reduction of Multiclass Classification to Binary Classification. Performs reduction using one against all strategy. For a multiclass classification with k classes, train k models (one per class). Each example is scored against all k models and the model with highest score is picked to label the example.

Usage

ml_one_vs_rest(
  x,
  formula = NULL,
  classifier = NULL,
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  uid = random_string("one_vs_rest_"),
  ...
)

Arguments

x       A spark_connection, ml_pipeline, or a tbl_spark.
formula Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see ft_r_formula for details.
classifier Object of class ml_estimator. Base binary classifier that we reduce multiclass classification into.
features_col Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by ft_r_formula.
label_col Label column name. The column should be a numeric column. Usually this column is output by ft_r_formula.
prediction_col Prediction column name.
uid       A character string used to uniquely identify the ML estimator.
...       Optional arguments; see Details.
Value

The object returned depends on the class of `x`. If it is a `spark_connection`, the function returns a `ml_estimator` object. If it is a `ml_pipeline`, it will return a pipeline with the predictor appended to it. If a `tbl_spark`, it will return a `tbl_spark` with the predictions added to it.

See Also

Other ml algorithms: `ml_aft_survival_regression()`, `ml_decision_tree_classifier()`, `ml_gbt_classifier()`, `ml_generalized_linear_regression()`, `ml_isotonic_regression()`, `ml_linear_regression()`, `ml_linear_svc()`, `ml_logistic_regression()`, `ml_multilayer_perceptron_classifier()`, `ml_naive_bayes()`, `ml_random_forest_classifier()`
### ml_pipeline

**Spark ML – Pipelines**

#### Description

Create Spark ML Pipelines

#### Usage

```r
ml_pipeline(x, ..., uid = random_string("pipeline_"))
```

#### Arguments

- **x**: Either a `spark_connection` or `ml_pipeline_stage` objects.
- **...**: `ml_pipeline_stage` objects.
- **uid**: A character string used to uniquely identify the ML estimator.

#### Value

When `x` is a `spark_connection`, `ml_pipeline()` returns an empty pipeline object. When `x` is a `ml_pipeline_stage`, `ml_pipeline()` returns an `ml_pipeline` with the stages set to `x` and any transformers or estimators given in `...`.

---

### ml_power_iteration

**Spark ML – Power Iteration Clustering**

#### Description

Power iteration clustering (PIC) is a scalable and efficient algorithm for clustering vertices of a graph given pairwise similarities as edge properties, described in the paper "Power Iteration Clustering" by Frank Lin and William W. Cohen. It computes a pseudo-eigenvector of the normalized affinity matrix of the graph via power iteration and uses it to cluster vertices. `spark.mllib` includes an implementation of PIC using GraphX as its backend. It takes an RDD of `(srcId, dstId, similarity)` tuples and outputs a model with the clustering assignments. The similarities must be nonnegative. PIC assumes that the similarity measure is symmetric. A pair `(srcId, dstId)` regardless of the ordering should appear at most once in the input data. If a pair is missing from input, their similarity is treated as zero.
Usage

ml_power_iteration(
  x,
  k = 4,
  max_iter = 20,
  init_mode = "random",
  src_col = "src",
  dst_col = "dst",
  weight_col = "weight",
  ...
)

Arguments

x A ‘spark_connection’ or a ‘tbl_spark’.
k The number of clusters to create.
max_iter The maximum number of iterations to run.
init_mode This can be either "random", which is the default, to use a random vector as vertex properties, or "degree" to use normalized sum similarities.
src_col Column in the input Spark dataframe containing 0-based indexes of all source vertices in the affinity matrix described in the PIC paper.
dst_col Column in the input Spark dataframe containing 0-based indexes of all destination vertices in the affinity matrix described in the PIC paper.
weight_col Column in the input Spark dataframe containing non-negative edge weights in the affinity matrix described in the PIC paper.
... Optional arguments. Currently unused.

Value

A 2-column R dataframe with columns named "id" and "cluster" describing the resulting cluster assignments

Examples

## Not run:

library(sparklyr)

sc <- spark_connect(master = "local")

r1 <- 1
n1 <- 80L
r2 <- 4
n2 <- 80L

gen_circle <- function(radius, num_pts) {
  # generate evenly distributed points on a circle centered at the origin
  seq(0, num_pts - 1) %>%
lapply(
  function(pt) {
    theta <- 2 * pi * pt / num_pts
    radius * c(cos(theta), sin(theta))
  }
)

guassian_similarity <- function(pt1, pt2) {
  dist2 <- sum((pt2 - pt1)^2)
  exp(-dist2 / 2)
}

gen_pic_data <- function() {
  # generate points on 2 concentric circle centered at the origin and then
  # compute pairwise Gaussian similarity values of all unordered pair of
  # points
  n <- n1 + n2
  pts <- append(gen_circle(r1, n1), gen_circle(r2, n2))
  num_unordered_pairs <- n * (n - 1) / 2

  src <- rep(0L, num_unordered_pairs)
  dst <- rep(0L, num_unordered_pairs)
  sim <- rep(0, num_unordered_pairs)

  idx <- 1
  for (i in seq(2, n)) {
    for (j in seq(i - 1)) {
      src[idx] <- i - 1L
      dst[idx] <- j - 1L
      sim[idx] <- guassian_similarity(pts[i], pts[j])
      idx <- idx + 1
    }
  }

dplyr::tibble(src = src, dst = dst, sim = sim)
}

pic_data <- copy_to(sc, gen_pic_data())

clusters <- ml_power_iteration(
  pic_data,
  src_col = "src", dst_col = "dst", weight_col = "sim", k = 2, max_iter = 40
)
print(clusters)

## End(Not run)
Frequent Pattern Mining – PrefixSpan

Description

PrefixSpan algorithm for mining frequent itemsets.

Usage

ml_prefixspan(
  x,
  seq_col = "sequence",
  min_support = 0.1,
  max_pattern_length = 10,
  max_local_proj_db_size = 3.2e+07,
  uid = random_string("prefixspan_"),
  ...
)

ml_freq_seq_patterns(model)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.
seq_col The name of the sequence column in dataset (defaults to "sequence"). Rows
with nulls in this column are ignored.
min_support The minimum support required to be considered a frequent sequential pattern.
max_pattern_length The maximum length of a frequent sequential pattern. Any frequent pattern
exceeding this length will not be included in the results.
max_local_proj_db_size The maximum number of items allowed in a prefix-projected database before local
iterative processing of the projected database begins. This parameter should
be tuned with respect to the size of your executors.
uid A character string used to uniquely identify the ML estimator.
... Optional arguments; currently unused.
model A Prefix Span model.

Examples

## Not run:
library(sparklyr)
sc <- spark_connect(master = "local", version = "2.4.0")

items_df <- dplyr::tibble(
  seq = list(
items_sdf <- copy_to(sc, items_df, overwrite = TRUE)

prefix_span_model <- ml_prefixspan(
  sc,
  seq_col = "seq",
  min_support = 0.5,
  max_pattern_length = 5,
  max_local_proj_db_size = 3200000
)

frequent_items <- prefix_span_model$frequent_sequential_patterns(items_sdf) %>% collect()

## End(Not run)

---

**ml_random_forest_classifier**

*Spark ML – Random Forest*

**Description**

Perform classification and regression using random forests.

**Usage**

```r
ml_random_forest_classifier(
  x,
  formula = NULL,
  num_trees = 20,
  subsampling_rate = 1,
  max_depth = 5,
  min_instances_per_node = 1,
  feature_subset_strategy = "auto",
  impurity = "gini",
  min_info_gain = 0,
  max_bins = 32,
  seed = NULL,
  thresholds = NULL,
  checkpoint_interval = 10,
  cache_node_ids = FALSE,
  max_memory_in_mb = 256,
  features_col = "features",
)```

label_col = "label",
prediction_col = "prediction",
probability_col = "probability",
raw_prediction_col = "rawPrediction",
uid = random_string("random_forest_classifier_"),
...
)

ml_random_forest(
  x,
  formula = NULL,
  type = c("auto", "regression", "classification"),
  features_col = "features",
  label_col = "label",
  prediction_col = "prediction",
  probability_col = "probability",
  raw_prediction_col = "rawPrediction",
  feature_subset_strategy = "auto",
  impurity = "auto",
  checkpoint_interval = 10,
  max_bins = 32,
  max_depth = 5,
  num_trees = 20,
  min_info_gain = 0,
  min_instances_per_node = 1,
  subsampling_rate = 1,
  seed = NULL,
  thresholds = NULL,
  cache_node_ids = FALSE,
  max_memory_in_mb = 256,
  uid = random_string("random_forest_"),
  response = NULL,
  features = NULL,
...
)

ml_random_forest_regressor(
  x,
  formula = NULL,
  num_trees = 20,
  subsampling_rate = 1,
  max_depth = 5,
  min_instances_per_node = 1,
  feature_subset_strategy = "auto",
  impurity = "variance",
  min_info_gain = 0,
  max_bins = 32,
  seed = NULL,
ml_random_forest_classifier

checkpoint_interval = 10,
cache_node_ids = FALSE,
max_memory_in_mb = 256,
features_col = "features",
label_col = "label",
prediction_col = "prediction",
uid = random_string("random_forest_regressor_"),
...
)

Arguments

x                  A spark_connection, ml_pipeline, or a tbl_spark.
formula            Used when x is a tbl_spark. R formula as a character string or a formula. This is used to transform the input dataframe before fitting, see ft_r_formula for details.
num_trees          Number of trees to train (>= 1). If 1, then no bootstrapping is used. If > 1, then bootstrapping is done.
subsampling_rate   Fraction of the training data used for learning each decision tree, in range (0, 1]. (default = 1.0)
max_depth          Maximum depth of the tree (>= 0); that is, the maximum number of nodes separating any leaves from the root of the tree.
min_instances_per_node Minimum number of instances each child must have after split.
feature_subset_strategy The number of features to consider for splits at each tree node. See details for options.
impurity           Criterion used for information gain calculation. Supported: "entropy" and "gini" (default) for classification and "variance" (default) for regression. For ml_decision_tree, setting "auto" will default to the appropriate criterion based on model type.
min_info_gain      Minimum information gain for a split to be considered at a tree node. Should be >= 0, defaults to 0.
max_bins           The maximum number of bins used for discretizing continuous features and for choosing how to split on features at each node. More bins give higher granularity.
seed               Seed for random numbers.
thresholds         Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0 excepting that at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the class’s threshold.
checkpoint_interval Set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations, defaults to 10.
cache_node_ids  If FALSE, the algorithm will pass trees to executors to match instances with nodes. If TRUE, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Defaults to FALSE.

max_memory_in_mb  Maximum memory in MB allocated to histogram aggregation. If too small, then 1 node will be split per iteration, and its aggregates may exceed this size. Defaults to 256.

features_col  Features column name, as a length-one character vector. The column should be single vector column of numeric values. Usually this column is output by \texttt{ft_r_formula}.

label_col  Label column name. The column should be a numeric column. Usually this column is output by \texttt{ft_r_formula}.

prediction_col  Prediction column name.

probability_col  Column name for predicted class conditional probabilities.

raw_prediction_col  Raw prediction (a.k.a. confidence) column name.

uid  A character string used to uniquely identify the ML estimator.

...  Optional arguments; see Details.

type  The type of model to fit. "regression" treats the response as a continuous variable, while "classification" treats the response as a categorical variable. When "auto" is used, the model type is inferred based on the response variable type – if it is a numeric type, then regression is used; classification otherwise.

response  (Deprecated) The name of the response column (as a length-one character vector.)

features  (Deprecated) The name of features (terms) to use for the model fit.

\textbf{Details}

The supported options for feature_subset_strategy are

- "auto": Choose automatically for task: If num_trees == 1, set to "all". If num_trees > 1 (forest), set to "sqrt" for classification and to "onethird" for regression.
- "all": use all features
- "onethird": use 1/3 of the features
- "sqrt": use use sqrt(number of features)
- "log2": use log2(number of features)
- "n": when n is in the range (0, 1.0], use n * number of features. When n is in the range (1, number of features), use n features. (default = "auto")

\texttt{ml_random_forest} is a wrapper around \texttt{ml_random_forest_regressor.tbl_spark} and \texttt{ml_random_forest_classifier.tbl_spark} and calls the appropriate method based on model type.
**Value**

The object returned depends on the class of \( x \). If it is a spark_connection, the function returns a ml_estimator object. If it is a ml_pipeline, it will return a pipeline with the predictor appended to it. If a tbl_spark, it will return a tbl_spark with the predictions added to it.

**See Also**

Other ml algorithms: \texttt{ml_aft_survival_regression()}, \texttt{ml_decision_tree_classifier()}, \texttt{ml_gbt_classifier()}, \texttt{ml_generalized_linear_regression()}, \texttt{ml_isotonic_regression()}, \texttt{ml_linear_regression()}, \texttt{ml_linear_svc()}, \texttt{ml_logistic_regression()}, \texttt{ml_multilayer_perceptron_classifier()}, \texttt{ml_naive_bayes()}, \texttt{ml_one_vs_rest()}

**Examples**

```r
## Not run:
sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

partitions <- iris_tbl %>%
  sdf_random_split(training = 0.7, test = 0.3, seed = 1111)
iris_training <- partitions$training
iris_test <- partitions$test

rf_model <- iris_training %>%
  ml_random_forest(Species ~ ., type = "classification")
pred <- ml_predict(rf_model, iris_test)
ml_multiclass_classification_evaluator(pred)
## End(Not run)
```

**ml_stage**

*Spark ML – Pipeline stage extraction*

**Description**

Extraction of stages from a Pipeline or PipelineModel object.

**Usage**

```r
ml_stage(x, stage)
ml_stages(x, stages = NULL)
```
Arguments

- `x`: A `ml_pipeline` or `ml_pipeline_model` object
- `stage`: The UID of a stage in the pipeline.
- `stages`: The UIDs of stages in the pipeline as a character vector.

Value

- For `ml_stage()`: The stage specified.
- For `ml_stages()`: A list of stages. If `stages` is not set, the function returns all stages of the pipeline in a list.

---

**ml_summary**

*Spark ML – Extraction of summary metrics*

**Description**

Extracts a metric from the summary object of a Spark ML model.

**Usage**

```
ml_summary(x, metric = NULL, allow_null = FALSE)
```

**Arguments**

- `x`: A Spark ML model that has a summary.
- `metric`: The name of the metric to extract. If not set, returns the summary object.
- `allow_null`: Whether null results are allowed when the metric is not found in the summary.

---

**ml_survival_regression_tidiers**

*Tidying methods for Spark ML Survival Regression*

**Description**

These methods summarize the results of Spark ML models into tidy forms.

**Usage**

```
## S3 method for class 'ml_model_aft_survival_regression'
tidy(x, ...)

## S3 method for class 'ml_model_aft_survival_regression'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model_aft_survival_regression'
glance(x, ...)
```
Arguments

x
... newdata

a Spark ML model.
extra arguments (not used.)
a tbl_spark of new data to use for prediction.

Description

These methods summarize the results of Spark ML models into tidy forms.

Usage

```r
## S3 method for class 'ml_model_decision_tree_classification'
tidy(x, ...)

## S3 method for class 'ml_model_decision_tree_regression'
tidy(x, ...)

## S3 method for class 'ml_model_decision_tree_classification'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model Decision tree_classification'
augment(x, new_data = NULL, ...)

## S3 method for class 'ml_model_decision_tree_regression'
augment(x, new_data = NULL, ...)

## S3 method for class 'ml_model Decision tree_regression'
augment(x, new_data = NULL, ...)

## S3 method for class 'ml_model Decision tree_classification'
aglance(x, ...)

## S3 method for class 'ml_model_Decision tree_classification'
aglance(x, ...)

## S3 method for class 'ml_model_random_forest_classification'
tidy(x, ...)

## S3 method for class 'ml_model_random_forest_regression'
tidy(x, ...)

## S3 method for class 'ml_model_random_forest_classification'
augment(x, newdata = NULL, ...)
```
## Arguments

- **x**
  - a Spark ML model.
- **...**
  - extra arguments (not used.)
- **newdata**
  - a tbl_spark of new data to use for prediction.
- **new_data**
  - a tbl_spark of new data to use for prediction.
**ml_uid**

*Spark ML – UID*

**Description**

Extracts the UID of an ML object.

**Usage**

`ml_uid(x)`

**Arguments**

- **x**: A Spark ML object

---

**ml_unsupervised_tidiers**

*Tidying methods for Spark ML unsupervised models*

**Description**

These methods summarize the results of Spark ML models into tidy forms.

**Usage**

```r
## S3 method for class 'ml_model_kmeans'
tidy(x, ...)

## S3 method for class 'ml_model_kmeans'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model_kmeans'
glance(x, ...)

## S3 method for class 'ml_model_bisecting_kmeans'
tidy(x, ...)

## S3 method for class 'ml_model_bisecting_kmeans'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model_bisecting_kmeans'
glance(x, ...)

## S3 method for class 'ml_model_gaussian_mixture'
tidy(x, ...)
```
## S3 method for class 'ml_model_gaussian_mixture'
augment(x, newdata = NULL, ...)

## S3 method for class 'ml_model_gaussian_mixture'
glance(x, ...)

### Arguments

- **x**: a Spark ML model.
- **...**: extra arguments (not used.)
- **newdata**: a tbl_spark of new data to use for prediction.

---

### mutate

**Mutate**

**Description**

See `mutate` for more details.

---

### na.replace

**Replace Missing Values in Objects**

**Description**

This S3 generic provides an interface for replacing `NA` values within an object.

**Usage**

`na.replace(object, ...)`

**Arguments**

- **object**: An R object.
- **...**: Arguments passed along to implementing methods.

---

### nest

**Nest**

**Description**

See `nest` for more details.
pivot_longer

Description

See `pivot_longer` for more details.

pivot_wider

Description

See `pivot_wider` for more details.

random_string

Description

Generate a random string with a given prefix.

Usage

```r
random_string(prefix = "table")
```

Arguments

- `prefix` A length-one character vector.
reactiveSpark  

*Reactive spark reader*

**Description**

Given a spark object, returns a reactive data source for the contents of the spark object. This function is most useful to read Spark streams.

**Usage**

```r
reactiveSpark(x, intervalMillis = 1000, session = NULL)
```

**Arguments**

- `x`: An object coercable to a Spark DataFrame.
- `intervalMillis`: Approximate number of milliseconds to wait to retrieve updated data frame. This can be a numeric value, or a function that returns a numeric value.
- `session`: The user session to associate this file reader with, or NULL if none. If non-null, the reader will automatically stop when the session ends.

registerDoSpark  

*Register a Parallel Backend*

**Description**

Registers a parallel backend using the `foreach` package.

**Usage**

```r
registerDoSpark(spark_conn, parallelism = NULL, ...)
```

**Arguments**

- `spark_conn`: Spark connection to use.
- `parallelism`: Level of parallelism to use for task execution (if unspecified, then it will take the value of `SparkContext.defaultParallelism()` which by default is the number of cores available to the ‘sparklyr’ application).
- `...`: additional options for sparklyr parallel backend (currently only the only valid option is ‘nocompile’)

**Value**

None
Examples

```r
## Not run:
sc <- spark_connect(master = "local")
registerDoSpark(sc, nocompile = FALSE)
## End(Not run)
```

### Description

Registering an extension package will result in the package being automatically scanned for spark dependencies when a connection to Spark is created.

#### Usage

```r
register_extension(package)
```

#### Arguments

- `package` The package(s) to register.

#### Note

Packages should typically register their extensions in their `.onLoad` hook – this ensures that their extensions are registered when their namespaces are loaded.

### Description

`replace_na` for more details.

### Description

See `right_join` for more details.
sdf-saveload  
Save / Load a Spark DataFrame

Description
Routines for saving and loading Spark DataFrames.

Usage
sdf_save_table(x, name, overwrite = FALSE, append = FALSE)
sdf_load_table(sc, name)
sdf_save_parquet(x, path, overwrite = FALSE, append = FALSE)
sdf_load_parquet(sc, path)

Arguments
x  A spark_connection, ml_pipeline, or a tbl_spark.
name The table name to assign to the saved Spark DataFrame.
overwrite Boolean; overwrite a pre-existing table of the same name?
append Boolean; append to a pre-existing table of the same name?
sc A spark_connection object.
path The path where the Spark DataFrame should be saved.

sdf-transform-methods  
Spark ML – Transform, fit, and predict methods (sdf_ interface)

Description
Deprecated methods for transformation, fit, and prediction. These are mirrors of the corresponding ml-transform-methods.

Usage
sdf_predict(x, model, ...)
sdf_transform(x, transformer, ...)
sdf_fit(x, estimator, ...)
sdf_fit_and_transform(x, estimator, ...)
sdf_along  

Arguments  

- **x**: A tbl_spark.  
- **model**: A ml_transformer or a ml_model object.  
- **...**: Optional arguments passed to the corresponding ml_methods.  
- **transformer**: A ml_transformer object.  
- **estimator**: A ml_estimator object.  

Value  

sdf_predict(), sdf_transform(), and sdf_fit_and_transform() return a transformed dataframe whereas sdf_fit() returns a ml_transformer.

---

**sdf_along**  

*Create DataFrame for along Object*  

Description  

Creates a DataFrame along the given object.

Usage  

sdf_along(sc, along, repartition = NULL, type = c("integer", "integer64"))

Arguments  

- **sc**: The associated Spark connection.  
- **along**: Takes the length from the length of this argument.  
- **repartition**: The number of partitions to use when distributing the data across the Spark cluster.  
- **type**: The data type to use for the index, either "integer" or "integer64".

---

**sdf_bind**  

*Bind multiple Spark DataFrames by row and column*

Description  

sdf_bind_rows() and sdf_bind_cols() are implementation of the common pattern of do.call(rbind, sdfs) or do.call(cbind, sdfs) for binding many Spark DataFrames into one.

Usage  

sdf_bind_rows(..., id = NULL)  

sdf_bind_cols(...)
**Arguments**

... Spark tbls to combine.

Each argument can either be a Spark DataFrame or a list of Spark DataFrames. When row-binding, columns are matched by name, and any missing columns will be filled with NA.

When column-binding, rows are matched by position, so all data frames must have the same number of rows.

**id**

Data frame identifier.

When `id` is supplied, a new column of identifiers is created to link each row to its original Spark DataFrame. The labels are taken from the named arguments to `sdf_bind_rows()`. When a list of Spark DataFrames is supplied, the labels are taken from the names of the list. If no names are found a numeric sequence is used instead.

**Details**

The output of `sdf_bind_rows()` will contain a column if that column appears in any of the inputs.

**Value**

`sdf_bind_rows()` and `sdf_bind_cols()` return `tbl_spark`

---

<table>
<thead>
<tr>
<th>sdf_broadcast</th>
<th>Broadcast hint</th>
</tr>
</thead>
</table>

**Description**

Used to force broadcast hash joins.

**Usage**

`sdf_broadcast(x)`

**Arguments**

`x` A `spark_connection`, `ml_pipeline`, or a `tbl_spark`. 
### sdf_checkpoint

**Checkpoint a Spark DataFrame**

**Description**

Checkpoint a Spark DataFrame

**Usage**

```r
sdf_checkpoint(x, eager = TRUE)
```

**Arguments**

- `x`: an object coercible to a Spark DataFrame
- `eager`: whether to truncate the lineage of the DataFrame

### sdf_coalesce

**Coalesces a Spark DataFrame**

**Description**

Coalesces a Spark DataFrame

**Usage**

```r
sdf_coalesce(x, partitions)
```

**Arguments**

- `x`: A `spark_connection`, `ml_pipeline`, or `tbl_spark`.
- `partitions`: number of partitions
sdf_collect  

**Collect a Spark DataFrame into R.**

**Description**

Collects a Spark dataframe into R.

**Usage**

\[
\text{sdf}\_\text{collect}(\text{object}, \text{impl} = \text{c("row-wise", "row-wise-iter", "column-wise"}), ...)\]

**Arguments**

- **object**: Spark dataframe to collect
- **impl**: Which implementation to use while collecting Spark dataframe - row-wise: fetch the entire dataframe into memory and then process it row-by-row - row-wise-iter: iterate through the dataframe using RDD local iterator, processing one row at a time (hence reducing memory footprint) - column-wise: fetch the entire dataframe into memory and then process it column-by-column  
  
  NOTE: (1) this will not apply to streaming or arrow use cases  
  (2) this parameter will only affect implementation detail, and will not affect result of `sdf_collect`, and should only be set if performance profiling indicates any particular choice will be significantly better than the default choice ("row-wise")

... Additional options.

sdf_copy_to  

**Copy an Object into Spark**

**Description**

Copy an object into Spark, and return an \( R \) object wrapping the copied object (typically, a Spark DataFrame).

**Usage**

\[
\text{sdf}\_\text{copy}_\text{to}(\text{sc, x, name, memory, repartition, overwrite, struct_columns, ...})\]

\[
\text{sdf}\_\text{import}(\text{x, sc, name, memory, repartition, overwrite, struct_columns, ...})\]
Arguments

sc  The associated Spark connection.
x  An R object from which a Spark DataFrame can be generated.
name  The name to assign to the copied table in Spark.
memory  Boolean; should the table be cached into memory?
repartition  The number of partitions to use when distributing the table across the Spark cluster. The default (0) can be used to avoid partitioning.
overwrite  Boolean; overwrite a pre-existing table with the name name if one already exists?
struct_columns  (only supported with Spark 2.4.0 or higher) A list of columns from the source data frame that should be converted to Spark SQL StructType columns. The source columns can contain either json strings or nested lists. All rows within each source column should have identical schemas (because otherwise the conversion result will contain unexpected null values or missing values as Spark currently does not support schema discovery on individual rows within a struct column).

...  Optional arguments, passed to implementing methods.

Advanced Usage

sdf_copy_to is an S3 generic that, by default, dispatches to sdf_import. Package authors that would like to implement sdf_copy_to for a custom object type can accomplish this by implementing the associated method on sdf_import.

See Also

Other Spark data frames: sdf_distinct(), sdf_random_split(), sdf_register(), sdf_sample(), sdf_sort(), sdf_weighted_sample()

Examples

```r
## Not run:
sc <- spark_connect(master = "spark://HOST:PORT")
sdf_copy_to(sc, iris)

## End(Not run)
```
Usage

sdf_crosstab(x, col1, col2)

Arguments

x A Spark DataFrame
col1 The name of the first column. Distinct items will make the first item of each row.
col2 The name of the second column. Distinct items will make the column names of the DataFrame.

Value

A DataFrame containing the contingency table.

sdf_debug_string Debug Info for Spark DataFrame

Description

Prints plan of execution to generate x. This plan will, among other things, show the number of partitions in parenthesis at the far left and indicate stages using indentation.

Usage

sdf_debug_string(x, print = TRUE)

Arguments

x An R object wrapping, or containing, a Spark DataFrame.
print Print debug information?

sdf_describe Compute summary statistics for columns of a data frame

Description

Compute summary statistics for columns of a data frame

Usage

sdf_describe(x, cols = colnames(x))

Arguments

x An object coercible to a Spark DataFrame
cols Columns to compute statistics for, given as a character vector
sdf_dim  Support for Dimension Operations

Description
sdf_dim(), sdf_nrow() and sdf_ncol() provide similar functionality to dim(), nrow() and ncol().

Usage
sdf_dim(x)
sdf_nrow(x)
sdf_ncol(x)

Arguments
x  An object (usually a spark_tbl).

sdf_distinct  Invoke distinct on a Spark DataFrame

Description
Invoke distinct on a Spark DataFrame

Usage
sdf_distinct(x, ..., name)

Arguments
x  A Spark DataFrame.
...  Optional variables to use when determining uniqueness. If there are multiple rows for a given combination of inputs, only the first row will be preserved. If omitted, will use all variables.
name  A name to assign this table. Passed to [sdf_register()].

See Also
Other Spark data frames: sdf_copy_to(), sdf_random_split(), sdf_register(), sdf_sample(), sdf_sort(), sdf_weighted_sample()
sdf_drop_duplicates  Remove duplicates from a Spark DataFrame

Description

Remove duplicates from a Spark DataFrame

Usage

sdf_drop_duplicates(x, cols = NULL)

Arguments

x An object coercible to a Spark DataFrame
cols Subset of Columns to consider, given as a character vector

sdf_expand_grid  Create a Spark dataframe containing all combinations of inputs

Description

Given one or more R vectors/factors or single-column Spark dataframes, perform an expand.grid operation on all of them and store the result in a Spark dataframe

Usage

sdf_expand_grid(
  sc,
  ..., broadcast_vars = NULL,
  memory = TRUE,
  repartition = NULL,
  partition_by = NULL
)

Arguments

sc The associated Spark connection.
... Each input variable can be either a R vector/factor or a Spark dataframe. Unnamed inputs will assume the default names of 'Var1', 'Var2', etc in the result, similar to what 'expand.grid' does for unnamed inputs.
broadcast_vars Indicates which input(s) should be broadcasted to all nodes of the Spark cluster during the join process (default: none).
memory Boolean; whether the resulting Spark dataframe should be cached into memory (default: TRUE)
sdf_from_avro

### Description

Convert column(s) from avro format

### Usage

```r
sdf_from_avro(x, cols)
```

### Arguments

- **x**: An object coercible to a Spark DataFrame
- **cols**: Named list of columns to transform from Avro format plus a valid Avro schema string for each column, where column names are keys and column schema strings are values (e.g., `c(example_primitive_col = "string", example_complex_col = "{"type":"record","name":"person","fields":[{"name":"person_name","type":"string"},{"name":"person_id","type":"long"}]" )

-----

sdf_is_streaming

### Description

Is the given Spark DataFrame a streaming data?

### Usage

```r
sdf_is_streaming(x)
```

### Arguments

- **x**: A spark_connection, ml_pipeline, or a tbl_spark.
sdf_last_index \hspace{1cm} \textit{Returns the last index of a Spark DataFrame}

\textbf{Description}

Returns the last index of a Spark DataFrame. The Spark \texttt{mapPartitionsWithIndex} function is used to iterate through the last nonempty partition of the RDD to find the last record.

\textbf{Usage}

\begin{verbatim}
sdf_last_index(x, id = "id")
\end{verbatim}

\textbf{Arguments}

- \texttt{x} \hspace{1cm} A \texttt{spark_connection}, \texttt{ml_pipeline}, or a \texttt{tbl_spark}.
- \texttt{id} \hspace{1cm} The name of the index column.

\begin{verbatim}
sdf_len(sc, length, repartition = NULL, type = c("integer", "integer64"))
\end{verbatim}

\textbf{Description}

Creates a DataFrame for the given length.

\textbf{Usage}

\begin{verbatim}
sdf_len(sc, length, repartition = NULL, type = c("integer", "integer64"))
\end{verbatim}

\textbf{Arguments}

- \texttt{sc} \hspace{1cm} The associated Spark connection.
- \texttt{length} \hspace{1cm} The desired length of the sequence.
- \texttt{repartition} \hspace{1cm} The number of partitions to use when distributing the data across the Spark cluster.
- \texttt{type} \hspace{1cm} The data type to use for the index, either "integer" or "integer64".
sdf_num_partitions

Gets number of partitions of a Spark DataFrame

Description

Gets number of partitions of a Spark DataFrame

Usage

sdf_num_partitions(x)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.

sdf_partition_sizes

Compute the number of records within each partition of a Spark DataFrame

Description

Compute the number of records within each partition of a Spark DataFrame

Usage

sdf_partition_sizes(x)

Arguments

x A spark_connection, ml_pipeline, or a tbl_spark.

Examples

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "spark://HOST:PORT")
example_sdf <- sdf_len(sc, 100L, repartition = 10L)
example_sdf %>%
sdf_partition_sizes() %>%
print()

## End(Not run)
```
**sdfPersist**

**Persist a Spark DataFrame**

**Description**

Persist a Spark DataFrame, forcing any pending computations and (optionally) serializing the results to disk.

**Usage**

```r
sdf_persist(x, storage.level = "MEMORY_AND_DISK", name = NULL)
```

**Arguments**

- `x`: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `storage.level`: The storage level to be used. Please view the Spark Documentation for information on what storage levels are accepted.
- `name`: A name to assign this table. Passed to `sdf_register()`.

**Details**

Spark DataFrames invoke their operations lazily – pending operations are deferred until their results are actually needed. Persisting a Spark DataFrame effectively `forces` any pending computations, and then persists the generated Spark DataFrame as requested (to memory, to disk, or otherwise).

Users of Spark should be careful to persist the results of any computations which are non-deterministic – otherwise, one might see that the values within a column seem to `change` as new operations are performed on that data set.

**sdfPivot**

**Pivot a Spark DataFrame**

**Description**

Construct a pivot table over a Spark Dataframe, using a syntax similar to that from `reshape2::dcast`.

**Usage**

```r
sdf_pivot(x, formula, fun.aggregate = "count")
```
sdf_project

Project features onto principal components

Description

Project features onto principal components

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **formula**: A two-sided R formula of the form `x_1 + x_2 + ... ~ y_1`. The left-hand side of the formula indicates which variables are used for grouping, and the right-hand side indicates which variable is used for pivoting. Currently, only a single pivot column is supported.
- **fun.aggregate**: How should the grouped dataset be aggregated? Can be a length-one character vector, giving the name of a Spark aggregation function to be called; a named R list mapping column names to an aggregation method, or an R function that is invoked on the grouped dataset.

Examples

```r
## Not run:
library(sparklyr)
library(dplyr)

sc <- spark_connect(master = "local")
iris_tbl <- sdf_copy_to(sc, iris, name = "iris_tbl", overwrite = TRUE)

# aggregating by mean
iris_tbl %>%
  mutate(Petal_Width = ifelse(Petal_Width > 1.5, "High", "Low")) %>%
  sdf_pivot(Petal_Width ~ Species,
            fun.aggregate = list(Petal_Length = "mean"))

# aggregating all observations in a list
iris_tbl %>%
  mutate(Petal_Width = ifelse(Petal_Width > 1.5, "High", "Low")) %>%
  sdf_pivot(Petal_Width ~ Species,
            fun.aggregate = list(Petal_Length = "collect_list"))

## End(Not run)
```
Usage

```r
sdf_project(
  object,
  newdata,
  features = dimnames(object$pc)[[1]],
  feature_prefix = NULL,
  ...
)
```

Arguments

- **object**: A Spark PCA model object
- **newdata**: An object coercible to a Spark DataFrame
- **features**: A vector of names of columns to be projected
- **feature_prefix**: The prefix used in naming the output features
- **...**: Optional arguments; currently unused.

---

`sdf_quantile` *Compute (Approximate) Quantiles with a Spark DataFrame*

Description

Given a numeric column within a Spark DataFrame, compute approximate quantiles.

Usage

```r
sdf_quantile(
  x,
  column,
  probabilities = c(0, 0.25, 0.5, 0.75, 1),
  relative.error = 1e-05,
  weight.column = NULL
)
```

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **column**: The column(s) for which quantiles should be computed. Multiple columns are only supported in Spark 2.0+.
- **probabilities**: A numeric vector of probabilities, for which quantiles should be computed.
- **relative.error**: The maximal possible difference between the actual percentile of a result and its expected percentile (e.g., if `relative.error` is 0.01 and `probabilities` is 0.95, then any value between the 94th and 96th percentile will be considered an acceptable approximation).
If not NULL, then a generalized version of the Greenwald-Khanna algorithm will be run to compute weighted percentiles, with each sample from ‘column’ having a relative weight specified by the corresponding value in ‘weight.column’. The weights can be considered as relative frequencies of sample data points.

### sdf_random_split

**Description**

Partition a Spark DataFrame into multiple groups. This routine is useful for splitting a DataFrame into, for example, training and test datasets.

**Usage**

```r
sdf_random_split(
  x,
  ..., weights = NULL,
  seed = sample(.Machine$integer.max, 1)
)
```

**Arguments**

- `x`: An object coercable to a Spark DataFrame.
- `...`: Named parameters, mapping table names to weights. The weights will be normalized such that they sum to 1.
- `weights`: An alternate mechanism for supplying weights – when specified, this takes precedence over the `...` arguments.
- `seed`: Random seed to use for randomly partitioning the dataset. Set this if you want your partitioning to be reproducible on repeated runs.

**Details**

The sampling weights define the probability that a particular observation will be assigned to a particular partition, not the resulting size of the partition. This implies that partitioning a DataFrame with, for example,

```r
sdf_random_split(x, training = 0.5, test = 0.5)
```

is not guaranteed to produce training and test partitions of equal size.

**Value**

An R list of tbl_sparks.
sdf_rbeta

Generate random samples from a Beta distribution

Description

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a Beta distribution.

Usage

sdf_rbeta(
  sc,
  n,
  shape1,
  shape2,
  num_partitions = NULL,
  seed = NULL,
  output_col = "x"
)

Arguments

sc A Spark connection.
n Sample Size (default: 1000).
shape1 Non-negative parameter (alpha) of the Beta distribution.
**sdf_rbinom**

Generate random samples from a binomial distribution

**Description**

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a binomial distribution.

**Usage**

```r
sdf_rbinom(
  sc,
  n,
  size,
  prob,
  num_partitions = NULL,
  seed = NULL,
  output_col = "x"
)
```

**Arguments**

- `sc` A Spark connection.
- `n` Sample Size (default: 1000).
- `size` Number of trials (zero or more).
- `prob` Probability of success on each trial.
- `num_partitions` Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
- `seed` Random seed (default: a random long integer).
- `output_col` Name of the output column containing sample values (default: "x").

**See Also**

Other Spark statistical routines: `sdf_rbinom()`, `sdf_rcauchy()`, `sdf_rchisq()`, `sdf_rexp()`, `sdf_rgamma()`, `sdf_rgeom()`, `sdf_rhyper()`, `sdf_rlnorm()`, `sdf_rnorm()`, `sdf_rpois()`, `sdf_rt()`, `sdf_runif()`, `sdf_rweibull()`
sdf_rcauchy

See Also

Other Spark statistical routines: sdf_rbeta(), sdf_rcauchy(), sdf_rchisq(), sdf_rexp(), sdf_rgamma(), sdf_rgeom(), sdf_rhyper(), sdf_rlnorm(), sdf_rnorm(), sdf_rpois(), sdf_rt(), sdf_runif(), sdf_rweibull()

sdf_rcauchy

Generate random samples from a Cauchy distribution

Description

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a Cauchy distribution.

Usage

sdf_rcauchy(
  sc,
  n,
  location = 0,
  scale = 1,
  num_partitions = NULL,
  seed = NULL,
  output_col = "x"
)

Arguments

sc A Spark connection.
n Sample Size (default: 1000).
location Location parameter of the distribution.
scale Scale parameter of the distribution.
num_partitions Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
seed Random seed (default: a random long integer).
output_col Name of the output column containing sample values (default: "x").

See Also

Other Spark statistical routines: sdf_rbeta(), sdf_rbinom(), sdf_rchisq(), sdf_rexp(), sdf_rgamma(), sdf_rgeom(), sdf_rhyper(), sdf_rlnorm(), sdf_rnorm(), sdf_rpois(), sdf_rt(), sdf_runif(), sdf_rweibull()
sdf_rchisq

Generate random samples from a chi-squared distribution

Description
Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a chi-squared distribution.

Usage
sdf_rchisq(sc, n, df, num_partitions = NULL, seed = NULL, output_col = "x")

Arguments
- `sc`: A Spark connection.
- `n`: Sample Size (default: 1000).
- `df`: Degrees of freedom (non-negative, but can be non-integer).
- `num_partitions`: Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
- `seed`: Random seed (default: a random long integer).
- `output_col`: Name of the output column containing sample values (default: "x").

See Also
Other Spark statistical routines: sdf_rbeta(), sdf_rbinom(), sdf_rcauchy(), sdf_rexp(), sdf_rgamma(), sdf_rgeometric(), sdf_rhyper(), sdf_rlnorm(), sdf_rnorm(), sdf_rpois(), sdf_rt(), sdf_runiform(), sdf_rweibull()

sdf_read_column
Read a Column from a Spark DataFrame

Description
Read a single column from a Spark DataFrame, and return the contents of that column back to R.

Usage
sdf_read_column(x, column)

Arguments
- `x`: A spark_connection, ml_pipeline, or a tbl_spark.
- `column`: The name of a column within x.

Details
It is expected for this operation to preserve row order.
**sdf_register**

Register a Spark DataFrame

Description

Registers a Spark DataFrame (giving it a table name for the Spark SQL context), and returns a `tbl_spark`.

Usage

`sdf_register(x, name = NULL)`

Arguments

- `x`: A Spark DataFrame.
- `name`: A name to assign this table.

See Also

Other Spark data frames: `sdf_copy_to()`, `sdf_distinct()`, `sdf_random_split()`, `sdf_sample()`, `sdf_sort()`, `sdf_weighted_sample()`

---

**sdf_repartition**

Repartition a Spark DataFrame

Description

Repartition a Spark DataFrame

Usage

`sdf_repartition(x, partitions = NULL, partition_by = NULL)`

Arguments

- `x`: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- `partitions`: number of partitions
- `partition_by`: vector of column names used for partitioning, only supported for Spark 2.0+
sdf_residuals.ml_model_generalized_linear_regression

Model Residuals

Description

This generic method returns a Spark DataFrame with model residuals added as a column to the model training data.

Usage

```r
## S3 method for class 'ml_model_generalized_linear_regression'
sdf_residuals(
  object,
  type = c("deviance", "pearson", "working", "response"),
  ...)
```

```r
## S3 method for class 'ml_model_linear_regression'
sdf_residuals(object, ...)
```

Arguments

- **object**: Spark ML model object.
- **type**: type of residuals which should be returned.
- **...**: additional arguments

sdf_rexp

Generate random samples from an exponential distribution

Description

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from an exponential distribution.

Usage

```r
sdf_rexp(sc, n, rate = 1, num_partitions = NULL, seed = NULL, output_col = "x")
```
sdf_rgamma

Generate random samples from a Gamma distribution

Description

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a Gamma distribution.

Usage

sdf_rgamma(
  sc,
  n,
  shape,
  rate = 1,
  num_partitions = NULL,
  seed = NULL,
  output_col = "x"
)

Arguments

- **sc**: A Spark connection.
- **n**: Sample Size (default: 1000).
- **shape**: Shape parameter (greater than 0) for the Gamma distribution.
- **rate**: Rate parameter (greater than 0) for the Gamma distribution (scale is 1/rate).
- **num_partitions**: Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
- **seed**: Random seed (default: a random long integer).
- **output_col**: Name of the output column containing sample values (default: "x").

See Also

Other Spark statistical routines: sdf_rbeta(), sdf_rbinom(), sdf_rcauchy(), sdf_rchisq(), sdf_rgamma(), sdf_rgeom(), sdf_rhyper(), sdf_rlnorm(), sdf_rnorm(), sdf_rpois(), sdf_rt(), sdf_runif(), sdf_rweibull()
sdf_rgeom

See Also

Other Spark statistical routines: sdf_rbeta(), sdf_rbinom(), sdf_rcauchy(), sdf_rchisq(), sdf_rexp(), sdf_rgeom(), sdf_rhyper(), sdf_rlnorm(), sdf_rnorm(), sdf_rpois(), sdf_rt(), sdf_runif(), sdf_rweibull()

sdf_rgeom

Generate random samples from a geometric distribution

Description

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a geometric distribution.

Usage

sdf_rgeom(sc, n, prob, num_partitions = NULL, seed = NULL, output_col = "x")

Arguments

sc A Spark connection.

n Sample Size (default: 1000).

prob Probability of success in each trial.

num_partitions Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).

seed Random seed (default: a random long integer).

output_col Name of the output column containing sample values (default: "x").

See Also

Other Spark statistical routines: sdf_rbeta(), sdf_rbinom(), sdf_rcauchy(), sdf_rchisq(), sdf_rexp(), sdf_rgamma(), sdf_rhyper(), sdf_rlnorm(), sdf_rnorm(), sdf_rpois(), sdf_rt(), sdf_runif(), sdf_rweibull()
sdf_rhyper

| Generate random samples from a hypergeometric distribution |

Description

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a hypergeometric distribution.

Usage

```r
sdf_rhyper(
  sc,
  nn,
  m,
  n,
  k,
  num_partitions = NULL,
  seed = NULL,
  output_col = "x"
)
```

Arguments

- **sc**: A Spark connection.
- **nn**: Sample Size.
- **m**: The number of successes among the population.
- **n**: The number of failures among the population.
- **k**: The number of draws.
- **num_partitions**: Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
- **seed**: Random seed (default: a random long integer).
- **output_col**: Name of the output column containing sample values (default: "x").

See Also

Other Spark statistical routines: `sdf_rbeta()`, `sdf_rbinom()`, `sdf_rcauchy()`, `sdf_rchisq()`, `sdf_rexp()`, `sdf_rgamma()`, `sdf_rgeom()`, `sdf_rlnorm()`, `sdf_rnorm()`, `sdf_rpois()`, `sdf_rt()`, `sdf_runif()`, `sdf_rweibull()`
**sdf_rlnorm**

*Generate random samples from a log normal distribution*

---

**Description**

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a log normal distribution.

**Usage**

```r
sdf_rlnorm(
  sc,
  n,
  meanlog = 0,
  sdlog = 1,
  num_partitions = NULL,
  seed = NULL,
  output_col = "x"
)
```

**Arguments**

- `sc`: A Spark connection.
- `n`: Sample Size (default: 1000).
- `meanlog`: The mean of the normally distributed natural logarithm of this distribution.
- `sdlog`: The Standard deviation of the normally distributed natural logarithm of this distribution.
- `num_partitions`: Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
- `seed`: Random seed (default: a random long integer).
- `output_col`: Name of the output column containing sample values (default: "x").

**See Also**

Other Spark statistical routines: `sdf_rbeta()`, `sdf_rbinom()`, `sdf_rcauchy()`, `sdf_rchisq()`, `sdf_rexp()`, `sdf_rgamma()`, `sdf_rgeom()`, `sdf_rhyper()`, `sdf_rnorm()`, `sdf_rpois()`, `sdf_rt()`, `sdf_runif()`, `sdf_rweibull()`
sdf_rnorm

Generate random samples from the standard normal distribution

Description

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from the standard normal distribution.

Usage

sdf_rnorm(
    sc,
    n,
    mean = 0,
    sd = 1,
    num_partitions = NULL,
    seed = NULL,
    output_col = "x"
)

Arguments

sc
    A Spark connection.

n
    Sample Size (default: 1000).

mean
    The mean value of the normal distribution.

sd
    The standard deviation of the normal distribution.

num_partitions
    Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).

seed
    Random seed (default: a random long integer).

output_col
    Name of the output column containing sample values (default: "x").

See Also

Other Spark statistical routines: sdf_rbeta(), sdf_rbinom(), sdf_rcauchy(), sdf_rchisq(), sdf_rexp(), sdf_rgamma(), sdf_rgeom(), sdf_rhyper(), sdf_rlnorm(), sdf_rpois(), sdf_rt(), sdf_runif(), sdf_rweibull()
**sdf_rpois**

*Generate random samples from a Poisson distribution*

**Description**

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a Poisson distribution.

**Usage**

`sdf_rpois(sc, n, lambda, num_partitions = NULL, seed = NULL, output_col = "x")`

**Arguments**

- **sc**: A Spark connection.
- **n**: Sample Size (default: 1000).
- **lambda**: Mean, or lambda, of the Poisson distribution.
- **num_partitions**: Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
- **seed**: Random seed (default: a random long integer).
- **output_col**: Name of the output column containing sample values (default: "x").

**See Also**

Other Spark statistical routines: `sdf_rbeta()`, `sdf_rbinom()`, `sdf_rcauchy()`, `sdf_rchisq()`, `sdf_rexp()`, `sdf_rgamma()`, `sdf_rgeom()`, `sdf_rhyper()`, `sdf_rlnorm()`, `sdf_rnorm()`, `sdf_rnorm()`, `sdf_rt()`, `sdf_runif()`, `sdf_rweibull()`

---

**sdf_rt**

*Generate random samples from a t-distribution*

**Description**

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a t-distribution.

**Usage**

`sdf_rt(sc, n, df, num_partitions = NULL, seed = NULL, output_col = "x")`
**sdf_runif**

Generate random samples from the uniform distribution U(0, 1).

**Arguments**

- `sc`: A Spark connection.
- `n`: Sample Size (default: 1000).
- `df`: Degrees of freedom (> 0, maybe non-integer).
- `num_partitions`: Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
- `seed`: Random seed (default: a random long integer).
- `output_col`: Name of the output column containing sample values (default: "x").

**Description**

Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from the uniform distribution U(0, 1).

**Usage**

```r
sdf_runif(
  sc,
  n,
  min = 0,
  max = 1,
  num_partitions = NULL,
  seed = NULL,
  output_col = "x"
)
```

**See Also**

Other Spark statistical routines: `sdf_rbeta()`, `sdf_rbinom()`, `sdf_rcauchy()`, `sdf_rchisq()`, `sdf_rexp()`, `sdf_rgamma()`, `sdf_rgeom()`, `sdf_rhyper()`, `sdf_rlnorm()`, `sdf_rnorm()`, `sdf_rpois()`, `sdf_runif()`, `sdf_rweibull()`
sdf_rweibull

Generate random samples from a Weibull distribution.

Description
Generator method for creating a single-column Spark dataframes comprised of i.i.d. samples from a Weibull distribution.

Usage
sdf_rweibull(sc, n, shape, scale = 1, num_partitions = NULL, seed = NULL, output_col = "x")

Arguments
sc A Spark connection.
n Sample Size (default: 1000).
shape The shape of the Weibull distribution.
scale The scale of the Weibull distribution (default: 1).
num_partitions Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).
seed Random seed (default: a random long integer).
output_col Name of the output column containing sample values (default: "x").

See Also
Other Spark statistical routines: sdf_rbeta(), sdf_rbinom(), sdf_rcauchy(), sdf_rchisq(), sdf_rexp(), sdf_rgamma(), sdf_rgeom(), sdf_rhyper(), sdf_rlnorm(), sdf_rnorm(), sdf_rpois(), sdf_rt(), sdf_rweibull()
### sdf_sample

*Randomly Sample Rows from a Spark DataFrame*

**Description**

Draw a random sample of rows (with or without replacement) from a Spark DataFrame.

**Usage**

```
sdf_sample(x, fraction = 1, replacement = TRUE, seed = NULL)
```

**Arguments**

- **x**: An object coercable to a Spark DataFrame.
- **fraction**: The fraction to sample.
- **replacement**: Boolean; sample with replacement?
- **seed**: An (optional) integer seed.

**See Also**

Other Spark data frames: `sdf_copy_to()`, `sdf_distinct()`, `sdf_random_split()`, `sdf_register()`, `sdf_sort()`, `sdf_weighted_sample()`

### sdf_schema

*Read the Schema of a Spark DataFrame*

**Description**

Read the schema of a Spark DataFrame.

**Usage**

```
sdf_schema(x, expand_nested_cols = FALSE, expand_struct_cols = FALSE)
```

**Arguments**

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **expand_nested_cols**: Whether to expand columns containing nested array of structs (which are usually created by `tidyr::nest` on a Spark data frame).
- **expand_struct_cols**: Whether to expand columns containing structs.
Details
The type column returned gives the string representation of the underlying Spark type for that column; for example, a vector of numeric values would be returned with the type "DoubleType". Please see the Spark Scala API Documentation for information on what types are available and exposed by Spark.

Value
An \texttt{R} list, with each list element describing the name and type of a column.

\begin{verbatim}
\textbf{sdf_separate_column} \hspace{1cm} Separate a Vector Column into Scalar Columns
\end{verbatim}

Description
Given a vector column in a Spark DataFrame, split that into \texttt{n} separate columns, each column made up of the different elements in the column \texttt{column}.

Usage
\begin{verbatim}
sdf_separate_column(x, column, into = NULL)
\end{verbatim}

Arguments
\begin{itemize}
  \item \texttt{x} \hspace{1cm} A \texttt{spark\_connection}, \texttt{ml\_pipeline}, or a \texttt{tbl\_spark}.
  \item \texttt{column} \hspace{1cm} The name of a (vector-typed) column.
  \item \texttt{into} \hspace{1cm} A specification of the columns that should be generated from \texttt{column}. This can either be a vector of column names, or an \texttt{R} list mapping column names to the (1-based) index at which a particular vector element should be extracted.
\end{itemize}

\begin{verbatim}
\textbf{sdf_seq} \hspace{1cm} Create DataFrame for Range
\end{verbatim}

Description
Creates a DataFrame for the given range

Usage
\begin{verbatim}
sdf_seq(
  sc,
  from = 1L,
  to = 1L,
  by = 1L,
  repartition = NULL,
  type = c("integer", "integer64")
)
\end{verbatim}
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc</td>
<td>The associated Spark connection.</td>
</tr>
<tr>
<td>from, to</td>
<td>The start and end to use as a range</td>
</tr>
<tr>
<td>by</td>
<td>The increment of the sequence.</td>
</tr>
<tr>
<td>repartition</td>
<td>The number of partitions to use when distributing the data across the Spark cluster. Defaults to the minimum number of partitions.</td>
</tr>
<tr>
<td>type</td>
<td>The data type to use for the index, either &quot;integer&quot; or &quot;integer64&quot;.</td>
</tr>
</tbody>
</table>

sdf_sort       Sort a Spark DataFrame

Description

Sort a Spark DataFrame by one or more columns, with each column sorted in ascending order.

Usage

sdf_sort(x, columns)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>An object coercable to a Spark DataFrame.</td>
</tr>
<tr>
<td>columns</td>
<td>The column(s) to sort by.</td>
</tr>
</tbody>
</table>

See Also

Other Spark data frames: sdf_copy_to(), sdf_distinct(), sdf_random_split(), sdf_register(), sdf_sample(), sdf_weighted_sample()

sdf_sql       Spark DataFrame from SQL

Description

Defines a Spark DataFrame from a SQL query, useful to create Spark DataFrames without collecting the results immediately.

Usage

sdf_sql(sc, sql)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc</td>
<td>A spark_connection.</td>
</tr>
<tr>
<td>sql</td>
<td>a 'SQL’ query used to generate a Spark DataFrame.</td>
</tr>
</tbody>
</table>
### sdf_to_avro

**Convert column(s) to avro format**

**Description**

Convert column(s) to avro format

**Usage**

`sdf_to_avro(x, cols = colnames(x))`

**Arguments**

- **x** 
  An object coercible to a Spark DataFrame
- **cols** 
  Subset of Columns to convert into avro format

---

### sdf_unnest_longer

**Unnest longer**

**Description**

Expand a struct column or an array column within a Spark dataframe into one or more rows, similar what to `tidyr::unnest_longer` does to an R dataframe. An index column, if included, will be 1-based if 'col' is an array column.

**Usage**

`sdf_unnest_longer(
    data,
    col,
    values_to = NULL,
    indices_to = NULL,
    include_indices = NULL,
    names_repair = "check_unique",
    ptype = list(),
    transform = list()
)`

**Arguments**

- **data** 
  The Spark dataframe to be unnested
- **col** 
  The struct column to extract components from
- **values_to** 
  Name of column to store vector values. Defaults to 'col'.
- **indices_to** 
  A string giving the name of column which will contain the inner names or position (if not named) of the values. Defaults to 'col' with '_id' suffix
include_indices

Whether to include an index column. An index column will be included by
default if 'col' is a struct column. It will also be included if 'indices_to' is not
'NULL'.

names_repair

Strategy for fixing duplicate column names (the semantic will be exactly identi-
cal to that of '.name_repair' option in tibble)

ptype

Optionally, supply an R data frame prototype for the output. Each column of the
unnested result will be casted based on the Spark equivalent of the type of the
column with the same name within 'ptype', e.g., if 'ptype' has a column 'x' of
type 'character', then column 'x' of the unnested result will be casted from its
original SQL type to StringType.

transform

Optionally, a named list of transformation functions applied

Examples

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local", version = "2.4.0")

# unnesting a struct column
sdf <- copy_to(
  sc,
  dplyr::tibble(
    x = 1:3,
    y = list(list(a = 1, b = 2), list(a = 3, b = 4), list(a = 5, b = 6))
  )
)
unnested <- sdf %>% sdf_unnest_longer(y, indices_to = "attr")

# unnesting an array column
sdf <- copy_to(
  sc,
  dplyr::tibble(
    x = 1:3,
    y = list(1:10, 1:5, 1:2)
  )
)
unnested <- sdf %>% sdf_unnest_longer(y, indices_to = "array_idx")
## End(Not run)
```

### sdf_unnest_wider

**Unnest wider**

sdf_unnest_wider

Description
Flatten a struct column within a Spark dataframe into one or more columns, similar what to tidyr::unnest_wider does to an R dataframe

Usage
sdf_unnest_wider(
  data,
  col,
  names_sep = NULL,
  names_repair = "check_unique",
  ptype = list(),
  transform = list()
)

Arguments
data The Spark dataframe to be unnested
col The struct column to extract components from
names_sep If ‘NULL’, the default, the names will be left as is. If a string, the inner and outer names will be pasted together using ‘names_sep’ as the delimiter.
names_repair Strategy for fixing duplicate column names (the semantic will be exactly identical to that of ‘.name_repair’ option in tibble)
ptype Optionally, supply an R data frame prototype for the output. Each column of the unnested result will be casted based on the Spark equivalent of the type of the column with the same name within ‘ptype’, e.g., if ‘ptype’ has a column ‘x’ of type ‘character’, then column ‘x’ of the unnested result will be casted from its original SQL type to StringType.
transform Optionally, a named list of transformation functions applied to each component (e.g., list(‘x = as.character’) to cast column ‘x’ to String).

Examples
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local", version = "2.4.0")
sdf <- copy_to(
  sc,
  dplyr::tibble(
    x = 1:3,
    y = list(list(a = 1, b = 2), list(a = 3, b = 4), list(a = 5, b = 6))
  )
)

# flatten struct column 'y' into two separate columns 'y_a' and 'y_b'
unnested <- sdf %>% sdf_unnest_wider(y, names_sep = "_")
**sdf_weighted_sample**  
**Perform Weighted Random Sampling on a Spark DataFrame**

**Description**

Draw a random sample of rows (with or without replacement) from a Spark DataFrame. If the sampling is done without replacement, then it will be conceptually equivalent to an iterative process such that in each step the probability of adding a row to the sample set is equal to its weight divided by summation of weights of all rows that are not in the sample set yet in that step.

**Usage**

```r
sdf_weighted_sample(x, weight_col, k, replacement = TRUE, seed = NULL)
```

**Arguments**

- `x`: An object coercable to a Spark DataFrame.
- `weight_col`: Name of the weight column.
- `k`: Sample set size.
- `replacement`: Whether to sample with replacement.
- `seed`: An (optional) integer seed.

**See Also**

Other Spark data frames: `sdf_copy_to()`, `sdf_distinct()`, `sdf_random_split()`, `sdf_register()`, `sdf_sample()`, `sdf_sort()`

---

**sdf_with_sequential_id**  
**Add a Sequential ID Column to a Spark DataFrame**

**Description**

Add a sequential ID column to a Spark DataFrame. The Spark `zipWithIndex` function is used to produce these. This differs from `sdf_with_unique_id` in that the IDs generated are independent of partitioning.

**Usage**

```r
sdf_with_sequential_id(x, id = "id", from = 1L)
```
sdf_with_unique_id

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **id**: The name of the column to host the generated IDs.
- **from**: The starting value of the id column.

Description

Add a unique ID column to a Spark DataFrame. The Spark `monotonicallyIncreasingId` function is used to produce these and is guaranteed to produce unique, monotonically increasing ids; however, there is no guarantee that these IDs will be sequential. The table is persisted immediately after the column is generated, to ensure that the column is stable – otherwise, it can differ across new computations.

Usage

```r
sdf_with_unique_id(x, id = "id")
```

Arguments

- **x**: A `spark_connection`, `ml_pipeline`, or a `tbl_spark`.
- **id**: The name of the column to host the generated IDs.

select

Description

See `select` for more details.

separate

Description

See `separate` for more details.
### spark-api

**Access the Spark API**

**Description**

Access the commonly-used Spark objects associated with a Spark instance. These objects provide access to different facets of the Spark API.

**Usage**

```
spark_context(sc)
java_context(sc)
hive_context(sc)
spark_session(sc)
```

**Arguments**

| sc      | A spark_connection. |

**Details**

The [Scala API documentation](#) is useful for discovering what methods are available for each of these objects. Use `invoke` to call methods on these objects.

**Spark Context**

The main entry point for Spark functionality. The Spark Context represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.

**Java Spark Context**

A Java-friendly version of the aforementioned Spark Context.

**Hive Context**

An instance of the Spark SQL execution engine that integrates with data stored in Hive. Configuration for Hive is read from `hive-site.xml` on the classpath.

Starting with Spark >= 2.0.0, the Hive Context class has been deprecated – it is superceded by the Spark Session class, and `hive_context` will return a Spark Session object instead. Note that both classes share a SQL interface, and therefore one can invoke SQL through these objects.
Spark Session

Available since Spark 2.0.0, the Spark Session unifies the Spark Context and Hive Context classes into a single interface. Its use is recommended over the older APIs for code targeting Spark 2.0.0 and above.

---

Description

These routines allow you to manage your connections to Spark.

Call `spark_disconnect()` on each open Spark connection

Usage

```r
spark_connect(
  master,
  spark_home = Sys.getenv("SPARK_HOME"),
  method = c("shell", "livy", "databricks", "test", "qubole", "synapse"),
  app_name = "sparklyr",
  version = NULL,
  config = spark_config(),
  extensions = sparklyr::registered_extensions(),
  packages = NULL,
  scala_version = NULL,
  ...
)
```

```r
spark_connection_is_open(sc)
```

```r
spark_disconnect(sc, ...)
```

```r
spark_disconnect_all(...)
```

```r
spark_submit(
  master,
  file,
  spark_home = Sys.getenv("SPARK_HOME"),
  app_name = "sparklyr",
  version = NULL,
  config = spark_config(),
  extensions = sparklyr::registered_extensions(),
  scala_version = NULL,
  ...
)
```
**Arguments**

- **master**: Spark cluster url to connect to. Use "local" to connect to a local instance of Spark installed via `spark_install`.
- **spark_home**: The path to a Spark installation. Defaults to the path provided by the `SPARK_HOME` environment variable. If `SPARK_HOME` is defined, it will always be used unless the version parameter is specified to force the use of a locally installed version.
- **method**: The method used to connect to Spark. Default connection method is "shell" to connect using `spark-submit`, use "livy" to perform remote connections using HTTP, or "databricks" when using a Databricks clusters.
- **app_name**: The application name to be used while running in the Spark cluster.
- **version**: The version of Spark to use. Required for "local" Spark connections, optional otherwise.
- **config**: Custom configuration for the generated Spark connection. See `spark_config` for details.
- **extensions**: Extension R packages to enable for this connection. By default, all packages enabled through the use of `sparklyr::register_extension` will be passed here.
- **packages**: A list of Spark packages to load. For example, "delta" or "kafka" to enable Delta Lake or Kafka. Also supports full versions like "io.delta:delta-core_2.11:0.4.0". This is similar to adding packages into the `sparklyr.shell.packages` configuration option. Notice that the `version` parameter is used to choose the correct package, otherwise assumes the latest version is being used.
- **scala_version**: Load the sparklyr jar file that is built with the version of Scala specified (this currently only makes sense for Spark 2.4, where sparklyr will by default assume Spark 2.4 on current host is built with Scala 2.11, and therefore `scala_version = '2.12'` is needed if sparklyr is connecting to Spark 2.4 built with Scala 2.12)
- **...**: Additional params to be passed to each `spark_disconnect()` call (e.g., `terminate = TRUE`)
- **sc**: A spark_connection.
- **file**: Path to R source file to submit for batch execution.

**Details**

By default, when using `method = "livy"`, jars are downloaded from GitHub. But an alternative path (local to Livy server or on HDFS or HTTP(s)) to sparklyr JAR can also be specified through the `sparklyr.livy.jar` setting.

**Examples**

```r
conf <- spark_config()
conf$'sparklyr.shell.conf' <- c(
  "spark.executor.extraJavaOptions=-Duser.timezone='UTC'",
  "spark.driver.extraJavaOptions=-Duser.timezone='UTC'",
  "spark.sql.session.timeZone='UTC'"
)
```
sparklyr_get_backend_port

Return the port number of a 'sparklyr' backend.

Description
Retrieve the port number of the 'sparklyr' backend associated with a Spark connection.

Usage
sparklyr_get_backend_port(sc)

Arguments
sc A spark_connection.

Value
The port number of the 'sparklyr' backend associated with sc.

spark_adaptive_query_execution

Retrieves or sets status of Spark AQE

Description
Retrieves or sets whether Spark adaptive query execution is enabled

Usage
spark_adaptive_query_execution(sc, enable = NULL)

Arguments
sc A spark_connection.
enable Whether to enable Spark adaptive query execution. Defaults to NULL to retrieve configuration entries.
See Also

Other Spark runtime configuration: `spark_advisory_shuffle_partition_size()`, `spark_autoBroadcastJoinThreshold()`, `spark_coalesce_initial_num_partitions()`, `spark_coalesce_min_num_partitions()`, `spark_coalesce_shuffle_partitions()`, `spark_session_config()`

---

`spark_advisory_shuffle_partition_size`

*Retrieves or sets advisory size of the shuffle partition*

---

**Description**

Retrieves or sets advisory size in bytes of the shuffle partition during adaptive optimization

**Usage**

```
spark_advisory_shuffle_partition_size(sc, size = NULL)
```

**Arguments**

- `sc`: A `spark_connection`.
- `size`: Advisory size in bytes of the shuffle partition. Defaults to `NULL` to retrieve configuration entries.

**See Also**

Other Spark runtime configuration: `spark_adaptiveQueryExecution()`, `spark_autoBroadcastJoinThreshold()`, `spark_coalesce_initial_num_partitions()`, `spark_coalesce_min_num_partitions()`, `spark_coalesce_shuffle_partitions()`, `spark_session_config()`

---

`spark_apply`

*Apply an R Function in Spark*

---

**Description**

Applies an R function to a Spark object (typically, a Spark DataFrame).
Usage

```
spark_apply(
  x,
  f,
  columns = NULL,
  memory = TRUE,
  group_by = NULL,
  packages = NULL,
  context = NULL,
  name = NULL,
  barrier = NULL,
  fetch_result_as_sdf = TRUE,
  partition_index_param = "",
  arrow_max_records_per_batch = NULL,
  auto_deps = FALSE,
  ...
)
```

Arguments

- **x**: An object (usually a `spark_tbl`) coercable to a Spark DataFrame.
- **f**: A function that transforms a data frame partition into a data frame. The function `f` has signature `f(df, context, group1, group2, ...)` where `df` is a data frame with the data to be processed, `context` is an optional object passed as the `context` parameter and `group1` to `groupN` contain the values of the `group_by` values. When `group_by` is not specified, `f` takes only one argument. Can also be an `rlang` anonymous function. For example, `~ .x + 1` to define an expression that adds one to the given `.x` data frame.
- **columns**: A vector of column names or a named vector of column types for the transformed object. When not specified, a sample of 10 rows is taken to infer out the output columns automatically, to avoid this performance penalty, specify the column types. The sample size is configurable using the `sparklyr.apply.schema.infer` configuration option.
- **memory**: Boolean; should the table be cached into memory?
- **group_by**: Column name used to group by data frame partitions.
- **packages**: Boolean to distribute `.libPaths()` packages to each node, a list of packages to distribute, or a package bundle created with `spark_apply_bundle()`. Defaults to `TRUE` or the `sparklyr.apply.packages` value set in `spark_config()`. For clusters using Yarn cluster mode, packages can point to a package bundle created using `spark_apply_bundle()` and made available as a Spark file using `config$sparklyr.shell.files`. For clusters using Livy, packages can be manually installed on the driver node. For offline clusters where `available.packages()` is not available, manually download the packages database from https://cran.r-project.org/web/packages/packages.rds and set `Sys.setenv(sparklyr.apply.packagesdb = "<path-to-rds>")`. Otherwise, all packages will be used by default.
For clusters where R packages already installed in every worker node, the `spark.r.libpaths` config entry can be set in `spark_config()` to the local packages library. To specify multiple paths collapse them (without spaces) with a comma delimiter (e.g., "/lib/path/one,/lib/path/two").

- **context**: Optional object to be serialized and passed back to `f()`.
- **name**: Optional table name while registering the resulting data frame.
- **barrier**: Optional to support Barrier Execution Mode in the scheduler.
- **fetch_result_as_sdf**: Whether to return the transformed results in a Spark Dataframe (defaults to `TRUE`). When set to `FALSE`, results will be returned as a list of R objects instead. NOTE: `fetch_result_as_sdf` must be set to `FALSE` when the transformation function being applied is returning R objects that cannot be stored in a Spark Dataframe (e.g., complex numbers or any other R data type that does not have an equivalent representation among Spark SQL data types).

- **partition_index_param**: Optional if non-empty, then `f` also receives the index of the partition being processed as a named argument with this name, in addition to all positional argument(s) it will receive.

  NOTE: when `fetch_result_as_sdf` is set to `FALSE`, object returned from the transformation function also must be serializable by the `base::serialize` function in R.

- **arrow_max_records_per_batch**: Maximum size of each Arrow record batch, ignored if Arrow serialization is not enabled.

- **auto_deps**: [Experimental] Whether to infer all required R packages by examining the closure `f()` and only distribute required R and their transitive dependencies to Spark worker nodes (default: `FALSE`). NOTE: this option will only take effect if `packages` is set to `TRUE` or is a character vector of R package names. If `packages` is a character vector of R package names, then both the set of packages specified by `packages` and the set of inferred packages will be distributed to Spark workers.

  ... Optional arguments; currently unused.

### Configuration

`spark_config()` settings can be specified to change the workers environment. For instance, to set additional environment variables to each worker node use the `sparklyr.apply.env.*` config, to launch workers without `--vanilla` use `sparklyr.apply.options.vanilla set to FALSE`, to run a custom script before launching Rscript use `sparklyr.apply.options.rscript.before`.

### Examples

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local[3]")
```
# creates an Spark data frame with 10 elements then multiply times 10 in R
sdf_len(sc, 10) %>% spark_apply(function(df) df * 10)

# using barrier mode
sdf_len(sc, 3, repartition = 3) %>%
  spark_apply(nrow, barrier = TRUE, columns = c(id = "integer")) %>%
  collect()

## End(Not run)

---

### spark_apply_bundle

Create Bundle for Spark Apply

**Description**

Creates a bundle of packages for `spark_apply()`.

**Usage**

```r
spark_apply_bundle(packages = TRUE, base_path = getwd(), session_id = NULL)
```

**Arguments**

- **packages**: List of packages to pack or `TRUE` to pack all.
- **base_path**: Base path used to store the resulting bundle.
- **session_id**: An optional ID string to include in the bundle file name to allow the bundle to be session-specific.

---

### spark_apply_log

Log Writer for Spark Apply

**Description**

Writes data to log under `spark_apply()`.

**Usage**

```r
spark_apply_log(..., level = "INFO")
```

**Arguments**

- **...**: Arguments to write to log.
- **level**: Severity level for this entry; recommended values: `INFO`, `ERROR` or `WARN`.
spark_coalesce_initial_num_partitions

Retrieves or sets initial number of shuffle partitions before coalescing

Description

Retrieves or sets initial number of shuffle partitions before coalescing

Usage

spark_coalesce_initial_num_partitions(sc, num_partitions = NULL)

Arguments

sc A spark_connection.
num_partitions Initial number of shuffle partitions before coalescing. Defaults to NULL to retrieve configuration entries.

spark_auto_broadcast_join_threshold

Retrieves or sets the auto broadcast join threshold

Description

Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this value to -1 broadcasting can be disabled. Note that currently statistics are only supported for Hive Metastore tables where the command ‘ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan’ has been run, and file-based data source tables where the statistics are computed directly on the files of data.

Usage

spark_auto_broadcast_join_threshold(sc, threshold = NULL)

Arguments

sc A spark_connection.
threshold Maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. Defaults to NULL to retrieve configuration entries.

See Also

Other Spark runtime configuration: spark_adaptive_query_execution(), spark_advisory_shuffle_partition_size(), spark_coalesce_initial_num_partitions(), spark_coalesce_min_num_partitions(), spark_coalesce_shuffle_partitions(), spark_session_config()
**spark_coalesce_min_num_partitions**

Retrieves or sets the minimum number of shuffle partitions after coalescing.

**Description**

Retrieves or sets the minimum number of shuffle partitions after coalescing.

**Usage**

```r
spark_coalesce_min_num_partitions(sc, num_partitions = NULL)
```

**Arguments**

- `sc`: A spark_connection.
- `num_partitions`: Minimum number of shuffle partitions after coalescing. Defaults to NULL to retrieve configuration entries.

**See Also**

Other Spark runtime configuration: `spark_adaptive_query_execution()`, `spark_advisory_shuffle_partition_size()`, `spark_auto_broadcast_join_threshold()`, `spark_coalesce_initial_num_partitions()`, `spark_coalesce_shuffle_partitions()`, `spark_session_config()`

---

**spark_coalesce_shuffle_partitions**

Retrieves or sets whether coalescing contiguous shuffle partitions is enabled.

**Description**

Retrieves or sets whether coalescing contiguous shuffle partitions is enabled.

**Usage**

```r
spark_coalesce_shuffle_partitions(sc, enable = NULL)
```
**sparkCompilationSpec**

**Define a Spark Compilation Specification**

**Description**

For use with `compilePackageJars`. The Spark compilation specification is used when compiling Spark extension Java Archives, and defines which versions of Spark, as well as which versions of Scala, should be used for compilation.

**Usage**

```r
sparkCompilationSpec(
  spark_version = NULL,
  spark_home = NULL,
  scalac_path = NULL,
  scala_filter = NULL,
  jar_name = NULL,
  jar_path = NULL,
  jar_dep = NULL,
  embedded_srcs = "embedded_sources.R"
)
```

**Arguments**

- `spark_version` (character): The Spark version to build against. This can be left unset if the path to a suitable Spark home is supplied.
- `spark_home` (character): The path to a Spark home installation. This can be left unset if `spark_version` is supplied; in such a case, `sparklyr` will attempt to discover the associated Spark installation using `spark_home_dir`.
- `scalac_path` (character): The path to the scalac compiler to be used during compilation of your Spark extension. Note that you should ensure the version of scalac selected matches the version of Spark you are compiling against.

**See Also**

Other Spark runtime configuration: `spark_adaptive_query_execution()`, `spark_advisory_shuffle_partition_size()`, `spark_auto_broadcast_join_threshold()`, `spark_coalesce_initial_num_partitions()`, `spark_coalesce_min_num_partitions()`, `sparkSessionConfig()`
spark_config

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scala_filter</td>
<td>An optional R function that can be used to filter which scala files are used during compilation. This can be useful if you have auxiliary files that should only be included with certain versions of Spark.</td>
</tr>
<tr>
<td>jar_name</td>
<td>The name to be assigned to the generated jar.</td>
</tr>
<tr>
<td>jar_path</td>
<td>The path to the jar tool to be used during compilation of your Spark extension.</td>
</tr>
<tr>
<td>jar_dep</td>
<td>An optional list of additional jar dependencies.</td>
</tr>
<tr>
<td>embedded_srcs</td>
<td>Embedded source file(s) under &lt;R package root&gt;/java to be included in the root of the resulting jar file as resources</td>
</tr>
</tbody>
</table>

Details

Most Spark extensions won't need to define their own compilation specification, and can instead rely on the default behavior of compile_package_jars.

---

**Description**

Read Spark Configuration

**Usage**

spark_config(file = "config.yml", use_default = TRUE)

**Arguments**

- **file**: Name of the configuration file
- **use_default**: TRUE to use the built-in defaults provided in this package

**Details**

Read Spark configuration using the **config** package.

**Value**

Named list with configuration data
**Description**

Convenience function to initialize a Kubernetes configuration instead of `spark_config()`, exposes common properties to set in Kubernetes clusters.

**Usage**

```r
spark_config_kubernetes(
  master,
  version = "3.2.3",
  image = "spark:sparklyr",
  driver = random_string("sparklyr-"),
  account = "spark",
  jars = "local:///opt/sparklyr",
  forward = TRUE,
  executors = NULL,
  conf = NULL,
  timeout = 120,
  ports = c(8880, 8881, 4040),
  fix_config = identical(.Platform$OS.type, "windows"),
  ...
)
```

**Arguments**

- `master`  
  Kubernetes url to connect to, found by running `kubectl cluster-info`.

- `version`  
  The version of Spark being used.

- `image`  
  Container image to use to launch Spark and sparklyr. Also known as `spark.kubernetes.container.image`.

- `driver`  
  Name of the driver pod. If not set, the driver pod name is set to "sparklyr" suffixed by id to avoid name conflicts. Also known as `spark.kubernetes.driver.pod.name`.

- `account`  
  Service account that is used when running the driver pod. The driver pod uses this service account when requesting executor pods from the API server. Also known as `spark.kubernetes.authenticate.driver.serviceAccountName`.

- `jars`  
  Path to the sparklyr jars: either, a local path inside the container image with the sparklyr jars copied when the image was created or, a path accessible by the container where the sparklyr jars were copied. You can find a path to the sparklyr jars by running `system.file("java/", package = "sparklyr")`.

- `forward`  
  Should ports used in sparklyr be forwarded automatically through Kubernetes? Default to TRUE which runs `kubectl port-forward` and `pkill kubectl` on disconnection.

- `executors`  
  Number of executors to request while connecting.
spark_config_settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conf</td>
<td>A named list of additional entries to add to sparklyr.shell.conf.</td>
</tr>
<tr>
<td>timeout</td>
<td>Total seconds to wait before giving up on connection.</td>
</tr>
<tr>
<td>ports</td>
<td>Ports to forward using kubectl.</td>
</tr>
<tr>
<td>fix_config</td>
<td>Should the spark-defaults.conf get fixed? TRUE for Windows.</td>
</tr>
<tr>
<td>...</td>
<td>Additional parameters, currently not in use.</td>
</tr>
</tbody>
</table>

spark_config_settings  Retrieve Available Settings

Description

Retrieves available sparklyr settings that can be used in configuration files or spark_config().

Usage

spark_config_settings()

spark_connection  Retrieve the Spark Connection Associated with an R Object

Description

Retrieve the spark_connection associated with an R object.

Usage

spark_connection(x, ...)

Arguments

x  An R object from which a spark_connection can be obtained.
... Optional arguments; currently unused.

spark_connection-class  spark_connection class

Description

spark_connection class
**spark_connection_find**  *Find Spark Connection*

**Description**

Finds an active spark connection in the environment given the connection parameters.

**Usage**

```r
df_connection_find(master = NULL, app_name = NULL, method = NULL)
```

**Arguments**

- **master**: The Spark master parameter.
- **app_name**: The Spark application name.
- **method**: The method used to connect to Spark.

**spark_connect_method**  *Function that negotiates the connection with the Spark back-end*

**Description**

Function that negotiates the connection with the Spark back-end.

**Usage**

```r
df_connect_method(
  x, 
  method, 
  master, 
  spark_home, 
  config, 
  app_name, 
  version, 
  hadoop_version, 
  extensions, 
  scala_version, 
  ...
)
```
Arguments

- **x**: A dummy method object to determine which code to use to connect.
- **method**: The method used to connect to Spark. Default connection method is "shell" to connect using spark-submit, use "livy" to perform remote connections using HTTP, or "databricks" when using a Databricks clusters.
- **master**: Spark cluster url to connect to. Use "local" to connect to a local instance of Spark installed via `spark_install`.
- **spark_home**: The path to a Spark installation. Defaults to the path provided by the `SPARK_HOME` environment variable. If `SPARK_HOME` is defined, it will always be used unless the version parameter is specified to force the use of a locally installed version.
- **config**: Custom configuration for the generated Spark connection. See `spark_config` for details.
- **app_name**: The application name to be used while running in the Spark cluster.
- **version**: The version of Spark to use. Required for "local" Spark connections, optional otherwise.
- **hadoop_version**: Version of Hadoop to use.
- **extensions**: Extension R packages to enable for this connection. By default, all packages enabled through the use of `sparklyr::register_extension` will be passed here.
- **scala_version**: Load the sparklyr jar file that is built with the version of Scala specified (this currently only makes sense for Spark 2.4, where sparklyr will by default assume Spark 2.4 on current host is built with Scala 2.11, and therefore ‘scala_version = '2.12’ is needed if sparklyr is connecting to Spark 2.4 built with Scala 2.12)
- **...**: Additional params to be passed to each ‘spark_disconnect()’ call (e.g., ‘terminate = TRUE’)

---

### spark_context_config

*Runtime configuration interface for the Spark Context.*

#### Description

Retrieves the runtime configuration interface for the Spark Context.

#### Usage

```r
call(spark_context_config(sc))
```

#### Arguments

- **sc**: A `spark_connection`. 
spark_dataframe  Retrieve a Spark DataFrame

Description
This S3 generic is used to access a Spark DataFrame object (as a Java object reference) from an R object.

Usage
spark_dataframe(x, ...)

Arguments
- x: An R object wrapping, or containing, a Spark DataFrame.
- ...: Optional arguments; currently unused.

Value
A `spark_jobj` representing a Java object reference to a Spark DataFrame.

spark_default_compilation_spec  Default Compilation Specification for Spark Extensions

Description
This is the default compilation specification used for Spark extensions, when used with `compile_package_jars`.

Usage
spark_default_compilation_spec(
  pkg = infer_active_package_name(),
  locations = NULL
)

Arguments
- pkg: The package containing Spark extensions to be compiled.
- locations: Additional locations to scan. By default, the directories `/opt/scala` and `/usr/local/scala` will be scanned.
**spark_dependency**

Define a Spark dependency consisting of a set of custom JARs, Spark packages, and customized dbplyr SQL translation env.

**Usage**

```r
spark_dependency(
  jars = NULL,
  packages = NULL,
  initializer = NULL,
  catalog = NULL,
  repositories = NULL,
  dbplyr_sql_variant = NULL,
  ...
)
```

**Arguments**

- **jars** Character vector of full paths to JAR files.
- **packages** Character vector of Spark packages names.
- **initializer** Optional callback function called when initializing a connection.
- **catalog** Optional location where extension JAR files can be downloaded for Livy.
- **repositories** Character vector of Spark package repositories.
- **dbplyr_sql_variant** Customization of dbplyr SQL translation env. Must be a named list of the following form: `list(scalar = list(scalar_fn1 = ..., scalar_fn2 = ..., <etc>), aggregate = list(agg_fn1 = ..., agg_fn2 = ..., <etc>), window = list(wnd_fn1 = ..., wnd_fn2 = ..., <etc>))` See `sql_variant` for details.

**Value**

An object of type ‘spark_dependency’
spark_dependency_fallback

*Fallback to Spark Dependency*

**Description**

Helper function to assist falling back to previous Spark versions.

**Usage**

```r
spark_dependency_fallback(spark_version, supported_versions)
```

**Arguments**

- `spark_version` The Spark version being requested in `spark_dependencies`.
- `supported_versions` The Spark versions that are supported by this extension.

**Value**

A Spark version to use.

---

spark_extension

*Create Spark Extension*

**Description**

Creates an R package ready to be used as an Spark extension.

**Usage**

```r
spark_extension(path)
```

**Arguments**

- `path` Location where the extension will be created.
**spark_home_set**  
*Set the SPARK_HOME environment variable*

**Description**

Set the SPARK_HOME environment variable. This slightly speeds up some operations, including the connection time.

**Usage**

```r
spark_home_set(path = NULL, ...)
```

**Arguments**

- `path`  
  A string containing the path to the installation location of Spark. If NULL, the path to the most latest Spark/Hadoop versions is used.
- `...`  
  Additional parameters not currently used.

**Value**

The function is mostly invoked for the side-effect of setting the SPARK_HOME environment variable. It also returns TRUE if the environment was successfully set, and FALSE otherwise.

**Examples**

```r
## Not run:
# Not run due to side-effects
spark_home_set()

## End(Not run)
```

---

**spark_ide_connection_open**  
*Set of functions to provide integration with the RStudio IDE*

**Description**

Set of functions to provide integration with the RStudio IDE
Usage

spark_ide_connection_open(con, env, connect_call)

spark_ide_connection_closed(con)

spark_ide_connection_updated(con, hint)

spark_ide_connection_actions(con)

spark_ide_objects(con, catalog, schema, name, type)

spark_ide_columns(
    con,
    table = NULL,
    view = NULL,
    catalog = NULL,
    schema = NULL
)

spark_ide_preview(
    con,
    rowLimit,
    table = NULL,
    view = NULL,
    catalog = NULL,
    schema = NULL
)

Arguments

con Valid Spark connection
env R environment of the interactive R session
connect_call R code that can be used to re-connect to the Spark connection
hint Name of the Spark connection that the RStudio IDE can use as reference.
catalog Name of the top level of the requested table or view
schema Name of the second most top level of the requested level or view
name The name of the view or table being requested
type Type of the object being requested, 'view' or 'table'
table Name of the requested table
view Name of the requested view
rowLimit The number of rows to show in the 'Preview' pane of the RStudio IDE

Details

These function are meant for downstream packages, that provide additional backends to 'sparklyr', to override the opening, closing, update, and preview functionality. The arguments are driven
by what the RStudio IDE API expects them to be, so this is the reason why some use ‘type’ to
designated views or tables, and others have one argument for ‘table’, and another for ‘view’.

spark_insert_table  Inserts a Spark DataFrame into a Spark table

Description

Inserts a Spark DataFrame into a Spark table

Usage

spark_insert_table(
  x,
  name,
  mode = NULL,
  overwrite = FALSE,
  options = list(),
  ...
)

Arguments

x  A Spark DataFrame or dplyr operation
name  The name to assign to the newly generated table.
mode  A character element. Specifies the behavior when data or table already exists.
  Supported values include: ‘error’, ‘append’, ‘overwrite’ and ignore. Notice that
  ‘overwrite’ will also change the column structure.
  For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.
overwrite  Boolean; overwrite the table with the given name if it already exists?
options  A list of strings with additional options.
...  Optional arguments; currently unused.

See Also

Other Spark serialization routines: collect_from_rds(), spark_load_table(), spark_read(),
spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(), spark_read_image(),
spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(), spark_read_parquet(),
spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(), spark_write_avro(),
spark_write_binary(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(), spark_write_orc(),
spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()
spark_install

**Description**

Install versions of Spark for use with local Spark connections (i.e. `spark_connect(master = "local")`)

**Usage**

```r
spark_install(
  version = NULL,
  hadoop_version = NULL,
  reset = TRUE,
  logging = "INFO",
  verbose = interactive()
)
```

```r
spark_uninstall(version, hadoop_version)
```

```r
spark_install_dir()
```

```r
spark_install_tar(tarfile)
```

```r
spark_installed_versions()
```

```r
spark_available_versions(
  show_hadoop = FALSE,
  show_minor = FALSE,
  show_future = FALSE
)
```

**Arguments**

- **version**: Version of Spark to install. See `spark_available_versions` for a list of supported versions
- **hadoop_version**: Version of Hadoop to install. See `spark_available_versions` for a list of supported versions
- **reset**: Attempts to reset settings to defaults.
- **logging**: Logging level to configure install. Supported options: "WARN", "INFO"
- **verbose**: Report information as Spark is downloaded / installed
- **tarfile**: Path to TAR file conforming to the pattern spark-###-bin-(hadoop)?### where ### reference spark and hadoop versions respectively.
- **show_hadoop**: Show Hadoop distributions?
- **show_minor**: Show minor Spark versions?
- **show_future**: Should future versions which have not been released be shown?
**spark_integ_test_skip**

*Value*

List with information about the installed version.

---

**Description**

*It lets the package know if it should test a particular functionality or not*  

**Usage**

```
spark_integ_test_skip(sc, test_name)
```

**Arguments**

- `sc` Spark connection
- `test_name` The name of the test

**Details**

It expects a boolean to be returned. If TRUE, the corresponding test will be skipped. If FALSE the test will be conducted.

---

**spark_jobj**

*Retrieve a Spark JVM Object Reference*

---

**Description**

This S3 generic is used for accessing the underlying Java Virtual Machine (JVM) Spark objects associated with R objects. These objects act as references to Spark objects living in the JVM. Methods on these objects can be called with the `invoke` family of functions.

**Usage**

```
spark_jobj(x, ...)
```

**Arguments**

- `x` An R object containing, or wrapping, a `spark_jobj`.  
- `...` Optional arguments; currently unused.

**See Also**

`invoke`, for calling methods on Java object references.
spark_job-class  

Description

spark_job class

spark_last_error  

Description

Surfaces the last error from Spark captured by internal ‘spark_error’ function

Usage

spark_last_error()

spark_load_table  

Description

Reads from a Spark Table into a Spark DataFrame.

Usage

spark_load_table(
  sc,
  name,
  path,
  options = list(),
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE
)
spark_log

View Entries in the Spark Log

Description

View the most recent entries in the Spark log. This can be useful when inspecting output / errors produced by Spark during the invocation of various commands.

Usage

spark_log(sc, n = 100, filter = NULL, ...)

Arguments

sc  
A spark_connection.

n  
The max number of log entries to retrieve. Use NULL to retrieve all entries within the log.

filter  
Character string to filter log entries.

...  
Optional arguments; currently unused.
spark_read

Read file(s) into a Spark DataFrame using a custom reader

Description

Run a custom R function on Spark workers to ingest data from one or more files into a Spark DataFrame, assuming all files follow the same schema.

Usage

spark_read(sc, paths, reader, columns, packages = TRUE, ...)

Arguments

sc  
A spark_connection.

paths  
A character vector of one or more file URIs (e.g., c("hdfs://localhost:9000/file.txt", "hdfs://localhost:9000/file2.txt"))

reader  
A self-contained R function that takes a single file URI as argument and returns the data read from that file as a data frame.

columns  
a named list of column names and column types of the resulting data frame (e.g., list(column_1 = "integer", column_2 = "character")), or a list of column names only if column types should be inferred from the data (e.g., list("column_1", "column_2")), or NULL if column types should be inferred and resulting data frame can have arbitrary column names

packages  
A list of R packages to distribute to Spark workers

...  
Optional arguments; currently unused.

See Also

Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(), spark_read_image(), spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(), spark_read_parquet(), spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(), spark_write_avro(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(), spark_write_orc(), spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()

Examples

### Not run:

library(sparklyr)
sc <- spark_connect(
  master = "yarn",
  spark_home = "/spark/spark-2.4.5-bin-hadoop2.7"
)
spark_read_avro

# This is a contrived example to show reader tasks will be distributed across
# all Spark worker nodes
spark_read(
    sc,
    rep("/dev/null", 10),
    reader = function(path) system("hostname", intern = TRUE),
    columns = c(hostname = "string")
) %>% sdf_collect()

## End(Not run)

---

**spark_read_avro**

*Read Apache Avro data into a Spark DataFrame.*

---

**Description**

Notice this functionality requires the Spark connection `sc` to be instantiated with either an explicitly specified Spark version (i.e., `spark_connect(..., version = <version>, packages = c("avro", <other package(s)>), ...)` or a specific version of Spark avro package to use (e.g., `spark_connect(..., packages = c("org.apache.spark:spark-avro_2.12:3.0.0", <other package(s)>), ...)`).

**Usage**

```r
spark_read_avro(
    sc,
    name = NULL,
    path = name,
    avro_schema = NULL,
    ignore_extension = TRUE,
    repartition = 0,
    memory = TRUE,
    overwrite = TRUE
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc</td>
<td>A <code>spark_connection</code>.</td>
</tr>
<tr>
<td>name</td>
<td>The name to assign to the newly generated table.</td>
</tr>
<tr>
<td>path</td>
<td>The path to the file. Needs to be accessible from the cluster. Supports the “hdfs://”, “s3a://” and “file://” protocols.</td>
</tr>
<tr>
<td>avro_schema</td>
<td>Optional Avro schema in JSON format</td>
</tr>
<tr>
<td>ignore_extension</td>
<td>If enabled, all files with and without .avro extension are loaded (default: TRUE)</td>
</tr>
<tr>
<td>repartition</td>
<td>The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.</td>
</tr>
</tbody>
</table>
Read binary data into a Spark DataFrame.

Description

Read binary files within a directory and convert each file into a record within the resulting Spark dataframe. The output will be a Spark dataframe with the following columns and possibly partition columns:

- path: StringType
- modificationTime: TimestampType
- length: LongType
- content: BinaryType

Usage

spark_read_binary(
    sc,
    name = NULL,
    dir = name,
    path_glob_filter = "*",
    recursive_file_lookup = FALSE,
    repartition = 0,
    memory = TRUE,
    overwrite = TRUE
)

Arguments

- sc: A spark_connection.
- name: The name to assign to the newly generated table.
- dir: Directory to read binary files from.
spark_read_csv

Read a CSV file into a Spark DataFrame

Description

Read a tabular data file into a Spark DataFrame.

Usage

```r
spark_read_csv(
  sc,
  name = NULL,
  path = name,
  header = TRUE,
  columns = NULL,
  infer_schema = is.null(columns),
  delimiter = ",",
  quote = "\"",
  escape = "\\",
  charset = "UTF-8",
  null_value = NULL,
  options = list(),
  repartition = 0,
)```

path_glob_filter

Glob pattern of binary files to be loaded (e.g., "*.jpg").

recursive_file_lookup

If FALSE (default), then partition discovery will be enabled (i.e., if a partition naming scheme is present, then partitions specified by subdirectory names such as "date=2019-07-01" will be created and files outside subdirectories following a partition naming scheme will be ignored). If TRUE, then all nested directories will be searched even if their names do not follow a partition naming scheme.

repartition

The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.

memory

Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)

overwrite

Boolean; overwrite the table with the given name if it already exists?

See Also

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_source()`, `spark_write_table()`,
spark_read_csv

memory = TRUE,
overwrite = TRUE,

...)

Arguments

sc
A spark_connection.
name
The name to assign to the newly generated table.
path
The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
header
Boolean; should the first row of data be used as a header? Defaults to TRUE.
columns
A vector of column names or a named vector of column types. If specified, the elements can be "binary" for BinaryType, "boolean" for BooleanType, "byte" for Byte Type, "integer" for IntegerType, "integer64" for LongType, "double" for DoubleType, "character" for StringType, "timestamp" for TimestampType and "date" for DateType.
infer_schema
Boolean; should column types be automatically inferred? Requires one extra pass over the data. Defaults to is.null(columns).
delimiter
The character used to delimit each column. Defaults to ",", ".
quote
The character used as a quote. Defaults to ",", ".
escape
The character used to escape other characters. Defaults to ",", "\".
charset
The character set. Defaults to "UTF-8".
null_value
The character to use for null, or missing, values. Defaults to NULL.
options
A list of strings with additional options.
repartition
The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
memory
Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
overwrite
Boolean; overwrite the table with the given name if it already exists?
...
Optional arguments; currently unused.

Details

You can read data from HDFS (hdfs://), S3 (s3a://), as well as the local file system (file://).
When header is FALSE, the column names are generated with a V prefix; e.g. V1, V2, ....

See Also

Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(),
spark_read(), spark_read_avro(), spark_read_binary(), spark_read_delta(), spark_read_image(),
spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(), spark_read_parquet(),
spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(), spark_write_avro(),
spark_write_binary(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(), spark_write_orc(),
spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()
spark_read_delta

Description
Read from Delta Lake into a Spark DataFrame.

Usage
spark_read_delta(
  sc,
  path,
  name = NULL,
  version = NULL,
  timestamp = NULL,
  options = list(),
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE,
  ...
)

Arguments
- **sc**: A spark_connection.
- **path**: The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
- **name**: The name to assign to the newly generated table.
- **version**: The version of the delta table to read.
- **timestamp**: The timestamp of the delta table to read. For example, "2019-01-01" or "2019-01-01'T'00:00:00.000Z".
- **options**: A list of strings with additional options.
- **repartition**: The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
- **memory**: Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
- **overwrite**: Boolean; overwrite the table with the given name if it already exists?
- **...**: Optional arguments; currently unused.

See Also
Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(), spark_read(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_image(), spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(), spark_read_parquet(), spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(), spark_write_avro(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(), spark_write_orc(), spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()
spark_read_image

Read image data into a Spark DataFrame.

Description

Read image files within a directory and convert each file into a record within the resulting Spark dataframe. The output will be a Spark dataframe consisting of struct types containing the following attributes:

- origin: StringType
- height: IntegerType
- width: IntegerType
- nChannels: IntegerType
- mode: IntegerType
- data: BinaryType

Usage

```r
spark_read_image(
  sc,
  name = NULL,
  dir = name,
  drop_invalid = TRUE,
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE
)
```

Arguments

- `sc` A spark_connection.
- `name` The name to assign to the newly generated table.
- `dir` Directory to read binary files from.
- `drop_invalid` Whether to drop files that are not valid images from the result (default: TRUE).
- `repartition` The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
- `memory` Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
- `overwrite` Boolean; overwrite the table with the given name if it already exists?
See Also

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_source()`, `spark_write_table()`, `spark_write_text()`

---

**spark_read_jdbc**

Read from JDBC connection into a Spark DataFrame.

**Description**

Read from JDBC connection into a Spark DataFrame.

**Usage**

```
spark_read_jdbc(
  sc,
  name,
  options = list(),
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE,
  columns = NULL,
  ...
)
```

**Arguments**

- **sc**: A `spark_connection`.
- **name**: The name to assign to the newly generated table.
- **options**: A list of strings with additional options. See [https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration](https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration).
- **repartition**: The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
- **memory**: Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
- **overwrite**: Boolean; overwrite the table with the given name if it already exists?
- **columns**: A vector of column names or a named vector of column types. If specified, the elements can be "binary" for `BinaryType`, "boolean" for `BooleanType`, "byte" for `ByteType`, "integer" for `IntegerType`, "integer64" for `LongType`, "double" for `DoubleType`, "character" for `StringType`, "timestamp" for `TimestampType` and "date" for `DateType`.
- **...**: Optional arguments; currently unused.
spark_read_json

Read a JSON file into a Spark DataFrame

Description

Read a table serialized in the JavaScript Object Notation format into a Spark DataFrame.

Usage

spark_read_json(
  sc,
  name = NULL,
  path = name,
  options = list(),
  repartition = 0,
  memory = TRUE,
)
spark_read_libsvm

```
overwrite = TRUE,
columns = NULL,
...
```

**Arguments**

- `sc` A `spark_connection`.
- `name` The name to assign to the newly generated table.
- `path` The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
- `options` A list of strings with additional options.
- `repartition` The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
- `memory` Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
- `overwrite` Boolean; overwrite the table with the given name if it already exists?
- `columns` A vector of column names or a named vector of column types. If specified, the elements can be "binary" for BinaryType, "boolean" for BooleanType, "byte" for ByteType, "integer" for IntegerType, "integer64" for LongType, "double" for DoubleType, "character" for StringType, "timestamp" for TimestampType and "date" for DateType.
- `...` Optional arguments; currently unused.

**Details**

You can read data from HDFS (hdfs://), S3 (s3a://), as well as the local file system (file://).

**See Also**

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_source()`, `spark_write_table()`, `spark_write_text()`

---

**spark_read_libsvm**  
Read libsvm file into a Spark DataFrame.

**Description**

Read libsvm file into a Spark DataFrame.
spark_read_orc

Usage

spark_read_libsvm(
  sc,
  name = NULL,
  path = name,
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE,
  options = list(),
  ...
)

Arguments

sc  A spark_connection.
name  The name to assign to the newly generated table.
path  The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
repartition  The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
memory  Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
overwrite  Boolean; overwrite the table with the given name if it already exists?
options  A list of strings with additional options.
...  Optional arguments; currently unused.

See Also

Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(),
spark_read(),spark_read_avro(),spark_read_binary(),spark_read_csv(),spark_read_delta(),
spark_read_image(),spark_read_jdbc(),spark_read_json(),spark_read_orc(),spark_read_parquet(),
spark_read_source(),spark_read_table(),spark_read_text(),spark_save_table(),spark_write_avro(),
spark_write_csv(),spark_write_delta(),spark_write_jdbc(),spark_write_json(),spark_write_orc(),
spark_write_parquet(),spark_write_source(),spark_write_table(),spark_write_text()

spark_read_orc  Read a ORC file into a Spark DataFrame

Description

Read a ORC file into a Spark DataFrame.
spark_read_orc

Usage

spark_read_orc(
  sc,
  name = NULL,
  path = name,
  options = list(),
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE,
  columns = NULL,
  schema = NULL,
  ...
)

Arguments

sc  A spark_connection.
name  The name to assign to the newly generated table.
path  The path to the file. Needs to be accessible from the cluster. Supports the
       "hdfs://", "s3a://" and "file://" protocols.
options  A list of strings with additional options. See https://spark.apache.org/
         docs/latest/sql-programming-guide.html#configuration.
repartition  The number of partitions used to distribute the generated table. Use 0 (the de-
             fault) to avoid partitioning.
memory  Boolean; should the data be loaded eagerly into memory? (That is, should the
table be cached?)
overwrite  Boolean; overwrite the table with the given name if it already exists?
columns  A vector of column names or a named vector of column types. If specified,
          the elements can be "binary" for BinaryType, "boolean" for BooleanType,
          "byte" for ByteType, "integer" for IntegerType, "integer64" for LongType,
          "double" for DoubleType, "character" for StringType, "timestamp" for
          TimestampType and "date" for DateType.
schema  A (java) read schema. Useful for optimizing read operation on nested data.
         ...
         Optional arguments; currently unused.

Details

You can read data from HDFS (hdfs://), S3 (s3a://), as well as the local file system (file://).

See Also

Other Spark serialization routines: collect_from_rds(), spark_insert_table(),
spark_load_table(),
spark_read(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(),
spark_read_image(), spark_read_jdbc(), spark_read_json(),
spark_read_libsvm(),
spark_read_parquet(),
spark_read_source(),
spark_read_table(), spark_read_text(),
spark_save_table(),
spark_write_avro(),
spark_write_binary(),
spark_write_image(),
spark_write_parquet(),
spark_write_json(),
spark_write_orc(),
spark_write_source(),
spark_write_table(),
spark_write_text()
spark_read_parquet  

Read a Parquet file into a Spark DataFrame

Description
Read a Parquet file into a Spark DataFrame.

Usage
spark_read_parquet(
  sc,
  name = NULL,
  path = name,
  options = list(),
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE,
  columns = NULL,
  schema = NULL,
  ...
)

Arguments
- **sc**: A spark_connection.
- **name**: The name to assign to the newly generated table.
- **path**: The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
- **options**: A list of strings with additional options. See https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration.
- **repartition**: The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
- **memory**: Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
- **overwrite**: Boolean; overwrite the table with the given name if it already exists?
- **columns**: A vector of column names or a named vector of column types. If specified, the elements can be "binary" for BinaryType, "boolean" for BooleanType, "byte" for ByteType, "integer" for IntegerType, "integer64" for LongType, "double" for DoubleType, "character" for StringType, "timestamp" for TimestampType and "date" for DateType.
- **schema**: A (java) read schema. Useful for optimizing read operation on nested data.
- **...**: Optional arguments; currently unused.
You can read data from HDFS (hdfs://), S3 (s3a://), as well as the local file system (file://).

See Also
Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_source()`, `spark_write_table()`, `spark_write_text()`

Description
Read from a generic source into a Spark DataFrame.

Usage
```
spark_read_source(
  sc,
  name = NULL,
  path = name,
  source,
  options = list(),
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE,
  columns = NULL,
  ...
)
```

Arguments
- `sc` A `spark_connection`.
- `name` The name to assign to the newly generated table.
- `path` The path to the file. Needs to be accessible from the cluster. Supports the “hdfs://”, “s3a://” and “file://” protocols.
- `source` A data source capable of reading data.
- `options` A list of strings with additional options. See https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration.
- `repartition` The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
memory  Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)
overwrite Boolean; overwrite the table with the given name if it already exists?
columns A vector of column names or a named vector of column types. If specified, the elements can be "binary" for BinaryType, "boolean" for BooleanType, "byte" for ByteType, "integer" for IntegerType, "integer64" for LongType, "double" for DoubleType, "character" for StringType, "timestamp" for TimestampType and "date" for DateType.

See Also
Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(),
spark_read(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(),
spark_read_image(), spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(),
spark_read_parquet(), spark_read_table(), spark_read_text(), spark_save_table(), spark_write_avro(),
spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(), spark_write_orc(),
spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()

spark_read_table  Reads from a Spark Table into a Spark DataFrame.

Description
Reads from a Spark Table into a Spark DataFrame.

Usage

spark_read_table(
  sc,
  name,
  options = list(),
  repartition = 0,
  memory = TRUE,
  columns = NULL,
  ...
)

Arguments

sc A spark_connection.
name The name to assign to the newly generated table.
options A list of strings with additional options. See https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration.
repartition The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.
Read a Text file into a Spark DataFrame

Description

Read a Text file into a Spark DataFrame

Usage

spark_read_text(
  sc,
  name = NULL,
  path = name,
  repartition = 0,
  memory = TRUE,
  overwrite = TRUE,
  options = list(),
  whole = FALSE,
  ...
)

Arguments

sc A spark_connection.
name The name to assign to the newly generated table.
path The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
spar**k**\_save\_table

The number of partitions used to distribute the generated table. Use 0 (the default) to avoid partitioning.

Boolean; should the data be loaded eagerly into memory? (That is, should the table be cached?)

Boolean; overwrite the table with the given name if it already exists?

A list of strings with additional options.

Read the entire text file as a single entry? Defaults to FALSE.

Optional arguments; currently unused.

Details

You can read data from HDFS (hdfs://), S3 (s3a://), as well as the local file system (file://).

See Also

Other Spark serialization routines: collect\_from\_rds(), spark\_insert\_table(), spark\_load\_table(), spark\_read\(), spark\_read\_avro\(), spark\_read\_binary\(), spark\_read\_csv\(), spark\_read\_delta\(), spark\_read\_image\(), spark\_read\_jdbc\(), spark\_read\_json\(), spark\_read\_libsvm\(), spark\_read\_orc\(), spark\_read\_parquet\(), spark\_read\_source\(), spark\_read\_table\(), spark\_save\_table\(), spark\_write\_avro\(), spark\_write\_csv\(), spark\_write\_delta\(), spark\_write\_jdbc\(), spark\_write\_json\(), spark\_write\_orc\(), spark\_write\_parquet\(), spark\_write\_source\(), spark\_write\_table\(), spark\_write\_text\()

**spark\_save\_table**  
Saves a Spark DataFrame as a Spark table

Description

Saves a Spark DataFrame and as a Spark table.

Usage

`spark\_save\_table(x, path, mode = NULL, options = list())`

Arguments

- **x**: A Spark DataFrame or dplyr operation
- **path**: The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://", and "file://" protocols.
- **mode**: A character element. Specifies the behavior when data or table already exists. Supported values include: ‘error’, ‘append’, ‘overwrite’ and ignore. Notice that ‘overwrite’ will also change the column structure. For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.
- **options**: A list of strings with additional options.
spark_session_config

Runtime configuration interface for the Spark Session

Description

Retrieves or sets runtime configuration entries for the Spark Session.

Usage

```r
spark_session_config(sc, config = TRUE, value = NULL)
```

Arguments

- **sc**: A `spark_connection`.
- **config**: The configuration entry name(s) (e.g., "spark.sql.shuffle.partitions"). Defaults to NULL to retrieve all configuration entries.
- **value**: The configuration value to be set. Defaults to NULL to retrieve configuration entries.

See Also

Other Spark runtime configuration:

- `spark_adaptive_query_execution`
- `spark_advisory_shuffle_partition_size`
- `spark_auto_broadcast_join_threshold`
- `spark_coalesce_initial_num_partitions`
- `spark_coalesce_min_num_partitions`
- `spark_coalesce_shuffle_partitions`

spark_statistical_routines

Generate random samples from some distribution

Description

Generator methods for creating single-column Spark dataframes comprised of i.i.d. samples from some distribution.
spark_version

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc</td>
<td>A Spark connection.</td>
</tr>
<tr>
<td>n</td>
<td>Sample Size (default: 1000).</td>
</tr>
<tr>
<td>num_partitions</td>
<td>Number of partitions in the resulting Spark dataframe (default: default parallelism of the Spark cluster).</td>
</tr>
<tr>
<td>seed</td>
<td>Random seed (default: a random long integer).</td>
</tr>
<tr>
<td>output_col</td>
<td>Name of the output column containing sample values (default: &quot;x&quot;).</td>
</tr>
</tbody>
</table>

spark_table_name

Generate a Table Name from Expression

Description

Attempts to generate a table name from an expression; otherwise, assigns an auto-generated generic name with "sparklyr_" prefix.

Usage

spark_table_name(expr)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>The expression to attempt to use as name</td>
</tr>
</tbody>
</table>

spark_version

Get the Spark Version Associated with a Spark Connection

Description

Retrieve the version of Spark associated with a Spark connection.

Usage

spark_version(sc)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc</td>
<td>A spark_connection.</td>
</tr>
</tbody>
</table>

Details

Suffixes for e.g. preview versions, or snapshotted versions, are trimmed – if you require the full Spark version, you can retrieve it with invoke(spark_context(sc), "version").

Value

The Spark version as a numeric_version.
spark_version_from_home

Get the Spark Version Associated with a Spark Installation

Description

Retrieve the version of Spark associated with a Spark installation.

Usage

spark_version_from_home(spark_home, default = NULL)

Arguments

spark_home The path to a Spark installation.
default The default version to be inferred, in case version lookup failed, e.g. no Spark installation was found at spark_home.

spark_web

Open the Spark web interface

Description

Open the Spark web interface

Usage

spark_web(sc, ...)

Arguments

sc A spark_connection.
... Optional arguments; currently unused.
\section*{spark_write

Write Spark DataFrame to file using a custom writer

\subsection*{Description}

Run a custom R function on Spark worker to write a Spark DataFrame into file(s). If Spark’s speculative execution feature is enabled (i.e., ‘spark.speculation’ is true), then each write task may be executed more than once and the user-defined writer function will need to ensure no concurrent writes happen to the same file path (e.g., by appending UUID to each file name).

\subsection*{Usage}

\begin{verbatim}
spark_write(x, writer, paths, packages = NULL)
\end{verbatim}

\subsection*{Arguments}

\begin{itemize}
\item \textbf{x} \hspace{1cm} A Spark Dataframe to be saved into file(s)
\item \textbf{writer} \hspace{1cm} A writer function with the signature \texttt{function(partition, path)} where \texttt{partition} is a R dataframe containing all rows from one partition of the original Spark DataFrame \texttt{x} and \texttt{path} is a string specifying the file to write partition to
\item \textbf{paths} \hspace{1cm} A single destination path or a list of destination paths, each one specifying a location for a partition from \texttt{x} to be written to. If number of partition(s) in \texttt{x} is not equal to \texttt{length(paths)} then \texttt{x} will be re-partitioned to contain \texttt{length(paths)} partition(s)
\item \textbf{packages} \hspace{1cm} Boolean to distribute \texttt{.libPaths()} packages to each node, a list of packages to distribute, or a package bundle created with
\end{itemize}

\subsection*{Examples}

\begin{verbatim}
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local[3]"

# copy some test data into a Spark Dataframe
sdf <- sdf_copy_to(sc, iris, overwrite = TRUE)

# create a writer function
writer <- function(df, path) {
  write.csv(df, path)
}

spark_write(
  sdf,
  writer,
  # re-partition sdf into 3 partitions and write them to 3 separate files
  paths = list("file:///tmp/file1", "file:///tmp/file2", "file:///tmp/file3"),
\end{verbatim}
spark_write_avro

) spark_write(
    sdf,
    writer,
    # save all rows into a single file
    paths = list("file:///tmp/all_rows")
)

## End(Not run)

---

**spark_write_avro**

Serialize a Spark DataFrame into Apache Avro format

---

**Description**

Notice this functionality requires the Spark connection *sc* to be instantiated with either an explicitly specified Spark version (i.e., `spark_connect(..., version = <version>, packages = c("avro", <other package(s)>), ...)`) or a specific version of Spark avro package to use (e.g., `spark_connect(..., packages = c("org.apache.spark:spark-avro_2.12:3.0.0", <other package(s)>), ...)`).

**Usage**

```r
spark_write_avro(
  x,
  path,
  avro_schema = NULL,
  record_name = "topLevelRecord",
  record_namespace = "",
  compression = "snappy",
  partition_by = NULL
)
```

**Arguments**

- **x**: A Spark DataFrame or dplyr operation
- **path**: The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
- **avro_schema**: Optional Avro schema in JSON format
- **record_name**: Optional top level record name in write result (default: "topLevelRecord")
- **record_namespace**: Record namespace in write result (default: ")"
- **compression**: Compression codec to use (default: "snappy")
- **partition_by**: A character vector. Partitions the output by the given columns on the file system.
spark_write_csv

Describe

Write a Spark DataFrame to a CSV.

Usage

spark_write_csv(
  x,
  path,
  header = TRUE,
  delimiter = ",",
  quote = "\\","\\",
  escape = "\\",
  charset = "UTF-8",
  null_value = NULL,
  options = list(),
  mode = NULL,
  partition_by = NULL,
  ...
)

Arguments

x A Spark DataFrame or dplyr operation
path The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
header Should the first row of data be used as a header? Defaults to TRUE.
delimiter The character used to delimit each column, defaults to ".
quote The character used as a quote. Defaults to "\"".
escape The character used to escape other characters, defaults to \
charset The character set, defaults to "UTF-8".
null_value The character to use for default values, defaults to NULL.
options A list of strings with additional options.
spark_write_delta

spark_write_delta is a function in the Spark R interface that allows you to write a Spark DataFrame into Delta Lake.

**Description**

Writes a Spark DataFrame into Delta Lake.

**Usage**

spark_write_delta(
  x,
  path,
  mode = NULL,
  options = list(),
  partition_by = NULL,
  ...
)

**Arguments**

- **x**: A Spark DataFrame or dplyr operation
- **path**: The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
- **mode**: A character element. Specifies the behavior when data or table already exists. Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that 'overwrite' will also change the column structure. For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.

**See Also**

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_source()`, `spark_write_table()`, `spark_write_text()`
**spark_write_jdbc**

**Description**

Writes a Spark DataFrame into a JDBC table

**Usage**

```r
spark_write_jdbc(
  x,  # A Spark DataFrame or dplyr operation
  name,  # The name to assign to the newly generated table.
  mode = NULL,  # A character element. Specifies the behavior when data or table already exists. Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that 'overwrite' will also change the column structure. For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.
  options = list(),  # A list of strings with additional options.
  partition_by = NULL,  # A character vector. Partitions the output by the given columns on the file system.
  ...  # Optional arguments; currently unused.
)
```

**Arguments**

- `x`: A Spark DataFrame or dplyr operation
- `name`: The name to assign to the newly generated table.
- `mode`: A character element. Specifies the behavior when data or table already exists. Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that 'overwrite' will also change the column structure. For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.
- `options`: A list of strings with additional options.
- `partition_by`: A character vector. Partitions the output by the given columns on the file system.
- `...`: Optional arguments; currently unused.
spark_write_json

See Also

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_jdbc()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_source()`, `spark_write_table()`, `spark_write_text()`

Examples

```r
## Not run:
sc <- spark_connect(
  master = "local",
  config = list(
    'sparklyr.shell.driver-class-path' = "/usr/share/java/mysql-connector-java-8.0.25.jar"
  )
)
spark_write_jdbc(
  sdf_len(sc, 10),
  name = "my_sql_table",
  options = list(
    url = "jdbc:mysql://localhost:3306/my_sql_schema",
    driver = "com.mysql.jdbc.Driver",
    user = "me",
    password = "******",
    dbtable = "my_sql_table"
  )
)
## End(Not run)
```

spark_write_json

Write a Spark DataFrame to a JSON file

Description

Serialize a Spark DataFrame to the JavaScript Object Notation format.

Usage

```r
spark_write_json(
  x,  
  path,  
  mode = NULL,  
  options = list(),  
  partition_by = NULL,  
  ...  
)
```
spark_write_orc

Write a Spark DataFrame to an ORC file

spark_write_orc(x, path, mode = NULL, options = list(), partition_by = NULL, ...)

Arguments

x A Spark DataFrame or dplyr operation
path The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
mode A character element. Specifies the behavior when data or table already exists. Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that 'overwrite' will also change the column structure.
For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.
options A list of strings with additional options.
partition_by A character vector. Partitions the output by the given columns on the file system.
... Optional arguments; currently unused.

See Also

Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(), spark_read(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(),
spark_read_image(), spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(),
spark_read_parquet(), spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(),
spark_write_avro(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_orc(),
spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()
spark_write_parquet

Arguments

x
A Spark DataFrame or dplyr operation

path
The path to the file. Needs to be accessible from the cluster. Supports the
"hdfs://", "s3a://" and "file://" protocols.

mode
A character element. Specifies the behavior when data or table already exists.
Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that
'overwrite' will also change the column structure.
For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.

options
A list of strings with additional options. See https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration.

partition_by
A character vector. Partitions the output by the given columns on the file
system.

... Optional arguments; currently unused.

See Also

Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(),
spark_read(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(),
spark_read_image(), spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(),
spark_read_parquet(), spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(),
spark_write_avro(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(),
spark_write_parquet(), spark_write_source(), spark_write_table(), spark_write_text()

spark_write_parquet  Write a Spark DataFrame to a Parquet file

Description

Serialize a Spark DataFrame to the Parquet format.

Usage

spark_write_parquet(
  x,
  path,
  mode = NULL,
  options = list(),
  partition_by = NULL,
  ...
)
### spark_write_rds

**Write Spark DataFrame to RDS files**

#### Description

Write Spark dataframe to RDS files. Each partition of the dataframe will be exported to a separate RDS file so that all partitions can be processed in parallel.

#### Usage

```r
spark_write_rds(x, dest_uri)
```

#### Arguments

- `x` A Spark DataFrame to be exported
- `dest_uri` Can be a URI template containing 'partitionId' (e.g., "hdfs://my_data_part_{partitionId}.rds") or a list of URIs to be assigned to RDS output from all partitions (e.g., "hdfs://my_data_part_0.rds", "hdfs://my_data_part_1.rds", and so on) If working with a Spark instance running locally, then all URIs should be in "file://<local file path>" form. Otherwise the scheme of the URI should reflect the underlying file system.

---

### Arguments

- **x** A Spark DataFrame or dplyr operation
- **path** The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
- **mode** A character element. Specifies the behavior when data or table already exists. Supported values include: ‘error’, ‘append’, ‘overwrite’ and ignore. Notice that ‘overwrite’ will also change the column structure. For more details see also [https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes](https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes) for your version of Spark.
- **options** A list of strings with additional options. See [https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration](https://spark.apache.org/docs/latest/sql-programming-guide.html#configuration).
- **partition_by** A character vector. Partitions the output by the given columns on the file system.
- **...** Optional arguments; currently unused.

#### See Also

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_source()`, `spark_write_table()`, `spark_write_text()`
Spark instance is working with (e.g., "hdfs://"). If the resulting list of URI(s) does not contain unique values, then it will be post-processed with `make.unique()` to ensure uniqueness.

Value

A tibble containing partition ID and RDS file location for each partition of the input Spark dataframe.

---

**spark_write_source**  
*W*rites a Spark DataFrame into a generic source

**Description**

W*rites a Spark DataFrame into a generic source.*

**Usage**

```r
spark_write_source(
  x,  
  source,  
  mode = NULL,  
  options = list(),  
  partition_by = NULL,  
  ...  
)
```

**Arguments**

- `x`  
  A Spark DataFrame or dplyr operation.

- `source`  
  A data source capable of reading data.

- `mode`  
  A character element. Specifies the behavior when data or table already exists. Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that 'overwrite' will also change the column structure. For more details see also [https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes](https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes) for your version of Spark.

- `options`  
  A list of strings with additional options.

- `partition_by`  
  A character vector. Partitions the output by the given columns on the file system.

- `...`  
  Optional arguments; currently unused.

**See Also**

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_table()`, `spark_write_text()`
spark_write_table

Writes a Spark DataFrame into a Spark table

Description

Writes a Spark DataFrame into a Spark table

Usage

spark_write_table(
  x,
  name,
  mode = NULL,
  options = list(),
  partition_by = NULL,
  ...
)

Arguments

x  A Spark DataFrame or dplyr operation
name  The name to assign to the newly generated table.
mode  A character element. Specifies the behavior when data or table already exists. Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that 'overwrite' will also change the column structure. For more details see also https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes for your version of Spark.
options  A list of strings with additional options.
partition_by  A character vector. Partitions the output by the given columns on the file system.
...
Optional arguments; currently unused.

See Also

Other Spark serialization routines: collect_from_rds(), spark_insert_table(), spark_load_table(), spark_read(), spark_read_avro(), spark_read_binary(), spark_read_csv(), spark_read_delta(), spark_read_image(), spark_read_jdbc(), spark_read_json(), spark_read_libsvm(), spark_read_orc(), spark_read_parquet(), spark_read_source(), spark_read_table(), spark_read_text(), spark_save_table(), spark_write_avro(), spark_write_csv(), spark_write_delta(), spark_write_jdbc(), spark_write_json(), spark_write_orc(), spark_write_parquet(), spark_write_source(), spark_write_text()
spark_write_text

Write a Spark DataFrame to a Text file

Description

Serialize a Spark DataFrame to the plain text format.

Usage

```r
spark_write_text(
  x,
  path,
  mode = NULL,
  options = list(),
  partition_by = NULL,
  ...
)
```

Arguments

- **x**: A Spark DataFrame or dplyr operation.
- **path**: The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
- **mode**: A character element. Specifies the behavior when data or table already exists. Supported values include: 'error', 'append', 'overwrite' and ignore. Notice that 'overwrite' will also change the column structure. For more details see also [https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes](https://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes) for your version of Spark.
- **options**: A list of strings with additional options.
- **partition_by**: A character vector. Partitions the output by the given columns on the file system.
- **...**: Optional arguments; currently unused.

See Also

Other Spark serialization routines: `collect_from_rds()`, `spark_insert_table()`, `spark_load_table()`, `spark_read()`, `spark_read_avro()`, `spark_read_binary()`, `spark_read_csv()`, `spark_read_delta()`, `spark_read_image()`, `spark_read_jdbc()`, `spark_read_json()`, `spark_read_libsvm()`, `spark_read_orc()`, `spark_read_parquet()`, `spark_read_source()`, `spark_read_table()`, `spark_read_text()`, `spark_save_table()`, `spark_write_avro()`, `spark_write_csv()`, `spark_write_delta()`, `spark_write_jdbc()`, `spark_write_json()`, `spark_write_orc()`, `spark_write_parquet()`, `spark_write_source()`, `spark_write_table()`
stream_find

src_databases

Description
Show database list

Usage
src_databases(sc, ...)

Arguments
- sc: A spark_connection.
- ...: Optional arguments; currently unused.

stream_find

Description
Finds and returns a stream based on the stream’s identifier.

Usage
stream_find(sc, id)

Arguments
- sc: The associated Spark connection.
- id: The stream identifier to find.

Examples
```r
## Not run:
sc <- spark_connect(master = "local")
sdf_len(sc, 10) %>%
  spark_write_parquet(path = "parquet-in")

stream <- stream_read_parquet(sc, "parquet-in") %>%
  stream_write_parquet("parquet-out")

stream_id <- stream_id(stream)
stream_find(sc, stream_id)
## End(Not run)
```
stream_generate_test  Generate Test Stream

Description
Generates a local test stream, useful when testing streams locally.

Usage
stream_generate_test(
  df = rep(1:1000),
  path = "source",
  distribution = floor(10 + 1e+05 * stats::dbinom(1:20, 20, 0.5)),
  iterations = 50,
  interval = 1
)

Arguments
  df          The data frame used as a source of rows to the stream, will be cast to data frame if needed. Defaults to a sequence of one thousand entries.
  path        Path to save stream of files to, defaults to "source".
  distribution The distribution of rows to use over each iteration, defaults to a binomial distribution. The stream will cycle through the distribution if needed.
  iterations  Number of iterations to execute before stopping, defaults to fifty.
  interval    The interval in seconds use to write the stream, defaults to one second.

Details
This function requires the callr package to be installed.

stream_id  Spark Stream's Identifier

Description
Retrieves the identifier of the Spark stream.

Usage
stream_id(stream)

Arguments
  stream     The spark stream object.
stream_lag

Description

Given a streaming Spark dataframe as input, this function will return another streaming dataframe that contains all columns in the input and column(s) that are shifted behind by the offset(s) specified in ‘...’ (see example)

Usage

stream_lag(x, cols, thresholds = NULL)

Arguments

x An object coercable to a Spark Streaming DataFrame.

cols A list of expressions for a single or multiple variables to create that will contain the value of a previous entry.

thresholds Optional named list of timestamp column(s) and corresponding time duration(s) for determining whether a previous record is sufficiently recent relative to the current record. If the any of the time difference(s) between the current and a previous record is greater than the maximal duration allowed, then the previous record is discarded and will not be part of the query result. The durations can be specified with numeric types (which will be interpreted as max difference allowed in number of milliseconds between 2 UNIX timestamps) or time duration strings such as “5s”, “5sec”, “5min”, “5hour”, etc. Any timestamp column in ‘x’ that is not of timestamp of date Spark SQL types will be interpreted as number of milliseconds since the UNIX epoch.

Examples

```r
## Not run:
library(sparklyr)

sc <- spark_connect(master = "local", version = "2.2.0")

streaming_path <- tempfile("days_df_.")
days_df <- dplyr::tibble(
  today = weekdays(as.Date(seq(7), origin = "1970-01-01"))
)
num_iters <- 7
stream_generate_test(
  df = days_df,
  path = streaming_path,
  distribution = rep(nrow(days_df), num_iters),
  iterations = num_iters
)
```
stream_read_csv(sc, streaming_path) %>%
stream_lag(cols = c(yesterday = today - 1, two_days_ago = today - 2)) %>%
collect() %>%
print(n = 10L)

## End(Not run)

---

### stream_name

<table>
<thead>
<tr>
<th>Spark Stream's Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream_name</td>
<td>Retrieve the name of the Spark stream if available.</td>
</tr>
</tbody>
</table>

#### Usage

```r
stream_name(stream)
```

#### Arguments

- **stream**: The spark stream object.

---

### stream_read_csv

<table>
<thead>
<tr>
<th>Read files created by the stream</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream_read_csv</td>
<td>Read files created by the stream</td>
</tr>
</tbody>
</table>

#### Usage

```r
stream_read_csv(
  sc,
  path,
  name = NULL,
  header = TRUE,
  columns = NULL,
  delimiter = ",",
  quote = "\",
  escape = "\\",
  charset = "UTF-8",
  null_value = NULL,
  options = list(),
  ...
```
stream_read_csv

stream_read_text(sc, path, name = NULL, options = list(), ...)

stream_read_json(sc, path, name = NULL, columns = NULL, options = list(), ...)

stream_read_parquet(sc, path, name = NULL, columns = NULL, options = list(), ...)

stream_read_orc(sc, path, name = NULL, columns = NULL, options = list(), ...)

stream_read_kafka(sc, name = NULL, options = list(), ...)

stream_read_socket(sc, name = NULL, columns = NULL, options = list(), ...)

stream_read_delta(sc, path, name = NULL, options = list(), ...)

stream_read_cloudfiles(sc, path, name = NULL, options = list(), ...)

stream_read_table(sc, path, name = NULL, options = list(), ...)

Arguments

sc  A spark_connection.
path The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
name The name to assign to the newly generated stream.
header Boolean; should the first row of data be used as a header? Defaults to TRUE.
columns A vector of column names or a named vector of column types. If specified, the elements can be "binary" for BinaryType, "boolean" for BooleanType, "byte" for ByteType, "integer" for IntegerType, "integer64" for LongType, "double" for DoubleType, "character" for StringType, "timestamp" for TimestampType and "date" for DateType.
delimiter The character used to delimit each column. Defaults to ",".
quote The character used as a quote. Defaults to "".
escape The character used to escape other characters. Defaults to \".
charset The character set. Defaults to "UTF-8".
null_value The character to use for null, or missing, values. Defaults to NULL.
options A list of strings with additional options.
... Optional arguments; currently unused.
stream_render

Examples

## Not run:

```r
sc <- spark_connect(master = "local")
dir.create("csv-in")
write.csv(iris, "csv-in/data.csv", row.names = FALSE)
csv_path <- file.path("file://", getwd(), "csv-in")
stream <- stream_read_csv(sc, csv_path) %>% stream_write_csv("csv-out")
stream_stop(stream)
## End(Not run)
```

---

**stream_render**  
*Render Stream*

**Description**

Collects streaming statistics to render the stream as an `htmlwidget`.

**Usage**

```r
stream_render(stream = NULL, collect = 10, stats = NULL, ...)
```

**Arguments**

- `stream`  
  The stream to render
- `collect`  
  The interval in seconds to collect data before rendering the `htmlwidget`.
- `stats`  
  Optional stream statistics collected using `stream_stats()`, when specified, `stream` should be omitted.
- `...`  
  Additional optional arguments.

**Examples**

## Not run:

```r
library(sparklyr)
sc <- spark_connect(master = "local")
dir.create("iris-in")
write.csv(iris, "iris-in/iris.csv", row.names = FALSE)
stream <- stream_read_csv(sc, "iris-in/") %>%
  stream_write_csv("iris-out/")
```
stream_stats

Description

Collects streaming statistics, usually, to be used with `stream_render()` to render streaming statistics.

Usage

```
stream_stats(stream, stats = list())
```

Arguments

- `stream`: The stream to collect statistics from.
- `stats`: An optional stats object generated using `stream_stats()`.

Value

A stats object containing streaming statistics that can be passed back to the `stats` parameter to continue aggregating streaming stats.

Examples

```
## Not run:
sc <- spark_connect(master = "local")
sdf_len(sc, 10) %>%
  spark_write_parquet(path = "parquet-in")

stream <- stream_read_parquet(sc, "parquet-in") %>%
  stream_write_parquet("parquet-out")

stream_stats(stream)
## End(Not run)
```
**stream_stop**

*Stops a Spark Stream*

**Description**

Stops processing data from a Spark stream.

**Usage**

```python
stream_stop(stream)
```

**Arguments**

- `stream` The spark stream object to be stopped.

---

**stream_trigger_continuous**

*Spark Stream Continuous Trigger*

**Description**

Creates a Spark structured streaming trigger to execute continuously. This mode is the most performant but not all operations are supported.

**Usage**

```python
stream_trigger_continuous(checkpoint = 5000)
```

**Arguments**

- `checkpoint` The checkpoint interval specified in milliseconds.

**See Also**

- `stream_trigger_interval`
stream_trigger_interval

*Spark Stream Interval Trigger*

**Description**

Creates a Spark structured streaming trigger to execute over the specified interval.

**Usage**

```r
stream_trigger_interval(interval = 1000)
```

**Arguments**

- `interval`: The execution interval specified in milliseconds.

**See Also**

- `stream_trigger_continuous`

---

stream_view

*View Stream*

**Description**

Opens a Shiny gadget to visualize the given stream.

**Usage**

```r
stream_view(stream, ...)
```

**Arguments**

- `stream`: The stream to visualize.
- `...`: Additional optional arguments.

**Examples**

```r
## Not run:
library(sparklyr)
sc <- spark_connect(master = "local")

dir.create("iris-in")
write.csv(iris, "iris-in/iris.csv", row.names = FALSE)

stream_read_csv(sc, "iris-in/") %>%
```

---
stream_write_csv("iris-out/") %>%
stream_view() %>%
stream_stop()

## End(Not run)

---

**stream_watermark**

**Watermark Stream**

### Description

Ensures a stream has a watermark defined, which is required for some operations over streams.

### Usage

```r
stream_watermark(x, column = "timestamp", threshold = "10 minutes")
```

### Arguments

- **x**: An object coercable to a Spark Streaming DataFrame.
- **column**: The name of the column that contains the event time of the row, if the column is missing, a column with the current time will be added.
- **threshold**: The minimum delay to wait to data to arrive late, defaults to ten minutes.

---

**stream_write_csv**

**Write files to the stream**

### Description

Write files to the stream

### Usage

```r
stream_write_csv(
x, path, 
mode = c("append", "complete", "update"),
trigger = stream_trigger_interval(),
checkpoint = file.path(path, "checkpoint"),
header = TRUE,
delimiter = ",",
quote = "\\",
escape = "\\",
charset = "UTF-8",
null_value = NULL,
```

```
options = list(),
partition_by = NULL,
...
)

stream_write_text(
  x,
  path,
  mode = c("append", "complete", "update"),
  trigger = stream_trigger_interval(),
  checkpoint = file.path(path, "checkpoints", random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)

stream_write_json(
  x,
  path,
  mode = c("append", "complete", "update"),
  trigger = stream_trigger_interval(),
  checkpoint = file.path(path, "checkpoints", random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)

stream_write_parquet(
  x,
  path,
  mode = c("append", "complete", "update"),
  trigger = stream_trigger_interval(),
  checkpoint = file.path(path, "checkpoints", random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)

stream_write_orc(
  x,
  path,
  mode = c("append", "complete", "update"),
  trigger = stream_trigger_interval(),
  checkpoint = file.path(path, "checkpoints", random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)
stream_write_kafka(
  x,
  mode = c("append", "complete", "update"),
  trigger = stream_trigger_interval(),
  checkpoint = file.path("checkpoints", random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)

stream_write_console(
  x,
  mode = c("append", "complete", "update"),
  options = list(),
  trigger = stream_trigger_interval(),
  partition_by = NULL,
  ...
)

stream_write_delta(
  x,
  path,
  mode = c("append", "complete", "update"),
  checkpoint = file.path("checkpoints", random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)

## Arguments

- **x**: A Spark DataFrame or dplyr operation
- **path**: The path to the file. Needs to be accessible from the cluster. Supports the "hdfs:////", "s3a:////" and "file:////" protocols.
- **mode**: Specifies how data is written to a streaming sink. Valid values are "append", "complete" or "update".
- **trigger**: The trigger for the stream query, defaults to micro-batches running every 5 seconds. See `stream_trigger_interval` and `stream_trigger_continuous`.
- **checkpoint**: The location where the system will write all the checkpoint information to guarantee end-to-end fault-tolerance.
- **header**: Should the first row of data be used as a header? Defaults to TRUE.
- **delimiter**: The character used to delimit each column, defaults to ".
- **quote**: The character used as a quote. Defaults to `'"'`.
- **escape**: The character used to escape other characters, defaults to \\
- **charset**: The character set, defaults to "UTF-8".
null_value  The character to use for default values, defaults to NULL.
options     A list of strings with additional options.
partition_by Partitions the output by the given list of columns.
...         Optional arguments; currently unused.

See Also

Other Spark stream serialization: `stream_write_memory()`, `stream_write_table()`

Examples

```r
## Not run:
sc <- spark_connect(master = "local")
dir.create("csv-in")
write.csv(iris, "csv-in/data.csv", row.names = FALSE)
csv_path <- file.path("file://", getwd(), "csv-in")
stream <- stream_read_csv(sc, csv_path) %>% stream_write_csv("csv-out")
stream_stop(stream)
## End(Not run)
```

```
stream_write_memory  Write Memory Stream

Description

Writes a Spark dataframe stream into a memory stream.

Usage

```r
stream_write_memory(
  x,
  name = random_string("sparklyr_tmp_"),
  mode = c("append", "complete", "update"),
  trigger = stream_trigger_interval(),
  checkpoint = file.path("checkpoints", name, random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)
```
stream_write_table

Arguments

x A Spark DataFrame or dplyr operation
name The name to assign to the newly generated stream.
mode Specifies how data is written to a streaming sink. Valid values are "append", "complete" or "update".
trigger The trigger for the stream query, defaults to micro-batches running every 5 seconds. See stream_trigger_interval and stream_trigger_continuous.
checkpoint The location where the system will write all the checkpoint information to guarantee end-to-end fault-tolerance.
options A list of strings with additional options.
partition_by Partitions the output by the given list of columns.
... Optional arguments; currently unused.

See Also

Other Spark stream serialization: stream_write_csv(), stream_write_table()

stream_write_table Write Stream to Table

Description

Writes a Spark dataframe stream into a table.

Usage

stream_write_table(
  x,
  path = NULL,
  format = NULL,
  mode = c("append", "complete", "update"),
  checkpoint = file.path("checkpoints", random_string("")),
  options = list(),
  partition_by = NULL,
  ...
)

Arguments

x A Spark DataFrame or dplyr operation
path The path to the file. Needs to be accessible from the cluster. Supports the "hdfs://", "s3a://" and "file://" protocols.
format Specifies format of data written to table. E.g. "delta", "parquet". Defaults to NULL which will use system default format.
mode Specifies how data is written to a streaming sink. Valid values are "append", "complete" or "update".

checkpoint The location where the system will write all the checkpoint information to guarantee end-to-end fault-tolerance.

options A list of strings with additional options.

partition_by Partitions the output by the given list of columns.

... Optional arguments; currently unused.

See Also

Other Spark stream serialization: `stream_write_csv()`, `stream_write_memory()`

tbl_cache Cache a Spark Table

Description

Force a Spark table with name name to be loaded into memory. Operations on cached tables should normally (although not always) be more performant than the same operation performed on an uncached table.

Usage

tbl_cache(sc, name, force = TRUE)

Arguments

sc A spark_connection.
name The table name.
force Force the data to be loaded into memory? This is accomplished by calling the count API on the associated Spark DataFrame.

tbl_change_db Use specific database

Description

Use specific database

Usage

tbl_change_db(sc, name)

Arguments

sc A spark_connection.
name The database name.
### tbl_uncache

**Uncache a Spark Table**

**Description**
Force a Spark table with name name to be unloaded from memory.

**Usage**

tbl_uncache(sc, name)

**Arguments**

- **sc**: A spark_connection.
- **name**: The table name.

### transform_sdf

**transform a subset of column(s) in a Spark Dataframe**

**Description**
transform a subset of column(s) in a Spark Dataframe

**Usage**

transform_sdf(x, cols, fn)

**Arguments**

- **x**: An object coercible to a Spark DataFrame
- **cols**: Subset of columns to apply transformation to
- **fn**: Transformation function taking column name as the 1st parameter, the corresponding org.apache.spark.sql.Column object as the 2nd parameter, and returning a transformed org.apache.spark.sql.Column object

### unite

**Unite**

**Description**
See unite for more details.
unnest  

**Unnest**

**Description**

See `unnest` for more details.

### [.tbl_spark]  

**Subsetting operator for Spark dataframe**

**Description**

Subsetting operator for Spark dataframe allowing a subset of column(s) to be selected using syntaxes similar to those supported by R dataframes.

**Usage**

```r
## S3 method for class 'tbl_spark'
x[i]
```

**Arguments**

- `x`  
The Spark dataframe
- `i`  
Expression specifying subset of column(s) to include or exclude from the result (e.g., `"col1"`, `c("col1", "col2")`, `[1:10]`, `[-1]`, `[NULL]`, or `[]`)

### %->%  

**Infix operator for composing a lambda expression**

**Description**

Infix operator that allows a lambda expression to be composed in R and be translated to Spark SQL equivalent using `dbplyr::translate_sql` functionalities.

**Usage**

```r
params %->% ...  
```

**Arguments**

- `params`  
Parameter(s) of the lambda expression, can be either a single parameter or a comma separated listed of parameters in the form of `(param1, param2, ...)`  
(see examples)
- `...`  
Body of the lambda expression, *must be within parentheses*
Details

Notice when composing a lambda expression in R, the body of the lambda expression *must always be surrounded with parentheses*, otherwise a parsing error will occur.

Examples

```r
## Not run:
a %>% (mean(a) + 1) # translates to <SQL> `a` -> (AVG(`a`) OVER () + 1.0)
.(a, b) %>% (a < 1 && b > 1) # translates to <SQL> `a`, `b` -> (`a` < 1.0 AND `b` > 1.0)
## End(Not run)
```
Index

* Spark data frames
  sdf_copy_to, 166
  sdf_distinct, 169
  sdf_random_split, 177
  sdf_register, 182
  sdf_sample, 192
  sdf_sort, 194
  sdf_weighted_sample, 198

* Spark runtime configuration
  spark_adaptive_query_execution, 203
  spark_advisory_shuffle_partition_size, 204
  spark_auto_broadcast_join_threshold, 208
  spark_coalesce_initial_num_partitions, 208
  spark_coalesce_min_num_partitions, 209
  spark_coalesce_shuffle_partitions, 209
  spark_session_config, 243

* Spark serialization routines
  collect_from_rds, 11
  spark_insert_table, 221
  spark_load_table, 224
  spark_read, 226
  spark_read_avro, 227
  spark_read_binary, 228
  spark_read_csv, 229
  spark_read_delta, 231
  spark_read_image, 232
  spark_read_jdbc, 233
  spark_read_json, 234
  spark_read_libsvm, 235
  spark_read_orc, 236
  spark_read_parquet, 238
  spark_read_source, 239
  spark_read_table, 240
  spark_read_text, 241
  spark_save_table, 242
  spark_write_avro, 247
  spark_write_csv, 248
  spark_write_delta, 249
  spark_write_jdbc, 250
  spark_write_json, 251
  spark_write_orc, 252
  spark_write_parquet, 253
  spark_write_source, 255
  spark_write_table, 256
  spark_write_text, 257

* Spark statistical routines
  sdf_rbeta, 178
  sdf_rbinom, 179
  sdf_rcauchy, 180
  sdf_rchisq, 181
  sdf_rexp, 183
  sdf_rgamma, 184
  sdf_rgeom, 185
  sdf_rhyper, 186
  sdf_rlnorm, 187
  sdf_rnorm, 188
  sdf_rpois, 189
  sdf_rt, 189
  sdf_runif, 190
  sdf_rweibull, 191

* Spark stream serialization
  stream_write_csv, 267
  stream_write_memory, 270
  stream_write_table, 271

* feature transformers
  ft_binarizer, 16
  ft_bucketizer, 17
  ft_chisq_selector, 19
  ft_count_vectorizer, 20
  ft_dct, 22
  ft_elementwise_product, 23
  ft_feature_hasher, 24
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft_hashing_tf</td>
<td>26</td>
</tr>
<tr>
<td>ft_idf</td>
<td>27</td>
</tr>
<tr>
<td>ft_imputer</td>
<td>28</td>
</tr>
<tr>
<td>ft_index_to_string</td>
<td>29</td>
</tr>
<tr>
<td>ft_interaction</td>
<td>30</td>
</tr>
<tr>
<td>ft_lsh</td>
<td>31</td>
</tr>
<tr>
<td>ft_max_abs_scaler</td>
<td>34</td>
</tr>
<tr>
<td>ft_min_max_scaler</td>
<td>35</td>
</tr>
<tr>
<td>ft_ngram</td>
<td>37</td>
</tr>
<tr>
<td>ft_normalizer</td>
<td>38</td>
</tr>
<tr>
<td>ft_one_hot_encoder</td>
<td>39</td>
</tr>
<tr>
<td>ft_one_hot_encoder_estimator</td>
<td>40</td>
</tr>
<tr>
<td>ft_pca</td>
<td>41</td>
</tr>
<tr>
<td>ft_polynomial_expansion</td>
<td>43</td>
</tr>
<tr>
<td>ft_quantile_discretizer</td>
<td>44</td>
</tr>
<tr>
<td>ft_r_formula</td>
<td>48</td>
</tr>
<tr>
<td>ft_regex_tokenizer</td>
<td>46</td>
</tr>
<tr>
<td>ft_robust_scaler</td>
<td>47</td>
</tr>
<tr>
<td>ft_sql_transformer</td>
<td>50</td>
</tr>
<tr>
<td>ft_standard_scaler</td>
<td>51</td>
</tr>
<tr>
<td>ft_stop_words_remover</td>
<td>53</td>
</tr>
<tr>
<td>ft_string_indexer</td>
<td>54</td>
</tr>
<tr>
<td>ft_tokenizer</td>
<td>56</td>
</tr>
<tr>
<td>ft_vector_assembler</td>
<td>57</td>
</tr>
<tr>
<td>ft_vector_indexer</td>
<td>58</td>
</tr>
<tr>
<td>ft_vector_slicer</td>
<td>59</td>
</tr>
<tr>
<td>ft_word2vec</td>
<td>60</td>
</tr>
</tbody>
</table>

* ml algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml_aft_survival_regression</td>
<td>85</td>
</tr>
<tr>
<td>ml_decision_tree_classifier</td>
<td>96</td>
</tr>
<tr>
<td>ml_gbt_classifier</td>
<td>107</td>
</tr>
<tr>
<td>ml_generalized_linear_regression</td>
<td>112</td>
</tr>
<tr>
<td>ml_isotonic_regression</td>
<td>116</td>
</tr>
<tr>
<td>ml_linear_regression</td>
<td>125</td>
</tr>
<tr>
<td>ml_linear_svc</td>
<td>127</td>
</tr>
<tr>
<td>ml_logistic_regression</td>
<td>129</td>
</tr>
<tr>
<td>ml_multilayer_perceptron_classifier</td>
<td>137</td>
</tr>
<tr>
<td>ml_naive_bayes</td>
<td>140</td>
</tr>
<tr>
<td>ml_one_vs_rest</td>
<td>143</td>
</tr>
<tr>
<td>ml_random_forest_classifier</td>
<td>149</td>
</tr>
</tbody>
</table>

?join_by | 77

[.tbl_spark | 274
%
%> prescription is valid, 274

augment.ml_model_decision_tree_classification (ml_tree_tiders), 155
augment.ml_model_gbt_classification (ml_tree_tiders), 155
augment.ml_model_gbt_regression (ml_tree_tiders), 155
augment.ml_model_linear_regression (ml_glm_tiders), 114
augment.ml_model_logistic_regression (ml_logistic_regression_tiders), 132
augment.ml_model_random_forest_classification (ml_tree_tiders), 155
augment.ml_model_random_forest_regression (ml_tree_tiders), 155
augment.ml_model_aft_survival_regression (ml_survival_regression_tiders), 154
augment.ml_model_als (ml_als_tiders), 91
augment.ml_model_bisecting_kmeans (ml_unsupervised_tiders), 157
augment.ml_model_decision_tree_classification (ml_tree_tiders), 155
augment.ml_model_decision_tree_regression (ml_tree_tiders), 155
augment.ml_model_gaussian_mixture (ml_unsupervised_tiders), 157
augment.ml_model_gbt_classification (ml_tree_tiders), 155
augment.ml_model_gbt_regression (ml_tree_tiders), 155
augment.ml_model_generalized_linear_regression (ml_glm_tiders), 114
augment.ml_model_isotonic_regression (ml_isotonic_regression_tiders), 117
augment.ml_model_kmeans (ml_unsupervised_tiders), 157
augment.ml_model_lda (ml_lda_tiders), 124
augment.ml_model_linear_regression (ml_glm_tiders), 114
augment.ml_model_linear_svc (ml_linear_svc_tiders), 129
augment.ml_model_logistic_regression (ml_logistic_regression_tiders), 132
ft stop words remover, 16, 18, 20, 22–24, 26–31, 33, 35, 36, 38–42, 44, 46–48, 50–52, 53, 56, 57, 59–61, 100
ft string indexer model (ft string indexer), 54
ft vector indexer, 16, 18, 20, 22–24, 26–31, 33, 35, 36, 38–42, 44, 46–48, 50–52, 54, 56, 56, 57, 58, 60, 61
ft word2vec, 16, 18, 20, 22–24, 26–31, 33, 35, 36, 38–42, 44, 46–48, 50–52, 54, 56, 57, 59, 60, 60
full join, 61, 61
full join tbl_spark (join tbl_spark), 75
generic call interface, 62
get spark sql catalog implementation, 62
glance.ml model aft survival regression
(ml survival regression tidiers), 154
glance.ml model als (ml als tidiers), 91
glance.ml model bisecting kmeans
(ml unsupervised tidiers), 157
glance.ml model decision tree classification
(ml tree tidiers), 155
glance.ml model decision tree regression
(ml tree tidiers), 155
glance.ml model gaussian mixture
(ml unsupervised tidiers), 157
glance.ml model gbt classification
(ml tree tidiers), 155
glance.ml model gbt regression
(ml tree tidiers), 155
glance.ml model generalized linear regression
(ml glm tidiers), 114
glance.ml model isotonic regression
(ml isotonic regression tidiers), 117
glance.ml model kmeans
(ml unsupervised tidiers), 157
glance.ml model lda (ml lda tidiers), 124
glance.ml model linear regression
(ml glm tidiers), 114
glance.ml model linear svc
(ml linear svc tidiers), 129
glance.ml model logistic regression
(ml logistic regression tidiers), 132
glance.ml model multilayer perceptron classification
(ml multilayer perceptron tidiers), 140
glance.ml model naive bayes
(ml naive bayes tidiers), 142
glance.ml model pca (ml pca tidiers), 144
glance.ml model random forest classification
(ml tree tidiers), 155
glance.ml_model_random_forest_regression
   (ml_tree_tidiers), 155

hive_context (spark-api), 200
hive_context_config, 63
hof_aggregate, 63
hof_array_sort, 64
hof_exists, 65
hof_filter, 65
hof_forall, 66
hof_map_filter, 67
hof_map_zip_with, 68
hof_transform, 69
hof_transform_keys, 70
hof_transform_values, 71
hof_zip_with, 71

inner_join, 72, 72
inner_join.tbl_spark (join.tbl_spark), 75
invoke, 73, 200, 223
invoke_new (invoke), 73
invoke_static (invoke), 73
is_ml_estimator (ml-transform-methods), 82
is_ml_transformer (ml-transform-methods), 82

j_invoke, 77
j_invoke_new (j_invoke), 77
j_invoke_static (j_invoke), 77
jarray, 73
java_context (spark-api), 200
jfloat, 74
jfloat_array, 75
join.tbl_spark, 75
join_by(), 76, 77

left_join, 78, 78
left_join.tbl_spark (join.tbl_spark), 75
list_sparklyr_jars, 78
livy_config, 78
livy_service_start, 80
livy_service_stop (livy_service_start), 80

ml-params, 80
ml-persistence, 81
ml-transform-methods, 82, 162
ml-tuning, 83
ml_aft_survival_regression, 85, 99, 111,
   114, 117, 126, 128, 131, 139, 142,
   144, 153
ml_als, 88
ml_als_tidiers, 91
ml_approx_nearest_neighbors
   (ft_lsh_utils), 33
ml_approx_similarity_join
   (ft_lsh_utils), 33
ml_association_rules (ml_fpgrowth), 105
ml_binary_classification_eval
   (ml_evaluator), 102
ml_binary_classification_evaluator
   (ml_evaluator), 102
ml_bisecting_kmeans, 91
ml_chisquare_test, 93
ml_classification_eval (ml_evaluator), 102
ml_clustering_evaluator, 94
ml_compute_cost (ml_kmeans), 118
ml_compute_silhouette_measure
   (ml_kmeans), 118
ml_corr, 95
ml_cross_validator (ml-tuning), 83
ml_decision_tree
   (ml_decision_tree_classifier), 96
ml_decision_tree_classifier, 87, 96, 111,
   114, 117, 126, 128, 131, 139, 142,
   144, 153
ml_decision_tree_regressor
   (ml_decision_tree_classifier), 96
ml_default_stop_words, 54, 100
ml_describe_topics (ml_lda), 120
ml_evaluate, 101
ml_evaluator, 84, 102
ml_feature_importances, 104
ml_find_synonyms (ft_word2vec), 60
ml_fit (ml-transform-methods), 82
ml_fit_and_transform
   (ml-transform-methods), 82
ml_fpgrowth, 105
ml_freq_itemsets (ml_fpgrowth), 105
ml_freq_seq_patterns (ml_prefixspan), 148
ml_gaussian_mixture, 106
INDEX

ml_gbt_classifier, 87, 99, 107, 114, 117, 126, 128, 131, 139, 142, 144, 153
ml_gbt_regressor (ml_gbt_classifier), 107
ml_generalized_linear_regression, 87, 99, 111, 112, 117, 126, 128, 131, 139, 142, 144, 153
ml_glm_tidiers, 114
ml_gradient_boosted_trees (ml_gbt_classifier), 107
ml_is_set (ml-params), 80
ml_isotonic_regression, 87, 99, 111, 114, 116, 126, 128, 131, 139, 142, 144, 153
ml_isotonic_regression_tidiers, 117
ml_kmeans, 118
ml_kmeans_cluster_eval, 120
ml_labels (ft_string_indexer), 54
ml_lda, 120
ml_lda_tidiers, 124
ml_linear_regression, 87, 99, 111, 114, 117, 125, 128, 131, 139, 142, 144, 153
ml_linear_svc, 87, 99, 111, 114, 117, 126, 127, 131, 139, 142, 144, 153
ml_linear_svc_tidiers, 129
ml_load (ml-persistence), 81
ml_log_likelihood (ml_lda), 120
ml_log_perplexity (ml_lda), 120
ml_logistic_regression, 87, 99, 111, 114, 117, 126, 128, 129, 139, 142, 144, 153
ml_logistic_regression_tidiers, 132
ml_metrics_binary, 133
ml_metrics_multiclass, 134
ml_metrics_regression, 135
ml_model_data, 136
ml_multiclass_classification_evaluator (ml_evaluator), 102
ml_multilayer_perceptron (ml_multilayer_perceptron_classifier), 137
ml_multilayer_perceptron_classifier, 87, 99, 111, 114, 117, 126, 128, 131, 137, 142, 144, 153
ml_multilayer_perceptron_tidiers, 140
ml_naive_bayes, 87, 99, 111, 114, 117, 126, 128, 131, 139, 140, 144, 153
ml_naive_bayes_tidiers, 142
ml_one_vs_rest, 87, 99, 111, 114, 117, 126, 128, 131, 139, 142, 143, 153
ml_param (ml-params), 80
ml_param_map (ml-params), 80
ml_params (ml-params), 80
ml_pca (ft_pca), 41
ml_pca_tidiers, 144
ml_pipeline, 145
ml_power_iteration, 145
ml_predict (ml-transform-methods), 82
ml_prefixspan, 148
ml_random_forest (ml_random_forest_classifier), 149
ml_random_forest_classifier, 87, 99, 111, 114, 117, 126, 128, 131, 139, 142, 144, 149
ml_random_forest_regressor (ml_random_forest_classifier), 149
ml_recommend (ml_als), 88
ml_regression_evaluator (ml_evaluator), 102
ml_save (ml-persistence), 81
ml_stage, 153
ml_stages (ml-stage), 153
ml_sub_models (ml-tuning), 83
ml_summary, 154
ml_survival_regression (ml_aft_survival_regression), 85
ml_survival_regression_tidiers, 154
ml_topics_matrix (ml_lda), 120
ml_train_validation_split (ml-tuning), 83
ml_transform (ml-transform-methods), 82
ml_tree_feature_importance (ml_feature_importances), 104
ml_tree_tidiers, 155
ml_uid, 157
ml_unsupervised_tidiers, 157
ml_validation_metrics (ml-tuning), 83
ml_vocabulary (ft_count_vectorizer), 20
mutate, 158, 158
NA, 158
na.replace, 158
nest, 158, 158
numeric_version, 244
pivot_longer, 159, 159
pivot_wider, 159, 159
random_string, 159
reactiveSpark, 160
register_extension, 161
registerDoSpark, 160
registered_extensions
  (register_extension), 161
replace_na, 161, 161
right_join, 161, 161
right_join.tbl_spark (join.tbl_spark), 75
sdf-saveload, 162
sdf-transform-methods, 82, 162
sdf_along, 163
sdf_bound, 163
sdf_bound_cols (sdf_bound), 163
sdf_bound_rows (sdf_bound), 163
sdf_broadcast, 164
sdf_checkpoint, 165
sdf_coalesce, 165
sdf_collect, 166
sdf_copy_to, 166, 169, 178, 182, 192, 194, 198
sdf_crosstab, 167
sdf_debug_string, 168
sdf_describe, 168
sdf_dim, 169
sdf_distinct, 167, 169, 178, 182, 192, 194, 198
sdf_drop_duplicates, 170
sdf_expand_grid, 170
sdf_fit (sdf-transform-methods), 162
sdf_fit_and_transform
  (sdf-transform-methods), 162
sdf_from_avro, 171
sdf_import (sdf_copy_to), 166
sdf_is_streaming, 171
sdf_last_index, 172
sdf_len, 172
sdf_load_parquet (sdf-saveload), 162
sdf_load_table (sdf-saveload), 162
sdf_ncol (sdf_dim), 169
sdf_nrow (sdf_dim), 169
sdf_num_partitions, 173
sdf_partition (sdf_random_split), 177
sdf_partition_sizes, 173
sdf_persist, 174
sdf_pivot, 174
sdf_predict, 94, 103
sdf_predict (sdf-transform-methods), 162
sdf_project, 175
sdf_quantile, 176
sdf_random_split, 167, 169, 177, 182, 192, 194, 198
sdf_rbeta, 178, 180, 181, 184–191
sdf_rbinom, 179, 179, 180, 181, 184–191
sdf_rcauchy, 179, 180, 180, 181, 184–191
sdf_rchisq, 179, 180, 181, 184–191
sdf_read_column, 181
sdf_register, 167, 169, 178, 182, 192, 194, 198
sdf_repartition, 182
sdf_residuals
  (sdf_residuals.ml_model_generalized_linear_regression), 183
sdf_residuals.ml_model_generalized_linear_regression, 183
sdf_rexp, 179–181, 183, 185–191
sdf_rgamma, 179–181, 184, 184, 185–191
sdf_rgeom, 179–181, 184, 185, 186–191
sdf_rhyper, 179–181, 184, 185, 186, 187–191
sdf_rlnorm, 179–181, 184–186, 187, 188–191
sdf_rnorm, 179–181, 184–187, 188, 189–191
sdf_rpois, 179–181, 184–188, 189, 190, 191
sdf_rt, 179–181, 184–189, 189, 191
sdf_runif, 179–181, 184–190, 190, 191
sdf_rweibull, 179–181, 184–191, 191
sdf_sample, 167, 169, 178, 182, 192, 194, 198
sdf_save_parquet (sdf-saveload), 162
sdf_save_table (sdf-saveload), 162
sdf_schema, 192
sdf_separate_column, 193
sdf_seq, 193
sdf_sort, 167, 169, 178, 182, 192, 194, 198
sdf_sql, 194
sdf_to_avro, 195
sdf_transform (sdf-transform-methods), 162
sdf_unnest_longer, 195
sdf_unnest_wider, 196
INDEX

sdf_weighted_sample, 167, 169, 178, 182, 192, 194, 198
sdf_with_sequential_id, 198
sdf_with_unique_id, 199
select, 199, 199
separate, 199, 199
spark-api, 200
spark-connections, 201
spark_adaptive_query_execution, 203, 204, 208–210, 243
spark_advisory_shuffle_partition_size, 204, 208–210, 243
spark_apply, 204
spark_apply_bundle, 207
spark_apply_log, 207
spark_auto_broadcast_join_threshold, 204, 208, 209, 210, 243
spark_available_versions (spark_install), 222
spark_coalesce_initial_num_partitions, 204, 208, 209, 210, 243
spark_coalesce_min_num_partitions, 204, 208, 209, 210, 243
spark_coalesce_shuffle_partitions, 204, 208, 209, 210, 243
spark_compilation_spec, 210
spark_config, 202, 211, 215
spark_config_kubernetes, 212
spark_config_settings, 213
spark_connect (spark-connections), 201
spark_connect_method, 214
spark_connection, 213
spark_connection-class, 213
spark_connection_find, 214
spark_connection_is_open (spark-connections), 201
spark_context (spark-api), 200
spark_context_config, 215
spark_dataframe, 216
spark_default_compilation_spec, 216
spark_dependency, 217
spark_dependency_fallback, 218
spark_disconnect (spark-connections), 201
spark_disconnect_all (spark-connections), 201
spark_extension, 218
spark_get_checkpoint_dir (checkpoint_directory), 10
spark_home_dir, 210
spark_home_set, 219
spark_ide_columns (spark_ide_connection_open), 219
spark_ide_connection_actions (spark_ide_connection_open), 219
spark_ide_connection_closed (spark_ide_connection_open), 219
spark_ide_connection_open, 219
spark_ide_connection_updated (spark_ide_connection_open), 219
spark_ide_objects (spark_ide_connection_open), 219
spark_ide_preview (spark_ide_connection_open), 219
spark_install, 202, 215, 222
spark_install_dir (spark_install), 222
spark_install_tar (spark_install), 222
spark_installed_versions (spark_install), 222
spark_integ_test_skip, 223
spark_job, 216, 223
spark_job-class, 224
spark_last_error, 224
spark_log, 225
spark_read_csv, 11, 221, 225, 226, 228, 229, 231, 233–237, 239–243, 248–257
sparklyr::register_extension, 202, 215
sparklyr_get_backend_port, 203
sql_variant, 217
src_databases, 258
stream_find, 258
stream_generate_test, 259
stream_id, 259
stream_lag, 260
stream_name, 261
stream_read_cloudfiles, 261
(stream_read_csv), 261
stream_read_csv, 261
stream_read_delta(stream_read_csv), 261
stream_read_json (stream_read_csv), 261
stream_read_kafka (stream_read_csv), 261
stream_read_orc (stream_read_csv), 261
stream_read_parquet (stream_read_csv), 261
stream_read_socket (stream_read_csv), 261
spark_write_delta, 11, 221, 225, 226, 228–231, 233–237, 239–243, 248, 249, 249, 251–257
spark_write_rds, 254
sparklyr::register_extension, 202, 215
sparklyr_get_backend_port, 203
sql_variant, 217
src_databases, 258
stream_find, 258
stream_generate_test, 259
stream_id, 259
stream_lag, 260
stream_name, 261
stream_read_cloudfiles, 261
(stream_read_csv), 261
stream_read_csv, 261
stream_read_delta(stream_read_csv), 261
stream_read_json (stream_read_csv), 261
stream_read_kafka (stream_read_csv), 261
stream_read_orc (stream_read_csv), 261
stream_read_parquet (stream_read_csv), 261
stream_read_socket (stream_read_csv), 261
stream_read_table (stream_read_csv), 261
stream_read_text (stream_read_csv), 261
stream_render, 263
stream_stats, 264
stream_stop, 265
stream_trigger_continuous, 265, 266, 269, 271
stream_trigger_interval, 265, 266, 269, 271
stream_view, 266
stream_watermark, 267
stream_write_console
(stream_write_csv), 267
stream_write_csv, 267, 271, 272
stream_write_delta (stream_write_csv), 267
stream_write_json (stream_write_csv), 267
stream_write_kafka (stream_write_csv), 267
stream_write_memory, 270, 270, 272
stream_write_orc (stream_write_csv), 267
stream_write_parquet (stream_write_csv), 267
stream_write_table, 270, 271, 271
stream_write_text (stream_write_csv), 267
tbl_cache, 272
tbl_change_db, 272
tbl_uncache, 273
tibble, 196, 197
tidy.ml_model_aft_survival_regression (ml_glm_tidiers), 154
 tidy.ml_model_generalized_linear_regression (ml_glm_tidiers), 114
tidy.ml_model_isotonic_regression (ml_isotonic_regression_tidiers), 117
tidy.ml_model_kmeans (ml_unsupervised_tidiers), 157
tidy.ml_model_lda (ml_lda_tidiers), 124
tidy.ml_model_linear_regression (ml_glm_tidiers), 114
tidy.ml_model_linear_svc (ml_linear_svc_tidiers), 129
tidy.ml_model_logistic_regression (ml_logistic_regression_tidiers), 132
tidy.ml_model_multilayer_perceptron_classification (ml_multilayer_perceptron_tidiers), 140
tidy.ml_model_naive_bayes (ml_naive_bayes_tidiers), 142
tidy.ml_model_pca (ml_pca_tidiers), 144
tidy.ml_model_random_forest_classification (ml_tree_tidiers), 155
tidy.ml_model_random_forest_regression (ml_tree_tidiers), 155
transform_sdf, 273
unite, 273, 273
unnest, 274, 274