leapyear

Release 0.0.0+e880722f08b0ea637b999a6642a6db0216bf5679

unknown

Jul 22, 2022

CONTENTS

Welcome to the Python API documentation for the LeapYear Secure ML module. This documentation is meant to provide a detailed reference for programmatically interacting with the LeapYear Core software.

Visit guides.leapyear.io to access higher-level material such as:

- an executive-level introduction to LeapYear.
- an architecture overview.
- a step-by-step tutorial for building your first model in LeapYear.
- a data science reference containing code snippets for data exploration, feature engineering, supervised and unsupervised learning.
- best practices for differentially private analytics.
- guides for admins and architects.

CHAPTER

ONE

TABLE OF CONTENTS:

1.1 Getting Started

1.1.1 Connecting to LeapYear and Exploring

The first step to using LeapYear's data security platform for analysis is getting connected. To get started, we'll import the *Client* object from the *leapyear* python library and connect to the LeapYear server using our user credentials.

Credentials used for this tutorial:

```
>>> url = 'http://localhost:{}'.format(os.environ.get('LY_PORT', 4401))
>>> username = 'tutorial_user'
>>> password = 'abcdefghiXYZ1!'
```

Import the *Client* object:

```
>>> from leapyear import Client
```

Create a connection:

```
>>> client = Client(url, username, password)
>>> client.connected
True
>>> client.close()
>>> client.connected
False
```

Alternatively, *Client* is also a context manager, so the connection is automatically closed at the end of a `with` block:

```
>>> with Client(url, username, password) as client:
...     # carry out computations with connection to LeapYear
...     client.connected
True
>>> client.connected
False
```

Databases, Tables and Columns

Once we've obtained a connection to LeapYear, we can look through the databases and tables that are available for data analysis:

```
>>> client = Client(url, username, password)
```

Examine databases available to the user:

```
>>> client.databases.keys()
dict_keys(['tutorial'])
>>> tutorial_db = client.databases['tutorial']
>>> tutorial_db
<Database tutorial>
```

Examine tables within the database *tutorial*:

```
>>> sorted(tutorial_db.tables.keys())
['classification',
 'regression1',
 'regression2',
 'twoclass']
>>> example1 = tutorial_db.tables['regression1']
>>> example1
<Table tutorial.regression1>
```

Examine the columns on table *tutorial_db.regression1*:

```
>>> example1.columns
{'x0': <TableColumn tutorial.regression1.x0: type='REAL' bounds=(-4.0, 4.0)␣
↪nullable=False>,
 'x1': <TableColumn tutorial.regression1.x1: type='REAL' bounds=(-4.0, 4.0)␣
↪nullable=False>,
 'x2': <TableColumn tutorial.regression1.x2: type='REAL' bounds=(-4.0, 4.0)␣
↪nullable=False>,
 'y':  <TableColumn tutorial.regression1.y:  type='REAL' bounds=(-400.0, 400.0)␣
↪nullable=False>}
```

Column Types

TableColumn objects include their type, bounds, and nullability.

```
>>> col_x0 = example1.columns['x0']
>>> col_x0.type
<ColumnType.REAL: 'REAL'>
>>> col_x0.bounds
(-4.0, 4.0)
>>> col_x0.nullable
False
```

The possible types are: `BOOL`, `INT`, `REAL`, `FACTOR`, `DATE`, `TEXT`, and `DATETIME`.

`INT`, `REAL`, `DATE`, and `DATETIME` have publicly available bounds, representing the lower and upper limits of the data in the column. `FACTOR` also has bounds, representing the set of strings available in the column. `BOOL` and `TEXT` columns have no bounds.

1.1.2 The DataSet Class

Once we've established a connection to the LeapYear server using the *Client* class, we can import the *DataSet* to access and analyze tables.

```
>>> from leapyear import DataSet
```

We can access tables, either using the client interface as above:

```
>>> ds_example1 = DataSet.from_table(example1)
```

or by directly referencing the table by name:

```
>>> ds_example1 = DataSet.from_table('tutorial.regression1')
```

The *DataSet* class is the primary way of interacting with data in the LeapYear system. A *DataSet* is associated with collection of *Attributes*, which can be used to compute statistics. The *DataSet* class allows the user to manipulate and analyze the attributes of a data source using a variety of relational operations such as column selection, row selection based on conditions, unions, joins, etc.

An instance of the *Attribute* class represents either an individual named column in the *DataSet* or a transformation of one or several of such columns via supported operations. Attributes also have types, which can be inspected the same as the types in a *DataSet* schema. *Attributes* can be manipulated using most built in Python operations, such as `+`, `*`, and `abs`.

```
>>> ds_example1.schema
Schema([('x0', AttributeType(name='REAL', nullable=False, domain=(-4, 4))),
        ('x1', AttributeType(name='REAL', nullable=False, domain=(-4, 4))),
        ('x2', AttributeType(name='REAL', nullable=False, domain=(-4, 4))),
        ('y',  AttributeType(name='REAL', nullable=False, domain=(-400, 400)))])
>>> ds_example1.schema['x0']
AttributeType(name='REAL', nullable=False, domain=(-4, 4))
>>> ds_example1.schema['x0'].name
'REAL'
>>> ds_example1.schema['x0'].nullable
False
>>> ds_example1.schema['x0'].domain
(-4.0, 4.0)
>>> attr_x0 = ds_example1['x0']
>>> attr_x0
<Attribute: x0>
>>> attr_x0 + 4
<Attribute: x0 + 4>
>>> attr_x0.type
AttributeType(name='REAL', nullable=False, domain=(-4, 4))
>>> attr_x0.type.name
'REAL'
>>> attr_x0.type.nullable
False
>>> attr_x0.type.domain
(-4.0, 4.0)
```

In the following example, we'll take a few attributes from the table `tutorial.regression1`, adding one to the `x1` attribute and multiplying `x2` by three. The bounds are altered to reflect the change.

```
>>> ds1 = ds_example1.map_attributes(
...     {'x1': lambda att: att + 1.0, 'x2': lambda att: att * 3.0}
```

(continues on next page)

(continued from previous page)

```
... )
>>> ds1.schema
Schema([('x0', AttributeType(name='REAL', nullable=False, domain=(-4, 4))),
        ('x1', AttributeType(name='REAL', nullable=False, domain=(-3, 5))),
        ('x2', AttributeType(name='REAL', nullable=False, domain=(-12, 12))),
        ('y',  AttributeType(name='REAL', nullable=False, domain=(-400, 400)))])
```

We can use *DataSet* to filter the data to examine subsets of the data, e.g. by applying predicates to the data:

```
>>> ds2 = ds_example1.where(ds_example1['x1'] > 1)
>>> ds2.schema
Schema([('x0', AttributeType(name='REAL', nullable=False, domain=(-4, 4))),
        ('x1', AttributeType(name='REAL', nullable=False, domain=(1, 4))),
        ('x2', AttributeType(name='REAL', nullable=False, domain=(-4, 4))),
        ('y',  AttributeType(name='REAL', nullable=False, domain=(-400, 400)))])
```

1.2 Data Analysis

1.2.1 Statistics

The LeapYear system is designed to allow access to various statistical functions and develop machine learning models based on data in *DataSet*. The analytics function is not executed until the *run()* method is called on it. This allows inspection of the overall workflow and early reporting of errors. All analysis functions are located in the *leapyear.analytics* module.

```
>>> import leapyear.analytics as analytics
```

Many common statistics functions are available including:

- *count()*
- *count_distinct()*
- *median()*
- *min()*
- *max()*
- *mean()*
- *sum()*
- *variance()*
- *histogram()*

Next is an example of obtaining simple statistics from the dataset:

```
>>> mean_analysis = analytics.mean('x0', ds_example1)
>>> mean_analysis.run()
0.039159280186637294
>>> variance_analysis = analytics.variance('x0', ds_example1)
>>> variance_analysis.run()
1.0477940098374177
>>> quantile_analysis = analytics.quantile(0.25, 'x0', ds_example1)
```

(continues on next page)

(continued from previous page)

```
>>> quantile_analysis.run()
-0.6575000000000001
```

By combining statistics with the ability to transform and filter data, we can look at various statistics associated to subsets of the data:

```
>>> analytics.mean('x0', ds_example1).run()
0.039159280186637294
>>> ds2 = ds_example1.where(ds_example1['x1'] > 1)
>>> analytics.mean('x0', ds2).run()
0.14454229785771325
```

1.2.2 Machine Learning

The *leapyear.analytics* module also supports various machine learning (ML) models, including

- regression-based models (linear, logistic, generalized),
- tree-based models (random forests for classification and regression tasks),
- unsupervised models (e.g. K-means, PCA),
- the ability do optimize model hyperparemeters via search with cross-validation, and
- the ability to evaluate model performance based on a variety of common validation metrics.

In this section we will share some examples of the machine learning tools provided by the LeapYear system.

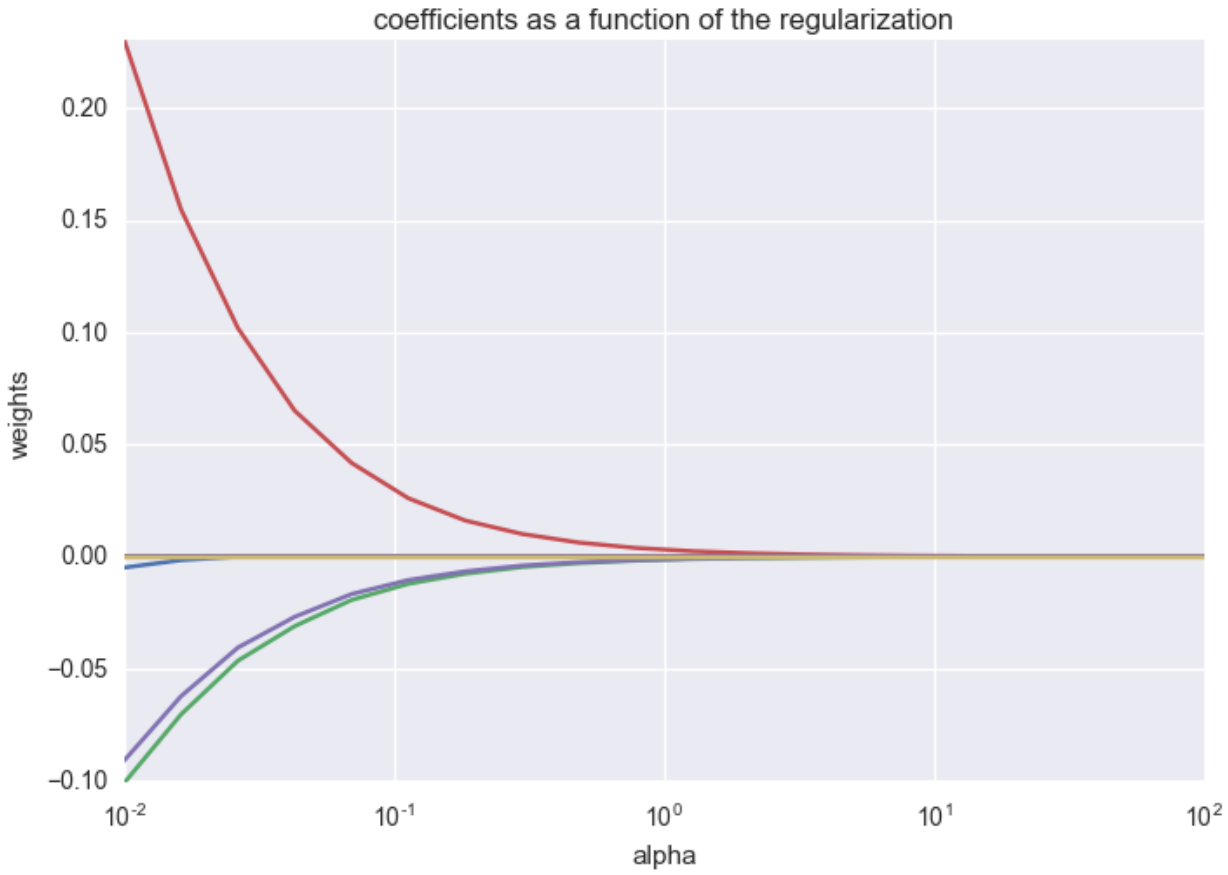
The Effect of L2 Regularization on Model Coefficients

The following example code shows a common theoretical result from ML: as the L2 regularization parameter *alpha* increases, we see the coefficients of the model gradually approach zero. This is depicted in the graph generated below:

```
>>> n_alphas = 20
>>> alphas = np.logspace(-2,2, n_alphas)
>>>
>>> # example3 has 0 and 1 in the y column. Here, we convert 1 to True and 0 to False
>>> ds_example3 = DataSet\
...     .from_table('tutorial.classification')\
...     .map_attribute('y', lambda att: att.decode({1: True}).coalesce(False))
>>>
>>> models = []
>>> for alpha in alphas:
...     model = analytics.generalized_logreg(
...         ['x0','x1','x2','x3','x4','x5','x6','x7','x8','x9'],
...         'y',
...         ds_example3,
...         affine=False,
...         l1reg=0.001,
...         l2reg=alpha
...     ).run()
...     models.append(model)
>>>
>>> coefs = np.array([np.append(m.coefficients, m.intercept) for m in models]).
↪reshape((n_alphas,11))
```

Plotting the coefficients with respect to alpha values:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.plot(alphas, coefs)
>>> plt.xscale('log')
>>> plt.xlabel('alpha')
>>> plt.ylabel('weights')
>>> plt.title('coefficients as a function of the regularization')
>>> plt.axis('tight')
>>> plt.show()
```



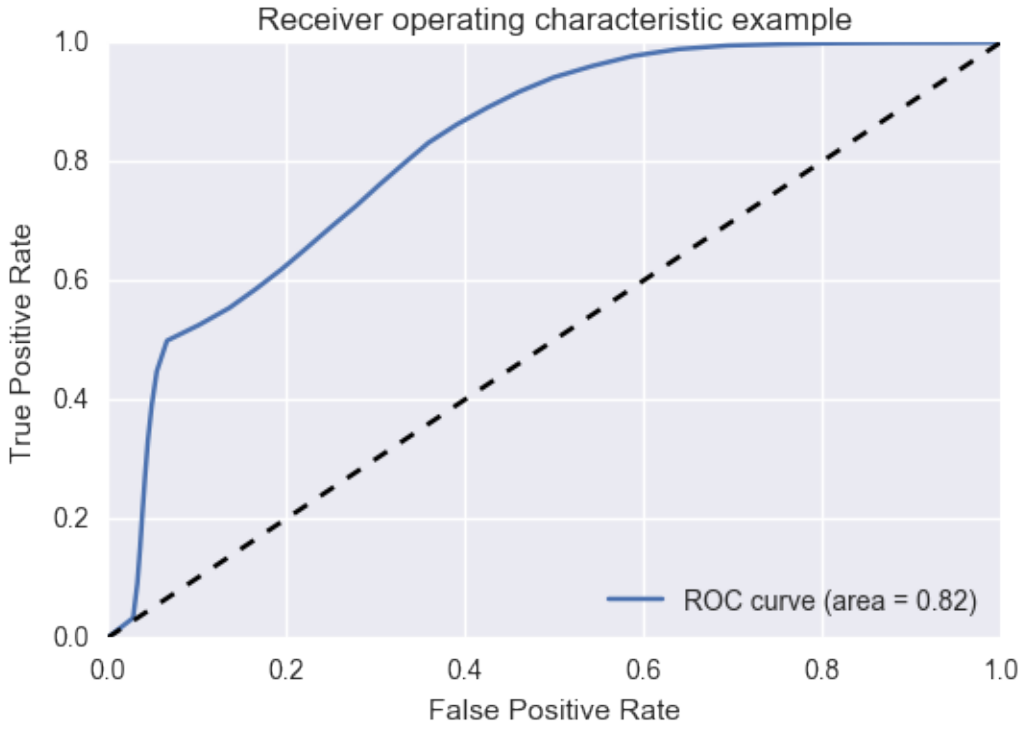
Training a Simple Logistic Regression Model

This example shows how to compute a logistic regression classifier and evaluate it's performance using the receiver operating characteristic (ROC) curve.

```
>>> ds_train = ds_example3.split(0, [80, 20])
>>> ds_test = ds_example3.split(1, [80, 20])
>>> glm = analytics.generalized_logreg(['x1'], 'y', ds_train, affine=True, l1reg=0,␣
↪l2reg=0.01).run()
>>> cc = analytics.roc(glm, ['x1'], 'y', ds_test, thresholds=32).run()
```

Plot the ROC and display the area under the ROC:

```
>>> plt.figure()
>>> plt.plot(cc.fpr, cc.tpr, label='ROC curve (area = %0.2f)' % cc.auc_roc)
>>> plt.plot([0, 1], [0, 1], 'k--')
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.title('Receiver operating characteristic example')
>>> plt.legend(loc="lower right")
>>> plt.show()
```



Training a Random Forest

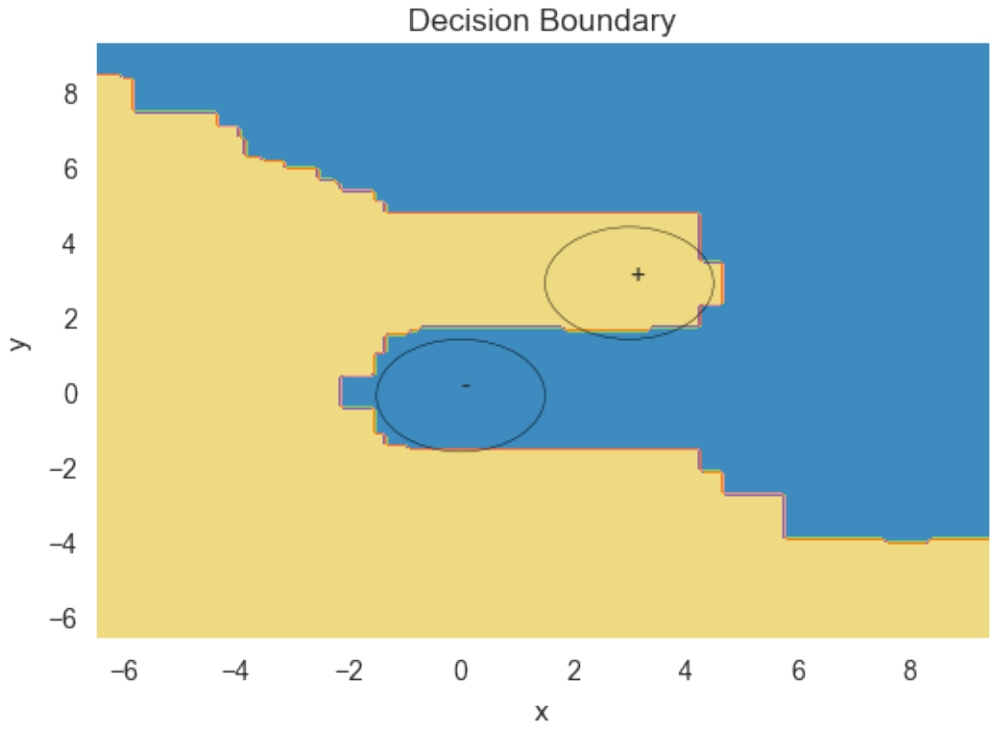
In this example we train a random forest classifier on a binary classification problem associated to two overlapping gaussian distributions centered at `(0,0)` and `(3,3)`. Points around `(0,0)` are labeled as in the negative class while points around `(3,3)` are labeled as in the positive class.

```
>>> ds_example4 = DataSet.from_table('tutorial.twoclass')
>>> rf = analytics.random_forest(['x1', 'x2'], 'y', ds_example4, 100, 1).run()

>>> plot_colors = "br"
>>> plot_step = 0.1
>>>
>>> x_min, x_max = 1.5-8, 1.5+8
>>> y_min, y_max = 1.5-8, 1.5+8
>>> xx, yy = np.meshgrid(
...     np.arange(x_min, x_max, plot_step),
...     np.arange(y_min, y_max, plot_step)
... )
>>> Z = rf.predict(np.c_[xx.ravel(), yy.ravel()])
>>> Z = Z.reshape(xx.shape)
```

Plot the decision boundary:

```
>>> fig, ax = plt.subplots()
>>> plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
>>> # Draw circles centered at the gaussian distributions
>>> ax.add_artist(plt.Circle((0,0), 1.5, color='k', fill=False))
>>> ax.add_artist(plt.Circle((3,3), 1.5, color='k', fill=False))
>>> ax.text(3, 3, '+')
>>> ax.text(0, 0, '-')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('Decision Boundary')
```



This concludes the user tutorial section, so the connection should be closed.

```
>>> client.close()
>>> client.connected
False
```

1.3 Management and Administration

Administration tasks use the *Client* class from the *leapyear* module and admin classes from the *leapyear.admin*. These admin classes include:

- *User*
- *Database*
- *Table*
- Column

These classes provide API's for various administrator tasks on the LeapYear system. All of the examples in the administrative examples section will require correct permissions.

1.3.1 Managing the LeapYear Server

Management requires sufficient privileges. The examples below assume the *lyadmin* user is an administrator of the LeapYear deployment system.

```
>>> client = Client(url, 'lyadmin', ROOT_PASSWORD)
>>> client.connected
True
```

User Management

User objects are used as the primary API for managing users. Below is an example of a user being created, their password updated, and finally their account is disabled.

```
>>> # Create the user
>>> user = User('new_user', password)
>>> client.create(user)
>>> 'new_user' in client.users
True
>>>
>>> # Update the user's password
>>> new_password = '{}100'.format(password)
>>> user.update(password=new_password)
<User new_user>
>>>
>>> # Disable the user
>>> user.enabled
True
>>>
>>> user.enabled = False
>>> user.enabled
False
```

Database Management

Database objects are used to view and manipulate databases on the server.

```
>>> # create database
>>> client.create(Database('sales'))
>>>
>>> # retrieve a reference to the database
>>> sales_database = client.databases['sales']
>>>
>>> # drop database
>>> client.drop(sales_database)
```

Table Management

Table objects are used to view and manipulate tables in a database on the server. Below is an example of how to define a data source (table) object on the LeapYear server.

```
>>> credentials = 'hdfs:///path/to/data.parquet'
>>>
>>> # create a table
>>> accounts = Database('accounts')
>>> table = Table('users', credentials=credentials, database=accounts)
>>>
>>> client.create(accounts)
>>> client.create(table)
>>>
>>> # retrieve a reference to the table
>>> users_table = accounts.tables['users']
>>>
>>> # drop a table
>>> client.drop(users_table)
```

1.4 LeapYear Python API

1.4.1 Module leapyear

The LeapYear Client connects the python API to the LeapYear server.

While connected to the server, the connection information is stored in a resource manager, so direct access to the client is not necessary for all operations.

The main objects that can be directly accessed by the *Client* object are databases, users and permissions resources. Many resources are cached to reduce network latency, however the `load()` methods for resource objects will force refreshing of the metadata.

```
>>> from leapyear import Client
>>> from leapyear.admin import Database, Table, ColumnDefinition
```

Connecting to the server

- Method 1 - manually open and close the connection.

```
>>> SERVER_URL = 'https://ly-server:4401'
>>> client = Client(url=SERVER_URL)
>>> print(client.databases)
{}
>>> client.close()
```

- Method 2 - Use a context to automatically close the connection.

```
>>> with Client(url=SERVER_URL) as client:
...     print(client.databases)
{}
```

Create a User

There are two ways to create an object on the LeapYear server. The first is to invoke the *Client.create()* method.

```
>>> with Client(url=SERVER_URL) as client:
...     client.create(User('new_user', 'password'))
```

The second is to call the `create()` method on the object.

```
>>> with Client(url=SERVER_URL) as client:
...     User('new_user', 'password').create()
```

Create a Database

```
>>> with Client(url=SERVER_URL) as client:
...     db = Database('db_name')
...     client.create(db)
...     print(db.tables)
{}
```

Create a Table

```
>>> TABLE_CREDENTIALS = 'hdfs://path/to/data.parq'
>>> columns = [
>>>    ColumnDefinition('x', type="REAL", bounds=(-3.0, 3.0)),
>>>    ColumnDefinition('y', type="BOOL"),
>>> ]
>>> with Client(url=SERVER_URL) as client:
...     tbl = Table(
...         'tbl_name',
...         columns=columns,
...         credentials=TABLE_CREDENTIALS,
...         database=db,
...     )
...     tbl.create()
```

The Client class

class leapyear.client.**Client**(*url='http://localhost:4401'*, *username=None*, *password=None*, ***, *authenticate=<function ly_login>*, *default_analysis_caching=True*, *default_allow_max_budget_allocation=True*, *public_key_auth=False*)

A class that wraps a connection to a LeapYear system.

__init__(*url='http://localhost:4401'*, *username=None*, *password=None*, ***, *authenticate=<function ly_login>*, *default_analysis_caching=True*, *default_allow_max_budget_allocation=True*, *public_key_auth=False*)

Initialize a Client object.

Parameters

- **url** (str) – The URL of LeapYear Core. Automatically set to the value of the *LY_API_URL* environment variable or *http://localhost:4401* if not specified.

- **username** (Optional[str]) – Optional username to pass to the *authenticate* parameter
- **password** (Optional[str]) – Optional password to pass to the *authenticate* parameter
- **authenticate** (AuthenticateFn) – A callback that should log in the user and return a LeapYear token. By default, logs in with the username and password of a LeapYear user, if username and password are provided. Otherwise, uses the token in the LY_JWT environment variable.
- **default_analysis_caching** (bool) – Whether to cache analysis results by default.
- **default_allow_max_budget_allocation** (bool) – Whether to allow running analyses on data sets that will automatically consume near the maximum privacy exposure per computation. If enabled, computations on small data sets that use the maximum privacy exposure will be blocked by default. This behavior can be overwritten at the computation level.
- **public_key_auth** (bool) – Use public key auth to authenticate requests to the LeapYear API. Requires setup via these instructions: https://guides.leapyear.io/docs/api-access

Raises

- **AuthenticationError** – when login fails.
- ***InvalidURL*** – when the url parameter is formatted incorrectly.

close()
Close the LeapYear connection.

Return type None

logout()
Logout and prevent any further actions using the current token.

Return type None

clear_analysis_cache()
Clear all analyses from the cache.

Return type None

clear_all_caches()
Clear all of the caches.

Return type None

count_analysis_cache()
Return the number of analyses in the analysis cache.

Return type int

unpersist_all_relations()
Clear all cached datasets.

Return type None

property logger
Get the Logger object for the client.

Return type Logger

property url
Get the URL of the connection.

Return type str

property username
Get the user currently logged in the server.

Return type str

property connected
Check if the client can successfully connect + authenticate to the LeapYear system.

Return type bool

property status
Get the status of the Client's connection to the LeapYear system.

Return type ClientStatus

create(*obj*, *ignore_if_exists=False*, *drop_if_exists=False*)
Create an object on the LeapYear server.

Return type None

create_async(*obj*)
Create an object on the LeapYear server asynchronously.

Return type *AsyncJob*

update(*obj*, ***kwargs*)
Update an object on the LeapYear server.

Return type None

drop(*obj*, *ignore_missing=False*)
Drop an object on the LeapYear server.

Return type None

property privacy_profiles
Get all privacy profiles available on the server.

Return type Mapping[str, PrivacyProfile]

property databases
Get all databases available on the server.

Return type Mapping[str, Database]

property current_user
Get the currently logged in user.

Return type User

property users
Get users available on the server.

Return type Mapping[str, User]

property groups
Get groups available on the server.

Return type Mapping[str, Group]

property jobs
Get running jobs on the server.

Return type List[AsyncJobInformation]

property recent_finished_jobs
Get the most recent finished jobs on the server.

Return type List[AsyncJobInformation]

1.4.2 Module leapyear.admin

Administrative objects for LeapYear.

Database class

class leapyear.admin.**Database**(*name*, *, *description=None*, *privacy_profile=None*, *privacy_limit=None*, *db_id=None*)
Database object.

classmethod all()
Get all databases.

Returns Iterator over all of the databases available on the server.

Return type all_databases

property id
Get the ID.

Return type int

property tables
Get the tables of the database.

Return type Mapping[str, ForwardRef]

property views
Get the views of the database.

This property can only be seen by admins

Return type Mapping[str, ForwardRef]

property privacy_params
Get the privacy parameter of the database.

Return type PrivacyProfileParams

property description
Get the database's description.

Return type Optional[str]

property privacy_profile
Get the database's Privacy Profile.

Return type str

set_privacy_profile(*privacy_profile*)
Set the database's privacy profile asynchronously.

Parameters **privacy_profile** (PrivacyProfile) – The new privacy profile.

Example

```
>>> db = c.databases["db1"]
>>> pp = c.privacy_profiles["Custom profile 1"]
>>> db.set_privacy_profile(pp)
```

Return type None

get_privacy_limit()
Get the database's privacy limit.

Return type PrivacyLimit

set_privacy_limit(*privacy_limit*)
Set the database's privacy limit.

Return type None

load()
Load the database.

create(*, *ignore_if_exists=False*)
Create the database.

Return type Database

drop(*, *ignore_missing=False*)
Drop the database.

get_access(*subject=None*)
Get the access level of the given subject.

Parameters **subject** (Union[User, Group, None]) – A User or Group object. If none is provided, use the currently logged in user.

Return type DatabaseAccessType

set_access(*subject*, *access*)
Grant the given access level to a subject.

Parameters

- **subject** (Union[User, Group]) – A User or Group object.
- **access** (ForwardRef) – The access level to grant.

Return type None

property name
Get the name.

Return type str

Table class

class leapyear.admin.**Table**(*name*, *, *database*, *columns=None*, *credentials=None*, *description=None*, *public=None*, *table_id=None*, *watch_folder=None*, ***kwargs*)

Table object.

__init__(*name*, *, *database*, *columns=None*, *credentials=None*, *description=None*, *public=None*, *table_id=None*, *watch_folder=None*, ***kwargs*)

Initialize a Table object.

Parameters

- **name** – The table name.
- **columns** – The columns to create the Table with. If no columns provided, the schema will be auto detected from the data.
- **credentials** – The credentials to the first data slice to be added to the Table.
- **description** – The table's description.
- **database** – The database this table belongs to.
- **public** – Whether this table should be a public table.
- **watch_folder** – When True, the 'credentials' parameter should point to a directory of parquet files that will be watched for automatic data slice uploads. Only applicable for table creation.

property id

Get the ID.

Return type int

property status

Get the status of the table.

Return type str

property status_with_error

Get the status of the table and potentially the error information.

Return type TableStatus

property columns

Get the columns of the table.

Return type Mapping[str, ForwardRef]

property description

Get the table's description.

property database

Get the database the table belongs to.

Return type str

property public

Identify whether the table is a public table.

Return type bool

get_privacy_limit()

Show the privacy limit associated with the table.

Returns a value of *None* when the table is public.

set_privacy_limit(*privacy_limit*)
Set the privacy limit associated with the table.

Throws an error when the table is public.

Return type None

property privacy_spent
Show the privacy spent for the current user on this table as a percentage.

Returns the privacy spent () associated with all the information disclosed so far by the LeapYear platform to the current user working with this table. The value is represented as a percentage of the privacy limit (0, 10, 20, ... 100) set by the administrator. The value can exceed 100% if the admin forcibly lowers the privacy limit below the current user's privacy spent. No queries can be run on a table where the privacy spent is at or above 100%.

If the table is public, returns *None* instead.

Returns Privacy exposure, expressed as a percentage of the limit.

Return type float

Examples

1. Review the current level of privacy spent.

```
>>> from leapyear.admin import Database, Table
>>> db = Database('db')
>>> t = Table('table', database=db)
>>> print(t.privacy_spent)
50
```

get_user_privacy_spent(*user*)
Show the privacy spent for a user on this table.

Returns the privacy spent (), as a float, associated with all the information disclosed so far by the LeapYear platform to a user working with this table, and the privacy limit as an (,) pair in a PrivacyLimit object.

Returns *None* instead, if the table is public.

This method is only available to authorized administrators, or to a user attempting to retreive their own privacy spent.

set_user_privacy_limit(*user*, *privacy_limit*)
Allow the administrator to set the privacy limit for a user on this table.

Sets the privacy limit as a (,) pair in a PrivacyLimit object for the user, on this table, that is considered acceptable by the administrator. If this method is not called, the user uses the privacy limit from the table.

If this is called with a public table, nothing happens.

This method is only available to authorized users with system admin privileges.

Return type None

load()
Load the table.

create_async()
Create the table asynchronously.

Return type *AsyncJob*

drop(*, *ignore_missing=False*)
Drop the table.

set_all_columns_access(*subject*, *access*)
Set the given access for all columns in the table.

If the table is public, the only legal access levels are full access and no access. Setting any other value will result in an error.

property slices
Show Data Slices for the table.

Return type List[ForwardRef]

add_data_slice(**args*, ***kwargs*)
Add a data slice like add_data_slice_async, except runs synchronously.

Return type None

add_data_slice_async(*file_credentials*, *, *update_column_bounds=False*)
Add a file to the list of data slices of the table.

Return type *AsyncJob*

create(*, *ignore_if_exists=False*)
Create the object synchronously.

Functionally equivalent to `.create_async().wait(max_timeout_sec=None).`

Return type AsyncCreateable

property name
Get the name.

Return type str

ColumnDefinition class

class leapyear.admin.**ColumnDefinition**(*name*, *, *type*, *bounds=None*, *nullable=False*, *description=None*, *infer_bounds=False*)
The definition of a column for creating a Table with an explicit schema.

Example usage:

```
>>> table = Table(
...     columns=[ColumnDefinition("col1", type="INT", bounds=(0, 10))],
...     ...
... )
>>> table.create()
```

Changing values in a ColumnDefinition has no effect after a table is created. See the TableColumn documentation for functions to update column attributes after creating a table.

name
str

type
ColumnType

bounds
ColumnBounds

nullable
bool

description
str | None

infer_bounds
bool

__new__(**kwargs)
Create and return a new object. See help(type) for accurate signature.

TableColumn class

class leapyear.admin.**TableColumn**(*, *database*, *table*, *id*, *name*, *type*, *bounds*, *nullable*, *description*)

A column in a table.

property id
Get the id of the column.

Return type int

property table
Get the table that the column belongs to.

Return type str

property database
Get the database that the column belongs to.

Return type str

property type
Get the type of the column.

Return type ColumnType

property bounds
Get the bounds of the column.

Return type Union[None, Tuple[int, int], Tuple[float, float], Tuple[date, date], Tuple[datetime, datetime], Set[str]]

property nullable
Get the nullability of the column.

Return type bool

property description
Get the description of the column.

Return type str

update(**kwargs)
Update the Column's type, bounds, or nullable.

All of the parameters are optional. If anything is not provided, it's left unchanged.

Parameters

- **type** (*Union[ColumnType, str]*) –
- **bounds** (*ColumnBounds*) –

- **nullable** (*bool*) –
- **infer_bounds** (*bool*) –

Return type *AsyncJob*

set_description(*description*)
Set the description of the Column.

Return type None

get_access(*subject=None*)
Get the access level of the given subject.

Parameters **subject** – A User or Group object. If none is provided, use the currently logged in user.

set_access(*subject*, *access*)
Grant the given access level to a subject.

If this is a column of a public table, only Full Access and No Access are legal values. Setting any other value will result in an error.

Parameters

- **subject** (Union[User, Group]) – A User or Group object.
- **access** (ForwardRef) – The access level to grant.

Return type None

class leapyear.admin.**ColumnType**(*value*)
A column type.

BOOL = 'BOOL'
A BOOL column has no bounds.

INT = 'INT'
An INT column whose bounds should be a `(int, int)` pair.

REAL = 'REAL'
A REAL column whose bounds should be a `(float, float)` pair.

FACTOR = 'FACTOR'
A FACTOR column whose bounds should be a list of strings.

TEXT = 'TEXT'
A TEXT column has no bounds.

DATE = 'DATE'
A DATE column whose bounds should be a `(datetime.date, datetime.date)` pair, containing dates of the form `1970-01-31`.

DATETIME = 'DATETIME'
A DATETIME column whose bounds should be a `(datetime.datetime, datetime.datetime)` pair, containing datetimes of the form `1970-01-31T000000`.

ID = 'ID'
An ID column has no bounds.

leapyear.admin.**ColumnBounds**
A type alias representing the union of all possible column bounds described in *ColumnType*

View class

class leapyear.admin.**View**(*name*, *, *database*, *dataset*, *num_partitions=1*, *partitioning_columns=[]*, *sort_within_partitions_by_columns=[]*, *nominal_partitioning_columns=[]*, *description=None*, ***kwargs*)

View object.

A view is a dataset that can be persisted on disk (materialized), across restarts of the LeapYear application. Analysts referencing a materialized view will be using the dataset that is on disk, instead of re-calculating any transformations defined on the dataset.

A guide on how to use views can be found here.

Analysts should load views either from `Database.views` or using the `database.view` notation; for example:

```
>>> db = client.databases['db1']
>>> view1 = db.views['view1']
>>> ds1 = DataSet.from_view(view1)
```

```
>>> ds2 = DataSet.from_view('db1.view1')
```

Parameters

- **name** (str) – The view's name. Views must have unique names, including de-materialized views. View names cannot include any of these characters: `,;{}()="`, or newlines (`\n`), or tabs (`\t`)
- **database** (Union[str, Database]) – The database that the view belongs to. This should be the database that the tables referenced in the DataSet belong to.
- **dataset** (ForwardRef) – The DataSet that will be stored as a view.
- **num_partitions** – The number of partitions that the view will be split into. This will only be used if *partitioning_columns* is also set.
- **partitioning_columns** – The columns by which to bucket (cluster) the view into partitions. This must be used with *num_partitions*. The view will have *num_partitions* number of partitions, and records with the same values for the *partitioning_columns* will be in the same partition.
- **sort_within_partitions_by_columns** – The columns used to sort rows within each partition.
- **nominal_partitioning_columns** – The columns by which to partition the view. This should be used by itself, without any other partition parameters.
- **description** (Optional[str]) – Description of the view.

property database

Get the database associated to the view.

Return type str

property description

Get the description associated to the view.

Return type str

dematerialize()

Dematerialize the view.

This is the preferred method to free disk space used by a view.

load()
Load the view.

create_async()
Create the view asynchronously.

Return type AsyncCreateJob

drop(*, *ignore_missing=False*)
Drop (and unregister) the view.

Admins should NOT drop a view unless they wish to also discard the entries in the analysis cache associated with that view. Instead, admins should use the *dematerialize* method.

create(*, *ignore_if_exists=False*)
Create the object synchronously.

Functionally equivalent to `.create_async().wait(max_timeout_sec=None).`

Return type AsyncCreateable

property name
Get the name.

Return type str

User class

class leapyear.admin.**User**(*username*, *password=None*, *, *is_root=None*, *enabled=None*, *user_id=None*, *subj_id=None*)
User object.

classmethod all()
All Users.

Returns All users on the LeapYear server.

Return type Iterator[*User*]

property id
Get the ID.

Return type int

property subj_id
Get the subject ID.

Return type int

property username
Get the username.

Return type str

property is_root
Whether the user is a root user.

Return type bool

property enabled
Whether the user is enabled.

Return type bool

property groups
Get the groups of a user.

Returns All groups of the user on the LeapYear server.

Return type List[Group]

load()
Load the information for the user.

Return type User

create(*, *ignore_if_exists=False*)
Create the user.

Return type User

update(*, *password=None*, *enabled=None*)
Update the user.

Return type User

property name
Get the name.

Return type str

Privacy Profile class

class leapyear.admin.**PrivacyProfile**(*name*, *, *params=None*, *hidden=None*, *verified=None*, *description=None*, *profile_id=None*)
PrivacyProfile object.

classmethod all()
Get all privacy profiles.

Return type Iterator[ForwardRef]

property id
Get the ID.

Return type int

property description
Get the privacy profile description.

Return type str

property hidden
Get whether the profile is hidden in the Data Manager.

Return type bool

property verified
Get whether the profile is verified.

Return type bool

property params
Get the parameters of the privacy profile.

Return type PrivacyProfileParams

load()
Load the privacy profile.

create(*, *ignore_if_exists=False*)
Create the privacy profile.

Return type PrivacyProfile

update(*params=None*, *hidden=None*)
Update the privacy profile's params.

Parameters

- **params** – The parameters to be updated.
- **hidden** – Whether or not the privacy profile should be hidden in Data Manager.

Permission objects

class leapyear.admin.**DatabaseAccessType**(*value*)
AccessType for Databases.

NO_ACCESS_TO_DB = 'NO_ACCESS_TO_DB'
Prevents user from accessing database

SHOW_DATABASE = 'SHOW_DATABASE'
Allows a user to see this database and the tables it contains, including their public metadata

ADMINISTER_DATABASE = 'ADMINISTER_DATABASE'
Allows a user to administer this database - e.g. add data sources, grant user access

class leapyear.admin.**ColumnAccessType**(*value*)
AccessType for Columns.

NO_ACCESS = 'NO_ACCESS'
Prevents user from accessing column

COMPUTE = 'COMPUTE'
Allows a user to run randomized computations

FULL_ACCESS = 'FULL_ACCESS'
Allows a user to run randomized computations and view and retrieve raw data

COMPARE = 'COMPARE'

1.4.3 Module leapyear.admin.grants

Convenience functions for retrieving grants on resources.

Access Summaries

class leapyear.admin.grants.**DatabaseAccess**(*subject: Union[leapyear.admin.user.User, leapyear.admin.group.Group]*, *database: leapyear.admin.database.Database*, *access: leapyear.admin.database.DatabaseAccessType*)
Access between a subject and a database.

property subject
Subject

property database
Database

property access
DatabaseAccessType

class leapyear.admin.grants.**TableAccess**(*subject: Union[leapyear.admin.user.User, leapyear.admin.group.Group]*, *table: leapyear.admin.table.Table*, *columns: Mapping[str, leapyear.admin.table.ColumnAccessType]*)

Access between a subject and a table.

property subject
Subject

property table
Table

property columns
Mapping[str, ColumnAccessType]

Functions

leapyear.admin.grants.**all_access_on_database**(*db*)

Fetch the database access for all users or groups.

Examples

```
>>> db = client.databases['db']
>>> for subject, access_db, access in all_grants_on_database(db):
...     assert isinstance(subject, User) or isinstance(subject, Group)
...     assert access_db == db
...     assert isinstance(access, DatabaseAccessType)
...
```

Parameters **db** (Database) –

Returns

Return type List[*DatabaseAccess*]

leapyear.admin.grants.**all_access_on_table**(*table*)

Fetch the column access for all columns in the given table for all users or groups.

Examples

```
>>> table = client.databases['db'].tables['table']
>>> for subject, access_table, columns in all_access_on_table(table):
...     assert isinstance(subject, User) or isinstance(subject, Group)
...     assert access_table == table
...     for column_name, column_access in columns.items():
...         assert isinstance(column_name, str)
...         assert isinstance(column_access, ColumnAccessType)
...
```

Parameters **table** (Table) –

Returns

Return type List[*TableAccess*]

leapyear.admin.grants.**all_database_accesses_for_subject**(*subject*)

Fetch the access of all databases for the given subject.

Examples

```
>>> user = client.users['user']
>>> for subject, db, access in all_database_accesses_for_subject(user):
...     assert subject == user
...     assert isinstance(db, Database)
...     assert isinstance(access, DatabaseAccessType)
...
```

Parameters **subject** (Union[User, Group]) –

Returns

Return type List[*DatabaseAccess*]

1.4.4 Module leapyear.jobs

Classes for managing asynchronous jobs in LeapYear Core.

Inspecting Job status

class leapyear.jobs.**AsyncJob**(*async_job_id*, *, *on_success=None*)

A helper for polling the result of an asynchronous job.

Represents an accessor for an asynchronous job on the LeapYear server. It contains a unique job_id for a specific job and methods to check the results of or cancel the job at a later time.

class leapyear.jobs.**AsyncJobStatus**(*status: leapyear.jobs.AsyncJobState*, *result: Optional[Union[leapyear._tidl.protocol.query.QueryResult, leapyear.exceptions.APIError]]*, *start_time: datetime.datetime*, *end_time: Optional[datetime.datetime]*)

Result of checking the status of an asynchronous job.

status

The current status of the job.

Return type AsyncJobState

result

None if the job is still running, otherwise either the JobResult or a APIError error.

Return type Optional[ResultOrError]

start_time

The time the job was started.

Return type datetime

end_time

The time the job finished, or None if the job is still running.

Return type Optional[datetime]

property elapsed_time
Return a timedelta representing the amount of time taken by the analysis.

Return type timedelta

class leapyear.jobs.**AsyncJobState**(*value*)
The current state of a job.

AsyncJobStateRunning = 'AsyncJobStateRunning'

AsyncJobStateFinished = 'AsyncJobStateFinished'

AsyncJobStateFailed = 'AsyncJobStateFailed'

AsyncJobStateCancelled = 'AsyncJobStateCancelled'

1.4.5 Module leapyear.dataset

DataSet and Attribute.

DataSets combine a data source (Table) with data transformations which subsequent computations can be performed on. A data set has a schema which describes the types of attributes (columns) in the data source. Attributes can be manipulated as if they were built-in python types (int, float, bool, ...). DataSet also provides the following lazy methods for transforming data:

- *project()*: select one or more attributes (columns) from the data source.
- *where()*: select rows that satisfy a filter condition.
- *union()*: combine two data sets with matching schemas.
- *split()*: create subsets whose size is a fraction of the total data size.
- *splits()*: yield all fractional partitions of the data set.
- stratified(): create subsets of the data with fixed attribute prevalence.
- *group_by()*: create aggregated views of the data set.
- *join()*: combine rows of two datasets where certain data elements match.
- *transform()*: apply a linear transformation to the specified data elements.

Further details for each transformation can be found in their respective documentation below. Transformations are lazy in the sense that they are not evaluated until a computation is executed; however, each transformation requires DataSet schema to be re-evaluated, which relies on a live connection to the LeapYear server.

Computations and machine learning analytics that process the data set are found in *leapyear.analytics*.

The examples below rely on a connection to LeapYear server, which can be established as follows:

```
>>> from leapyear import Client
>>> client = Client(url='http://ly-server:4401', username='admin', password='password
↪')
```

Examples

- Load a dataset and examine its schema:

```
>>> pds = DataSet.from_table('db.table')
>>> pds.schema
OrderedDict([
    ('attr1', Type(tag=BOOL, contents=())),
    ('attr2', Type(tag=INT, contents=(0, 20))),
    ('attr3', Type(tag=REAL, contents=(-1, 1))),
])
>>> pds.attributes
['attr1', 'attr2', 'attr3']
```

- Create a new attribute and find its type:

```
>>> pds['attr2_gt_10'] = pds['attr2'] > 10
>>> pds.attributes
['attr1', 'attr2', 'attr3', 'attr2_gt_10']
>>> pds.schema['attr2_gt_10']
Type(tag=BOOL, contents=())
```

- Select instances using a predicate:

```
>>> pds_positive_attr3 = pds.where(pds['attr3'] > 0)
>>> pds_positive_attr3.schema['attr3']
Type(tag=REAL, contents=(0, 1))
```

- Calculate the mean of a single attribute:

```
>>> import leapyear.analytics as analytics
>>> mean_analysis = analytics.mean(pds_positive_attr3['attr3'])
>>> mean_analysis
computation: MEAN(attr3 > 0)
attributes:
    attr3: db.table.attr3 (0 <= x <= 1)
>>> mean_analysis.run()  # run the computation on the LeapYear server.
0.02
```

DataSet class

class leapyear.dataset.**DataSet**(*relation*)

DataSet object.

property relation

Get the DataSet relation.

Return type Relation

property schema

Get the DataSet schema, including data types, values allowed.

Return type Schema

classmethod from_view(*view*)

Create a dataset from a view.

Parameters **view** (Union[str, View]) – The view or the name of the view on the LeapYear server.

Returns The dataset with a LeapYear view as its source.

Return type *DataSet*

classmethod from_table(*table*, *, *slices=None*, *all_slices=False*)
Create a dataset from a table.

Parameters

- **table** – The table or the name of the table on the LeapYear server.
- **slices** – A list of ranges of table slices to use.
- **all_slices** – Set to True to use all slices in the Table at the time this is run. Note that if a new slice has been added, this will create a DataSet with the new slice, causing analyses to miss the analysis cache when rerunning.

Returns The dataset with a LeapYear table as its source.

Return type *DataSet*

property attributes
Return the DataSet attributes.

Return type Iterable[Attribute]

get_attribute(*key*)
Select one attribute from the data set.

Parameters **key** (Union[str, Attribute]) – The item to project.

Returns The projected attribute.

Return type *Attribute*

drop_attributes(*keys*)
Drop Attributes.

Parameters **keys** (Iterable[str]) – A list of attribute names to drop.

Returns New DataSet without the specified attributes.

Return type *DataSet*

drop_attribute(*key*)
Drop an attribute.

Parameters **key** (str) – The attribute name to drop.

Returns New DataSet without the specified attribute.

Return type *DataSet*

with_attributes(*name_attrs*)
Return a new DataSet with additional attributes.

Parameters **name_attrs** (Mapping[str, Any]) – A dictionary associating new attribute names to expressions. Attribute expressions can include python literals and references to existing attributes.

Returns The DataSet with new attributes appended.

Return type *DataSet*

with_attribute(*name*, *attr*)
Return a new DataSet with an additional attribute.

Parameters

- **name** (str) – Name of the new attribute
- **attr** (Any) – Attribute expression, can include python literals and references to existing attributes.

Returns New DataSet with new attribute appended.

Return type *DataSet*

with_attributes_renamed(*rename*)
Rename attributes using a mapping.

Parameters **rename** (Mapping[str, str]) – Dictionary mapping old names to new names.

Returns New DataSet with renamed attributes.

Return type *DataSet*

Examples

1. Create new dataset ds2 with renamed columns 'ZipCode' and 'Name' which were named 'zip_code' and 'name' respectively. Similarly, rename attributes in another dataset ds3. Then, finally these datasets can be merged using union:

```
>>> ds2 = ds1.with_attributes_renamed({'zip_code':'ZipCode','name':'Name'})
>>> ds4 = ds3.with_attributes_renamed({'zipcode':'ZipCode','name_str':'Name'})
>>> ds4 = ds4.union(ds2)
```

with_attribute_renamed(*old_name*, *new_name*)
Rename an attribute.

Parameters

- **old_name** (str) – Old attribute name.
- **new_name** (str) – New attribute name.

Returns New DataSet with this attribute renamed.

Return type *DataSet*

map_attributes(*name_lambdas*)
Map attributes and create a new DataSet.

Parameters **name_lambdas** (Mapping[str, Callable[[Attribute], Attribute]]) – Mapping from attribute names to functions.

Returns New DataSet with mapped attributes.

Return type *DataSet*

map_attribute(*name*, *func*)
Map an attribute and create a new DataSet.

Parameters

- **name** (str) – The attribute name to map.
- **func** (Callable[[Attribute], Attribute]) – Function to convert the attribute.

Returns New DataSet with mapped attributes.

Return type *DataSet*

project(*new_keys*)
Select multiple attributes from the data set.

Projection (): Creates a new DataSet with only selected attributes from this DataSet.

Parameters **new_keys** (Iterable[str]) – The list of attributes.

Returns The new DataSet containing only the selected attributes.

Return type *DataSet*

select(**attrs*)
Create a new DataSet by selecting column names or Attributes.

Parameters **attrs** (Union[str, Attribute]) – Attribute name(s) or Attribute object(s), separated by comma.

Returns The new DataSet containing the selected attributes.

Return type *DataSet*

select_as(*mapping*)
Create a new DataSet by mapping from column names or Attributes to new names.

Parameters **mapping** (Mapping[str, Union[str, Attribute]]) – A dictionary associating new attribute names to expressions. Attribute expressions can include python literals and references to existing attributes.

Returns The new DataSet containing the selected attributes.

Return type *DataSet*

where(*clause*)
Select/filter rows using a filter expression.

Selection (): LeapYear's where clause creates a new DataSet based on the filter condition. Its schema may include smaller domain of possible values.

Parameters **clause** (Attribute) – A filter Attribute: a single attribute of type *BOOL*.

Returns A filtered DataSet.

Return type *DataSet*

union(*other*, *distinct=False*)
Union or concatenate datasets.

Union (): Concatenates data sets with matching schema.

Parameters

- **other** (ForwardRef) – The dataset to union with. Note: schema of *other* must match that of *self*, including the order of the attributes. The order of attributes can be aligned like so:

```
>>> ds2 = ds2[list(ds1.schema.keys())]
>>> ds_union = ds1.union(ds2)
```

- **distinct** (bool) – Remove duplicate rows.

Returns A combined dataset.

Return type *DataSet*

join(*other*, *on*, *right_on=None*, *join_type='inner'*, *unique_left_keys=False*, *unique_right_keys=False*, *left_suffix=''*, *right_suffix=''*, ***kwargs*)

Combine two DataSets by joining using a key, as in SQL JOIN statement.

If *right_k* (or *left_k*) is specified, the right (or left) data set will include no more than *k* rows for each matching row in the left (or right) data set.

Parameters

- **other** (ForwardRef) – The other DataSet to join with.
- **on** (Union[str, List[str]]) – The key(s) to join on. If using suffixes, the suffix must NOT be appended by the caller.
- **right_on** (Union[str, List[str], None]) – The key(s) on the right table, if different than those in *on*. None if both tables have the same key names. If using suffixes, the suffix must NOT be appended by the caller.
- **join_type** (str) – LeapYear supports a variety of joins listed in the left column in the table below. Any value of `join_type` from the right column will use that particular join. If no value is specified, an inner join will be used. A `left_outer_public` join can be run only on a right public table.

Join	join_type
Inner	`"inner"` (default)
Outer	`"outer"`, `"full"`, `"full_outer"`, or `"fullouter"`
Left	`"left"`, `"left_outer"`, or `"leftouter"`
Right	`"right"`, `"right_outer"`, or `"rightouter"`
Left Antijoin	`"left_anti"` or `"leftanti"`
Left Semijoin	`"left_semi"` or `"leftsemi"`
Left Outer Public	`"left_outer_public"`

- **unique_left_keys** (bool) – If the left DataSet is known to have unique keys, setting this to True will run an optimized join algorithms. Warning: Setting this to True if the keys are not unique will cause data loss!
- **unique_right_keys** (bool) – If the right DataSet is known to have unique keys, setting this to True will run an optimized join algorithms. Warning: Setting this to True if the keys are not unique will cause some data rows to not show up in the output DataSet!
- **left_suffix** (str) – Optional suffix to append to the column names of the left DataSet.
- **right_suffix** (str) – Optional suffix to append to the column names of the right DataSet.
- **-> cache** (*kwargs*) – Optional StorageLevel for persisting intermediate datasets for performance enhancement. The meaning of these values is documented in: https://spark.apache.org/docs/2.4.5/programming-guide.html#rdd-persistence

Returns The joined DataSet.

Return type *DataSet*

Examples

1. Joining two datasets on a common key:

```
>>> ds1 = example_ds1.project(['key', 'col1'])
>>> ds2 = example_ds2.project(['key', 'col2'])
>>> ds = ds1.join(ds2, 'key')
>>> list(ds.schema.keys())
['key', 'col1', 'col2']
```

2. Joining two datasets on a single key but with a different name on the right:

```
>>> ds1 = example_ds1.project(['key1', 'col1'])
>>> ds2 = example_ds2.project(['key2', 'col2'])
>>> ds = ds1.join(ds2, 'key1', right_on='key2')
>>> list(ds.schema.keys())
['key1', 'col1', 'key2', 'col2']
```

3. Joining when a column is duplicated is an error:

```
>>> ds1 = example_ds1.project(['key', 'col1', 'col3'])
>>> ds2 = example_ds2.project(['key', 'col2', 'col3'])
>>> ds = ds1.join(ds2, 'key')
APIError: Invalid schema: repeated aliases: col3
```

4. Joining when the key is missing from one relation: Fails because in this case a right key has to be specified.

```
>>> ds1 = example_ds1.project(['key1', 'col1'])
>>> ds2 = example_ds2.project(['key2', 'col2'])
>>> ds = ds1.join(ds2, 'key1')
APIError: Error parsing scope, missing variable declaration for `key1`
```

5. Joining with multiple keys:

```
>>> ds1 = example_ds1.project(['key1_1', 'key2_1', 'col1'])
>>> ds2 = example_ds2.project(['key1_2', 'key2_2', 'col2'])
>>> ds = ds1.join(ds2, ['key1_1', 'key2_1'], right_on=['key1_2', 'key2_2'])
>>> list(ds.schema.keys())
['key1_1', 'key2_1', 'col1', 'key1_2', 'key2_2', 'col2']
```

6. Joining with different number of keys results in an error:

```
>>> ds1 = example_ds1.project(['key1_1', 'key2_1', 'col1'])
>>> ds2 = example_ds2.project(['key1_2', 'key2_2', 'col2'])
>>> ds = ds1.join(ds2, ['key1_1', 'key2_1'], right_on='key1_2')
APIError: Invalid schema: join key length mismatch ...
```

7. Joining with specifying that the keys are unique:

```
>>> ds1 = example_ds1.project(['key1', 'col1'])
>>> ds2 = example_ds2.project(['key1', 'col2'])
>>> ds = ds1.join(ds2, 'key1', unique_left_keys=True, unique_right_keys=True)
>>> list(ds.schema.keys())
['key1', 'col1', 'col2']
```

8. Different join types:

```
>>> ds1 = example_ds1.project(['key1', 'col1'])
>>> ds2 = example_ds2.project(['key1', 'col2'])
>>> ds = ds1.join(ds2, 'key1', join_type='outer')
>>> list(ds.schema.keys())
['key1', 'col1', 'col2']
```

9. Left semi join:

```
>>> ds1 = example_ds1.project(['key1', 'col1'])
>>> ds2 = example_ds2.project(['key1', 'col2'])
>>> ds = ds1.join(ds2, 'key1', join_type='left_semi')
>>> list(ds.schema.keys())
['key1', 'col1']
```

10. Joining when the nulability of keys is different:

```
>>> ds1 = example_ds1.select([col('key1').decode({0: 0}).alias('nkey1'), 'col1
↪'])
>>> ds2 = example_ds2.project(['key2', 'col2'])
>>> ds = ds1.join(ds2, 'nkey', right_on='key2')
>>> list(ds.schema.keys())
['nkey1', 'col1', 'key2', 'col2']
```

11. Joining when keys have different but coercible types:

```
>>> realKey1 = col('intKey1').as_real().alias('realKey1')
>>> ds1 = example_ds1.select([realKey1, 'col1'])
>>> ds2 = example_ds2.project(['intKey2', 'col2'])
>>> ds = ds1.join(ds2, 'realKey1', right_on='intKey2')
>>> list(ds.schema.keys())
['intKey1', 'col1', 'intKey2', 'col2']
```

12. Joining when keys are factors (upcasted to common type):

```
>>> keyFactor1 = col('key1').decode({k: 'A' for k in range(10)}).
>>>                 as_factor().alias('keyFactor1')
>>> keyFactor2 = col('key2').decode({k: 'B' for k in range(10)}).
>>>                 as_factor().alias('keyFactor2')
>>> ds1 = example_ds1.select([keyFactor1, 'col1'])
>>> ds2 = example_ds2.select([keyFactor2, 'col2'])
>>> ds = ds1.join(ds2, 'keyFactor1', right_on='keyFactor2')
>>> list(ds.schema.keys())
['key1', 'col1', 'key2', 'col2']
```

13. Joining when the keys have mismatched types is an error (e.g. factor and bool):

```
>>> keyFactor1 = col('key1').as_factor().alias('keyFactor1')
>>> ds1 = example_ds1.project(['key1', 'col1'])
>>> ds2 = example_ds2.project(['key2', 'col2'])
>>> ds = ds1.join(ds3, 'keyFactor1', right_on='key2')
APIError: Invalid schema: join column type mismatch ...
```

14. Joining with suffixes to disambiguate column names:

```
>>> ds1 = example_ds1.project(['key', 'col1'])
>>> ds2 = example_ds2.project(['key', 'col2'])
>>> ds = ds1.join(ds2, 'key', left_suffix='_l', right_suffix='_r')
>>> list(ds.schema.keys())
['key_l', 'col1_l', 'key_r', 'col2_r']
```

15. Joining with only left suffixes to disambiguate column names:

```
>>> ds1 = example_ds1.project(['key', 'col1'])
>>> ds2 = example_ds2.project(['key', 'col2'])
>>> ds = ds1.join(ds2, 'key', left_suffix='_l')
>>> list(ds.schema.keys())
['key_l', 'col1_l', 'key', 'col2']
```

16. Joining with only right suffixes to disambiguate column names:

```
>>> ds1 = example_ds1.project(['key', 'col1'])
>>> ds2 = example_ds2.project(['key', 'col2'])
>>> ds = ds1.join(ds2, 'key', right_suffix='_r')
>>> list(ds.schema.keys())
['key', 'col1', 'key_r', 'col2_r']
```

17. Joining with specific cache level for intermediate caches:

```
>>> ds = ds1.join(ds2, on="key", cache=StorageLevel.DISK_ONLY, n_partitions=1)
```

prepare_join (*join_on*, *k*, *n_partitions*)
Prepare DataSet for join.

Parameters

- **join_on** (Union[str, List[str]]) – The key(s) to join on. If using suffixes, the suffix must NOT be appended by the caller.
- **k** (int) – The k value to filter the DataSet.
- **n_partitions** (int) – Number of partitions to repartition the DataSet.

Return type ForwardRef

unpersist_join_cache (*other*, *on*, *right_on=None*, *join_type='inner'*, *unique_left_keys=False*, *unique_right_keys=False*, *left_suffix=''*, *right_suffix=''*, ***kwargs*)
Unpersist intermediate caches used by .join().

Parameters

- **NOTE** (--) –
- **other** (ForwardRef) – The other DataSet to join with.
- **on** (Union[str, List[str]]) – The key(s) to join on. If using suffixes, the suffix must NOT be appended by the caller.
- **right_on** (Union[str, List[str], None]) – The key(s) on the right table, if different than those in *on*. None if both tables have the same key names. If using suffixes, the suffix must NOT be appended by the caller.
- **join_type** (str) – LeapYear supports a variety of joins listed in the left column in the table below. Any value of join_type from the right column will use that particular join. If no value is specified, an inner join will be used. A left_outer_public join can be run only on a right public table.

Join	join_type
Inner	"inner" (default)
Outer	"outer", "full", "full_outer", or "fullouter"
Left	"left", "left_outer", or "leftouter"
Right	"right", "right_outer", or "rightouter"
Left Antijoin	"left_anti" or "leftanti"
Left Semijoin	"left_semi" or "leftsemi"
Left Outer Public	"left_outer_public"

- **unique_left_keys** (bool) – If the left DataSet is known to have unique keys, setting this to True will run an optimized join algorithms. Warning: Setting this to True if the keys are not unique will cause data loss!
- **unique_right_keys** (bool) – If the right DataSet is known to have unique keys, setting this to True will run an optimized join algorithms. Warning: Setting this to True if the keys are not unique will cause some data rows to not show up in the output DataSet!
- **left_suffix** (str) – Optional suffix to append to the column names of the left DataSet.
- **right_suffix** (str) – Optional suffix to append to the column names of the right DataSet.

Return type ForwardRef

group_by (**grouping*)

Aggregate data by a categorical column(s).

Parameters **grouping** (Union[str, Attribute]) – The attribute or attributes to group by, separated by comma.

Returns A new *GroupedData* object with groupings as specified. It can be used with *agg* function to create a DataSet with derived aggregate attributes, see examples below.

Return type *GroupedData*

Examples

1. Group by multiple columns ('col1' and 'col2') in Dataset ds and aggregate 'col3' and 'col4':

```
>>> ds_group = ds.group_by('col1', 'col2').agg((['col3'], 'count'), (['col4'],
↪ 'count'))
```

2. Group by single column 'col1' in Dataset and compute aggregate of 'col3' and 'col4':

```
>>> ds_group = ds.group_by('col1').agg((['col3'], 'max'), (['col4'], 'mean'))
```

split (*index*, *proportions*, *complement=False*)

Split and select one partition of the dataset.

Selection (): Splits are specified by proportions.

Parameters

- **index** (int) – The split number
- **proportions** (List[int]) – The proportions to split the dataset by, represented as a list of integers. For example, [1,1,2] will split into 3 datasets with 1/4, 1/4 and 1/2 of the data in each, respectively.
- **complement** (bool) – If True, returns a DataSet which is the complement of the split (e.g. all rows not in the split). Default: False.

Returns The ith-partition of the dataset.

Return type *DataSet*

splits (*proportions*)

Split the dataset and return an iterator over the resulting partitions.

Selection (): Splits are specified by proportions.

Parameters **proportions** (List[int]) – The proportions to split the dataset by, represented as a list of integers. For example, [1,1,2] will split into 3 datasets with 1/4, 1/4 and 1/2 of the data in each, respectively.

Returns Iterator over the DataSet objects representing partitions.

Return type Iterator[*DataSet*]

Examples

1. Create 80/20 split of a Dataset ds1:

```
>>> traintest, holdout = ds1.splits((8,2))
```

stratified_split (*index*, *proportions*, *stratified*, *complement=False*)

Split by stratifying a categorical attribute.

Selection (): Each split will contain approximately the same proportion of values from each category.

As an example, for Boolean stratification, each split will contain the same proportion of True/False values.

Parameters

- **index** (int) – The split number

- **proportions** (List[int]) – The proportions to split the dataset by, represented as a list of integers. For example, [1,1,2] will split into 3 datasets with 1/4, 1/4 and 1/2 of the data in each, respectively.
- **stratified** (str) – The column to stratify against. Must be Boolean or Factor. Must not be nullable.
- **complement** (bool) – If True, returns a DataSet which is the complement of the split (e.g. all rows not in the split). Default: False.

Returns The ith-partition of the dataset.

Return type *DataSet*

stratified_splits(*proportions*, *stratified*)

Split by stratifying a categorical attribute.

Selection (). For boolean stratification, each split will maintain the same proportion of True/False values.

Parameters

- **proportions** (List[int]) – The proportions to split the dataset by, represented as a list of integers. For example, [1,1,2] will split into 3 datasets with 1/4, 1/4 and 1/2 of the data in each, respectively.
- **stratified** (str) – The column to stratify against. Must be Boolean or Factor. Must not be nullable.

Returns Iterator over the DataSet objects representing partitions.

Return type Iterable[*DataSet*]

kfold(*n_folds=3*)

Split the dataset into train/test pairs using k-fold strategy.

Each fold can then be used as a validation set once while k-1 remaining folds form the training set. If the dataset has size N, each (train, test) pair will be sized N*(k-1)/k and N/k respectively.

Parameters **n_folds** (int) – Number of folds. Must be at least 2.

Returns Iterator over the k pairs of (train, test) partitions.

Return type Iterable[Tuple["DataSet", "DataSet"]]

stratified_kfold(*stratified*, *n_folds=3*)

Split the dataset into train/test pairs using k-fold stratified splits.

Each fold can then be used as a validation set once while k-1 remaining folds form the training set. If the dataset has size N, each (train, test) pair will be sized N*(k-1)/k and N/k respectively.

Parameters

- **stratified** (str) – The column to stratify against. Must be Boolean or Factor. Must not be nullable.
- **n_folds** (int) – Number of folds. Must be at least 2.

Returns The iterator over the k pairs of (train, test) partitions.

Return type Iterable[Tuple["DataSet", "DataSet"]]

rows(*limit=None*, *max_timeout_sec=None*)

Retrieve rows.

If the user has permission to do so, the function returns a generator of OrderedDict objects representing the attribute names and values from each row. The generator requires connection to the server over its entire lifetime.

Parameters

- **limit** (Optional[int]) – Maximum number of rows to output.
- **max_timeout_sec** (Optional[float]) – Specifies the maximum amount of time (in seconds) the user is willing to wait for a response. If set to None, wait will poll the server indefinitely. Defaults to None.

Returns The iterator over the rows of the input dataset, each row being represented as an OrderedDict objects mapping attribute names to their values in this row.

Return type Iterator[Mapping[str, Value]]

rows_pandas(*limit=None*)

Retrieve rows as a pandas DataFrame.

Parameters **limit** (Optional[int]) – Maximum number of rows to output.

Returns The rows of the input dataset.

Return type DataFrame

head(*n=10*)

Retrieve rows, if the user has permission to do so, see *rows()*.

Parameters **n** (int) – Maximum number of rows to output.

Returns The iterator over the rows of the input dataset, each row being represented as an OrderedDict objects mapping attribute names to their values in this row.

Return type Iterator[Mapping[str, Value]]

head_pandas(*n=10*)

Retrieve rows, if the user has permission to do so, see *rows_pandas()*.

Parameters **n** (int) – Maximum number of rows to output.

Returns The rows of the input dataset.

Return type DataFrame

example_rows()

Retrieve 10 rows of example data from the DataSet.

The returned data is based only on the public metadata. The generated data does not depend on the true data at all and should not be used for data inference.

The function requires an active connection to the LeapYear server.

Does not support TEXT attributes - consider dropping them using *drop_attributes()* before running *example_rows()*.

Returns The iterator over the rows of the generated dataset; each row is represented as an OrderedDict objects mapping attribute names to their generated values.

Return type Iterator[Mapping[str, Any]]

Example

```
>>> pds = DataSet.from_table('db.table')
>>> pds.example_rows()
```

To turn the data into a pandas DataFrame, use

```
>>> import pandas
>>> df = pandas.DataFrame.from_dict(pds.example_rows())
```

Alternatively, use *example_rows_pandas()*.

example_rows_pandas()
Retrieve 10 rows of example data from the DataSet. See *example_rows()*.

Returns The example rows.

Return type DataFrame

transform(*attrs*, *transformation*, *name*)
Apply linear transformation to a list of attributes based on a matrix.

Parameters

- **attrs** (List[Attribute]) – Expressions to use as input to the matrix multiplication, in order.
- **transformation** (List[List[float]]) – Matrix to use to define linear transformation via matrix multiplication - e.g. output of *leapyear.analytics.pca()*.
- **name** (str) – Common prefix for the name of the attributes to be created.

Returns DataSet with transformed attributes appended.

Return type *DataSet*

sample(*with_replacement*, *fraction*, *seed=None*)
Return a sampled subset of the rows.

Parameters

- **with_replacement** (bool) – Allow sampling rows with replacement, creating duplicated rows.
- **fraction** (float) – Fraction of rows to generate.
- **seed** (Optional[int]) – Optional seed of the random number generator. If not supplied, a random seed will be used.

Returns DataSet containing sampled subset of the rows.

Return type *DataSet*

distinct()
Return a DataSet that contains only the unique rows from this Dataset.

Return type ForwardRef

drop_duplicates(*subset=None*)
Return a DataSet with duplicates in the provided columns dropped.

If all columns are named or subset is None, this is equivalent to *distinct()*.

Parameters **subset** (Optional[List[str]]) – Subset of columns, or None if all columns should be used.

Return type ForwardRef

except_(*ds*)
Return a DataSet of rows in this DataSet but not in another DataSet.

Parameters **ds** (ForwardRef) – DataSet to compare with.

Returns DataSet containing rows in this DataSet but not in another DataSet.

Return type "DataSet"

difference(*ds*)
Return a DataSet of rows in this DataSet but not in another DataSet.

Parameters **ds** (ForwardRef) – DataSet to compare with.

Returns DataSet containing rows in this DataSet but not in another DataSet.

Return type "DataSet"

symmetric_difference(*ds*)
Return the symmetric difference of the two DataSets.

Parameters **ds** (ForwardRef) – DataSet to compare with.

Returns DataSet containing rows that are not in the intersection of the two DataSets.

Return type "DataSet"

intersect(*ds*)
Return intersection of the two DataSets.

Parameters **ds** (ForwardRef) – DataSet to compare with.

Returns DataSet containing rows that are in the intersection of the two DataSets.

Return type "DataSet"

order_by(**attrs*)
Order DataSet by the given expressions.

Parameters **attrs** (Union[str, Attribute]) – Attribute expressions to order by, separated by commas.

Returns DataSet sorted by the given expressions.

Return type "DataSet"

limit(*n*)
Limit the number of rows.

Parameters **n** (int) – Number of rows to limit to.

Returns DataSet filtered to the first n rows.

Return type "DataSet"

cache(*storageLevel=StorageLevel.MEMORY_AND_DISK*)
Cache a Dataset on disk on the server-side.

Parameters **storageLevel** (StorageLevel) – StorageLevel for persisting datasets. The meaning of these values is documented in: https://spark.apache.org/docs/2.4.5/programming-guide.html#rdd-persistence

Returns DataSet, which indicates to the system to lazily cache the DataSet

Return type "DataSet"

unpersist()
Immediately begins unpersisting the DataSet on the server-side.

Returns

Return type *None*

Example

Build a cache and unpersist it

```
>>> la.count_rows(ds.cache()).run()
>>> la.mean(ds["foo"]).run()          # this will hit the cache
>>> ds.unpersist()
>>> la.sum(ds["foo"]).run()           # this will no longer hit the cache
```

repartition(*numPartitions*, **attrs*)
Repartition a Dataset by hashing the columns.

Returns DataSet partitioned according to the specified parameters.

Return type "DataSet"

sortWithinPartitions(**attrs*)
Sorts Dataset rows by the provided columns within partitions, not globally.

Returns DataSet sorted within partitions.

Return type "DataSet"

replace(*col*, *replacement*)
Replace values matching keys in replacement map.

Parameters

- **col** (str) – The name of the column to work with.
- **replacement** (Mapping[Union[float, bool, int, str, datetime, date], Union[ForwardRef, Expression, float, bool, int, str, datetime, date]]) – A mapping associating each value to be replaced – with an expression it should be replaced with.

Returns New Dataset where specified values in a given column are replaced according to the replacement map.

Return type "DataSet"

fill(*col*, *value*)
Fill missing values in a given column.

Parameters

- **col** – The name of the column to work with.
- **value** (Union[ForwardRef, Expression, float, bool, int, str, datetime, date]) – Expression to fill NULL values of the column with.

Returns A new DataSet where NULL values in a given column are replaced with the specified expression.

Return type "DataSet"

drop(**cols*, *how='any'*)
Drop rows where specified columns contain NULL values.

Parameters

- **cols** (Union[str, Attribute]) – Attributes to consider when dropping rows.
- **how** (str) – If 'any', rows with NULL values in any of the specified columns will be dropped. If 'all', rows with NULL values in all of the specified columns will be dropped.

Returns A new DataSet filtered to rows where specified columns contain NULL values (any or all, depending on the value of the *how* parameter).

Return type "DataSet"

join_pandas (*from_col*, *dataframe*, *key_col*, *value_col*)

Join a column of pandas data with the data set.

See join_data for extended details.

Parameters

- **from_col** (Union[Attribute, str]) – An expression or attribute name in the data set to join the data to. The type of this column should match the keys in the mapping.
- **dataframe** (DataFrame) – The pandas DataFrame that contains values to map to in this data set. The DataFrame cannot be empty and has a limit of 100,000 rows.
- **key_col** (str) – The name of the pandas column to obtain the keys to match from_col. If duplicated keys occur, only one key from the data set is used in the join.
- **value_col** (str) – The name of the pandas column to obtain the values of the mapping.

Returns The original data set with a new column containing analyst-supplied values.

Return type *DataSet*

join_data (*from_col*, *new_col*, *mapping*)

Join key-value data to the data set.

Analyst supplied data can be added to an existing sensitive data set without a loss to privacy. This is a replacement for the decode expression when there are many keys (>100).

Values matching the keys of the dictionary mapping are replaced by the associated values. Values that do not match any of the keys are replaced by NULL. Keys and values may be python literals supported by the LeapYear client, or other Attributes.

Note: If the combination of keys assure there should be no NULL values, the client will not automatically convert the result of *join_data* to a non-nullable type. The user must use *coalesce* to remove NULL from the domain of possible values.

Parameters

- **from_col** (Union[Attribute, str]) – An expression or attribute name in the data set to join the data to. The type of this column should match the keys in the mapping.
- **new_col** (str) – The name of the new attribute that is added to the returned data set with the same type as the mapping values.
- **mapping** (Mapping[Union[float, bool, int, str, datetime, date], Union[float, bool, int, str, datetime, date]]) – Python dictionary of key-value pairs to add to the data set. The keys should be unique. The mapping cannot be empty and has a limit of 100,000 keys.

Returns The original data set with a new column containing analyst-supplied values.

Return type *DataSet*

Example

Suppose that we have a table with a `Sex` column containing the values `male` and `female`:

Sex	Age
male	22
female	38
female	26
female	35
male	35

We'd like to encode the abbreviations (coming from the first letter) as a new column, coming from a pandas DataFrame of the following form:

Sex	first_letter
male	m
female	f

To do so, we call this function, assuming the LeapYear DataSet is called `ds`, and we wish to call the new column `sex_first_letter`:

```
new_ds = ds.join_data("Sex", "sex_first_letter", {"female": "f", "male": "m"})
```

This will produce a DataSet which looks like:

Sex	Age	sex_first_letter
male	22	m
female	38	f
female	26	f
female	35	f
male	35	m

predict (*model*, *attrs=None*, *name='predict'*)
Return a DataSet with a prediction column for the model.

Parameters

- **model** (Union[*ClusterModel*, *GLM*, *RandomForestClassifier*, *RandomForestRegressor*, *GradientBoostedTreeClassifier*]) – The model to predict outcomes with.
- **attrs** (Optional[List[Union[Attribute, str]]]) – The attributes to use in the transformation, or None if all the attributes should be used.
- **name** (str) – Name or common prefix of the attribute(s) to be created.

Returns The original dataset with the prediction column(s) appended.

Return type *DataSet*

predict_proba (*model*, *attrs=None*, *name='proba'*)
Return a DataSet with prediction probability columns for the model.

Parameters

- **model** (Union[*GLM*, *RandomForestClassifier*, *GradientBoostedTreeClassifier*]) – The model to predict outcomes with.
- **attrs** (Optional[List[Union[Attribute, str]]]) – The attributes to use in the transformation, or None if all the attributes should be used.
- **name** (str) – Name or common prefix of the attribute(s) to be created.

Returns The original dataset with the prediction probability column(s) appended.

Return type *DataSet*

Attribute class

class leapyear.dataset.**Attribute**(*expr*, *relation*, ***, *ordering=OrderSpec(osDirection=SortDirection.Asc, osNullOrdering=None)*, *name=None*)

An attribute of a Dataset.

This exists for transformations to be performed on single attributes of the dataset. For example, the attribute *height* might be measured in meters however centimeters might be more appropriate, so an intermediate Attribute can be extracted from the DataSet and manipulated.

All attribute manipulations are lazy; they are not evaluated until they are needed to perform an analysis on the LeapYear server.

classmethod clear_cache()
Clear the memo cache of attribute types.

Return type None

property type
Get the data type of this attribute.

Return type AttributeType

property expression
Get the Attribute's expression.

Return type Expression

property relation
Get the relation this Attribute originates from.

Return type Optional[Relation]

property ordering
Get the ordering of this attribute.

Return type OrderSpec

property name
Get the name or alias of this attribute.

Return type str

alias(*name*)
Associate an alias with an attribute.

Parameters **name** (str) – The name to associate with an attribute.

Return type ForwardRef

is_not()
Return the boolean inverse of the attribute.

Return type ForwardRef

sign()
Return the sign of each element (1, 0 or -1).

Return type ForwardRef

floor()
Apply the floor transformation to the attribute.

Each attribute value is transformed to the largest integer less than or equal to the attribute value.

Return type ForwardRef

ceil()
Apply the ceiling transformation to the attribute.

Each attribute value is transformed to the smallest integer greater than or equal to the attribute value.

Return type ForwardRef

exp()
Apply exponent transformation to an attribute.

Return type ForwardRef

expm1()
Apply exponent transformation to an attribute and subtract one.

Return type ForwardRef

sqrt()
Apply square root transformation to an attribute.

Return type ForwardRef

log()
Apply natural logarithm transformation to an attribute.

Return type ForwardRef

log1p()
Apply natural logarithm transformation to an attribute and add 1.

Return type ForwardRef

sigmoid()
Apply sigmoid transformation to an attribute.

Return type ForwardRef

replace(*mapping*)
Replace specified values with the new values or attribute expressions.

Values matching the keys of the mapping and replaced by the associated values. Values that do not match any of the keys are kept.

Keys and values may be python literals supported by the LeapYear client, or other Attributes.

Parameters **mapping** (Mapping[Union[float, bool, int, str, datetime, date], NewType()(AttributeLike, Union[ForwardRef, Expression, float, bool, int, str, datetime, date])]) – A mapping from values of this attribute's type (T) to

another set of values and a different type (U). U may be a python literal type (int, bool, datetime, ...) or another Attribute.

Returns The converted attribute.

Return type *Attribute*

property microsecond
Return the microseconds part of a temporal type.

Return type ForwardRef

property second
Return the seconds part of a temporal type.

Return type ForwardRef

property minute
Return the minutes part of a temporal type.

Return type ForwardRef

property hour
Return the hours part of a temporal type.

Return type ForwardRef

property day
Return the days part of a temporal type.

Return type ForwardRef

property month
Return the months part of a temporal type.

Return type ForwardRef

property year
Return the years part of a temporal type.

Return type ForwardRef

greatest(*attrs*)
Take the elementwise maximum.

Return NULL if and only if all attribute values are NULL.

Return type ForwardRef

least(*attrs*)
Take the elementwise minimum.

Return NULL if and only if all entries are NULL.

Return type ForwardRef

coalesce(*fallthrough*, *attrs=None*)
Convert all NULL values of an attribute to a value.

This function will extend the type of the attribute if necessary and drop the nullable tag from the attribute data type.

Parameters

- **fallthrough** (Any) – The value of the same type as this attribute that all NULL values get converted into.

- **attrs** (Optional[List[ForwardRef]]) – The list of attributes to try before the fallthrough case.

Returns The non-nullable attribute with extended type range (if necessary).

Return type *Attribute*

Examples

1. Use coalesce to replace missing values with '1' and create a new attribute 'col1_trn':

```
>>> ds2 = ds1.with_attributes({'col1_trn':ds1['col1'].coalesce('1')})
```

isnull()
Boolean check whether an attribute value is NULL.

Return a boolean attribute which resolves to True whenever the underlying attribute value is NULL.

Return type ForwardRef

notnull()
Boolean inverse of *isnull()*.

Return type ForwardRef

decode(*mapping*)
Map specified values to the new values or attribute expressions.

Values matching the keys of the mapping and replaced by the associated values.

Values that do not match any of the keys are replaced by NULL.

Keys and values may be python literals supported by the LeapYear client, or other Attributes.

If the combination of keys assure there should be no NULL values, the client will not automatically convert the result of *decode* to a non-nullable type. The user must use *coalesce* to remove NULL from the domain of possible values.

Parameters **mapping** (Mapping[Any, Any]) – A mapping from values of this attribute's type (T) to another set of values and a different type (U). U may be a python literal type (int, bool, datetime, ...) or another Attribute.

Returns The converted attribute.

Return type *Attribute*

Examples

1. Create a new column 'col1_trn' that is based on values in 'col1'. If the value matches 'pattern' we assign the same string otherwise we assign 'Other' using the decode function:

```
>>> import leapyear.functions as f
>>> some_func = lambda x: (x != 'pattern').decode({True:'Other', False:
↪'pattern'})
>>> ds2 = ds1.with_attributes({'col1_trn': some_func(f.col('col1'))})
```

2. Create a new column 'col1_trn' that is based on values in 'col1'. If the value == 0 then 'col1_trn' takes the value 0. Otherwise, it takes the value of 'col2':

```
>>> import leapyear.functions as f
>>> some_func = lambda x: (x==0).decode({True:0,False:f.col('col2')})
>>> ds2 = ds1.with_attributes({'col1_trn': some_func(f.col('col1'))})
```

3. Create a new column 'col1_trn' that is based on values in 'col1' and 'col2. Based on an expression involving 'col1' and 'col2', 'col1_trn' takes the value 'col1 or 'col2':

```
>>> import leapyear.functions as f
>>> def some_func(x,y):
        return ((x==0)&(y!=0)).decode({True:f.col('col1'),False:f.col('col2')}
↪)
>>> ds2 = ds1.with_attributes({'col1_trn': some_func(f.col('col1'),f.col('col2
↪'))})
```

is_in (*options*)
Boolean check whether an attribute value is in a list.

Return a boolean attribute which resolves to `True` whenever the underlying attribute matches one of the list entries.

Return type `ForwardRef`

text_to_bool ()
Convert the attribute to a boolean.

Strings that match (case insensitively) "1", "y", "yes", "t", or "true" are converted to `True`;

Strings that match (case insensitively) "0", "n", "no", "f", or "false" are converted to `False`. Other strings are treated as `NULL`.

Return type `ForwardRef`

text_to_real (*lb*, *ub*)
Convert a text attribute to a real-valued attribute.

Parameters

- **lb** (`float`) – The lower bound for the domain of possible values of the new attribute. All values below this will be converted to `NULL`.
- **ub** (`float`) – The upper bound for the domain of possible values of the new attribute. All values below this will be converted to `NULL`.

Return type `ForwardRef`

text_to_factor (*distinct_vals_list*)
Convert a text attribute to a factor attribute.

Parameters **distinct_vals_list** – Input list of distinct values the input text column takes. Rows with values that are not in this list will be filled with nulls.

Return type `ForwardRef`

text_to_int (*lb*, *ub*)
Convert a text attribute to an integer-valued attribute.

NOTE: integers close to `MAX_INT64` or `MIN_INT64` are not represented precisely.

Parameters

- **lb** (`int`) – The lower bound for the domain of possible values of the new attribute. All values below this will be converted to `NULL`.

- **ub** (int) – The upper bound for the domain of possible values of the new attribute. All values below this will be converted to NULL.

Return type ForwardRef

as_real()
Convert the attribute to a real.

Return type ForwardRef

as_factor()
Represent this attribute as a factor attribute.

Converts the attribute of type *INT* or *BOOL* to *FACTOR*.

Note: For more complex conversions, consider *decode()*.

Examples

1. Convert an attribute 'col1' in ds1 to factor:

```
>>> ds2 = ds1.with_attributes({'col1_fac':ds1['col1'].as_factor()})
```

Return type ForwardRef

asc()
Sort ascending.

Causes the attribute to be sorted in ascending order when the sorting order is applied to the dataset.

Examples

1. Order a dataset by multiple cols and drop duplicates:

```
>>> ds2 = ds1.order_by(ds1['col1'].asc(),ds1['col2'].asc(),ds1['col3'].asc())
>>> ds2 = ds2.drop_duplicates(['col2'])
```

Return type ForwardRef

asc_nulls_first()
Sort ascending, missing values first.

Causes the attribute to be sorted in ascending order when the sorting order is applied to the dataset, with NULL values first.

Return type ForwardRef

asc_nulls_last()
Sort ascending, missing values last.

Causes the attribute to be sorted in ascending order when the sorting order is applied to the dataset, with NULL values last.

Return type ForwardRef

desc()
Sort decending.

Causes the attribute to be sorted in descending order when the sorting order is applied to the dataset.

Return type ForwardRef

desc_nulls_first()
Sort descending, missing values first.

Causes the attribute to be sorted in descending order when the sorting order is applied to the dataset, with NULL values first.

Return type ForwardRef

desc_nulls_last()
Sort descending, missing values last.

Causes the attribute to be sorted in descending order when the sorting order is applied to the dataset, with NULL values last.

Return type ForwardRef

class leapyear.dataset.**AttributeType**(**args*, ***kwargs*)
The type of an Attribute.

An AttributeType is a read-only object returned by the server, and should never be constructed directly.

property name
Get the name of an AttributeType, e.g. INT.

Return type str

property nullable
Get the nullability of an AttributeType.

Return type bool

property domain
Get the domain of an AttributeType.

The return type is dependent on the name of the type. e.g. the domain of an INT AttributeType is a tuple of the lower and upper bound, like (0, 10).

Return type Any

Aliases

leapyear.dataset.attribute.**AttributeLike**(*x*)

Attribute-like objects are those that can be readily converted to an attribute.

These include existing attributes, dates, strings, integers, floats and expressions based on these objects.

Grouping and Windowing classes

class leapyear.dataset.**GroupedData**(*grouping*, *rel*)
The result of a *DataSet.group_by()*.

Run *agg()* to get a DataSet.

agg(**attr_aggs*, *max_count=None*, ***kwargs*)
Specify the aggregations to perform on the GroupedData.

Parameters

- **attr_aggs** (Tuple[List[Union[Attribute, str]], Union[str, Aggregation]]) – A list of pairs. The second element of the pair is the aggregation to perform; the first is a list of columns on which to perform it. This is a list and not just a single attribute to support nullary and binary aggregations.

  Available aggregations:

  – *min*
  – *max*
  – *and*
  – *or*
  – *count_distinct*
  – *approx_count_distinct*
  – *count*
  – *mean*
  – *stddev*
  – *stddev_samp*
  – *stddev_pop*
  – *variance*
  – *var_samp*
  – *var_pop*
  – *skewness*
  – *kurtosis*
  – *sum*
  – *sum_distinct*
  – *covar_samp*
  – *covar_pop*
  – *corr*

- **max_count** (Optional[int]) – If count aggregate is requested, this parameter will be used as an upper bound in the schema of the derived aggregate count attribute(s) of the grouped *DataSet*. If not supplied, the upper bound would be inferred from the data before returning the resulting *DataSet* object.

Note: When this parameter is set too high, differentially private computations on the derived aggregate attribute would include higher randomization effect. When it is set too low, all counts higher than `max_count` will be replaced by `max_count`.

- **kwargs** (*Any*) – All keyword arguments are passed to the function *run* when the parameter `max_count` has to be inferred from the data. This includes `max_timeout_sec`, which defaults to 300 seconds.

Returns A *DataSet* object, containing aggregation results.

Return type *DataSet*

Example

To compute correlation of height and weight as well as the count, run:

```
>>> gd.agg((['height','weight'], 'corr'), ([], 'count'))
```

class leapyear.dataset.**Window**

Utility functions for defining WindowSpec.

Examples

```
>>> # ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
>>> window = Window.order_by("date").rows_between(
>>>     Window.unbounded_preceding, Window.current_row,
>>> )
```

```
>>> # PARTITION BY country ORDER BY date ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING
>>> window = Window.order_by("date").partition_by("country").rows_between(-3, 3)
```

classmethod partition_by(**cols*)

Create a *WindowSpec* with the partitioning defined.

Parameters **cols** (NewType()(AttributeLike, Union[ForwardRef, Expression, float, bool, int, str, datetime, date])) – each attribute that should be used to partition windows. If using an attribute name, user must use *leapyear.functions.col* in order to reference the attribute.

Return type WindowSpec

classmethod order_by(**cols*)

Create a *WindowSpec* with the ordering defined.

Parameters **cols** (NewType()(AttributeLike, Union[ForwardRef, Expression, float, bool, int, str, datetime, date])) – each attribute that should be used to order within a window. If using an attribute name, user must use *leapyear.functions.col* in order to reference the attribute.

Return type WindowSpec

classmethod rows_between(*start*, *end*)

Create a *WindowSpec* with the frame boundaries defined.

Create a *WindowSpec* with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive). Both *start* and *end* are relative positions from the current row. For example, "0" means "current row", while "-1"

means the row before the current row, and "5" means the fifth row after the current row. We recommend users use *Window.unboundedPreceding*, *Window.unboundedFollowing*, and *Window.currentRow* to specify special boundary values, rather than using integral values directly.

Note: windows of 1000 rows or more are not currently supported for expressions other than lead and lag.

Parameters

- **start** (int) – boundary start, inclusive. The frame is unbounded if this is `Window.unboundedPreceding`, or any value less than or equal to -9223372036854775808.
- **end** (int) – boundary end, inclusive. The frame is unbounded if this is `Window.unboundedFollowing`, or any value greater than or equal to 9223372036854775807.

Return type `WindowSpec`

1.4.6 Module leapyear.functions

Functions for Attributes.

Datetime functions

Time functions for Attributes.

Missing functions compared to spark: * current_date * current_timestamp * date_format * from_unixtime * from_utc_timestamp * unix_timestamp

`leapyear.functions.time.`**`add_months`**(*start_date*, *num_months*)

Return the date that is *num_months* after *start_date*.

NOTE: This function can take a datetime as input, but produces a date output regardless.

Examples

```
>>> str(dt)
'2004-02-29 23:59:59'
>>> add_months(dt, 1)
'2004-03-29'
>>> add_months(dt, 13)
'2005-03-01'
```

Return type `Attribute`

`leapyear.functions.time.`**`date_add`**(*start*, *days*)

Return the date that is *days* days after *start*.

NOTE: This function can take a datetime as input, but produces a date output regardless.

Examples

```
>>> str(dt)
'2004-02-28 23:59:59'
>>> date_add(dt, 1)
'2004-02-29'
>>> date_add(dt, 2)
'2004-03-01'
```

Return type Attribute

leapyear.functions.time.**date_sub**(*start*, *days*)

Return the date that is *days* days before *start*.

NOTE: This function can take a datetime as input, but produces a date output regardless.

Examples

```
>>> str(dt)
'2004-03-01 23:59:59'
>>> date_sub(dt, 1)
'2004-02-29'
>>> date_sub(dt, 2)
'2004-02-29'
```

Return type Attribute

leapyear.functions.time.**datediff**(*end*, *start*)

Return the number of days from *start* to *end*.

Examples

1. Create date diff column 'date_sub' that is a difference between datetime column 'col1' and datetime column 'col2' in ds1:

```
>>> import leapyear.functions as f
>>> ds2 = ds1.with_attributes({'date_sub':f.time.datediff(f.col('col1'),f.col(
↪'col2'))})
```

Return type Attribute

leapyear.functions.time.**dayofmonth**(*e*)

Extract the day of the month as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**dayofyear**(*e*)

Extract the day of the year as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**hour**(*e*)

Extract the hours as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**last_day**(*e*)
Return the last day of the month which the given date belongs to.

NOTE: This function can take a datetime as input, but produces a date output regardless.

Examples

```
>>> str(dt)
'2004-02-01 23:59:59'
>>> last_day(dt)
'2004-02-29'
```

Return type Attribute

leapyear.functions.time.**minute**(*e*)
Extract the minutes as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**month**(*e*)
Extract the month as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**months_between**(*date1*, *date2*)
Return integer number of months between dates date1 and date2.

This is based on both the day and the month, not the number of days between the dates. Note in the last line of the examples that there is 1 month between dateC and dateB even though there are only 29 days between them. The result is negative if date1 is >=1 month before date2. The number of months is always rounded down to the nearest integer.

Examples

```
>>> str(dateA)
'2004-01-01'
>>> str(dateB)
'2004-02-01'
>>> str(dateC)
'2004-03-02'
>>> months_between(dateA, dateB)
-1
>>> months_between(dateB, dateA)
1
>>> months_between(dateC, dateB)
1
```

Return type Attribute

leapyear.functions.time.**next_day**(*date*, *day_of_week*)
Return the next date that falls on the specified day of the week.

NOTE: This function can take a datetime as input, but produces a date output regardless.

Examples

```
>>> str(dt)
'2004-02-23 23:59:59'
>>> next_day(dt, "Sunday")
'2004-02-29'
```

Return type Attribute

leapyear.functions.time.**quarter**(*e*)
Extract the quarter as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**second**(*e*)
Extract the seconds as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**to_date**(*e*)
Convert the column into Date type.

Examples

1. Create truncated column 'date_trunc' from a datetime column 'col1' in ds1:

```
>>> import leapyear.functions as f
>>> ds2 = ds1.with_attributes({'date_trunc':f.time.to_date(f.col('col1'))})
```

Return type Attribute

leapyear.functions.time.**to_datetime**(*ts*)
Convert the column into Datetime type.

Return type Attribute

leapyear.functions.time.**trunc**(*date*, *fmt*)
Return *date* truncated to the unit specified by the format.

fmt can be one of: *"year"*, *"month"*, *"day"*

NOTE: This function can take a datetime as input, but produces a date output regardless.

Examples

```
>>> str(dt)
'2004-02-29 23:59:59'
>>> trunc(dt, "month")
'2004-02-01'
```

Return type Attribute

leapyear.functions.time.**weekofyear**(*e*)
Extract the week number as an integer from a given date/datetime attribute.

Week numbers range from 1 to (up to) 53, and new weeks start on Monday. Dates at the begining / end of a year may be considered to be part of a week from the previous / next year. Full documentation of week numbering can be found here.

Return type Attribute

leapyear.functions.time.**year**(*e*)
Extract the year as an integer from a given date/datetime attribute.

Return type Attribute

leapyear.functions.time.**yearofweek**(*e*)
Extract the year based on the ISO week numbering where a week associated the previous year may spill over into the next calendar year.

Full documentation of week numbering can be found here.

Return type Attribute

leapyear.functions.time.**year_with_week**(*e*)
Extract the year and the ISO week from a Date column and returns a Factor which combines the two.

For example, given a row containing 'dt(2021,1,1)' this function returns '2020-53'.

Full documentation with week numbering can be found here.

Return type Attribute

leapyear.functions.time.**dayofweek**(*e*)
Extract the day of the week number as an integer from a given date/datetime attribute, where Monday = 1, . . . , Sunday = 7.

Return type Attribute

leapyear.functions.time.**parse_clamped_time**(*e*, *bounds*, **, fmt='yyyy-MM-dd HH:mm:ss'*)
Parse a Text column and return a DateTime column using the given format clamped to bounds.

Format specification follows: https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html

Examples

```
>>> import datetime.datetime as dt
>>> parse_clamped_time(f.col("date"), bounds=(dt(2000, 1, 1), dt(2020, 12, 31)),␣
↪fmt="yyyMMdd HHmmss")
```

Return type Attribute

Math functions

Math functions for Attributes.

leapyear.functions.math.**acos**(*e*)
Compute the cosine inverse of the given value.

The returned angle is in the range 0 through π.

Return type Attribute

leapyear.functions.math.**asin**(*e*)
Compute the sine inverse of the given value.

The returned angle is in the range -pi/2 through pi/2.

Return type Attribute

leapyear.functions.math.**atan**(*e*)
Compute the tangent inverse of the given value.

Return type Attribute

leapyear.functions.math.**cbrt**(*e*)
Compute the cube-root of the given value.

Return type Attribute

leapyear.functions.math.**ceil**(*e*)
Compute the ceiling of the given value.

Return type Attribute

leapyear.functions.math.**cos**(*e*)
Compute the cosine of the given value.

Return type Attribute

leapyear.functions.math.**cosh**(*e*)
Compute the hyperbolic cosine of the given value.

Return type Attribute

leapyear.functions.math.**degrees**(*e*)
Convert an angle measured in radians to an equivalent angle measured in degrees.

Return type Attribute

leapyear.functions.math.**exp**(*e*)
Compute the exponential of the given value.

Return type Attribute

leapyear.functions.math.**expm1**(*e*)
Compute the exponential of the given value minus one.

Return type Attribute

leapyear.functions.math.**floor**(*e*)
Compute the floor of the given value.

Return type Attribute

leapyear.functions.math.**hypot**(*x*, *y*)
Compute sqrt(x^2 + y^2) without intermediate overflow or underflow.

Return type Attribute

leapyear.functions.math.**log**(*e*)
Compute the natural logarithm of the given value.

Return type Attribute

leapyear.functions.math.**log10**(*e*)
Compute the logarithm of the given value in base 10.

Return type Attribute

leapyear.functions.math.**log1p**(*e*)
Compute the natural logarithm of the given value plus one.

Return type Attribute

leapyear.functions.math.**log2**(*e*)
Compute the logarithm of the given column in base 2.

Return type Attribute

leapyear.functions.math.**sigmoid**(*e*)
Compute the sigmoid of the given value.

Return type Attribute

leapyear.functions.math.**pow**(*base*, *exp*)
Return the value of the first argument raised to the power of the second argument.

Return type Attribute

leapyear.functions.math.**radians**(*e*)
Convert an angle measured in degrees to an equivalent angle measured in radians.

Return type Attribute

leapyear.functions.math.**round**(*e*, *scale=0*)
Round the value of e to scale decimal places.

Examples

1. Create a new attribute 'col1_trn' which is derived by rounding 'col1' to 2 digits:

```
>>> import leapyear.functions as f
>>> ds2 = ds1.with_attributes({'col1_trn': f.math.round(f.col('col1'),2)})
```

Return type Attribute

leapyear.functions.math.**signum**(*e*)
Compute the signum of the given value.

Return type Attribute

leapyear.functions.math.**sin**(*e*)
Compute the sine of the given value.

Return type Attribute

leapyear.functions.math.**sinh**(*e*)
Compute the hyperbolic sine of the given value.

Return type Attribute

leapyear.functions.math.**sqrt**(*e*)
Compute the square root of the specified float value.

Return type Attribute

leapyear.functions.math.**tan**(*e*)
Compute the tangent of the given value.

Return type Attribute

leapyear.functions.math.**tanh**(*e*)
Compute the hyperbolic tangent of the given value.

Return type Attribute

leapyear.functions.math.**erf**(*e*)
Compute the error function on the given value.

Return type Attribute

leapyear.functions.math.**erfc**(*e*)
Compute the error function on the given value.

Return type Attribute

leapyear.functions.math.**inverf**(*e*)
Compute the error function on the given value.

Return type Attribute

leapyear.functions.math.**inverfc**(*e*)
Compute the error function on the given value.

Return type Attribute

Non-aggregate functions

Non-aggregate functions for Attributes.

leapyear.functions.non_aggregate.**all**(**exprs*)
Return True if all columns are True.

Return type Attribute

leapyear.functions.non_aggregate.**attr_in**(*input_attr_name*, *in_list*)
Expression for filtering rows based on attribute values that are IN a list of values.

Parameters

- **input_attr_name** (*String*) – Name of Attribute based on which to filter
- **in_list** (*Iterable*) – List values for filtering

Returns Boolean Attribute

Return type *Attribute*

Examples

1. Filtering based on an attribute taking a value from a list of values:

```
>>> import leapyear.functions as f
>>> ds2 = ds.where(f.attr_in('col1',in_list=[val1,val2,val3]))
```

leapyear.functions.non_aggregate.**attr_not_in**(*input_attr_name*, *not_in_list*)
Expression for filtering rows based on attribute values that are IN a list of values.

Parameters

- **input_attr_name** (str) – Name of Attribute based on which to filter
- **not_in_list** (Iterable) – List values for filtering

Returns Boolean Attribute

Return type *Attribute*

Examples

1. Filtering based on an attribute NOT taking a value from a list of values:

```
>>> import leapyear.functions as f
>>> ds2 = ds.where(f.attr_not_in('col1',not_in_list=[val1,val2,val3]))
```

leapyear.functions.non_aggregate.**any**(**exprs*)
Return True if any column is True.

Return type Attribute

leapyear.functions.non_aggregate.**abs**(*e*)
Compute the absolute value.

Return type Attribute

leapyear.functions.non_aggregate.**col**(*col_name*)
Return an Attribute based on the given name.

Return type Attribute

leapyear.functions.non_aggregate.**column**(*col_name*)
Return an Attribute based on the given name.

Return type Attribute

leapyear.functions.non_aggregate.**greatest**(**exprs*)
Return the greatest value of the list of column names, skipping null values.

Return type Attribute

leapyear.functions.non_aggregate.**isnull**(*e*)
Return true iff the column is null.

Return type Attribute

leapyear.functions.non_aggregate.**least**(**exprs*)
Return the greatest value of the list of column names, skipping null values.

Return type Attribute

leapyear.functions.non_aggregate.**lit**(*literal*)
Create an Attribute of literal value.

Return type Attribute

leapyear.functions.non_aggregate.**negate**(*e*)
Unary minus.

Return type Attribute

leapyear.functions.non_aggregate.**not_**(*e*)
Inversion of boolean expression.

Return type Attribute

leapyear.functions.non_aggregate.**when**(*condition*, *value*)
Evaluate a list of conditions and returns one of multiple possible result expressions.

If otherwise is not defined at the end, null is returned for unmatched conditions.

Example

Encoding gender string column into an integer.

```
>>> df.select(when(col("gender") == "male", 0)
...           .when(col("gender") == "female", 1)
...           .otherwise(2))
```

Return type Attribute

leapyear.functions.non_aggregate.**to_text**(*e*)
Convert an attribute to a string representation.

Return type Attribute

String functions

Math functions for Attributes.

leapyear.functions.string.**ascii**(*e*)
Convert the first character of the string to its ASCII value.

Return type Attribute

leapyear.functions.string.**concat**(**exprs*, *sep=None*)
Concatenation of strings with optional separator.

Concatenating factors {'a', 'b'} and {'c', 'd'} gives {'ac', 'ad', 'bc', 'cd'}. Including the separator " " gives {'a c', 'a d', 'b c', 'c d'}.

If any of the expressions are type TEXT, then the result will be TEXT.

Examples

1. Create new column 'col1_trn' which concatenation of 'col1' and 'col2' with separator '-'. It can be used to contruct a date column from day and month columns:

```
>>> import leapyear.functions as f
>>> ds2 = ds1.with_attributes({'col1_trn':f.concat(ds1['col1'],ds1['col2'],sep='-
↪')})
```

Return type Attribute

leapyear.functions.string.**instr**(*attr*, *substr*)
Give the position of substring in the Attribute, otherwise 0.

Return type Attribute

leapyear.functions.string.**length**(*attr*)
Return the length of the string.

Return type Attribute

leapyear.functions.string.**levenshtein**(*attr1*, *attr2*)
Compute the Levenshtein distance between strings.

Return type Attribute

leapyear.functions.string.**locate**(*attr*, *substr*, *pos=None*)
Give the position of substr in the Attribute, optionally after pos.

Position is returned as a positive integer starting at 1 if the substring is found, and 0 if the substring is not found.

leapyear.functions.string.**lpad**(*attr*, *len_*, *pad*)
Pad the string on the left with pad to make the total length *len_*.

When pad is an empty string, the string is returned without modification, or truncated to *len_*.

Return type Attribute

leapyear.functions.string.**ltrim**(*attr*)
Remove whitespace characters on the beginning of the string.

Return type Attribute

leapyear.functions.string.**reverse**(*attr*)
Reverse the string.

Return type Attribute

leapyear.functions.string.**repeat**(*attr*, *n*)
Repeat the string n times.

Return type Attribute

leapyear.functions.string.**rpad**(*attr*, *len_*, *pad*)
Pad the string on the right with pad to make the total length *len_*.

When pad is an empty string, the string is returned without modification, or truncated to *len_*.

Return type Attribute

leapyear.functions.string.**rtrim**(*attr*)
Remove whitespace characters at the end of the string.

Return type Attribute

leapyear.functions.string.**soundex**(*attr*)
Return the soundex code for the specified expression.

Return type Attribute

leapyear.functions.string.**substring**(*attr*, *start*, *len_*)
Return the substring of length *len_* starting at start.

Return type Attribute

leapyear.functions.string.**substring_index**(*attr*, *delim*, *len_*)
Return the substring from string str before count occurrences of the delimiter delim.

If count is positive, everything the left of the final delimiter (counting from left) is returned. If count is negative, every to the right of the final delimiter (counting from the right) is returned. substring_index performs a case-sensitive match when searching for delim.

Return type Attribute

leapyear.functions.string.**translate**(*attr*, *match*, *replace*)
Translate any character in the src by a character in replace.

The characters in replace correspond to the characters in matching. The translate will happen when any character in the string matches the character in the match.

Return type Attribute

leapyear.functions.string.**trim**(*attr*)
Remove the whitespace from the beginning and end of the string.

Return type Attribute

leapyear.functions.string.**lex_lt**(*attr1*, *attr2*)
Lexicographical less-than operation.

Return type Attribute

leapyear.functions.string.**lex_lte**(*attr1*, *attr2*)
Lexicographical less-than-or-equal operation.

Return type Attribute

leapyear.functions.string.**lex_gt**(*attr1*, *attr2*)
Lexicographical greater-than operation.

Return type Attribute

leapyear.functions.string.**lex_gte**(*attr1*, *attr2*)
Lexicographical greater-than-or-equal operation.

Return type Attribute

leapyear.functions.string.**remove_accents**(*attr*)
Remove all accents from the string.

Return type Attribute

leapyear.functions.string.**lower**(*attr*)
Convert strings to lowercase.

Return type Attribute

leapyear.functions.string.**upper**(*attr*)
Convert strings to uppercase.

Return type Attribute

leapyear.functions.string.**regex_extract**(*attr*, *pattern*, *group_idx*)
Use a regular expression pattern to extract a part of the string.

This function always results in a nullable Text type.

Return type Attribute

leapyear.functions.string.**regex_replace**(*attr*, *pattern*, *replace*)
Use a regular expression pattern to replace a part of the string.

This function always results in a nullable Text type.

Return type Attribute

Windowing functions

Window functions for Attributes.

leapyear.functions.window.**lead**(*e*, *i*)
Compute the lead value.

Return type WindowAttribute

leapyear.functions.window.**lag**(*e*, *i*)
Compute the lag value.

Return type WindowAttribute

leapyear.functions.window.**first**(*e*)
Compute the first non-null value.

Return type WindowAttribute

leapyear.functions.window.**last**(*e*)
Compute the last non-null value.

Return type WindowAttribute

leapyear.functions.window.**count**(*e=None*)
Compute the number of non-null entries.

Return type WindowAttribute

leapyear.functions.window.**approx_count_distinct**(*attrs*)
Compute the number of distinct entries (using an approximate streaming algorithm).

Return type WindowAttribute

leapyear.functions.window.**mean**(*e*)
Compute the mean value.

Examples

1. Create a window specification that partitions based on 'col1' and orders by a 'date_col' and picks past 14 rows from current row 0. Then, we can compute mean of 'col2' over this window:

```
>>> import leapyear.functions as f
>>> from leapyear.dataset import Window
>>> ws1 = Window.partition_by(f.col('col1'))
                .order_by(f.col('date_col').asc())
                .rows_between(start=-14,end=0)
```

(continues on next page)

(continued from previous page)

```
>>> ds2 = ds1.project(['date_col','col1','col2'])
             .with_attribute('mean_col2', f.window.mean(f.col('col2')).over(ws1))
```

Return type WindowAttribute

leapyear.functions.window.**avg**(*e*)
Compute the mean value.

Examples

1. Create a window specification that partitions based on 'col1' and orders by a 'date_col' and picks past 14 rows from current row 0. Then, we can compute mean of 'col2' over this window:

```
>>> import leapyear.functions as f
>>> from leapyear.dataset import Window
>>> ws1 = Window.partition_by(f.col('col1'))
                .order_by(f.col('date_col').asc())
                .rows_between(start=-14,end=0)
>>> ds2 = ds1.project(['date_col','col1','col2'])
             .with_attribute('mean_col2', f.window.mean(f.col('col2')).over(ws1))
```

Return type WindowAttribute

leapyear.functions.window.**sum**(*e*)
Compute the sum.

Return type WindowAttribute

leapyear.functions.window.**or_**(*e*)
Compute the or of boolean values.

Return type WindowAttribute

leapyear.functions.window.**and_**(*e*)
Compute the and of boolean values.

Return type WindowAttribute

leapyear.functions.window.**min**(*e*)
Compute the min value.

Return type WindowAttribute

leapyear.functions.window.**max**(*e*)
Compute the max value.

Return type WindowAttribute

leapyear.functions.window.**stddev**(*e*)
Compute the sample standard deviation.

Return type WindowAttribute

leapyear.functions.window.**stddev_samp**(*e*)
Compute the sample standard deviation.

Return type WindowAttribute

leapyear.functions.window.**stddev_pop**(*e*)
Compute the population standard deviation.

Return type WindowAttribute

leapyear.functions.window.**variance**(*e*)
Compute the sample variance.

Return type WindowAttribute

leapyear.functions.window.**variance_samp**(*e*)
Compute the sample variance.

Return type WindowAttribute

leapyear.functions.window.**variance_pop**(*e*)
Compute the population variance.

Return type WindowAttribute

leapyear.functions.window.**skewness**(*e*)
Compute the skewness.

Return type WindowAttribute

leapyear.functions.window.**kurtosis**(*e*)
Compute the kurtosis.

Return type WindowAttribute

leapyear.functions.window.**covar_samp**(*e1*, *e2*)
Compute the sample covariance.

Return type WindowAttribute

leapyear.functions.window.**covar_pop**(*e1*, *e2*)
Compute the population covariance.

Return type WindowAttribute

leapyear.functions.window.**corr**(*e1*, *e2*)
Compute the correlation.

Return type WindowAttribute

1.4.7 Module leapyear.feature

Feature engineering classes.

OneHotEncoder class

class leapyear.feature.**OneHotEncoder**(*input_cols*, *max_size=32*, *drop_originals=True*, *drop_last=True*)
One-hot encode attributes.

FACTOR and INT columns that can have less than max_size values are converted to BOOL columns indicating the presence of the value. Will only work with non-nullable columns. Nullable columns can be converted to non-nullable with *leapyear.dataset.Attribute.coalesce()*.

The last category is not included by default, where the categories are sorted lexicographically based on their characters' ASCII values.

Examples

1. Using OneHotEncoder on two columns 'col1' and 'col2' in Dataset 'ds1':

```
>>> ohe = OneHotEncoder(['col1', 'col2'], drop_originals=True, max_size=64)
>>> ds2 = ohe.transform(ds1)
```

Parameters

- **input_cols** (*Sequence[str]*) – the names of the input columns.
- **max_size** (*int*) – maximum number of values to one hot encode per column. (default: 32)
- **drop_originals** (*bool*) – drop columns not derived from the input columns. (default: True)
- **drop_last** (*bool*) – drop the last column containing redundant information. (default: True)

BoundsScaler class

class leapyear.feature.**BoundsScaler**(*input_cols*, *lower=0.0*, *upper=1.0*)

Scale the attributes by the bounds of the type.

BOOL, INT and REAL columns are scaled so all values fall between min and max (inclusive). In contrast to MinMaxScaler and StandardScaler, there is no privacy leakage using this class.

Examples

1. Using BoundsScaler on two columns 'col1' and 'col2' in Dataset 'ds1':

```
>>> bs = BoundsScaler(['col1','col2'])
>>> ds2 = bs.fit_transform(ds1)
```

Parameters

- **input_cols** (*Sequence[str]*) – the names of the input columns.
- **lower** (*float*) – attributes are scaled to this lower bound (default 0.)
- **upper** (*float*) – attributes are scaled to this upper bound (default 1.)

BoundsAbsScaler class

class leapyear.feature.**BoundsAbsScaler**(*input_cols*)

Scale the attributes by the max absolute value of the type bounds.

INT and REAL columns are scaled so all values fall between -1 and 1, with no shifting of the data.

Examples

1. Using BoundsAbsScaler on two columns 'col1' and 'col2' in Dataset 'ds1':

```
>>> bs = BoundsAbsScaler(['col1','col2'])
>>> ds2 = bs.fit_transform(ds1)
```

Parameters **input_cols** (*Sequence[str]*) – the names of the input columns.

MinMaxScaler class

class leapyear.feature.**MinMaxScaler**(*input_cols*, *min_=0.0*, *max_=1.0*)

Scale the attributes by the min and max of the attribute.

BOOL, INT and REAL columns are scaled so all values fall between min and max (inclusive).

Examples

1. Using MinMaxScaler on two columns 'col1' and 'col2' in Dataset 'ds1':

```
>>> ms = MinMaxScaler(['col1','col2'], min_=0.0, max_=1.0)
>>> ds2 = ms.fit_transform(ds1)
```

Parameters

- **input_cols** (*Sequence[str]*) – the names of the input columns.
- **min** (*float*) – attributes are scaled to this lower bound (default 0.)
- **max** (*float*) – attributes are scaled to this lower bound (default 1.)

MaxAbsScaler class

class leapyear.feature.**MaxAbsScaler**(*input_cols*)

Scale the attributes by the max absolute value of the min and the max.

INT and REAL columns are scaled so that values smaller than the absolute value of the min or max fall between -1 and 1, with no shifting of the data.

Examples

1. Using BoundsAbsScaler on two columns 'col1' and 'col2' in Dataset 'ds1':

```
>>> bs = BoundsAbsScaler(['col1','col2'])
>>> ds2 = bs.fit_transform(ds1)
```

Parameters **input_cols** (*Sequence[str]*) – the names of the input columns.

StandardScaler class

class leapyear.feature.**StandardScaler**(*input_cols*, *with_mean=True*, *with_stdev=True*)
Scale the attributes to be centered at zero with unit variance.

INT and REAL columns are scaled with removed mean and scaled to unit variance.

Examples

1. Using StandardScaler on columns 'col1' and 'col2' in Dataset 'ds1':

```
>>> ss = StandardScaler(['col1','col2'], with_mean=True, with_stdev=False )
>>> ds2 = ss.fit_transform(ds1)
```

Parameters

- **input_cols** (*Sequence[str]*) – the names of the input columns.
- **with_mean** (*bool*) – Remove the mean from the attributes.
- **with_stdev** (*bool*) – Scale the attribute to have unit standard deviation.

ScaleTransformModel class

class leapyear.feature.**ScaleTransformModel**(*attr_lower_upper*, *lower*, *upper*, *scale_bool*)
Scale attributes to new values.

Shift and scale each attribute so the 2 values associated with each attribute are mapped to the 2 values in *new_values*.

Normalizer class

class leapyear.feature.**Normalizer**(*input_cols*, *p*)
Compute the p-norm of attributes and normalize by norm value.

Will only work on INT and REAL columns.

Examples

1. Using Normalizer on columns 'col1' and 'col2' in Dataset 'ds1':

```
>>> norm_trn = Normalizer(['col1','col2'], p = 2)
>>> ds2 = norm_trn.fit_transform(ds1)
```

Parameters

- **input_cols** (*Sequence[str]*) – The list of the input columns that needs to normalized.
- **p** (*int*) – norm p

fit_transform(*dataset*, ***kwargs*)
Normalize a set of attributes.

Computes p norm.

Return type DataSet

Winsorizer class

class leapyear.feature.**Winsorizer**(*input_col*, *lo_val*, *hi_val*)
Bound the non-null values of the attribute to be within lo_val and hi_val.

Will only work on non-nullable INT and REAL columns. Nullable columns can be converted to non-nullable with *leapyear.dataset.Attribute.coalesce()*.

When transformed, returns the DataSet with an additional attribute that is winsorised between the given low and high value for the specified attribute.

Examples

1. Using Winsorizer on column 'col1' in Dataset 'ds1':

```
>>> wins = Winsorizer('col1', lo_val = 0, hi_val = 1)
>>> ds2 = wins.fit_transform(ds1)
```

Parameters

- **input_col** (*str*) – The name of the input columns.
- **lo_val** (*float*) – After transformation attribute will be greater than or equal to lo_val
- **hi_val** (*float*) – After transformation attribute will be lesser than or equal to hi_val

fit_transform(*dataset*, ***kwargs*)
Winsorize the specified attribute.

Called after providing lo and hi vals.

Return type DataSet

Bucketizer class

class leapyear.feature.**Bucketizer**(*input_col*, *split_vals*)
Quantize the attribute according to thresholds specified in split_vals.

attr < split_vals[0] -> bin 0 attr >= split_vals[0] and attr < split_vals[1] -> bin 1 . . . attr >= split_vals[-1] -> bin len(split_vals)

Works INT and REAL columns.

Examples

1. Using Bucketizer on column 'col1' in Dataset 'ds1':

```
>>> split_vals = [0, 0.25, 0.75]
>>> buck = Bucketizer('col1', split_vals = split_vals )
>>> ds2 = buck.fit_transform(ds1)
```

Parameters

- **input_col** (*str*) – the names of the input column.
- **split_vals** (*Sequence[float]*) – Thresholds for creating the bins

fit_transform(*dataset*, ***kwargs*)
For testing use only.

Return type DataSet

1.4.8 Module leapyear.analytics

Statistics and machine learning algorithms.

LeapYear analyses are functions that are executed by the server to compute statistics or to perform machine learning tasks on *DataSets*. These functions return an *Analysis* type, which is executed on the server by calling the *run()* method.

For simple statistics, such as *count()* or *mean()*, the values can be extracted using the following pattern:

```
>>> from leapyear import Client, DataSet
>>> from leapyear.analytics import count_rows, mean
>>> client = Client(url='http://ly-server:4401', username='admin', password='password
↪')
>>> dataset = DataSet.from_table('db.table')
>>> dataset_rows_analysis = count_rows(dataset)
>>> n_rows = dataset_rows_analysis.run()
>>> print(n_rows)
10473
>>> dataset_mean_x_analysis = mean('x0', dataset)
>>> mean_x = dataset_mean_x_analysis.run()
>>> print(mean_x)
5.234212346345
```

The computation of all univariate statistics follows the pattern for *mean()*. For more complicated machine learning tasks, multiple columns must be specified, depending on the task.

Unsupervised learning tasks (like clustering) will generally require the specification of which features in the *DataSet* to use. Supervised learning tasks (like regression) will additionally require the specification of a target variable.

For example, we can train a linear regression model as follows:

```
>>> from leapyear.analytics import generalized_linreg
>>> regression = generalized_linreg(['x0', 'x1'], 'y', dataset, affine=True, l2reg=1.
↪0)
>>> model = regression.run()
```

Helper routines are available for performing cross-validation (see *cross_val_score_linreg()*). Note that, unlike other analyses, they are immediately executed (without calling `run()`):

```
>>> from leapyear.analytics import cross_val_score_linreg
>>> cross_val_score = cross_val_score_linreg(
>>>     ['x0', 'x1'], 'y', dataset, cv=3,
>>>     affine=True, l1reg=0.1, l2reg=1.0, scorer='mse'
>>> )
```

Data Analysis

leapyear.analytics.**count**(*attr*, *dataset=None*, *drop_nulls=False*, *target_relative_error=None*, *max_budget=None*)

Analysis: Count the elements of an attribute.

This analysis can be executed using the *run* method to compute the approximate count of elements, including NULL values.

The user can request additional information about the computation with *run(rich_result=True)*. In this case, an object of *RandomizationInterval*, will be generated likely including the precise value of the computation on the data sample.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the count of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to ignore NULL values. Default: False.
- **target_relative_error** (Optional[float]) – A float value between 0 and 1 indicating the level of relative error that should be targeted for this computation. If specified, the system will attempt to ensure that the absolute value of the relative error between the randomized result and the true count is roughly *target_relative_error*. If this is not possible due to budget constraints set by the admin, the system will return a randomized result with the smallest randomization effect allowed. If not specified, a default value specified by the admin is used. This can only be used if the admin has turned on the adaptive count feature.
- **max_budget** (Optional[float]) – The maximum amount of budget that the system should spend while trying to achieve *target_relative_error*. If specified, the absolute amount of budget spent will not exceed *max_budget*. If the user specifies a value greater than the maximum budget for the computation set by the admin, system will use the admin-set maximum. If the system can achieve *target_relative_error* while spending less than *max_budget*, it will do so.

Returns Analysis object that can be executed using the *run* method.

Return type *CountAnalysisWithRI*

leapyear.analytics.**count_rows**(*dataset*, *target_relative_error=None*, *max_budget=None*)

Analysis: Count the number of rows in a dataset.

This analysis can be executed using the *run* method to compute the approximate number of rows in the dataset.

The user can request additional information about the computation with *run(rich_result=True)*. In this case, an object of *RandomizationInterval* will be generated, likely including the precise value of the computation on the data sample.

Parameters

- **dataset** (DataSet) – The input dataset.

- **target_relative_error** (Optional[float]) – A float value between 0 and 1 indicating the level of relative error that should be targeted for this computation. If specified, the system will attempt to ensure that the absolute value of the relative error between the randomized result and the true count is roughly *target_relative_error*. If this is not possible due to budget constraints set by the admin, the system will return a randomized result with the smallest randomization effect allowed. If not specified, a default value specified by the admin is used. This can only be used if the admin has turned on the adaptive count feature.
- **max_budget** (Optional[float]) – The maximum amount of budget that the system should spend while trying to achieve *target_relative_error*. If specified, the absolute amount of budget spent will not exceed *max_budget*. If the user specifies a value greater than the maximum budget for the computation set by the admin, system will use the admin-set maximum. If the system can achieve *target_relative_error* while spending less than *max_budget*, it will do so.

Returns Analysis object that can be executed using the *run* method.

Return type *CountAnalysisWithRI*

leapyear.analytics.**count_distinct**(*attr*, *dataset=None*, *drop_nulls=False*, *target_relative_error=None*, *max_budget=None*)

Analysis: Count the unique elements of an attribute.

Parameters

- **attr** (Union[Attribute, str, Sequence[Union[Attribute, str]]]) – The attribute or attributes to compute the distinct count of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) –

  Remove any records with null. Unique values associated with records containing nulls are not included in the count.

  target_relative_error A float value between 0 and 1 indicating the level of relative error that should be targeted for this computation. If specified, the system will attempt to ensure that the absolute value of the relative error between the randomized result and the true count is roughly *target_relative_error*. If this is not possible due to budget constraints set by the admin, the system will return a randomized result with the smallest randomization effect allowed. If not specified, a default value specified by the admin is used. This can only be used if the admin has turned on the adaptive count feature.
- **max_budget** (Optional[float]) – The maximum amount of budget that the system should spend while trying to achieve *target_relative_error*. If specified, the absolute amount of budget spent will not exceed *max_budget*. If the user specifies a value greater than the maximum budget for the computation set by the admin, system will use the admin-set maximum. If the system can achieve *target_relative_error* while spending less than *max_budget*, it will do so.

Returns Prepared analysis of the count.

Return type *Analysis*

leapyear.analytics.**count_distinct_rows**(*dataset*, *target_relative_error=None*, *max_budget=None*)

Analysis: Count the number of distinct rows in a dataset.

Parameters

- **dataset** (DataSet) – The input dataset.
- **target_relative_error** (Optional[float]) – A float value between 0 and 1 indicating the level of relative error that should be targeted for this computation. If specified, the system will attempt to ensure that the absolute value of the relative error between the randomized result and the true count is roughly *target_relative_error*. If this is not possible due to budget constraints set by the admin, the system will return a randomized result with the smallest randomization effect allowed. If not specified, a default value specified by the admin is used. This can only be used if the admin has turned on the adaptive count feature.
- **max_budget** (Optional[float]) – The maximum amount of budget that the system should spend while trying to achieve *target_relative_error*. If specified, the absolute amount of budget spent will not exceed *max_budget*. If the user specifies a value greater than the maximum budget for the computation set by the admin, system will use the admin-set maximum. If the system can achieve *target_relative_error* while spending less than *max_budget*, it will do so.

Returns Analysis for counting the number of distinct rows.

Return type *ScalarAnalysis*

leapyear.analytics.**mean**(*attr*, *dataset=None*, *drop_nulls=False*)

Analysis: Compute the mean of an attribute.

This analysis can be executed using the *run* method to compute the approximate mean of the attribute.

The user can request additional information about the computation with *run(rich_result=True)*. In this case, the *RandomizationInterval* object will be generated, likely including the precise value of the computation on the data sample.

Note: If the attribute is nullable, setting drop_nulls=True is necessary for the computation to go through.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the mean of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring NULL values.

Returns Analysis object that can be executed using the *run* method.

Return type ScalarAnalysisWithCI

leapyear.analytics.**sum**(*attr*, *dataset=None*, *drop_nulls=False*)

Analysis: Compute the sum of a numeric attribute.

This analysis can be executed using the *run* method to compute the approximate mean of the attribute.

The user can request additional information about the computation with *run(rich_result=True)*. In this case, the *RandomizationInterval* object will be generated, likely including the precise value of the computation on the data sample.

Note: If the attribute is nullable, setting drop_nulls=True is necessary for the copmutation to go through.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the sum of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.

- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring NULL values.

Returns Analysis object that can be executed using the *run* method.

Return type *ScalarAnalysisWithRI*

leapyear.analytics.**variance**(*attr*, *dataset=None*, *drop_nulls=False*)

Analysis: Compute the variance of an attribute.

This analysis can be executed using the *run* method to compute the approximate mean of the attribute.

The user can request additional information about the computation with *run(rich_result=True)*. In this case, the *RandomizationInterval* object will be generated, likely including the precise value of the computation on the data sample.

Note: If the attribute is nullable, setting drop_nulls=True is necessary for the copmutation to go through.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the variance of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring NULL values.

Returns Analysis object that can be executed using the *run* method.

Return type ScalarAnalysisWithCI

leapyear.analytics.**min**(*attr*, *dataset=None*, *drop_nulls=False*)

Analysis: Compute the minimum value of an attribute.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the min of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring nulls.

Returns Prepared analysis of the min.

Return type *ScalarAnalysis*

Note: The minimum reported is the 1/1000 quantile of the attribute.

When the attribute being analyzed has a very narrow range of possible values, the minimum returned may be inaccurate. As an extreme example, if the width of the interval of possible values of an attribute is less than 0.01, the minimum returned will be a fixed number that does not depend on the data distribution. For such cases, scale the attribute by 10/width, compute the minimum, and rescale the returned value by width/10.

The result of this analysis may be very different from the true minimum of the data sample in the following two scenarios:

1. When the underlying attribute distribution has significant outliers (e.g. a very long tail) - this is because the minimum computed is the 1/1000 quantile of the attribute, and

2. When the public lower bound is very different from the true minimum of the data sample - this is because differential privacy is aiming to minimize the effect of individual records on the output.

---

leapyear.analytics.**max**(*attr*, *dataset=None*, *drop_nulls=False*)

Analysis: Compute the maximum value of an attribute.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the max of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring nulls.

Returns Prepared analysis of the max.

Return type *Analysis*

Note: The maximum reported is the 999/1000 quantile of the attribute.

When the attribute being analyzed has a very narrow range of possible values, the maximum returned may be inaccurate. As an extreme example, if the width of the interval of possible values of an attribute is less than `0.01`, the maximum returned will be a fixed number that does not depend on the data distribution. For such cases, scale the attribute by `10/width`, compute the maximum, and rescale the returned value by `width/10`.

The result of this analysis may be very different from the true maximum of the data sample in the following two scenarios:

1. When the underlying attribute distribution has significant outliers (e.g. a very long tail) - this is because the maximum computed is the 999/1000 quantile of the attribute, and

2. When the public upper bound is very different from the true maximum of the data sample - this is because differential privacy is aiming to minimize the effect of individual records on the output.

---

leapyear.analytics.**median**(*attr*, *dataset=None*, *drop_nulls=False*)

Analysis: Compute the median value of an attribute.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the median of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring nulls.

Returns Prepared analysis of the median.

Return type *Analysis*

Note: When the attribute being analyzed has a very narrow range of possible values, the median returned may be inaccurate. As an extreme example, if the width of the interval of possible values of an attribute is less than `0.01`, the median returned will be a fixed number that does not depend on the data distribution. For such cases, scale the attribute by `10/width`, compute the median, and rescale the returned value by `width/10`.

---

leapyear.analytics.**quantile**(*q*, *attr*, *dataset=None*, *drop_nulls=False*)
Analysis: Compute a certain quantile q of an attribute.

Parameters

- **q** (float) – Quantile to compute, which must be between 0 and 1 inclusive.
- **attr** (Union[Attribute, str]) – The attribute to compute the quantile of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring nulls.

Returns Prepared analysis of the quantile.

Return type *Analysis*

Note: When the attribute being analyzed has a very narrow range of possible values, the quantile returned may be inaccurate. As an extreme example, if the width of the interval of possible values of an attribute is less than `0.01`, the quantile returned will be a fixed number that does not depend on the data distribution. For such cases, scale the attribute by `10/width`, compute the quantile, and rescale the returned value by `width/10`.

---

leapyear.analytics.**skewness**(*attr*, *dataset=None*, *drop_nulls=False*)
Analysis: Compute the skewness of an attribute.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the skewness of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring nulls.

Returns Prepared analysis of the skewness.

Return type *Analysis*

leapyear.analytics.**kurtosis**(*attr*, *dataset=None*, *drop_nulls=False*)
Analysis: Compute the excess kurtosis of an attribute.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the kurtosis of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring nulls.

Returns Prepared analysis of the kurtosis.

Return type *Analysis*

leapyear.analytics.**iqr**(*attr*, *dataset=None*, *drop_nulls=False*)
Analysis: Compute the interquartile range of an attribute.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the interquartile range of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string. When *attr* is an Attribute this field is ignored.
- **drop_nulls** (bool) – Whether to allow running on a nullable column, ignoring nulls.

Returns Prepared analysis of the iqr.

Return type *Analysis*

leapyear.analytics.**histogram**(*attr*, *dataset=None*, *bins=10*, *interval=None*)

Analysis: Compute the histogram of the attribute in the dataset.

Parameters

- **attr** (Union[Attribute, str]) – The attribute to compute the histogram of. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *attr* is a string.
- **bins** (int) – Number of bins between the bounds. (default=10)
- **interval** (Optional[Tuple[float, float]]) – The lower and upper bound of the histogram. Defaults to attribute bounds if None.

Returns Prepared analysis of the histogram.

Return type *Analysis*

leapyear.analytics.**histogram2d**(*x_attr*, *y_attr*, *dataset=None*, *x_bins=10*, *y_bins=10*, *x_range=None*, *y_range=None*)

Analysis: Compute the 2D histogram of two attributes in the dataset.

Parameters

- **x_attr** (Union[Attribute, str]) – The attribute to use to compute the first dimension of the histogram.. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **y_attr** (Union[Attribute, str]) – The attribute to use to compute the first dimension of the histogram.. Either a standalone attribute or the name of an attribute from a dataset provided by *dataset*.
- **dataset** (Optional[DataSet]) – The dataset to use when *x_attr* or *y_attr* are strings.
- **x_bins** (int) – Number of bins between the bounds in the first attribute.
- **y_bins** (int) – Number of bins between the bounds in the second attribute.
- **x_range** (Optional[Tuple[float, float]]) – The lower and upper bound of the first attribute for the histogram.
- **y_range** (Optional[Tuple[float, float]]) – The lower and upper bound of the second attribute for the histogram.

Returns Prepared analysis of the histogram.

Return type *Analysis*

leapyear.analytics.**correlation_matrix**(*xs*, *dataset*, ***, *center=True*, *scale=True*, ***kwargs*)

Analysis: Compute the correlation matrix of the set of attributes.

NOTE: This analysis does not require `run()`.

Parameters

- **xs** (Sequence[str]) – A list of attribute names to compute correlation matrix for.
- **dataset** (DataSet) – The DataSet containing these attributes.
- **center** (bool) – Whether to center the columns before computing correlation matrix. If False, proceed assuming the columns are already centered.
- **scale** (bool) – Whether to divide covariance matrix by number of rows. If False, do not divide.
- **max_timeout_sec** – Specifies the maximum amount of time (in seconds) the user is willing to wait for a response. If set to None, this function will poll the server indefinitely. If it is run with scale or center set to True, the timeout will be multiplied. Defaults to waiting forever.

Returns The correlation matrix.

Return type np.ndarray

leapyear.analytics.**covariance_matrix**(*xs*, *dataset*, ***, *center=True*, *scale=True*, ***kwargs*)

Analysis: Compute the covariance matrix of the set of attributes.

NOTE: This analysis does not require run().

Parameters

- **xs** (Sequence[str]) – A list of attribute names that are the features.
- **dataset** (DataSet) – The DataSet of the attributes.
- **center** (bool) – Whether to center the columns before compute the covariance matrix. If False, assume the columns are centered.
- **scale** (bool) – Whether to divide the matrix by number of rows. If False, do not divide.
- **max_timeout_sec** – Specifies the maximum amount of time (in seconds) the user is willing to wait for a response. If set to None, this function will poll the server indefinitely. If it is run with scale or center set to True, the timeout will be multiplied. Defaults to waiting forever.

Returns The covariance matrix.

Return type np.ndarray

leapyear.analytics.**describe**(*dataset*, *attributes=None*)

Describe the columns of the dataset for use in data exploration.

The describe function provides a way for an analyst to perform initial rough data exploration on a dataset. To get more accurate statistics, the individual functions *mean()*, *count()*, et cetera, are recommended. This function does not use the analysis cache of the other statistics functions.

Numeric columns are described by their count, mean, standard deviation, minimum, maximum and the quartiles. Categorical columns (factors and booleans) are described by their count, distinct count and frequency of the most frequent element.

Parameters

- **dataset** (DataSet) – The DataSet to be described
- **attributes** (Union[None, Attribute, str, Sequence[Union[Attribute, str]]]) – The attributes to describe. If a value is not provided, or None, describe all attributes.

Returns Prepared analysis for describing the dataset. Execute the analysis using the *run()* method.

Return type *DescribeAnalysis*

leapyear.analytics.**groupby_agg_view**(*dataset*, *attrs*, *agg_attr=None*, *agg_type=<GroupByAggType.COUNT: 1>*, ***, *max_groupby_agg_keys=100000000*, *size_threshold=None*, *agg_attr_and_type=None*)

Compute aggregate statistic within each group and output aggregate results.

Only groups with estimated size larger than `minimum_dataset_size` will be returned. This parameter can be set in *run*.

Parameters

- **dataset** (DataSet) – The DataSet to perform groupby and aggregation on
- **attrs** (Sequence[Union[Attribute, str]]) – List of attributes to perform groupby.
- **agg_attr** (Optional[str]) – Compute aggregate statistics on this column within each group
- **agg_type** (Union[GroupByAggType, str]) – Aggregate type. 'count', 'mean' or 'sum'.
- **max_groupby_agg_keys** (int) – This value prevents submitting computations that have a very large number of groupby keys. By default, it raises GroupbyAggTooManyKeysError if the number of groups exceeds 100000000.
- **size_threshold** (Optional[int]) – Deprecated: see `minimum_dataset_size` in the *run* method.
- **agg_attr_and_type** (Union[Tuple[Union[GroupByAggType, str], Optional[str]], Sequence[Tuple[Union[GroupByAggType, str], Optional[str]]], None]) – List of tuples (agg_type, agg_attr). Compute aggregate statistics defined by the agg_type on the column within each group. agg_type can be 'count', 'mean' or 'sum'.

Returns Analysis object that can be executed using *run* method to return aggregation results. The results can be accessed as a pandas dataframe using `.to_dataframe()`.

Return type *GroupbyAggAnalysis*

Note: privacy exposure estimate for this analysis is not supported.

Example

For each age group and gender, compute the mean income.

```
>>> groupby_agg_view(ds, ["AGE", "GENDER"], "INCOME", "mean").run(minimum_dataset_
↪size=1000)
```

For each week, compute the mean and total transaction amount.

```
>>> groupby_agg_view(ds, ["WEEK"], agg_attr_and_type=[("mean", "AMOUNT"), ("sum",
↪"AMOUNT")]).run()
```

Look at Randomization Intervals for each group (only for 'count' and 'sum').

```
>>> rr = groupby_agg_view(ds, ["WEEK"], "AMOUNT", "mean").run(rich_result=True)
>>> ri_dict = rr.metadata
>>> ri_dict
{
    (1, ): RandomizationInterval(...),
```

(continues on next page)

(continued from previous page)

```
    (2, ): RandomizationInterval(...)
    ...
}
>>> ri_dict[(1, )]
RandomizationInterval(...)
```

Look at Randomization Interval for multiple aggregate results.

```
>>> rr = groupby_agg_view(ds, ["YEAR", "WEEK"], agg_attr_and_type=[("mean",
↪"AMOUNT"), ("sum", "AMOUNT")]).run()
>>> ri_dict = rr.metadata
>>> ri_dict[(2020, 1)][0]
RandomizationInterval(...)
```

Machine Learning

Unsupervised learning

leapyear.analytics.**kmeans**(*xs*, *dataset*, *n_iters=10*, *n_clusters=3*)

Analysis: K-means clustering.

Identifies centers of clusters for a set of data points, by

1. Randomly initializing a chosen number of cluster centers (centroids) in the feature space
2. Associating each data point with the nearest centroid
3. Iteratively adjusting centroids to locations based on differentially private computation of the mean for each feature

Parameters

- **xs** (List[str]) – A list of attribute names that are the features.
- **dataset** (DataSet) – The DataSet of the attributes.
- **n_iters** (int) – Number of iterations to run k-means for
- **n_clusters** (int) – Number of clusters to generate

Returns Analysis object that can be executed using the `run()` method. Once executed, it would output clustering analysis results, such as centroids.

Return type *ClusteringAnalysis*

leapyear.analytics.**eval_kmeans**(*centroids*, *xs*, *dataset*)

Analysis: Evaluate the K-means model.

Evaluate the clustering model by computing the Normalized Intra Cluster Variance (NICV).

Parameters

- **centroids** (*ClusterModel*) – The model (generated using *kmeans*) to evaluate
- **xs** (List[str]) – A list of attribute names that are the features.
- **dataset** (DataSet) – The DataSet of the attributes.

Returns Analysis representing evaluation of a clustering model. It can be executed using the `run()` method to output evaluation metric value.

Return type *ScalarAnalysis*

leapyear.analytics.**pca**(*xs*, *dataset*, ***kwargs*)

Principal Component Analysis.

Compute the Principal Component Analysis (PCA) of the set of attributes using a differentially private algorithm.

NOTE: This analysis does not require `run()`.

Parameters

- **xs** (`List[str]`) – A list of attribute names representing features to be considered for this analysis.
- **dataset** (`DataSet`) – DataSet that includes these attributes.
- **max_timeout_sec** – Specifies the maximum amount of time (in seconds) the user is willing to wait for a response. If set to `None`, this function will poll the server indefinitely. If it is run with `scale` or `center` set to True, the timeout will be multiplied. Defaults to waiting forever.

Return type `Tuple[ndarray, ndarray]`

Returns

- *explained_variances* – Variance explained by each of the principal components - in other words, variance of each principal component coordinate when considered as feature on the input dataset.
- *pca_matrix* – Transformation matrix, that can be used to translate original features to principal component coordinates. If all principal components are included, this becomes a square matrix corresponding to orthogonal transformation (e.g. reflection).

  This matrix can be used to generate principal component features using *leapyear.dataset.DataSet.transform()* operation, as in:

  ```
  tfds = ds.transform(x_vars, pca_matrix, 'pca')
  ```

  NOTE: Signs may not match PCA transformation matrix computed by scikit-learn.

Supervised learning

leapyear.analytics.**basic_linreg**(*xs*, *y*, *dataset*, ***, *affine=True*, *l1reg=0.0*, *l2reg=1.0*, *parameter_bounds=None*)

Analysis: Linear regression.

Implements a differentially private algorithm to represent outcome (target) variable as a linear combination of selected features.

Note: To help ensure that the differentially private training process can effectively optimize regression coefficients, it's important to re-scale features (both dependent/target and independent/explanatory) to a similar domain (e.g. [0,1]). This can be done using `leapyear.feature.BoundsScaler` and will help ensure that the domain being searched for the coefficient will include the optimal model. See LeapYear guides for this and other recommendations on training accurate regressions.

Note: Differentially private regressions aim to optimize models for predictive tasks, while protecting sensitive information from being learned from the trained model. They are not guaranteed to result in coefficients close

to those that a non-differentially private algorithm would learn. In other words, while regressions trained with differential privacy may excel at predictive tasks, keep in mind that these were not designed for inference.

Please see the guides for more details on using regressions in LeapYear.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **affine** (bool) – If True, fit an intercept term.
- **l1reg** (float) – The L1 regularization. Default value: 0.0.
- **l2reg** (float) – The L2 regularization. Default value: 1.0. Must be at least 0.0001 to limit the randomization effect for models optimized via objective perturbation.
- **parameter_bounds** (Optional[List[Tuple[float, float]]]) – Restriction on the model parameters, including the intercept. Required for differential privacy. Default value: -10.0 to 10.0 for each parameter

Returns Analysis representing the regression problem. It can be executed using the run() method to output calibrated model.

Return type *GenLinAnalysis*

leapyear.analytics.**generalized_linreg**(*xs*, *y*, *dataset*, ***, *affine=True*, *l2reg=1.0*, *weight=None*, *offset=None*, *max_iters=25*, *family='gaussian'*, *link='identity'*, *link_power=0*, *variance_power=1*)

Analysis: Generalized linear regression.

Implements a differentially private algorithm to represent outcome (target) variable as a linear combination of selected features. This computation is based on the iterative weighted least squares algorithm. Trains using the "glm" algorithm.

Available generalizations include:

- offset of outputs based on pre-existing model - this enables modeling of residual
- use of alternative link functions applied to the linear combination of features
- application of regularization and weights during model optimization.

Note: Differentially private regressions aim to optimize models for predictive tasks, while protecting sensitive information from being learned from the trained model. They are not guaranteed to result in coefficients close to those that a non-differentially private algorithm would learn. In other words, while regressions trained with differential privacy may excel at predictive tasks, keep in mind that these were not designed for inference.

Please see the guides for more details on using regressions in LeapYear.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of Attributes or attribute names to be used as features.
- **y** (Union[Attribute, str]) – The Attribute or attribute name of the target.

- **dataset** (DataSet) – The DataSet of the attributes.
- **affine** (bool) – True if the algorithm should fit an intercept term.
- **l2reg** (float) – The L2 regularization parameter. Must be non-negative.
- **weight** (Union[Attribute, str, None]) – Optional column used to weight each sample. Implies generalized regression.
- **offset** (Union[Attribute, str, None]) – Optional column for offset in offset regression. Implies generalized regression.
- **max_iters** (Optional[int]) – Optional maximum number of iterations for fitting the regression. Note that regardless of this setting, the system would often stop before reaching max_iterations - e.g. after a single iteration. In such cases, higher value for max_iterations may lead to less privacy allocated to each iteration, and ultimately, higher randomization effect.
- **family** (Optional[str]) – Optional distribution of the label. Implies generalized regression. Possible values here are 'gaussian' (the default), 'poisson', 'gamma' and 'tweedie'.
- **link** (Optional[str]) – Optional link function between mean of label distribution and prediction. Implies generalized regression. Possible values depend on family: 'gaussian' supports only 'identity' (default), 'log' and 'inverse'; 'poisson' supports only 'log' (default), 'identity' and 'sqrt'; 'gamma' supports only 'inverse' (default), 'identity' and 'log'. There is no link function for the 'tweedie' family, use variance_power and link_power parameters instead.
- **link_power** (Optional[int]) – For the 'tweedie' distribution only, the exponent of the link function. Default value is 0, which is equivalent to 'identity' link.
- **variance_power** (Optional[int]) – For the 'tweedie' distribution only, the exponent of the variance. Default value is 1, which is equivalent to 'gaussian' family.

Returns Analysis of the regression problem, which could be executed using `run()` function to output calibrated model.

Return type *GenLinAnalysis*

leapyear.analytics.**logreg**(*xs*, *y*, *dataset*, *affine=True*, *l1reg=0.0*, *l2reg=1.0*)

Analysis: Logistic regression.

Implements a differentially private algorithm to represent outcome (target) variable as a logit-transformation of a linear combination of selected features. Trains using the "basic" algorithm.

Available generalizations include

- regularization applied during model optimization

Note: To help ensure that the differentially private training process can effectively optimize regression coefficients, it's important to re-scale features (both dependent/target and independent/explanatory) to a similar domain (e.g. [0,1]). This can be done using `leapyear.feature.BoundsScaler` and will help ensure that the domain being searched for the coefficient will include the optimal model. See LeapYear guides for this and other recommendations on training accurate regressions.

Note: Differentially private regressions aim to optimize models for predictive tasks, while protecting sensitive information from being learned from the trained model. They are not guaranteed to result in coefficients close

to those that a non-differentially private algorithm would learn. In other words, while regressions trained with differential privacy may excel at predictive tasks, keep in mind that these were not designed for inference.

Please see the guides for more details on using regressions in LeapYear.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **affine** (bool) – If True, fit an intercept term.
- **l1reg** (float) – The L1 regularization. Default value: 0.0.
- **l2reg** (float) – The L2 regularization. Default value: 1.0.

Returns Analysis training the logistic regression model. It can be executed using the run() method to output the calibrated model.

Return type *GenLinAnalysis*

leapyear.analytics.**generalized_logreg**(*xs*, *y*, *dataset*, ***, *affine=True*, *l1reg=1.0*, *l2reg=1.0*, *weight=None*, *offset=None*, *max_iters=25*, *link='logit'*)

Analysis: Generalized logistic regression.

Implements a differentially private algorithm to represent the outcome (target) variable as a logit-transformation of a linear combination of selected features. This computation is based on the iterative weighted least squares algorithm. Trains using the "glm" algorithm.

Available generalizations include:

- offset of outputs based on pre-existing model - this enables modeling of residual
- use of alternative link functions applied to the linear combination of features
- application of regularization and weights during model optimization.

Note: Differentially private regressions aim to optimize models for predictive tasks, while protecting sensitive information from being learned from the trained model. They are not guaranteed to result in coefficients close to those that a non-differentially private algorithm would learn. In other words, while regressions trained with differential privacy may excel at predictive tasks, keep in mind that these were not designed for inference.

Please see the guides for more details on using regressions in LeapYear.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of Attributes or attribute names to be used as features.
- **y** (Union[Attribute, str]) – The Attribute or attribute name of the target.
- **dataset** (DataSet) – The DataSet of the attributes.
- **affine** (bool) – True if the algorithm should fit an intercept term.
- **l2reg** (float) – The L2 regularization parameter. Must be non-negative.

- **weight** (Union[Attribute, str, None]) – Optional column used to weight each sample. Implies generalized regression.
- **offset** (Union[Attribute, str, None]) – Optional column for offset in offset regression. Implies generalized regression.
- **max_iters** (Optional[int]) – Optional maximum number of iterations for fitting the regression. Note that regardless of this setting, the system would often stop before reaching max_iterations - e.g. after a single iteration. In such cases, higher value for max_iterations may lead to less privacy allocated to each iteration, and ultimately, higher randomization effect.
- **link** (Optional[str]) – Optional link function between mean of label distribution and prediction. Implies generalized regression. Possible values are 'logit' (default), 'probit' and 'cloglog'.

Returns Analysis of the regression problem, which could be executed using `run()` function to output calibrated model.

Return type *GenLinAnalysis*

leapyear.analytics.**gradient_boosted_tree_classifier**(*xs*, *y*, *dataset*, *max_depth=3*, *max_iters=5*, *max_bins=32*)

Analysis: Gradient boosted tree classifier.

This analysis trains a randomized variant of gradient boosted tree classifier to predict a BOOLEAN outcome (target).

The algorithm works by iteratively training individual decision trees to predict a "residual" of the model built so far, and then integrating each newly built decision tree into the ensemble model to better predict the probability of the positive label.

Weights are used at different stages:

- during training of individual decision trees, to focus attention on the areas where the model consistently underperforms, and
- when combining individual decision trees to predict probability of the positive label.

Calibrated level of randomization is applied to individual leaves of the decision trees to help protect privacy of the individual records used for model training.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attributes or attribute names that are used as explanatory features for the analysis. Each attribute must be either BOOL, INT, REAL or FACTOR. Nullable types are not supported and must be converted to non-nullable - e.g. via *coalesce*.
- **y** (Union[Attribute, str]) – The attribute or attribute name that is used as an outcome (target) of the classification model. Must be BOOLEAN type, as only binary classification models are supported. Nullable types are not supported and must be converted to non-nullable - e.g. via *coalesce*.
- **dataset** (DataSet) – The DataSet containing both explanatory features and outcome attributes.
- **max_depth** (int) – The maximum depth (or height) of any tree in the ensemble produced by the algorithm. Default: 3
- **max_iters** (int) – The maximum number of iterations of the algorithm. This corresponds to the maximum number of individual decision trees in the ensemble. Default: 5

- **max_bins** (int) – The maximum number of bins for features used in constructing trees. Default: 32

Note: Maximum number of bins should be set to no less than the number of distinct possible values of the FACTOR attributes used as explanatory features.

Returns Analysis that will train the gradient boosted tree classifier. It can be executed using the `run()` method.

Return type *GradientBoostedTreeClassifierModelAnalysis*

See also:

***Gradient tree boosting*:** https://en.wikipedia.org/wiki/Gradient_boosting#Gradient_tree_boosting

leapyear.analytics.**random_forest**(*xs*, *y*, *dataset*, *n_trees=100*, *height=3*)

Analysis: Random Forest Classifier.

Generate a random forest model to predict probability associated with each target class.

Random forests combine many decision trees in order to reduce the risk of overfitting.

Each decision tree is developed on a random subset of observations - and is limited to prescribed height.

Individual node split decisions are made to maximize split value (or gain) - with a variation that a differentially private algorithm is used to count the number of observations belonging to each target class on both sides of the split.

Specifically, split value (or gain) is defined as reduction in combined Gini impurity measure, associated with introducing the split for a given parent node. Here

- Gini impurity for any given node (parent or child) is calculated based on distribution of observations within the node across different outcome (target) classes
- To compute combined impurity of the pair of nodes, individual node impurities for the two children nodes are averaged proportionately to their share of observations

Categorical features are typically handled by evaluating various splits corresponding to random subsets of the available categories.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are used as features for explanatory analysis.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome (target).
- **dataset** (DataSet) – The DataSet containing both explanatory features and outcome attributes.
- **n_trees** (int) – The number of trees to use in the random forest. Default: 100
- **height** (int) – The maximum height of the trees. Default: 3

Returns Analysis training the random forest model. It can be executed using the `run()` method to output the analysis results which include the calibrated random forest model, feature importance statistics, etc.

Return type *ForestModelClassifierAnalysis*

See also:

***Gini impurity*:** https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity

leapyear.analytics.**regression_trees**(*xs*, *y*, *dataset*, *n_trees=100*, *height=3*)
Analysis: Random Forest Regressor (regression trees).

Generate a regression trees model to predict value of target variable.

Regression trees are built similarly to random forests, but instead of predicting the probability that the target variable takes a certain categorical value (i.e., classification), they predict a real value of the target variable (i.e., regression).

The impurity metric in this case is the variance of the target variable for the datapoints that fall into the current node's partition.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are used as features for explanatory analysis.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome (target).
- **dataset** (DataSet) – The DataSet containing both explanatory features and target attribute.
- **n_trees** (int) – The number of trees to use in the random forest. Default: 100.
- **height** (int) – The maximum height of the trees. Default: 3.

Returns Analysis training the regression trees model. It can be executed using the `run()` method to output the analysis results which include the calibrated random forest model, feature importance statistics, etc.

Return type *ForestModelRegressionAnalysis*

leapyear.analytics.**eval_linreg**(*glm*, *xs*, *y*, *dataset*, *metric='mse'*)
Analysis: Evaluate a linear regression model.

Parameters

- **glm** (*GLM*) – The model (generated using *linreg*) to evaluate
- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **metric** (Union[str, Metric]) – Linear regression evaluation metric: 'mse'/'mean_squared_error' or 'mae'/'mean_absolute_error'.

Note: During the calculation of `mse` and `mae` metrics, the individual values of absolute error are restricted to be no greater than the length of the interval of possible values of the target attribute, as seen in the dataset schema. For example, if the target attribute contains values in the interval [-50, 50], then the absolute error of any individual prediction used in computing the mean will be no greater than 100.

Returns Analysis representing evaluation of a regression model. It can be executed using the `run()` method to output evaluation metric value.

Return type *ScalarAnalysis*

leapyear.analytics.**eval_logreg**(*glm*, *xs*, *y*, *dataset*, *metric='accuracy'*)
Analysis: Evaluate a logistic regression model.

Parameters

- **glm** (*GLM*) – The model (generated using *logreg*) to evaluate
- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **metric** (Union[str, Metric]) – Logistic regression evaluation metric. Examples: 'accuracy', 'logloss', 'auroc', 'aupr'.

Returns Analysis representing evaluation of a logistic regression model. It can be executed using the `run()` method to output evaluation metric value.

Return type *ScalarAnalysis*

leapyear.analytics.**eval_gbt_classifier**(*gbt*, *xs*, *y*, *dataset*, *metric='accuracy'*)

Analysis: Evaluate a gradient boosted tree (GBT) classifier model.

Parameters

- **gbt** (*GradientBoostedTreeClassifier*) – The model to evaluate.
- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **metric** (Union[str, Metric]) – GBT evaluation metric. Currently only supports 'accuracy'.

Returns Analysis representing evaluation of a GBT classifier model. It can be executed using the `run()` method to output the value of the evaluation metric.

Return type *ScalarAnalysis*

leapyear.analytics.**eval_random_forest**(*rf*, *xs*, *y*, *dataset*, *metric='accuracy'*)

Analysis: Evaluate a random forest model.

Parameters

- **rf** (*RandomForestClassifier*) – The model (generated using *random_forest*) to evaluate
- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **metric** (Union[str, Metric]) – Forest evaluation metric. Examples: 'logloss', 'accuracy', 'auroc', 'aupr'

Returns Analysis representing evaluation of a random forest model. It can be executed using the `run()` method to output evaluation metric value.

Return type *ScalarAnalysis*

leapyear.analytics.**eval_regression_trees**(*rf*, *xs*, *y*, *dataset*, *metric='mse'*)

Analysis: Evaluate a regression trees model.

Parameters

- **rf** (*RandomForestClassifier*) – The model (generated using *regression_trees*) to evaluate

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **metric** (Union[str, Metric]) – Model evaluation metric. Examples: 'mse'/'mean_squared_error' or 'mae'/'mean_absolute_error'.

Note: During the calculation of mse and mae metrics, the individual values of absolute error are restricted to be no greater than the length of the interval of possible values of the target attribute, as seen in the dataset schema. For example, if the target attribute contains values in the interval [-50, 50], then the absolute error of any individual prediction used in computing the mean will be no greater than 100.

Returns Analysis representing evaluation of a regression trees model. It can be executed using the run() method to output evaluation metric value.

Return type *ScalarAnalysis*

leapyear.analytics.**roc**(*model*, *xs*, *y*, *dataset*, *thresholds=5*)

Compute the ConfusionCurves.

For each threshold value, compute the normalized confusion matrix using the model. The confusion matrix contains the true positive rate, the true negative rate, the false positive rate and the false negative rate.

Parameters

- **model** (Union[*GLM*, *RandomForestClassifier*]) – The model to evaluate the confusion curves on.
- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **thresholds** (Union[int, Sequence[float]]) – If int, then generate approximatly *thresholds* (rounded to the closest power of 2) number of thresholds using recursive medians. If a sequence of floats, then use the list as the thresholds.

Returns Analysis of the confusion curve, which can be executed using the run() method to output various evaluation metrics.

Return type *ConfusionModelAnalysis*

leapyear.analytics.**cross_val_score_linreg**(*xs*, *y*, *dataset*, ***, *affine=True*, *l1reg=1.0*, *l2reg=1.0*, *cv=3*, *metric='mean_squared_error'*, *parameter_bounds=None*)

Analysis: Compute the linear regression cross validation score of the set of attributes.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **affine** (bool) – If True, fit an intercept term.
- **l1reg** (float) – The L1 regularization. Default value: 1.0.

- **l2reg** (float) – The L2 regularization. Default value: 1.0. Must be at least 0.0001 to limit the randomization effect.
- **cv** (int) – Number of folds in k-fold cross validation.
- **metric** (Union[str, Metric]) – The metric for evaluating the regression. Examples: 'mae', 'mse', 'r2'.
- **parameter_bounds** (Optional[List[Tuple[float, float]]]) – Restriction on the model parameters, including the intercept. Required for differential privacy. Default value: -10.0 to 10.0 for each parameter

Returns Analysis of the cross-validation scores for the regression model. It can be executed using the run() method to generate cross-validation results.

Return type *CrossValidationAnalysis*

leapyear.analytics.**cross_val_score_logreg**(*xs*, *y*, *dataset*, *cv=3*, *affine=True*, *l1reg=0.1*, *l2reg=0.1*, *metric='accuracy'*)

Analysis: Compute the logistic regression cross validation score of the set of attributes.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **affine** (bool) – If True, fit an intercept term.
- **l1reg** (float) – The L1 regularization. Default value: 0.1.
- **l2reg** (float) – The L2 regularization. Default value: 0.1. Must be at least 0.0001 to limit the randomization effect.
- **cv** (int) – Number of folds in k-fold cross validation.
- **metric** (Union[str, Metric]) – The metric for evaluating the logistic regression. Examples: 'accuracy', 'logloss', 'auroc', 'aupr'.

Returns Analysis of the cross-validation scores for the regression model. It can be executed using the run() method to generate cross-validation results.

Return type *CrossValidationAnalysis*

leapyear.analytics.**cross_val_score_random_forest**(*xs*, *y*, *dataset*, *n_trees=100*, *height=3*, *cv=3*, *metric='mean_squared_error'*)

Analysis: Compute the random forest cross validation score of the set of attributes.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **n_trees** (int) – Number of trees.
- **height** (int) – Maximum height of trees.
- **cv** (int) – Number of folds in k-fold cross validation
- **metric** (Union[str, Metric]) – The metric for evaluating the regression

Returns Analysis of the cross-validation scores for the random forest model. It can be executed using the `run()` method to generate cross-validation results.

Return type *CrossValidationAnalysis*

`leapyear.analytics.`**`cross_val_score_regression_trees`**(*xs*, *y*, *dataset*, *n_trees=100*, *height=3*, *cv=3*, *metric='mean_squared_error'*)

Analysis: Compute the regression trees cross validation score of the set of attributes.

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attribute names that are the features.
- **y** (Union[Attribute, str]) – The attribute name that is the outcome.
- **dataset** (DataSet) – The DataSet of the attributes.
- **n_trees** (int) – Number of trees.
- **height** (int) – Maximum height of trees.
- **cv** (int) – Number of folds in k-fold cross validation
- **metric** (Union[str, Metric]) – The metric for evaluating the regression

Returns Analysis of the cross-validation scores for the regression trees model. It can be executed using the `run()` method to generate cross-validation results.

Return type *CrossValidationAnalysis*

`leapyear.analytics.`**`hyperopt_linreg`**(*xs*, *y*, *dataset*, ***, *cv*, *train_fraction*, *metric*, *n_iter=100*, *l1_bounds=(1e-10, 10000000000.0)*, *l2_bounds=(1e-10, 10000000000.0)*, *fit_intercept=None*, *parameter_bounds=None*)

Analysis: Hyperparameter optimization for linear regression.

Calibrate a linear regression model by optimizing its cross-validation score with respect to model hyperparameters - *L_1* and *L_2* regularization parameters and presence of intercept.

See below for a pseudo-code of the algorithm:

```
Split the dataset into ds_train_val/ds_holdout based on train_fraction.
Use k-fold cross validation to split ds_train_val into cv pairs (ds_train, ds_
↪val).
Initialize cv_history = []
For 1..n_iter
    pick a set of hyperparameters (hp) to test based on cv_history.
    use hp to calibrate a model on each cross-validation set
    evaluate it on corresponding sample set-aside for cross-validation
    compute an average cv score and append it to cv_history.
Pick the hyper parameters with the best cv score.
Train a model using the complete ds_train_val data set.
Evaluate the model on the holdout data set.
Return resulting model and its performance on the holdout set.
```

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attributes that are the features.
- **y** (Union[Attribute, str]) – The target attribute.
- **dataset** (DataSet) – The dataset containing the attributes.

- **cv** (int) – The number of cross-validation steps to perform for each candidate set of hyperparameters.
- **train_fraction** (float) – The fraction of the dataset to use set aside for model training and cross-validation – to be split further according to k-fold cross-validation strategy.
- **metric** (Union[str, Metric]) – Model performance metric to optimize. Examples: 'mean_squared_error', 'mean_absolute_error', 'r2'
- **n_iter** (int) – The number of optimization steps. Default: 100
- **l1_bounds** (Tuple[float, float]) – Lower and upper bounds for l1 regularization. Default: (1E-10, 1E10)
- **l2_bounds** (Tuple[float, float]) – Lower and upper bounds for l2 regularization. Default: (1E-10, 1E10)
- **fit_intercept** (Optional[bool]) – If None, search will consider both options.
- **parameter_bounds** (Optional[List[Tuple[float, float]]]) – Restriction on the model parameters, including the intercept. Required for differential privacy. Default value: `-10.0 to 10.0 for each parameter`

Returns

Analysis object representing the model calibration process with hyperparameter optimization. It can be executed using the `run()` method to output the analysis results, including

a) model calibrated with recommended hyperparameters and

b) its performance on the holdout dataset.

Return type *HyperOptAnalysis*

See also:

***Paper with hyperparameter optimization algorithm*:** http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

leapyear.analytics.**hyperopt_logreg**(*xs*, *y*, *dataset*, *cv*, *train_fraction*, *metric*, *n_iter=100*, *l1_bounds=(1e-10, 10000000000.0)*, *l2_bounds=(1e-10, 10000000000.0)*, *fit_intercept=None*)

Analysis: Hyperparameter optimization for logistic regression.

Calibrate a logistic regression model by optimizing its cross-validation score with respect to model hyperparameters - *L_1* and *L_2* regularization parameters and presence of intercept.

See below for a pseudo-code of the algorithm:

```
Split the dataset into ds_train_val/ds_holdout based on train_fraction.
Use k-fold cross validation to split ds_train_val into cv pairs (ds_train, ds_
↪val).
Initialize cv_history = []
For 1..n_iter
    pick a set of hyperparameters (hp) to test based on cv_history.
    use hp to calibrate a model on each cross-validation set
    evaluate it on corresponding sample set-aside for cross-validation
    compute an average cv score and append it to cv_history.
Pick the hyper parameters with the best cv score.
Train a model using the complete ds_train_val data set.
Evaluate the model on the holdout data set.
Return resulting model and its performance on the holdout set.
```

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attributes that are the features.
- **y** (Union[Attribute, str]) – The target attribute.
- **dataset** (DataSet) – The dataset containing the attributes.
- **cv** (int) – The number of cross-validation steps to perform for each candidate set of hyperparameters.
- **train_fraction** (float) – The fraction of the dataset to set aside for model training and cross-validation – to be split further according to k-fold cross-validation strategy.
- **metric** (Union[str, Metric]) – Model performance metric to optimize. Examples: 'accuracy', 'logloss', 'auroc', 'aupr'.
- **n_iter** (int) – The number of optimization steps. Default: 100
- **l1_bounds** (Tuple[float, float]) – Lower and upper bounds for l1 regularization. Default: (1E-10, 1E10)
- **l2_bounds** (Tuple[float, float]) – Lower and upper bounds for l2 regularization. Default: (1E-10, 1E10)
- **fit_intercept** (Optional[bool]) – If None, search will consider both options.

Returns

Analysis object representing the model calibration process with hyperparameter optimization. It can be executed using the run() method to output the analysis results, including

a) model calibrated with recommended hyperparameters and

b) its performance on the holdout dataset.

Return type *HyperOptAnalysis*

See also:

***Paper with hyperparameter optimization algorithm*:** http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

leapyear.analytics.**hyperopt_rf**(*xs*, *y*, *dataset*, *cv*, *train_fraction*, *metric*, *n_iter=100*, *max_trees=1000*, *max_depth=20*)

Analysis: Hyperparameter optimization for a random forest model.

Calibrate a random forest model by optimizing its cross-validation score with respect to model hyperparameters - number of trees and individual tree depth (or height) limit.

See below for a pseudo-code of the algorithm:

```
Split the dataset into ds_train_val/ds_holdout based on train_fraction.
Use k-fold cross validation to split ds_train_val into cv pairs (ds_train, ds_
↪val).
Initialize cv_history = []
For 1..n_iter
    pick a set of hyperparameters (hp) to test based on cv_history.
    use hp to calibrate a model on each cross-validation set
    evaluate it on corresponding sample set-aside for cross-validation
    compute an average cv score and append it to cv_history.
Pick the hyper parameters with the best cv score.
Train a model using the complete ds_train_val data set.
```

(continues on next page)

(continued from previous page)

```
Evaluate the model on the holdout data set.
Return resulting model and its performance on the holdout set.
```

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attributes that are the features.
- **y** (Union[Attribute, str]) – The target attribute.
- **dataset** (DataSet) – The dataset containing the attributes.
- **cv** (int) – The number of cross-validation steps to perform for each candidate set of hyperparameters.
- **train_fraction** (float) – The fraction of the dataset to set aside for model training and cross-validation – to be split further according to k-fold cross-validation strategy.
- **metric** (Union[str, Metric]) – The metric to optimize. Examples: 'accuracy', 'logloss', 'auroc', 'aupr'
- **n_iter** (int) – The number of optimization steps. Default: 100
- **max_trees** (int) – Maximum number of trees. Default: 1000
- **max_depth** (int) – Maximum tree depth. Default: 20

Returns

Analysis object representing the model calibration process with hyperparameter optimization. It can be executed using the `run()` method to output the analysis results, including

a) model calibrated with recommended hyperparameters and

b) its performance on the holdout dataset.

Return type *HyperOptAnalysis*

See also:

***Paper with hyperparameter optimization algorithm*:** http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

leapyear.analytics.**hyperopt_regression_trees**(*xs*, *y*, *dataset*, *cv*, *train_fraction*, *metric*, *n_iter=100*, *max_trees=1000*, *max_depth=20*)

Analysis: Hyperparameter optimization for a regression trees model.

Calibrate a regression trees model by optimizing its cross-validation score with respect to model hyperparameters - number of trees and individual tree depth (or height) limit.

See below for a pseudo-code of the algorithm:

```
Split the dataset into ds_train_val/ds_holdout based on train_fraction.
Use k-fold cross validation to split ds_train_val into cv pairs (ds_train, ds_
↪val).
Initialize cv_history = []
For 1..n_iter
    pick a set of hyperparameters (hp) to test based on cv_history.
    use hp to calibrate a model on each cross-validation set
    evaluate it on corresponding sample set-aside for cross-validation
    compute an average cv score and append it to cv_history.
```

(continues on next page)

(continued from previous page)

```
Pick the hyper parameters with the best cv score.
Train a model using the complete ds_train_val data set.
Evaluate the model on the holdout data set.
Return resulting model and its performance on the holdout set.
```

Parameters

- **xs** (List[Union[Attribute, str]]) – A list of attributes that are the features.
- **y** (Union[Attribute, str]) – The target attribute.
- **dataset** (DataSet) – The dataset containing the attributes.
- **cv** (int) – The number of cross-validation steps to perform for each candidate set of hyperparameters.
- **train_fraction** (float) – The fraction of the dataset to set aside for model training and cross-validation – to be split further according to k-fold cross-validation strategy.
- **metric** (Union[str, Metric]) – The metric to optimize. Examples: 'mae', 'mse', 'r2'
- **n_iter** (int) – The number of optimization steps. Default: 100
- **max_trees** (int) – Maximum number of trees. Default: 1000
- **max_depth** (int) – Maximum tree depth. Default: 20

Returns

Analysis object representing the model calibration process with hyperparameter optimization. It can be executed using the `run()` method to output the analysis results, including

a) model calibrated with recommended hyperparameters and

b) its performance on the holdout dataset.

Return type *HyperOptAnalysis*

See also:

***Paper with hyperparameter optimization algorithm*:** http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

Context Managers

leapyear.analytics.**ignore_computation_cache**()

Temporary context where computations do not utilize the computation cache.

The computation cache is intended to prevent wasting privacy exposure on queries that were previously computed. Entering this context manager will disable the use of the cache and allow repeated computations to return different differentially private answers.

Example

An administrator wants to run a count multiple times to estimate the random distribution of responses around the precise value.

```
>>> with ignore_computation_cache():
>>>     results = [la.count_rows(table).run() for _ in range(10)]
```

See also:

- To override the behavior for a single computation, see the `cache` keyword argument in *`run()`* or *`check()`*.
- The `default_analysis_caching` keyword argument in *`Client`* will temporarily be overwritten within this context manager.

Note: Additional permissions may be required to disable the computation cache.

Return type None

`leapyear.analytics.`**`precise_computations`**(*precise=True*)

Temporary context specifying if the computations are precise or not.

Computations requested within this context would be executed in precise mode, where differential privacy is not applied.

Parameters **`precise`** (`bool`) – True to enable precise computations within the context, False to disable them.

Example

An administrator wants to compare the responses of a number of computations with and without differential privacy applied. Precise mode may not be available for all computations.

```
>>> def my_computation():
>>>     symbols = ("APPL", "GOOG", "MSFT"):
>>>     return [la.count_rows(table.where(col("SYM") == lit(val)).run() for val␣
↪in symbols]
>>>
>>> res_dp = my_computation()
>>> with precise_computations():
>>>    res_no_dp = my_computation()
```

See also:

- To override the behavior for a single computation, see the `precise` keyword argument in *`run()`* or *`check()`*.

Note: Additional permissions may be necessary to enable precise computations.

Return type None

Save/Load Models

LeapYear save and load machine learning models utilities.

leapyear.ml_import_export.**save**(*model*, *path_or_fd*)

Save machine learning models in json to either a file or a file-like object.

Parameters

- **model** (Union[*ClusterModel*, *GLM*, *GradientBoostedTreeClassifier*, *RandomForestClassifier*, *RandomForestRegressor*, RichResult[Union[*ClusterModel*, *GLM*, *GradientBoostedTreeClassifier*, *RandomForestClassifier*, *RandomForestRegressor*], Any]]) – Any machine learning model executed using the `run()` method.
- **path** – The path where to save the file in the file system or a descriptor for an in-memory stream.

Example

```
>>> from leapyear.ml_import_export import save
>>> save(model, 'model.json')
```

Return type None

leapyear.ml_import_export.**load**(*path_or_fd*, *expected_model_type=None*, ***kwargs*)

Load machine learning models from a file-like object.

Parameters

- **path** – The path in the file system or an in-memory stream from where to load the model.
- **expected_mode_type** – If None it won't check that the model being loaded is of the type specified. Otherwise it checks that the model loaded is of the type expected.
- **rf_type** – When loading RandomForest models with serialization number 0, setting this to "classification" or "regression" will load the model as a RandomForestClassifier or RandomForestRegressor objects, respectively. If not specified, a RandomForest model will raise an error. The value is ignored for all other model types.

Examples

1. Loading a previously saved model of unspecified type

```
>>> from leapyear.ml_import_export import load
>>> model = load('model.json')
```

2. Loading a previously saved RandomForestClassifier model

```
>>> from leapyear.ml_import_export import load
>>> model = load('random_forest_classifier.json', RandomForestClassifier)
```

Return type Union[*ClusterModel*, *GLM*, *GradientBoostedTreeClassifier*, *RandomForestClassifier*, *RandomForestRegressor*]

1.4.9 Module leapyear.analytics.classes

Classes related to LeapYear analyses.

Analysis Classes

This section documents classes that define and process analyses in the LeapYear system.

Main Classes

class leapyear.analytics.classes.**Analysis**(*analysis*, *relation*)

Any analysis that can be performed on the LeapYear server.

It contains an analysis and a relation on which the analysis should be performed.

check(*, *cache=None*, *allow_max_budget_allocation=None*, *precise=None*, ***kwargs*)

Check the analysis for errors.

If any errors are present, the function will raise a descriptive error. If no errors are found, then the function will return self.

Return type ForwardRef

run(*, *detach: None*, *cache: Optional[bool] = None*, *allow_max_budget_allocation: Optional[bool] = None*, *precise: Optional[bool] = None*, *rich_result: bool = False*, *max_timeout_sec: Optional[float] = None*, *minimum_dataset_size: Optional[int] = None*, ***kwargs: Any*) → *leapyear.analytics.classes.AsyncAnalysis*[_Result, _Model, _ModelMetadata]

run(*, *rich_result: None*, *cache: Optional[bool] = None*, *allow_max_budget_allocation: Optional[bool] = None*, *precise: Optional[bool] = None*, *max_timeout_sec: Optional[float] = None*, *minimum_dataset_size: Optional[int] = None*, ***kwargs: Any*) → leapyear.analytics.classes.RichResult[_Model, _ModelMetadata]

run(*, *cache: Optional[bool] = None*, *allow_max_budget_allocation: Optional[bool] = None*, *precise: Optional[bool] = None*, *max_timeout_sec: Optional[float] = None*, *minimum_dataset_size: Optional[int] = None*, ***kwargs: Any*) → _Model

Run analysis.

Parameters

- **detach** – If `True` when the analysis is sent to the server, it will return immediately with an AsyncAnalysis object. The analysis will be evaluated in the background on the server. The client can check the result of the analysis later using the AsyncAnalysis object.
- **cache** – If `True`, then the first time this analysis is executed on the LeapYear server, the result will be cached. Subsequent calls to the identical analysis (with `cache=True`) will fetch the cached version and not contribute to the security cost. If `None`, the default caching behavior will be obtained from the connection.
- **allow_max_budget_allocation** – Default is `True`. If `False`, raise an *leapyear.exceptions.DataSetTooSmallException* when the randomness calibration system would run an analysis with the maximum privacy exposure per computation. If `None`, the default value will be obtained from the connection.
- **precise** – When `True`, request an answer with no noise added to the computation.
- **rich_result** – When `True`, return a result with additional (potentially analysis-specific) metadata, including the privacy exposure expended in the process of performing the analysis. Defaults to `False`.

- **max_timeout_sec** – When `detach=False`, specifies the maximum amount of time (in seconds) the user is willing to wait for a response. If set to `None`, the analysis will poll the server indefinitely. When computing on big data or long-running machine learning tasks, we recommend using the `detach=True` feature and use the functions provided in `AsyncAnalysis`. Defaults to waiting forever.
- **minimum_dataset_size** – When `minimum_dataset_size` is set, prevent computations on data sets that have fewer rows than the specified value. We recommend using this when an analysis could filter down to a small number of records, potentially consuming more privacy budget than is desired. Setting this will spend a small amount of privacy budget to estimate the number of rows involved in a computation. This value is superseded by an admin-defined `minimum_dataset_size` parameter, if the admin's value is larger.

Returns The result of the analysis. Multiple return types are possible.

Return type Union[_Model, *AsyncAnalysis*, RichResult[_Model, _ModelMetadata]]

maximum_privacy_exposure(*minimum_dataset_size=None*)

Maximum privacy exposure associated with running this analysis.

Estimate the maximum incremental privacy exposure that could result from running this computation for the current user. The result is represented as a percentage of privacy exposure limit for each data source.

Note: Estimating maximum privacy exposure may incur a small amount of privacy exposure.

Parameters **minimum_dataset_size** (`Optional[int]`) – When `minimum_dataset_size` is set, prevent computations on data sets that have fewer rows than the specified value.

Returns A dictionary that maps a `TableIdentifier` to the estimated maximum fractional privacy exposure that running the analysis would incur for the associated table.

Return type *FractionalPrivacyExposure*

class leapyear.analytics.classes.**AsyncAnalysis**(*async_job_id*, *, *analysis*, *rich_result*)

Asynchronous job for running analysis queries.

check_status()

Check the status of the given asynchronous job.

Return type *AsyncJobStatus*

cancel()

Cancel the job.

wait_to_cancel(***kwargs*)

Wait for the given asynchronous job to finish.

Same as 'wait', but doesn't error on cancellations. Takes the same arguments as 'wait'.

Return type `None`

process_result(*result*)

Process the result of running an analysis.

Return type `Union`[~_Model, `ForwardRef`]

serialize()

Serialize an external analysis to a string.

Return type `str`

Analysis Subclasses

This section describes the subclasses of *Analysis*, generally determined by what type of output they produce.

class leapyear.analytics.classes.**BoundsAnalysis**(*analysis*, *relation*)
Analysis that results in a lower and upper bound.

class leapyear.analytics.classes.**ClusteringAnalysis**(*analysis*, *relation*)
Analysis that results in a clustering model.

class leapyear.analytics.classes.**ConfusionModelAnalysis**(*analysis*, *relation*)
Analysis that results in a ConfusionCurve object.

class leapyear.analytics.classes.**CountAnalysis**(*analysis*, *relation*)
Analysis that results in a scalar count value.

class leapyear.analytics.classes.**CountAnalysisWithRI**(*analysis*, *relation*)
Analysis that computes a scalar count value.

The user can request additional information about the computation with *run(rich_result=True)*. A *RandomizationInterval* object will be generated.

class leapyear.analytics.classes.**CrossValidationAnalysis**(*analysis*, *relation*)
Analysis that results in multiple results of the same type.

class leapyear.analytics.classes.**DescribeAnalysis**(*analysis*, *relation*)
Analysis that produces a model describing a dataset.

class leapyear.analytics.classes.**FailAnalysis**(*analysis*, *relation*)
An analysis that always fails.

class leapyear.analytics.classes.**ForestModelClassifierAnalysis**(*analysis*, *relation*)
Analysis that results in a forest model.

class leapyear.analytics.classes.**ForestModelRegressionAnalysis**(*analysis*, *relation*)
Analysis that results in a forest model.

class leapyear.analytics.classes.**GradientBoostedTreeClassifierModelAnalysis**(*analysis*, *relation*)
Gradient boosted tree classifier analysis.

When executed, this analysis returns a model object of class *GradientBoostedTreeClassifier*.

class leapyear.analytics.classes.**GroupbyAggAnalysis**(*analysis*, *relation*)
Analysis that results in a *GroupbyAgg*.

class leapyear.analytics.classes.**GenLinAnalysis**(*analysis*, *relation*)
Analysis that results in a generalized linear model.

class leapyear.analytics.classes.**Histogram2DAnalysis**(*analysis*, *relation*)
Analysis that results in a 2d histogram.

class leapyear.analytics.classes.**HistogramAnalysis**(*analysis*, *relation*)
Analysis that results in a histogram.

class leapyear.analytics.classes.**HyperOptAnalysis**(*analysis*, *relation*)
Analysis that returns the result of hyperparameter optimization.

class leapyear.analytics.classes.**MatrixAnalysis**(*analysis*, *relation*)
Analysis that results in a matrix of float values.

class leapyear.analytics.classes.**ScalarAnalysis**(*analysis*, *relation*)
Analysis that results in a scalar float value.

class leapyear.analytics.classes.**ScalarAnalysisWithRI**(*analysis*, *relation*)
Analysis that computes a scalar value.

The user can request additional information about the computation with *run(rich_result=True)*. In this case, a *RandomizationInterval* object will be generated.

This likely interval is likely to include the exact value of the computation on the data sample.

class leapyear.analytics.classes.**ScalarFromHistogramAnalysis**(*f*, **args*)
Analysis that uses a histogram to compute a scalar value.

class leapyear.analytics.classes.**SleepAnalysis**(*analysis*, *relation*)
An analysis that will sleep for a set amount of microseconds.

class leapyear.analytics.classes.**TypeAnalysis**(*analysis*, *relation*)
Analysis that results list of counts associated with types.

Rich Results

This section documents classes related to rich results and privacy exposure measurements.

class leapyear.analytics.classes.**RandomizationInterval**(*estimation_method: str*, *confidence_level: float*, *low: float*, *high: float*)
An interval estimating the uncertainty in the exact answer, given randomized output.

A *RandomizationInterval* can be generated for a subset of the analyses offered by LeapYear by running an analysis with *run(rich_result = True)*.

The interval between `low` and `high` is expected to include the exact value of the computation on the data sample with the stated `confidence_level` (e.g. 95%).

Parameters

- **confidence_level** (*float*) – The confidence that the exact answer lies within the interval.
- **low** (*float*) – The lower bound of the interval.
- **high** (*float*) – The upper bound of the interval.
- **estimation_method** (*str*) – The method used to compute the interval depends on analysis type.

  With `'bayesian'` method, the randomization interval is obtained using a posterior distribution analysis based on non-informative prior, the knowledge of randomized output and the scale of the randomization effect applied.

  With `'approximate'` method, the randomization interval is estimated using simplified simulation process.

  Note: The `'approximate'` estimation method tends to produce biased intervals for small data samples.

  ---

class leapyear.analytics.classes.**FractionalPrivacyExposure**
A fractional measure of privacy exposure for a collection of tables.

This is represented by a dictionary, mapping a `TableIdentifier` to the fraction of total privacy exposure that has been expended for the table corresponding to the `TableIdentifier`.

Aggregate Results

This section documents classes related to aggregate resuls from group by operations.

class leapyear.analytics.classes.**GroupbyAgg**(*aggregate_type: Sequence[_ml.GroupbyAgg]*, *key_columns: Sequence[Attribute]*, *aggs: Mapping[Tuple[Any, ...], float]*)

A GroupBy Aggregate.

property aggregate_type
Alias for field number 0

property key_columns
Alias for field number 1

property aggs
Alias for field number 2

to_dataframe(*groups_as_index=True*)
Convert to a pandas DataFrame.

Parameters **groups_as_index** (*bool, optional*) – Whether the groupBy columns should be the `MultiIndex` of the resulting `DataFrame`. If `False`, then the groupBy columns are made into columns of the output. By default True.

Returns A `pandas DataFrame`, with a multi-index corresponding to the key columns of the groupBy operation if `groups_as_index` = True.

Return type pd.DataFrame

1.4.10 Module leapyear.model

LeapYear models.

Data objects generated from training or evaluating models used in machine learning.

Regression-Based Models

class leapyear.model.**GLM**(*affinity: bool*, *l1reg: float*, *l2reg: float*, *model: GeneralizedLinearModel*)
A representation of a trained Generalized Linear Model (GLM).

Differentially private versions of GLMs are calibrated using various methods, e.g.

- *leapyear.analytics.logreg()*,
- leapyear.analytics.linreg(),
- The variants of these methods that optimize model hyperparameters.

Objects of this class store parameters and structure of a regression model and can be used to generate predictions for regression and classification problems.

property affinity
Alias for field number 0

property l1reg
Alias for field number 1

property l2reg
Alias for field number 2

property model
Alias for field number 3

property coefficients
Model coefficients, excluding intercepts.

Return type ndarray

property intercept
Model intercept, if model has only one coefficient set.

Return type float

property intercepts
Model intercepts, if any.

Return type ndarray

property model_type
Model type (e.g. linear, logistic).

Return type GeneralizedLinearModelType

decision_function(*xs*)
Decision function of the generalized linear model.

Computes the height of the regression function (xbeta) at the provided points. This is purely linear transformation of the input features.

In case of logistic model, model would ultimately classify observations based on the sign of this decision function.

Parameters **xs** (ndarray) – a set of datapoints for which to predict

Returns The predicted decision function

Return type np.ndarray

predict(*xs*)
Prediction function of the generalized linear model.

For linear problems, returns the height of the regression line (decision function) at the data points provided.

For classification problems, returns boolean classification choice, which is based on the sign of this decision function.

Parameters **xs** (ndarray) – a set of datapoints for which to predict

Returns the predictions for the points according to the model

Return type np.ndarray

predict_proba(*xs*)
Probabilities given by generalized linear model.

For logistic classification problems, returns probability that the model assigns to a positive response (True outcome variable) for each of the data points provided.

Parameters **xs** (ndarray) – array with input data

Returns array of probability scores assigned by the model

Return type np.ndarray

predict_log_proba(*xs*)

Logarithm of probabilities given by generalized linear model.

For logistic classification problems, returns natural logarithm of probability that the model assigns to a True outcome for each of the data points provided.

Parameters **xs** (ndarray) – array with input data

Returns array of log-probability scores assigned by the model

Return type np.ndarray

to_dict()

Convert to a dictionary.

Return type Dict[str, Any]

classmethod from_dict(*d*)

Convert from a dictionary.

Return type ForwardRef

to_shap()

Convert the trained model to SHAP format.

The converted model can then be used to construct a *LinearExplainer* object able to generate Shapley explanations for new records to which the model would be applied to.

Note that:

- model execution and generation of model score explanations is expected to be done in a production setting by an automated system with direct access to record-level information.
- feature explanations for categorical features are currently not supported. Consider one-hot encoding features to get the benefits of explainable model scores.

Examples

```
>>> import shap
>>> from leapyear import analytics as la
>>> ...
>>> glm_model = la.logreg(xs, y, ds).run()
>>> glm_explainer = shap.LinearExplainer(glm_model.to_shap(), X_reference)
>>> glm_shap_values = glm_explainer.shap_values(X_to_predict)
>>> ...
```

In this example:

- The input *X_reference* used to initialize the explainer object is a *pandas.DataFrame* containing explanatory variables in the same order as used to train models. It is used to infer what model scores and feature distribution should be considered "typical".
- The input *X_to_predict* is a *pandas.DataFrame* capturing the explanatory variables in the same order as used to train models.

See https://shap.readthedocs.io/en/latest/generated/shap.explainers.Linear.html

LeapYear has been tested with SHAP version 0.39.0. Older or newer versions are not guaranteed to work.

Return type _ShapGLM

Tree-Based Models

class leapyear.model.**RandomForestClassifier**(*ntrees: int*, *height: int*, *model: DecisionForest*)

A representation of a trained Random Forest classification model.

Provides methods for making predictions and report on feature importance statistics.

property ntrees
Alias for field number 0

property height
Alias for field number 1

property model
Alias for field number 2

predict(*xs*)
Prediction function of the random forest classification model.

For classification problems, returns the most likely class according to the model.

Parameters **xs** (ndarray) – array with input data

Returns array of most likely outcome labels assigned by the model

Return type np.ndarray

predict_proba(*xs*)
Prediction probability function of the random forest model.

For each of the data points provided, returns probability that the model assigns to any given outcome.

Parameters **xs** (ndarray) – array with input data

Returns array of probability scores assigned by the model to input data points and possible outcomes

Return type np.ndarray

predict_log_proba(*xs*)
Logarithm of probabilities given by random forest model.

For each of the data points provided, returns natural logarithm of probability that the model assigns to any given outcome.

Parameters **xs** (ndarray) – array with input data

Returns array of log-probability scores assigned by the model to input data points and possible outcomes

Return type np.ndarray

property feature_importance
Relative feature importance.

Feature importances are derived based on the information collected during model training with differentially private computations, specifically:

1. For each tree and for each split of the tree, lookup value (gain) of introducing the split, as calculated on training data during model calibration - and attribute it to the splitting feature. See *leapyear.analytics.random_forest()* for specific calculation of split gain based on a notion of Gini impurity.

2. To compute tree-specific feature importances, sum up split gains across all splits within each tree, weighted (multiplied) by parent node size, and re-scale these tree-specific feature importances to sum up to 1 for each tree.
3. Average feature importances across all trees in the random forest ensemble to get final feature importance.

References:

- Hastie, Tibshirani, Friedman. "The Elements of Statistical Learning, 2nd Edition." 2001.

Return type `Mapping[int, float]`

to_dict()
Convert to a dictionary.

Return type `Dict[str, Any]`

classmethod from_dict(*d*)
Convert from a dictionary.

Return type `ForwardRef`

to_shap()
Convert the trained model to SHAP format.

The converted model can then be used to construct a *TreeExplainer* object able to generate Shapley explanations for new records to which the model would be applied to.

Note that:

- model execution and generation of model score explanations is expected to be done in a production setting by an automated system with direct access to record-level information.
- feature explanations for categorical features are currently not supported. Consider one-hot encoding features to get the benefits of explainable model scores.

Examples

```
>>> import shap
>>> from leapyear import analytics as la
>>> ...
>>> rfc_model = la.random_forest (xs, y, ds).run()
>>> rfc_explainer = shap.TreeExplainer(rfc_model.to_shap(), X_reference)
>>> rfc_shap_values = rfc_explainer.shap_values(X_to_predict)
>>> ...
```

In this example:

- The input *X_reference* used to initialize the explainer object is a *pandas.DataFrame* containing explanatory variables in the same order as used to train models. It is used to infer what model scores and feature distribution should be considered "typical".
- The input *X_to_predict* is a *pandas.DataFrame* capturing the explanatory variables in the same order as used to train models.

See https://shap.readthedocs.io/en/latest/generated/shap.explainers.Tree.html

LeapYear has been tested with SHAP version 0.39.0. Older or newer versions are not guaranteed to work.

Return type `Dict`

class leapyear.model.**RandomForestRegressor**(*ntrees: int*, *height: int*, *model: DecisionForest*)

A representation of a trained Random Forest regression model.

property ntrees
Alias for field number 0

property height
Alias for field number 1

property model
Alias for field number 2

predict(*xs*)
Prediction function of the random forest regression model.

For each of the data points provided, returns the prediction that the model assigns.

Parameters **xs** (ndarray) – array with input data

Returns array of predictions assigned by the model to input data points

Return type np.ndarray

to_dict()
Convert to a dictionary.

Return type Dict[str, Any]

classmethod from_dict(*d*)
Convert from a dictionary.

Return type ForwardRef

to_shap()
Convert the trained model to SHAP format.

The converted model can then be used to construct a *TreeExplainer* object able to generate Shapley explanations for new records to which the model would be applied to.

Note that:

- model execution and generation of model score explanations is expected to be done in a production setting by an automated system with direct access to record-level information.
- feature explanations for categorical features are currently not supported. Consider one-hot encoding features to get the benefits of explainable model scores.

Examples

```
>>> import shap
>>> from leapyear import analytics as la
>>> ...
>>> rf_model = la.regression_trees(xs, y, ds).run()
>>> rf_explainer = shap.TreeExplainer(rf_model.to_shap(), X_reference)
>>> rf_shap_values = rf_explainer.shap_values(X_to_predict)
>>> ...
```

In this example:

- The input *X_reference* used to initialize the explainer object is a *pandas.DataFrame* containing explanatory variables in the same order as used to train models. It is used to infer what model scores and feature distribution should be considered "typical".

- The input *X_to_predict* is a *pandas.DataFrame* capturing the explanatory variables in the same order as used to train models.

See https://shap.readthedocs.io/en/latest/generated/shap.explainers.Tree.html

LeapYear has been tested with SHAP version 0.39.0. Older or newer versions are not guaranteed to work.

Return type Dict

class leapyear.model.**GradientBoostedTreeClassifier**(*max_depth: int*, *model: WeightedDecisionForest*)

A representation of a trained gradient boosted tree classifier model.

This includes two named fields:

- `max_depth` - the maximum depth of the individual decision trees.
- `model` - a model object of class `WeightedDecisionForest`, including information about individual decision trees and their weights.

property max_depth
Alias for field number 0

property model
Alias for field number 1

predict(*xs*)
Prediction function of the gradient boosted tree classification model.

For classification problems, returns the most likely class according to the model.

Parameters **xs** (ndarray) – array with input data

Returns array of most likely outcome labels assigned by the model

Return type np.ndarray

predict_proba(*xs*)
Prediction probability function of the GBT model.

For each of the data points provided, returns probability that the model assigns to any given outcome.

Parameters **xs** (ndarray) – array with input data

Returns array of probability scores assigned by the model to input data points and possible outcomes

Return type np.ndarray

predict_log_proba(*xs*)
Logarithm of probabilities given by GBT model.

For each of the data points provided, returns natural logarithm of probability that the model assigns to any given outcome.

Parameters **xs** (ndarray) – array with input data

Returns array of log-probability scores assigned by the model to input data points and possible outcomes

Return type np.ndarray

to_dict()
Convert to a dictionary.

Return type Dict[str, Any]

classmethod from_dict(*d*)
Convert from a dictionary.

Return type ForwardRef

to_shap()
Convert the trained model to SHAP format.

The converted model can then be used to construct a *TreeExplainer* object able to generate Shapley explanations for new records to which the model would be applied to.

Note that:

- model execution and generation of model score explanations is expected to be done in a production setting by an automated system with direct access to record-level information.
- feature explanations for categorical features are currently not supported. Consider one-hot encoding features to get the benefits of explainable model scores.

Examples

```
>>> import shap
>>> from leapyear import analytics as la
>>> ...
>>> gbt_model = la.gradient_boosted_tree_classifier(xs, y, ds).run()
>>> gbt_explainer = shap.TreeExplainer(gbt_model.to_shap(), X_reference)
>>> gbt_shap_values = gbt_explainer.shap_values(X_to_predict)
>>> ...
```

In this example:

- The input *X_reference* used to initialize the explainer object is a *pandas.DataFrame* containing explanatory variables in the same order as used to train models. It is used to infer what model scores and feature distribution should be considered "typical".
- The input *X_to_predict* is a *pandas.DataFrame* capturing the explanatory variables in the same order as used to train models.

See https://shap.readthedocs.io/en/latest/generated/shap.explainers.Tree.html

LeapYear has been tested with SHAP version 0.39.0. Older or newer versions are not guaranteed to work.

Return type Dict

Clustering Models

class leapyear.model.**ClusterModel**(*niters: int*, *nclusters: int*, *model: _CM*)
A representation of the trained K-means clustering model.

This model is generated by running a K-means clustering algorithm *leapyear.analytics.kmeans()* and contains cluster centroids (centers).

property niters
Alias for field number 0

property nclusters
Alias for field number 1

property model
Alias for field number 2

property centroids
Model centroids.

Return type ndarray[Any, dtype[float64]]

predict(*xs*)
Prediction function of the clustering model.

Returns the labels for each point in xs.

Parameters **xs** (ndarray) – A 2-dimensional array of data points.

Returns The associated cluster labels predicted by the the model.

Return type np.ndarray

to_dict()
Convert to a dictionary.

Return type Dict[str, Any]

classmethod from_dict(*d*)
Convert from a dictionary.

Return type ForwardRef

Model Evaluation Objects

class leapyear.model.**ConfusionCurve**(*model: _CC*)
The Confusion curve object.

This model is generated from running *leapyear.analytics.roc()* and contains the metrics of true positive, false positive, true negative and false negative rates for a sequence of thresholds. Other common metrics are provided as properties of this model.

property model
Alias for field number 0

property df
Return a dataframe containing most of the analytics.

Return type DataFrame

property thresholds
Thresholds.

Outputs the list of thresholds used for generating confusion curve.

Return type ndarray

property tpr
Compute true positive rates.

Outputs a list of true positive rate (sensitivity, recall) values, associated with chosen thresholds.

Aliases: tpr (true positive rate), sensitivity, recall

See also:

***Sensitivity and specificity*:** https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Return type ndarray

property sensitivity
Compute true positive rates.

Outputs a list of true positive rate (sensitivity, recall) values, associated with chosen thresholds.

Aliases: `tpr` (true positive rate), `sensitivity`, `recall`

See also:

***Sensitivity and specificity*:** https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Return type `ndarray`

property recall
Compute true positive rates.

Outputs a list of true positive rate (sensitivity, recall) values, associated with chosen thresholds.

Aliases: `tpr` (true positive rate), `sensitivity`, `recall`

See also:

***Sensitivity and specificity*:** https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Return type `ndarray`

property fpr
Compute false positive rates.

Outputs a list of false positive rate (fallout) values, associated with chosen thresholds.

Aliases: `fpr` (false positive rate), `fallout`

See also:

***False positive rate*:** https://en.wikipedia.org/wiki/False_positive_rate

Return type `ndarray`

property fallout
Compute false positive rates.

Outputs a list of false positive rate (fallout) values, associated with chosen thresholds.

Aliases: `fpr` (false positive rate), `fallout`

See also:

***False positive rate*:** https://en.wikipedia.org/wiki/False_positive_rate

Return type `ndarray`

property tnr
Compute true negative rates.

Outputs a list of true negative rate (specificity) values, associated with chosen thresholds.

Aliases: `tnr` (true negative rate), `specificity`

See also:

***Sensitivity and specificity*:** https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Return type ndarray

property specificity

Compute true negative rates.

Outputs a list of true negative rate (specificity) values, associated with chosen thresholds.

Aliases: tnr (true negative rate), specificity

See also:

***Sensitivity and specificity*:** https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Return type ndarray

property fnr

Compute false negative rates.

Outputs a list of false negative rate (miss rate) values, associated with chosen thresholds.

Aliases: fnr (false negative rate), missrate

See also:

***False negative rates*:** https://en.wikipedia.org/wiki/Type_I_and_type_II_errors

Return type ndarray

property missrate

Compute false negative rates.

Outputs a list of false negative rate (miss rate) values, associated with chosen thresholds.

Aliases: fnr (false negative rate), missrate

See also:

***False negative rates*:** https://en.wikipedia.org/wiki/Type_I_and_type_II_errors

Return type ndarray

property precision

Compute precision.

Aliases: precision, ppv (positive predictive value)

See also:

***Precision and recall*:** https://en.wikipedia.org/wiki/Precision_and_recall

Return type ndarray

property ppv

Compute precision.

Aliases: precision, ppv (positive predictive value)

See also:

Precision and recall: https://en.wikipedia.org/wiki/Precision_and_recall

Return type ndarray

property npv
Negative predictive value.

Return type ndarray

property accuracy
Accuracy.

Return type ndarray

property f1score
F1-score.

Return type ndarray

property mcc
Matthews correlation coefficient.

Return type ndarray

property auc_roc
Area under the ROC curve.

Calculates the area under Receiver Operating Characteristic (ROC) curve.

Return type ndarray

property auc_pr
Area under the Precision-Recall curve.

Return type ndarray

property gmeasure
the geometric mean of the precision and recall.

Type G-measure

Return type ndarray

fscore(*beta*)
Fbeta-score.

The Fbeta-score is the weighted harmonic mean between the precision and recall.

Parameters **beta** (float) – Non-negative float for the relative proportion of precision and recall.

Returns

Return type The Fbeta score

1.4.11 Module leapyear.exceptions

All errors and exceptions.

exception leapyear.exceptions.**ClientError**
General client errors.

static from_response(*response*, *errors=None*)
Create a ClientError from a response.

Return type *ClientError*

class leapyear.exceptions.**ErrorInfo**(*message: str*, *details: Optional[str] = None*)
Information about a particular error.

property message
Alias for field number 0

property details
Alias for field number 1

exception leapyear.exceptions.**APIError**(*message*, *, *response=None*, *errors=None*)
Error from the API Server.

exception leapyear.exceptions.**DataSetTooSmallException**(***kwargs*)
Re-throw of DataSetTooSmall exception.

This exception is triggered and interrupts the computation when the LeapYear system is unable to guarantee a result that meets the target accuracy while staying within the maximum Privacy Exposure allowed for the computation.

The primary cause for this exception is that the data sample being analyzed is too small, and therefore providing an accurate result would reveal too much information about a single record.

See also:

Built-in safeguards and methods to override exceptions

exception leapyear.exceptions.**DataSetSizeBlockedByPrivacyProfileException**(***kwargs*)
Re-throw of DataSetSizeBlockedByProfile exception.

This exception is triggered and interrupts the computation when the LeapYear system detects that the data set size is smaller than the minimum size required by the privacy profile.

The primary cause for this exception is that the data sample being analyzed is too small, and therefore providing an accurate result would reveal too much information about a single record.

See also:

Built-in safeguards and methods to override exceptions

exception leapyear.exceptions.**HighPrivacyExposureException**(***kwargs*)
Re-throw of HighPrivacyExposure exception.

This exception occurs when it is estimated that the Privacy Exposure will reach or exceed the Privacy Exposure Limit for that particular computation. This could be due to either:

- the data sample is too small and LeapYear estimates that a large privacy exposure will be incurred, or
- the computation involves a relation that generates a large sensitivity multiplier.

The precise reason for the exception is not made explicit because of potential privacy concerns.

See also:

Built-in safeguards and methods to override exceptions

Limiting the effect of data derivation steps on computation sensitivity

exception leapyear.exceptions.**SensitivityMultiplierTooLargeException**(*logbase10_sensitivity_multiplier*, ***kwargs*)

Re-throw of SensitivityMultiplierTooLarge exception.

This exception is triggered and interrupts a computation when the LeapYear system estimates that the computation:

- will reach or exceed the maximum Privacy Exposure allowed for the computation; AND
- is based on a derived dataset that generates an excessively large sensitivity multiplier.

See also:

Built-in safeguards and methods to override exceptions

Limiting the effect of data derivation steps on computation sensitivity

exception leapyear.exceptions.**InvalidURL**
URL can not be parsed.

exception leapyear.exceptions.**TLSError**
Errors from TLS.

exception leapyear.exceptions.**InvalidJson**
Json was not constructed correctly.

exception leapyear.exceptions.**ConnectionError**
Error establishing connection.

exception leapyear.exceptions.**TokenExpiredError**
Authentication token was expired.

exception leapyear.exceptions.**AsyncTimeoutError**
Async job timed out.

exception leapyear.exceptions.**AsyncCancelledError**
Async job was cancelled.

exception leapyear.exceptions.**GroupbyAggTooManyKeysError**
Too many keys for a GroupbyAgg operation.

exception leapyear.exceptions.**LoadUnsupportedModelException**
This exception is raised when trying to load or save an unsupported type of model.

exception leapyear.exceptions.**LoadModelMismatchException**(*actual_type*, *expected_type*)

This exception is raised when there is a mismatch with loaded and expected models.

exception leapyear.exceptions.**LoadModelVersionException**(*model*, *name_file*)
This exception is raised when loading a module with a version not supported.

exception leapyear.exceptions.**SaveUnsupportedModelException**
This exception is raised when trying to save an unsupported model.

exception leapyear.exceptions.**PublicKeyCredentialsError**(*message*)
Exceptions originating from problems with public key auth credentials.

1.4.12 Module leapyear.ext

Quick client

leapyear.ext.user.**client**(*config_file=PosixPath('~/.leapyear_client.ini')*, *debug=False*, ***kwargs*)

Use environment variables or a configuration file to quickly connect to LeapYear.

This function uses values found in environment variables, a config file, keyword arguments and built-in defaults to try to establish a connection to a LeapYear server. In order of precedence, values are taken from the `kwargs` of this function, then environment variable, the config file, and finally, default values, if they exist.

By default, the config file is `~/.leapyear_client.ini`. The config file requires a `[leapyear.io]` section where values can be found.

The following table contains the names of the values when specified by particular methods, and the default values if no other value can be determined.

environment variable	ini key/keyword argument	default value
LY_URL	`url`	`'http://localhost:4401'`
LY_USERNAME	`username`	`None`
LY_PASSWORD	`password`	`None`
LY_DEFAULT_ANALYSIS_CACHING	`default_analysis_caching`	`True`
LY_DEFAULT_ALLOW_MAX_BUDGET_ALLOCATION	`default_allow_max_budget_allocation`	`True`
LY_LOGGING_LEVEL	`logging_level`	`'NOTSET'`

At least `username` and `password` must be supplied to establish a connection to the LeapYear server.

`logging_level` should be the name of a logging level in `logging`.

Example

Contents of `~/.leapyear_client.ini` are

```
[leapyear.io]
username = alice
password = lihjAgsd324$
url = http://api.leapyear.domain.com:4401
```

Next, we execute a basic test with `debug=True` to see the values that are passed to the *`Client`* constructor.

```
>>> from leapyear.ext.user import client
>>> import logging
>>> logging.basicConfig()
>>> c = client(debug=True)
DEBUG:leapyear.ext.user:Found config file with [leapyear.io] section.
DEBUG:leapyear.ext.user:Resolved the following values:
DEBUG:leapyear.ext.user:  url <- <str: 'http://api.leapyear.domain.com:4401'>
DEBUG:leapyear.ext.user:  username <- <str: 'alice'>
DEBUG:leapyear.ext.user:  password <- <str: 'lihjAgsd324$'>
DEBUG:leapyear.ext.user:  default_analysis_caching <- <bool: True>
DEBUG:leapyear.ext.user:  default_allow_max_budget_allocationt <- <bool: True>
DEBUG:leapyear.ext.user:  logging_level <- <int: 0>
>>> print(c.connected)
True
>>> c.close()
```

Parameters

- **config_file** (*pathlib.Path*) – Specify an alternate config file.
- **debug** (*bool*) – Set to `True` to enable extra debugging information. `basicConfig()` may be useful to run so that debugging information will print to stdout.

Return type *Client*

CHAPTER

TWO

REFERENCE

- genindex

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

D

E

F

G

H

M

N

S

T

U

V

W

Y