doubt

Release 4.0.0

Dan Saattrup Nielsen

Jan 21, 2022

CONTENTS:

CHAPTER

ONE

DOUBT

1.1 doubt package

1.1.1 Subpackages

doubt.datasets package

Submodules

doubt.datasets.airfoil module

Airfoil data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.airfoil.**Airfoil**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

The NASA data set comprises different size NACA 0012 airfoils at various wind tunnel speeds and angles of attack. The span of the airfoil and the observer position were the same in all of the experiments.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

int: Frequency, in Hertzs

float: Angle of attack, in degrees

float: Chord length, in meters

float: Free-stream velocity, in meters per second

float: Suction side displacement thickness, in meters

Targets:

float: Scaled sound pressure level, in decibels

Source: https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

Examples

Load in the data set:

```
>>> dataset = Airfoil()
>>> dataset.shape
(1503, 6)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((1503, 5), (1503,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((1181, 5), (1181,), (322, 5), (322,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.bike_sharing_daily module

Daily bike sharing data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.bike_sharing_daily.**BikeSharingDaily**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Bike sharing systems are new generation of traditional bike rentals where whole process from membership, rental and return back has become automatic. Through these systems, user is able to easily rent a bike from a particular position and return back at another position. Currently, there are about over 500 bike-sharing programs around the world which is composed of over 500 thousands bicycles. Today, there exists great interest in these systems due to their important role in traffic, environmental and health issues.

Apart from interesting real world applications of bike sharing systems, the characteristics of data being generated by these systems make them attractive for the research. Opposed to other transport services such as bus or subway,

the duration of travel, departure and arrival position is explicitly recorded in these systems. This feature turns bike sharing system into a virtual sensor network that can be used for sensing mobility in the city. Hence, it is expected that most of important events in the city could be detected via monitoring these data.

Parameters `cache` (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

`cache`
The name of the cache.

Type str or None

`shape`
Dimensions of the data set

Type tuple of integers

`columns`
List of column names in the data set

Type list of strings

Features:

instant (int): Record index

season (int): The season, with 1 = winter, 2 = spring, 3 = summer and 4 = autumn

yr (int): The year, with 0 = 2011 and 1 = 2012

mnth (int): The month, from 1 to 12 inclusive

holiday (int): Whether day is a holiday or not, binary valued

weekday (int): The day of the week, from 0 to 6 inclusive

workingday (int): Working day, 1 if day is neither weekend nor holiday, otherwise 0

weathersit (int): Weather, encoded as

1. Clear, few clouds, partly cloudy
2. Mist and cloudy, mist and broken clouds, mist and few clouds
3. Light snow, light rain and thunderstorm and scattered clouds, light rain and scattered clouds
4. Heavy rain and ice pallets and thunderstorm and mist, or snow and fog

temp (float): Max-min normalised temperature in Celsius, from -8 to +39

atemp (float): Max-min normalised feeling temperature in Celsius, from -16 to +50

hum (float): Scaled max-min normalised humidity, from 0 to 1

windspeed (float): Scaled max-min normalised wind speed, from 0 to 1

Targets:

casual (int): Count of casual users

registered (int): Count of registered users

cnt (int): Sum of casual and registered users

Source: https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset

Examples

Load in the data set:

```
>>> dataset = BikeSharingDaily()
>>> dataset.shape
(731, 15)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((731, 12), (731, 3))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((574, 12), (574, 3), (157, 12), (157, 3))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.bike_sharing_hourly module

Hourly bike sharing data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.bike_sharing_hourly.**BikeSharingHourly**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Bike sharing systems are new generation of traditional bike rentals where whole process from membership, rental and return back has become automatic. Through these systems, user is able to easily rent a bike from a particular position and return back at another position. Currently, there are about over 500 bike-sharing programs around the world which is composed of over 500 thousands bicycles. Today, there exists great interest in these systems due to their important role in traffic, environmental and health issues.

Apart from interesting real world applications of bike sharing systems, the characteristics of data being generated by these systems make them attractive for the research. Opposed to other transport services such as bus or subway, the duration of travel, departure and arrival position is explicitly recorded in these systems. This feature turns bike sharing system into a virtual sensor network that can be used for sensing mobility in the city. Hence, it is expected that most of important events in the city could be detected via monitoring these data.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache
The name of the cache.

Type str or None

shape
Dimensions of the data set

Type tuple of integers

columns
List of column names in the data set

Type list of strings

Features:

instant (int): Record index

season (int): The season, with 1 = winter, 2 = spring, 3 = summer and 4 = autumn

yr (int): The year, with 0 = 2011 and 1 = 2012

mnth (int): The month, from 1 to 12 inclusive

hr (int): The hour of the day, from 0 to 23 inclusive

holiday (int): Whether day is a holiday or not, binary valued

weekday (int): The day of the week, from 0 to 6 inclusive

workingday (int): Working day, 1 if day is neither weekend nor holiday, otherwise 0

weathersit (int): Weather, encoded as

1. Clear, few clouds, partly cloudy
2. Mist and cloudy, mist and broken clouds, mist and few clouds
3. Light snow, light rain and thunderstorm and scattered clouds, light rain and scattered clouds
4. Heavy rain and ice pallets and thunderstorm and mist, or snow and fog

temp (float): Max-min normalised temperature in Celsius, from -8 to +39

atemp (float): Max-min normalised feeling temperature in Celsius, from -16 to +50

hum (float): Scaled max-min normalised humidity, from 0 to 1

windspeed (float): Scaled max-min normalised wind speed, from 0 to 1

Targets:

casual (int): Count of casual users

registered (int): Count of registered users

cnt (int): Sum of casual and registered users

Source: https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset

Examples

Load in the data set:

```
>>> dataset = BikeSharingHourly()
>>> dataset.shape
(17379, 16)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((17379, 13), (17379, 3))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((13873, 13), (13873, 3), (3506, 13), (3506, 3))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.blog module

Blog post data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.blog.**Blog**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

This data originates from blog posts. The raw HTML-documents of the blog posts were crawled and processed. The prediction task associated with the data is the prediction of the number of comments in the upcoming 24 hours. In order to simulate this situation, we choose a basetime (in the past) and select the blog posts that were published at most 72 hours before the selected base date/time. Then, we calculate all the features of the selected blog posts from the information that was available at the basetime, therefore each instance corresponds to a blog post. The target is the number of comments that the blog post received in the next 24 hours relative to the basetime.

In the train data, the basetimes were in the years 2010 and 2011. In the test data the basetimes were in February and March 2012. This simulates the real-world situtation in which training data from the past is available to predict events in the future.

The train data was generated from different basetimes that may temporally overlap. Therefore, if you simply split the train into disjoint partitions, the underlying time intervals may overlap. Therefore, the you should use the provided, temporally disjoint train and test splits in order to ensure that the evaluation is fair.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

`cache`
The name of the cache.

Type str or None

`shape`
Dimensions of the data set

Type tuple of integers

`columns`
List of column names in the data set

Type list of strings

Features:

Features 0-49 (float): 50 features containing the average, standard deviation, minimum, maximum and median of feature 50-59 for the source of the current blog post, by which we mean the blog on which the post appeared. For example, myblog.blog.org would be the source of the post myblog.blog.org/post_2010_09_10

Feature 50 (int): Total number of comments before basetime

Feature 51 (int): Number of comments in the last 24 hours before the basetime

Feature 52 (int): If T1 is the datetime 48 hours before basetime and T2 is the datetime 24 hours before basetime, then this is the number of comments in the time period between T1 and T2

Feature 53 (int): Number of comments in the first 24 hours after the publication of the blog post, but before basetime

Feature 54 (int): The difference between Feature 51 and Feature 52

Features 55-59 (int): The same thing as Features 50-51, but for links (trackbacks) instead of comments

Feature 60 (float): The length of time between the publication of the blog post and basetime

Feature 61 (int): The length of the blog post

Features 62-261 (int): The 200 bag of words features for 200 frequent words of the text of the blog post

Features 262-268 (int): Binary indicators for the weekday (Monday-Sunday) of the basetime

Features 269-275 (int): Binary indicators for the weekday (Monday-Sunday) of the date of publication of the blog post

Feature 276 (int): Number of parent pages: we consider a blog post P as a parent of blog post B if B is a reply (trackback) to P

Features 277-279 (float): Minimum, maximum and average of the number of comments the parents received

Targets:

int: The number of comments in the next 24 hours (relative to baseline)

Source: https://archive.ics.uci.edu/ml/datasets/BlogFeedback

Examples

Load in the data set:

```
>>> dataset = Blog()
>>> dataset.shape
(52397, 281)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((52397, 279), (52397,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((41949, 279), (41949,), (10448, 279), (10448,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.concrete module

Concrete data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.concrete.**Concrete**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Concrete is the most important material in civil engineering. The concrete compressive strength is a highly nonlinear function of age and ingredients.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

Cement (float): Kg of cement in an m3 mixture

Blast Furnace Slag (float): Kg of blast furnace slag in an m3 mixture

Fly Ash (float): Kg of fly ash in an m3 mixture

Water (float): Kg of water in an m3 mixture

Superplasticiser (float): Kg of superplasticiser in an m3 mixture

Coarse Aggregate (float): Kg of coarse aggregate in an m3 mixture

Fine Aggregate (float): Kg of fine aggregate in an m3 mixture

Age (int): Age in days, between 1 and 365 inclusive

Targets:

Concrete Compressive Strength (float): Concrete compressive strength in megapascals

Source: https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength

Examples

Load in the data set:

```
>>> dataset = Concrete()
>>> dataset.shape
(1030, 9)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((1030, 8), (1030,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((807, 8), (807,), (223, 8), (223,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.cpu module

CPU data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.cpu.**CPU**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Relative CPU Performance Data, described in terms of its cycle time, memory size, etc.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

vendor_name (string): Name of the vendor, 30 unique values

model_name (string): Name of the model

myct (int): Machine cycle time in nanoseconds

mmin (int): Minimum main memory in kilobytes

mmax (int): Maximum main memory in kilobytes

cach (int): Cache memory in kilobytes

chmin (int): Minimum channels in units

chmax (int): Maximum channels in units

Targets:

prp (int): Published relative performance

Source: https://archive.ics.uci.edu/ml/datasets/Computer+Hardware

Examples

Load in the data set:

```
>>> dataset = CPU()
>>> dataset.shape
(209, 9)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((209, 8), (209,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((162, 8), (162,), (47, 8), (47,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.facebook_comments module

Facebook comments data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.facebook_comments.**FacebookComments**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Instances in this dataset contain features extracted from Facebook posts. The task associated with the data is to predict how many comments the post will receive.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

page_popularity (int): Defines the popularity of support for the source of the document

page_checkins (int): Describes how many individuals so far visited this place. This feature is only associated with places; e.g., some institution, place, theater, etc.

page_talking_about (int): Defines the daily interest of individuals towards source of the document/post. The people who actually come back to the page, after liking the page. This include activities such as comments, likes to a post, shares etc., by visitors to the page

page_category (int): Defines the category of the source of the document; e.g., place, institution, branch etc.

agg[n] for n=0..24 (float): These features are aggreagted by page, by calculating min, max, average, median and standard deviation of essential features

cc1 (int): The total number of comments before selected base date/time

cc2 (int): The number of comments in the last 24 hours, relative to base date/time

cc3 (int): The number of comments in the last 48 to last 24 hours relative to base date/time

cc4 (int): The number of comments in the first 24 hours after the publication of post but before base date/time

cc5 (int): The difference between cc2 and cc3

base_time (int): Selected time in order to simulate the scenario, ranges from 0 to 71

post_length (int): Character count in the post

post_share_count (int): This feature counts the number of shares of the post, how many people had shared this post onto their timeline

post_promotion_status (int): Binary feature. To reach more people with posts in News Feed, individuals can promote their post and this feature indicates whether the post is promoted or not

h_local (int): This describes the hours for which we have received the target variable/comments. Ranges from 0 to 23

day_published[n] for n=0..6 (int): Binary feature. This represents the day (Sunday-Saturday) on which the post was published

day[n] for n=0..6 (int): Binary feature. This represents the day (Sunday-Saturday) on selected base date/time

Targets: ncomments (int): The number of comments in the next *h_local* hours

Source: https://archive.ics.uci.edu/ml/datasets/Facebook+Comment+Volume+Dataset

Examples

Load in the data set:

```
>>> dataset = FacebookComments()
>>> dataset.shape
(199030, 54)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((199030, 54), (199030,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((159211, 54), (159211,), (39819, 54), (39819,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.facebook_metrics module

Facebook metrics data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.facebook_metrics.**FacebookMetrics**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

The data is related to posts' published during the year of 2014 on the Facebook's page of a renowned cosmetics brand.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

page_likes(int): The total number of likes of the Facebook page at the given time.

post_type (int): The type of post. Here 0 means 'Photo', 1 means 'Status', 2 means 'Link' and 3 means 'Video'

post_category (int): The category of the post.

post_month (int): The month the post was posted, from 1 to 12 inclusive.

post_weekday (int): The day of the week the post was posted, from 1 to 7 inclusive.

post_hour (int): The hour the post was posted, from 0 to 23 inclusive

paid (int): Binary feature, whether the post was paid for.

Targets:

total_reach (int): The lifetime post total reach.

total_impressions (int): The lifetime post total impressions.

engaged_users (int): The lifetime engaged users.

post_consumers (int): The lifetime post consumers.

post_consumptions (int): The lifetime post consumptions.

post_impressions (int): The lifetime post impressions by people who liked the page.

post_reach (int): The lifetime post reach by people who liked the page.

post_engagements (int): The lifetime people who have liked the page and engaged with the post.

comments (int): The number of comments.

shares (int): The number of shares.

total_interactions (int): The total number of interactions

Source: https://archive.ics.uci.edu/ml/datasets/Facebook+metrics

Examples

Load in the data set:

```
>>> dataset = FacebookMetrics()
>>> dataset.shape
(500, 18)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((500, 7), (500, 11))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((388, 7), (388, 11), (112, 7), (112, 11))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.fish_bioconcentration module

Fish bioconcentration data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.fish_bioconcentration.**FishBioconcentration**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

This dataset contains manually-curated experimental bioconcentration factor (BCF) for 1058 molecules (continuous values). Each row contains a molecule, identified by a CAS number, a name (if available), and a SMILES string. Additionally, the KOW (experimental or predicted) is reported. In this database, you will also find Extended Connectivity Fingerprints (binary vectors of 1024 bits), to be used as independent variables to predict the BCF.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache
The name of the cache.

Type str or None

shape
Dimensions of the data set

Type tuple of integers

columns
List of column names in the data set

Type list of strings

Features:

logkow (float): Octanol water paritioning coefficient (experimental or predicted, as indicated by KOW type

kow_exp (int): Indicates whether logKOW is experimental or predicted, with 1 denoting experimental and 0 denoting predicted

smiles_[idx] for idx = 0..125 (int): Encoding of SMILES string to identify the 2D molecular structure. The encoding is as follows, where 'x' is a padding string to ensure that all the SMILES strings are of the same length:

- 0 = 'x'
- 1 = '#'
- 2 = '('
- 3 = ')'
- 4 = '+'
- 5 = '-'
- 6 = '/'
- 7 = '1'
- 8 = '2'

- 9 = '3'
- 10 = '4'
- 11 = '5'
- 12 = '6'
- 13 = '7'
- 14 = '8'
- 15 = '='
- 16 = '@'
- 17 = 'B'
- 18 = 'C'
- 19 = 'F'
- 20 = 'H'
- 21 = 'I'
- 22 = 'N'
- 23 = 'O'
- 24 = 'P'
- 25 = 'S'
- 26 = '['
- 27 = ''
- 28 = ']'
- 29 = 'c'
- 30 = 'i'
- 31 = 'l'
- 32 = 'n'
- 33 = 'o'
- 34 = 'r'
- 35 = 's'

Targets:

logbcf (float): Experimental fish bioconcentration factor (logarithm form)

Source: https://archive.ics.uci.edu/ml/datasets/QSAR+fish+bioconcentration+factor+%28BCF%29

Examples

Load in the data set:

```
>>> dataset = FishBioconcentration()
>>> dataset.shape
(1054, 129)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((1054, 128), (1054,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((825, 128), (825,), (229, 128), (229,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.fish_toxicity module

Fish toxicity data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.fish_toxicity.**FishToxicity**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

This dataset was used to develop quantitative regression QSAR models to predict acute aquatic toxicity towards the fish Pimephales promelas (fathead minnow) on a set of 908 chemicals. LC50 data, which is the concentration that causes death in 50% of test fish over a test duration of 96 hours, was used as model response

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

CIC0 (float): Information indices

SM1_Dz(Z) (float): 2D matrix-based descriptors

GATS1i (float): 2D autocorrelations

NdsCH (int) Atom-type counts

NdssC (int) Atom-type counts

MLOGP (float): Molecular properties

Targets:

LC50 (float): A concentration that causes death in 50% of test fish over a test duration of 96 hours. In -log(mol/L) units.

Source: https://archive.ics.uci.edu/ml/datasets/QSAR+fish+toxicity

Examples

Load in the data set:

```
>>> dataset = FishToxicity()
>>> dataset.shape
(908, 7)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((908, 6), (908,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((708, 6), (708,), (200, 6), (200,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.forest_fire module

Forest fire data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.forest_fire.**ForestFire**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

This is a difficult regression task, where the aim is to predict the burned area of forest fires, in the northeast region of Portugal, by using meteorological and other data.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

X (float): The x-axis spatial coordinate within the Montesinho park map. Ranges from 1 to 9.

Y (float): The y-axis spatial coordinate within the Montesinho park map Ranges from 2 to 9.

month (int): Month of the year. Ranges from 0 to 11

day (int): Day of the week. Ranges from 0 to 6

FFMC (float): FFMC index from the FWI system. Ranges from 18.7 to 96.20

DMC (float): DMC index from the FWI system. Ranges from 1.1 to 291.3

DC (float): DC index from the FWI system. Ranges from 7.9 to 860.6

ISI (float): ISI index from the FWI system. Ranges from 0.0 to 56.1

temp (float): Temperature in Celsius degrees. Ranges from 2.2 to 33.3

RH (float): Relative humidity in %. Ranges from 15.0 to 100.0

wind (float): Wind speed in km/h. Ranges from 0.4 to 9.4

rain (float): Outside rain in mm/m2. Ranges from 0.0 to 6.4

Targets:

area (float): The burned area of the forest (in ha). Ranges from 0.00 to 1090.84

Notes

The target variable is very skewed towards 0.0, thus it may make sense to model with the logarithm transform.

Source: https://archive.ics.uci.edu/ml/datasets/Forest+Fires

Examples

Load in the data set:

```
>>> dataset = ForestFire()
>>> dataset.shape
(517, 13)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((517, 12), (517,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((401, 12), (401,), (116, 12), (116,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.gas_turbine module

Gas turbine data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.gas_turbine.**GasTurbine**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Data have been generated from a sophisticated simulator of a Gas Turbines (GT), mounted on a Frigate characterized by a COmbined Diesel eLectric And Gas (CODLAG) propulsion plant type.

The experiments have been carried out by means of a numerical simulator of a naval vessel (Frigate) characterized by a Gas Turbine (GT) propulsion plant. The different blocks forming the complete simulator (Propeller, Hull, GT, Gear Box and Controller) have been developed and fine tuned over the year on several similar real propulsion plants. In view of these observations the available data are in agreement with a possible real vessel.

In this release of the simulator it is also possible to take into account the performance decay over time of the GT components such as GT compressor and turbines.

The propulsion system behaviour has been described with this parameters:

- Ship speed (linear function of the lever position lp).

- Compressor degradation coefficient kMc.
- Turbine degradation coefficient kMt.

so that each possible degradation state can be described by a combination of this triple (lp,kMt,kMc).

The range of decay of compressor and turbine has been sampled with an uniform grid of precision 0.001 so to have a good granularity of representation.

In particular for the compressor decay state discretization the kMc coefficient has been investigated in the domain [1; 0.95], and the turbine coefficient in the domain [1; 0.975].

Ship speed has been investigated sampling the range of feasible speed from 3 knots to 27 knots with a granularity of representation equal to tree knots.

A series of measures (16 features) which indirectly represents of the state of the system subject to performance decay has been acquired and stored in the dataset over the parameter's space.

Parameters `cache` (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

`cache`
The name of the cache.

Type str or None

`shape`
Dimensions of the data set

Type tuple of integers

`columns`
List of column names in the data set

Type list of strings

Features:

lever_position (float) The position of the lever

ship_speed (float): The ship speed, in knots

shaft_torque (float): The shaft torque of the gas turbine, in kN m

turbine_revolution_rate (float): The gas turbine rate of revolutions, in rpm

generator_revolution_rate (float): The gas generator rate of revolutions, in rpm

starboard_propeller_torque (float): The torque of the starboard propeller, in kN

port_propeller_torque (float): The torque of the port propeller, in kN

turbine_exit_temp (float): Height pressure turbine exit temperature, in celcius

inlet_temp (float): Gas turbine compressor inlet air temperature, in celcius

outlet_temp (float): Gas turbine compressor outlet air temperature, in celcius

turbine_exit_pres (float): Height pressure turbine exit pressure, in bar

inlet_pres (float): Gas turbine compressor inlet air pressure, in bar

outlet_pres (float): Gas turbine compressor outlet air pressure, in bar

exhaust_pres (float): Gas turbine exhaust gas pressure, in bar

turbine_injection_control (float): Turbine injection control, in percent

fuel_flow (float): Fuel flow, in kg/s

Targets:

compressor_decay (type): Gas turbine compressor decay state coefficient

turbine_decay (type): Gas turbine decay state coefficient

Source: https://archive.ics.uci.edu/ml/datasets/Condition+Based+Maintenance+of+Naval+Propulsion+Plants

Examples

Load in the data set:

```
>>> dataset = GasTurbine()
>>> dataset.shape
(11934, 18)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((11934, 16), (11934, 2))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((9516, 16), (9516, 2), (2418, 16), (2418, 2))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.nanotube module

Nanotube data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.nanotube.**Nanotube**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

CASTEP can simulate a wide range of properties of materials proprieties using density functional theory (DFT). DFT is the most successful method calculates atomic coordinates faster than other mathematical approaches, and it also reaches more accurate results. The dataset is generated with CASTEP using CNT geometry optimization. Many CNTs are simulated in CASTEP, then geometry optimizations are calculated. Initial coordinates of all carbon atoms are generated randomly. Different chiral vectors are used for each CNT simulation.

The atom type is selected as carbon, bond length is used as 1.42 AÂ° (default value). CNT calculation parameters are used as default parameters. To finalize the computation, CASTEP uses a parameter named as elec_energy_tol (electrical energy tolerance) (default 1x10-5 eV) which represents that the change in the total energy from one iteration to the next remains below some tolerance value per atom for a few self-consistent field steps. Initial

atomic coordinates (u, v, w), chiral vector (n, m) and calculated atomic coordinates (u, v, w) are obtained from the output files.

Parameters `cache` (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

`cache`
The name of the cache.

Type str or None

`shape`
Dimensions of the data set

Type tuple of integers

`columns`
List of column names in the data set

Type list of strings

Features:

Chiral indice n (int): n parameter of the selected chiral vector

Chiral indice m (int): m parameter of the selected chiral vector

Initial atomic coordinate u (float): Randomly generated u parameter of the initial atomic coordinates of all carbon atoms.

Initial atomic coordinate v (float): Randomly generated v parameter of the initial atomic coordinates of all carbon atoms.

Initial atomic coordinate w (float): Randomly generated w parameter of the initial atomic coordinates of all carbon atoms.

Targets:

Calculated atomic coordinates u (float): Calculated u parameter of the atomic coordinates of all carbon atoms

Calculated atomic coordinates v (float): Calculated v parameter of the atomic coordinates of all carbon atoms

Calculated atomic coordinates w (float): Calculated w parameter of the atomic coordinates of all carbon atoms

Sources: https://archive.ics.uci.edu/ml/datasets/Carbon+Nanotubes https://doi.org/10.1007/s00339-016-0153-1 https://doi.org/10.17341/gazimmfd.337642

Examples

Load in the data set:

```
>>> dataset = Nanotube()
>>> dataset.shape
(10721, 8)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((10721, 5), (10721, 3))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((8541, 5), (8541, 3), (2180, 5), (2180, 3))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.new_taipei_housing module

New Taipei Housing data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.new_taipei_housing.**NewTaipeiHousing**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

The "real estate valuation" is a regression problem. The market historical data set of real estate valuation are collected from Sindian Dist., New Taipei City, Taiwan.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache
The name of the cache.

Type str or None

shape
Dimensions of the data set

Type tuple of integers

columns
List of column names in the data set

Type list of strings

Features:

transaction_date (float): The transaction date encoded as a floating point value. For instance, 2013.250 is March 2013 and 2013.500 is June March

house_age (float): The age of the house

mrt_distance (float): Distance to the nearest MRT station

n_stores (int): Number of convenience stores

lat (float): Latitude

lng (float): Longitude

Targets:

house_price (float): House price of unit area

Source: https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set

Examples

Load in the data set:

```
>>> dataset = NewTaipeiHousing()
>>> dataset.shape
(414, 7)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((414, 6), (414,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((323, 6), (323,), (91, 6), (91,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.parkinsons module

Parkinsons data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.parkinsons.**Parkinsons**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

This dataset is composed of a range of biomedical voice measurements from 42 people with early-stage Parkinson's disease recruited to a six-month trial of a telemonitoring device for remote symptom progression monitoring. The recordings were automatically captured in the patient's homes.

Columns in the table contain subject number, subject age, subject gender, time interval from baseline recruitment date, motor UPDRS, total UPDRS, and 16 biomedical voice measures. Each row corresponds to one of 5,875 voice recording from these individuals. The main aim of the data is to predict the motor and total UPDRS scores ('motor_UPDRS' and 'total_UPDRS') from the 16 voice measures.

Parameters `cache` (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

`cache`
The name of the cache.

Type str or None

`shape`
Dimensions of the data set

Type tuple of integers

`columns`
List of column names in the data set

Type list of strings

Features:

subject# (int): Integer that uniquely identifies each subject

age (int): Subject age

sex (int): Binary feature. Subject sex, with 0 being male and 1 female

test_time (float): Time since recruitment into the trial. The integer part is the number of days since recruitment

Jitter(%) (float): Measure of variation in fundamental frequency

Jitter(Abs) (float): Measure of variation in fundamental frequency

Jitter:RAP (float): Measure of variation in fundamental frequency

Jitter:PPQ5 (float): Measure of variation in fundamental frequency

Jitter:DDP (float): Measure of variation in fundamental frequency

Shimmer (float): Measure of variation in amplitude

Shimmer(dB) (float): Measure of variation in amplitude

Shimmer:APQ3 (float): Measure of variation in amplitude

Shimmer:APQ5 (float): Measure of variation in amplitude

Shimmer:APQ11 (float): Measure of variation in amplitude

Shimmer:DDA (float): Measure of variation in amplitude

NHR (float): Measure of ratio of noise to tonal components in the voice

HNR (float): Measure of ratio of noise to tonal components in the voice

RPDE (float): A nonlinear dynamical complexity measure

DFA (float): Signal fractal scaling exponent

PPE (float): A nonlinear measure of fundamental frequency variation

Targets:

motor_UPDRS (float): Clinician's motor UPDRS score, linearly interpolated

total_UPDRS (float): Clinician's total UPDRS score, linearly interpolated

Source: https://archive.ics.uci.edu/ml/datasets/Parkinsons+Telemonitoring

Examples

Load in the data set:

```
>>> dataset = Parkinsons()
>>> dataset.shape
(5875, 22)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((5875, 20), (5875, 2))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((4659, 20), (4659, 2), (1216, 20), (1216, 2))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.power_plant module

Power plant data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.power_plant.**PowerPlant**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

The dataset contains 9568 data points collected from a Combined Cycle Power Plant over 6 years (2006-2011), when the power plant was set to work with full load. Features consist of hourly average ambient variables Temperature (T), Ambient Pressure (AP), Relative Humidity (RH) and Exhaust Vacuum (V) to predict the net hourly electrical energy output (EP) of the plant.

A combined cycle power plant (CCPP) is composed of gas turbines (GT), steam turbines (ST) and heat recovery steam generators. In a CCPP, the electricity is generated by gas and steam turbines, which are combined in one cycle, and is transferred from one turbine to another. While the Vacuum is colected from and has effect on the Steam Turbine, he other three of the ambient variables effect the GT performance.

For comparability with our baseline studies, and to allow 5x2 fold statistical tests be carried out, we provide the data shuffled five times. For each shuffling 2-fold CV is carried out and the resulting 10 measurements are used for statistical testing.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape
Dimensions of the data set

Type tuple of integers

columns
List of column names in the data set

Type list of strings

Features:

AT (float): Hourly average temperature in Celsius, ranges from 1.81 to 37.11

V (float): Hourly average exhaust vacuum in cm Hg, ranges from 25.36 to 81.56

AP (float): Hourly average ambient pressure in millibar, ranges from 992.89 to 1033.30

RH (float): Hourly average relative humidity in percent, ranges from 25.56 to 100.16

Targets:

PE (float): Net hourly electrical energy output in MW, ranges from 420.26 to 495.76

Source: https://archive.ics.uci.edu/ml/datasets/Combined+Cycle+Power+Plant

Examples

Load in the data set:

```
>>> dataset = PowerPlant()
>>> dataset.shape
(9568, 5)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((9568, 4), (9568,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((7633, 4), (7633,), (1935, 4), (1935,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.protein module

Protein data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.protein.**Protein**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

This is a data set of Physicochemical Properties of Protein Tertiary Structure. The data set is taken from CASP 5-9. There are 45730 decoys and size varying from 0 to 21 armstrong.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

F1 (float): Total surface area

F2 (float): Non polar exposed area

F3 (float): Fractional area of exposed non polar residue

F4 (float): Fractional area of exposed non polar part of residue

F5 (float): Molecular mass weighted exposed area

F6 (float): Average deviation from standard exposed area of residue

F7 (float): Euclidean distance

F8 (float): Secondary structure penalty

F9 (float): Spacial Distribution constraints (N,K Value)

Targets:

RMSD (float): Size of the residue

Source: https://archive.ics.uci.edu/ml/datasets/Physicochemical+Properties+of+Protein+Tertiary+Structure

Examples

Load in the data set:

```
>>> dataset = Protein()
>>> dataset.shape
(45730, 10)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((45730, 9), (45730,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((36580, 9), (36580,), (9150, 9), (9150,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.servo module

Servo data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.servo.**Servo**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Data was from a simulation of a servo system.

Ross Quinlan:

This data was given to me by Karl Ulrich at MIT in 1986. I didn't record his description at the time, but here's his subsequent (1992) recollection:

"I seem to remember that the data was from a simulation of a servo system involving a servo amplifier, a motor, a lead screw/nut, and a sliding carriage of some sort. It may have been on of the translational axes of a robot on the 9th floor of the AI lab. In any case, the output value is almost certainly a rise time, or the time required for the system to respond to a step change in a position set point."

(Quinlan, ML'93)

"This is an interesting collection of data provided by Karl Ulrich. It covers an extremely non-linear phenomenon - predicting the rise time of a servomechanism in terms of two (continuous) gain settings and two (discrete) choices of mechanical linkages."

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache
The name of the cache.

Type str or None

shape
Dimensions of the data set

Type tuple of integers

columns
List of column names in the data set

Type list of strings

Features:

motor (int): Motor, ranges from 0 to 4 inclusive

screw (int): Screw, ranges from 0 to 4 inclusive

pgain (int): PGain, ranges from 3 to 6 inclusive

vgain (int): VGain, ranges from 1 to 5 inclusive

Targets:

class (float): Class values, ranges from 0.13 to 7.10 inclusive

Source: https://archive.ics.uci.edu/ml/datasets/Servo

Examples

Load in the data set:

```
>>> dataset = Servo()
>>> dataset.shape
(167, 5)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((167, 4), (167,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((131, 4), (131,), (36, 4), (36,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.solar_flare module

Solar flare data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.solar_flare.**SolarFlare**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Each class attribute counts the number of solar flares of a certain class that occur in a 24 hour period.

The database contains 3 potential classes, one for the number of times a certain type of solar flare occured in a 24 hour period.

Each instance represents captured features for 1 active region on the sun.

The data are divided into two sections. The second section (flare.data2) has had much more error correction applied to the it, and has consequently been treated as more reliable.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

class (int): Code for class (modified Zurich class). Ranges from 0 to 6 inclusive

spot_size (int): Code for largest spot size. Ranges from 0 to 5 inclusive

spot_distr (int): Code for spot distribution. Ranges from 0 to 3 inclusive

activity (int): Binary feature indicating 1 = reduced and 2 = unchanged

evolution (int): 0 = decay, 1 = no growth and 2 = growth

flare_activity (int): Previous 24 hour flare activity code, where 0 = nothing as big as an M1, 1 = one M1 and 2 = more activity than one M1

is_complex (int): Binary feature indicating historically complex

became_complex (int): Binary feature indicating whether the region became historically complex on this pass across the sun's disk

large (int): Binary feature, indicating whether area is large

large_spot (int): Binary feature, indicating whether the area of the largest spot is greater than 5

Targets:

C-class (int): C-class flares production by this region in the following 24 hours (common flares)

M-class (int): M-class flares production by this region in the following 24 hours (common flares)

X-class (int): X-class flares production by this region in the following 24 hours (common flares)

Source: https://archive.ics.uci.edu/ml/datasets/Solar+Flare

Examples

Load in the data set:

```
>>> dataset = SolarFlare()
>>> dataset.shape
(1066, 13)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((1066, 10), (1066, 3))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((837, 10), (837, 3), (229, 10), (229, 3))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.space_shuttle module

Space shuttle data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.space_shuttle.**SpaceShuttle**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

The motivation for collecting this database was the explosion of the USA Space Shuttle Challenger on 28 January, 1986. An investigation ensued into the reliability of the shuttle's propulsion system. The explosion was eventually traced to the failure of one of the three field joints on one of the two solid booster rockets. Each of these six field joints includes two O-rings, designated as primary and secondary, which fail when phenomena called erosion and blowby both occur.

The night before the launch a decision had to be made regarding launch safety. The discussion among engineers and managers leading to this decision included concern that the probability of failure of the O-rings depended on the temperature t at launch, which was forecase to be 31 degrees F. There are strong engineering reasons based on the composition of O-rings to support the judgment that failure probability may rise monotonically as temperature drops. One other variable, the pressure s at which safety testing for field join leaks was performed, was available, but its relevance to the failure process was unclear.

Draper's paper includes a menacing figure graphing the number of field joints experiencing stress vs. liftoff temperature for the 23 shuttle flights previous to the Challenger disaster. No previous liftoff temperature was under 53 degrees F. Although tremendous extrapolation must be done from the given data to assess risk at 31 degrees F, it is obvious even to the layman "to foresee the unacceptably high risk created by launching at 31 degrees F." For more information, see Draper (1993) or the other previous analyses.

The task is to predict the number of O-rings that will experience thermal distress for a given flight when the launch temperature is below freezing.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache
The name of the cache.

Type str or None

shape
Dimensions of the data set

Type tuple of integers

columns
List of column names in the data set

Type list of strings

Features:

idx (int): Temporal order of flight

temp (int): Launch temperature in Fahrenheit

pres (int): Leak-check pressure in psi

n_risky_rings (int): Number of O-rings at risk on a given flight

Targets:

n_distressed_rings (int): Number of O-rings experiencing thermal distress

Source: https://archive.ics.uci.edu/ml/datasets/Challenger+USA+Space+Shuttle+O-Ring

Examples

Load in the data set:

```
>>> dataset = SpaceShuttle()
>>> dataset.shape
(23, 5)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((23, 4), (23,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((20, 4), (20,), (3, 4), (3,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.stocks module

Stocks data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.stocks.**Stocks**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

There are three disadvantages of weighted scoring stock selection models. First, they cannot identify the relations between weights of stock-picking concepts and performances of portfolios. Second, they cannot systematically discover the optimal combination for weights of concepts to optimize the performances. Third, they are unable to meet various investors' preferences.

This study aims to more efficiently construct weighted scoring stock selection models to overcome these disadvantages. Since the weights of stock-picking concepts in a weighted scoring stock selection model can be regarded as components in a mixture, we used the simplex centroid mixture design to obtain the experimental sets of weights. These sets of weights are simulated with US stock market historical data to obtain their performances. Performance prediction models were built with the simulated performance data set and artificial neural networks.

Furthermore, the optimization models to reflect investors' preferences were built up, and the performance prediction models were employed as the kernel of the optimization models so that the optimal solutions can now be solved with optimization techniques. The empirical values of the performances of the optimal weighting combinations generated by the optimization models showed that they can meet various investors' preferences and outperform those of S&P's 500 not only during the training period but also during the testing period.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache
The name of the cache.

Type str or None

shape
Dimensions of the data set

Type tuple of integers

columns
List of column names in the data set

Type list of strings

Features:

bp (float): Large B/P

roe (float): Large ROE

sp (float): Large S/P

return_rate (float): Large return rate in the last quarter

market_value (float): Large market value

small_risk (float): Small systematic risk

orig_annual_return (float): Annual return

orig_excess_return (float): Excess return

orig_risk (float): Systematic risk

orig_total_risk (float): Total risk

orig_abs_win_rate (float): Absolute win rate

orig_rel_win_rate (float): Relative win rate

Targets:

annual_return (float): Annual return

excess_return (float): Excess return

risk (float): Systematic risk

total_risk (float): Total risk

abs_win_rate (float): Absolute win rate

rel_win_rate (float): Relative win rate

Source: https://archive.ics.uci.edu/ml/datasets/Stock+portfolio+performance

Examples

Load in the data set:

```
>>> dataset = Stocks()
>>> dataset.shape
(252, 19)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((252, 12), (252, 6))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((197, 12), (197, 6), (55, 12), (55, 6))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.superconductivity module

Superconductivity data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.superconductivity.**Superconductivity**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

This dataset contains data on 21,263 superconductors and their relevant features. The goal here is to predict the critical temperature based on the features extracted.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

- number_of_elements (int)
- mean_atomic_mass (float)
- wtd_mean_atomic_mass (float)
- gmean_atomic_mass (float)
- wtd_gmean_atomic_mass (float)
- entropy_atomic_mass (float)
- wtd_entropy_atomic_mass (float)
- range_atomic_mass (float)
- wtd_range_atomic_mass (float)
- std_atomic_mass (float)
- wtd_std_atomic_mass (float)
- mean_fie (float)
- wtd_mean_fie (float)

- gmean_fie (float)
- wtd_gmean_fie (float)
- entropy_fie (float)
- wtd_entropy_fie (float)
- range_fie (float)
- wtd_range_fie (float)
- std_fie (float)
- wtd_std_fie (float)
- mean_atomic_radius (float)
- wtd_mean_atomic_radius (float)
- gmean_atomic_radius (float)
- wtd_gmean_atomic_radius (float)
- entropy_atomic_radius (float)
- wtd_entropy_atomic_radius (float)
- range_atomic_radius (float)
- wtd_range_atomic_radius (float)
- std_atomic_radius (float)
- wtd_std_atomic_radius (float)
- mean_Density (float)
- wtd_mean_Density (float)
- gmean_Density (float)
- wtd_gmean_Density (float)
- entropy_Density (float)
- wtd_entropy_Density (float)
- range_Density (float)
- wtd_range_Density (float)
- std_Density (float)
- wtd_std_Density (float)
- mean_ElectronAffinity (float)
- wtd_mean_ElectronAffinity (float)
- gmean_ElectronAffinity (float)
- wtd_gmean_ElectronAffinity (float)
- entropy_ElectronAffinity (float)
- wtd_entropy_ElectronAffinity (float)
- range_ElectronAffinity (float)
- wtd_range_ElectronAffinity (float)

- std_ElectronAffinity (float)
- wtd_std_ElectronAffinity (float)
- mean_FusionHeat (float)
- wtd_mean_FusionHeat (float)
- gmean_FusionHeat (float)
- wtd_gmean_FusionHeat (float)
- entropy_FusionHeat (float)
- wtd_entropy_FusionHeat (float)
- range_FusionHeat (float)
- wtd_range_FusionHeat (float)
- std_FusionHeat (float)
- wtd_std_FusionHeat (float)
- mean_ThermalConductivity (float)
- wtd_mean_ThermalConductivity (float)
- gmean_ThermalConductivity (float)
- wtd_gmean_ThermalConductivity (float)
- entropy_ThermalConductivity (float)
- wtd_entropy_ThermalConductivity (float)
- range_ThermalConductivity (float)
- wtd_range_ThermalConductivity (float)
- std_ThermalConductivity (float)
- wtd_std_ThermalConductivity (float)
- mean_Valence (float)
- wtd_mean_Valence (float)
- gmean_Valence (float)
- wtd_gmean_Valence (float)
- entropy_Valence (float)
- wtd_entropy_Valence (float)
- range_Valence (float)
- wtd_range_Valence (float)
- std_Valence (float)
- wtd_std_Valence (float)

Targets:

- critical_temp (float)

Source: https://archive.ics.uci.edu/ml/datasets/Superconductivty+Data

Examples

Load in the data set:

```
>>> dataset = Superconductivity()
>>> dataset.shape
(21263, 82)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((21263, 81), (21263,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((17004, 81), (17004,), (4259, 81), (4259,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.tehran_housing module

Tehran housing data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.tehran_housing.**TehranHousing**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Data set includes construction cost, sale prices, project variables, and economic variables corresponding to real estate single-family residential apartments in Tehran, Iran.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

start_year (int): Start year in the Persian calendar

start_quarter (int) Start quarter in the Persian calendar

completion_year (int) Completion year in the Persian calendar

completion_quarter (int) Completion quarter in the Persian calendar

V-1..V-8 (floats): Project physical and financial variables

V-11-1..29-1 (floats): Economic variables and indices in time, lag 1

V-11-2..29-2 (floats): Economic variables and indices in time, lag 2

V-11-3..29-3 (floats): Economic variables and indices in time, lag 3

V-11-4..29-4 (floats): Economic variables and indices in time, lag 4

V-11-5..29-5 (floats): Economic variables and indices in time, lag 5

Targets: construction_cost (float) sale_price (float)

Source: https://archive.ics.uci.edu/ml/datasets/Residential+Building+Data+Set

Examples

Load in the data set:

```
>>> dataset = TehranHousing()
>>> dataset.shape
(371, 109)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((371, 107), (371, 2))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((288, 107), (288, 2), (83, 107), (83, 2))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

doubt.datasets.yacht module

Yacht data set.

This data set is from the UCI data set archive, with the description being the original description verbatim. Some feature names may have been altered, based on the description.

class doubt.datasets.yacht.**Yacht**(*cache: Optional[str] = '.dataset_cache'*)

Bases: doubt.datasets._dataset.BaseDataset

Prediction of residuary resistance of sailing yachts at the initial design stage is of a great value for evaluating the ship's performance and for estimating the required propulsive power. Essential inputs include the basic hull dimensions and the boat velocity.

The Delft data set comprises 308 full-scale experiments, which were performed at the Delft Ship Hydromechanics Laboratory for that purpose.

These experiments include 22 different hull forms, derived from a parent form closely related to the "Standfast 43" designed by Frans Maas.

Parameters **cache** (*str or None, optional*) – The name of the cache. It will be saved to *cache* in the current working directory. If None then no cache will be saved. Defaults to '.dataset_cache'.

cache

The name of the cache.

Type str or None

shape

Dimensions of the data set

Type tuple of integers

columns

List of column names in the data set

Type list of strings

Features:

pos (float): Longitudinal position of the center of buoyancy, adimensional

prismatic (float): Prismatic coefficient, adimensional

displacement (float): Length-displacement ratio, adimensional

beam_draught (float): Beam-draught ratio, adimensional

length_beam (float): Length-beam ratio, adimensional

froude_no (float): Froude number, adimensional

Targets:

resistance (float): Residuary resistance per unit weight of displacement, adimensional

Source: https://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics

Examples

Load in the data set:

```
>>> dataset = Yacht()
>>> dataset.shape
(308, 7)
```

Split the data set into features and targets, as NumPy arrays:

```
>>> X, y = dataset.split()
>>> X.shape, y.shape
((308, 6), (308,))
```

Perform a train/test split, also outputting NumPy arrays:

```
>>> train_test_split = dataset.split(test_size=0.2, random_seed=42)
>>> X_train, X_test, y_train, y_test = train_test_split
>>> X_train.shape, y_train.shape, X_test.shape, y_test.shape
((235, 6), (235,), (73, 6), (73,))
```

Output the underlying Pandas DataFrame:

```
>>> df = dataset.to_pandas()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

Module contents

doubt.models package

Subpackages

doubt.models.boot package

Submodules

doubt.models.boot.boot module

Bootstrap wrapper for datasets and models

class doubt.models.boot.boot.**Boot**(*input: object*, *random_seed: Optional[float] = None*)

Bases: `object`

Bootstrap wrapper for datasets and models.

Datasets can be any sequence of numeric input, from which bootstrapped statistics can be calculated, with confidence intervals included.

The models can be any model that is either callable or equipped with a *predict* method, such as all the models in *scikit-learn*, *pytorch* and *tensorflow*, and the bootstrapped model can then produce predictions with prediction intervals.

The bootstrapped prediction intervals are computed using the an extension of method from [2] which also takes validation error into account. To remedy this, the .632+ bootstrap estimate from [1] has been used. Read more in [3].

Parameters

- **input** (*float array or model*) – Either a dataset to calculate bootstrapped statistics on, or an model for which bootstrapped predictions will be computed.
- **random_seed** (*float or None*) – The random seed used for bootstrapping. If set to None then no seed will be set. Defaults to None.

Examples

Compute the bootstrap distribution of the mean, with a 95% confidence interval:

```
>>> from doubt.datasets import FishToxicity
>>> X, y = FishToxicity().split()
>>> boot = Boot(y, random_seed=42)
>>> boot.compute_statistic(np.mean)
(4.064430616740088, array([3.97621225, 4.16582087]))
```

Alternatively, we can output the whole bootstrap distribution:

```
>>> boot.compute_statistic(np.mean, n_boots=3, return_all=True)
(4.064430616740088, array([4.05705947, 4.06197577, 4.05728414]))
```

Wrap a scikit-learn model and get prediction intervals:

```
>>> from sklearn.linear_model import LinearRegression
>>> from doubt.datasets import PowerPlant
>>> X, y = PowerPlant().split()
>>> linreg = Boot(LinearRegression(), random_seed=42)
>>> linreg = linreg.fit(X, y)
>>> linreg.predict([10, 30, 1000, 50], uncertainty=0.05)
(481.99688920651676, array([473.50425407, 490.14061895]))
```

Sources:

[1]: Friedman, J., Hastie, T., & Tibshirani, R. (2001). The elements of statistical learning (Vol. 1, No. 10). New York: Springer series in statistics.

[2]: Kumar, S., & Srivistava, A. N. (2012). Bootstrap prediction intervals in non-parametric regression with applications to anomaly detection.

[3]: https://saattrupdan.github.io/2020-03-01-bootstrap-prediction/

doubt.models.boot.boot.**compute_statistic**(*self*, *statistic: Callable[[Sequence[Union[float, int]]], float]*, *n_boots: Optional[int] = None*, *uncertainty: float = 0.05*, *quantiles: Optional[Sequence[float]] = None*, *return_all: bool = False*) → Union[float, Tuple[float, numpy.ndarray]]

Compute bootstrapped statistic.

Parameters

- **statistic** (*numeric array -> float*) – The statistic to be computed on bootstrapped samples.

- **n_boots** (*int or None*) – The number of resamples to bootstrap. If None then it is set to the square root of the data set. Defaults to None
- **uncertainty** (*float*) – The uncertainty used to compute the confidence interval of the bootstrapped statistic. Not used if *return_all* is set to True or if *quantiles* is not None. Defaults to 0.05.
- **quantiles** (*sequence of floats or None, optional*) – List of quantiles to output, as an alternative to the *uncertainty* argument, and will not be used if that argument is set. If None then *uncertainty* is used. Defaults to None.
- **return_all** (*bool*) – Whether all bootstrapped statistics should be returned instead of the confidence interval. Defaults to False.

Returns The statistic, and if *uncertainty* is set then also the confidence interval, or if *quantiles* is set then also the specified quantiles, or if *return_all* is set then also all of the bootstrapped statistics.

Return type a float or a pair of a float and an array of floats

doubt.models.boot.boot.**fit**(*self*, *X: Sequence[float]*, *y: Sequence[float]*, *n_boots: Optional[int] = None*)

Fits the model to the data.

Parameters

- **X** (*float array*) – The array containing the data set, either of shape (f,) or (n, f), with n being the number of samples and f being the number of features.
- **y** (*float array*) – The array containing the target values, of shape (n,)
- **n_boots** (*int or None*) – The number of resamples to bootstrap. If None then it is set to the square root of the data set. Defaults to None

doubt.models.boot.boot.**predict**(*self*, *X: Sequence[float]*, *n_boots: Optional[int] = None*, *uncertainty: Optional[float] = None*, *quantiles: Optional[Sequence[float]] = None*) → Tuple[Union[float, numpy.ndarray], numpy.ndarray]

Compute bootstrapped predictions.

Parameters

- **X** (*float array*) – The array containing the data set, either of shape (f,) or (n, f), with n being the number of samples and f being the number of features.
- **n_boots** (*int or None, optional*) – The number of resamples to bootstrap. If None then it is set to the square root of the data set. Defaults to None
- **uncertainty** (*float or None, optional*) – The uncertainty used to compute the prediction interval of the bootstrapped prediction. If None then no prediction intervals are returned. Defaults to None.
- **quantiles** (*sequence of floats or None, optional*) – List of quantiles to output, as an alternative to the *uncertainty* argument, and will not be used if that argument is set. If None then *uncertainty* is used. Defaults to None.

Returns The bootstrapped predictions, and the confidence intervals if *uncertainty* is not None, or the specified quantiles if *quantiles* is not None.

Return type float array or pair of float arrays

Module contents

doubt.models.glm package

Submodules

doubt.models.glm.quantile_loss module

Implementation of the quantile loss function

doubt.models.glm.quantile_loss.**quantile_loss**(*predictions: Sequence[float]*, *targets: Sequence[float]*, *quantile: float*) → float

Quantile loss function.

Parameters

- **predictions** (*sequence of floats*) – Model predictions, of shape [n_samples,].
- **targets** (*sequence of floats*) – Target values, of shape [n_samples,].
- **quantile** (*float*) – The quantile we are seeking. Must be between 0 and 1.

Returns The quantile loss.

Return type float

doubt.models.glm.quantile_loss.**smooth_quantile_loss**(*predictions: Sequence[float]*, *targets: Sequence[float]*, *quantile: float*, *alpha: float = 0.4*) → float

The smooth quantile loss function from [1].

Parameters

- **predictions** (*sequence of floats*) – Model predictions, of shape [n_samples,].
- **targets** (*sequence of floats*) – Target values, of shape [n_samples,].
- **quantile** (*float*) – The quantile we are seeking. Must be between 0 and 1.
- **alpha** (*float, optional*) – Smoothing parameter. Defaults to 0.4.

Returns The smooth quantile loss.

Return type float

Sources:

[1]: Songfeng Zheng (2011). Gradient Descent Algorithms for Quantile Regression With Smooth Approximation. International Journal of Machine Learning and Cybernetics.

doubt.models.glm.quantile_regressor module

Quantile regression for generalised linear models

class doubt.models.glm.quantile_regressor.**QuantileRegressor**(*model: Union[sklearn.linear_model._base.LinearRegression, sklearn.linear_model._glm.glm.GeneralizedLinearRegres max_iter: Optional[int] = None, uncertainty: float = 0.05, quantiles: Optional[Sequence[float]] = None, alpha: float = 0.4*)

Bases: doubt.models._model.BaseModel

Quantile regression for generalised linear models.

This uses BFGS optimisation of the smooth quantile loss from [1].

Parameters

- **max_iter** (*int*) – The maximal number of iterations to train the model for. Defaults to 10,000.
- **uncertainty** (*float*) – The uncertainty in the prediction intervals. Must be between 0 and 1. Defaults to 0.05.
- **quantiles** (*sequence of floats or None, optional*) – List of quantiles to output, as an alternative to the *uncertainty* argument, and will not be used if that argument is set. If None then *uncertainty* is used. Defaults to None.
- **alpha** (*float, optional*) – Smoothing parameter. Defaults to 0.4.

Examples

Fitting and predicting follows scikit-learn syntax:

```
>>> from doubt.datasets import Concrete
>>> from sklearn.linear_model import PoissonRegressor
>>> X, y = Concrete().split(random_seed=42)
>>> model = QuantileRegressor(PoissonRegressor(max_iter=10_000),
...                           uncertainty=0.05)
>>> model.fit(X, y).predict(X)[0].shape
(1030,)
>>> x = [500, 0, 0, 100, 2, 1000, 500, 20]
>>> pred, interval = model.predict(x)
>>> pred, interval
(78.50224243713622, array([ 19.27889844, 172.71408196]))
```

Sources:

[1]: Songfeng Zheng (2011). Gradient Descent Algorithms for Quantile Regression With Smooth Approximation. International Journal of Machine Learning and Cybernetics.

fit(*X: Sequence[Sequence[float]], y: Sequence[float], random_seed: Optional[int] = None*)

Fit the model.

Parameters

- **X** (*float matrix*) – The array containing the data set, either of shape (n,) or (n, f), with n being the number of samples and f being the number of features.
- **y** (*float array*) – The target array, of shape (n,).

predict(*X: Sequence[Sequence[float]]*) → Tuple[Union[float, numpy.ndarray], numpy.ndarray]
Compute model predictions.

Parameters **X** (*float matrix*) – The array containing the data set, either of shape (n,) or (n, f), with n being the number of samples and f being the number of features.

Returns The predictions, of shape (n,), and the prediction intervals, of shape (n, 2).

Return type pair of float arrays

score(*X: Sequence[float]*, *y: Sequence[float]*) → float
Compute either the R^2 value or the negative pinball loss.

If *uncertainty* is not set in the constructor then the R^2 value will be returned, and otherwise the mean of the two negative pinball losses corresponding to the two quantiles will be returned.

The pinball loss is computed as quantile * (target - prediction) if target >= prediction, and (1 - quantile)(prediction - target) otherwise.

Parameters

- **X** (*float array*) – The array containing the data set, either of shape (n,) or (n, f), with n being the number of samples and f being the number of features.
- **y** (*float array*) – The target array, of shape (n,).

Returns The negative pinball loss.

Return type float

Module contents

doubt.models.tree package

Submodules

doubt.models.tree.forest module

Quantile regression forests

class doubt.models.tree.forest.**QuantileRegressionForest**(*n_estimators: int = 100*, *criterion: str = 'mse'*, *splitter: str = 'best'*, *max_features: Optional[Union[int, float, str]] = None*, *max_depth: Optional[int] = None*, *min_samples_split: Union[int, float] = 2*, *min_samples_leaf: Union[int, float] = 5*, *min_weight_fraction_leaf: float = 0.0*, *max_leaf_nodes: Optional[int] = None*, *n_jobs: int = - 1*, *random_seed: Optional[int] = None*, *verbose: bool = False*)

Bases: doubt.models._model.BaseModel

A random forest for regression which can output quantiles as well.

Parameters

- **n_estimators** (*int, optional*) – The number of trees in the forest. Defaults to 100.
- **criterion** (*string, optional*) – The function to measure the quality of a split. Supported criteria are 'mse' for the mean squared error, which is equal to variance reduction as feature selection criterion, and 'mae' for the mean absolute error. Defaults to 'mse'.
- **splitter** (*string, optional*) – The strategy used to choose the split at each node. Supported strategies are 'best' to choose the best split and 'random' to choose the best random split. Defaults to 'best'.
- **max_features** (*int, float, string or None, optional*) – The number of features to consider when looking for the best split:
  - If int, then consider *max_features* features at each split.
  - If float, then *max_features* is a percentage and *int(max_features * n_features)* features are considered at each split.
  - If 'auto', then *max_features=n_features*.
  - If 'sqrt', then *max_features=sqrt(n_features)*.
  - If 'log2', then *max_features=log2(n_features)*.
  - If None, then *max_features=n_features*.

  Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features. Defaults to None.
- **max_depth** (*int or None, optional*) – The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Defaults to None.
- **min_samples_split** (*int or float, optional*) – The minimum number of samples required to split an internal node:
  - If int, then consider *min_samples_split* as the minimum number.
  - If float, then *min_samples_split* is a percentage and *ceil(min_samples_split * n_samples)* are the minimum number of samples for each split. Defaults to 2.
- **min_samples_leaf** (*int or float, optional*) – The minimum number of samples required to be at a leaf node:
  - If int, then consider *min_samples_leaf* as the minimum number.
  - If float, then *min_samples_leaf* is a percentage and *ceil(min_samples_leaf * n_samples)* are the minimum number of samples for each node. Defaults to 5.
- **min_weight_fraction_leaf** (*float, optional*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided. Defaults to 0.0.
- **max_leaf_nodes** (*int or None, optional*) – Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. Defaults to None.
- **n_jobs** (*int, optional*) – The number of CPU cores used in fitting and predicting. If -1 then all available CPU cores will be used. Defaults to -1.

- **random_seed** (*int, RandomState instance or None, optional*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Defaults to None.
- **verbose** (*bool, optional*) – Whether extra output should be printed during training and inference. Defaults to False.

Examples

Fitting and predicting follows scikit-learn syntax:

```
>>> from doubt.datasets import Concrete
>>> X, y = Concrete().split()
>>> forest = QuantileRegressionForest(random_seed=42,
...                                   max_leaf_nodes=8)
>>> forest.fit(X, y).predict(X).shape
(1030,)
>>> preds = forest.predict(np.ones(8))
>>> 16 < preds < 17
True
```

Instead of only returning the prediction, we can also return a prediction interval:

```
>>> preds, interval = forest.predict(np.ones(8), uncertainty=0.25)
>>> interval[0] < preds < interval[1]
True
```

fit(*X*, *y*, *verbose: Optional[bool] = None*)

Fit decision trees in parallel.

Parameters

- **X** (*array-like or sparse matrix*) – The input samples, of shape [n_samples, n_features]. Internally, it will be converted to *dtype=np.float32* and if a sparse matrix is provided to a sparse *csr_matrix*.
- **y** (*array-like*) – The target values (class labels) as integers or strings, of shape [n_samples] or [n_samples, n_outputs].
- **verbose** (*bool or None, optional*) – Whether extra output should be printed during training. If None then the initialised value of the *verbose* parameter will be used. Defaults to None.

predict(*X: Sequence[Union[float, int]]*, *uncertainty: Optional[float] = None*, *quantiles: Optional[Sequence[float]] = None*, *verbose: Optional[bool] = None*) → Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]

Predict regression value for X.

Parameters

- **X** (*array-like or sparse matrix*) – The input samples, of shape [n_samples, n_features]. Internally, it will be converted to *dtype=np.float32* and if a sparse matrix is provided to a sparse *csr_matrix*.
- **uncertainty** (*float or None, optional*) – Value ranging from 0 to 1. If None then no prediction intervals will be returned. Defaults to None.

- **quantiles** (*sequence of floats or None, optional*) – List of quantiles to output, as an alternative to the *uncertainty* argument, and will not be used if that argument is set. If None then *uncertainty* is used. Defaults to None.
- **verbose** (*bool or None, optional*) – Whether extra output should be printed during inference. If None then the initialised value of the *verbose* parameter will be used. Defaults to None.

Returns Either array with predictions, of shape [n_samples,], or a pair of arrays with the first one being the predictions and the second one being the desired quantiles/intervals, of shape [2, n_samples] if *uncertainty* is not None, and [n_quantiles, n_samples] if *quantiles* is not None.

Return type Array or pair of arrays

doubt.models.tree.tree module

Quantile regression trees

class doubt.models.tree.tree.**BaseTreeQuantileRegressor**(*, *criterion*, *splitter*, *max_depth*, *min_samples_split*, *min_samples_leaf*, *min_weight_fraction_leaf*, *max_features*, *max_leaf_nodes*, *random_state*, *min_impurity_decrease*, *min_impurity_split*, *class_weight=None*, *ccp_alpha=0.0*)

Bases: sklearn.tree._classes.BaseDecisionTree

fit(*X: Sequence[Union[float, int]]*, *y: Sequence[Union[float, int]]*, *sample_weight: Optional[Sequence[Union[float, int]]] = None*, *check_input: bool = True*, *X_idx_sorted: Optional[Sequence[Union[float, int]]] = None*)

Build a decision tree classifier from the training set (X, y).

Parameters

- **X** (*array-like or sparse matrix*) – The training input samples, of shape [n_samples, n_features]. Internally, it will be converted to *dtype=np.float32* and if a sparse matrix is provided to a sparse *csc_matrix*.
- **y** (*array-like*) – The target values (class labels) as integers or strings, of shape [n_samples] or [n_samples, n_outputs].
- **sample_weight** (*array-like or None, optional*) – Sample weights of shape = [n_samples]. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node. Defaults to None.
- **check_input** (*boolean, optional*) – Allow to bypass several input checking. Don't use this parameter unless you know what you do. Defaults to True.
- **X_idx_sorted** (*array-like or None, optional*) – The indexes of the sorted training input samples, of shape [n_samples, n_features]. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do. Defaults to None.

predict(*X: Sequence[Union[float, int]]*, *uncertainty: Optional[float] = None*, *quantiles: Optional[Sequence[float]] = None*, *check_input: bool = True*) → Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]

Predict regression value for X.

Parameters

- **X** (*array-like or sparse matrix*) – The input samples, of shape [n_samples, n_features]. Internally, it will be converted to *dtype=np.float32* and if a sparse matrix is provided to a sparse *csr_matrix*.
- **uncertainty** (*float or None, optional*) – Value ranging from 0 to 1. If None then no prediction intervals will be returned. Defaults to None.
- **quantiles** (*sequence of floats or None, optional*) – List of quantiles to output, as an alternative to the *uncertainty* argument, and will not be used if that argument is set. If None then *uncertainty* is used. Defaults to None.
- **check_input** (*boolean, optional*) – Allow to bypass several input checking. Don't use this parameter unless you know what you do. Defaults to True.

Returns Either array with predictions, of shape [n_samples,], or a pair of arrays with the first one being the predictions and the second one being the desired quantiles/intervals, of shape [n_samples, 2] if *uncertainty* is not None, and [n_samples, n_quantiles] if *quantiles* is not None.

Return type Array or pair of arrays

class doubt.models.tree.tree.**QuantileRegressionTree**(*criterion: str = 'mse'*, *splitter: str = 'best'*, *max_features: Optional[Union[int, float, str]] = None*, *max_depth: Optional[int] = None*, *min_samples_split: Union[int, float] = 2*, *min_samples_leaf: Union[int, float] = 1*, *min_weight_fraction_leaf: float = 0.0*, *max_leaf_nodes: Optional[int] = None*, *random_seed: Optional[Union[int, numpy.random.mtrand.RandomState]] = None*)

Bases: sklearn.tree._classes.DecisionTreeRegressor, *doubt.models.tree.tree.BaseTreeQuantileRegressor*

A decision tree regressor that provides quantile estimates.

Parameters

- **criterion** (*string, optional*) – The function to measure the quality of a split. Supported criteria are 'mse' for the mean squared error, which is equal to variance reduction as feature selection criterion, and 'mae' for the mean absolute error. Defaults to 'mse'.
- **splitter** (*string, optional*) – The strategy used to choose the split at each node. Supported strategies are 'best' to choose the best split and 'random' to choose the best random split. Defaults to 'best'.
- **max_features** (*int, float, string or None, optional*) – The number of features to consider when looking for the best split: - If int, then consider *max_features* features at each split. - If float, then *max_features* is a percentage and

  *int(max_features * n_features)* features are considered at each split.

  - If 'auto', then *max_features=n_features*.
  - If 'sqrt', then *max_features=sqrt(n_features)*.
  - If 'log2', then *max_features=log2(n_features)*.
  - If None, then *max_features=n_features*.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than *max_features* features. Defaults to None.

- **max_depth** (*int or None, optional*) – The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Defaults to None.
- **min_samples_split** (*int or float, optional*) – The minimum number of samples required to split an internal node: - If int, then consider *min_samples_split* as the minimum number. - If float, then *min_samples_split* is a percentage and

  *ceil(min_samples_split * n_samples)* are the minimum number of samples for each split. Defaults to 2.
- **min_samples_leaf** (*int or float, optional*) – The minimum number of samples required to be at a leaf node: - If int, then consider *min_samples_leaf* as the minimum number. - If float, then *min_samples_leaf* is a percentage and

  *ceil(min_samples_leaf * n_samples)* are the minimum number of samples for each node. Defaults to 1.
- **min_weight_fraction_leaf** (*float, optional*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided. Defaults to 0.0.
- **max_leaf_nodes** (*int or None, optional*) – Grow a tree with *max_leaf_nodes* in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. Defaults to None.
- **random_seed** (*int, RandomState instance or None, optional*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Defaults to None.

feature_importances_
The feature importances, of shape = [n_features]. The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Type array

max_features_
The inferred value of max_features.

Type int

n_features_
The number of features when *fit* is performed.

Type int

n_outputs_
The number of outputs when *fit* is performed.

Type int

tree_
The underlying Tree object.

Type Tree object

y_train_
Train target values.

Type array-like

y_train_leaves_
Cache the leaf nodes that each training sample falls into. y_train_leaves_[i] is the leaf that y_train[i] ends up at.

Type array-like

doubt.models.tree.utils module

Utility functions used in tree models

doubt.models.tree.utils.**weighted_percentile**(*arr: Sequence[Union[float, int]]*, *quantile: float*, *weights: Optional[Sequence[Union[float, int]]] = None*, *sorter: Optional[Sequence[Union[float, int]]] = None*)

Returns the weighted percentile of an array.

See [1] for an explanation of this concept.

Parameters

- **arr** (*array-like*) – Samples at which the quantile should be computed, of shape [n_samples,].
- **quantile** (*float*) – Quantile, between 0.0 and 1.0.
- **weights** (*array-like, optional*) – The weights, of shape = (n_samples,). Here weights[i] is the weight given to point a[i] while computing the quantile. If weights[i] is zero, a[i] is simply ignored during the percentile computation. If None then uniform weights will be used. Defaults to None.
- **sorter** (*array-like, optional*) – Array of shape [n_samples,], indicating the indices sorting *arr*. Thus, if provided, we assume that arr[sorter] is sorted. If None then *arr* will be sorted. Defaults to None.

Returns

float Weighted percentile of *arr* at *quantile*.

Return type percentile

Raises **ValueError** – If *quantile* is not between 0.0 and 1.0, or if *arr* and *weights* are of different lengths.

Sources: [1]: https://en.wikipedia.org/wiki/Percentile#The_weighted_percentile_method

Module contents

Module contents

1.1.2 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

D

F

G

M

N

P

Q

S

T

W

Y