pandas-ml Documentation

Release 0.3.0

sinhrks

October 01, 2016

Contents

Contents:

CHAPTER 1

What's new

1.1 v0.4.0

1.1.1 Enhancement

- Support scikit-learn v0.17.x.
- Support imbalanced-learn via `.imbalance` accessor.

1.1.2 Bug Fix

- `ModelFrame.columns` may not be preserved via `.transform` using `FunctionTransformer`, `KernelCenterer`, `MaxAbsScaler` and `RobustScaler`.

1.2 v0.3.1

1.2.1 Enhancement

- `inverse_transform` now reverts original `ModelFrame.columns` information.

1.2.2 Bug Fix

- Assigning `Series` to `ModelFrame.data` property raises `TypeError`

1.3 v0.3.0

1.3.1 Enhancement

- Support `xgboost` via `ModelFrame.xgboost` accessor.

1.4 v0.2.0

1.4.1 Enhancement

- `ModelFrame.transform` can preserve column names for some `sklearn.preprocessing` transformation.
- Added `ModelSeries.fit`, `transform`, `fit_transform` and `inverse_transform` for preprocessing purpose.
- `ModelFrame` can be initialized from `statsmodels` datasets.
- `ModelFrame.cross_validation.iterate` and `ModelFrame.cross_validation.train_test_split` now keep index of original dataset, and added `reset_index` keyword to control this behaviour.

1.4.2 Bug Fix

- `target` kw may be ignored when initializing `ModelFrame` with `np.ndarray` and `columns` kwds.
- `linear_model.enet_path` doesn't accept additional keywords.
- Initializing `ModelFrame` with named `Series` may have duplicated target columns.
- `ModelFrame.target_name` may not be preserved when sliced.

1.5 v0.1.1

1.5.1 Enhancement

- Added `sklearn.learning_curve`, `neural_network`, `random_projection`

1.6 v0.1.0

- Initial Release

CHAPTER 2

Data Handling

2.1 Data Preparation

This section describes how to prepare basic data format named `ModelFrame`. `ModelFrame` defines a metadata to specify target (response variable) and data (explanatory variable / features). Using these metadata, `ModelFrame` can call other statistics/ML functions in more simple way.

You can create `ModelFrame` as the same manner as `pandas.DataFrame`. The below example shows how to create basic `ModelFrame`, which DOESN'T have target values.

```
>>> import pandas_ml as pdml

>>> df = pdml.ModelFrame({'A': [1, 2, 3], 'B': [2, 3, 4],
...                       'C': [3, 4, 5]}, index=['a', 'b', 'c'])
>>> df
   A  B  C
a  1  2  3
b  2  3  4
c  3  4  5

>>> type(df)
<class 'pandas_ml.core.frame.ModelFrame'>
```

You can check whether the created `ModelFrame` has target values using `ModelFrame.has_target()` method.

```
>>> df.has_target()
False
```

Target values can be specifyied via `target` keyword. You can simply pass a column name to be handled as target. Target column name can be confirmed via `target_name` property.

```
>>> df2 = pdml.ModelFrame({'A': [1, 2, 3], 'B': [2, 3, 4],
...                        'C': [3, 4, 5]}, target='A')
>>> df2
   A  B  C
0  1  2  3
1  2  3  4
2  3  4  5

>>> df2.has_target()
True

>>> df2.target_name
'A'
```

Also, you can pass any list-likes to be handled as a target. In this case, target column will be named as ".target".

```
>>> df3 = pdml.ModelFrame({'A': [1, 2, 3], 'B': [2, 3, 4],
...                        'C': [3, 4, 5]}, target=[4, 5, 6])
>>> df3
   .target  A  B  C
0        4  1  2  3
1        5  2  3  4
2        6  3  4  5

>>> df3.has_target()
True

>>> df3.target_name
'.target'
```

Also, you can pass `pandas.DataFrame` and `pandas.Series` as data and target.

```
>>> import pandas as pd
df4 = pdml.ModelFrame({'A': [1, 2, 3], 'B': [2, 3, 4],
...                    'C': [3, 4, 5]}, target=pd.Series([4, 5, 6]))
>>> df4
   .target  A  B  C
0        4  1  2  3
1        5  2  3  4
2        6  3  4  5

>>> df4.has_target()
True

>>> df4.target_name
'.target'
```

Note: Target values are mandatory to perform operations which require response variable, such as regression and supervised learning.

2.2 Data Manipulation

You can maniluplate `ModelFrame` like `pandas.DataFrame`. Because `ModelFrame` inherits `pandas.DataFrame`, all the `pandas` methods / functions can be applied to `ModelFrame`.

Sliced results will be `ModelSeries` (simple wrapper for `pandas.Series` to support some data manipulation) or `ModelFrame`

```
>>> df
   A  B  C
a  1  2  3
b  2  3  4
c  3  4  5

>>> sliced = df['A']
>>> sliced
a    1
b    2
c    3
```

```
Name: A, dtype: int64

>>> type(sliced)
<class 'pandas_ml.core.series.ModelSeries'>

>>> subset = df[['A', 'B']]
>>> subset
   A  B
a  1  2
b  2  3
c  3  4

>>> type(subset)
<class 'pandas_ml.core.frame.ModelFrame'>
```

`ModelFrame` has a special properties `data` to access data (features) and `target` to access target.

```
>>> df2
   A  B  C
0  1  2  3
1  2  3  4
2  3  4  5

>>> df2.target_name
'A'

>>> df2.data
   B  C
0  2  3
1  3  4
2  4  5

>>> df2.target
0    1
1    2
2    3
Name: A, dtype: int64
```

You can update data and target via properties. Also, columns / value assignment are supported as the same as `pandas.DataFrame`.

```
>>> df2.target = [9, 9, 9]
>>> df2
   A  B  C
0  9  2  3
1  9  3  4
2  9  4  5

>>> df2.data = pd.DataFrame({'X': [1, 2, 3], 'Y': [4, 5, 6]})
>>> df2
   A  X  Y
0  9  1  4
1  9  2  5
2  9  3  6

>>> df2['X'] = [0, 0, 0]
>>> df2
   A  X  Y
0  9  0  4
```

```
1  9  0  5
2  9  0  6
```

You can change target column specifying `target_name` property.

```
>>> df2.target_name
'A'

>>> df2.target_name = 'X'
>>> df2.target_name
'X'
```

If the specified column doesn't exist in `ModelFrame`, it should reset target to `None`. Current target will be regarded as data.

```
>>> df2.target_name
'X'

>>> df2.target_name = 'XXXX'
>>> df2.has_target()
False

>>> df2.data
   A  X  Y
0  9  0  4
1  9  0  5
2  9  0  6
```

CHAPTER 3

Use scikit-learn

This section describes how to use `scikit-learn` functionalities via `pandas-ml`.

3.1 Basics

You can create `ModelFrame` instance from `scikit-learn` datasets directly.

```
>>> import pandas_ml as pdml
>>> import sklearn.datasets as datasets

>>> df = pdml.ModelFrame(datasets.load_iris())
>>> df.head()
   .target  sepal length (cm)  sepal width (cm)  petal length (cm)  \
0        0                5.1               3.5                1.4
1        0                4.9               3.0                1.4
2        0                4.7               3.2                1.3
3        0                4.6               3.1                1.5
4        0                5.0               3.6                1.4

   petal width (cm)
0               0.2
1               0.2
2               0.2
3               0.2
4               0.2

# make columns be readable
>>> df.columns = ['.target', 'sepal length', 'sepal width', 'petal length', 'petal width']
```

`ModelFrame` has accessor methods which makes easier access to `scikit-learn` namespace.

```
>>> df.cluster.KMeans
<class 'sklearn.cluster.k_means_.KMeans'>
```

Following table shows `scikit-learn` module and corresponding `ModelFrame` module. Some accessors has its abbreviated versions.

scikit-learn	ModelFrame accessor
sklearn.cluster	ModelFrame.cluster
sklearn.covariance	ModelFrame.covariance

Continued on next page

Table 3.1 – continued from previous page

scikit-learn	ModelFrame accessor
sklearn.cross_decomposition	ModelFrame.cross_decomposition
sklearn.cross_validation	ModelFrame.cross_validation, crv
sklearn.datasets	(not accesible from accessor)
sklearn.decomposition	ModelFrame.decomposition
sklearn.dummy	ModelFrame.dummy
sklearn.ensemble	ModelFrame.ensemble
sklearn.feature_extraction	ModelFrame.feature_extraction
sklearn.feature_selection	ModelFrame.feature_selection
sklearn.gaussian_process	ModelFrame.gaussian_process
sklearn.grid_search	ModelFrame.grid_search
sklearn.isotonic	ModelFrame.isotonic
sklearn.kernel_approximation	ModelFrame.kernel_approximation
sklearn.lda	ModelFrame.lda
sklearn.learning_curve	ModelFrame.learning_curve
sklearn.linear_model	ModelFrame.linear_model, lm
sklearn.manifold	ModelFrame.manifold
sklearn.metrics	ModelFrame.metrics
sklearn.mixture	ModelFrame.mixture
sklearn.multiclass	ModelFrame.multiclass
sklearn.naive_bayes	ModelFrame.naive_bayes
sklearn.neighbors	ModelFrame.neighbors
sklearn.neural_network	ModelFrame.neural_network
sklearn.pipeline	ModelFrame.pipeline
sklearn.preprocessing	ModelFrame.preprocessing, pp
sklearn.qda	ModelFrame.qda
sklearn.semi_supervised	ModelFrame.semi_supervised
sklearn.svm	ModelFrame.svm
sklearn.tree	ModelFrame.tree
sklearn.utils	(not accesible from accessor)

Thus, you can instanciate each estimator via `ModelFrame` accessors. Once create an estimator, you can pass it to `ModelFrame.fit` then `predict`. `ModelFrame` automatically uses its data and target properties for each operations.

```
>>> estimator = df.cluster.KMeans(n_clusters=3)
>>> df.fit(estimator)

>>> predicted = df.predict(estimator)
>>> predicted
0    1
1    1
2    1
...
147    2
148    2
149    0
Length: 150, dtype: int32
```

`ModelFrame` preserves the most recently used estimator in `estimator` atribute, and predicted results in `predicted` attibute.

```
>>> df.estimator
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3, n_init=10,
    n_jobs=1, precompute_distances=True, random_state=None, tol=0.0001,
    verbose=0)

>>> df.predicted
0    1
1    1
2    1
...
147    2
148    2
149    0
Length: 150, dtype: int32
```

`ModelFrame` has following methods corresponding to various `scikit-learn` estimators. The last results are saved as corresponding `ModelFrame` properties.

`ModelFrame` method	`ModelFrame` property
`ModelFrame.fit`	(None)
`ModelFrame.transform`	(None)
`ModelFrame.fit_transform`	(None)
`ModelFrame.inverse_transform`	(None)
`ModelFrame.predict`	`ModelFrame.predicted`
`ModelFrame.fit_predict`	`ModelFrame.predicted`
`ModelFrame.score`	(None)
`ModelFrame.predict_proba`	`ModelFrame.proba`
`ModelFrame.predict_log_proba`	`ModelFrame.log_proba`
`ModelFrame.decision_function`	`ModelFrame.decision`

Note: If you access to a property before calling `ModelFrame` methods, `ModelFrame` automatically calls corresponding method of the latest estimator and return the result.

Following example shows to perform PCA, then revert principal components back to original space. `inverse_transform` should revert the original columns.

```
>>> estimator = df.decomposition.PCA()
>>> df.fit(estimator)

>>> transformed = df.transform(estimator)
>>> transformed.head()
   .target         0         1         2         3
0        0 -2.684207 -0.326607  0.021512  0.001006
1        0 -2.715391  0.169557  0.203521  0.099602
2        0 -2.889820  0.137346 -0.024709  0.019305
3        0 -2.746437  0.311124 -0.037672 -0.075955
4        0 -2.728593 -0.333925 -0.096230 -0.063129

>>> type(transformed)
<class 'pandas_ml.core.frame.ModelFrame'>

>>> transformed.inverse_transform(estimator)
     .target  sepal length  sepal width  petal length  petal width
0          0           5.1          3.5           1.4          0.2
1          0           4.9          3.0           1.4          0.2
2          0           4.7          3.2           1.3          0.2
```

```
3           0            4.6           3.1            1.5           0.2
4           0            5.0           3.6            1.4           0.2
..        ...            ...           ...            ...           ...
145         2            6.7           3.0            5.2           2.3
146         2            6.3           2.5            5.0           1.9
147         2            6.5           3.0            5.2           2.0
148         2            6.2           3.4            5.4           2.3
149         2            5.9           3.0            5.1           1.8

[150 rows x 5 columns]
```

If `ModelFrame` both has `target` and `predicted` values, the model evaluation can be performed using functions available in `ModelFrame.metrics`.

```
>>> estimator = df.svm.SVC()
>>> df.fit(estimator)

>>> df.predict(estimator)
0    0
1    0
2    0
...
147    2
148    2
149    2
Length: 150, dtype: int64

>>> df.predicted
0    0
1    0
2    0
...
147    2
148    2
149    2
Length: 150, dtype: int64

>>> df.metrics.confusion_matrix()
Predicted   0   1   2
Target
0          50   0   0
1           0  48   2
2           0   0  50
```

3.2 Use Module Level Functions

Some `scikit-learn` modules define functions which handle data without instanciating estimators. You can call these functions from accessor methods directly, and `ModelFrame` will pass corresponding data on background. Following example shows to use `sklearn.cluster.k_means` function to perform K-means.

Important: When you use module level function, `ModelFrame.predicted` WILL NOT be updated. Thus, using estimator is recommended.

```
# no need to pass data explicitly
# sklearn.cluster.kmeans returns centroids, cluster labels and inertia
>>> c, l, i = df.cluster.k_means(n_clusters=3)
>>> l
0     1
1     1
2     1
...
147    2
148    2
149    0
Length: 150, dtype: int32
```

3.3 Pipeline

`ModelFrame` can handle pipeline as the same as normal estimators.

```
>>> estimators = [('reduce_dim', df.decomposition.PCA()),
...               ('svm', df.svm.SVC())]
>>> pipe = df.pipeline.Pipeline(estimators)
>>> df.fit(pipe)

>>> df.predict(pipe)
0    0
1    0
2    0
...
147    2
148    2
149    2
Length: 150, dtype: int64
```

Above expression is the same as below:

```
>>> df2 = df.copy()
>>> df2 = df2.fit_transform(df2.decomposition.PCA())
>>> svm = df2.svm.SVC()
>>> df2.fit(svm)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.0,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
>>> df2.predict(svm)
0     0
1     0
2     0
...
147    2
148    2
149    2
Length: 150, dtype: int64
```

3.4 Cross Validation

`scikit-learn` has some classes for cross validation. `cross_validation.train_test_split` splits data to training and test set. You can access to the function via `cross_validation` accessor.

```
>>> train_df, test_df = df.cross_validation.train_test_split()
>>> train_df
     .target  sepal length  sepal width  petal length  petal width
124        2           6.7          3.3           5.7          2.1
117        2           7.7          3.8           6.7          2.2
123        2           6.3          2.7           4.9          1.8
65         1           6.7          3.1           4.4          1.4
133        2           6.3          2.8           5.1          1.5
..       ...           ...          ...           ...          ...
93         1           5.0          2.3           3.3          1.0
46         0           5.1          3.8           1.6          0.2
121        2           5.6          2.8           4.9          2.0
91         1           6.1          3.0           4.6          1.4
147        2           6.5          3.0           5.2          2.0

[112 rows x 5 columns]


>>> test_df
     .target  sepal length  sepal width  petal length  petal width
146        2           6.3          2.5           5.0          1.9
75         1           6.6          3.0           4.4          1.4
138        2           6.0          3.0           4.8          1.8
77         1           6.7          3.0           5.0          1.7
36         0           5.5          3.5           1.3          0.2
..       ...           ...          ...           ...          ...
14         0           5.8          4.0           1.2          0.2
141        2           6.9          3.1           5.1          2.3
100        2           6.3          3.3           6.0          2.5
83         1           6.0          2.7           5.1          1.6
114        2           5.8          2.8           5.1          2.4

[38 rows x 5 columns]
```

Also, there are some iterative classes which returns indexes for training sets and test sets. You can slice `ModelFrame` using these indexes.

```
>>> kf = df.cross_validation.KFold(n=150, n_folds=3)
>>> for train_index, test_index in kf:
...     print('training set shape: ', df.iloc[train_index, :].shape,
...           'test set shape: ', df.iloc[test_index, :].shape)
('training set shape: ', (100, 5), 'test set shape: ', (50, 5))
('training set shape: ', (100, 5), 'test set shape: ', (50, 5))
('training set shape: ', (100, 5), 'test set shape: ', (50, 5))
```

For further simplification, `ModelFrame.cross_validation.iterate` can accept such iterators and returns `ModelFrame` corresponding to training and test data.

```
>>> kf = df.cross_validation.KFold(n=150, n_folds=3)
>>> for train_df, test_df in df.cross_validation.iterate(kf):
...     print('training set shape: ', train_df.shape,
...           'test set shape: ', test_df.shape)
('training set shape: ', (100, 5), 'test set shape: ', (50, 5))
```

```
('training set shape: ', (100, 5), 'test set shape: ', (50, 5))
('training set shape: ', (100, 5), 'test set shape: ', (50, 5))
```

3.5 Grid Search

You can perform grid search using `ModelFrame.fit`.

```
>>> tuned_parameters = [{'kernel': ['rbf'], 'gamma': [1e-3, 1e-4],
...                      'C': [1, 10, 100]},
...                     {'kernel': ['linear'], 'C': [1, 10, 100]}]

>>> df = pdml.ModelFrame(datasets.load_digits())
>>> cv = df.grid_search.GridSearchCV(df.svm.SVC(C=1), tuned_parameters,
...                                  cv=5, scoring='precision')

>>> df.fit(cv)

>>> cv.best_estimator_
SVC(C=10, cache_size=200, class_weight=None, coef0=0.0, degree=3, gamma=0.001,
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

In addition, `ModelFrame.grid_search` has a `describe` function to organize each grid search result as `ModelFrame` accepting estimator.

```
>>> df.grid_search.describe(cv)
       mean       std    C   gamma  kernel
0  0.974108  0.013139    1  0.0010     rbf
1  0.951416  0.020010    1  0.0001     rbf
2  0.975372  0.011280   10  0.0010     rbf
3  0.962534  0.020218   10  0.0001     rbf
4  0.975372  0.011280  100  0.0010     rbf
5  0.964695  0.016686  100  0.0001     rbf
6  0.951811  0.018410    1     NaN  linear
7  0.951811  0.018410   10     NaN  linear
8  0.951811  0.018410  100     NaN  linear
```

CHAPTER 4

Handling imbalanced data

This section describes how to use imbalanced-learn functionalities via `pandas-ml` to handle imbalanced data.

4.1 Sampling

Assuming we have `ModelFrame` which has imbalanced target values. The `ModelFrame` has data with 80 observations labeld with `0` and 20 observations labeled with `1`.

```
>>> import numpy as np
>>> import pandas_ml as pdml
>>> df = pdml.ModelFrame(np.random.randn(100, 5),
...                      target=np.array([0, 1]).repeat([80, 20]),
...                      columns=list('ABCDE'))
>>> df
    .target         A         B         C         D         E
0         0  1.467859  1.637449  0.175770  0.189108  0.775139
1         0 -1.706293 -0.598930 -0.343427  0.355235 -1.348378
2         0  0.030542  0.393779 -1.891991  0.041062  0.055530
3         0  0.320321 -1.062963 -0.416418 -0.629776  1.126027
..      ...       ...       ...       ...       ...       ...
96        1 -1.199039  0.055702  0.675555 -0.416601 -1.676259
97        1 -1.264182 -0.167390 -0.939794 -0.638733 -0.806794
98        1 -0.616754  1.667483 -1.858449 -0.259630  1.236777
99        1 -1.374068 -0.400435 -1.825555  0.824052 -0.335694

[100 rows x 6 columns]

>>> df.target.value_counts()
0    80
1    20
Name: .target, dtype: int64
```

You can access `imbalanced-learn` namespace via `.imbalance` accessor. Passing instanciated under-sampling class to `ModelFrame.fit_sample` returns under sampled `ModelFrame` (Note that `.index` is reset).

```
>>> sampler = df.imbalance.under_sampling.ClusterCentroids()
>>> sampler
ClusterCentroids(n_jobs=-1, random_state=None, ratio='auto')

>>> sampled = df.fit_sample(sampler)
>>> sampled
    .target         A         B         C         D         E
```

```
0         1  0.232841 -1.364282  1.436854  0.563796 -0.372866
1         1 -0.159551  0.473617 -2.024209  0.760444 -0.820403
2         1  1.495356 -2.144495  0.076485  1.219948  0.382995
3         1 -0.736887  1.399623  0.557098  0.621909 -0.507285
..      ...       ...       ...       ...       ...       ...
36        0  0.429978 -1.421307  0.771368  1.704277  0.645590
37        0  1.408448  0.132760 -1.082301 -1.195149  0.155057
38        0  0.362793 -0.682171  1.026482  0.663343 -2.371229
39        0 -0.796293 -0.196428 -0.747574  2.228031 -0.468669

[40 rows x 6 columns]

>>> sampled.target.value_counts()
1    20
0    20
Name: .target, dtype: int64
```

As the same manner, you can perform over-sampling.

```
>>> sampler = df.imbalance.over_sampling.SMOTE()
>>> sampler
SMOTE(k=5, kind='regular', m=10, n_jobs=-1, out_step=0.5, random_state=None,
ratio='auto')

>>> sampled = df.fit_sample(sampler)
>>> sampled
     .target         A         B         C         D         E
0          0  1.467859  1.637449  0.175770  0.189108  0.775139
1          0 -1.706293 -0.598930 -0.343427  0.355235 -1.348378
2          0  0.030542  0.393779 -1.891991  0.041062  0.055530
3          0  0.320321 -1.062963 -0.416418 -0.629776  1.126027
..       ...       ...       ...       ...       ...       ...
156        1 -1.279399  0.218171 -0.487836 -0.573564  0.582580
157        1 -0.736964  0.239095 -0.422025 -0.841780  0.221591
158        1 -0.273911 -0.305608 -0.886088  0.062414 -0.001241
159        1  0.073145 -0.167884 -0.781611 -0.016734 -0.045330

[160 rows x 6 columns]'

>>> sampled.target.value_counts()
1    80
0    80
Name: .target, dtype: int64
```

Following table shows `imbalanced-learn` module and corresponding `ModelFrame` module.

`imbalanced-learn`	`ModelFrame` accessor
`imblearn.under_sampling`	`ModelFrame.imbalance.under_sampling`
`imblearn.over_sampling`	`ModelFrame.imbalance.over_sampling`
`imblearn.combine`	`ModelFrame.imbalance.combine`
`imblearn.ensemble`	`ModelFrame.imbalance.ensemble`

CHAPTER 5

Use XGBoost

This section describes how to use `XGBoost` functionalities via `pandas-ml`.

Use `scikit-learn` digits dataset as sample data.

```
>>> import pandas_ml as pdml
>>> import sklearn.datasets as datasets

>>> df = pdml.ModelFrame(datasets.load_digits())
>>> df.head()
   .target  0  1  2 ...  60  61  62  63
0        0  0  0  5 ...  10   0   0   0
1        1  0  0  0 ...  16  10   0   0
2        2  0  0  0 ...  11  16   9   0
3        3  0  0  7 ...  13   9   0   0
4        4  0  0  0 ...  16   4   0   0

[5 rows x 65 columns]
```

As an estimator, `XGBClassifier` and `XGBRegressor` are available via `xgboost` accessor. See XGBoost Scikit-learn API for details.

```
>>> df.xgboost.XGBClassifier
<class 'xgboost.sklearn.XGBClassifier'>

>>> df.xgboost.XGBRegressor
<class 'xgboost.sklearn.XGBRegressor'>
```

You can use these estimators like `scikit-learn` estimators.

```
>>> train_df, test_df = df.cross_validation.train_test_split()

>>> estimator = df.xgboost.XGBClassifier()

>>> train_df.fit(estimator)
XGBClassifier(base_score=0.5, colsample_bytree=1, gamma=0, learning_rate=0.1,
       max_delta_step=0, max_depth=3, min_child_weight=1, missing=None,
       n_estimators=100, nthread=-1, objective='multi:softprob', seed=0,
       silent=True, subsample=1)

>>> predicted = test_df.predict(estimator)

>>> predicted
1371    2
1090    3
```

```
1299    2
...
1286    8
1632    3
538     2
dtype: int64

>>> test_df.metrics.confusion_matrix()
Predicted   0   1   2   3 ...   6   7   8   9
Target                    ...
0          53   0   0   0 ...   0   0   1   0
1           0  46   0   0 ...   0   0   0   0
2           0   0  51   1 ...   0   0   1   0
3           0   0   0  33 ...   0   0   1   0
4           0   0   0   0 ...   0   0   0   1
5           0   0   0   0 ...   1   0   0   1
6           0   0   0   0 ...  39   0   1   0
7           0   0   0   0 ...   0  40   0   1
8           1   0   0   0 ...   1   0  32   2
9           0   1   0   0 ...   0   1   1  51

[10 rows x 10 columns]
```

Also, plotting functions are available via `xgboost` accessor.

```
>>> train_df.xgboost.plot_importance()
# importance plot will be displayed
```

`XGBoost` estimators can be passed to other `scikit-learn` APIs. Following example shows to perform a grid search.

```
>>> tuned_parameters = [{'max_depth': [3, 4]}]
>>> cv = df.grid_search.GridSearchCV(df.xgb.XGBClassifier(), tuned_parameters, cv=5)

>>> df.fit(cv)
>>> df.grid_search.describe(cv)
       mean       std  max_depth
0  0.917641  0.032600          3
1  0.919310  0.026644          4
```

CHAPTER 6

Use patsy

This section describes data transformation using `patsy`. `ModelFrame.transform` can accept `patsy` style formula.

```
>>> import pandas_ml as pdml

# create modelframe which doesn't have target
>>> df = pdml.ModelFrame({'X': [1, 2, 3], 'Y': [2, 3, 4],
...                       'Z': [3, 4, 5]}, index=['a', 'b', 'c'])

>>> df
   X  Y  Z
a  1  2  3
b  2  3  4
c  3  4  5

# transform with patsy formula
>>> transformed = df.transform('Z ~ Y + X')
>>> transformed
   Z  Intercept  Y  X
a  3          1  2  1
b  4          1  3  2
c  5          1  4  3

# transformed data should have target specified by formula
>>> transformed.target
a    3
b    4
c    5
Name: Z, dtype: float64

>>> transformed.data
   Intercept  Y  X
a          1  2  1
b          1  3  2
c          1  4  3
```

If you do not want intercept, specify with `0`.

```
>>> df.transform('Z ~ Y + 0')
   Z  Y
a  3  2
b  4  3
c  5  4
```

Also, you can use formula which doesn't have left side.

```
# create modelframe which has target
>>> df2 = pdml.ModelFrame({'X': [1, 2, 3], 'Y': [2, 3, 4],'Z': [3, 4, 5]},
...                       target =[7, 8, 9], index=['a', 'b', 'c'])

>>> df2
   .target  X  Y  Z
a        7  1  2  3
b        8  2  3  4
c        9  3  4  5

# overwrite data with transformed data
>>> df2.data = df2.transform('Y + Z')
>>> df2
   .target  Intercept  Y  Z
a        7          1  2  3
b        8          1  3  4
c        9          1  4  5

# data has been updated based on formula
>>> df2.data
   Intercept  Y  Z
a          1  2  3
b          1  3  4
c          1  4  5

# target is not changed
>>> df2.target
a    7
b    8
c    9
Name: .target, dtype: int64
```

Below example is performing deviation coding via patsy formula.

```
>>> df3 = pdml.ModelFrame({'X': [1, 2, 3, 4, 5], 'Y': [1, 3, 2, 2, 1],
...                        'Z': [1, 1, 1, 2, 2]}, target='Z',
...                        index=['a', 'b', 'c', 'd', 'e'])
```

```
>>> df3
   X  Y  Z
a  1  1  1
b  2  3  1
c  3  2  1
d  4  2  2
e  5  1  2
```

```
>>> df3.transform('C(X, Sum)')
   Intercept  C(X, Sum)[S.1]  C(X, Sum)[S.2]  C(X, Sum)[S.3]  C(X, Sum)[S.4]
a          1               1               0               0               0
b          1               0               1               0               0
c          1               0               0               1               0
d          1               0               0               0               1
e          1              -1              -1              -1              -1
```

```
>>> df3.transform('C(Y, Sum)')
   Intercept  C(Y, Sum)[S.1]  C(Y, Sum)[S.2]
a          1               1               0
```

b	1	-1	-1
c	1	0	1
d	1	0	1
e	1	1	0

CHAPTER 7

Confusion matrix

Import `ConfusionMatrix`

```
from pandas_ml.confusion_matrix import ConfusionMatrix
```

Define actual values (`y_true`) and predicted values (`y_pred`)

```
y_true = ['rabbit', 'cat', 'rabbit', 'rabbit', 'cat', 'dog', 'dog', 'rabbit', 'rabbit', 'cat', 'dog',
y_pred = ['cat', 'cat', 'rabbit', 'dog', 'cat', 'rabbit', 'dog', 'cat', 'rabbit', 'cat', 'rabbit', 'r
```

Let's define a (non binary) confusion matrix

```
confusion_matrix = ConfusionMatrix(y_true, y_pred)
print("Confusion matrix:\n%s" % confusion_matrix)
```

You can see it

```
Predicted  cat  dog  rabbit  __all__
Actual
cat          3    0       0        3
dog          0    1       2        3
rabbit       2    1       3        6
__all__      5    2       5       12
```

7.1 Matplotlib plot of a confusion matrix

Inside a IPython notebook add this line as first cell

```
%matplotlib inline
```

You can plot confusion matrix using:

```
import matplotlib.pyplot as plt

confusion_matrix.plot()
```

If you are not using inline mode, you need to use to show confusion matrix plot.

```
plt.show()
```

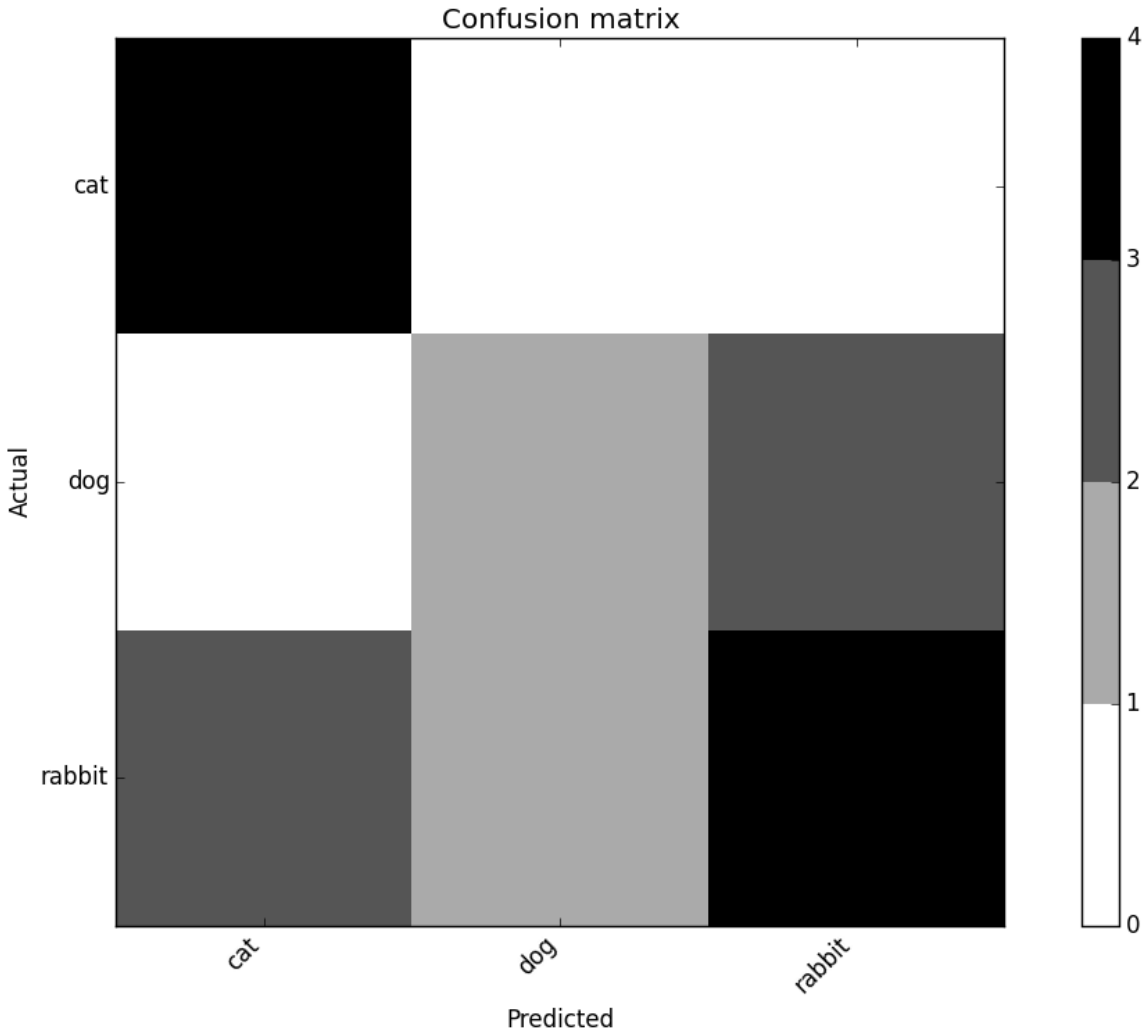

Fig. 7.1: confusion_matrix

7.2 Matplotlib plot of a normalized confusion matrix

```
confusion_matrix.plot(normalized=True)
plt.show()
```

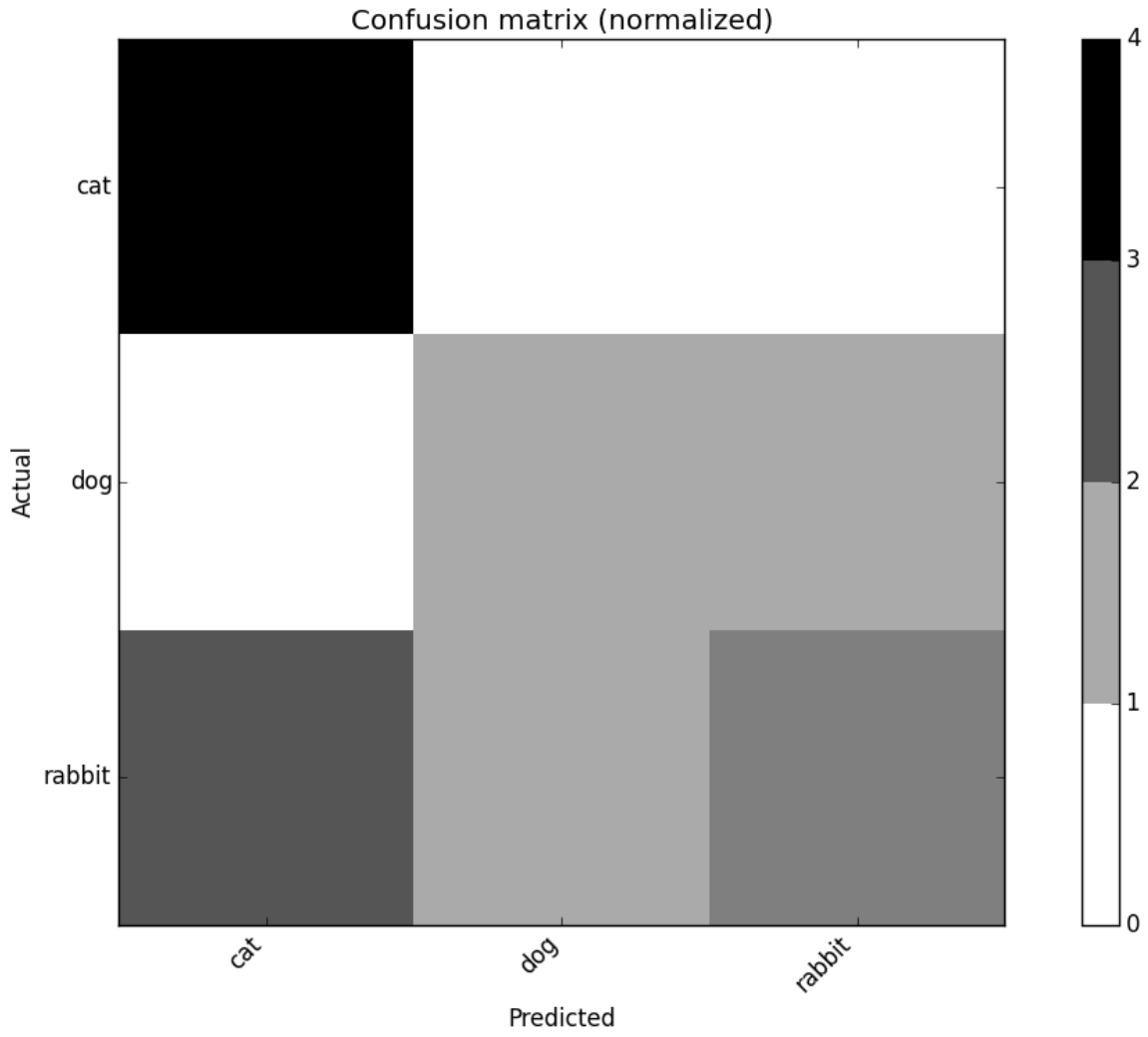

Fig. 7.2: confusion_matrix_norm

7.3 Binary confusion matrix

Import `BinaryConfusionMatrix` and `Backend`

```
from pandas_ml.confusion_matrix import BinaryConfusionMatrix, Backend
```

Define actual values (`y_true`) and predicted values (`y_pred`)

```
y_true = [ True,  True, False, False, False,  True, False,  True,  True,
           False,  True, False, False, False, False, False,  True, False,
            True,  True,  True,  True, False, False, False,  True, False,
            True, False, False, False, False,  True,  True, False, False,
           False,  True,  True,  True,  True, False, False, False, False,
            True, False, False, False, False, False, False, False, False,
           False,  True,  True, False,  True, False,  True,  True,  True,
           False, False,  True, False,  True, False, False,  True, False,
           False, False, False, False, False, False, False,  True, False,
            True,  True,  True,  True, False, False,  True, False,  True,
            True, False,  True, False,  True, False, False,  True,  True,
           False, False,  True,  True, False, False, False, False, False,
           False,  True,  True, False]

y_pred = [False, False, False, False, False,  True, False, False,  True,
          False,  True, False, False, False, False, False, False, False,
           True,  True,  True,  True, False, False, False, False, False,
          False, False, False, False, False,  True, False, False, False,
          False,  True, False, False, False, False, False, False, False,
           True, False, False, False, False, False, False, False, False,
          False,  True, False, False, False, False, False, False, False,
          False, False,  True, False, False, False, False,  True, False,
          False, False, False, False, False, False, False,  True, False,
          False,  True, False, False, False, False,  True, False,  True,
           True, False, False, False,  True, False, False,  True,  True,
          False, False,  True,  True, False, False, False, False, False,
          False,  True, False, False]
```

Let's define a binary confusion matrix

```
binary_confusion_matrix = BinaryConfusionMatrix(y_true, y_pred)
print("Binary confusion matrix:\n%s" % binary_confusion_matrix)
```

It display as a nicely labeled Pandas DataFrame

```
Binary confusion matrix:
Predicted  False  True  __all__
Actual
False         67     0       67
True          21    24       45
__all__       88    24      112
```

You can get useful attributes such as True Positive (TP), True Negative (TN) ...

```
print(binary_confusion_matrix.TP)
```

7.4 Matplotlib plot of a binary confusion matrix

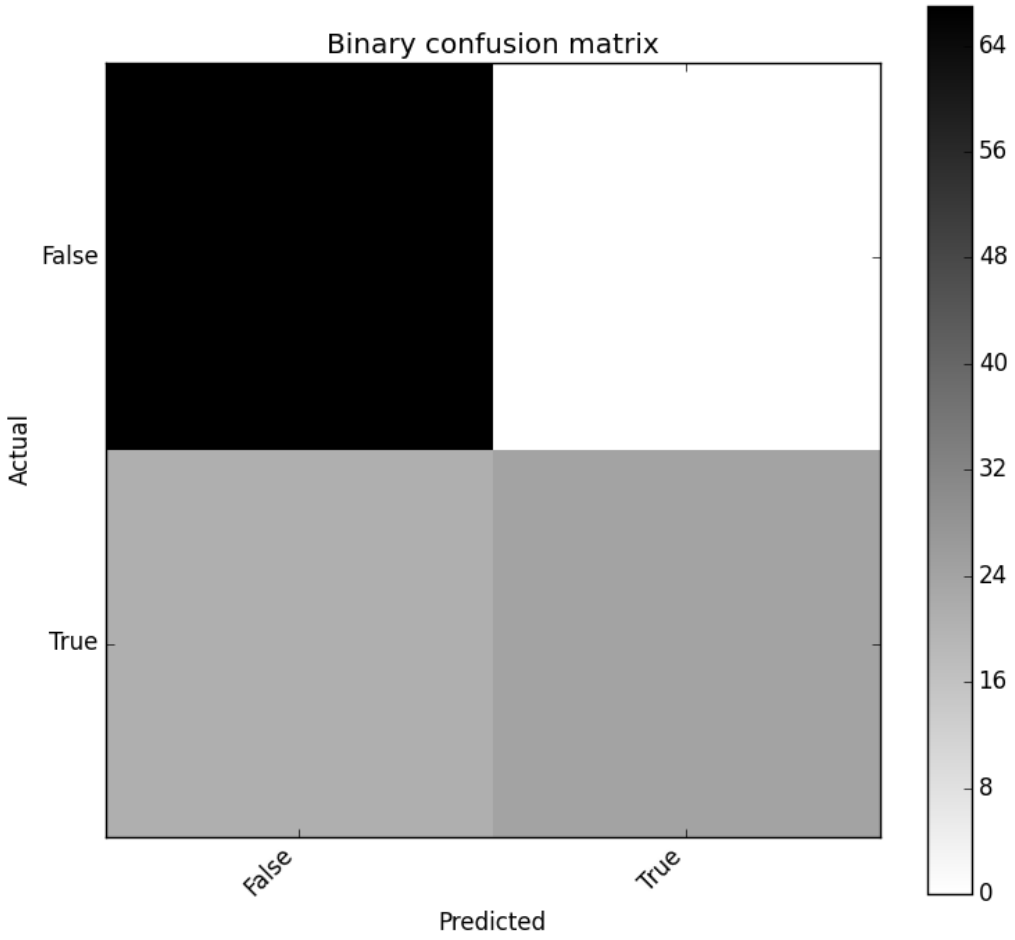

Fig. 7.3: binary_confusion_matrix

```
binary_confusion_matrix.plot()
plt.show()
```

7.5 Matplotlib plot of a normalized binary confusion matrix

```
binary_confusion_matrix.plot(normalized=True)
plt.show()
```

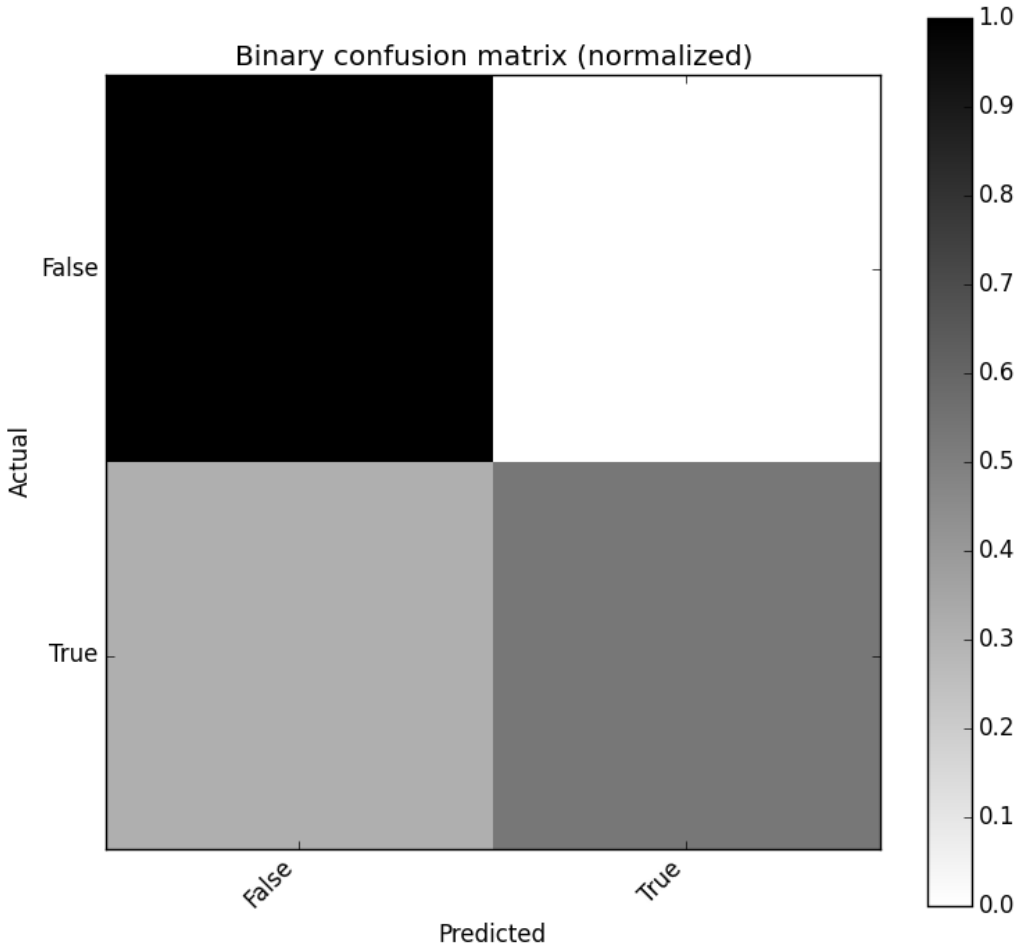

Fig. 7.4: binary_confusion_matrix_norm

7.6 Seaborn plot of a binary confusion matrix (ToDo)

```
from pandas_ml.confusion_matrix import Backend
binary_confusion_matrix.plot(backend=Backend.Seaborn)
```

7.7 Confusion matrix and class statistics

Overall statistics and class statistics of confusion matrix can be easily displayed.

```
y_true = [600, 200, 200, 200, 200, 200, 200, 200, 500, 500, 500, 200, 200, 200, 200, 200, 200, 200, 2
y_pred = [100, 200, 200, 100, 100, 200, 200, 200, 100, 200, 500, 100, 100, 100, 100, 100, 100, 100, 5
cm = ConfusionMatrix(y_true, y_pred)
cm.print_stats()
```

You should get:

```
Confusion Matrix:

Classes  100  200  500  600  __all__
Actual
100        0    0    0    0        0
200        9    6    1    0       16
500        1    1    1    0        3
600        1    0    0    0        1
__all__   11    7    2    0       20


Overall Statistics:

Accuracy: 0.35
95% CI: (0.1539092047845412, 0.59218853453282805)
No Information Rate: ToDo
P-Value [Acc > NIR]: 0.978585644357
Kappa: 0.0780141843972
Mcnemar's Test P-Value: ToDo


Class Statistics:

Classes                                 100         200         500    600
Population                               20          20          20     20
Condition positive                        0          16           3      1
Condition negative                       20           4          17     19
Test outcome positive                    11           7           2      0
Test outcome negative                     9          13          18     20
TP: True Positive                         0           6           1      0
TN: True Negative                         9           3          16     19
FP: False Positive                       11           1           1      0
FN: False Negative                        0          10           2      1
TPR: Sensivity                          NaN       0.375   0.3333333      0
TNR=SPC: Specificity                   0.45        0.75   0.9411765      1
PPV: Pos Pred Value = Precision           0   0.8571429         0.5    NaN
NPV: Neg Pred Value                       1   0.2307692   0.8888889   0.95
FPR: False-out                         0.55        0.25  0.05882353      0
FDR: False Discovery Rate                 1   0.1428571         0.5    NaN
FNR: Miss Rate                          NaN       0.625   0.6666667      1
```

```
ACC: Accuracy                          0.45        0.45        0.85  0.95
F1 score                                  0   0.5217391         0.4     0
MCC: Matthews correlation coefficient   NaN   0.1048285    0.326732   NaN
Informedness                            NaN       0.125   0.2745098     0
Markedness                                0  0.08791209   0.3888889   NaN
Prevalence                                0         0.8        0.15  0.05
LR+: Positive likelihood ratio          NaN         1.5    5.666667   NaN
LR-: Negative likelihood ratio          NaN   0.8333333   0.7083333     1
DOR: Diagnostic odds ratio              NaN         1.8           8   NaN
FOR: False omission rate                  0   0.7692308   0.1111111  0.05
```

Statistics are also available as an OrderedDict using:

```
cm.stats()
```

API:

CHAPTER 8

pandas_ml.core package

8.1 Submodules

8.2 Module contents

CHAPTER 9

pandas_ml.skaccessors package

9.1 Subpackages

9.1.1 pandas_ml.skaccessors.test package

Submodules

Module contents

9.2 Submodules

9.3 Module contents

CHAPTER 10

pandas_ml.xgboost package

10.1 Subpackages

10.1.1 pandas_ml.xgboost.test package

Submodules

Module contents

10.2 Submodules

10.3 Module contents