expandas Documentation

Release 0.2.0

sinhrks

April 04, 2015

Contents

Contents:

CHAPTER 1

What's new

1.1 v0.2.0

1.1.1 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.

1.1.2 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.

1.2 v0.1.1

1.2.1 Enhancement

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

1.3 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 expandas as expd

>>> df = expd.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 'expandas.core.frame.ModelFrame'>
```

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

```
>>> 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 = expd.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 = expd.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 = expd.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 access to each property as the same as `pandas.DataFrame`. 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 'expandas.core.series.ModelSeries'>
```

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

>>> type(subset)
<class 'expandas.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, in addition to standard `pandas.DataFrame` ways.

```
>>> 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. Specifying a column which doesn't exist in `ModelFrame` results in target column to be data column.

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

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

3.1 Basics

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

```
>>> import expandas as expd
>>> import sklearn.datasets as datasets

>>> df = expd.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.

Note: Currently, `ModelFrame` can handle target which consists from a single column. Modules which uses multiple target columns cannot be handled automatically, and marked with *(WIP)*.

---

scikit-learn	ModelFrame accessor
sklearn.cluster	ModelFrame.cluster
sklearn.covariance	ModelFrame.covariance
sklearn.cross_decomposition	ModelFrame.cross_decomposition (WIP)
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 (WIP)
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.

```
>>> 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 'expandas.core.frame.ModelFrame'>

>>> transformed.inverse_transform(estimator)
     .target    0    1    2    3
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]
```

Note: `columns` information will be lost once transformed to principal components.

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
0          0           4.8          3.4           1.9          0.2
1          1           6.3          3.3           4.7          1.6
2          0           4.8          3.4           1.6          0.2
3          2           7.7          2.6           6.9          2.3
4          0           5.4          3.4           1.7          0.2
..       ...           ...          ...           ...          ...
107        0           5.1          3.7           1.5          0.4
108        1           6.7          3.1           4.7          1.5
109        0           4.7          3.2           1.3          0.2
110        0           5.8          4.0           1.2          0.2
111        0           5.1          3.5           1.4          0.2

[112 rows x 5 columns]

>>> test_df
    .target  sepal length  sepal width  petal length  petal width
0         2           6.3          2.7           4.9          1.8
1         0           4.5          2.3           1.3          0.3
2         2           5.8          2.8           5.1          2.4
3         0           4.3          3.0           1.1          0.1
4         0           5.0          3.0           1.6          0.2
..      ...           ...          ...           ...          ...
33        1           6.7          3.1           4.4          1.4
34        0           4.6          3.6           1.0          0.2
35        1           5.7          3.0           4.2          1.2
36        1           5.9          3.0           4.2          1.5
37        2           6.4          2.8           5.6          2.1

[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 = expd.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

Use patsy

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

```
>>> import expandas as expd

# create modelframe which doesn't have target
>>> df = expd.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 = expd.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 = expd.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
```

API:

CHAPTER 5

expandas.core package

5.1 Submodules

5.2 Module contents

CHAPTER 6

expandas.skaccessors package

6.1 Subpackages

6.1.1 expandas.skaccessors.test package

Submodules

Module contents

6.2 Submodules

6.3 Module contents