pmll Documentation

Release 0.2.2

Kirill Pavlov

July 09, 2014

Contents

Contents:

CHAPTER 1

Getting started

1.1 Install

To install *pmll* use pip:

```
pip install pmll
```

1.2 Data structures

There are two main data structures in the package: *feature* and *dataset*.

1.2.1 Feature

Feature class is used to describe objects. Each feature has title and type.

Features are in feature space, they have title and optional type. Type is used to define possible operations with features, it could be *lin*, *nom*, *bin*, *rank*.

```
In [1]: from pmll.feature import Feature, FeatureNom, FeatureLin
In [2]: from collections import namedtuple
In [3]: Parallelepiped = namedtuple("Parallelepiped", ["colour", "length", "height", "width"])
In [4]: cube = Parallelepiped("red", 2, 2, 2)
```

Lets define features for new object: colour is nominal, others are linear.

```
In [5]: f = FeatureNom("colour")
In [6]: f(cube)
Out[6]: 'red'

In [7]: f1, f2, f3 = FeatureLin("length"), FeatureLin("height"), FeatureLin("width")
```

It is possible to multiply linear features, for example square equals length times height

```
In [8]: square = f1 * f2
In [9]: square.title
Out[9]: 'height*length'

In [10]: square(cube)
Out[10]: 4.0
```

```
In [11]: volume = square * f3
In [12]: volume.title
Out[12]: 'height*length*width'

In [13]: volume(cube)
Out[13]: 8.0
```

There are trigonometric functions available

```
In [14]: from pmll.feature.operations import sin, cos, tan
In [15]: f1sin = sin(f1)
In [16]: f1sin(cube)
Out[10]: 0.9092974268256817
```

1.2.2 Data

Data class is used to represent objects using their features.

```
In [1]: from pmll.data import Data
In [2]: from pmll.feature import FeatureLin
In [3]: d = Data([[0, 1, 2], [3, 1, 4], [-2, 0, 1], [2, -1, -1], [0, 0, 2]])
```

Data is used for objects manipulation. It also provides some statistical information, such as Variance inflation factor:

```
In [4]: d.vif
Out[4]: [0.22780361757105938, 0.5469767441860465, 0.34108527131782956]
```

and for each feature there is basic statistical information:

```
In [5]: d.stat
Out[5]:
{<class 'pmll.feature.models.FeatureLin'>: f0 (scale=lin): {'max': 3.0,
  'mean': 0.59999999999999998,
  'min': -2.0,
  'std': 1.7435595774162693,
  'var': 3.04},
 <class 'pmll.feature.models.FeatureLin'>: f1 (scale=lin): {'max': 1.0,
  'mean': 0.20000000000000001,
  'min': -1.0,
  'std': 0.74833147735478833,
  'var': 0.56000000000000005},
 <class 'pmll.feature.models.FeatureLin'>: f2 (scale=lin): {'max': 4.0,
  'mean': 1.6000000000000001,
  'min': -1.0,
  'std': 1.6248076809271921,
  'var': 2.6400000000000001}}
```

Data object could be converted to numpy.matrix if features are linear:

```
In [6]: d.matrix
Out[6]:
matrix([[ 0.,  1.,  2.],
        [ 3.,  1.,  4.],
        [-2.,  0.,  1.],
        [ 2., -1., -1.],
        [ 0.,  0.,  2.]])
```

To extend data objects, add them directly to data.objects. To extend data features, it is possible to sum data objects. Note, that data objects sould have the same number of objects and not intersected features.

```
In [7]: d2 = Data([[3], [7], [1], [-2], [4]], features=[FeatureLin('f3')])
In [8]: d = d + d2
```

1.2.3 Quality metrics calculation

In this part we use the same data object *d* as before. We will predict one feature using others and calculate quality metrics. In terms of data mining problem, it would be quality measure on train set.

```
In [9]: X, Y = d[:, :-1].matrix, d[:, -1:].matrix
```

We use least squares method here:

```
In [10]: w = (X.T * X) ** (-1) * X.T * Y
In [11]: w
Out[11]: matrix([[ 0.07751938], [-0.02325581],  [1.71317829]])
```

Prediction for train set would be

```
In [12]: prediction = X * w
In [13]: prediction
Out[13]:
matrix([[ 3.40310078],
        [ 7.0620155 ],
        [ 1.55813953],
        [-1.53488372],
        [ 3.42635659]])
```

Lets measure quality of prediction of train set.

```
In [14]: from pmll.metrics.base import QualityMeasurerLinear
In [15]: q = QualityMeasurerLinear()
In [16]: for p, y in zip(prediction.tolist(), Y.tolist()):
   ....:     q.append(p[0], y[0])

In [17]: (q.mse, q.mae, q.rmse, q.nrmse, q.cvrmse)
Out[17]:
(0.20465116279069767,
0.41240310077519365,
0.4523838666339657,
0.050264874070440634,
0.17399379485921757)
```

1.3 Feature Generation

Feature generation is sweet with pmll. Unlike other libraries, pmll works with features, rather than data matrix columns. It allows to perform operations in feature space and then get objects for current data features.

Polynomial regression

Consider function as data object. First feature is x values, second feature is y values.

```
In [1]: from pmll.data import Data
In [2]: from pmll.feature import FeatureLin
In [3]: import math
In [4]: d = Data([[x, math.sin(x)] for x in range(5)], features=[FeatureLin('x'), FeatureLin('y')])
```

Problem is to predict y value for x=4 using previous values. Linear model without feature generation is y = x * w

```
In [5]: X, Y = d.matrix[:-1, :-1], d.matrix[:-1, -1:]
In [6]: w = (X.T * X) ** (-1) * X.T * Y
In [7]: error = (d.matrix[-1:, :-1] * w - d.matrix[-1:, -1:]) ** 2
In [8]: error
Out[8]: matrix([[ 2.68232763]])
```

Lets generate features. Start with constant, so, model is y = w_0 + x * w_1 or (1, x) * (w_0, w_1)

```
In [9]: d.features = [d.features[0] ** 0] + d.features
In [10]: d.matrix
Out[10]: matrix([[ 1.        ,  0.        ,  0.        ],
                 [ 1.        ,  1.        ,  0.84147098],
                 [ 1.        ,  2.        ,  0.90929743],
                 [ 1.        ,  3.        ,  0.14112001],
                 [ 1.        ,  4.        , -0.7568025 ]])

In [11]: X, Y = d.matrix[:-1, :-1], d.matrix[:-1, -1:]
In [12]: w = (X.T * X) ** (-1) * X.T * Y
In [13]: error = (d.matrix[-1:, :-1] * w - d.matrix[-1:, -1:]) ** 2
In [14]: error
Out[14]: matrix([[ 1.8294489]])
```

Finally we use model y = w_3 * x^3 + w_2 * x^2 + w_1 * x + w_0

```
In [15]: d.features = d.features[:2] + [d.features[1] ** 2, d.features[1] ** 3] + d.features[2:]
In [14]: error
Out[14]: matrix([[ 0.59077377]])
```

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*