2.3 Using standard interface

February 22, 2017

1 Using standard interface

The following notebooks presents the basic usage of native XGBoost Python interface.

Flight-plan: - load libraries and prepare data, - specify parameters, - train classifier, - make predictions

1.0.1 Loading libraries

Begin with loading all required libraries in one place:

```
In [1]: import numpy as np
        import xgboost as xgb
```

1.0.2 Loading data

We are going to use bundled Agaricus dataset which can be downloaded here.

> This data set records biological attributes of different mushroom species, and the target is to predict whether it is poisonous
>
> This data set includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom;

It consist of 8124 instances, characterized by 22 attributes (both numeric and categorical). The target class is either 0 or 1 which means binary classification problem.

> **Important**: XGBoost handles only numeric variables.

Lucily all the data have alreay been pre-process for us. Categorical variables have been encoded, and all instances divided into train and test datasets. You will know how to do this on your own in later lectures.

Data needs to be stored in `DMatrix` object which is designed to handle sparse datasets. It can be populated in couple ways: - using libsvm format txt file, - using Numpy 2D array (most popular), - using XGBoost binary buffer file

In this case we'll use first option.

Libsvm files stores only non-zero elements in format

```
<label> <feature_a>:<value_a> <feature_c>:<value_c> ...
<feature_z>:<value_z>
```

Any missing features indicate that it's corresponding value is 0.

```
In [2]: dtrain = xgb.DMatrix('../data/agaricus.txt.train')
        dtest = xgb.DMatrix('../data/agaricus.txt.test')
```

Let's examine what was loaded:

```
In [3]: print("Train dataset contains {0} rows and {1} columns".format(dtrain.num_row(), dtrain.
        print("Test dataset contains {0} rows and {1} columns".format(dtest.num_row(), dtest.num
```

```
Train dataset contains 6513 rows and 127 columns
Test dataset contains 1611 rows and 127 columns
```

```
In [4]: print("Train possible labels: ")
        print(np.unique(dtrain.get_label()))

        print("\nTest possible labels: ")
        print(np.unique(dtest.get_label()))
```

```
Train possible labels:
[ 0.  1.]

Test possible labels:
[ 0.  1.]
```

1.0.3 Specify training parameters

Let's make the following assuptions and adjust algorithm parameters to it: - we are dealing with binary classification problem (`'objective':'binary:logistic'`), - we want shallow single trees with no more than 2 levels (`'max_depth':2`), - we don't any oupout (`'silent':1`), - we want algorithm to learn fast and aggressively (`'eta':1`), - we want to iterate only 5 rounds

```
In [5]: params = {
            'objective':'binary:logistic',
            'max_depth':2,
            'silent':1,
            'eta':1
        }

        num_rounds = 5
```

1.0.4 Training classifier

To train the classifier we simply pass to it a training dataset, parameters list and information about number of iterations.

```
In [6]: bst = xgb.train(params, dtrain, num_rounds)
```

We can also observe performance on test dataset using `watchlist`

```
In [7]: watchlist  = [(dtest,'test'), (dtrain,'train')] # native interface only
        bst = xgb.train(params, dtrain, num_rounds, watchlist)

[0]     test-error:0.042831     train-error:0.046522
[1]     test-error:0.021726     train-error:0.022263
[2]     test-error:0.006207     train-error:0.007063
[3]     test-error:0.018001     train-error:0.0152
[4]     test-error:0.006207     train-error:0.007063
```

1.0.5 Make predictions

```
In [8]: preds_prob = bst.predict(dtest)
        preds_prob

Out[8]: array([ 0.08073306,  0.92217326,  0.08073306, ...,  0.98059034,
                0.01182149,  0.98059034], dtype=float32)
```

Calculate simple accuracy metric to verify the results. Of course validation should be performed accordingly to the dataset, but in this case accuracy is sufficient.

```
In [9]: labels = dtest.get_label()
        preds = preds_prob > 0.5 # threshold
        correct = 0

        for i in range(len(preds)):
            if (labels[i] == preds[i]):
                correct += 1

        print('Predicted correctly: {0}/{1}'.format(correct, len(preds)))
        print('Error: {0:.4f}'.format(1-correct/len(preds)))

Predicted correctly: 1601/1611
Error: 0.0062
```