3.4 Evaluate results

February 22, 2017

1 Evaluate results

In this notebook you will see how measure the quality of the algorithm performance.

You will learn how to: - use predefined evaluation metrics, - write your own evaluation metrics, - use early stopping feature, - cross validate results

1.0.1 Prepare data

Begin with loading all required libraries:

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

        from pprint import pprint

        # reproducibility
        seed = 123
        np.random.seed(seed)
```

Load agaricus dataset from file:

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

Specify training parameters - we are going to use 5 decision tree stumps with average learning rate.

```
In [3]: # specify general training parameters
        params = {
            'objective':'binary:logistic',
            'max_depth':1,
            'silent':1,
            'eta':0.5
        }

        num_rounds = 5
```

Before training the model let's also specify `watchlist` array to observe it's performance on the both datasets.

```
In [4]: watchlist  = [(dtest,'test'), (dtrain,'train')]
```

1.0.2 Using predefined evaluation metrics

What is already available? There are already some predefined metrics availabe. You can use them as the input for the `eval_metric` parameter while training the model.

- `rmse` - root mean square error,
- `mae` - mean absolute error,
- `logloss` - negative log-likelihood
- `error` - binary classification error rate. It is calculated as `#(wrong cases)/#(all cases)`. Treat predicted values with probability $p > 0.5$ as positive,
- `merror` - multiclass classification error rate. It is calculated as `#(wrong cases)/#(all cases)`,
- `auc` - area under curve,
- `ndcg` - normalized discounted cumulative gain,
- `map` - mean average precision

By default an `error` metric will be used.

```
In [5]: bst = xgb.train(params, dtrain, num_rounds, watchlist)
```

```
[0]        test-error:0.11049       train-error:0.113926
[1]        test-error:0.11049       train-error:0.113926
[2]        test-error:0.03352       train-error:0.030401
[3]        test-error:0.027312       train-error:0.021495
[4]        test-error:0.031037       train-error:0.025487
```

To change is simply specify the `eval_metric` argument to the `params` dictionary.

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

```
[0]        test-logloss:0.457887       train-logloss:0.460108
[1]        test-logloss:0.383911       train-logloss:0.378728
[2]        test-logloss:0.312678       train-logloss:0.308061
[3]        test-logloss:0.26912       train-logloss:0.26139
[4]        test-logloss:0.239746       train-logloss:0.232174
```

You can also use multiple evaluation metrics at one time

```
In [7]: params['eval_metric'] = ['logloss', 'auc']
        bst = xgb.train(params, dtrain, num_rounds, watchlist)
```

```
[0]        test-logloss:0.457887       test-auc:0.892138       train-logloss:0.460108       t
[1]        test-logloss:0.383911       test-auc:0.938901       train-logloss:0.378728       t
[2]        test-logloss:0.312678       test-auc:0.976157       train-logloss:0.308061       t
[3]        test-logloss:0.26912       test-auc:0.979685       train-logloss:0.26139       tra
[4]        test-logloss:0.239746       test-auc:0.9785       train-logloss:0.232174       tra
```

1.0.3 Creating custom evaluation metric

In order to create our own evaluation metric, the only thing needed to do is to create a method taking two arguments - predicted probabilities and `DMatrix` object holding training data.

In this example our classification metric will simply count the number of misclassified examples assuming that classes with $p > 0.5$ are positive. You can change this threshold if you want more certainty.

The algorithm is getting better when the number of misclassified examples is getting lower. Remember to also set the argument `maximize=False` while training.

```
In [8]: # custom evaluation metric
        def misclassified(pred_probs, dtrain):
            labels = dtrain.get_label() # obtain true labels
            preds = pred_probs > 0.5 # obtain predicted values
            return 'misclassified', np.sum(labels != preds)
```

```
In [9]: bst = xgb.train(params, dtrain, num_rounds, watchlist, feval=misclassified, maximize=Fal
```

```
[0]        test-misclassified:178        train-misclassified:742
[1]        test-misclassified:178        train-misclassified:742
[2]        test-misclassified:54        train-misclassified:198
[3]        test-misclassified:44        train-misclassified:140
[4]        test-misclassified:50        train-misclassified:166
```

You can see that even though the `params` dictionary is holding `eval_metric` key these values are being ignored and overwritten by `feval`.

1.0.4 Extracting the evaluation results

You can get evaluation scores by declaring a dictionary for holding values and passing it as a parameter for `evals_result` argument.

```
In [10]: evals_result = {}
         bst = xgb.train(params, dtrain, num_rounds, watchlist, feval=misclassified, maximize=Fa
```

```
[0]        test-misclassified:178        train-misclassified:742
[1]        test-misclassified:178        train-misclassified:742
[2]        test-misclassified:54        train-misclassified:198
[3]        test-misclassified:44        train-misclassified:140
[4]        test-misclassified:50        train-misclassified:166
```

Now you can reuse these scores (i.e. for plotting)

```
In [11]: pprint(evals_result)
```

```
{'test': {'misclassified': [178.0, 178.0, 54.0, 44.0, 50.0]},
 'train': {'misclassified': [742.0, 742.0, 198.0, 140.0, 166.0]}}
```

1.0.5 Early stopping

There is a nice optimization trick when fitting multiple trees.

You can train the model until the validation score **stops** improving. Validation error needs to decrease at least every `early_stopping_rounds` to continue training. This approach results in simpler model, because the lowest number of trees will be found (simplicity).

In the following example a total number of 1500 trees is to be created, but we are telling it to stop if the validation score does not improve for last ten iterations.

```
In [12]: params['eval_metric'] = 'error'
         num_rounds = 1500

         bst = xgb.train(params, dtrain, num_rounds, watchlist, early_stopping_rounds=10)
```

```
[0]	test-error:0.11049	train-error:0.113926
Multiple eval metrics have been passed: 'train-error' will be used for early stopping.

Will train until train-error hasn't improved in 10 rounds.
[1]	test-error:0.11049	train-error:0.113926
[2]	test-error:0.03352	train-error:0.030401
[3]	test-error:0.027312	train-error:0.021495
[4]	test-error:0.031037	train-error:0.025487
[5]	test-error:0.019243	train-error:0.01735
[6]	test-error:0.019243	train-error:0.01735
[7]	test-error:0.015518	train-error:0.013358
[8]	test-error:0.015518	train-error:0.013358
[9]	test-error:0.009311	train-error:0.007523
[10]	test-error:0.015518	train-error:0.013358
[11]	test-error:0.019243	train-error:0.01735
[12]	test-error:0.009311	train-error:0.007523
[13]	test-error:0.001862	train-error:0.001996
[14]	test-error:0.005587	train-error:0.005988
[15]	test-error:0.005587	train-error:0.005988
[16]	test-error:0.005587	train-error:0.005988
[17]	test-error:0.005587	train-error:0.005988
[18]	test-error:0.005587	train-error:0.005988
[19]	test-error:0.005587	train-error:0.005988
[20]	test-error:0.005587	train-error:0.005988
[21]	test-error:0.005587	train-error:0.005988
[22]	test-error:0.001862	train-error:0.001996
[23]	test-error:0.001862	train-error:0.001996
Stopping. Best iteration:
[13]	test-error:0.001862	train-error:0.001996
```

When using `early_stopping_rounds` parameter resulting model will have 3 additional fields - `bst.best_score`, `bst.best_iteration` and `bst.best_ntree_limit`.

```
In [13]: print("Booster best train score: {}".format(bst.best_score))
         print("Booster best iteration: {}".format(bst.best_iteration))
         print("Booster best number of trees limit: {}".format(bst.best_ntree_limit))
```

```
Booster best train score: 0.001996
Booster best iteration: 13
Booster best number of trees limit: 14
```

Also keep in mind that `train()` will return a model from the last iteration, not the best one.

1.0.6 Cross validating results

Native package provides an option for cross-validating results (but not as sophisticated as Sklearn package). The next input shows a basic execution. Notice that we are passing only single `DMatrix`, so it would be good to merge train and test into one object to have more training samples.

```
In [14]: num_rounds = 10 # how many estimators
         hist = xgb.cv(params, dtrain, num_rounds, nfold=10, metrics={'error'}, seed=seed)
         hist
```

Out[14]:

	test-error-mean	test-error-std	train-error-mean	train-error-std
0	0.113825	0.013186	0.113825	0.001465
1	0.113825	0.013186	0.113825	0.001465
2	0.030415	0.005698	0.030415	0.000633
3	0.021505	0.005277	0.021505	0.000586
4	0.025499	0.005461	0.025499	0.000607
5	0.020737	0.007627	0.019696	0.003491
6	0.017358	0.003369	0.017358	0.000374
7	0.015361	0.003699	0.014474	0.001923
8	0.013364	0.003766	0.013364	0.000419
9	0.012596	0.004700	0.011742	0.003820

Notice that:

- by default we get a pandas data frame object (can be changed with `as_pandas` param),
- metrics are passed as an argument (muliple values are allowed),
- we can use own evaluation metrics (param `feval` and `maximize`),
- we can use early stopping feature (param `early_stopping_rounds`)