3.3 Hyper-parameter tuning

February 22, 2017

1 Hyper-parameter tuning

As you know there are plenty of tunable parameters. Each one results in different output. The question is which combination results in best output.

The following notebook will show you how to use Scikit-learn modules for figuring out the best parameters for your models.

What's included: - data preparation, - finding best hyper-parameters using grid-search, - finding best hyper-parameters using randomized grid-search

1.0.1 Prepare data

Let's begin with loading all required libraries in one place and setting seed number for reproducibility.

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

        from xgboost.sklearn import XGBClassifier

        from sklearn.grid_search import GridSearchCV, RandomizedSearchCV
        from sklearn.datasets import make_classification
        from sklearn.cross_validation import StratifiedKFold

        from scipy.stats import randint, uniform

        # reproducibility
        seed = 342
        np.random.seed(seed)
```

Generate artificial dataset:

```
In [2]: X, y = make_classification(n_samples=1000, n_features=20, n_informative=8, n_redundant=3
```

Define cross-validation strategy for testing. Let's use `StratifiedKFold` which guarantees that target label is equally distributed across each fold:

```
In [3]: cv = StratifiedKFold(y, n_folds=10, shuffle=True, random_state=seed)
```

1.0.2 Grid-Search

In grid-search we start by defining a dictionary holding possible parameter values we want to test. **All** combinations will be evaluted.

```
In [4]: params_grid = {
            'max_depth': [1, 2, 3],
            'n_estimators': [5, 10, 25, 50],
            'learning_rate': np.linspace(1e-16, 1, 3)
        }
```

Add a dictionary for fixed parameters.

```
In [5]: params_fixed = {
            'objective': 'binary:logistic',
            'silent': 1
        }
```

Create a `GridSearchCV` estimator. We will be looking for combination giving the best accuracy.

```
In [6]: bst_grid = GridSearchCV(
            estimator=XGBClassifier(**params_fixed, seed=seed),
            param_grid=params_grid,
            cv=cv,
            scoring='accuracy'
        )
```

Before running the calculations notice that $3 * 4 * 3 * 10 = 360$ models will be created to test all combinations. You should always have rough estimations about what is going to happen.

```
In [7]: bst_grid.fit(X, y)
```

```
Out[7]: GridSearchCV(cv=sklearn.cross_validation.StratifiedKFold(labels=[0 1 ..., 1 1], n_folds=
                error_score='raise',
                estimator=XGBClassifier(base_score=0.5, colsample_bylevel=1, 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='binary:logistic', reg_alpha=0, reg_lambda=1,
                scale_pos_weight=1, seed=342, silent=1, subsample=1),
                fit_params={}, iid=True, n_jobs=1,
                param_grid={'n_estimators': [5, 10, 25, 50], 'learning_rate': array([  1.00000e-1
                pre_dispatch='2*n_jobs', refit=True, scoring='accuracy', verbose=0)
```

Now, we can look at all obtained scores, and try to manually see what matters and what not. A quick glance looks that the largeer `n_estimators` then the accuracy is higher.

```
In [8]: bst_grid.grid_scores_
```

```
Out[8]: [mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.49800, std: 0.00245, params: {'learning_rate': 9.9999999999999998e-17, 'n_estim
         mean: 0.84100, std: 0.03515, params: {'learning_rate': 0.5, 'n_estimators': 5, 'max_dep
         mean: 0.87300, std: 0.03374, params: {'learning_rate': 0.5, 'n_estimators': 10, 'max_de
         mean: 0.89200, std: 0.03375, params: {'learning_rate': 0.5, 'n_estimators': 25, 'max_de
         mean: 0.90200, std: 0.03262, params: {'learning_rate': 0.5, 'n_estimators': 50, 'max_de
         mean: 0.86400, std: 0.04665, params: {'learning_rate': 0.5, 'n_estimators': 5, 'max_dep
         mean: 0.89400, std: 0.04189, params: {'learning_rate': 0.5, 'n_estimators': 10, 'max_de
         mean: 0.92200, std: 0.02584, params: {'learning_rate': 0.5, 'n_estimators': 25, 'max_de
         mean: 0.92000, std: 0.02233, params: {'learning_rate': 0.5, 'n_estimators': 50, 'max_de
         mean: 0.89700, std: 0.03904, params: {'learning_rate': 0.5, 'n_estimators': 5, 'max_dep
         mean: 0.92000, std: 0.02864, params: {'learning_rate': 0.5, 'n_estimators': 10, 'max_de
         mean: 0.92300, std: 0.02193, params: {'learning_rate': 0.5, 'n_estimators': 25, 'max_de
         mean: 0.92400, std: 0.02255, params: {'learning_rate': 0.5, 'n_estimators': 50, 'max_de
         mean: 0.83500, std: 0.04939, params: {'learning_rate': 1.0, 'n_estimators': 5, 'max_dep
         mean: 0.86800, std: 0.03386, params: {'learning_rate': 1.0, 'n_estimators': 10, 'max_de
         mean: 0.89500, std: 0.02720, params: {'learning_rate': 1.0, 'n_estimators': 25, 'max_de
         mean: 0.90500, std: 0.02783, params: {'learning_rate': 1.0, 'n_estimators': 50, 'max_de
         mean: 0.87800, std: 0.03342, params: {'learning_rate': 1.0, 'n_estimators': 5, 'max_dep
         mean: 0.90800, std: 0.04261, params: {'learning_rate': 1.0, 'n_estimators': 10, 'max_de
         mean: 0.91000, std: 0.03632, params: {'learning_rate': 1.0, 'n_estimators': 25, 'max_de
         mean: 0.91300, std: 0.02449, params: {'learning_rate': 1.0, 'n_estimators': 50, 'max_de
         mean: 0.90500, std: 0.03112, params: {'learning_rate': 1.0, 'n_estimators': 5, 'max_dep
         mean: 0.91700, std: 0.02729, params: {'learning_rate': 1.0, 'n_estimators': 10, 'max_de
         mean: 0.92700, std: 0.03342, params: {'learning_rate': 1.0, 'n_estimators': 25, 'max_de
         mean: 0.93300, std: 0.02581, params: {'learning_rate': 1.0, 'n_estimators': 50, 'max_de
```

If there are many results, we can filter them manually to get the best combination

```
In [9]: print("Best accuracy obtained: {0}".format(bst_grid.best_score_))
        print("Parameters:")
        for key, value in bst_grid.best_params_.items():
            print("\t{}: {}".format(key, value))
```

```
Best accuracy obtained: 0.933
Parameters:
        learning_rate: 1.0
        n_estimators: 50
```

```
        max_depth: 3
```

Looking for best parameters is an iterative process. You should start with coarsed-granularity and move to to more detailed values.

1.0.3 Randomized Grid-Search

When the number of parameters and their values is getting big traditional grid-search approach quickly becomes ineffective. A possible solution might be to randomly pick certain parameters from their distribution. While it's not an exhaustive solution, it's worth giving a shot.

Create a parameters distribution dictionary:

```
In [10]: params_dist_grid = {
             'max_depth': [1, 2, 3, 4],
             'gamma': [0, 0.5, 1],
             'n_estimators': randint(1, 1001), # uniform discrete random distribution
             'learning_rate': uniform(), # gaussian distribution
             'subsample': uniform(), # gaussian distribution
             'colsample_bytree': uniform() # gaussian distribution
         }
```

Initialize `RandomizedSearchCV` to **randomly pick 10 combinations of parameters**. With this approach you can easily control the number of tested models.

```
In [11]: rs_grid = RandomizedSearchCV(
             estimator=XGBClassifier(**params_fixed, seed=seed),
             param_distributions=params_dist_grid,
             n_iter=10,
             cv=cv,
             scoring='accuracy',
             random_state=seed
         )
```

Fit the classifier:

```
In [12]: rs_grid.fit(X, y)
```

```
Out[12]: RandomizedSearchCV(cv=sklearn.cross_validation.StratifiedKFold(labels=[0 1 ..., 1 1], n
                    error_score='raise',
                    estimator=XGBClassifier(base_score=0.5, colsample_bylevel=1, colsample_bytree
                 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='binary:logistic', reg_alpha=0, reg_lambda=1,
                 scale_pos_weight=1, seed=342, silent=1, subsample=1),
                    fit_params={}, iid=True, n_iter=10, n_jobs=1,
                    param_distributions={'subsample': <scipy.stats._distn_infrastructure.rv_froze
                    pre_dispatch='2*n_jobs', random_state=342, refit=True,
                    scoring='accuracy', verbose=0)
```

One more time take a look at choosen parameters and their accuracy score:

```
In [13]: rs_grid.grid_scores_

Out[13]: [mean: 0.80200, std: 0.02403, params: {'subsample': 0.11676744056370758, 'n_estimators'
          mean: 0.90800, std: 0.02534, params: {'subsample': 0.4325346125891868, 'n_estimators':
          mean: 0.86400, std: 0.03584, params: {'subsample': 0.15239319471904489, 'n_estimators'
          mean: 0.90100, std: 0.02794, params: {'subsample': 0.70993001900730734, 'n_estimators'
          mean: 0.91200, std: 0.02440, params: {'subsample': 0.93610608633544701, 'n_estimators'
          mean: 0.92900, std: 0.01577, params: {'subsample': 0.76526283302535481, 'n_estimators'
          mean: 0.89900, std: 0.03200, params: {'subsample': 0.1047221390561941, 'n_estimators':
          mean: 0.89300, std: 0.02510, params: {'subsample': 0.70326840897694187, 'n_estimators'
          mean: 0.90300, std: 0.03573, params: {'subsample': 0.40219949856580106, 'n_estimators'
          mean: 0.88900, std: 0.02604, params: {'subsample': 0.18284845802969663, 'n_estimators'
```

There are also some handy properties allowing to quickly analyze best estimator, parameters and obtained score:

```
In [14]: rs_grid.best_estimator_

Out[14]: XGBClassifier(base_score=0.5, colsample_bylevel=1,
                colsample_bytree=0.80580143163765727, gamma=0,
                learning_rate=0.46363095388213049, max_delta_step=0, max_depth=4,
                min_child_weight=1, missing=None, n_estimators=281, nthread=-1,
                objective='binary:logistic', reg_alpha=0, reg_lambda=1,
                scale_pos_weight=1, seed=342, silent=1,
                subsample=0.76526283302535481)

In [15]: rs_grid.best_params_

Out[15]: {'colsample_bytree': 0.80580143163765727,
          'gamma': 0,
          'learning_rate': 0.46363095388213049,
          'max_depth': 4,
          'n_estimators': 281,
          'subsample': 0.76526283302535481}

In [16]: rs_grid.best_score_

Out[16]: 0.92900000000000005
```