Predicting human wine test preference, Part 3

XGBoost

The method that we used in the last notebook was boostrap agreggating (bagging). In this method we have 2 randomizing processes which ensure the uniqueness of the tree:

1. Pick a random number of features from the feature vector
2. Pick a random number of samples

We now turn our attention to XGBoost which is state-of-the-art in the world of Decision Trees. XGBoost uses a method known as gradient boosting. In the standard ensemble model, all models were trained in isolation and so they ended up doing the same mistakes.

XGBoost, however solves this by training the trees in succession each time improving on the mistakes of the previous model. Models are trained subsequently and each subsequent train aims to improve the errors made by the previous one.

Among the advantages of XGBoost one can mention

- Has default regularization (regularized boosting)
- Implements parallel processing that make it blazingly fast (also supports Hadoop implementation)
- High flexibility: It allows the users to define custom optimization objectives and evaluation criteria
- Has build-in routine to handle missing values
- Splits upto max_depth and then prunes the tree backwards
- Build-in Cross-Validation
- Continue where left of
    - Users can start training the XGBoost model from the last iteration of the previous run

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

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn.metrics import accuracy_score, confusion_matrix
```

In [2]:
```
features = pd.read_csv('data\winequality.csv')
labels = features['quality']
features.drop('quality', inplace=True, axis=1)
print('The dataset consists of {} samples and {} features'.format(*features.shape))

# Train/Test
train_features, test_features, train_labels, test_labels = train_test_split(features, labels, test_size
=0.2,
                                                                            random_state=1, stratify=lab
els)
```

```
The dataset consists of 4898 samples and 11 features
```

In [4]:
```
def evaluate_model(predictions, test_labels):
    """ Model evaluation tolerating a prediction of difference 1.
        That is if we predicted 3 or 5 but the real value was 4, we still count it as correct."""
    confusion_mat = confusion_matrix(test_labels, predictions)

    # Calculate tolerance micro precision
    sumTp = 0
    sumTpFp = sum(sum(confusion_mat))

    for i in range(len(confusion_mat)):
        for j in range(len(confusion_mat)):
            # element in main diagonal
            if (i == j):
                sumTp += confusion_mat[i][j]

            # element around main diagonal
            elif (j == i+1) & (i<5):
                sumTp += confusion_mat[i][j]

            elif (j == i-1) & (i>0):
                sumTp += confusion_mat[i][j]

    microP = sumTp/sumTpFp
    print("Tolerance precision:", round(100*microP, 2))
```

XGBoost can be used in two versions. First is the original version, second a wrapper provided for sklearn.

We first investiage the original version and initialize it with some kind of random values for the hyperparameters.

Docs on hyperparameters

In [5]:
```
# Convert into suitable data format for XGBoost
train = xgb.DMatrix(train_features, train_labels)
test = xgb.DMatrix(test_features, test_labels)

param = {
    'max_depth': 5,
    'eta': 0.3,
    'objective': 'multi:softmax',
    'num_class': 10,
    'gamma':0,
    'min_child_weight':1}

steps = 10

# Train the model
model1 = xgb.train(param, train, steps)

# Test
predictions = model1.predict(test)
evaluate_model(predictions, test_labels)
```

```
Tolerance precision: 96.22
```

An accuracy of 96.22% is not bad for a starter. Let us see if we can improve it using hyperparameters.

Hyperparemeter tunning

To do hyperparameter tuning we shall use the wrapper for sklearn

Docs on hyperparameters of the wrapped version

The parameters are divided into three categories. Some of the main parameters are listed below

1. General Parameters: Overal functioning
    - booster: Type of model to run at each iteration. Tree-based or linear model
    - num_feature: #of features in the feature vector that are actually used in boosting. Max by default
2. Booster Parameters. Tree considered here (but it also almost always outperforms the linear booster): Individual booster at each step
    - eta: the learning rate. Makes the model robust by shrinking the weights in each step. Dampens the decision power of the individual trees.
    - min_child_weight: Min sum of weights of all observations required in a child. Controls overfitting. Too higher values, however could also lead to underfit
    - max_depth: The maximal depth of the tree. Controls overfitting
    - max_leaf_nodes: The maximal number of nodes in the leafes. Alternative to max_depth
    - gamma: minimal loss reduction required to make the split
    - lambda: L2 regularization term on weights(Ridge). Increasing the value makes the model more conservative
    - alpha: L1 regularization term(Lasso). Likewise makes the model more convervative.
    - scale_pos_weight: A value greater than 0 helps in a faster convergance for imbalanced classes. 1 by default.
3. Learning Task Parameters: Optimization performed
    - objective: The lost function to be minimized
        - *binary:logistic* For a bernoulli distribution. Returns probabilities
        - *multi:softmax* For a multinomial distribution using softmax. Returns predicted class. Also need to set the *num_class* which defines the number of unique classes
        - *multi:soft* Same as softmax but it returns probabilities

First we try out random parameters out of a pool of huge number of parameters using **RandomSearchCV**. Then we shall narrow down the search using **GridSearchCV**.

The time taken to try out all can grow quite a lot because we are training an ensemble of decision trees many times over. Here we do 500 iterations totaling 1500 fits of CV.

In [57]:
```
model2 = xgb.XGBClassifier(objective='multi:softmax', booster='gbtree', random_state=1)
parameters = {
    'max_depth': [i for i in range(2,13)],
    'min_child_weight': [i for i in range(1,10)],
    'learning_rate': [i for i in np.linspace(0.1,0.3, 8)],
    'gamma': [i for i in np.linspace(0.1, 1.5, 4)],
    'reg_lambda ':[0,1],
    'scale_pos_weight':[1],
    'num_class':[10],
    'tree_method': ['auto', 'exact', 'approx', 'gpu_hist']
}

randomSearch = RandomizedSearchCV(estimator=model2, param_distributions=parameters,
                                  scoring='neg_mean_absolute_error',
                                  cv=3, verbose=5, n_jobs=-1, n_iter=500)

randomSearch.fit(train_features, train_labels);
```

```
Fitting 3 folds for each of 500 candidates, totalling 1500 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 tasks      | elapsed:   14.9s
[Parallel(n_jobs=-1)]: Done  64 tasks      | elapsed:  2.2min
[Parallel(n_jobs=-1)]: Done 154 tasks      | elapsed:  4.5min
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed:  7.6min
[Parallel(n_jobs=-1)]: Done 442 tasks      | elapsed: 11.6min
[Parallel(n_jobs=-1)]: Done 640 tasks      | elapsed: 16.7min
[Parallel(n_jobs=-1)]: Done 874 tasks      | elapsed: 23.6min
[Parallel(n_jobs=-1)]: Done 1144 tasks     | elapsed: 30.2min
[Parallel(n_jobs=-1)]: Done 1450 tasks     | elapsed: 39.2min
[Parallel(n_jobs=-1)]: Done 1500 out of 1500 | elapsed: 40.7min finished
```

The parameters that we tried out

In [18]:
```
parameters
```

Out[18]:
```
{'max_depth': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
 'min_child_weight': [1, 2, 3, 4, 5, 6, 7, 8, 9],
 'learning_rate': [0.1,
  0.12857142857142856,
  0.15714285714285714,
  0.18571428571428572,
  0.21428571428571433,
  0.24285714285714285,
  0.2714285714285714,
  0.3],
 'gamma': [0.1, 0.5666666666666667, 1.0333333333333332, 1.5],
 'reg_alpha ': [0, 1],
 'scale_pos_weight': [1],
 'num_class': [10],
 'tree_method': ['auto', 'exact', 'approx', 'gpu_hist']}
```

Best random parameters that the algorithm was able to find for us

In [60]:
```
randomSearch.best_params_
```

Out[60]:
```
{'tree_method': 'approx',
 'scale_pos_weight': 1,
 'reg_lambda ': 1,
 'num_class': 10,
 'min_child_weight': 6,
 'max_depth': 12,
 'learning_rate': 0.3,
 'gamma': 0.5666666666666667}
```

Evaluate model

In [51]:
```
best_model = randomSearch.best_estimator_
predictions2 = best_model.predict(test_features)
evaluate_model(predictions2, test_labels)
```

```
Tolerance precision: 96.12
```

The performance slightly decreased.

Now let's narrow down the search using **GridSearch**. We try 960 candidates for an unreasonable amount of time (in such cases apache spark might be of benefit).

In [7]:
```
model3 = xgb.XGBClassifier(objective='multi:softmax', booster='gbtree', random_state=1)
grid_params = {
    'max_depth': [i for i in range(10,15)],
    'min_child_weight': [i for i in range(5,8)],
    'learning_rate': [i for i in np.linspace(0.2,0.5, 4)],
    'gamma': [i for i in np.linspace(0.5, 0.7, 4)],
    'reg_lambda':[0.2, 0.5, 0.7, 1],
    'scale_pos_weight':[1],
    'num_class':[10],
    'tree_method': ['approx']
}
gridSearch = GridSearchCV(estimator=model3, param_grid=grid_params, cv=3, n_jobs=-1, verbose=5,
                          scoring='neg_mean_absolute_error')

gridSearch.fit(train_features, train_labels);
```

```
Fitting 3 folds for each of 960 candidates, totalling 2880 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 tasks      | elapsed:   43.6s
[Parallel(n_jobs=-1)]: Done  64 tasks      | elapsed:  3.4min
[Parallel(n_jobs=-1)]: Done 154 tasks      | elapsed:  7.5min
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed: 13.0min
[Parallel(n_jobs=-1)]: Done 442 tasks      | elapsed: 20.3min
[Parallel(n_jobs=-1)]: Done 640 tasks      | elapsed: 29.0min
[Parallel(n_jobs=-1)]: Done 874 tasks      | elapsed: 40.0min
[Parallel(n_jobs=-1)]: Done 1144 tasks     | elapsed: 52.5min
[Parallel(n_jobs=-1)]: Done 1450 tasks     | elapsed: 66.4min
[Parallel(n_jobs=-1)]: Done 1792 tasks     | elapsed: 82.7min
[Parallel(n_jobs=-1)]: Done 2170 tasks     | elapsed: 99.9min
[Parallel(n_jobs=-1)]: Done 2584 tasks     | elapsed: 119.5min
[Parallel(n_jobs=-1)]: Done 2880 out of 2880 | elapsed: 133.4min finished
```

In [13]:
```
best_model2 = gridSearch.best_estimator_
predictions3 = best_model2.predict(test_features)
evaluate_model(predictions3, test_labels)
```

```
Tolerance precision: 96.63
```

Conclusion

Although we managed to slightly increase the performance, this model does not perform better that the bagging implementation of Random Forests by sklearn as seen in notebook #2