Lecture Notes

Random Forests

You are familiar with decision trees, now it's time to learn about **Random Forests**, which is a collection of decision trees. The great thing about random forests is that - they almost always outperform a decision tree in terms of accuracy.

Ensembles

An ensemble means a group of things viewed as a whole rather than individually. In ensembles, a **collection of models** is used to make predictions, rather than individual models. Arguably, the most popular in the family of ensemble models is the random forest: an ensemble made by the **combination of a large number of decision trees.**

For an ensemble to work, each model of the ensemble should comply with the following conditions:

1. Each model should be **diverse**. Diversity ensures that the models serve **complementary** purposes, which means that the individual models make predictions **independent of each other.**
2. Each model should be acceptable. **Acceptability** implies that each model is at least **better than a random model**.

Consider a binary classification problem where the response variable is either 0 or 1. You have an ensemble of three models, where each model has an accuracy of 0.7 i.e. it is correct 70% of the times. The following table shows all the possible cases that can occur while classifying a test data point as 1 or 0. The column to the extreme right shows the probability of each case.

Case	Result of Each Model			Result of the Ensemble	Probability
	m1	m2	m3		
1	Correct	Correct	Correct	**Correct**	0.7*0.7*0.7 = 0.343
2	Correct	Correct	Incorrect	**Correct**	0.7*0.7*0.3 = 0.147
3	Correct	Incorrect	Correct	**Correct**	0.7*0.3*0.7 = 0.147
4	Incorrect	Correct	Correct	**Correct**	0.3*0.7*0.7 = 0.147
5	Incorrect	Incorrect	Correct	**Incorrect**	0.3*0.3*0.7 = 0.063
6	Incorrect	Correct	Incorrect	**Incorrect**	0.3*0.7*0.3 = 0.063
7	Correct	Incorrect	Incorrect	**Incorrect**	0.7*0.3*0.3 = 0.063
8	Incorrect	Incorrect	Incorrect	**Incorrect**	0.3*0.3*0.3 = 0.027

Figure 1- Ensemble models

In the table, there are 4 cases each where the decision of the final model (ensemble) is either correct or wrong. Let's assume that the probability of the ensemble being correct is p, and the probability of the ensemble being wrong is q.

For the data in the table, p and q can be calculated as follows:

$p = 0.343 + 0.147 + 0.147 + 0.147 = 0.784$

$q = 0.027 + 0.063 + 0.063 + 0.063 = 0.216 = 1 - p$

You can see how an ensemble of just three model gives a boost to the accuracy from 70% to 78.4%. In general, the more the number of models, the higher the accuracy of an ensemble is.

Creating a Random Forest

Random forests are created using a special ensemble method called **bagging**. Bagging stands for **Bootstrap Aggregation. Bootstrapping** means creating bootstrap samples from a given data set. A bootstrap sample is created by **sampling** the given data set **uniformly** and **with replacement**. A bootstrap sample typically contains about 30-70% data from the data set. **Aggregation** implies combining the results of different models present in the ensemble.

Random forests is an ensemble of many decision trees. A random forest is created in the following way:

1. Create a **bootstrap sample** from the training set.

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	Y
1	22	54	56	21	78	82	53	67	56	0
→ 2	34	82	34	84	67	23	76	32	24	1
→ 3	12	38	13	54	74	57	86	63	89	1
4	32	93	26	21	24	34	91	34	97	0
5	62	23	64	67	68	67	24	28	52	1
→ :	:	:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:	:	:
→ 98	23	19	86	89	57	21	53	54	44	1
99	56	12	54	90	25	86	78	34	38	0
→ 100	15	94	21	24	15	56	23	79	92	0

Figure 2 - Training set

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	Y
2	34	82	34	84	67	23	76	32	24	1
3	12	38	13	54	74	57	86	63	89	1
78	32	93	26	21	24	34	91	34	97	0
98	23	19	86	89	57	21	53	54	44	1
100	15	94	21	24	15	56	23	79	92	0

Figure 3- Bootstrap sample

2. Now construct a decision tree using the bootstrap sample. While splitting a node of the tree, only consider a **random subset of features**. Every time a node has to split, a different random subset of features will be considered.
3. Repeat the steps 1 and 2 *n* times, to construct *n* trees in the forest. Remember each tree is constructed independently, so it is possible to construct each tree in parallel.
4. While predicting a test case, each tree predicts individually, and the final prediction is given by the majority vote of all the trees.

There are several advantages of a random forest:

1. A random forest is more **stable** than any single decision tree because the results get averaged out; it is not affected by the instability and bias of an individual tree.
2. A random forest is **immune to the curse of dimensionality** since only a subset of features is used to split a node.
3. You can **parallelize** the training of a forest since each tree is constructed independently.

4. You can calculate the OOB (Out-of-Bag) error using the training set which gives a really good estimate of the performance of the forest on unseen data. Hence there is no need to split the data into training and validation; you can use all the data to train the forest.

OOB (Out-of-Bag) Error

The OOB error is calculated by using each observation of the training set as a test observation. Since each tree is built on a bootstrap sample, each observation can used as a test observation by those trees which did not have it in their bootstrap sample. All these trees predict on this observation and you get an error for a single observation. The final OOB error is calculated by calculating the error on each observation and aggregating it.

It turns out that the OOB error is as good as **cross validation error**.

Time taken to build a forest

To construct a forest of *S* trees, on a dataset which has *M* features and *N* observations, the time taken will depends on the following factors:

1. The **number of trees**. The time is directly proportional to the number of trees. But this time can be reduced by creating the trees in parallel.
2. The **size of bootstrap sample**. Generally the size of a bootstrap sample is 30-70% of *N*. The smaller the size the faster it takes to create a forest.
3. The **size of subset of features** while splitting a node. Generally this is taken as $\sqrt{M}$ in classification and M/3 in regression.

Random forests lab

Random Forest with default hyperparameters

```
# Importing random forest classifier from sklearn library
from sklearn.ensemble import RandomForestClassifier

# Running the random forest with default parameters.
rfc = RandomForestClassifier()
# fit
rfc.fit(X_train,y_train)
# Making predictions
predictions = rfc.predict(X_test)
```

```
# Importing classification report and confusion matrix from sklearn metrics
from sklearn.metrics import classification_report,confusion_matrix, accuracy_score

# Let's check the report of our default model
print(classification_report(y_test,predictions))
print(accuracy_score(y_test,predictions))
# Printing confusion matrix
print(confusion_matrix(y_test,predictions))
```

Hyperparameter Tuning

The following hyperparameters are present in a random forest classifier. Note that most of these hyperparameters are actually of the decision trees that are in the forest.

- n_estimators: integer, optional (default=10): The number of trees in the forest.
- criterion: string, optional (default= “gini”)The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. Note: this parameter is tree-specific.
- max_features : int, float, string or None, optional (default=”auto”)The number of features to consider when looking for the best split:
    - If int, then consider max_features features at each split.
    - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
    - If “auto”, then max_features=sqrt(n_features).
    - If “sqrt”, then max_features=sqrt(n_features) (same as “auto”).
    - If “log2”, then max_features=log2(n_features).
    - If None, then max_features=n_features.
- max_depth : integer or None, optional (default=None)The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- min_samples_split : int, float, optional (default=2)The minimum number of samples required to split an internal node:
    - If int, then consider min_samples_split as the minimum number.
    - If float, then min_samples_split is a percentage and ceil(min_samples_split, n_samples) are the minimum number of samples for each split.
- min_samples_leaf : int, float, optional (default=1)The minimum number of samples required to be at a leaf node:
    - If int, then consider min_samples_leaf as the minimum number.
    - If float, then min_samples_leaf is a percentage and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.
- min_weight_fraction_leaf : float, optional (default=0.)The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
- max_leaf_nodes : int or None, optional (default=None)Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

- min_impurity_split : float,Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Tuning max_depth

```
# GridSearchCV to find optimal n_estimators
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'max_depth': range(2, 20, 5)}

# instantiate the model
rf = RandomForestClassifier()


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

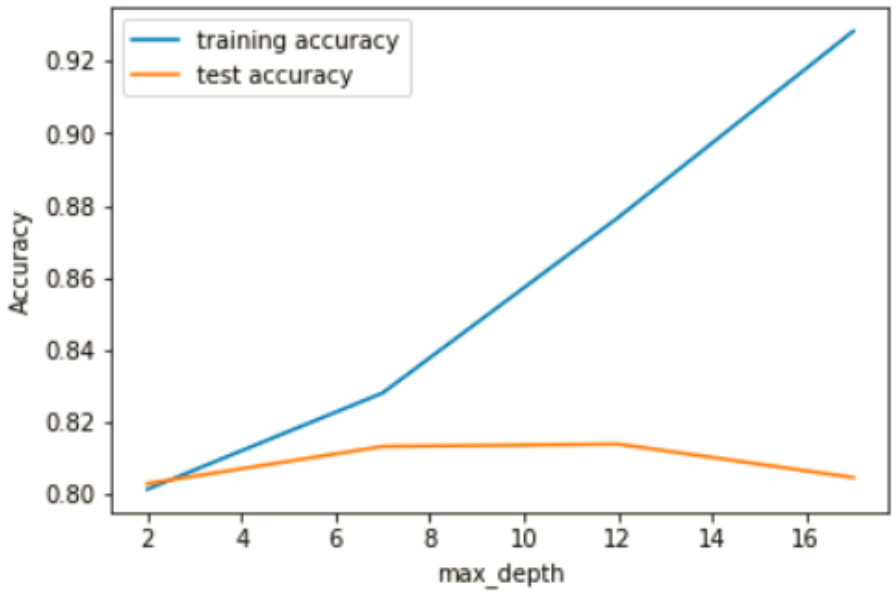

Figure 4- Tuning max_depth

You can see that as we increase the value of max_depth, both train and test scores increase till a point, but after that test score starts to decrease. The ensemble tries to overfit as we increase the max_depth.

Thus, controlling the depth of the constituent trees will help reduce overfitting in the forest.

Tuning n_estimators

```
# GridSearchCV to find optimal n_estimators
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'n_estimators': range(100, 1500, 400)}

# instantiate the model (note we are specifying a max_depth)
rf = RandomForestClassifier(max_depth=4)


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

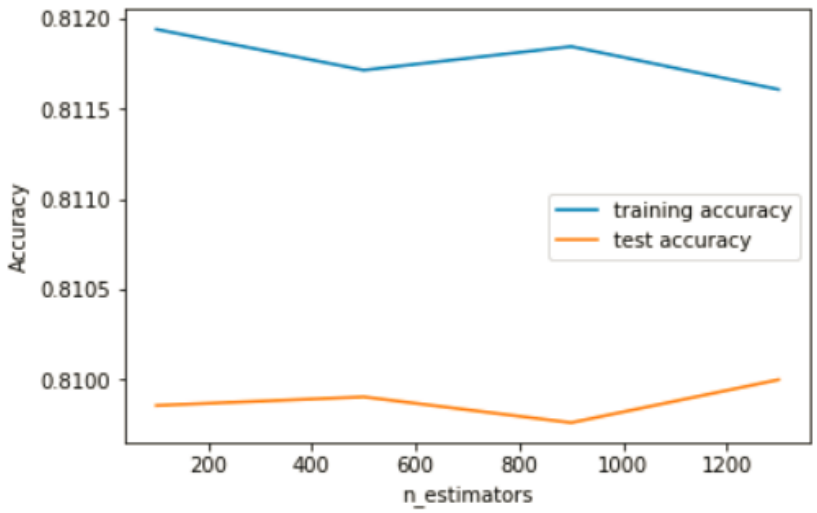

Figure 5- Tuning n_estimators

Tuning max_features

```
# GridSearchCV to find optimal max_features
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'max_features': [4, 8, 14, 20, 24]}

# instantiate the model
rf = RandomForestClassifier(max_depth=4)


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

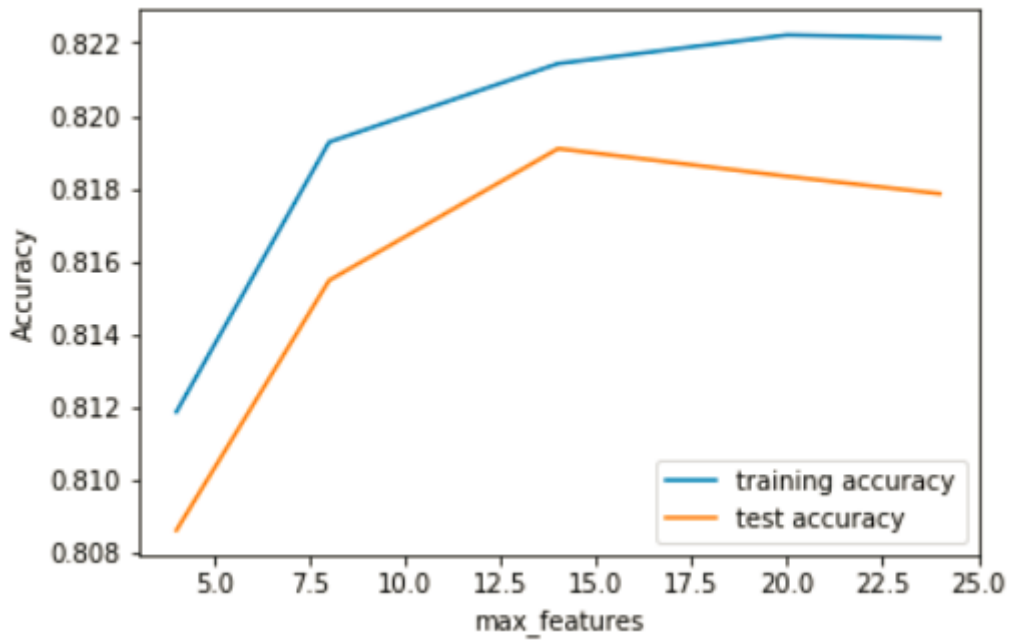

Figure 6-Tuning max_features

Grid Search to Find Optimal Hyperparameters

We can now find the optimal hyperparameters using GridSearchCV.

```
# Create the parameter grid based on the results of random search
param_grid = {
    'max_depth': [4,8,10],
    'min_samples_leaf': range(100, 400, 200),
    'min_samples_split': range(200, 500, 200),
    'n_estimators': [100,200, 300],
    'max_features': [5, 10]
}
# Create a based model
rf = RandomForestClassifier()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                           cv = 3, n_jobs = -1,verbose = 1)
# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# printing the optimal accuracy score and hyperparameters
print('We can get accuracy
of',grid_search.best_score_,'using',grid_search.best_params_)

#We can get accuracy of 0.818285714286 using {'max_features': 10, 'n_estimators': 200,
'max_depth': 8, 'min_samples_split': 200, 'min_samples_leaf': 100}
```

Fitting the final model with the best parameters obtained from grid search.

```
# model with the best hyperparameters
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(bootstrap=True,
                             max_depth=10,
                             min_samples_leaf=100,
                             min_samples_split=200,
                             max_features=10,
                             n_estimators=100)
# fit
rfc.fit(X_train,y_train)
# predict
```

```
predictions = rfc.predict(X_test)
# evaluation metrics
from sklearn.metrics import classification_report,confusion_matrix
print(classification_report(y_test,predictions))
print(confusion_matrix(y_test,predictions))
```