Ensemble Methods: Random Forest

1. Theoretical Motivation

The bagging procedure can be further enhanced by introducing random forests. As we discussed in bagging, with a set of *n* **independent** (or uncorrelated) observations $Z_1, Z_2, \cdots, Z_n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by $\frac{\sigma^2}{n}$. **In other words, averaging a set of observations reduces variance.**

However, if bootstrapped data are using instead, the observations $Z_1, Z_2, \cdots, Z_b$ will be correlated. In particular, they are identically distributed (but **not independent**) with some positive correlation ρ. It then holds that:

$$\bar{Z}_b = \rho\sigma^2 + \sigma^2 \frac{(1-\rho)}{b}$$

Q: So what?
A: While the second term goes to zero as the number of observations b increase, the first term remains constant.

This issue is particularly relevant for bagging with decision trees. For example, consider a situation in which there exists a feature that provides a very good split of the data. What would happen?

A: Such feature will be selected and split for every $g^{(*b)}$ at the root level and consequently we will end up with highly correlated predictions.

The major idea of **random forests** is to perform bagging in combination with a "**decorrelation**" of the trees by including only **a subset of randomly selected features** during the tree construction. This simple but powerful idea will decorrelate the trees, since the strong features might not be included in the subset.

Remark: In random forests, p features is chosen as split candidates from the full set of d features. Typically, we choose $p \approx \sqrt{d}$. For example, if we have 13 features in total, each time we will only use 4 randomly selected from the 13 features and only use 1 from the 4 features to spilt the tree.

2. Python Example: Random Forest for Classifying Digits

To demonstrate the capability of random forests, let's consider one piece of the optical character recognition problem: **the identification of hand-written digits**. In the wild, this problem involves both locating and identifying characters in an image. Here we'll take a shortcut and use `Scikit-Learn`'s set of pre-formatted digits, which is built into the library.

Loading and visualizing the digits data

We'll use `Scikit-Learn`'s data access interface and take a look at this data:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.images.shape
```

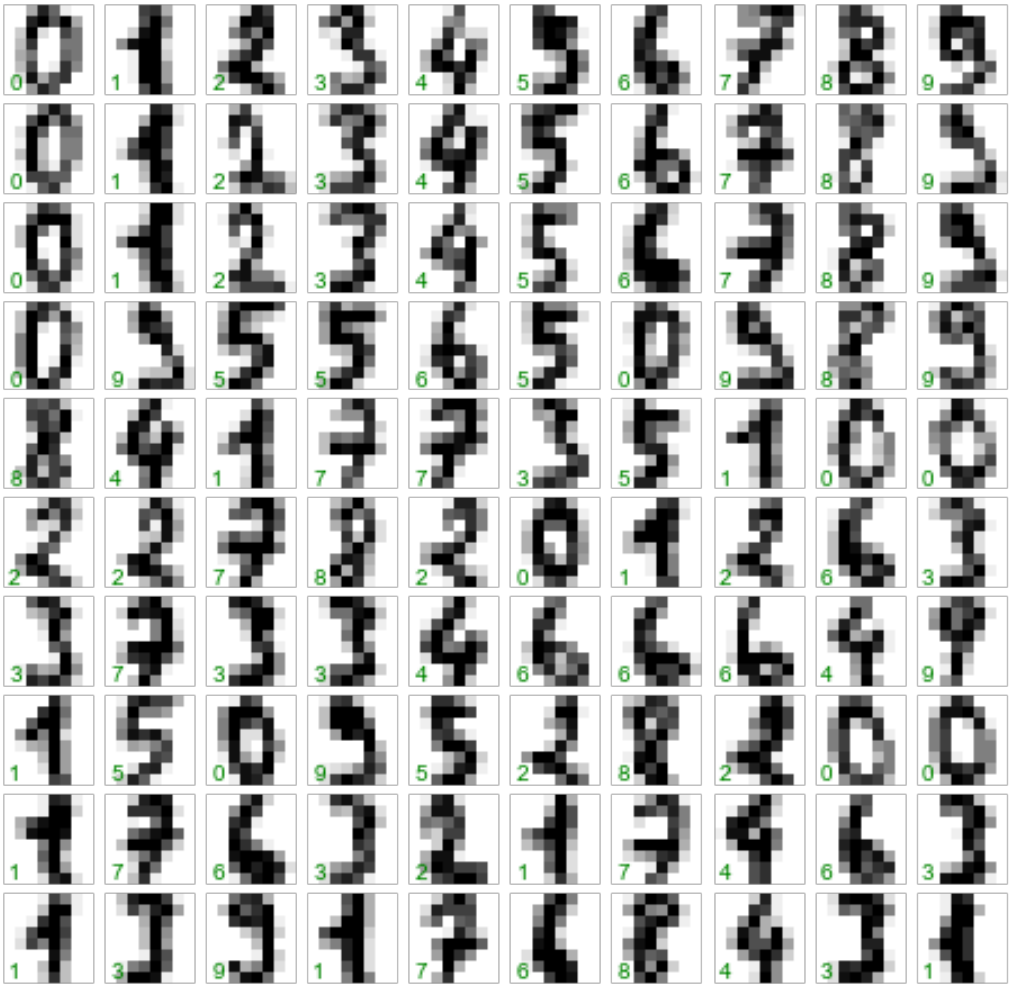
```
(1797, 8, 8)
```

The images data is a three-dimensional array: 1,797 samples each consisting of an 8 × 8 grid of pixels. Let's visualize the first hundred of these:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
```



In order to work with this data within `Scikit-Learn`, we need a two-dimensional, `[n_samples, n_features]` representation. We can accomplish this by treating each pixel in the image as a feature: that is, by flattening out the pixel arrays so that we have a length-64 array of pixel values representing each digit. Additionally, we need the target array, which gives the previously determined label for each digit. These two quantities are built into the digits dataset under the data and target attributes, respectively:

```
X = digits.data
X.shape
```

```
(1797, 64)
```

```
y = digits.target
y.shape
```

```
(1797,)
```

We see here that there are 1,797 samples and 64 features.

We can quickly classify the digits using a random forest as follows:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)
ypred = model.predict(Xtest)
```

We can take a look at the classification report for this classifier:

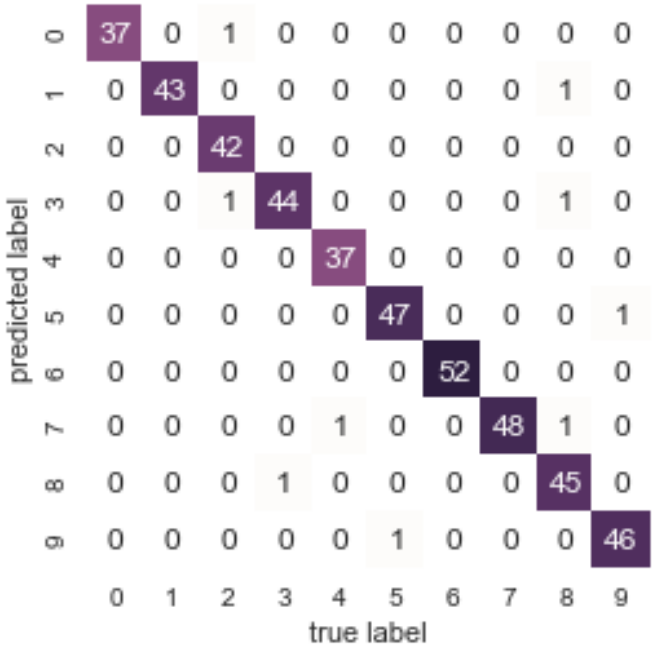
```
from sklearn import metrics
print(metrics.classification_report(ytest, ypred))
```

	precision	recall	f1-score	support
0	0.97	1.00	0.99	37
1	0.95	0.98	0.97	43
2	1.00	0.95	0.98	44
3	0.96	0.98	0.97	45
4	1.00	0.97	0.99	38
5	0.96	0.98	0.97	48
6	1.00	1.00	1.00	52
7	0.96	1.00	0.98	48
8	0.98	0.94	0.96	48
9	0.98	0.96	0.97	47
avg / total	0.98	0.98	0.98	450

And for good measure, plot the confusion matrix:

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, ypred)
cmap = sns.cubehelix_palette(light=1, as_cmap=True)
sns.heatmap(mat.T, square=True, cmap=cmap, annot=True, fmt='d',
cbar=False)
plt.xlabel('true label')
plt.ylabel('predicted label');
```



We find that a simple, untuned random forest results in a very accurate classification of the digits data.

3. Feature Importance

One great quality of Random Forests (and other ensemble tree methods) is that they make it easy to measure the relative importance of each feature (or attribute). `Scikit-Learn` measures a feature's importance by looking at how much the tree nodes that use that feature to reduce impurity on average (across all trees in the forest).

Fun Time: If a feature can reduce more impurity, the feature is more important. (1) Yes (2) No

`Scikit-Learn` computes this score automatically for each feature after training, then it scales the results so that the sum of all importance is equal to 1. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the iris dataset and outputs each feature's importance.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
```

```
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rnd_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"],
rnd_clf.feature_importances_):
    print(name, score)
```

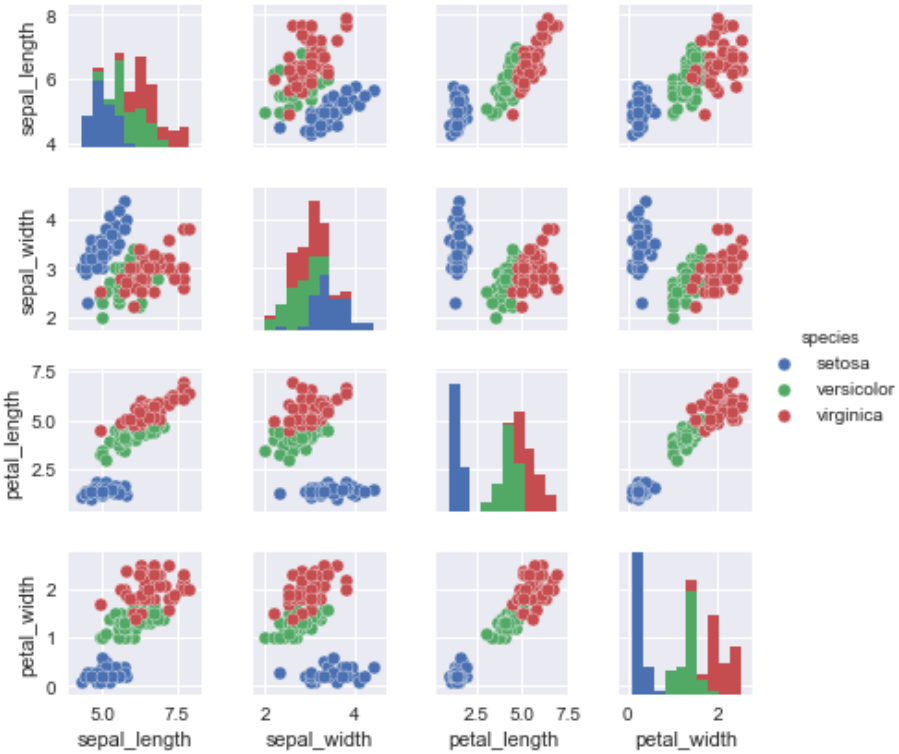
```
sepal length (cm) 0.107845222345
sepal width (cm) 0.0258824377512
petal length (cm) 0.435021720329
petal width (cm) 0.431250619575
```

We observe that the most important features are the petal length (43.5%) and width (43.1%), while sepal length and width are rather unimportant in comparison (10.8% and 2.6%, respectively).

Remark: For this simple case, we can also observe the feature importance directly from visualizing the data. Visualizing the multidimensional relationships among the samples is as easy as calling `sns.pairplot`:

```
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', height=1.5);
```

We can observe that the classification ability of `petal_length` and `pedal_width` is better than `sepal_length` and `sepal_width`, same as the conclusion drawn from feature importance of the Random Forest classifier.

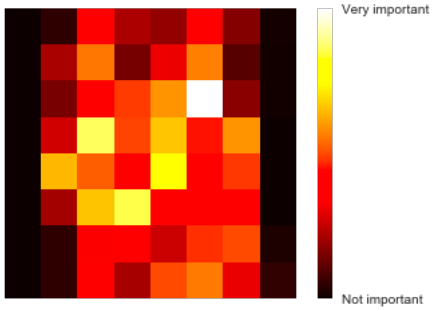


Similarly, if you plot each pixel's importance of a Random Forest classifier on the previous example of digit images with 8x8 pixels, you get the image represented in the figure below.

```
import matplotlib
def plot_digit(data):
    image = data.reshape(8, 8)
    plt.imshow(image, cmap = matplotlib.cm.hot,
               interpolation="nearest")
    plt.axis("off")

plot_digit(model.feature_importances_)

cbar = plt.colorbar(ticks=[model.feature_importances_.min(),
model.feature_importances_.max()])
cbar.ax.set_yticklabels(['Not important', 'Very important'])
```



Remark: Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

You can download the source codes `RFDigitsRecog.ipynb` and `RFFeatures.ipynb` to reproduce these two examples from the course website.