Lab 14 - Decision Trees in Python

April 6, 2016

This lab on Decision Trees is a Python adaptation of p. 324-331 of "Introduction to Statistical Learning with Applications in R" by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. Original adaptation by J. Warmenhoven, updated by R. Jordan Crouser at Smith College for SDS293: Machine Learning (Spring 2016).

```
In [ ]: import pandas as pd
        import numpy as np
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        import graphviz

        %matplotlib inline
```

1 8.3.1 Fitting Classification Trees

The `sklearn` library has a lot of useful tools for constructing classification and regression trees:

```
In [ ]: from sklearn.cross_validation import train_test_split
        from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier, export_graphviz
        from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
        from sklearn.metrics import confusion_matrix, mean_squared_error
```

We'll start by using **classification trees** to analyze the `Carseats` data set. In these data, `Sales` is a continuous variable, and so we begin by converting it to a binary variable. We use the `ifelse()` function to create a variable, called `High`, which takes on a value of `Yes` if the `Sales` variable exceeds 8, and takes on a value of `No` otherwise. We'll append this onto our dataFrame using the `.map()` function, and then do a little data cleaning to tidy things up:

```
In [ ]: df3 = pd.read_csv('Carseats.csv').drop('Unnamed: 0', axis=1)
        df3['High'] = df3.Sales.map(lambda x: 1 if x>8 else 0)
        df3.ShelveLoc = pd.factorize(df3.ShelveLoc)[0]
        df3.Urban = df3.Urban.map({'No':0, 'Yes':1})
        df3.US = df3.US.map({'No':0, 'Yes':1})
        df3.info()
```

In order to properly evaluate the performance of a classification tree on the data, we must estimate the test error rather than simply computing the training error. We first split the observations into a training set and a test set:

```
In [ ]: X = df3.drop(['Sales', 'High'], axis=1)
        y = df3.High

        X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.5, random_state=0)
```

We now use the `DecisionTreeClassifier()` function to fit a classification tree in order to predict `High` using all variables but `Sales` (that would be a little silly...). Unfortunately, manual pruning is not implemented in `sklearn`: http://scikit-learn.org/stable/modules/tree.html

However, we can limit the depth of a tree using the `max_depth` parameter:

```
In [ ]: clf = DecisionTreeClassifier(max_depth=6)
        clf.fit(X_train, y_train)
        clf.score(X_train, y_train)
```

We see that the training accuracy is 95.5%.

One of the most attractive properties of trees is that they can be graphically displayed. Unfortunately, this is a bit of a roundabout process in `sklearn`. We use the `export_graphviz()` function to export the tree structure to a temporary `.dot` file, and the `graphviz.Source()` function to display the image:

```
In [ ]: export_graphviz(clf, out_file="mytree.dot", feature_names=X_train.columns)
        with open("mytree.dot") as f:
            dot_graph = f.read()
        graphviz.Source(dot_graph)
```

The most important indicator of `High` sales appears to be `Price`.

Finally, let's evaluate the tree's performance on the test data. The `predict()` function can be used for this purpose. We can then build a confusion matrix, which shows that we are making correct predictions for around 74.5% of the test data set:

```
In [ ]: pred = clf.predict(X_test)
        cm = pd.DataFrame(confusion_matrix(y_test, pred).T, index=['No', 'Yes'], columns=['No', 'Yes'])
        print(cm)
        # 99+50/200 = 0.745
```

2 8.3.2 Fitting Regression Trees

Now let's try fitting a **regression tree** to the `Boston` data set from the `MASS` library. First, we create a training set, and fit the tree to the training data using `medv` (median home value) as our response:

```
In [ ]: boston_df = pd.read_csv('Boston.csv')
        X = boston_df.drop('medv', axis=1)
        y = boston_df.medv
        X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.5, random_state=0)

        # Pruning not supported. Choosing max depth 2)
        regr2 = DecisionTreeRegressor(max_depth=2)
        regr2.fit(X_train, y_train)
```

Let's take a look at the tree:

```
In [ ]: export_graphviz(regr2, out_file="mytree.dot", feature_names=X_train.columns)
        with open("mytree.dot") as f:
            dot_graph = f.read()
        graphviz.Source(dot_graph)
```

The variable `lstat` measures the percentage of individuals with lower socioeconomic status. The tree indicates that lower values of `lstat` correspond to more expensive houses. The tree predicts a median house price of $45,766 for larger homes (`rm` >= 7.435) in suburbs in which residents have high socioeconomic status (`lstat` < 7.81).

Now let's see how it does on the test data:

```
In [ ]: pred = regr2.predict(X_test)

        plt.scatter(pred, y_test, label='medv')
        plt.plot([0, 1], [0, 1], '--k', transform=plt.gca().transAxes)
        plt.xlabel('pred')
        plt.ylabel('y_test')

        mean_squared_error(y_test, pred)
```

The test set MSE associated with the regression tree is 28.8. The square root of the MSE is therefore around 5.37, indicating that this model leads to test predictions that are within around $5,370 of the true median home value for the suburb.

3 8.3.3 Bagging and Random Forests

Let's see if we can improve on this result using **bagging** and **random forests**. The exact results obtained in this section may depend on the version of `python` and the version of the `RandomForestRegressor` package installed on your computer, so don't stress out if you don't match up exactly with the book. Recall that **bagging** is simply a special case of a **random forest** with $m = p$. Therefore, the `RandomForestRegressor()` function can be used to perform both random forests and bagging. Let's start with bagging:

```
In [ ]: # Bagging: using all features
        regr1 = RandomForestRegressor(max_features=13, random_state=1)
        regr1.fit(X_train, y_train)
```

The argument `max_features` = `13` indicates that all 13 predictors should be considered for each split of the tree – in other words, that bagging should be done. How well does this bagged model perform on the test set?

```
In [ ]: pred = regr1.predict(X_test)
        plt.scatter(pred, y_test, label='medv')
        plt.plot([0, 1], [0, 1], '--k', transform=plt.gca().transAxes)
        plt.xlabel('pred')
        plt.ylabel('y_test')
        mean_squared_error(y_test, pred)
```

The test setMSE associated with the bagged regression tree is significantly lower than our single tree!

We can grow a random forest in exactly the same way, except that we'll use a smaller value of the `max_features` argument. Here we'll use `max_features` = `6`:

```
In [ ]: # Random forests: using 6 features
        regr2 = RandomForestRegressor(max_features=6, random_state=1)
        regr2.fit(X_train, y_train)

        pred = regr2.predict(X_test)
        mean_squared_error(y_test, pred)
```

The test set MSE is even lower; this indicates that random forests yielded an improvement over bagging in this case.

Using the `feature_importances_` attribute of the `RandomForestRegressor`, we can view the importance of each variable:

```
In [ ]: Importance = pd.DataFrame({'Importance':regr2.feature_importances_*100}, index=X.columns)
        Importance.sort_values(by='Importance', axis=0, ascending=True).plot(kind='barh', color='r', )
        plt.xlabel('Variable Importance')
        plt.gca().legend_ = None
```

The results indicate that across all of the trees considered in the random forest, the wealth level of the community (`lstat`) and the house size (`rm`) are by far the two most important variables.

4 8.3.4 Boosting

Now we'll use the `GradientBoostingRegressor` package to fit **boosted regression trees** to the `Boston` data set. The argument $\texttt{n}_\texttt{e}\texttt{stimators} = 500$ indicates that we want 500 trees, and the option $\texttt{interaction.depth} = 4$ limits the depth of each tree:

```
In [ ]: regr = GradientBoostingRegressor(n_estimators=500, learning_rate=0.01, max_depth=4, random_stat
        regr.fit(X_train, y_train)
```

Let's check out the feature importances again:

```
In [ ]: feature_importance = regr.feature_importances_*100
        rel_imp = pd.Series(feature_importance, index=X.columns).sort_values(inplace=False)
        rel_imp.T.plot(kind='barh', color='r', )
        plt.xlabel('Variable Importance')
        plt.gca().legend_ = None
```

We see that `lstat` and `rm` are again the most important variables by far. Now let's use the boosted model to predict `medv` on the test set:

```
In [ ]: mean_squared_error(y_test, regr.predict(X_test))
```

The test MSE obtained is similar to the test MSE for random forests and superior to that for bagging. If we want to, we can perform boosting with a different value of the shrinkage parameter λ. Here we take $\lambda = 0.2$:

```
In [ ]: regr2 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.2, max_depth=4, random_stat
        regr2.fit(X_train, y_train)
        mean_squared_error(y_test, regr2.predict(X_test))
```

In this case, using $\lambda = 0.2$ leads to a slightly lower test MSE than $\lambda = 0.01$.

To get credit for this lab, post your responses to the following questions: - What's one real-world scenario where you might try using Bagging? - What's one real-world scenario where you might try using Random Forests? - What's one real-world scenario where you might try using Boosting?

to Piazza: https://piazza.com/class/igwiv4w3ctb6rg?cid=53