Doing Basic Classifications with Decision Trees

- In this section we will perform basic classifications using Decision Trees.
- These are very nice models because they are easily understandable, and once trained in, scoring is very simple.
- Often, SQL statements can be used, which means that the outcome can be used by a lot of people.

Getting Ready

Decision Trees are the base class from which a large number of other classification methods are derived. It's a pretty simple idea that works well in a variety of situations.

Getting Data

First, let's get some classification data that we can practice on:

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(n_samples=1000,
   n_features=3, n_redundant=0)
```

0.1 Implementation

Working with Decision Trees is easy. We first need to import the object, and then fit the model:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier()
>>> dt.fit(X, y)
```

```
In [1]: from sklearn import datasets
        X, y = datasets.make_classification(n_samples=1000, n_features=3,
        n_redundant=0)

In [2]: from sklearn.tree import DecisionTreeClassifier
        dt = DecisionTreeClassifier()
        dt.fit(X, y)

Out[2]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                    max_features=None, max_leaf_nodes=None, min_samples_leaf=1,
                    min_samples_split=2, min_weight_fraction_leaf=0.0,
                    random_state=None, splitter='best')
```

```
  DecisionTreeClassifier(compute_importances=None, criterion="gini"
  max_depth=None, max_features=None,
  max_leaf_nodes=None, min_density=None,
  min_samples_leaf=1, min_samples_split=2,
  random_state=None, splitter='best')

>>> preds = dt.predict(X)
>>> (y == preds).mean()
   1.0
```

First, if you look at the `dt` object, it has several keyword arguments that determine how the object will behave. How we choose the object is important, so we'll look at the object's effects in detail.

Maximum Depth

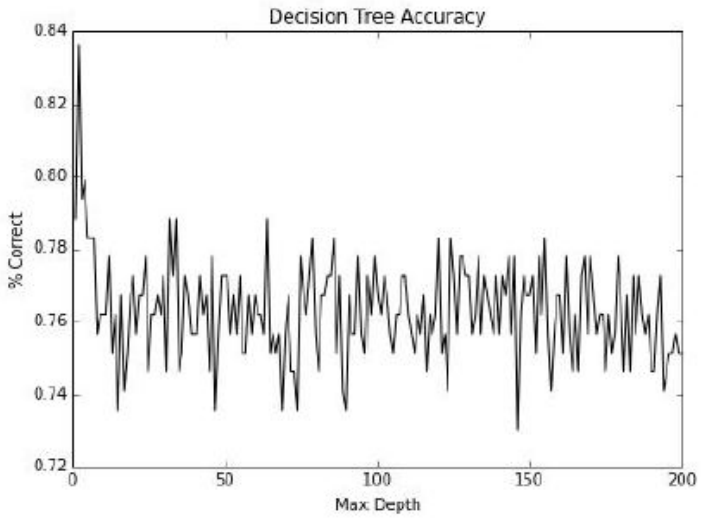
- The first detail we'll look at is `max_depth`. This is an important parameter. It determines how many branches are allowed.
- This is important because a Decision Tree can have a hard time generalizing out-of-sampled data with some sort of regularization.
- Later, we'll see how we can use several shallow Decision Trees to make a better learner.

- Let's create a more complex dataset and see what happens when we allow different `max_depth`. We'll use this dataset for the rest of the exercise:

```
>>> n_features=200
>>> X, y = datasets.make_classification(750,
n_features,n_informative=5)
>>> import numpy as np
>>>
>>> training = np.random.choice([True, False],
p=[.75, .25],size=len(y))
>>
>>> accuracies = []
>>>
>>> for x in np.arange(1, n_features+1):
>>> dt = DecisionTreeClassifier(max_depth=x)
>>> dt.fit(X[training], y[training])
>>> preds = dt.predict(X[~training])
>>> accuracies.append((preds == y[~training]).mean())
```

```
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.plot(range(1, n_features+1), accuracies, color='k')
>>> ax.set_title("Decision Tree Accuracy")
>>> ax.set_ylabel("% Correct")
>>> ax.set_xlabel("Max Depth")
```

We can see that we actually get pretty accurate at a low max depth. Let's take a closer look at the accuracy at low levels, say the first 15:




```
>>> N = 15
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.plot(range(1, n_features+1)[:N], accuracies[:N], color='k')
>>> ax.set_title("Decision Tree Accuracy")
>>> ax.set_ylabel("% Correct")
>>> ax.set_xlabel("Max Depth")
```

The following is the output:

There's the spike we saw earlier; it is interesting to see the quick drop though. It's more likely that Max Depth of 1 through 3 is fairly equivalent. Decision Trees are quite good at separating rules, but they need to be reigned in. We'll look at the `compute_importances` parameter here. It actually has a bit of a broader meaning for random forests, but we'll get acquainted with it.

```
>>> dt_ci = DecisionTreeClassifier(compute_importances=True)
```

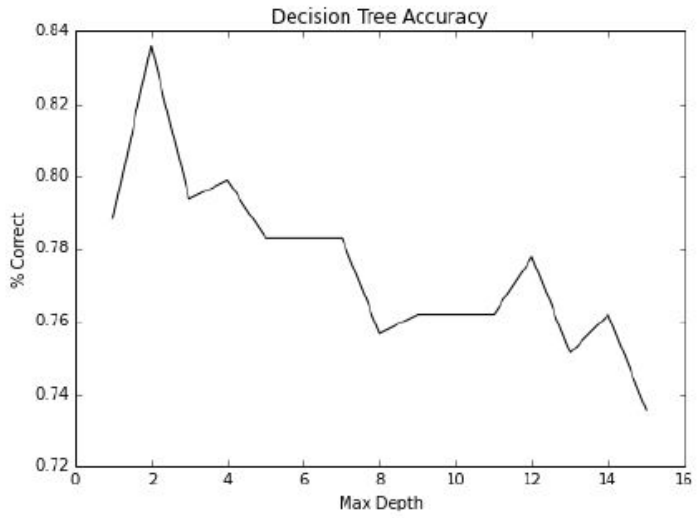


Figure 1

```
>>> dt.fit(X, y)
```

Plot the importances

```
>>> ne0 = dt.feature_importances_ != 0
>>> y_comp = dt.feature_importances_[ne0]
>>> x_comp = np.arange(len(dt.feature_importances_))[ne0]
>>>
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.bar(x_comp, y_comp)
```

The following is the output:

- Please note that you may get an error letting you know you'll no longer need to explicitly set compute importances.

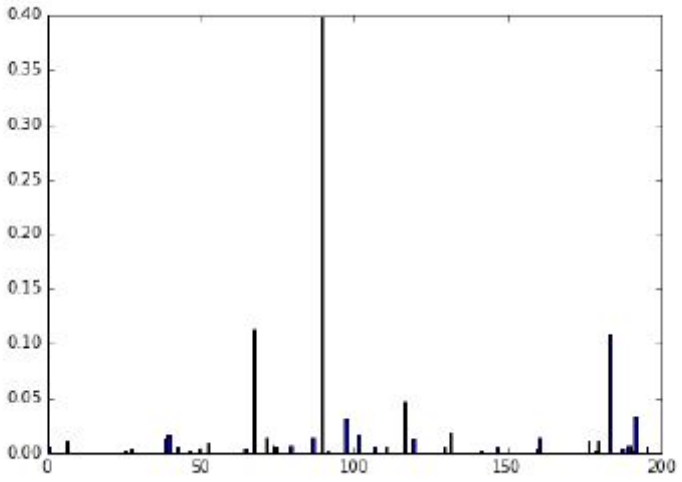


Figure 2

- As we can see, one of the features is by far the most important; several other features will follow up.

Theoretical Aspects

- In the simplest sense, we construct Decision Trees all the time.
- When thinking through situations and assigning probabilities to outcomes, we construct Decision Trees.
- Our rules are much more complex and involve a lot of context, but with Decision Trees, all we care about is the difference between outcomes, given that some information is already known about a feature.

Information Gain and Gini Impurity

- Now, let's discuss the differences between entropy and Gini impurity.
- Entropy is more than just the entropy value at any given variable; it states what the change in entropy is if we know an element's value.
- This is called Information Gain (IG); mathematically it looks like the following:

  $$\text{IG(Data, Known Features)} = H(Data) - H(Data|KnownFeatures)$$

- For Gini impurity, we care about how likely one of the data points will be mislabeled given the new information.
- Both entropy and Gini impurity have pros and cons; this said, if you see major differences in the working of entropy and Gini impurity, it will probably be a good idea to re-examine your assumptions.

Tuning a Decision Tree model

If we use just the basic implementation of a Decision Tree, it will probably not fit very well. Therefore, we need to tweak the parameters in order to get a good fit. This is very easy and won't require much effort.

Getting Ready

- In this exercise, we will take an in-depth look at what it takes to tune a Decision Tree classifier.
- There are several options, and in the previous recipe, we only looked at one of these options.
- We'll fit a basic model and actually look at what the Decision Tree looks like.
- Then, we'll re-examine after each decision and point out how various changes have influenced the structure.
- If you want to follow along in this exercise, you'll need to install **pydot**.

Implementation

Decision Trees have a lot more "knobs" when compared to most other algorithms, because of which it's easier to see what happens when we turn the knobs:

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(1000, 20, n_informative=3)
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier()
>>> dt.fit(X, y)
```

Ok, so now that we have a basic classifier fit, we can view it quite simply:

```
>>> from StringIO import StringIO
>>> from sklearn import tree
>>> import pydot
>>> str_buffer = StringIO()
>>> tree.export_graphviz(dt, out_file=str_buffer)
>>> graph = pydot.graph_from_dot_data(str_buffer.getvalue())
>>> graph.write("myfile.jpg")
```

The graph is almost certainly illegible, but hopefully this illustrates the complex trees that can be generated as a result of using an unoptimized decision tree: This is a very complex tree. It will most likely overfit the data. First, let's reduce the max depth value:

```
>>> dt = DecisionTreeClassifier(max_depth=5)
>>> dt.fit(X, y);
```

As an aside, if you're wondering why the semicolon, the repr by default, is seen, it is actually the model for a Decision Tree. For example, the fit function actually returns the Decision Tree object that allows chaining:

```
>>> dt = DecisionTreeClassifier(max_depth=5).fit(X, y)
```

Now, let's get back to the regularly scheduled program. As we will plot this a few times, let's create a function:

```
>>> def plot_dt(model, filename):
str_buffer = StringIO()
>>> tree.export_graphviz(model, out_file=str_buffer)
>>> graph = pydot.graph_from_dot_data(str_buffr.getvalue())
>>> graph.write_jpg(filename)
>>> plot_dt(dt, "myfile.png")
```

The following is the graph that will be generated: This is a much simpler tree. Let's look at what happens when we use entropy as the splitting criteria:

```
>>> dt = DecisionTreeClassifier(criterion='entropy',
max_depth=5).fit(X, y)
>>> plot(dt, "entropy.png")
```

The following is the graph that can be generated: It's good to see that the first two splits are the same features, and the first few after this are interspersed with similar amounts. This is a good sanity check. Also, note how entropy for the first split is 0.999, but for the first split when using the Gini impurity is 0.5. This has to do with how different the two measures of the split of a Decision Tree are.

However, if we want to create a Decision Tree with entropy, we must use the following command:

```
>>> dt = DecisionTreeClassifier(min_samples_leaf=10,
criterion='entropy',
max_depth=5).fit(X, y)
```

Overfitting

- Decision Trees, in general, suffer from overfitting. Quite often, left to it's own devices, a Decision Tree model will overfit, and therefore, we need to think about how best to avoid overfitting; this is done to avoid complexity.
- A simple model will more often work better in practice than not.
- We're about to see this very idea in practice. random forests will build on this idea of simple models.