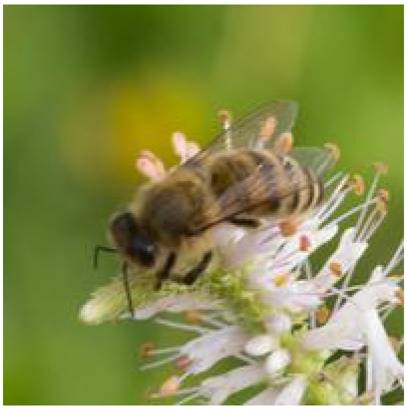
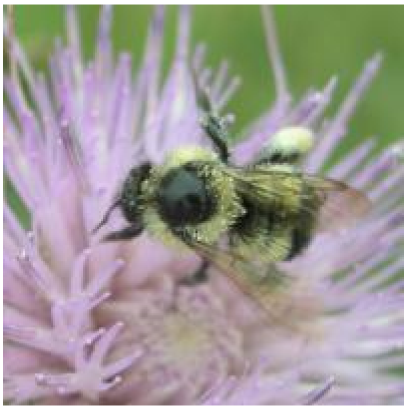
1. Import Python libraries



A honey bee (Apis).

Can a machine identify a bee as a honey bee or a bumble bee? These bees have different behaviors and appearances (https://www.thesca.org/connect/blog/bumblebees-vs-honeybees-what%E2%80%99s-difference-and-why-does-it-matter), but given the variety of backgrounds, positions, and image resolutions, it can be a challenge for machines to tell them apart.

Being able to identify bee species from images is a task that ultimately would allow researchers to more quickly and effectively collect field data. Pollinating bees have critical roles in both ecology and agriculture, and diseases like colony collapse disorder (http://news.harvard.edu/gazette/story/2015/07/pesticide-found-in-70-percent-of-massachusetts-honey-samples/) threaten these species. Identifying different species of bees in the wild means that we can better understand the prevalence and growth of these important insects.



A bumble bee (Bombus).

After loading and pre-processing images, this notebook walks through building a model that can automatically detect honey bees and bumble bees.

In [2]:

```
# used to change filepaths
import os

import matplotlib as mpl
import matplotlib.pyplot as plt
from IPython.display import display
%matplotlib inline

import pandas as pd
import numpy as np

# import Image from PIL
from PIL import Image

from skimage.feature import hog
from skimage.color import rgb2grey

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# import train_test_split from sklearn's model selection module
from sklearn.model_selection import train_test_split

# import SVC from sklearn's svm module
from sklearn.svm import SVC

# import accuracy_score from sklearn's metrics module
from sklearn.metrics import roc_curve, auc, accuracy_score
```

2. Display image of each bee type

Now that we have all of our imports ready, it is time to look at some images. We will load our `labels.csv` file into a dataframe called `labels` , where the index is the image name (e.g. an index of 1036 refers to an image named 1036.jpg) and the `genus` column tells us the bee type. `genus` takes the value of either `0.0` (Apis or honey bee) or `1.0` (Bombus or bumble bee).

The function `get_image` converts an index value from the dataframe into a file path where the image is located, opens the image using the [Image (https://pillow.readthedocs.io/en/5.1.x/reference/Image.html)](https://pillow.readthedocs.io/en/5.1.x/reference/Image.html) object in Pillow, and then returns the image as a numpy array.

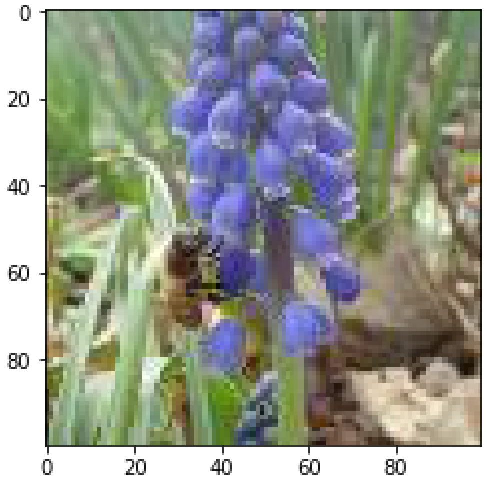
We'll use this function to load the sixth Apis image and then the sixth Bombus image in the dataframe.

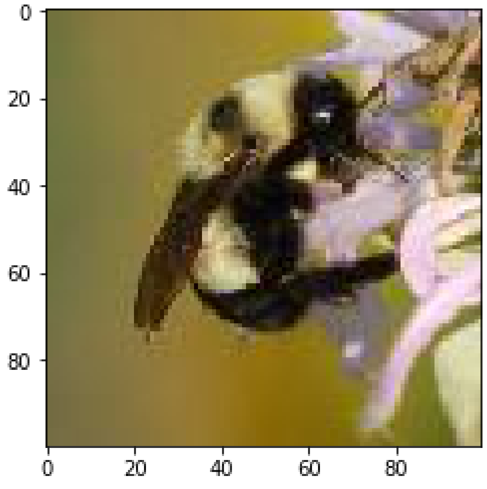
In [4]:

```
# load the labels using pandas
labels = pd.read_csv("datasets/labels.csv", index_col=0)

# show the first five rows of the dataframe using head
display(labels.head)

def get_image(row_id, root="datasets/"):
    """
    Converts an image number into the file path where the image is located,
    opens the image, and returns the image as a numpy array.
    """
    filename = "{}.jpg".format(row_id)
    file_path = os.path.join(root, filename)
    img = Image.open(file_path)
    return np.array(img)

# subset the dataframe to just Apis (genus is 0.0) get the value of the sixth item
apis_row = labels[labels.genus == 0.0].index[5]

# show the corresponding image of an Apis
plt.imshow(get_image(apis_row))
plt.show()

# subset the dataframe to just Bombus (genus is 1.0) get the value of the sixth ite
bombus_row = labels[labels.genus == 1.0].index[5]

# show the corresponding image of a Bombus
plt.imshow(get_image(bombus_row))
plt.show()
```

```
<bound method NDFrame.head of       genus
id
520     1.0
3800    1.0
3289    1.0
2695    1.0
4922    1.0
1934    1.0
3704    1.0
4017    1.0
3331    1.0
173     1.0
2983    1.0
116     1.0
4938    1.0
3524    1.0
346     1.0
631     1.0
3588    1.0
3321    1.0
2237    1.0
1180    1.0
1725    1.0
4067    1.0
2675    1.0
728     1.0
1815    1.0
1722    1.0
1377    1.0
```

```
432     1.0
1506    1.0
1705    1.0
...     ...
3229    0.0
2157    0.0
3926    0.0
3515    0.0
4645    0.0
1106    0.0
2851    0.0
630     0.0
938     0.0
2794    0.0
4612    0.0
3644    0.0
3001    0.0
3464    0.0
4618    0.0
2142    0.0
1087    0.0
2220    0.0
701     0.0
1279    0.0
134     0.0
3680    0.0
3987    0.0
2113    0.0
3015    0.0
650     0.0
3782    0.0
4026    0.0
1347    0.0
807     0.0

[500 rows x 1 columns]>
```





3. Image manipulation with rgb2grey

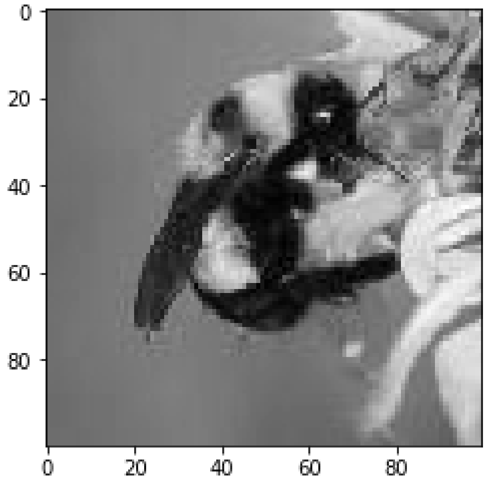
scikit-image has a number of image processing functions built into the library, for example, converting an image to greyscale. The `rgb2grey` (http://scikit-image.org/docs/dev/api/skimage.color.html#skimage.color.rgb2grey) function computes the luminance of an RGB image using the following formula `Y = 0.2125 R + 0.7154 G + 0.0721 B` .

Image data is represented as a matrix, where the depth is the number of channels. An RGB image has three channels (red, green, and blue) whereas the returned greyscale image has only one channel. Accordingly, the original color image has the dimensions `100x100x3` but after calling `rgb2grey` , the resulting greyscale image has only one channel, making the dimensions `100x100x1` .

In [6]:

```
# load a bombus image using our get_image function and bombus_row from the previous
bombus = get_image(bombus_row)

# print the shape of the bombus image
print('Color bombus image has shape: ', bombus.shape)

# convert the bombus image to greyscale
grey_bombus = rgb2grey(bombus)

# show the greyscale image
plt.imshow(grey_bombus, cmap=mpl.cm.gray)

# greyscale bombus image only has one channel
print('Greyscale bombus image has shape: ', ...)
```

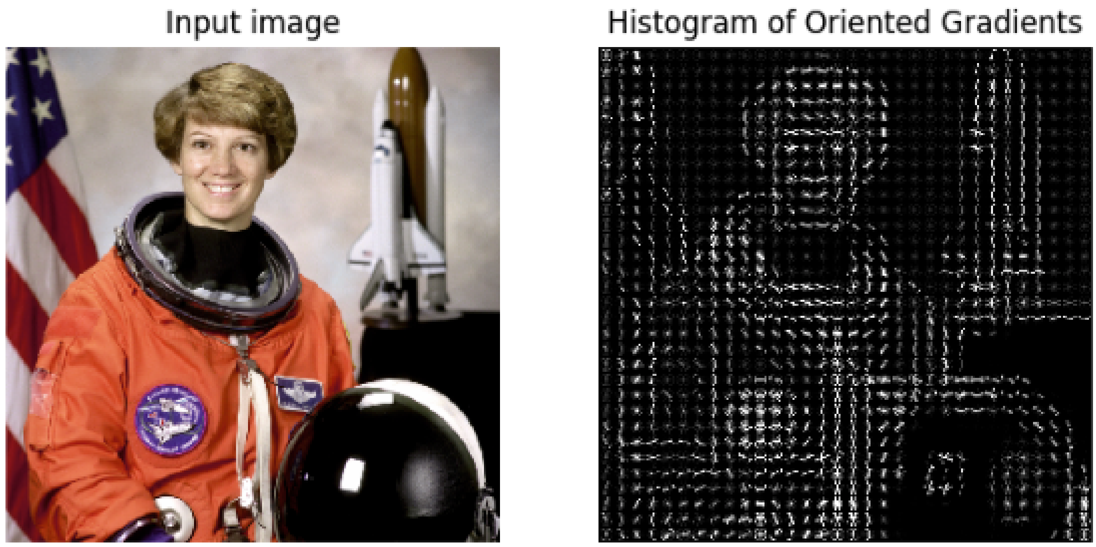
```
Color bombus image has shape:  (100, 100, 3)
Greyscale bombus image has shape:  Ellipsis
```



4. Histogram of oriented gradients

Now we need to turn these images into something that a machine learning algorithm can understand. Traditional computer vision techniques have relied on mathematical transforms to turn images into useful features. For example, you may want to detect edges of objects in an image, increase the contrast, or filter out particular colors.

We've got a matrix of pixel values, but those don't contain enough interesting information on their own for most algorithms. We need to help the algorithms along by picking out some of the salient features for them using the histogram of oriented gradients (http://scikit-image.org/docs/dev/auto_examples/features_detection/plot_hog.html) (HOG) descriptor. The idea behind HOG (https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients) is that an object's shape within an image can be inferred by its edges, and a way to identify edges is by looking at the direction of intensity gradients (i.e. changes in luminescence).



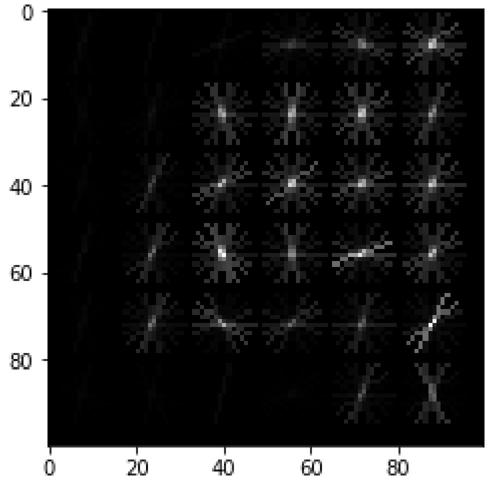
An image is divided in a grid fashion into cells, and for the pixels within each cell, a histogram of gradient directions is compiled. To improve invariance to highlights and shadows in an image, cells are block normalized, meaning an intensity value is calculated for a larger region of an image called a block and used to contrast normalize all cell-level histograms within each block. The HOG feature vector for the image is the concatenation of these cell-level histograms.

In [8]:

```
# run HOG using our greyscale bombus image
hog_features, hog_image = hog(grey_bombus,
                              visualize=True,
                              block_norm='L2-Hys',
                              pixels_per_cell=(16, 16))

# show our hog_image with a grey colormap
plt.imshow(hog_image, cmap=mpl.cm.gray)
```

Out[8]:

```
<matplotlib.image.AxesImage at 0x7f4dd784c6a0>
```



5. Create image features and flatten into a single row

Algorithms require data to be in a format where rows correspond to images and columns correspond to features. This means that all the information for a given image needs to be contained in a single row.

We want to provide our model with the raw pixel values from our images as well as the HOG features we just calculated. To do this, we will write a function called `create_features` that combines these two sets of features by flattening the three-dimensional array into a one-dimensional (flat) array.

In [10]:

```python
def create_features(img):
    # flatten three channel color image
    color_features = np.ndarray.flatten(img)
    # convert image to greyscale
    grey_image = rgb2grey(img)
    # get HOG features from greyscale image
    hog_features = hog(grey_image, block_norm='L2-Hys', pixels_per_cell=(16, 16))
    # combine color and hog features into a single array
    flat_features = np.hstack((color_features,hog_features))
    return flat_features

bombus_features = create_features(bombus)

# print shape of bombus_features
print(bombus_features.shape)
```

```
(31296,)
```

6. Loop over images to preprocess

Above we generated a flattened features array for the bombus image. Now it's time to loop over all of our images. We will create features for each image and then stack the flattened features arrays into a big matrix we can pass into our model.

In the `create_feature_matrix` function, we'll do the following:

- Load an image
- Generate a row of features using the `create_features` function above
- Stack the rows into a features matrix

In the resulting features matrix, rows correspond to images and columns to features.

In [12]:

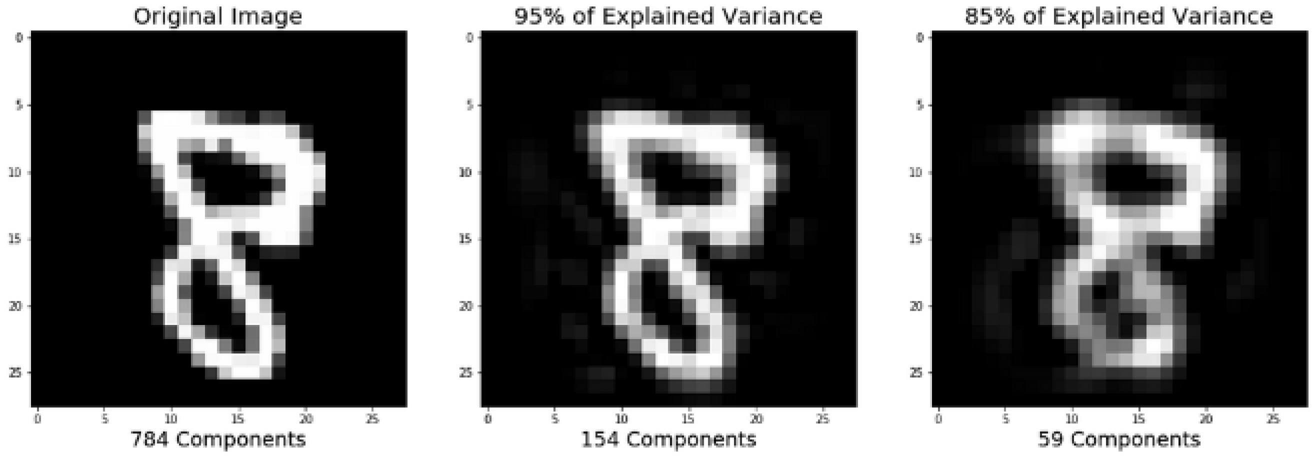
```python
def create_feature_matrix(label_dataframe):
    features_list = []

    for img_id in label_dataframe.index:
        # load image
        img = get_image(img_id)
        # get features for image
        image_features = create_features(img)
        features_list.append(image_features)

    # convert list of arrays into a matrix
    feature_matrix = np.array(features_list)
    return feature_matrix

# run create_feature_matrix on our dataframe of images
feature_matrix = create_feature_matrix(labels)
```

7. Scale feature matrix + PCA

Our features aren't quite done yet. Many machine learning methods are built to work best with data that has a mean of 0 and unit variance. Luckily, scikit-learn [provides a simple way (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html)](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) to rescale your data to work well using `StandardScaler` . They've got a more thorough explanation of why that is in the linked docs.

Remember also that we have over 31,000 features for each image and only 500 images total. To use an SVM, our model of choice, we also need to reduce the number of features we have using [principal component analysis (http://scikit-learn.org/stable/modules/decomposition.html#pca)](http://scikit-learn.org/stable/modules/decomposition.html#pca) (PCA).

PCA is a way of linearly transforming the data such that most of the information in the data is contained within a smaller number of features called components. Below is a visual [example (https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60)](https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60) from an image dataset containing handwritten numbers. The image on the left is the original image with 784 components. We can see that the image on the right (post PCA) captures the shape of the number quite effectively even with only 59 components.



In our case, we will keep 500 components. This means our feature matrix will only have 500 columns rather than the original 31,296.

In [14]:

```
# get shape of feature matrix
print('Feature matrix shape is: ', feature_matrix.shape)

# define standard scaler
ss = StandardScaler()
# run this on our feature matrix
bees_stand = ss.fit_transform(feature_matrix)

pca = PCA(n_components=500)
# use fit_transform to run PCA on our standardized matrix
bees_pca = pca.fit_transform(bees_stand)
# look at new shape
print('PCA matrix shape is: ', bees_pca.shape)
```

```
Feature matrix shape is:  (500, 31296)
PCA matrix shape is:  (500, 500)
```

8. Split into train and test sets

Now we need to convert our data into train and test sets. We'll use 70% of images as our training data and test our model on the remaining 30%. Scikit-learn's `train_test_split` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) function makes this easy.

In [16]:

```
X_train, X_test, y_train, y_test = train_test_split(bees_pca,
                                                    labels.genus.values,
                                                    test_size=.3,
                                                    random_state=1234123)

# look at the distrubution of labels in the train set
pd.Series(y_train).value_counts()
```

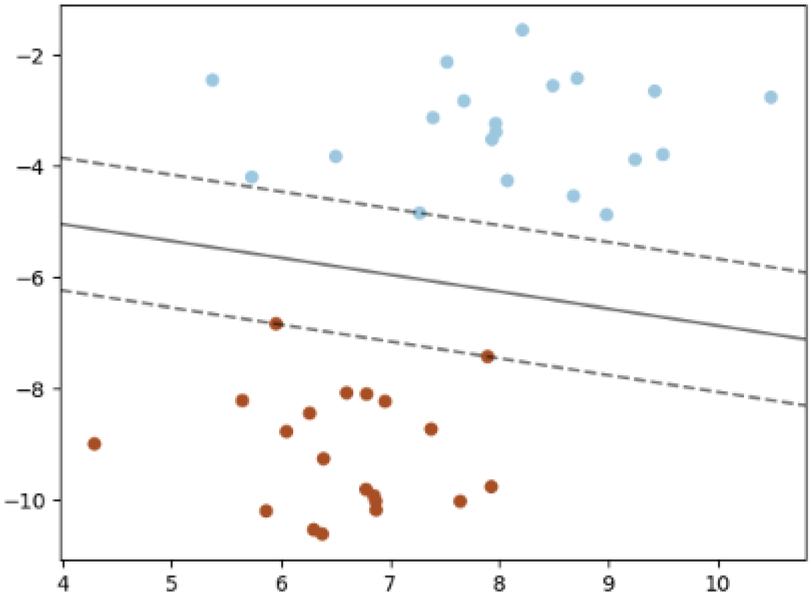
Out[16]:

```
0.0    175
1.0    175
dtype: int64
```

9. Train model

It's finally time to build our model! We'll use a support vector machine (http://scikit-learn.org/stable/modules/svm.html) (SVM), a type of supervised machine learning model used for regression, classification, and outlier detection." An SVM model (https://en.wikipedia.org/wiki/Support_vector_machine) is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall."

Here's a visualization of the maximum margin separating two classes using an SVM classifier with a linear kernel.



Since we have a classification task -- honey or bumble bee -- we will use the support vector classifier (SVC), a type of SVM. We imported this class at the top of the notebook.

In [18]:

```
# define support vector classifier
svm = SVC(kernel='linear',probability=True,random_state=42)

# fit model
svm.fit(X_train,y_train)
```

Out[18]:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='linea
r',
  max_iter=-1, probability=True, random_state=42, shrinking=True,
  tol=0.001, verbose=False)
```

10. Score model

Now we'll use our trained model to generate predictions for our test data. To see how well our model did, we'll calculate the accuracy by comparing our predicted labels for the test set with the true labels in the test set. Accuracy is the number of correct predictions divided by the total number of predictions. Scikit-learn's `accuracy_score` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html) function will do math for us. Sometimes accuracy can be misleading, but since we have an equal number of honey and bumble bees, it is a useful metric for this problem.

In [20]:

```
# generate predictions
y_pred = svm.predict(X_test)

# calculate accuracy
accuracy = accuracy_score(y_pred,y_test)
print('Model accuracy is: ', accuracy)
```

```
Model accuracy is:  0.68
```

11. ROC curve + AUC

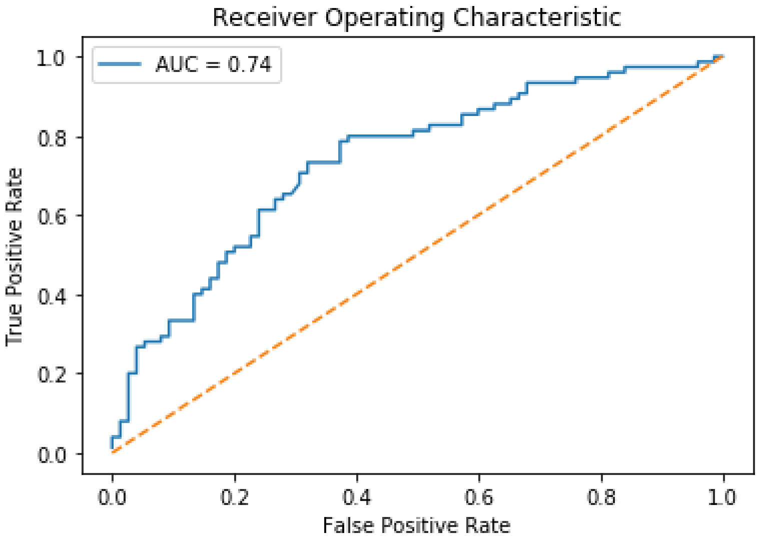
Above, we used `svm.predict` to predict either `0.0` or `1.0` for each image in `X_test`. Now, we'll use `svm.predict_proba` to get the probability that **each class** is the true label. For example, `predict_proba` returns `[0.46195176, 0.53804824]` for the first image, meaning there is a 46% chance the bee in the image is an Apis (`0.0`) and a 53% chance the bee in the image is a Bombus (`1.0`). Note that the two probabilities for each image always sum to 1.

Using the default settings, probabilities of 0.5 or above are assigned a class label of `1.0` and those below are assigned a `0.0`. However, this threshold can be adjusted. The receiver operating characteristic curve (https://en.wikipedia.org/wiki/Receiver_operating_characteristic) (ROC curve) plots the false positive rate and true positive rate at different thresholds. ROC curves are judged visually by how close they are to the upper lefthand corner.

The area under the curve (https://en.wikipedia.org/wiki/Receiver_operating_characteristic#Area_under_the_curve) (AUC) is also calculated, where 1 means every predicted label was correct. Generally, the worst score for AUC is 0.5, which is the performance of a model that randomly guesses. See the scikit-learn documentation for more resources and examples on ROC curves (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html) and AUC (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.auc.html).

In [22]:

```
# predict probabilities for X_test using predict_proba
probabilities = svm.predict_proba(X_test)

# select the probabilities for label 1.0
y_proba = probabilities[:,1]

# calculate false positive rate and true positive rate at different thresholds
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_proba, po

# calculate AUC
roc_auc = auc(false_positive_rate,true_positive_rate)

plt.title('Receiver Operating Characteristic')
# plot the false positive rate on the x axis and the true positive rate on the y ax
roc_plot = plt.plot(false_positive_rate,
                    true_positive_rate,
                    label='AUC = {:0.2f}'.format(roc_auc))

plt.legend(loc=0)
plt.plot([0,1], [0,1], ls='--')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate');
```


Receiver Operating Characteristic
AUC = 0.74
True Positive Rate
False Positive Rate