Assignment 2 - SVM vs Logistic Regression
Gina DiCarlo
Eun Il Kim

Algorithm

We want to first start off by getting our samples from the dataset in sklearn packet. We chose to grab the lfw face of Collin Powell and George Bush, this was done grabbing the minimum face value of 200. We resized the image to .4 to reduce the amount of pixels.

```
lfw = fetch_lfw_people(min_faces_per_person=200, resize=0.4)
```

We then took the amount of samples as well as its dimension of our dataset. Then placed the data for the pixels into our x, which the second column had n features. Then the y which was the name of each respective data set.

```
# n_samples are the total amount of images, h = height, w = width
n_samples, h, w = lfw.images.shape

#extract features as our X
X = lfw.data
n_features = X.shape[1]

# tie the name of the person respect to the feature
y = lfw.target
target_names = lfw.target_names
n_classes = target_names.shape[0] #number of people (which should be 2)
```

Then we used cross validation and divided the test into 4 separate sections where we have our training and test data.

```
#split into a training and testing set: 4 even splits
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=47)
```

Then we are finally ready to use PCA to calculate for eigenfaces and extract the most important features. We choose the limit of 200 components to keep when decomposing, with a random svd solver, and kept the variance scale of the component to increase accuracy. Then we got the eigenfaces from the pca's components, which was then given the structure of n_components, h, and w. Then we went ahead and normalized as well orthogonalized the training data after decomposing to form a training set that has be decomposed.

```
#Compute the PCA (eigenfaces) on the face dataset
n_components = 200
pca = PCA(n_components=n_components, svd_solver='randomized',
          whiten=True).fit(X_train)
eigenfaces = pca.components_.reshape((n_components, h, w))

#Must normalize and orthogonalize the decomposed data for training data and test data
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
```

We then had a function to visually demonstrate our resulted eigenfaces from pca, this function outputs several images that are limited to a 3 by 4 table and gives them a title. We set a figure's dimension using figure from matplot library and we build the frame and set the distance between each picture. Then we finally plot each eigenfaces in each respective position. We mapped the image based on gray image and reshape it based on h and w pixels.

```
#A helper function in order to plot faces
def plot_gallery(images, titles, h, w, n_row=3, n_col=4):
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())
# plot the gallery of the most significant eigenfaces

eigenface_titles = ["eigenface %d" % i for i in range(eigenfaces.shape[0])]
plot_gallery(eigenfaces, eigenface_titles, h, w)

plt.show()
```

Logistic Regression

The next step is to use our classification estimation method to identify/ predict our decomposed data. In this case it will be a logistic regression on the decomposed training data. We first imported a library called called LogisticRegression() we used our training data and attempted on training the data within our logistic regression and we obtain a function. Then made the predictions using our test data on our logistic regression function. Finally we attempted to score our function to test for accuracy.

```
#Predicting with logistic regression
```

```
lreg = LogisticRegression()
#fit using logistic regression
lreg.fit(X_train_pca, y_train)
#This will now predict the names of people
y_pred = lreg.predict(X_test_pca)
score = lreg.score(X_test_pca, y_test)
```

SVM

We also used the PCA to decompose the initial data and run our training/ test data separately. Unlike logistic regression SVM requires tuning parameters ‘C’ which is used for the margin’s softness, and ‘gamma’ used for the influence of the support vectors thereby affecting the shape of the resulting hyperplane. Param_grid will create an array of two lists which include different values of C and gamma. GridSearchCV allows us to specify a linear kernel and a balanced weight to all of the classes, using the previously defined param_grid it will mix and match C and gamma values, finding the best combination when the training data is fit to the model. When the model has been fit the next step is to predict on the the test data.

```
#param_gry allows the different values of C and gamma to be iterated and tested in
different combinations to find the best model
param_grid = {'C': [1e5, 1e3, 5e3, 5e1, 1],
              'gamma': [0.0001, 0.001, 0.01, 0.1, 0.5, 1], }
model = GridSearchCV(SVC(kernel='linear', class_weight='balanced'), param_grid)
model = model.fit(x_train_pca, y_train)
#using our model to predict, evaluating predictions
y_pred = model.predict(x_test_pca)
score = model.score(x_test_pca, y_test)
```

Then we printed out the classification report based on our prediction model as well as the confusion matrix to test for the accuracy.

```
print(classification_report(y_test, y_pred, target_names=target_names))
print("Confusion matrix:")
print(confusion_matrix(y_test, y_pred, labels=range(n_classes)))
```

Finally, we demonstrate the accuracy of our prediction function by showing how our classification matches the actual name or label. This was done by splitting up the last name of the target name and returning the truth value after comparing. Finally we push it to our previous function plot_gallery.

```
# plot the result of the prediction on a portion of the test set
def title(y_pred, y_test, target_names, i):
    pred_name = target_names[y_pred[i]].rsplit(' ', 1)[-1]
    true_name = target_names[y_test[i]].rsplit(' ', 1)[-1]
```

```
    return 'predicted: %s\ntrue:      %s' % (pred_name, true_name)

prediction_titles = [title(y_pred, y_test, target_names, i)
                     for i in range(y_pred.shape[0])]

plot_gallery(X_test, prediction_titles, h, w)
plt.show()
```

Comparison

In general we know that Logistic Regression and SVM separates data using different techniques. Logistic Regression attempts to maximize the probability of success of the data while SVM tries to maximize the distance of the support vectors to the margin (which depends on the chosen 'C' value.). Logistic Regression does not involve the same tuning parameters that SVM does ('C' and 'gamma') which makes SVM a bit more flexible when a specific model accuracy is in mind. When using SVM one must keep in mind that using these tuning parameters can have the opposite desired effect and give a worse model than intended.

Logistic Regression results:

```
Predicting people's names on the test set:
               precision    recall  f1-score   support

 Colin Powell       0.88      0.82      0.84        60
George W Bush       0.92      0.95      0.93       132

  avg / total       0.91      0.91      0.91       192

Confusion matrix:
[[ 49  11]
 [  7 125]]

Prediction score: 92.70%
```

SVM results:

```
Predicting people's names on the test set
               precision    recall  f1-score   support

 Colin Powell       0.88      0.77      0.82        60
George W Bush       0.90      0.95      0.93       132

  avg / total       0.90      0.90      0.89       192

[[ 46  14]
 [  6 126]]
```

```
Prediction score: 92.18%
```

As one can see from the results above, the two methods classified the results with a nearly identical accuracy. The f1-score is the weighted average of the precision (incorrectly labeling one face as another) and recall (how well the classifier identifies positive samples). The closer to 1 the better the accuracy of the results, Logistic Regression does a little better than SVM but not by much. The confusion matrices show similar results, the main diagonal represents the correctly labeled data, while the other values represent incorrectly labeled data. (See figure 1.) For both results the main diagonal has much higher numbers than the off-diagonal, and Logistic Regression is shown to perform a little bit better here as well. The prediction scores show the mean accuracy given test data, these scores are gained by using the .score() function for both SVM and Logistic Regression. Logistic Regression is approximately .52% more accurate for this data set when compared with SVM.

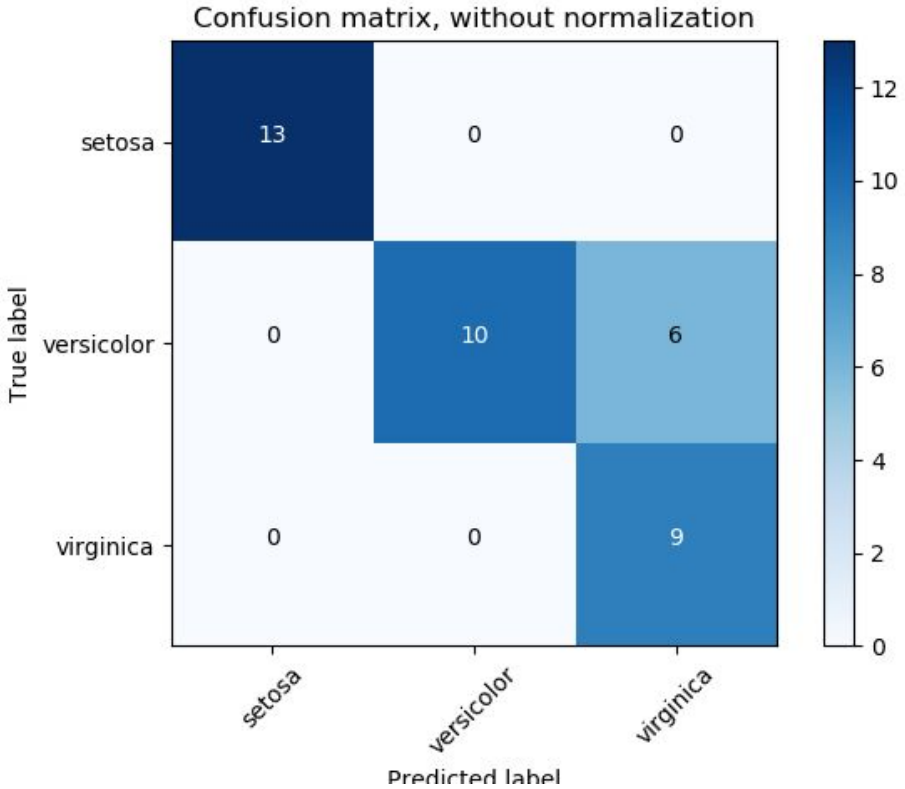
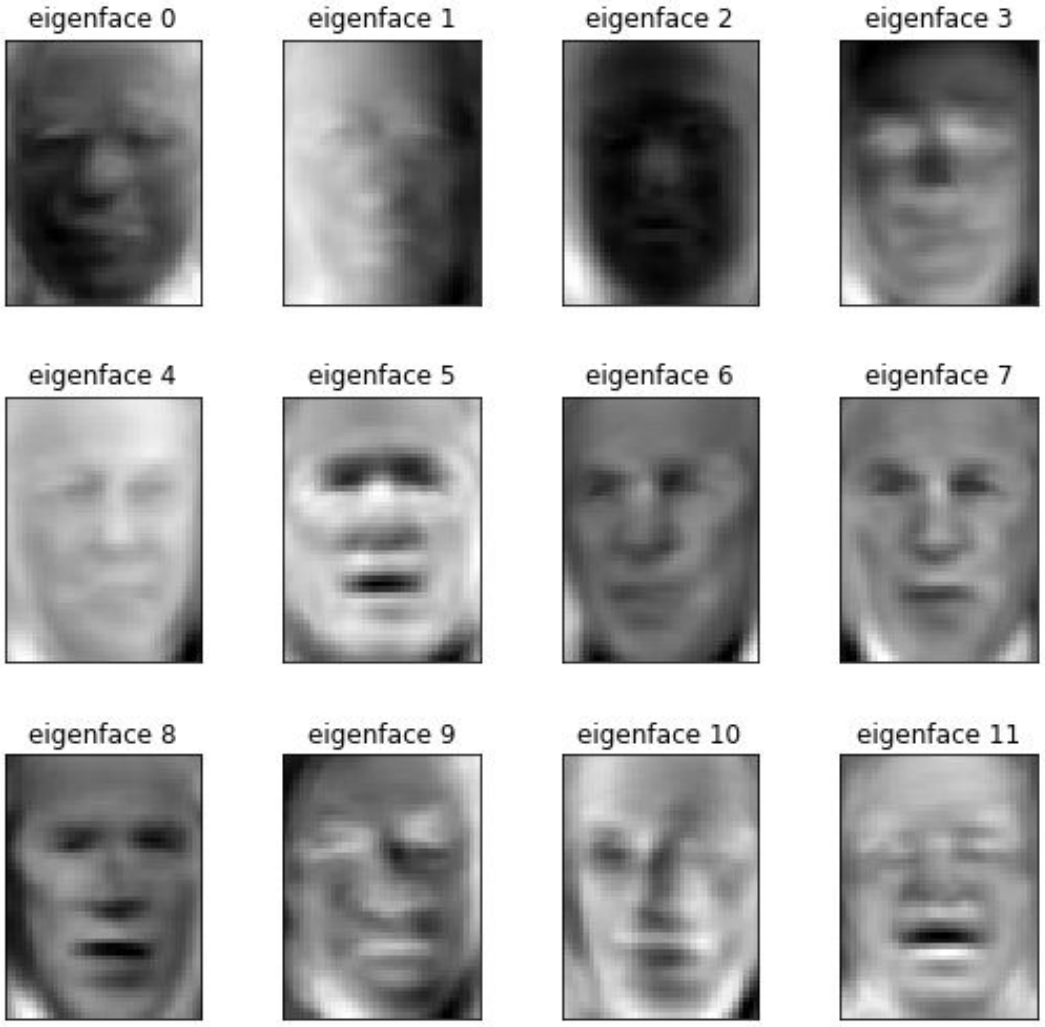


Figure 1: Confusion matrix for classification of the iris data set. Boxes with 0 and 6 values are the numbers of incorrectly data, while the main diagonal shows correctly labeled data. Source: http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py.

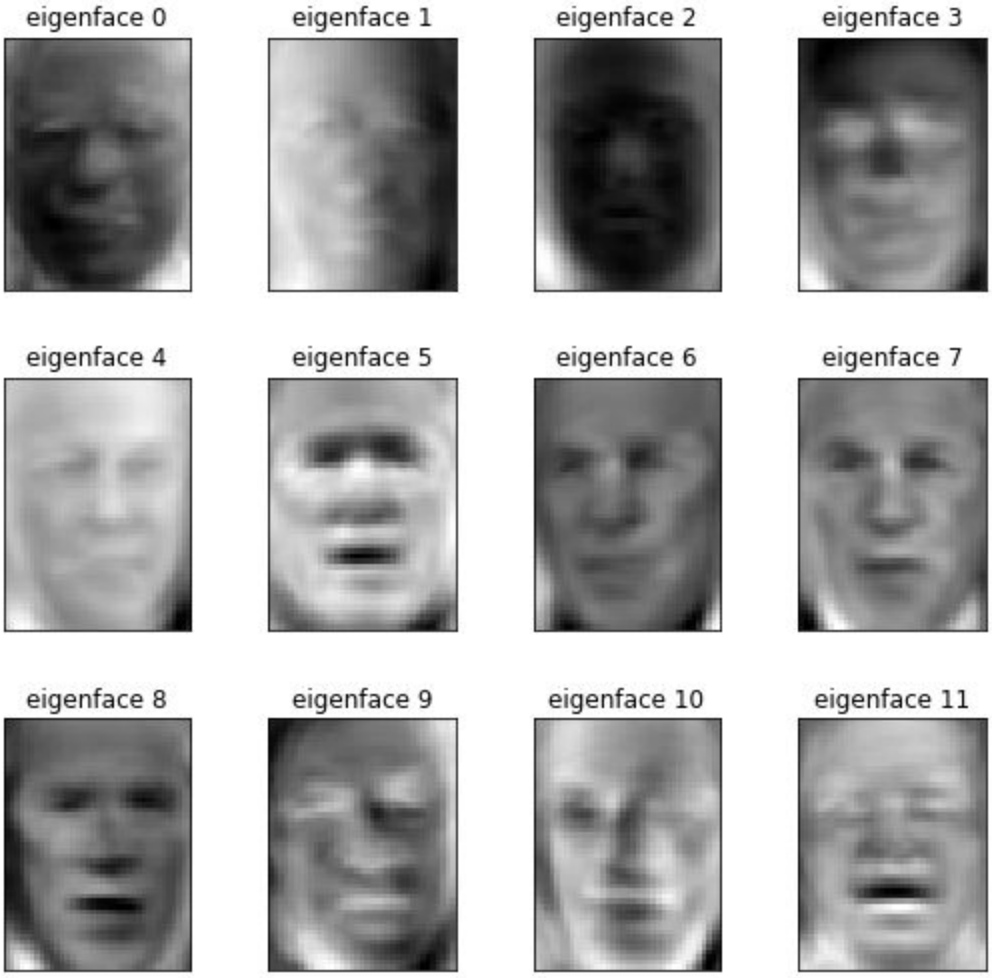
Speed wise Logistic Regression performs much faster with this data set, taking approximately 4.37 seconds to run while SVM took approximately 7.03 seconds. In general the computational complexity for Logistic Regression is O(n^3) while the computational complexity for SVM is O(v* n^2) where "v" is the number of support vectors. Considering the computational complexities it would seem to make sense that SVM would take less time to run, however since

SVM involves tuning parameters the Logistic Regression does not the testing of different ‘C’ and ‘gamma’ values together adds a significant amount of time to running SVM.

Eigenfaces Logistic Regression:



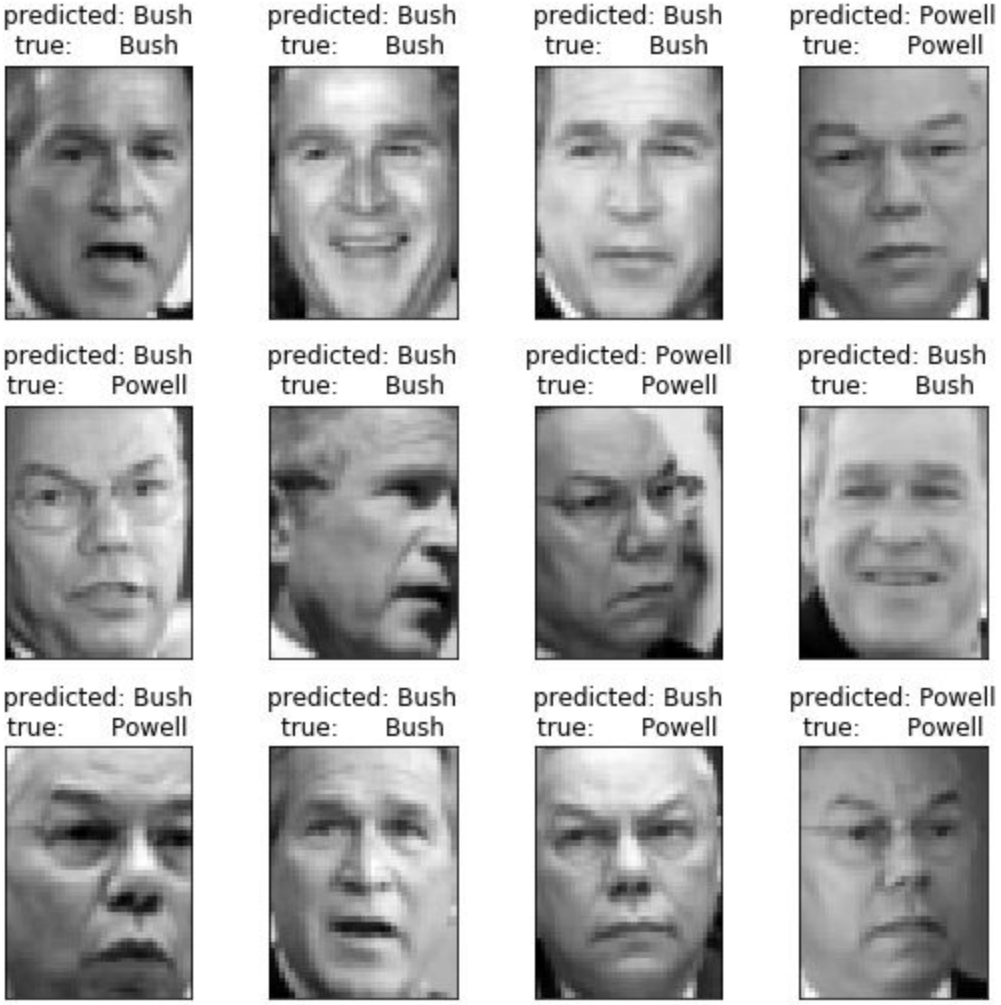
Eigenfaces SVM:



In accordance to nearly identical classification results, the eigenfaces produced from each algorithm are identical to the human eye.

The last step to the algorithms above is to create a gallery of some of the successfully classified faces, you can see the results here:

SVM:



Logistic Regression: