COMP135: Project 01

Shaikat Islam

November 6, 2019

1 Introduction

This paper explores two datasets: the popular MNIST[1] dataset, which corresponds to handwritten digits of 8 and 9, for Part 1, and images of sandals and shoes from the FASHION MNIST[2] dataset for Part 2. Part 1 follows the use of logistic regression models and determining the effect of various parameter optimizations on the accuracy of said models on the digit image data, and Part 2 follows a more creative approach to finding the best possible model on the Fashion MNIST dataset in determining the difference between shoes and sandals. Methodologies are explained in both sections, along with respective figures and images.

2 Part 1: MNIST Handwritten Digit Dataset

The dataset used for this part of the project was a subsection of the MNIST dataset, which only includes the digits 8 and 9. As a result, the training set was composed of 11,800 feature sets, with each feature composed of 784 pixels (corresponding to a 28x28 grid). The testing set was composed of 1983 feature sets. A logistic regression model was fitted to these datasets using `sklearn`, and various parameter tuning was performed, the results of which are outlined and discussed below.

2.1 Number of Iterations v. Accuracy and Log Loss

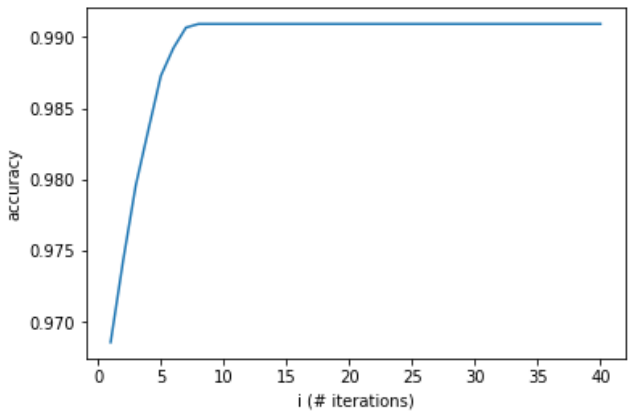

Figure 1: Number of Iterations vs. Accuracy for Logistic Regression Model

[1] http://yann.lecun.com/exdb/mnist/
[2] https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/

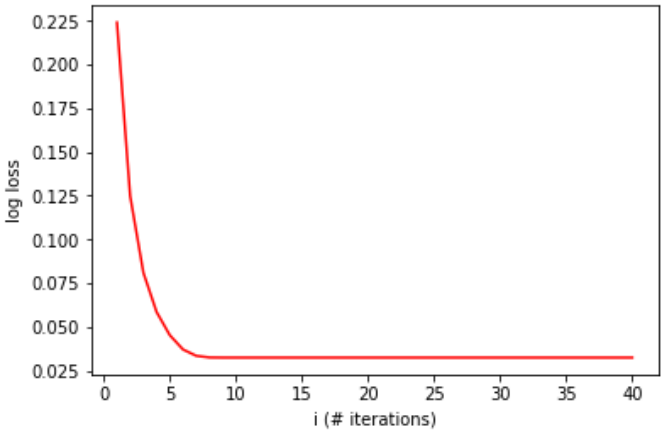

Figure 2: Number of Iterations v. Logistic Loss for Logistic Regression Model

For this section, logistic regression models were fit to the training data with `sklearn`, using the `liblinear` solver. All other values were left to default, and the primary exploration was the effect of limiting the iterations allowed for the models using the `max_iter` parameter. The `max_iter` parameter corresponded to the set of `i = {1, 2, ..., 40}`, and the corresponding accuracy (kept track with the `score()` function), and logistic loss (kept track with the `log_loss()` function) were plotted against the current value of the iteration number.

In Figure 1, the number of iterations were plotted against the accuracy of the model. For `max_iter=1`, the corresponding accuracy starts at 0.970, where it linearly increases to accuracy=.990, where the number of iterations is between 5 and 10. After this, the plot flat-lines, which suggests that as the number of iterations increases from i=10 to i=40, the accuracy remains constant. The accuracy of a model is calculated as $\frac{TP+TN}{FP+FN+TP+TN}$, which means that the cost of running the model as a function of computational power increases at some iteration i, where there are no perceived increases in precision as the model continues to run. The practice of early stopping can be used in this instance, given that the `LogisticRegression model` in `sklearn` uses an iterative coordinate descent algorithm[3] to minimize loss; early stopping[4] minimizes the risk of over-fitting on data, and once the learner reaches a point where accuracy no longer increases, training can stop. Coordinate descent minimizes along coordinates to find the minima of a function[5].

In Figure 2, the number of iterations were plotted against the calculated logistic loss on the training set. Logistic loss is calculated as $\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log p_i + (1-y_i)\log(1-p_i)$. From i=0 to i=5, the logistic loss decreased at a linear rate until it flat-lines from i=5 to i=40. This logistic loss function penalizes wrong predictions at a heavy cost [6], and it increases as the predicted probability deviates from the actual true value for the output class[7]. A perfect model would have a logistic loss of 0, and the model described in Figure 1 flat-lines at a logistic loss of around 0.025.

[3] https://stackoverflow.com/questions/38640109/logistic-regression-python-solvers-defintions
[4] https://en.wikipedia.org/wiki/Early_stopping
[5] https://en.wikipedia.org/wiki/Coordinate_descent
[6] https://towardsdatascience.com/optimization-loss-function-under-the-hood-part-ii-d20a239cde11
[7] http://wiki.fast.ai/index.php/Log_Loss

As similarly described above, early stopping methods would also prevent over-fitting on this data.

2.2 Plotting Initial Weights v. Number of Iterations

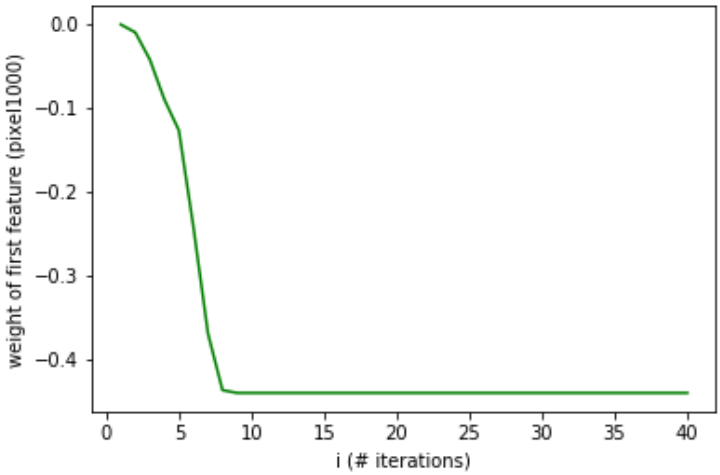

Figure 3: Weight value for pixel000 v. Number of Iteration

In this section, weights were accessed for `pixel000` using the `coef_` attribute of the `LogisticRegression` model. Using `coef_`, all 784 weights were reported, but in this case, the weight for the first feature set of the data, `pixel000`, was accessed using `coef_[0]`, and plotted accordingly against the value i, where i represents the number of iterations. As can be seen in Figure 3, the weight for `pixel000` starts at 0 at iteration 0, and increases until it reaches a final weight value greater than 0.4. This flat-lining, as also evident in Figures 1 and 2, corresponds to the convergence reached by the iterative coordinate decent algorithm described previously. Updates to the weight for `pixel000` no longer minimize loss or improve precision after a certain point, and that point is reached at a point i, $5< i <10$.

2.3 Plotting Initial Weights v. Number of Iterations

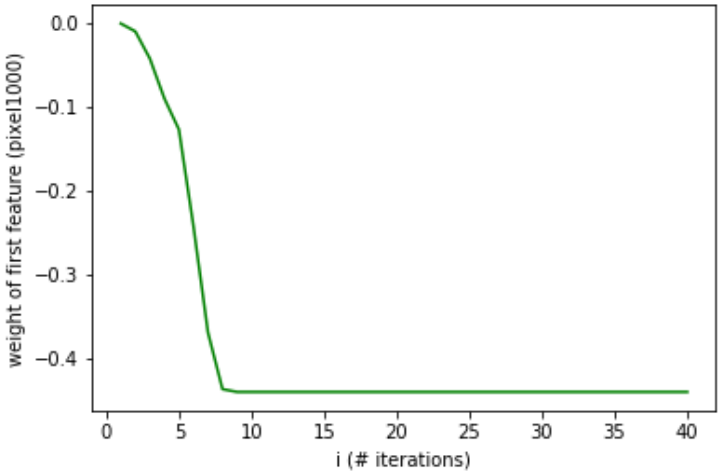

Figure 4: Weight value for pixel000 v. Number of Iteration

2.4 Exploring Values of Inverse Penalty `C` and Choosing the Best Model

In this section, different values of the inverse penalty `C` were used, where `C` is a value within the gridspace defined by `np.logspace(-9, 6, 31)`. Using the same `LogisticRegression` model with the `liblinear` solver as described previously, the best logistic loss value, as well as the

corresponding accuracy value were found, and the confusion matrix describing the best such model was determined. These results can be found below.

Inverse Penalty `C` : `0.0896895561864`
Corresponding Minimum Logistic Loss Value: `0.0896895561864`
Corresponding Accuracy Value: `0.980847457627`

		Predicted Output Labels		
		0	1	Total
True Output Labels	0	942	32	974
	1	33	976	1009
	Total	975	1008	1983

2.5 Interpreting False Positives and False Negatives

In this section, mistakes made by the best model, as defined by the values of `C`, and the corresponding logistic loss and accuracy above, were analyzed by finding false positives and false negatives within our model output set, and plotting them as images using `matplotlib` and the `imshow()` function. The results of this analysis can be seen in a 3x3 grid below, with each figure composed of 9 images corresponding to either a false negative, or a false positive.

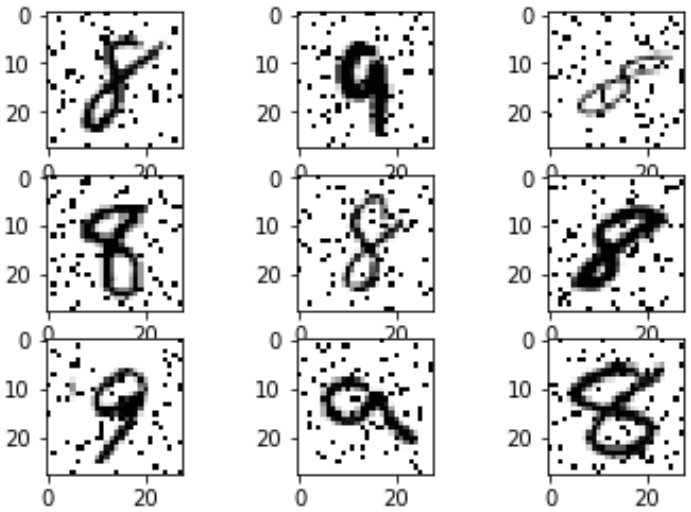

Figure 5: False positives taken from the output set. The model predicted that these were all representative of the digit 9.

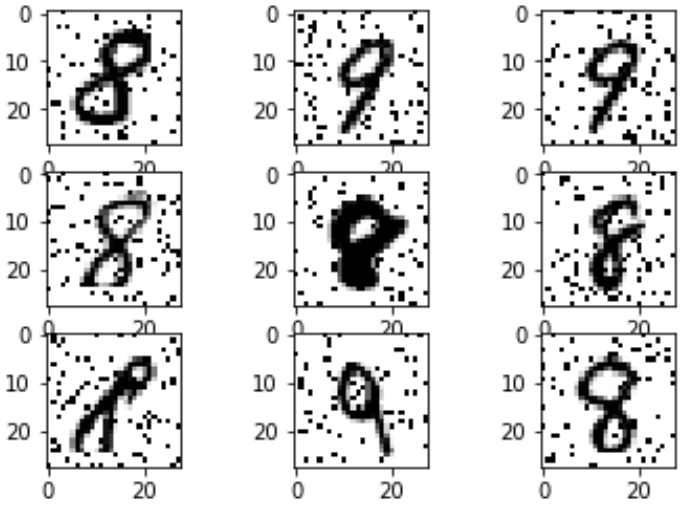

Figure 6: False negatives taken from the output set. The model predicted that these were all representative of the digit 8.

From the set of images describing false positives (Figure 5), it can clearly be seen that the classifier is making mistakes. Of the nine images, six are clearly representative of the digit 8, whereas 3 are representative of the digit 9. If these were truly false positives, all of the digits would be 8s, not 9s. A similar problem is found in Figure 6; all of these images are supposed to represent false negatives, meaning that the images here should be 9s, but the classifier mislabeled them as 8s. This can be explained in terms of the threshold being used to classify between an 8 or a 9 – `LogisticRegression` in `sklearn` uses a threshold of 0.5 to determine class labels, which in this case may not be the most adequate threshold to use for digit classification. In previous assignments, such as HW03, students were tasked to find the best possible threshold for classification, in combination with the best possible inverse penalty value `C`. In this case, using `LogisticRegression.predict()`, the classifier determines classes based on whether or not the calculated probability for a class is greater than, or less than and equal to, 0.5. Using this threshold solidifies the possibility that a classifier will make mistakes even in determining false positives and false negatives. One other possible outcome for this result is that the training data for the input set may have classified some of these images as 8s when they should have been 9s, but such a theory may only be validated by converting all of the feature sets into image representations and classifiying their output label by hand. One other possible reason for this discrepancy is human-error, attributed to erroneous code which may have miscalculated false negatives and false positives. For the sake of observation, the code used to determine these false negatives and false positives is outlined below:

```
def log_reg_inverse_penalty(input_set, output_set, test_set, y_true):
    ''' Performs logistic regression on 784 feature data

    Args
    ----
    input_set: 784 feature input data representing digits 8 or 9
    output_set: 1D array representing 0 (if digit is 8), or 1, if digit is 9

    Returns
    -------
    stdout: inverse penalty value that gives least log-loss on test set, accuracy score of the model, and confusion
    matrix
    cm_df: dataframe containing confusion matrix for best model (with greatest accuracy given an inverse penalty value c)
    '''
    C_grid = np.logspace(-9, 6, 31)
    loss_list = []
    c_list = []
    score_list = []
    for c in C_grid:
        logreg = LogisticRegression(C=c, solver='liblinear')
        logreg.fit(input_set, output_set)
        prob_float = logreg.predict_proba(test_set)
        logloss = log_loss(y_true, prob_float)
        loss_list.append(logloss)
        c_list.append(c)
        score_list.append(logreg.score(input_set, output_set))

    least_log_loss = min(loss_list)
    corresponding_c_val = c_list[loss_list.index(min(loss_list))]
    corresponding_score = score_list[loss_list.index(min(loss_list))]

    logreg = LogisticRegression(C=corresponding_c_val, solver='liblinear')
    logreg.fit(input_set, output_set)
    coef = logreg.coef_[0]
    y_pred_actual = logreg.predict(test_set)
    print(y_pred_actual[0])

    f_n = []
    f_p = []
    for i in range(len(y_true)):
        if y_true[i] != y_pred_actual[i] and y_true[i] == 0.0:
            f_p.append(np.asarray(input_set[i]))

    for i in range(len(y_true)):
        if y_true[i] != y_pred_actual[i] and y_true[i] == 1.0:
            f_n.append(np.asarray(input_set[i]))

    f_p = f_p[:9]
    f_n = f_n[:9]
```

Figure 7: Code used to determine false negatives and false positives.

```
def reshape_arr(arr):
    array = np.zeros([28, 28])
    count = 0
    for i in range(len(array)):
        for j in range(len(array[i])):
            element = arr[count]
            array[i][j] = element
            count+=1
    return array

def reshape_all(arr):
    ret = []
    for i in arr:
        i = reshape_arr(i)
        ret.append(i)
    return ret

fp_data = reshape_all(result_inv_penalty[1])

# False Positives
fig = plt.figure()
ax1 = fig.add_subplot(3,3,1)
ax1.imshow(fp_data[0], cmap='Greys', vmin=0.0, vmax=1.0)
ax2 = fig.add_subplot(3,3,2)
ax2.imshow(fp_data[1], cmap='Greys', vmin=0.0, vmax=1.0)
ax3 = fig.add_subplot(3,3,3)
ax3.imshow(fp_data[2], cmap='Greys', vmin=0.0, vmax=1.0)
ax4 = fig.add_subplot(3,3,4)
ax4.imshow(fp_data[3], cmap='Greys', vmin=0.0, vmax=1.0)
ax5 = fig.add_subplot(3,3,5)
ax5.imshow(fp_data[4], cmap='Greys', vmin=0.0, vmax=1.0)
ax6 = fig.add_subplot(3,3,6)
ax6.imshow(fp_data[5], cmap='Greys', vmin=0.0, vmax=1.0)
ax7 = fig.add_subplot(3,3,7)
ax7.imshow(fp_data[6], cmap='Greys', vmin=0.0, vmax=1.0)
ax8 = fig.add_subplot(3,3,8)
ax8.imshow(fp_data[7], cmap='Greys', vmin=0.0, vmax=1.0)
ax9 = fig.add_subplot(3,3,9)
ax9.imshow(fp_data[8], cmap='Greys', vmin=0.0, vmax=1.0)
```

Figure 8: Reshaping feature arrays into image representations for false positives.

```
fn_data = reshape_all(result_inv_penalty[2])

# False Negatives
fig1 = plt.figure()
ax1 = fig1.add_subplot(3,3,1)
ax1.imshow(fn_data[0], cmap='Greys', vmin=0.0, vmax=1.0)
ax2 = fig1.add_subplot(3,3,2)
ax2.imshow(fn_data[1], cmap='Greys', vmin=0.0, vmax=1.0)
ax3 = fig1.add_subplot(3,3,3)
ax3.imshow(fn_data[2], cmap='Greys', vmin=0.0, vmax=1.0)
ax4 = fig1.add_subplot(3,3,4)
ax4.imshow(fn_data[3], cmap='Greys', vmin=0.0, vmax=1.0)
ax5 = fig1.add_subplot(3,3,5)
ax5.imshow(fn_data[4], cmap='Greys', vmin=0.0, vmax=1.0)
ax6 = fig1.add_subplot(3,3,6)
ax6.imshow(fn_data[5], cmap='Greys', vmin=0.0, vmax=1.0)
ax7 = fig1.add_subplot(3,3,7)
ax7.imshow(fn_data[6], cmap='Greys', vmin=0.0, vmax=1.0)
ax8 = fig1.add_subplot(3,3,8)
ax8.imshow(fn_data[7], cmap='Greys', vmin=0.0, vmax=1.0)
ax9 = fig1.add_subplot(3,3,9)
ax9.imshow(fn_data[8], cmap='Greys', vmin=0.0, vmax=1.0)
```

Figure 9: Reshaping feature arrays into image representations for false negatives.

2.6 Analyzing Final Weights Produced by `LogisticRegression` Classifier with Best Inverse Penalty `C`

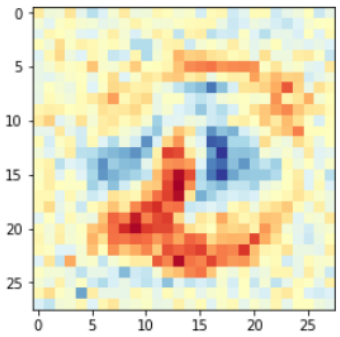

Figure 10: Image representation of the final weights produced by the best classifier. The weights are reconfigured into a 28x28 grid, and plotted as an image using `matplotlib's imshow()` method, using the `RdYlBu` colormap, with `vmin=0.0`, and `vmax=1.0`.

The final weights were taken from the best classifier using the `coef_` attribute of the `LogisticRegression` classifier. The shape of the resulting `NumPy` array was `(784,)`, signifying that it was 1-dimensional. As a result, it needed to be transformed into an array with shape equal to `(28, 28)`, and then plotted using `matplotlib's imshow()` method.

The pixels that correspond to an 8 (have negative weights) are those that are red in hue. Pixels that correspond to a 9 (have positive weights) are more blue in hue. Pixels that are neither red or blue, but correspond to a hue of yellow are empty spaces within images, and red and blue pixels that are outside regions of high red/blue pixel density correspond to added noise in the preprocessing of the MNIST dataset. These results can be verified using the values for `vmin` and `vmax`, along with the `RdYlBu` colormap[8]. Because `vmin=-0.5`, negative weights that are closer to -0.5 are more red in hue, and because `vmax=0.5`, weights that are more positive and closer to 0.5 are more blue in hue. Weights closer to 0 would have yellowish hues, corresponding to white space within an image. Furthermore, if one performed a qualitative analysis on this data, it would be reasonable to ascertain that the pixels more red in hue correspond to the digit 8 due to the similarity of the numeral 8 to the loops represented by the red pixels within the plot. A similar vein of reasoning could be used to say that the blue pixels correspond to the digit 9.

3 Part 3: Sneakers v. Sandals

This section explores the creation of `LogisticRegression` models using `sklearn`, as well as attempts to create the best possible model (based on scoring reported by the Gradescope Predictions Leaderboard), as well as the different feature transformations and parameters used to create such models. Seven models were created, and the methodology for each, along with the performance of the models are described in detail below.

3.1 Logistic Regression Model without Parameter Tuning or Feature Augmentation

For this model, `sklearn's LogisticRegression` model was used with a `liblinear` solver. The shape of the feature set for this data was `(12000, 784)`. The predictions of the model on the test set, which had a shape of `(2000, 784)` was outputted to a file named `yproba1_test.txt`, and the results were analyzed by Gradescope, outputting a value for error, as well as a value for the AUROC.

```
Results:
Error Rate:  0.04249999999999998
AUROC: 0.9932420000000001
Wall Time:  1.61s
```

[8] https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

3.2 Logistic Regression Model with Data Augmentation: Multiplying Training Set by 2x

For this trial, the training set, which had an original shape of `(12000, 784)` was transformed by adding horizontal flipping to the images, which resulted in the training set having a modified shape of `(24000, 784)`, which is an effective 2X increase in the size of the training set. The images were flipped using `NumPy's flipud(array)` function, and resultant images were appended to the original training set. This resulted in a surprising decrease in accuracy, as the model performed worse on the Leaderboard than the model without any data augmentation or feature tuning.

One possible reason for the increase in error rate from the baseline model is that the addition of horizontally flipped images may have overfitted the model to input data that is more similar in vein to the original input set. Another application of data augmentation that may result in better error rate includes the Gaussian transformation, which is explored in the next subsection.

```
Results:
Error Rate:  0.05300000000000005
AUROC: 0.98663
Wall Time:  4.61s
```

3.3 Logistic Regression Model with Data Augmentation II: Applying the Gaussian

For this trial, elements of the feature set were reshaped into 2D `NumPy` arrays of shape `28, 28`, upon which the `gaussian_filter()` function from the `scipy` library was applied on the 2D arrays with `sigma=784`, and then flattened to a 1D array of shape `(784,)`. This was done to better normalize, or 'smoothen' the data to remove outliers[9]. The resultant features were then added to the original input set, resulting in an input set 2x greater than the original. This resulted in better performance than the image transformation that flipped the images in subsection 3.2, but worse performance than simply using `sklearn's LogisticRegression` model with no parameter tuning or data augmentation.

```
Results:
Error Rate:  0.04400000000000004
AUROC: 0.9921749999999999
Wall Time:  1min 32s
```

3.4 Logistic Regression Model with Data Augmentation III: Box-Cox Transformation on Input Set

For this trial, elements of the training set were transformed with `scipy's boxcox1p()` function with a power parameter () of 0.25. This was done in an attempt to normalize the data[10]. The test set was also normalized, using the same parameters in the initial transformation of the training set. However, it is important to note that logistic regressions do not assume normalized data, so any affect of this transformation on the input set as perceived by increased results on the test set may be perceived as an anomaly. The formula for the Box-Cox transformation is as follows:

$$y(\lambda) = \begin{cases} \frac{(y+\lambda_2)^{\lambda_1}-1}{\lambda_1}, & \text{if } \lambda_1 \neq 0 \\ \log(y+\lambda_2), & \text{if } \lambda_1 = 0 \end{cases}$$

Because there were features equal to 0 within the initial dataset, a shift value was applied to the features in this transformation, after which the modified Box-Cox equation above was applied on the data. This transformation produced the best performance thus far, beating all previous models in terms of error rate and AUROC.

```
Results:
Error Rate:  0.040000000000000036
AUROC: 0.994132
Wall Time:  1.58s
```

[9] https://towardsdatascience.com/why-data-scientists-love-gaussian-6e7a7b726859
[10] https://www.statisticshowto.datasciencecentral.com/box-cox-transformation/

3.5 Logistic Regression Model with Data Augmentation IV: Box-Cox Transformation on Flipped Input Set

For this trial, elements of the training set were flipped as mentioned in subsection 3.2, and then the modifed Box-Cox, as described in the previous section was applied on the training set. This trial performed worse than all previous trials, but had an AUROC higher than the AUROC measured when simply flipping the images and increasing the training set by 2x. At this point, performance may have suffered because of data transformations that produced extremely variant input sets – these transformations may have further strayed from the original nature of the original training set, producing higher error rates and resulting in a further ungeneralized classifier.

```
Results:
Error Rate:  0.05449999999999999
AUROC: 0.988159
Wall Time:  5.43s
```

3.6 Logistic Regression Model with Feature Splitting: Splitting Test and Training Sets (2:1)

In this trial, the initial feature set was divided into a test and train set. 66 percent of the training set was used as the new training set, and the other 33 percent was kept as a test set. This was done in an attempt to reduce overfitting on the training data, and to better generalize on to the test data. `sklearn's model_selection.train_test_split` was used to split the data into a training and testing set. This resulted in the best performance so far, as described below.

```
Results:
Error Rate:  0.03949999999999998
AUROC: 0.993793
Wall Time:  976ms
```

3.7 Logistic Regression Model with Feature Splitting II: K-Fold Cross Validation (K=10)

In this trial, k-fold cross validation with a value of k=10 was used to reduce overfitting even further, but this trial had worse results than the 2:1 split. `sklearn's model_selection.KFold()` function was used with `n_splits=10`.

```
Results:
Error Rate:  0.04249999999999998
AUROC: 0.993114
Wall Time:  13.9ss
```

3.8 Concluding Notes on Methodology and Comparison of Model Performance

In the course of this project, seven models were created, and of the seven, the best performing models were those described in subsection 3.4 and 3.6 – the Box Cox transformation that normalized the training set data, and the feature splitting. When designing these models, I thought it would be best to build upon previous iterations. As a result, I started with a classifier with no changes to the parameters or training sets. This can be described as the baseline. Afterwards, I decided that adding flipped images to the training set may improve performance, but it did the opposite, and it increased the error rate. Next, I decided to apply another data transformation and apply the Gaussian transformation, which I believed would better smoothen the data by normalizing it further, and as a result, reduce overfitting and increase performance. This model did better than the previous data augmentation, but I wanted to explore more methods of transforming data, so I looked up the Box-Cox transformation, which had the best performance thus far in my search for the best classifier. I thought that using the Box-Cox transformation on the flipped input set would increase performance even further, but this model had the worst performance thus far, and the worst performance of all seven models in general. After performing many data augmentations, I decided to split the feature sets into a training and test set, and did so in subsection 3.6, which had the best performance of all the models thus far. Taking a step further, I decided to try K-fold Cross Validation with K=10, but this produced results most similar to the baseline.

In terms of plotting the performance of the models with respect to each other, the results of all the models are found in Figure 11, which includes Error Rate, AUROC, and Computation Time. A scatterplot representing the performance of each model labeled by subsection (3.1-3.7) is described in Figure 12 (error rate v. AUROC), and Figure 13 represents the performance of each model labeled by subsection in terms of error rate vs. computation time.

Model	Time(s)	Error Rate	AUROC
3.1	1.61	0.04249999999999998	0.9932420000000001
3.2	4.61	0.05300000000000005	0.98663
3.3	92	0.04400000000000004	0.9921749999999999
3.4	1.58	0.040000000000000036	0.994132
3.5	5.43	0.05449999999999999	0.988159
3.6	.976	0.03949999999999998	0.993793
3.7	13.9	0.04249999999999998	0.993114

Figure 11: Results in table form.

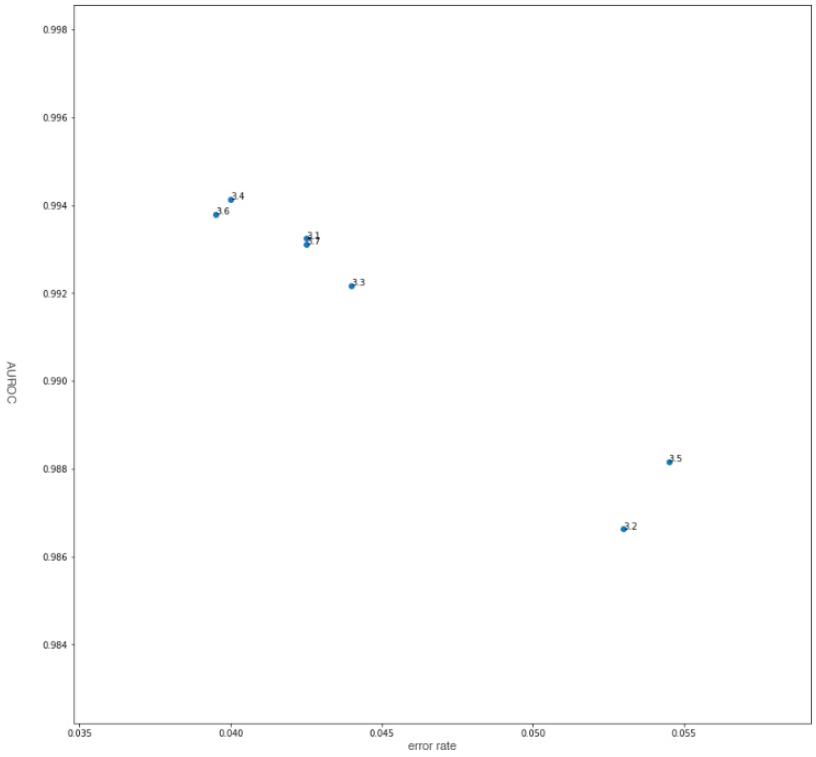

Figure 12: Error rate v. AUROC. Note models 3.4 and 3.6 (Box-Cox transformation and 2:1 Split) are the best performing models, at the top left of the graph.

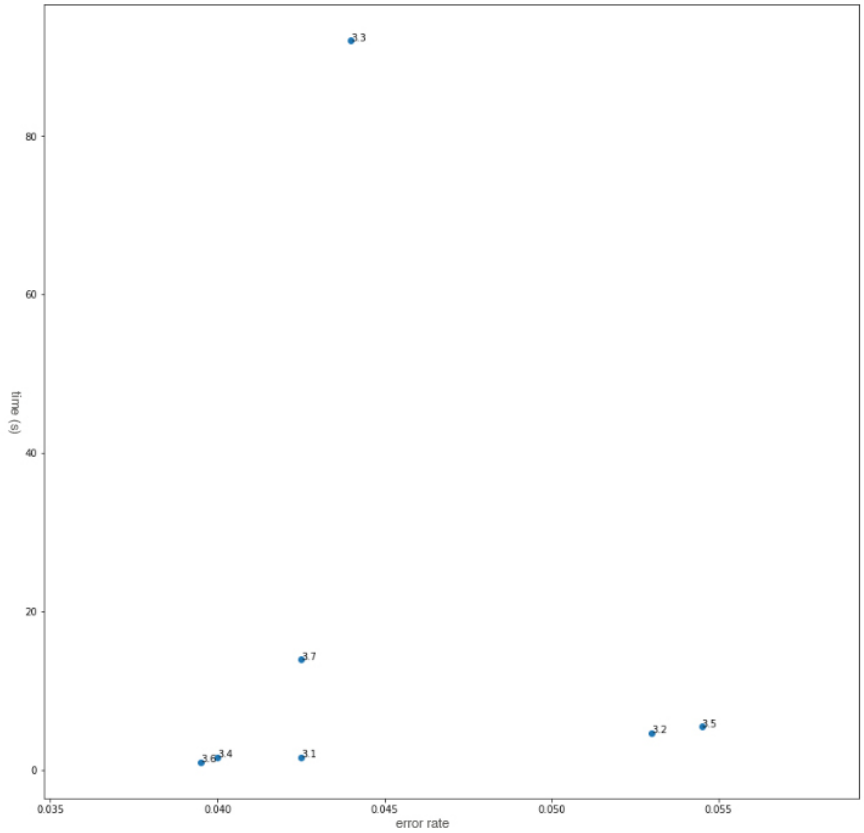

Figure 13: Error rate v. Time(s). Note models 3.4 and 3.6 (Box-Cox transformation and 2:1 Split) are again the best performing models, at the bottom left of the graph.

In the course of this project, I had many ideas for methodologies, but was unable to perform them: in the continuation of this project, I believe that it would be of note to explore `sklearn's GridSearch` to perform hyperparameter tuning on the `LogisticRegression` model, as well as Leave One Out Cross Validation to further test the performance of models on the number of splits with regard to the testing set.

References

1 http://yann.lecun.com/exdb/mnist/
2 https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/
3 https://stackoverflow.com/questions/38640109/logistic-regression-python-solvers-defintions
4 https://en.wikipedia.org/wiki/Early_stopping
5 https://en.wikipedia.org/wiki/Coordinate_descent
6 https://towardsdatascience.com/optimization-loss-function-under-the-hood-part-ii-d20a239cde11
7 http://wiki.fast.ai/index.php/Log_Loss
8 https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html
9 https://towardsdatascience.com/why-data-scientists-love-gaussian-6e7a7b726859
10 https://www.statisticshowto.datasciencecentral.com/box-cox-transformation/