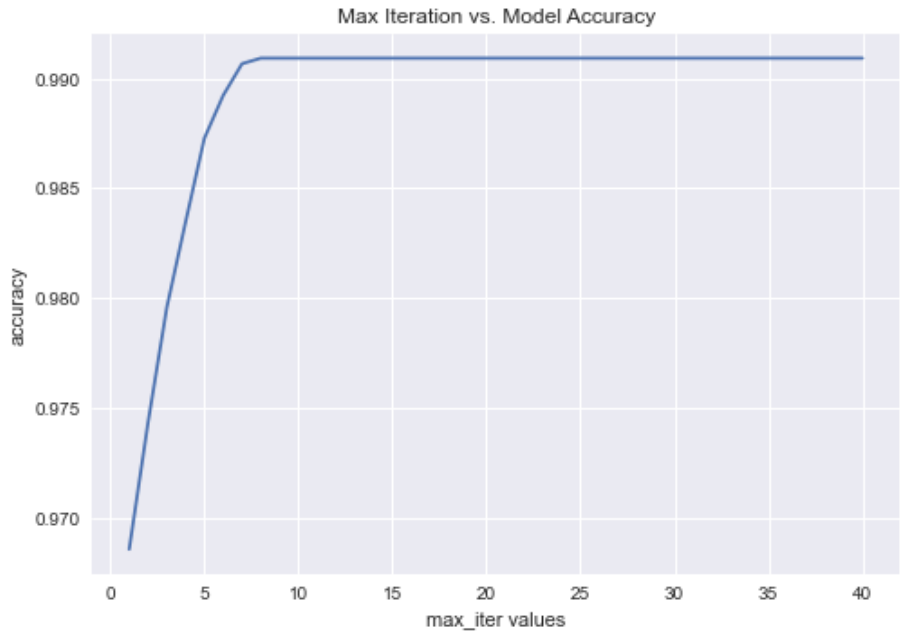
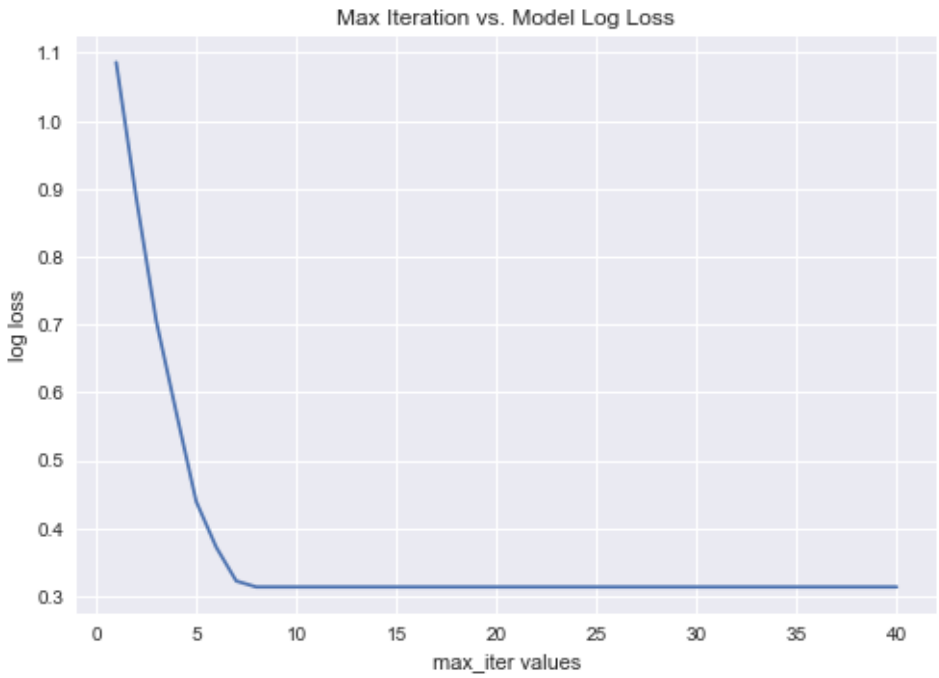
Mattia Danese
Project 1
4/4/21

Part One: Logistic Regression for Digit Classification

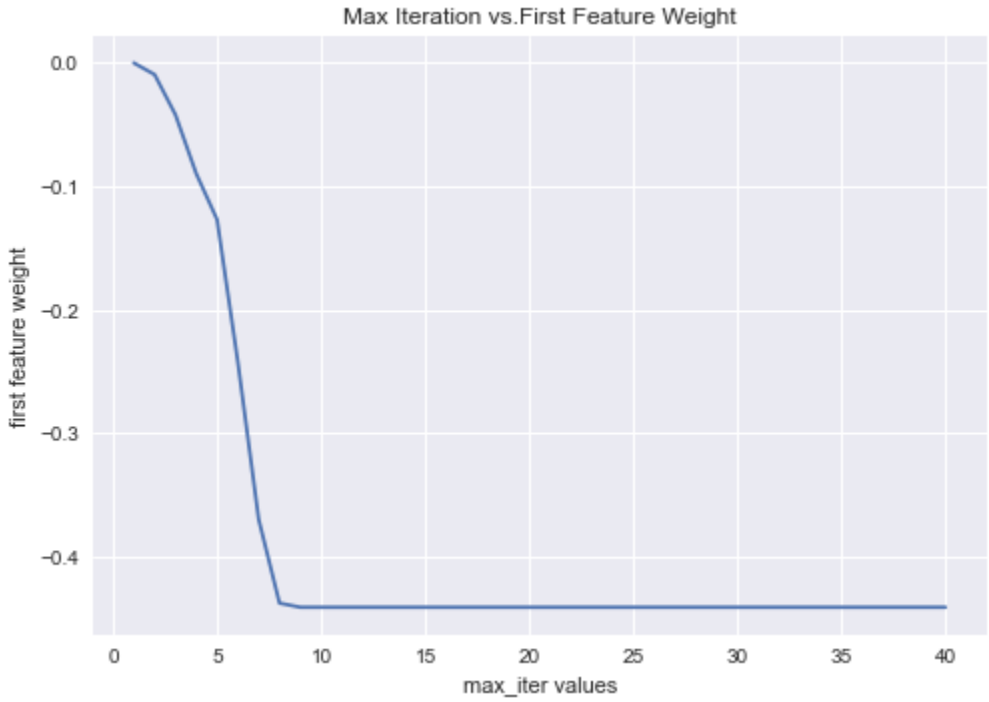
Question 1





Using the plots above and printing out the index of the minimum value in the list of log loss error and the maximum value in the list of accuracy of each model, I got that the optimal *max_iter* value is 8. The plots show that from *max_iter* values from 1 to 8, the log loss of the model linearly decreases and the accuracy of the model linearly increases. Additionally, from *max_iter* values from 8 to 40, the log loss and the accuracy of the model do not change. These results must mean that the model converges at *max_iter* = 8, as this is where log loss is minimized and accuracy is maximized. This is also supported by the log loss not decreasing and the accuracy not increasing further for *max_iter* values above 8; since the model already converged there is no more work to be done (for the model to converge and "be better") so any additional iterations are useless.

Question 2



The plot shows that the weight of the first feature linearly decreases from *max_iter* values of 1 to 8 and then does not change from *max_iter* values of 9 to 40. Similarly to the accuracy and log loss plots above, this must mean that the model converged at *max_iter* = 8 because the model is updating the weight of the first feature (i.e. trying to converge) up until *max_iter* = 8 and then does not update it anymore (the model converged so there is no need to update the weight of the first feature, or any other weight for that matter).

Question 3

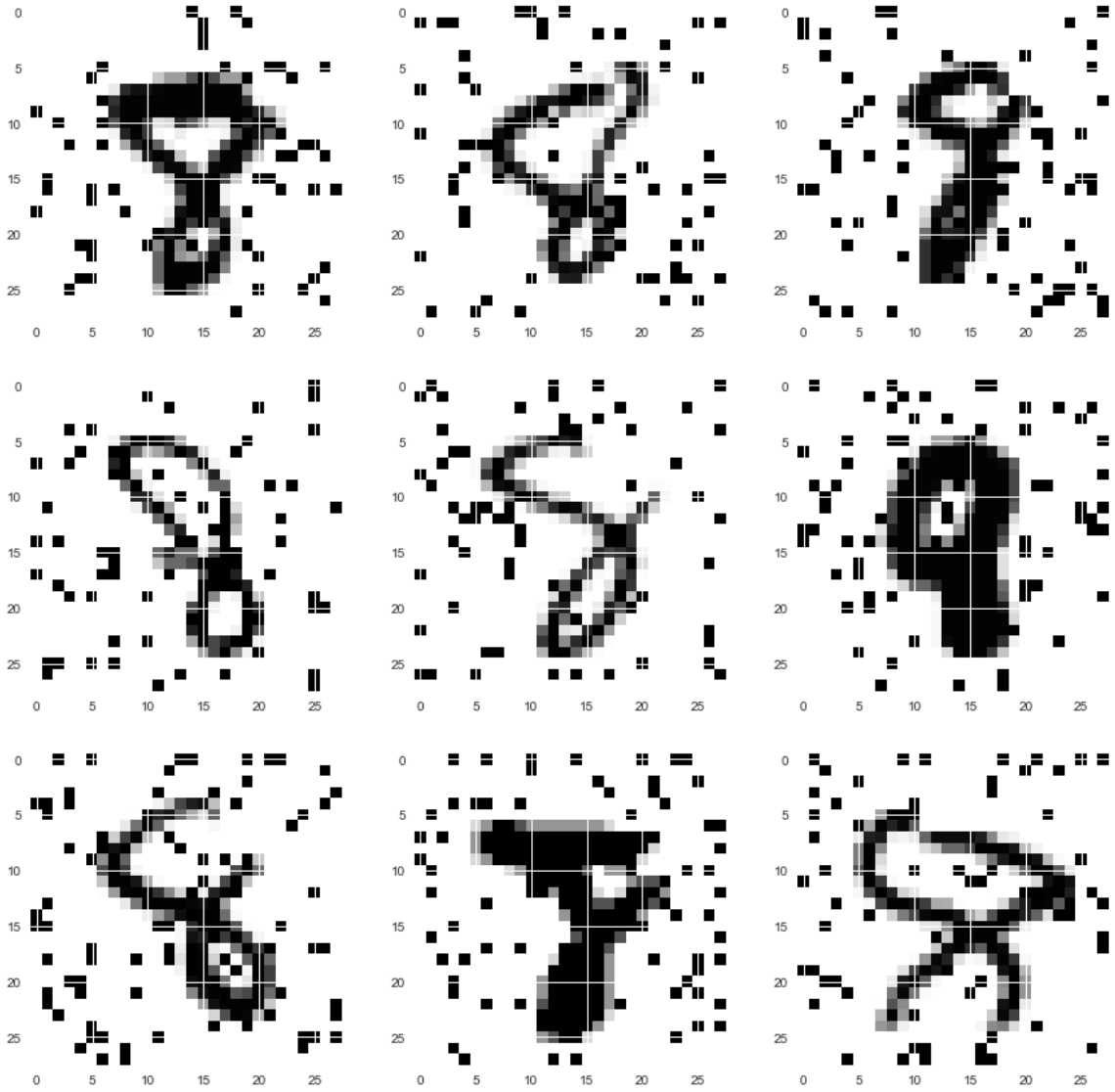
I found the best *C* value by creating an array with 30 regularly-spaced values, just like the spec suggested, and tested each one as the *C* value of a model. For each possible *C*, I recorded the log loss and accuracy of the model on the testing data. Below are my results:

```
C-value that results in lowest loss: 0.03162277660168379
Accuracy of model with best C-value: 0.9672213817448311
```

To produce the confusion matrix, I copied over the *calc_confusion_matrix_for_threshold* and *calc_TP_TN_FP_FN* functions from the third homework. To find the optimal threshold value, that is the threshold that results in the least amount of false positives and false negatives, I made an array with values 0.01, 0.02, …, 0.98, 0.99. I then looped over this array and recorded the sum of false positives and false negatives that each value produced by using the *calc_TP_TN_FP_FN* function in a very similar way as in the third homework. Doing this, I got that the threshold that resulted in the best results is 0.61. Finally, I used the *calc_confusion_matrix_for_threshold* function to print out the confusion matrix.

```
The threshold that produces the lowest sum of FP and FN is: 0.61

Predicted    0    1
True
0          950   24
1           34  975
```

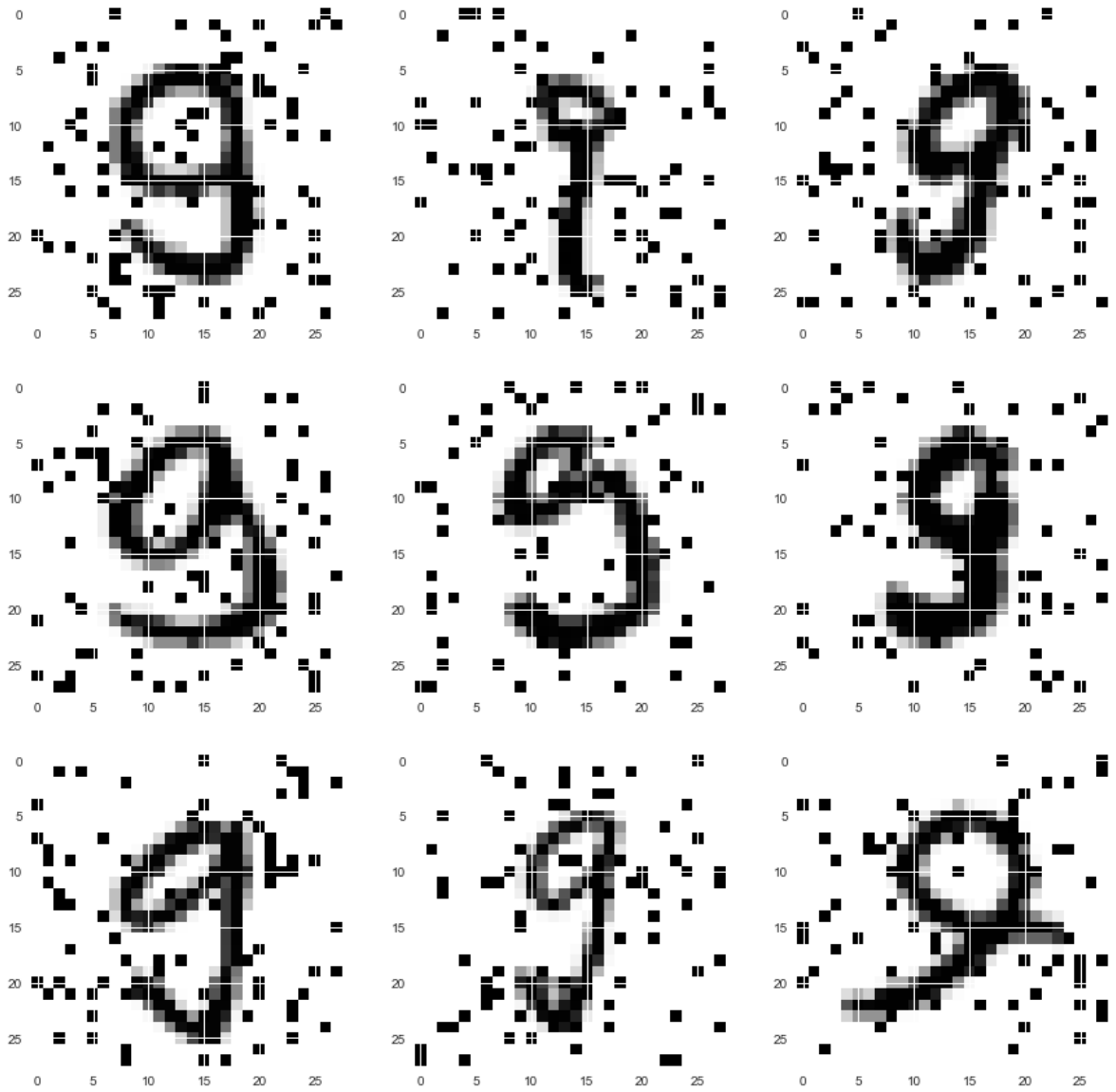
Question 4



**Refer to images as numbers, 1-9, where the top row has images in the order 1, 2, 3 the second row has images in the order 4, 5, 6, and the bottom row has images in the row 7, 8, 9

Above are nine images that the best model classified as 9's when they were actually 8's (i.e. they are false positives). The reason for why these images were misclassified can possibly be explained by the physical features of the images and/or any bias in the data set used to train the model. To start, images 3, 6, and 8 do not have a well defined lower loop. Their lower loop is squished which can be interpreted by the model as a line. This leads the model to mistakenly think that these 8's are 9's because, as far as the model can tell, these images have a line and an upper loop which resembles the shape of 9 more than that of an 8. Based on images 1, 2, and 4, the model might be making the mistake of thinking that these 8's are 9's because their loops are not completely unfilled (the pixels in the loops are not completely white space). This can definitely be the case if the training data set of the model had many more 8's that have

completely hollow loops than 8's that have filled or partially filled loops. Based on images 5, 7, and 9, the model might be making the mistake of thinking that these 8's are 9's because they have broken loops. Similar to the loops being filled or not, if the training data set had a lot more 8's whose loops were completely intact, then the model might mistakenly classify any 8 with broken loops as *not* an 8 (and thus a 9).

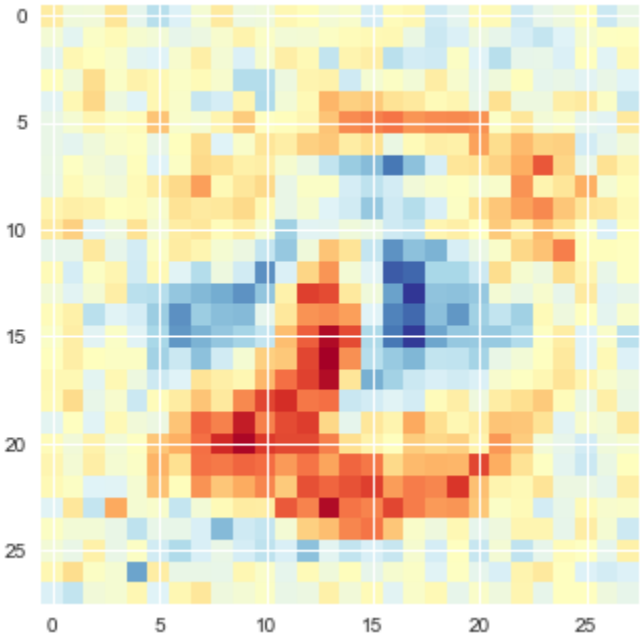


**Refer to images as numbers, 1-9, where the top row has images in the order 1, 2, 3 the second row has images in the order 4, 5, 6, and the bottom row has images in the row 7, 8, 9

Above are 9 images that the best model classified as 8's when they were actually 9's (i.e. they are false negatives). Much like the false positives previously mentioned, the reason for why these images were misclassified can possibly be explained by the features of the images and/or any

bias in the data set used to train the model. Most of the images, in fact all of them except for image 2, are written with a lower curve, just like a lowercase *g*. The model might have made the mistake of interpreting this lower curve as the lower (second) loop of an 8. Based on image 2, the model might also be making the mistake of classifying an image as **not** a 9 (and thus an 8) if the top circle of a 9 is filled in or relatively small. These two mistakes can definitely be the case if the training data set of the model has a lot more 9's which don't have a lower curve (just a straight line) and have a distinguishable loop at the top.

Question 5



Due to the parameters *vim*=-0.5 and *vmax*=0.5, red or redder colors correspond to negative weights close to -0.5 and blue or bluer colors correspond to positive weights closer to 0.5. An 8 has a top and bottom curve which can be represented by the two curves of red color near the y=5 and y=20 lines. A 9 typically has a solid vertical line, usually seen a bit to the right of the middle of the image, which can be represented by the deep blue vertical line near x=16. Additionally, an 8 is part of class 0 and a 9 is part of class 1, so it would make sense that the negative weights (red colors) correspond to 8's and positive weights (blue colors) correspond to 9's. Since the red/blue hot spots correlate to the features and class of an 8 and 9, it can be concluded that red pixels correspond to an 8 (and have negative weights) and blue pixels correspond to a 9 (and have positive weights). The yellow pixels mostly occupy the edges of the image and surround the interior mix of red and blue pixels, meaning that the yellow pixels correspond to white space. Moreover, yellow is in between red and blue on the color spectrum so it would be reasonable to assume that the weights of the yellow pixels are in between the weights of the red and blue pixels. This would make the weights of yellow pixels close to 0 which makes sense if these

weights correspond to white space as they would not be important in classifying an image as an 8 or a 9.

Part Two: Sneakers versus Sandals

Creating Base Model

```
Base Error on Training: 0.07686861826160421
Base Accuracy on Training: 0.9735416666666666

Base Error on Testing: 0.11082480098855174
Base Accuracy on Testing: 0.9616666666666667
```

Before adding any features to the data, I wanted to create a base model where all of its features and parameters were not changed; this would act as a baseline to evaluate the effectiveness of future feature transformations and parameter values. Since there was no correct output for the testing data set given, I split the training data set into a new training data set (80%) and a testing data set (20%). For this base model and all other models, I trained on this new training data set and recorded error and accuracy on both this new training data set and the testing data set I created. Above are the results for the base model.

Feature Transformation #1: Image Brightness

```
Feature 1 Error on Training: 0.07603652345192814
Feature 1 Accuracy on Training: 0.9736458333333333

Feature 1 Error on Testing: 0.11081855515413068
Feature 1 Accuracy on Testing: 0.9625
```

The first feature transformation I did was adding features indicating the brightness of an image in specific areas. I added a total of 14 features, one for each 2x2 square of an input image. This was quite easy to implement, I looped over the 2D array of each input feature array and summed up the pixels values of each 2x2 square. I then appended each of these sums to the current feature array. I thought of this feature because I noticed that a lot of sneakers took up the same space in the physical image and this could result in a brightness pattern between sneakers that could help distinguish them from sandals.

The reason why I added these features is because I noticed that the majority of sneakers have a lot more non-white pixels than the majority of sandals. In other words, sneakers are usually bigger than sandals, so they are simply composed of more “colored” pixels. Additionally, a lot of sandals have loops or “openness” in some way that leads to their images having more white space, while sneakers, analogous to a block, only have white space in surrounding areas. Images of sneakers having less white-space leads to the average brightness in the 2x2 areas of sneaker images to be, on average, darker than those of sandal images.

Above are the results of this feature transformation, and it is clear that adding these features did help the model. The error on both the training and testing datasets decreased and the accuracy on both the training and testing datasets increased.

Feature Transformation #2: Gaps in Image

```
Feature 2 Error on Training: 0.0474268088534182
Feature 2 Accuracy on Training: 0.98375

Feature 2 Error on Testing: 0.08592431478079178
Feature 2 Accuracy on Testing: 0.9683333333333334
```

The second feature transformation I did was adding features that gave information about total white space in an image. I added a total of 28 additional features, one feature per row in the image. To implement this, I looped over the rows of the image and made an array where, for each pixel, I would append a 1 (pixel is white space) or a 0 (pixel is not white space). Afterwards, I trimmed the outer 0's from both sides of the array and summed up all the remaining values, appending this value to the current feature array.

The reason why I added these features is because they give the model information about whether there are gaps, or openings, in the shoe or not. Both sneaker and sandals will have white space in the surrounding areas of the shoe, but I noticed that sandals also have white space *within* the shoe (i.e. gaps, loops, etc.). This results in a clear distinction between both shoes as images of sandals most likely have more total white space than images of sneakers.

Above are the results of this feature transformation, and it is clear that adding these features did help the model. The error on both the training and testing datasets decreased and the accuracy on both the training and testing datasets increased much more than those from the first feature transformation.

Feature Transformation #3: Dimensions of Shoe

```
Feature 3 Error on Training: 0.04538982959103112
Feature 3 Accuracy on Training: 0.9860416666666667

Feature 3 Error on Testing: 0.07615805350239484
Feature 3 Accuracy on Testing: 0.9708333333333333
```

The third feature transformation I did was adding a feature for every row and column (56 in total) with a value describing the amount of non-white pixels in that respective row or column.

To do this, I looped over all of the rows of each image and saw if any of its pixels had a value above 0 (making it non-white); if the row did have such pixels, then a counter for the row would be incremented as many times as there were such pixels. The value of this counter would be appended to the feature array, and the loop would continue to the next row. This process was then also repeated for all the columns of each image.

The reason why I added these features is because they give the model information about the height and width of the shoe. As I was looking through the input images, I noticed that most of the sneakers were shorter than most of the scandals, so I thought adding a feature regarding the shoe height may help the model classify a certain shoe better. All the shoes had pretty much the same width (the width of the image), so I did not suspect features regarding the width of the shoe having much of an impact, but I added these features anyways as they could only help (and the code was identical to that of the height features so it was rather easy to add).

Above are the results of this feature transformation, and it is clear that adding these features did help the model. In fact, this feature transformation increased the accuracy and decreased the losses, on both the training and testing datasets, the most out of all the other feature transformations.

Putting all Features Together

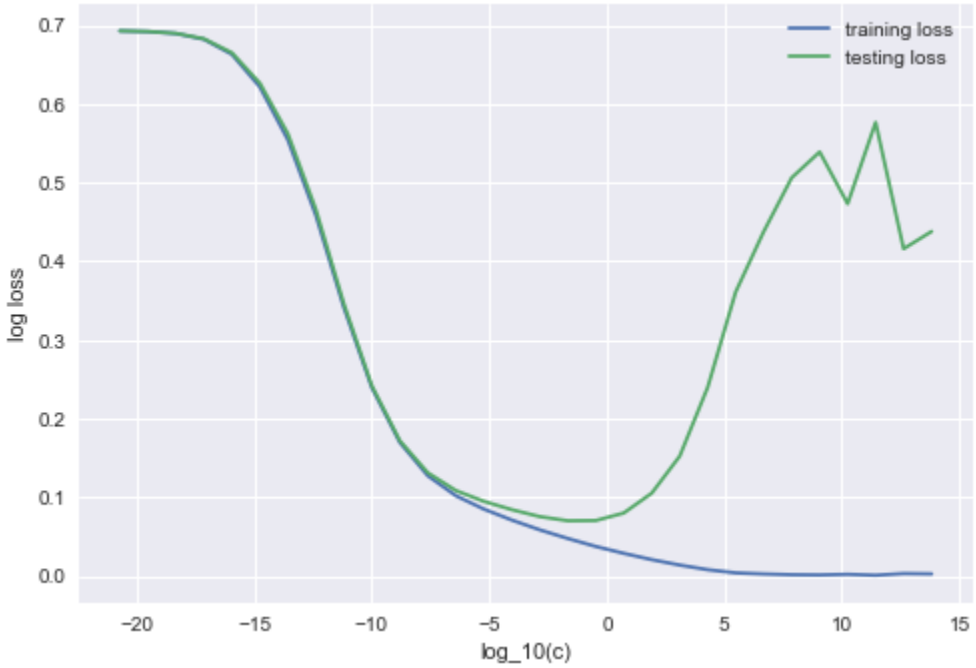
```
Feature 4 Error on Training: 0.03443265832867187
Feature 4 Accuracy on Training: 0.9891666666666666

Feature 4 Error on Testing: 0.07615805350239484
Feature 4 Accuracy on Testing: 0.975
```

Since each feature transformation made the model better (in terms of accuracy and loss) for both the training and testing data sets, I decided it would be a good idea to see what effect all three transformations would have on the same model. I transformed the training and testing data three times, so that they both have all the additional features, and recorded the accuracy and log loss of the model on both data sets. The results, as seen above, clearly show the biggest model improvement when compared to the base model or any of the single feature transformation models.

Optimizing Model Parameters

At this point I am confident that my feature transformations helped the model a substantial amount so I moved on to fine tune the model by finding optimal *C* and *max_iter* values. Much like Part One, I used an array with 30 regularly-spaced values and tested each value as a possible *C* value. Below are my results:

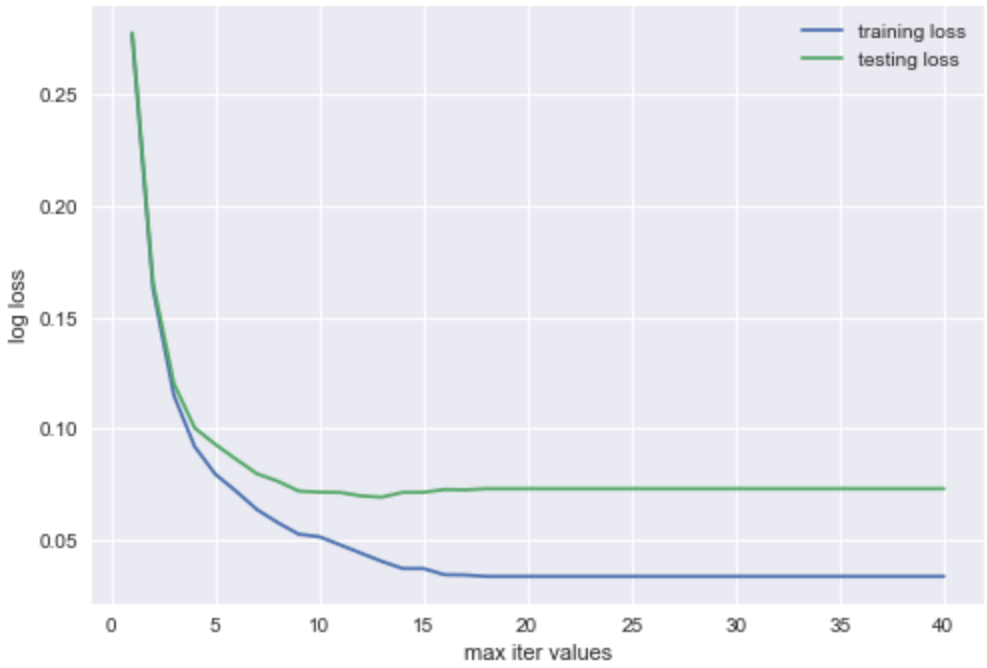
```
Best C for lowest training loss: 92367.08571873885
Best C for lowest testing loss: 0.18873918221350997
```



As the *C* value started to increase, both the testing and training loss decreased rapidly in the same manner. However, as *C* values started to get relatively large, the training loss began to flatten out while the testing loss began to shoot up again and reach levels almost as high as when the *C* value was very small. This is clear evidence that, for these values of *C*, the model was overfit to the training data which is why the training loss was basically 0.0 and the testing loss was relatively large. As a result, I decided to go with the *C* value that minimized testing loss (shown above) because, based on the graph, any *C* before it would lead to an underfit model and any *C* after it would lead to an overfit model. Additionally, there is not too much difference between the training and testing losses at this particular *C* , so the model at this *C* did particularly well when it saw new data (more on this in the next section).

The next model parameter I chose to optimize is *max_iter*. Again, very similar to Part One, I used the same array of possible *max_iter* values: 1, 2, …, 39, 40. I looped through this array, trying each value as the *max_iter* value and got the following plot:

```
max_iter for lowest training loss: 18
max_iter for lowest testing loss: 13
```



Based on this plot, the training and testing losses exponentially decrease from *max_iter* values of 1 to 15. From values of 16 and onwards, both training and testing loss flat line. This indicates that the model needs at least 15 iterations to converge on both data sets; any iterations fewer and the model will not converge, any iterations greater and the model will iterate an excessive amount of times. Therefore, I decided to go with a *max_iter* value of 15.

Creating the Final Model

Putting all feature transformations and parameter fine tuning together, I made a final model that had the following results:

```
Final Model Error on Training: 0.04782717890222892
Final Model Accuracy on Training: 0.9844791666666667

Final Model Error on Testing: 0.11081855515413068
Final Model Accuracy on Testing: 0.975
```

Surprisingly, this model performs slightly worse on the training and testing data than the model with just the three forms of feature transformation and no parameter fine tuning. I tested both models on the provided testing data and submitted the predictions of both models to see how they rank on the Gradescope Leaderboard. Strangely, the model with parameter fine tuning had a much lower loss rate (0.0214999) than the model with just feature transformation (0.026). This inconsistency of the final model producing much worse for known datasets and much better for

unknown datasets may be attributed to the “guesstimates” performed to get the optimal *C* and *max_iter* values. Looking back at those plots, there wasn’t a value for either parameter that minimized both training and testing error, so I had to try and pick such values that didn’t prioritize minimizing training loss too much over testing loss, and vice versa. Obviously, the parameter values I picked were not the most optimal, as the training and testing errors increased and the respective accuracies decreased, but I think they were good enough as they resulted in a 17% decrease in error on the unknown dataset (Part Three).