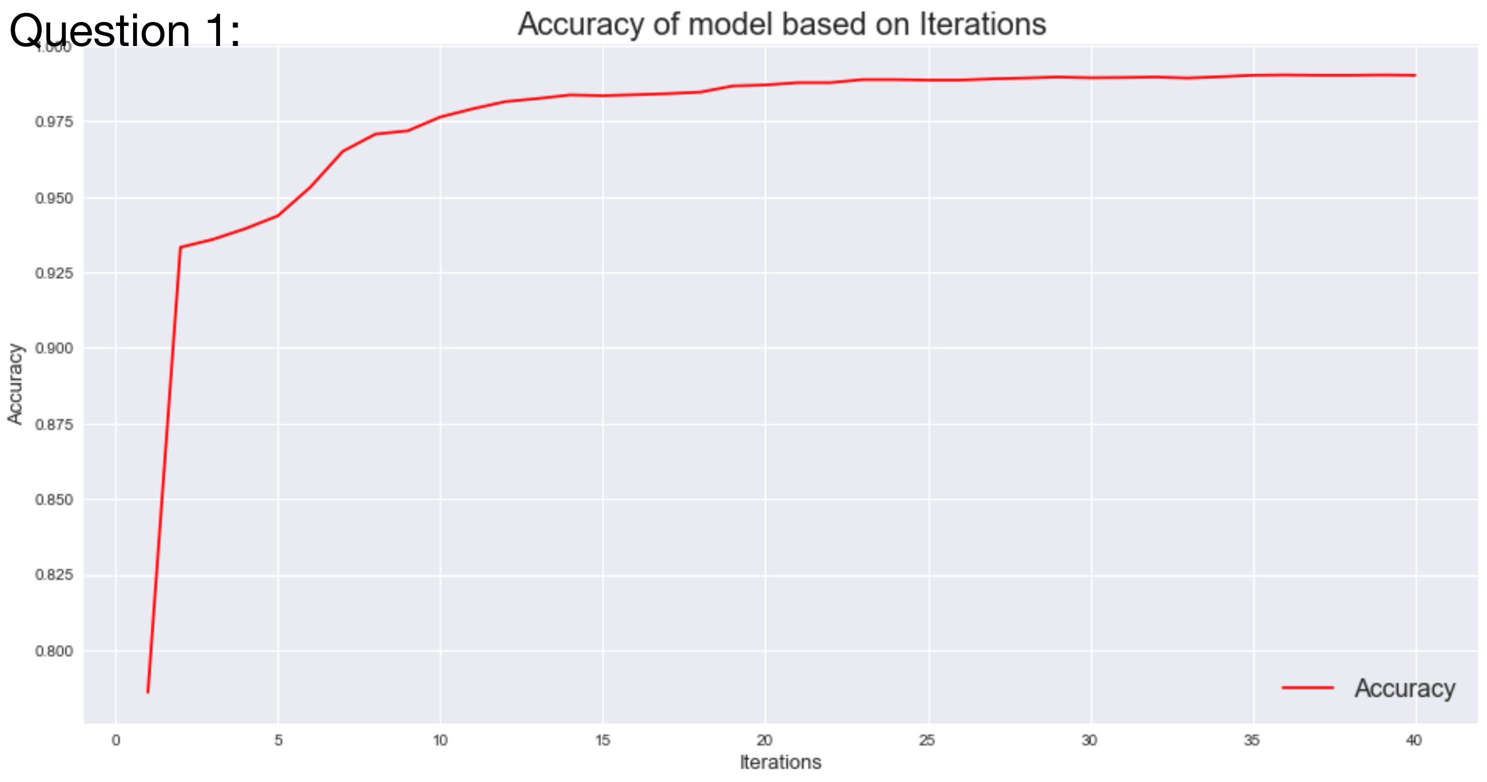
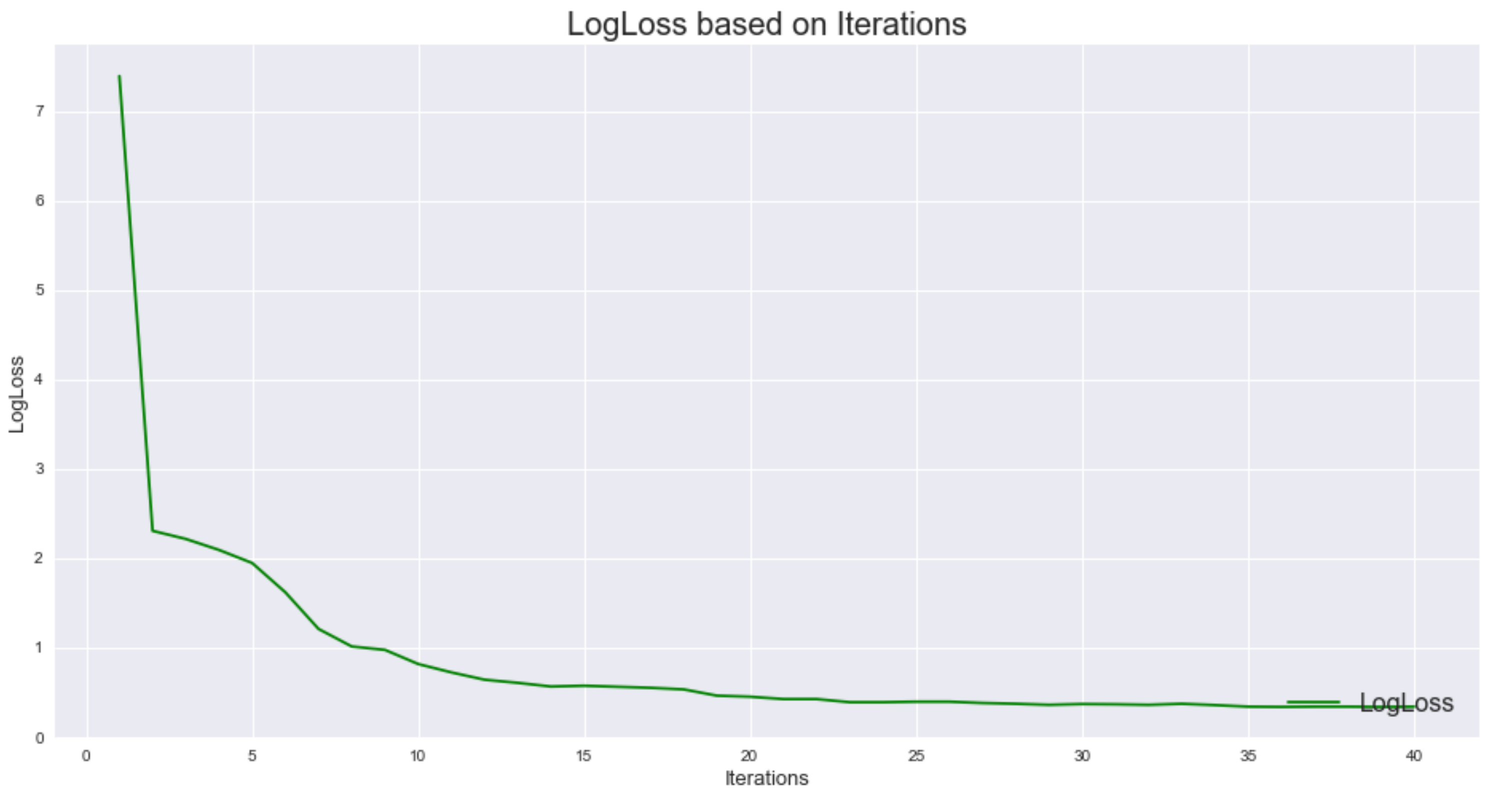
# Part One: Logistic Regression for Digit Classification

Question 1:

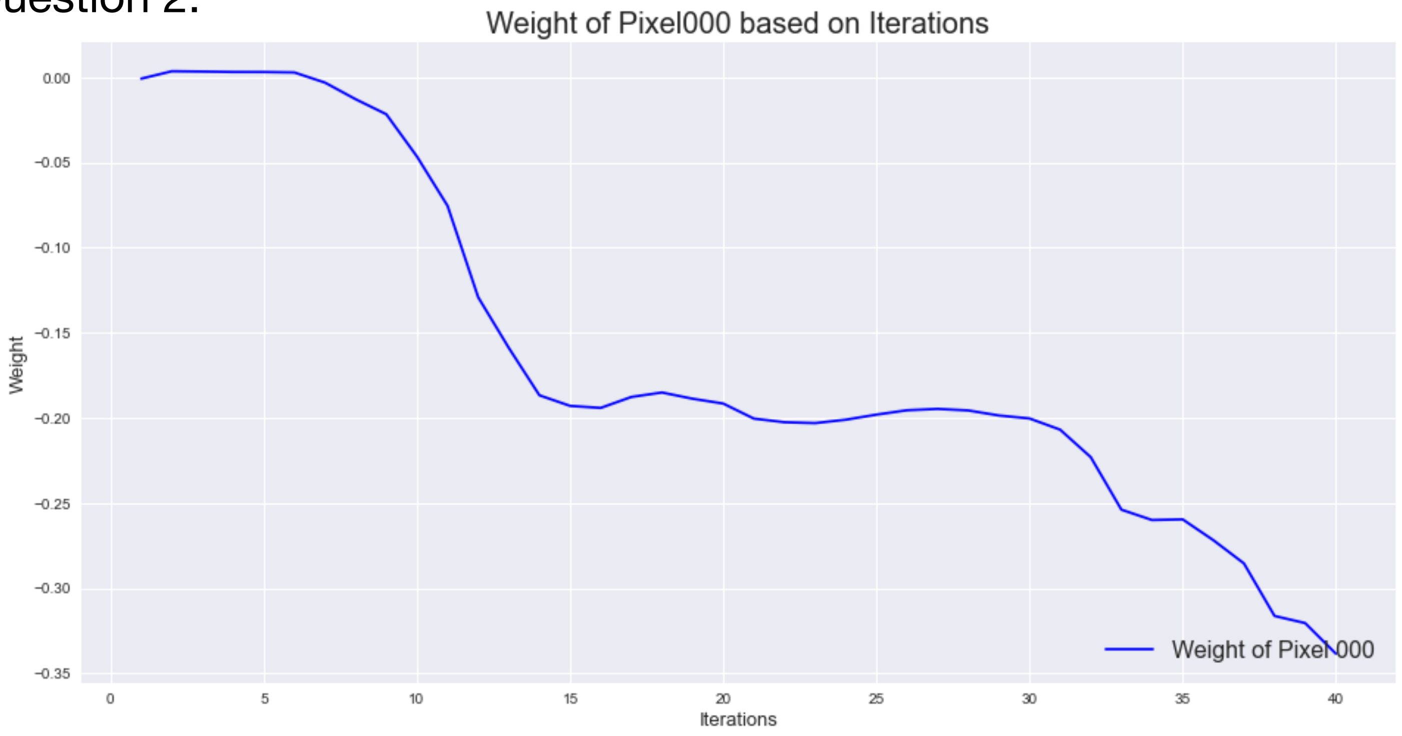


Based on our plot we can see accuracy increasing as we our iterations increase. This means that the model is better fitting the training data. Which can also cause overfitting of data it hasn't seen yet.



Based on our plot we can see that the Log loss is decreasing as the number of iterations increase. This means that the model is getting better maximum likelihood after each iteration. The biggest drop in log loss can be seen in the first few iterations meaning the model does a really good job changing up the model in a way has a really big decline in log loss. The lower the log loss however doesn't mean the model is better because one small mistake in the calculation of log loss can affect the whole log loss. Meaning if most of the predictions are very close to 0 loss but there is one that's really off, it can affect the log loss a lot.

## Question 2:

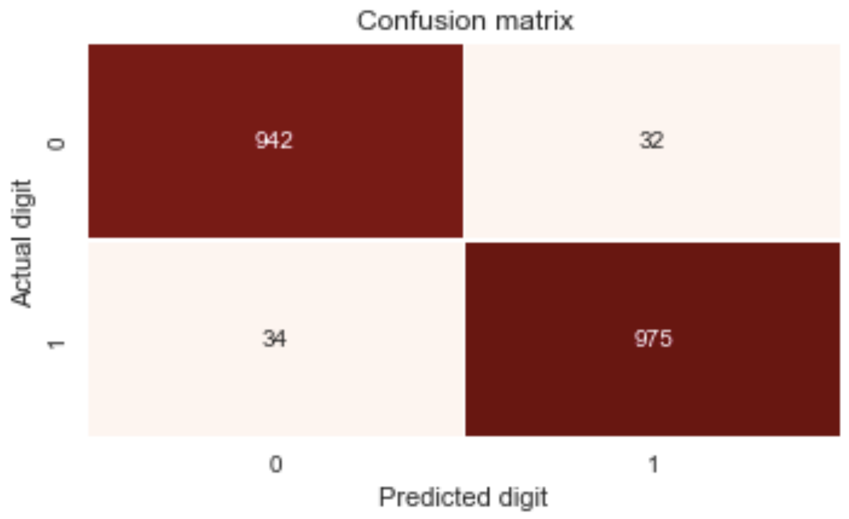


Based on the plot above we can see that the coefficient is changing for pixel 000. It decreases (getting more negative) as the iterations increase. This means that log odds is becoming more and more negative lowering the chance of the event happening. In this case the event being classified as a 9.

## Question 3:

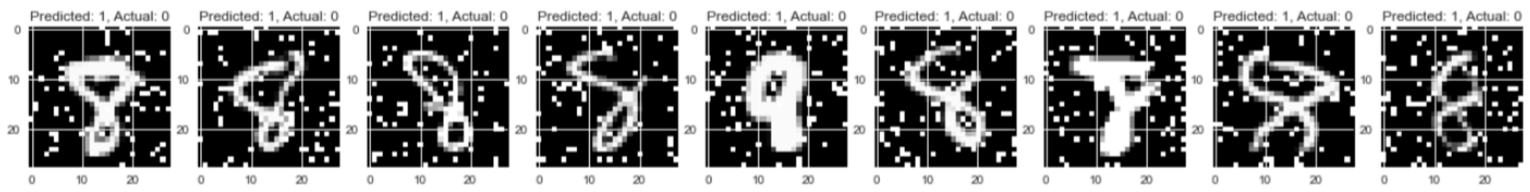
| C | LogLoss | Training Accuracy | Testing Accuracy |
|---|---|---|---|
| 1.000000e-01 | 1.149564 | 0.985339 | 0.966717 |

This shows the C parameter with the lowest LogLoss on test data, with its corresponding training and testing accuracy. The higher the C value, the more weight the model gives to the training data basically telling it to trust the data in the training set a lot more. When the C is lower it's gives more weight to the complexity penalty. Higher C value overfits the data to Training set.

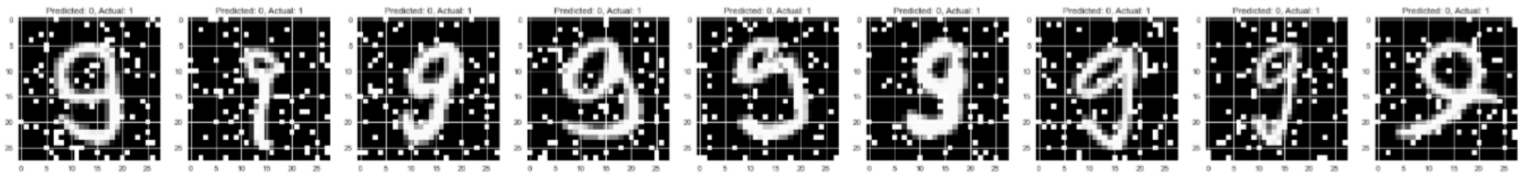


The confusion matrix is for the best C value seen above. It seems to do well in the testing and training accuracies reported above. A quick glance at the confusion matrix also confirms the accuracies seen, and the model did a good job in classifying if a digit is a "9" or not.

## Question 4:

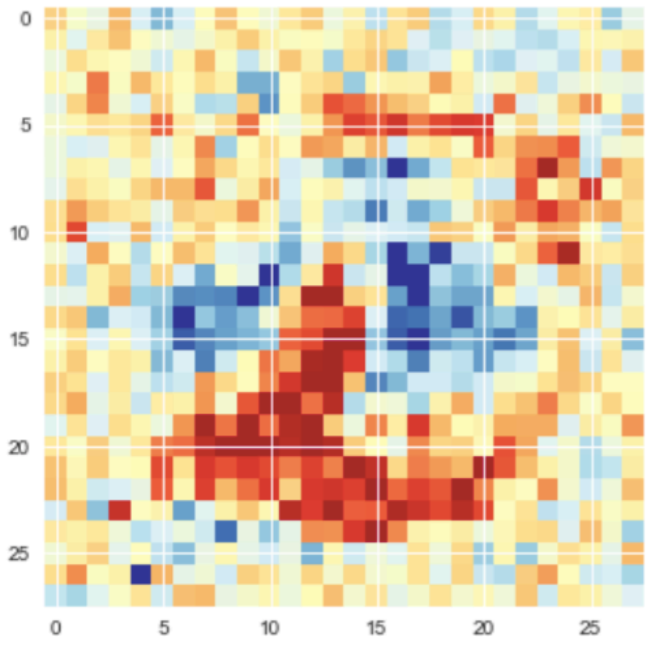


These pictures above are the false positives (Predicted:1, Actual:0) it looks like the classifier is misinterpreting the bottom part of the 8's. It looks like the bottom of the 8's aren't prominent enough in the false positives.



These pictures are the false negatives (Predicted:0, Actual:1). It looks like the bottom of the 9 is too close to the top of the 9 making it sometimes seem like an 8.

## Question 5:



From this image we can see that the negative coefficients which are labeled in red is trying to find the bottom loop of the number 8. This means that if that loop is present in any of the pictures it's less likely to be a 9, and since it's dark red it's a higher negative coefficient meaning it's chances are even lower to be a 9. The blue is representative of the positive coeffiecients and from the picture above you can see it outline a somewhat of a shape of a 9.

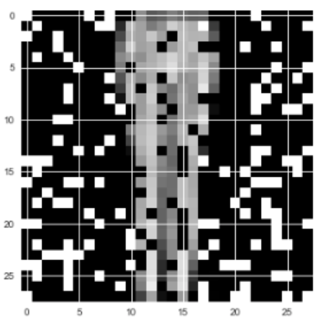
# Part Two: Trousers vs. Dresses

## Summary of models:

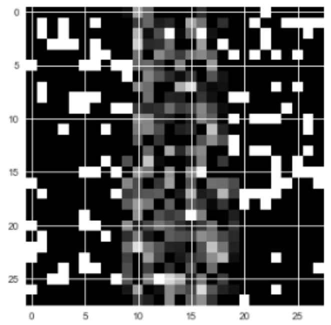
| Transformation | Test Error (Train-Test Split) | Testing (Gradescore) Error |
|---|---|---|
| **Baseline (LogisticRegression w/o changes)** | 0.9374 | 0.9335 |
| **Feature: Count of white pixels** | 0.9408 | 0.9285 |
| **Feature: Count of white and black pixels** | 0.9429 | 0.9285 |
| **Feature: Get rid of white pixel (noise)** | 0.9620 | 0.9651 |
| **Feature: Get rid of white pixel (noise) + random search CV (Best: C=0.09601, max_iter = 674; cv=5)** | | 0.9740 |

## About our data:

To start this part of the project, first I needed to see what type of data I was working with. I already knew that it was image data, but how did they actually look:



Trouser

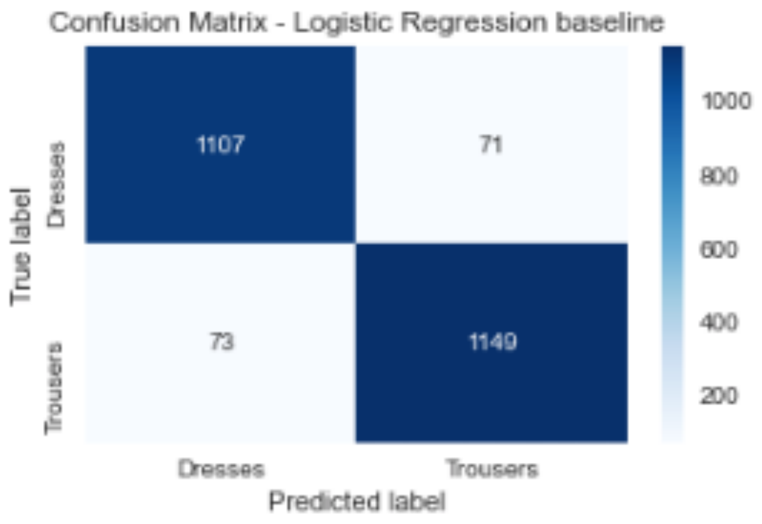


Dress

Good thing about this data, is it's already scaled to being between 0's and 1's and it's in black and white. This means that we don't need to perform any rescaling as all of the features follow the same scale. There are 6000 dresses and 6000 trousers in our dataset. It also looks like there are significant noise injected to each of the pictures.

## Baseline:

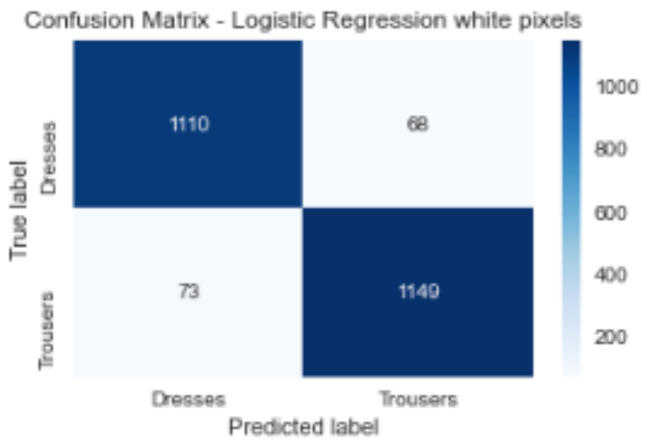
We need a baseline to compare our results. One way to approach this situation is to just guess the majority class, but in this case we have 6000 dresses and 6000 trousers so there's no majority and if we were to guess we'd only be right 50% of the time. So let's start off with a baseline of Logistic Regression without doing any feature transformation or any changes to the Logistic model other than the basic parameters that it is imported with. We can use 20% test size using a train/test split and random state of 34. These parameters will help us stay consistent in splitting the dataset. Our baseline accuracy is 0.93625 with a F1 score of 0.9374. The accuracy is pretty good for a baseline and the F1 score which represents the balance between precision and recall, is pretty high as well. Looking at the correlation matrix (below) we have similar amounts of false positives (76) and false negatives (77). Now that we have our baseline, we can try some feature transformations.



## Feature Transformations:

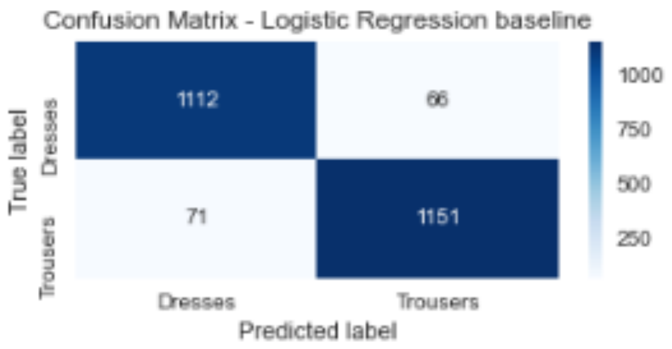
**COUNTING WHITE PIXELS:**

For the first feature transformation, I decided to do one of the suggested transformation: which was to count the overall numbers of white or black pixels. To do this, the algorithm would go through each row and iterate through their respective columns (pixels) to find any 1.0's (whites) and would sum up how many 1's it saw in that specific column. This would create a whole new feature with how many white pixels there were for that particular row. Once this was done, another logistic regression without any changes to the parameters were done. With this I got a testing (of train-test split) accuracy of 0.9408 which was a slight increase from baseline. Looking at the confusion matrix it looks like we did better at predicting trousers correctly so our false positives went down. When testing out this model in



grade scope though the accuracy had gone down from the baseline model, from 0.9335 to 0.9285. So this feature transformation didn't help as much as I had hoped.

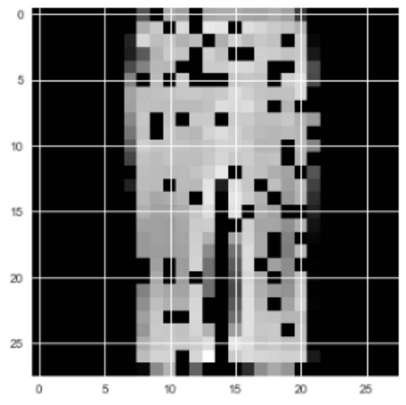
**COUNTING WHITE AND BLACK PIXELS:**



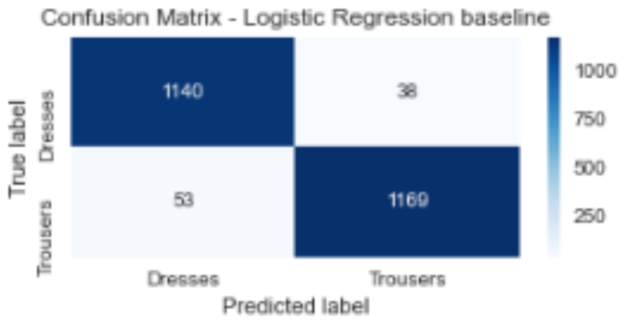
Adding to the previous transformation, I wanted to see if the addition of counting of black pixels would help at all. Using the same method as before we would iterate through the rows and columns to find any 0's which would pertain to the black pixels and sum it up for each row. Using this feature our testing accuracy for the train test split went up only a little bit to 0.9429 from 0.9408. Our false positives stayed the same but our false positives went down by a few. Testing this model out on grade scope gave us the same accuracy as the last feature transformation which was to just count white pixels, which was still lower than our baseline.

**GETTING RID OF WHITE NOISE:**

It was now back to the drawing board, the two feature transformation I tried didn't help much, instead it lowered our accuracy on the testing on grade scope. Looking more closely to the pictures, I realized that a lot of the white pixels that were present in the images were noise and was throwing off the model. What if we turned all of the white pixels into black pixels. Following a similar method to the counting of the white/black pixels, I iterated through each row and column to change any 1's to 0's. These are how the pictures looked after:



Trousers



That looks so much better without the white noise. After running a Logistic Regression on these images, with no parameter tuning, the testing accuracy on the train-test split went up to 0.9620! Which was almost a .02 increase in our accuracy, which is a lot! Looking at our confusion matrix we can see there is a significant decrease in both the false positives and the false negatives, meaning our model was doing a better job in classifying if an image was a trouser or a dress. After uploading the predictions to grade scope the model was able to achieve a 0.9651 accuracy!! Almost a 4 percent increase from our previous models and 3 percent better than our baseline model! This is an amazing step. Now that we were able to optimize our features we can try to combine the removal of the white noise and the count of black pixels. Surprisingly enough, it did not help our model to add the black pixels instead it lowered our accuracy, which could mean that it was overfitting the training data.

---

## Parameter Tuning

Now that our features have been optimized without parameter tuning, the next logical step is to tune our parameters. The way I approached this was to try GridSearchCV. GridSearch enables us to give a set of parameters for the model to try and see what combination of parameters give us the best accuracy. To start it off I gave np.logspace(-9, 6, 31) as a grid to try out the C in the hyper parameter. The 'C' refers to how much the classifier should trust the training data. The lower the 'C' it gives more weight to the complexity penalty, the higher the 'C' the more trust in the training data the model will have. We can also specify which penalty the model can try "L1" or "L2". Lastly I also tried a list of different max_iter parameters, 100, 500, 700, 1000. The problem with GridSearch is when you have a lot of data and lot of parameters you want to try out it takes a longer time. Even after an hour of waiting our GridSearch wasn't finished. Turns out there's a better way to tune our parameters for larger datasets, RandomSearchCV. It is the same idea of GridSearch but instead of trying every set of combination it'll randomly choose from the sets and see which is the best. In this case it is much faster and some online argue will lead to more accurate results than GridSearch. After our first RandomSearch CV (cv=3 to keep consistency) our best C was 1.14 and max_iter of 700 and penalty of L2. To save some time on the report, all of our RandomSearchCV's chose L2 as the penalty of choice. Once we got these results from the RandomSearch CV I wanted to get more granular and chose to try a list of max_iter from 600 - 800. After a few tweaks and a couple of different times we tried the RandomSearch our best and final model came with a C of 0.096 and max_iter of 674. Using this we were able to achieve our highest accuracy on the unseen data of 0.9740. The reason the table above doesn't have a training score is because we used the whole data set without splitting to run the random searchCV for it's final and best model. Finally, I also tried shifting images using data augmentation but the accuracy was lowered, which goes to show sometimes the more simple transformation is better for the model.