CS6316 Application Project

Yizhen Chen

Mitchell Campbell

Henry Muhlbauer

Table of Contents

[Part I 3](#_Toc58524886)

[Logistic Regression 3](#_Toc58524887)

[Implementation & Hyperparameter Performance Comparison 3](#_Toc58524888)

[Analysis 4](#_Toc58524889)

[K Nearest Neighbor 5](#_Toc58524890)

[Implementation & Hyperparameter Performance Comparison 5](#_Toc58524891)

[Analysis 6](#_Toc58524892)

[SVM 8](#_Toc58524893)

[Implementation & Hyperparameter Performance Comparison 8](#_Toc58524894)

[Analysis 8](#_Toc58524895)

[Decision Tree 9](#_Toc58524896)

[Implementation & Hyperparameter Performance Comparison 9](#_Toc58524897)

[Analysis 10](#_Toc58524898)

[Random Forest 12](#_Toc58524899)

[Implementation & Hyperparameter Performance Comparison 12](#_Toc58524900)

[Analysis 13](#_Toc58524901)

[Boosting 15](#_Toc58524902)

[Implementation & Hyperparameter Performance Comparison 15](#_Toc58524903)

[Analysis 16](#_Toc58524904)

[Part II 18](#_Toc58524905)

[Implementation & Hyperparameter Performance Comparison 18](#_Toc58524906)

[Analysis 19](#_Toc58524907)

[Summary of Findings 19](#_Toc58524908)

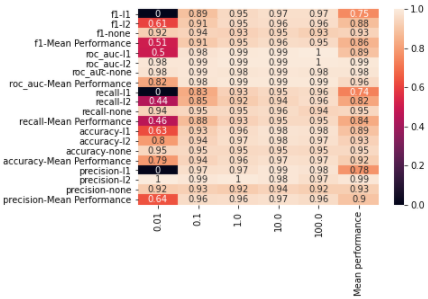
# Part I

## Logistic Regression

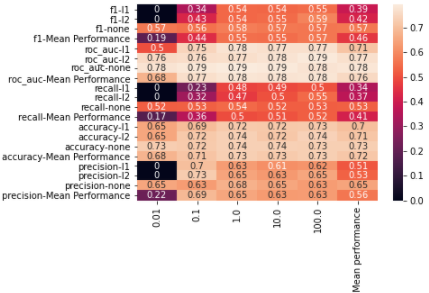
### Implementation & Hyperparameter Performance Comparison

Logistic regression was implemented on both datasets by first transforming any categorical variables to binary variables and then normalizing the data using the MinMaxScaler() function from the sklearn library. Logistic regression was then applied to both datasets using 10-fold cross validation, with the regularization norms and penalty strengths as hyperparameters. The averaged results across the cross validation were then recorded into a DataFrame object and plotted as heat maps, with the x-axis representing the penalty strength and the y-axis representing performance criteria for the different regularization norms:

**Dataset #1**

****

**Dataset #2**

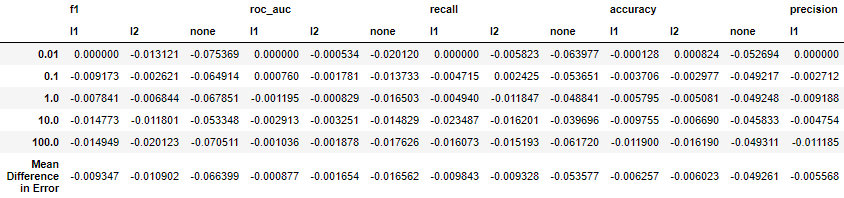
****

Analysis

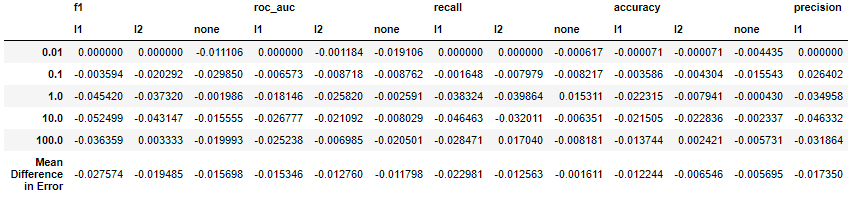
Based on the above heatmaps our analysis of the hyperparameters has led us to conclude that both L1 and L2 regularization led to a slight improvement in the performance of the models. In the first dataset, L1 and L2 regularization with a penalty strength of 100 seemed to outperform slightly for all performance criteria except for recall, and in the second dataset, L2 regularization with a penalty value of 100 outperformed slightly across all criteria as well. Note that for small penalty values less than 0.1, the algorithm often did not converge if regularization was specified.

We believe that these results represent the bias-variance tradeoff, in that the models for both datasets seemed to have benefited slightly from adding regularization. The role regularization plays in reducing variance can also be seen in the difference between training error and testing error:

**Dataset #1**



**Dataset #2**

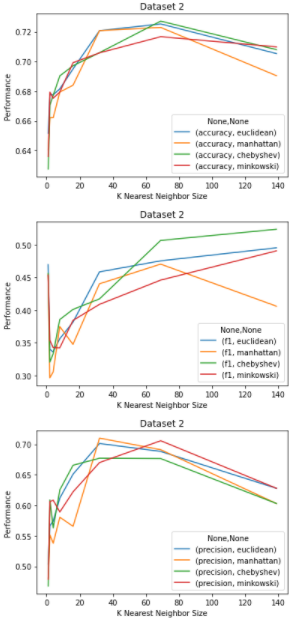
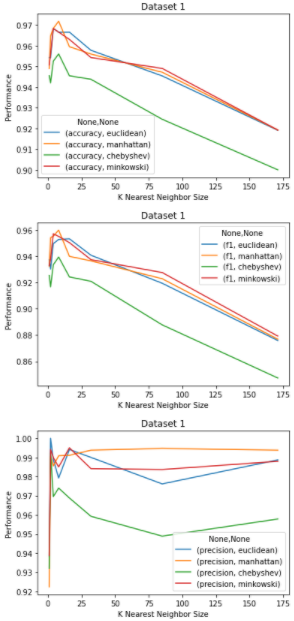


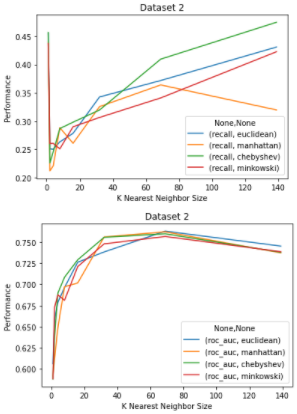
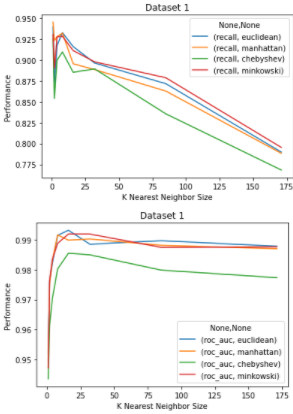
Although not true in every case, there is a general trend of the training-testing error difference being of smaller magnitude when regularization was specified. For instance, note how in Dataset #1 how the difference between testing and training accuracy is -0.006690 for L2 regularization with a penalty value equal to 10.0, whereas this difference was -0.045833 when regularization was not applied. Although the improvements are only slight, I think we can conclude that regularization improves the testing accuracy for both datasets.

## K Nearest Neighbor

### Implementation & Hyperparameter Performance Comparison

The K Nearest Neighbor algorithm was implemented on both datasets by first transforming any categorical variables to binary variables and then normalizing the data. The KNN algorithm was then applied to both datasets using 10-fold cross validation, with the different distance metrics and possible neighbor sizes being used as hyperparameters. The averaged results across the 10-fold cross validation were then recorded into a DataFrame object and plotted:



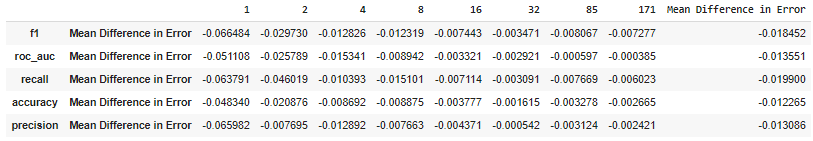


### Analysis

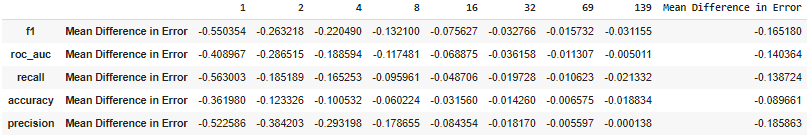
Based on the above graphs, our analysis of the hyperparameters has led us to conclude that the distance metric used in the KNN algorithm does not have a significant impact on the performance of the model. This is evident by the fact that all distance metrics trended in the same direction no matter what performance criteria was being used. Additionally, smaller neighbor sizes between 2 and 32 tended to perform better in the first data set whereas larger neighbor sizes between 32 and 140 tend to perform better for the second dataset. We believe that this is a sign of how the bias-variance tradeoff manifests itself within these datasets.

For instance, note how the difference between training error and testing error is much larger for the second dataset than it is for the first at small sizes of *k*:

**Dataset #1**

****

**Dataset #2**

****

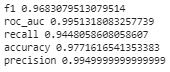
Note how across all performance criteria, the training error is far higher than the testing error for dataset 2 than it is for dataset 1, especially at smaller values of *k*. Our suspicion is that the first dataset is a more representative sample than the second dataset. As such this may explain why the KNN model for dataset 2 performs better for larger sample sizes, as that helps reduce the variance in the model. In contrast, since the first dataset seems a more representative sample, it seeks to minimize its bias as it is not as susceptible to variance in the model.

## SVM

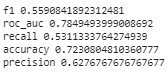
### Implementation & Hyperparameter Performance Comparison

We opted to implement a linear kernel in our SVM model for both dataset #1 as well as dataset #2, as their predictive power appeared like what was given by other models. As such we felt implementing a nonlinear kernel would give additional complexity to the model that could lead to overfitting. The average results for the 10-fold cross validation are below:

**Dataset #1**



**Dataset #2**

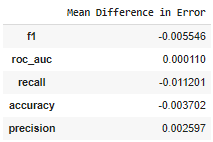
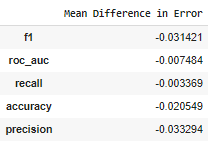


### Analysis

The results of the linear SVM model appear similar to the results of our other models. In the future it may be worth the effort implementing a non-linear kernel for dataset #2 to see if that improves the performance, although in theory this would most likely hurt performance as dataset #2 is more prone to overfitting.

Regarding the bias-variance tradeoff, both the training and testing scores seem to be relatively similar. We would expect these differences to increase in magnitude if we were to switch to nonlinear.

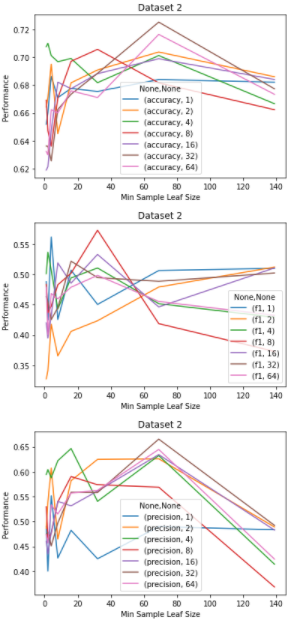
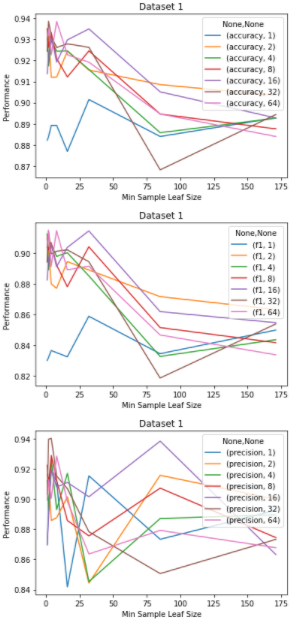
**Dataset #1 Dataset #2**

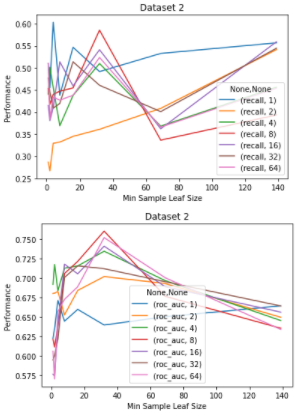
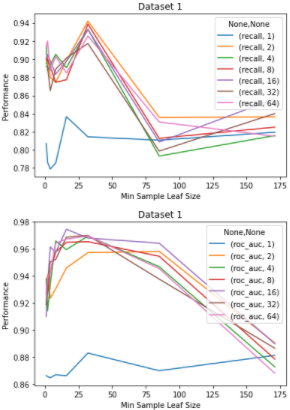
 

## Decision Tree

### Implementation & Hyperparameter Performance Comparison

Decision trees were implemented on both datasets by first transforming any categorical variables to binary variables and then applying 10-fold cross validation, with tree depth and minimum leaf sample size being used as hyperparameters. The averaged results across the 10-fold cross validation were then recorded into a DataFrame object and plotted:





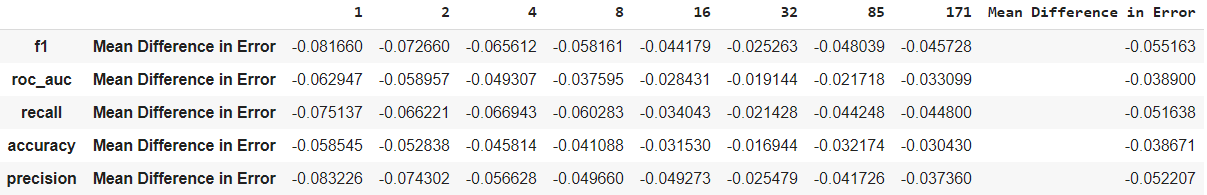
### Analysis

The difference in performance between decision trees with different max depths and minimum sample leaf sizes is less clear than in other models. For clarification, the x-axis corresponds with the minimum sample leaf size and each series corresponds with a different max depth. For dataset #1, the highest performance for each criterion seemed to be clustered around minimum sample leaf sizes between 1 and 32 for most max depths. Also, for dataset #1 a max depth of 1 always seemed to perform the worst across all criteria, although it is worth noting that no single max depth size was purely dominant. The relation between performance and the hyperparameters is equally unclear for dataset #2. No single combination of max depth and minimum leaf size seemed purely dominant, although if we had to pick a single strategy, we believe that a max depth of 8 and minimum leaf size of 32 offered the best performance across all criteria.

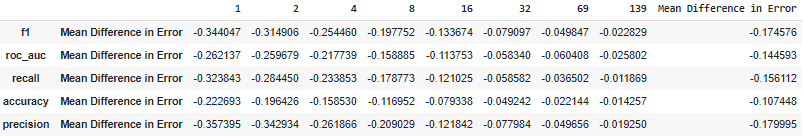
In our python code, we plotted what we believed were the best decision trees for each dataset. Although not presented in this document, it is worth mentioning that the root node in the tree for dataset #1 is variable x[20] and the root node in the tree for dataset #2 is x[8].

It is worth noting that the mean difference between training and testing error for this model is representative of the bias-variance tradeoff. For instance, note in the below tables how the difference between training and testing error is of larger magnitude for small minimum sample leaf sizes.

**Dataset #1**

****

**Dataset #2**

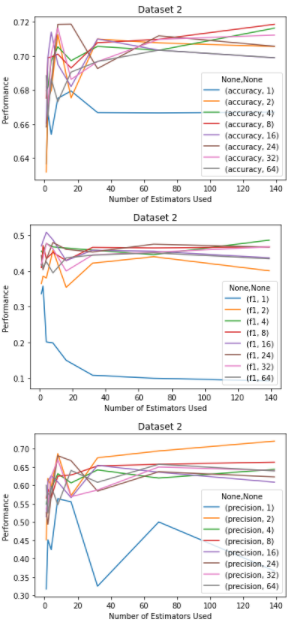
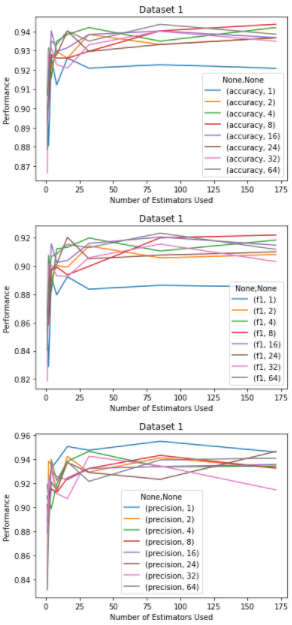


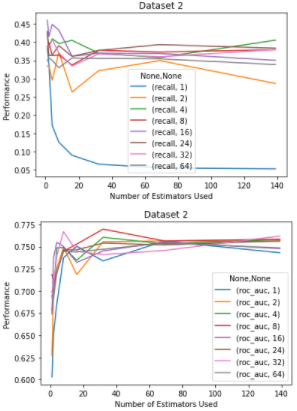
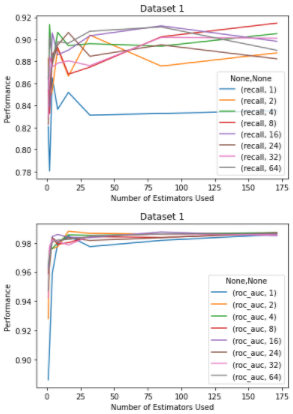
The reason why smaller minimum sample leaf sizes tend to have larger differences between training and testing error is because they are more prone to overfitting. As mentioned in the KNN analysis, these tables seem to confirm that the second dataset is more prone to high variance.

## Random Forest

### Implementation & Hyperparameter Performance Comparison

Random forest was implemented on both datasets by first transforming any categorical variables to binary variables and then applying 10-fold cross validation to both datasets, with number of trees and tree depth being used as hyperparameters. To simplify our analysis, the minimum leaf sample size was fixed at 32 to reduce the number of hyperparameters to keep track of. We chose 32 to be our minimum leaf sample size based on our decision tree analysis. The averaged results across the 10-fold cross validation were then recorded into a DataFrame object and plotted:



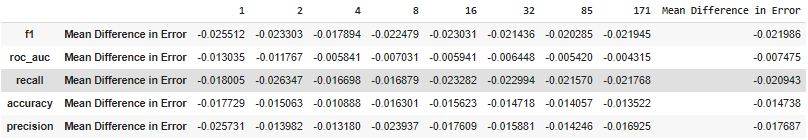


### Analysis

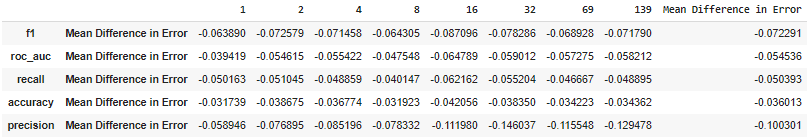
For clarification, the x-axis represents the number of trees used in the random forest and each series corresponds to a different maximum depth for each tree. In general, random forests with trees with a max depth of 1 almost always performed the worst no matter how many of them were included. For dataset #1, most criteria seemed to improve as the number of trees included increased up to around 8 trees, after which it mostly held steady. This appeared similarly true for the second dataset as well in that no consistent increase across all performance criteria could be found when more than 8 trees were used.

We believe that this is evidence of the bias-variance tradeoff. Notice how the mean difference in error stays relatively constant despite the varying number of trees used. We believe that this may be because each individual tree was already accounting for the inherent variance since the minimum leaf sample size was set to 32. Perhaps if this size were 1, we would see a more drastic increase in performance proportional to an increase in the number of trees used.

**Dataset #1**

****

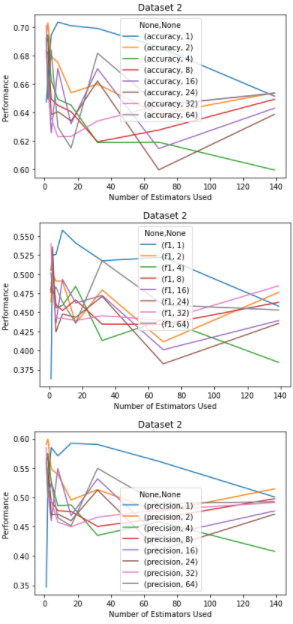
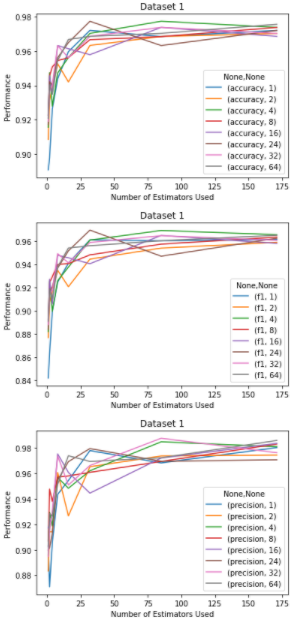
**Dataset #2**

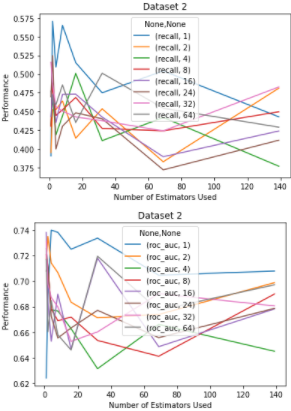
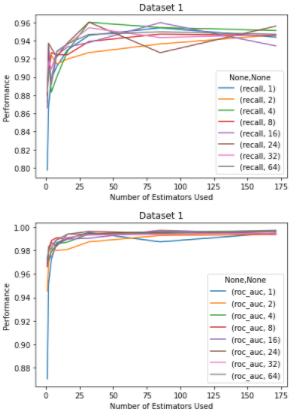
****

## Boosting

### Implementation & Hyperparameter Performance Comparison

AdaBoost was implemented using decision trees on both datasets by first transforming any categorical variables to binary variables and then applying 10-fold cross validation, with number of estimators and tree depth being used as hyperparameters. The averaged results across the 10-fold cross validation were then recorded into a DataFrame object and plotted:



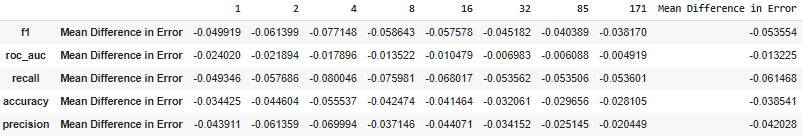


### Analysis

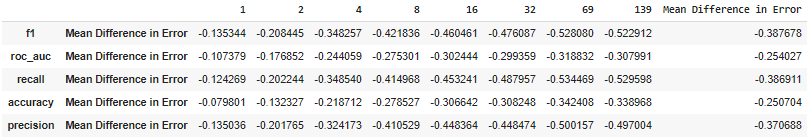
For clarification, the x-axis represents the number of estimators used in AdaBoost and each series corresponds to a maximum depth for each tree. For the first dataset, the AdaBoost can be seen converging towards an optimal value as the number of estimators increases. From this we can infer that max depth size of each tree estimator has less of an impact on the performance than the number of estimators used in Adaboost. On the contrary, in the second dataset a max depth size of 1 was consistently the best performer across all criteria. We believe that in this case when the tree estimators had a depth size of 1, they were less likely to overfit and therefore perform in Adaboost.

This appears to be evidence of the bias variance trade-off. As can be seen in the below tables, the second dataset appears more susceptible to overfitting, in that the difference between the training and testing errors is of larger magnitude in the second dataset. Additionally, it is worth noting that the differences become smaller in magnitude as number of estimators increase, which in turn helps prevent overfitting.

**Dataset #1**

****

**Dataset #2**

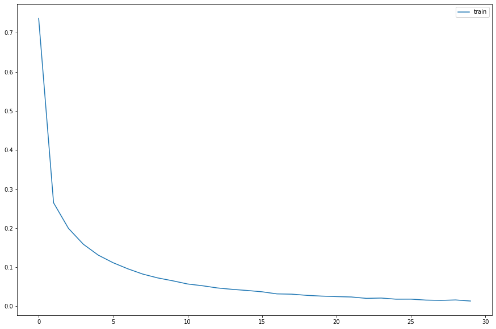
****

# Part II

### Implementation & Hyperparameter Performance Comparison

We implemented our neural network using PyTorch by first loading the training and testing data into a PyTorch Dataset structure, which also had the benefit of applying rotational transformations to the data to provide for a more robust training set. We then used the DataLoader function to create batches of the MNIST data of size 100. Then we defined our neural network to be a series of sequential layers, which contained two hidden layers with sigmoid activation and one output layer.

To train the neural network, we used CrossEntropyLoss as well as the Adam optimizer. One thing to note about the CrossEntropyLoss is that it automatically accounts for the softmax activation in the final layer per PyTorch’s documentation. We trained the neural network over 40 epochs using 4 sets of hyperparameters to examine the effects of using different learning rates and hidden layer sizes. Note that PyTorch by default randomly initializes the weights in its linear layers so no further action was required on our part in order to examine the effects of weight initialization on performance. The hyperparameters and results of these four tests are as follows:

**Test 1** CrossEntropyLoss vs Epoch

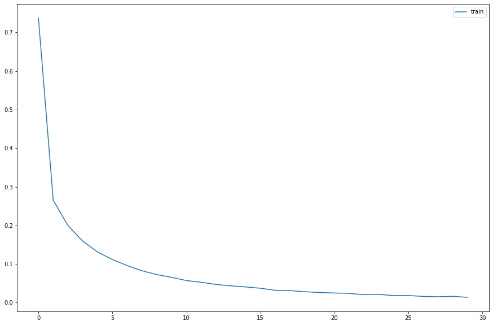
"lr": 0.001,

"hidden\_size": [256, 128],

"initializations": "default",

"accuracy": 98.140

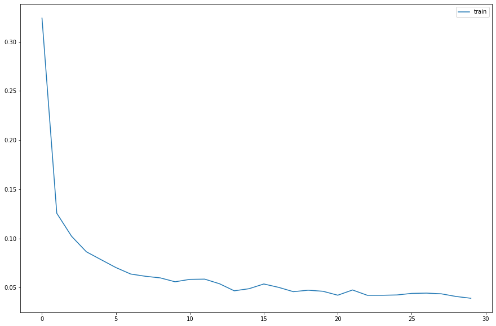
**Test 2**

 "lr": 0.005,

"hidden\_size": [256, 128],

"initializations": "default",

"accuracy": 98.280

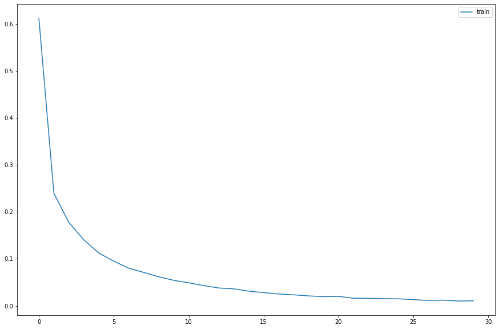
**Test 3**

"lr": 0.01,

"hidden\_size": [256, 128],

"initializations": "default",

"accuracy": 97.200

**Test 4** CrossEntropyLoss vs Epoch

"lr": 0.001,

"hidden\_size": [512, 256],

"initializations": "default",

"accuracy": 98.340

### 

### Analysis

Based on the above test results, it appears that a smaller learning rate (less than or equal to 0.01) as well as larger hidden layer sizes (512, 256) tended to predict the highest percentage of images correctly in the test dataset. The third test had the highest learning rate and was the only one with an accuracy lower than 98%. Similarly, in the fourth test had the largest hidden layers and had the highest accuracy, although its accuracy is only slightly higher.

# Summary of Findings

For Part I, we found that no single algorithm outperformed all others across all performance criteria for both datasets. In general, we found that the F1-measure and recall tended to be the criteria that had the lowest scores across both datasets and for all algorithms. The low recall may be due to our algorithms being relatively bad at identifying true positives, and since the recall is low it is understandable why the F-1 measure is also low, as that can be understood as the measure that balances recall with precision. At least for Part I, there appears to be a tradeoff between recall and precision, as evidenced by relatively higher precision values. Also, it is worth noting that algorithms on the second dataset were more susceptible to overfitting, and as such tended to benefit more from methods that reduced variance such as increasing neighbor size in KNN.

For Part II, we believe our network to be a relatively strong classifier as it was able to correctly identify approximately 98% of MNIST digits. One observation we had was that while increasing the size of layers can improve accuracy, these increases seem to diminish over time as evidenced by the small performance difference between a (512, 256) network and a (256, 128) network.