INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 3

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL774

Prof. Parag Singla

Due October 27, 2021

Contents

Chapter 1

Decision Trees

1.1 Decision Tree Construction

I used a recursive algorithm to construct Decision Tree using the training data. At each node, I first check if the incoming data is *pure* or not. If the data is *pure* (all labels belong to a single class), then I just return a **leaf** node with the label as the predicted value. Otherwise, the data is *impure* and hence, I choose *best* attribute, based on which I split the incoming data and construct the sub-tree recursively. Here, the *best* attribute is the one that leads to minimum *entropy* after split. Below, I mention the algorithm specifications in concise form:

- Stopping Criteria: Stop when all terminal nodes are *pure*. Hence, grow tree to maximum possible depth.
- Splitting Method:
  - Numerical Attribute: 2-way split on median
  - Categorical Attribute:
    - *Without one-hot encoding:* n-way split
    - *With one-hot encoding:* 2-way split

I constructed decision trees with and without one-hot encoding and noted down the results. The training, validation and test set accuracies obtained using the full-grown tree (for both cases) are specified below:

Without using one-hot encoding:
Training Accuracy (in %): 100.0
Validation Accuracy (in %): 87.15170278637771
Test Accuracy (in %): 87.14886087148861

With one-hot encoding:
Training Accuracy (in %): 100.0
Validation Accuracy (in %): 87.59398496240601
Test Accuracy (in %): 87.03826587038266

The plots for the training, validation and test set accuracies against the number of nodes in the tree are presented next page onwards.

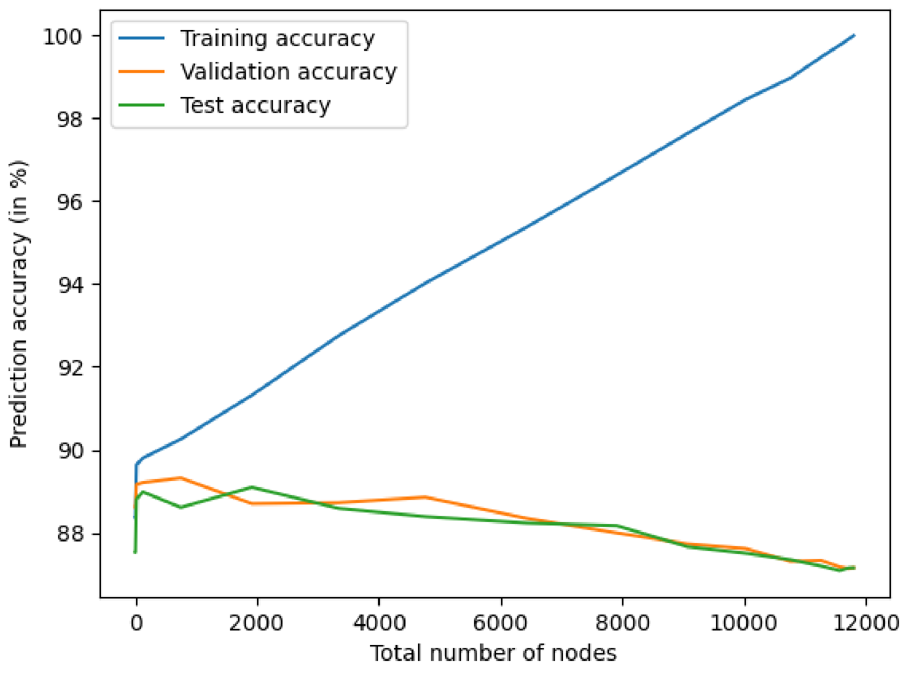


Figure 1.1: Prediction accuracy vs total number of nodes (without one-hot encoding)

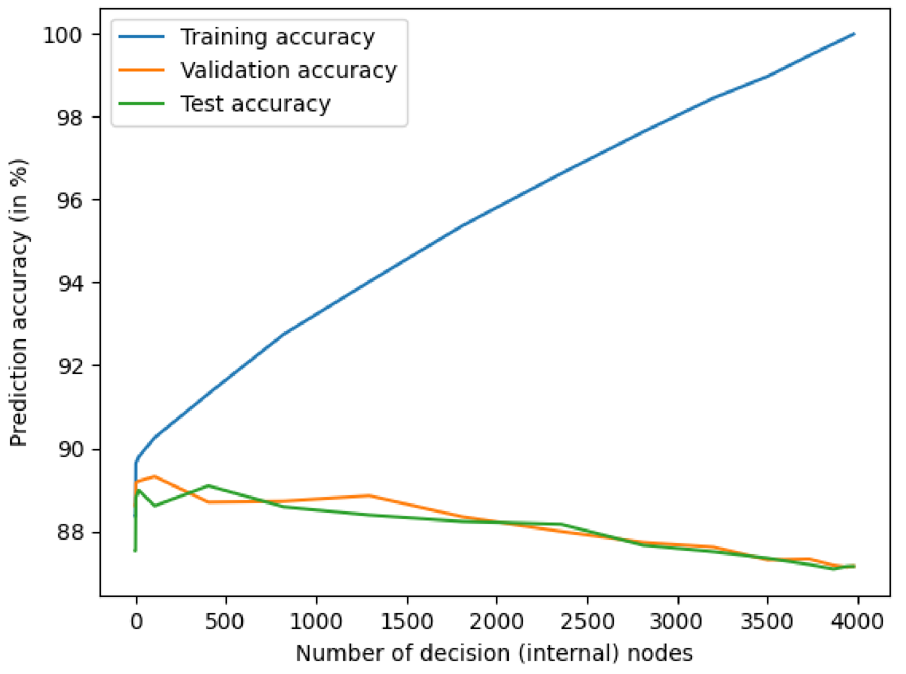


Figure 1.2: Prediction accuracy vs number of decision nodes (without one-hot encoding)

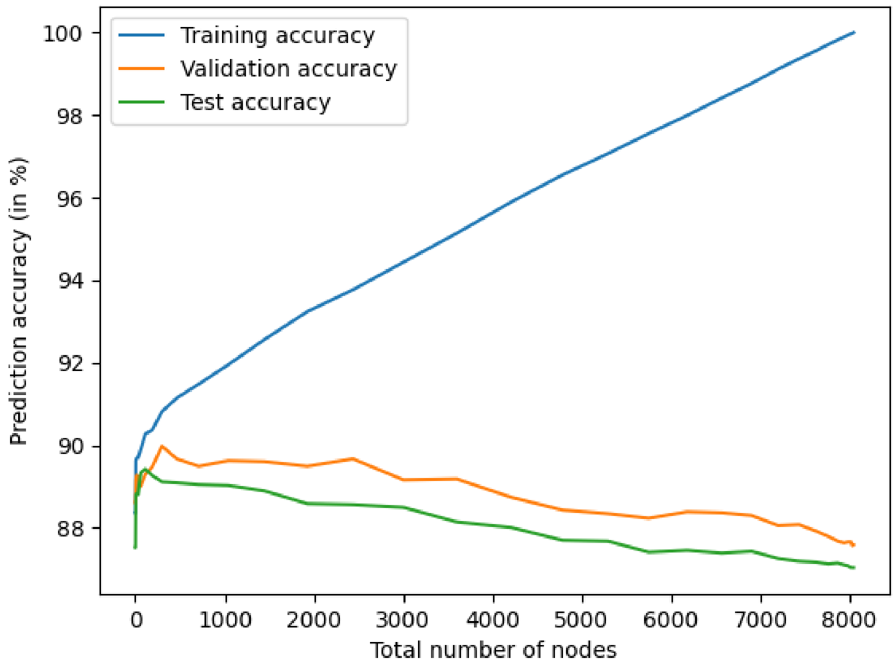


Figure 1.3: Prediction accuracy vs total number of nodes (with one-hot encoding)

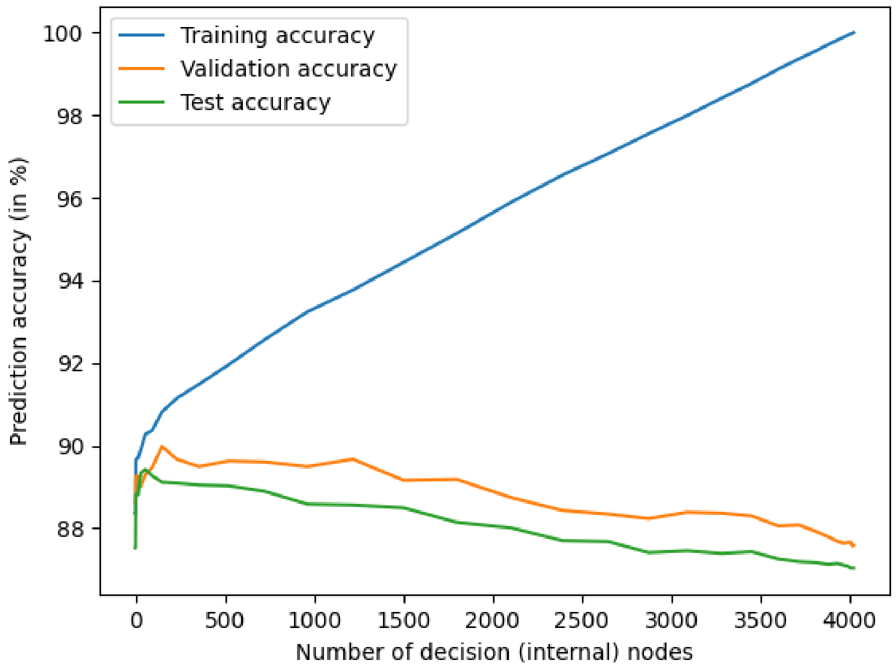


Figure 1.4: Prediction accuracy vs number of decision nodes (with one-hot encoding)

Note: The accuracy values were obtained depth-wise rather than exact node-wise for plotting figures: *Figure 1.1* to *1.4*

Comments:

- First thing I observe is that the final training accuracies for both the trees is 100% (perfect training). This can be explained as I continue growing the tree till all the terminal nodes become *pure*. Since, the prediction is 100% correct for *pure* nodes, therefore, using the decision tree on the same training set will lead to similar *pure* nodes, thus, giving 100% accuracy. Hence, this observation is justified.
- Secondly, I observe that as the tree is grown (in both cases), the training accuracy increases monotonically, but both, validation and test set accuracies first increase to a small peak (88-90%), then start decreasing. In case of normal encoding, this decrease of accuracy is not uniform, and there is some noise. However, in the long run (nodes $\geq$ 5000), the accuracies decrease even when using normal encoding. As far as monotonic increase of training accuracy is concerned, that is expected, as we are refining our training prediction every depth/split by selecting the attribute that minimizes the entropy. Hence, the curve for training accuracy is justified. Coming to plots of validation and test accuracies. Initially, when the number of nodes in the tree is very small, the model is actually *underfitting* the data (lack of enough nodes), and hence, the accuracy increases with increase in number of nodes (leading to initial peak). However, after that, due to continued training, the tree starts overfitting the training data and starts adjusting to training noise, due to which the tree fails to generalize to unseen data (validation and test). Hence, the validation and test set accuracies start declining.
- Lastly, I comment on the actual values obtained. The final validation accuracy is better when I am using one-hot encoding than when I am not. On the other hand, the final test accuracy is better when I am using the normal encoding. Also, even though the total number of nodes with the normal encoding is much greater than with one-hot encoding (12000 $>$ 8000), the total number of *decision* nodes in both the trees are almost the same (around 4000). This is reasonable, as we have not introduced any new features, but just one-hot encoded the original ones. Therefore, though we have more features in one-hot encoding version, majority of the decision branches will actually be redundant and lead directly to leaf nodes. Hence, we can *expect*, the number of decision nodes to be the same in both the cases. I stress however, that the memory requirement of tree formed using one-hot encoding is lower than the one formed using normal encoding, possibly because of lower branching factor.

1.2 Decision Tree Post Pruning

In this part, I used the fully grown tree (*section 1.1*) and iteratively pruned it using a variant of *Reduced Error Pruning.* Below, I specify the details of the algorithm used for pruning:

1. Do a level order traversal of the decision tree and find *bottom-up* ordering of decision (*internal*) nodes.
2. For each node (other than *root*) in ordering found above, do the following:
   (a) Determine accuracy on validation set using the current tree (pre-pruning)
   (b) Determine accuracy on validation set after pruning the *node* (sub-tree) (post-pruning)
   (c) If post-pruning accuracy is $\geq$ pre-pruning accuracy, then prune the *node*
3. Return the resultant decision *tree*

I pruned both the decision trees (with and without one-hot encoding) using the above algorithm and noted down the results. The training, validation and test set accuracies obtained using the resultant tree (for both cases) are specified below:

Without using one-hot encoding:
Training Accuracy (in %): 91.90168104401681
Validation Accuracy (in %): 92.79080053073861
Test Accuracy (in %): 89.49347489493475

With one-hot encoding:
Training Accuracy (in %): 92.66202167662021
Validation Accuracy (in %): 93.03405572755418
Test Accuracy (in %): 89.1395708913957

The plots for the training, validation and test set accuracies against the number of nodes in the tree are presented next page onwards.

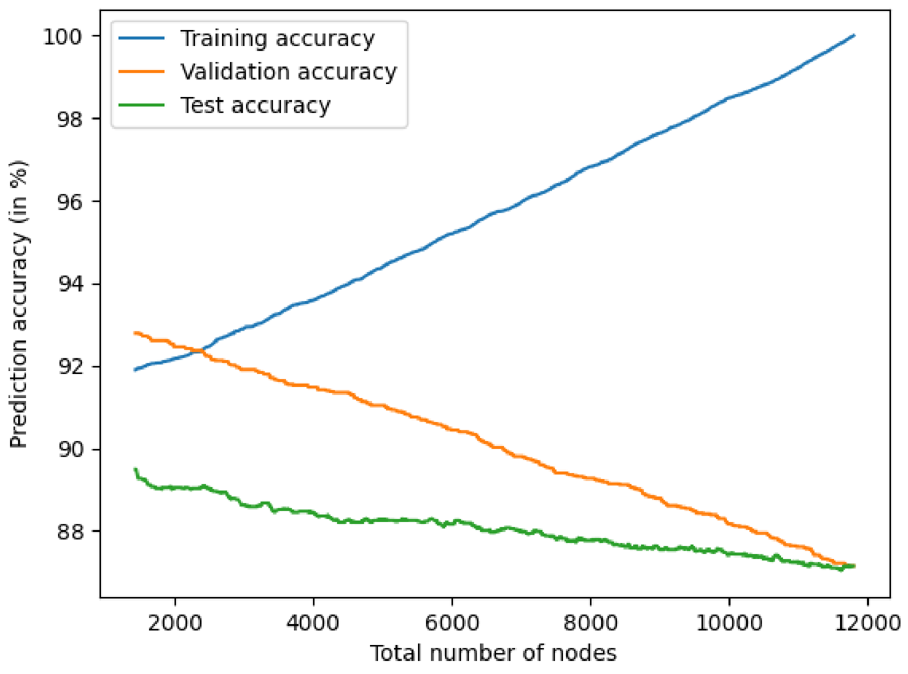


Figure 1.5: Prediction accuracy vs total number of nodes (without one-hot encoding)

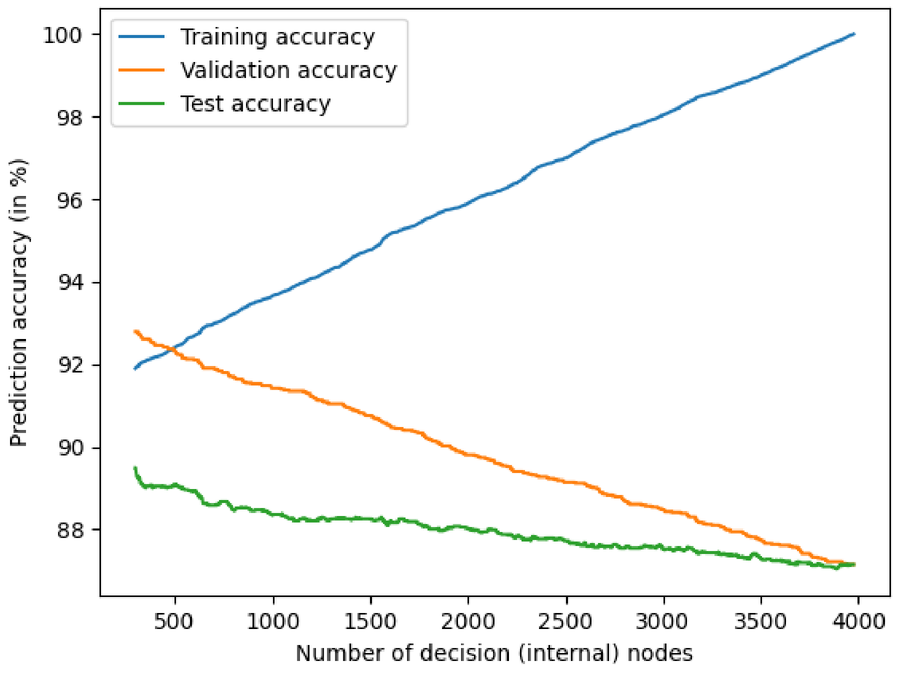


Figure 1.6: Prediction accuracy vs number of decision nodes (without one-hot encoding)

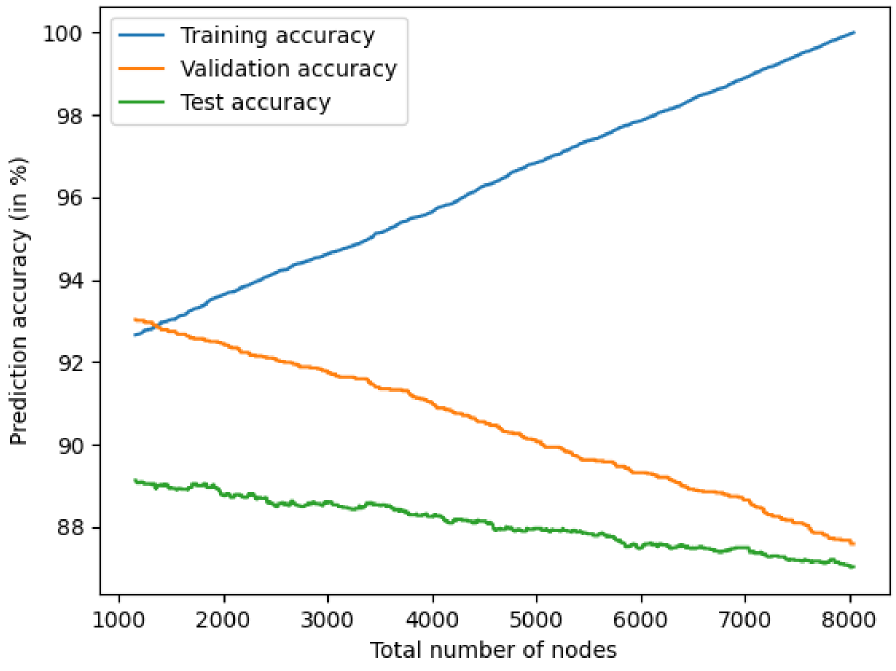


Figure 1.7: Prediction accuracy vs total number of nodes (with one-hot encoding)

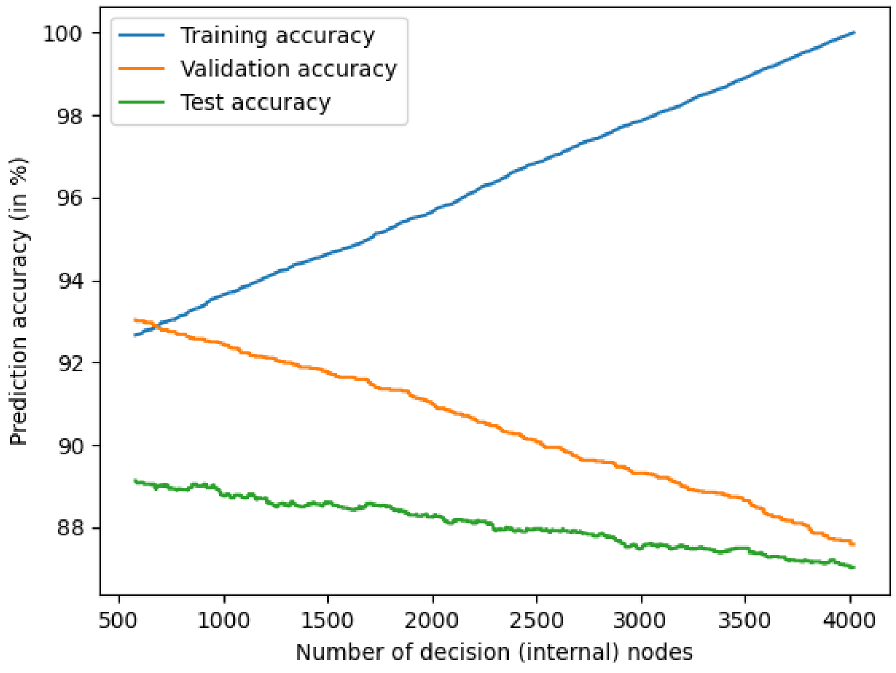


Figure 1.8: Prediction accuracy vs number of decision nodes (with one-hot encoding)

Note: The accuracy values were obtained exact node-wise for plotting figures: *Figure 1.5* to *1.8*

Comments:

- First thing I observe is that the training accuracy decreases monotonically for both the trees, as we prune more number of nodes. This can be explained as for each node that we prune, we get rid of some *pure* nodes, and thereby, introduce some approximations in making a prediction. This approximation leads to some prediction errors, which were not there to begin with, and hence, lead to a uniform decline in the training accuracy. Hence, the decrease in training accuracy is justified.
- Secondly, I observe that the validation accuracy increases monotonically with decrease in number of nodes in both trees. This is expected as I only prune a node (and sub-tree) below it if it does not harm the validation accuracy. Hence, the monotonic increase in validation accuracy is also justified.
- Thirdly, I observe that the test accuracy is also increasing in both the trees in the long run (successive pruning). Also, the final values of validation and test set accuracies are better than in *section 1.1* for both the trees. Moreover, the final training accuracy is now around 92% (in both cases), which is closer to final values of validation and test set accuracies. Hence, the pruned tree is a better representative of how well the model is generalizing to unseen data than the fully grown tree, where the accuracy values were in great disagreement.
- In total, I can conclude that post-pruning the fully grown tree has made the model un-learn training noise that was specific to the data and reduce over-fitting. At the same time, it has improved the accuracy over unseen data (test set) (2% increase) and hence, become a better fit/model.
- Lastly, I comment on the actual values obtained. The final validation accuracy is better when I am using one-hot encoding than when I am not. On the other hand, the final test accuracy is better when I am using the normal encoding. Also, the final number of nodes is almost the same in both the trees (around 1000), even though they did not start with the same count. This indicates that irrelevant (or noiseful) tests (decision nodes) were removed from both the trees and only critical tests were kept, leading to almost similar number of nodes in the final pruned-tree. I also stress on the fact that even though the memory requirement of pruned-tree is much lower than the fully grown tree, it is still a better estimator, due to reasons specified above. Hence, it is a *win-win*.

1.3 Random Forests

In this section, I trained random forests using RandomForestClassifier of scikit-learn library (Python). I also performed grid search over hyper-parameters space using GridSearchCV (scikit-learn) and found the optimal values of the parameters. The parameter space is defined by the following parameter grid:

```
param_grid = {
     'n_estimators': [50, 150, 250, 350, 450],
     'max_features': [0.1, 0.3, 0.5, 0.7, 0.9],
     'min_samples_split': [2, 4, 6, 8, 10]
}
```

Out-of-Bag accuracy was used as the scoring method for grid search, as specified in the assignment. Below, I mention the results that I obtained:
Note: One-hot encoding was used as scikit-learn does not handle categorical data
Note: GridSearchCV uses 5-fold cross-validation by default for determining best parameters

Best n_estimators: 150
Best max_features: 0.3
Best min_samples_split: 10

Accuracies obtained using *best* parameters:
Out-of-Bag accuracy (in %): 90.7266091572661
Training Accuracy (in %): 97.95399247953992
Validation Accuracy (in %): 90.71207430340557
Test Accuracy (in %): 89.98009289980094

Comparison: Comparing this model with model obtained after pruning in *section 1.2*

- The validation accuracy is higher for post-pruned tree ($93.034 > 90.71$). This is expected, as in the pruning-algorithm, I was only looking at the validation accuracy, when deciding whether or not I should prune a node. Hence, it is reasonable to assume that pruning method fits to some noise present even in the validation set, thereby, giving a larger accuracy value. This possibility of overfitting is explained further in the point below.
- The test accuracy is slightly better for random forest model ($89.98 > 89.14$) Also, the training accuracy is much better for random forest model ($97.95 > 92.66$). This increase can again be explained by observing how the pruning algorithm works. The pruning algorithm prunes a node if it does not harm the validation accuracy. In doing so, the algorithm might prune of some node, which represents a generalized pattern, but which got pruned just because it was not represented in statistically significant amount in the validation set. Hence, there is a possibility that we overfit the validation set in our attempt of reducing overfitting w.r.t training set. Hence, final training and test set accuracies are lower in the post-pruned tree.
- Random Forest on the other hand generates multiple decision trees using bootstrapped data and use averaging to improve accuracy and control-overfitting. Hence, it is less likely that the forest will collectively overfit any single set, thus, giving better test set accuracies.

1.4 Random Forests - Parameter Sensitivity Analysis

In this section, I use the same parameter grid as in *section 1.3*, and vary one parameter at a time (keeping other parameters equal to best parameter). I then plot the *Out-of-Bag*, validation and test accuracies, and analyse parameter sensitivity. Below, I present my plots.
Note: One-hot encoding was used as scikit-learn does not handle categorical data

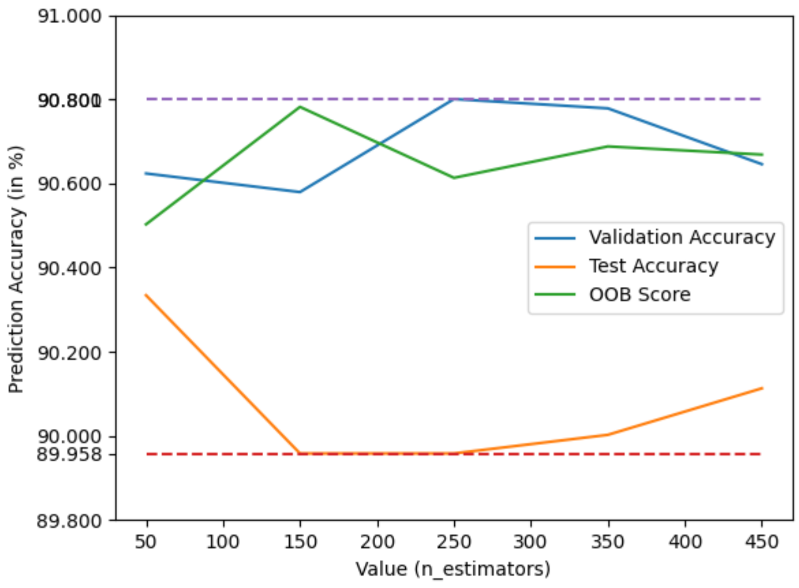


Figure 1.9: Prediction accuracy vs n_estimators

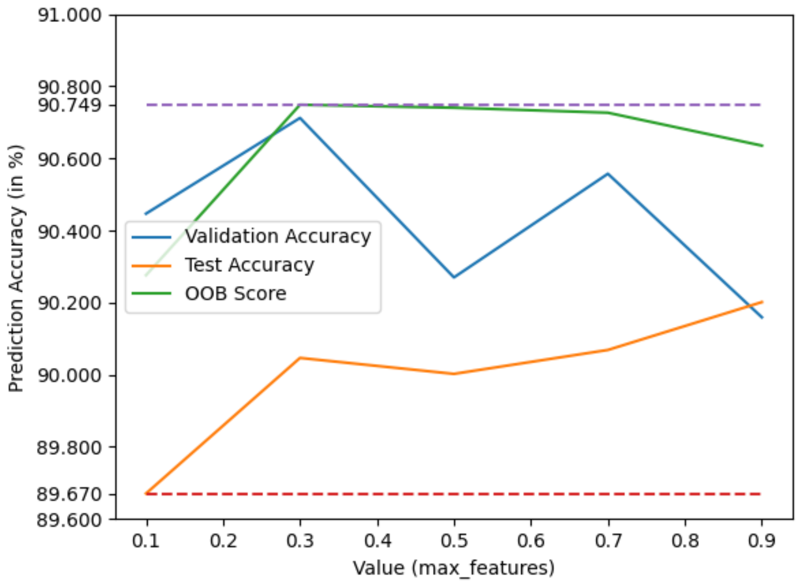


Figure 1.10: Prediction accuracy vs max_features

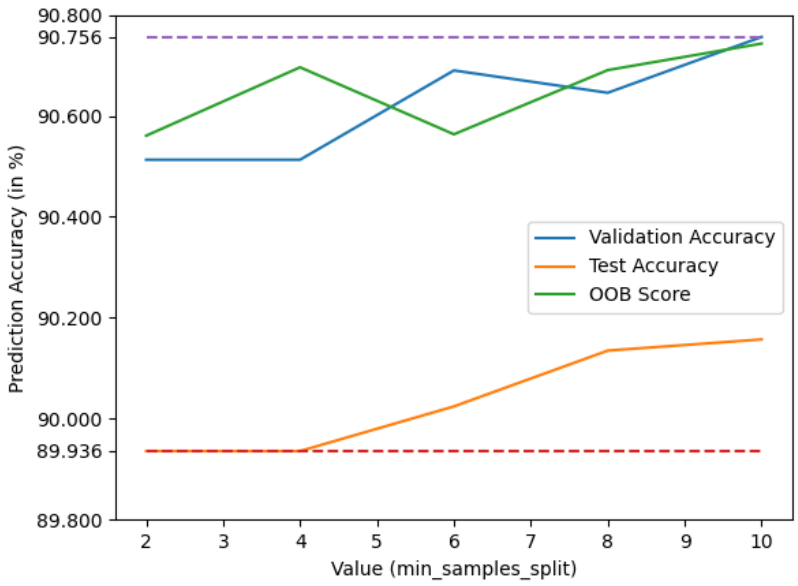


Figure 1.11: Prediction accuracy vs min_samples_split

Analysis:

- For each parameter (n_estimators, max_features, min_samples_split), the variation in validation and test set accuracy is not that great, when varying one of them (keeping others fixed). This can also be directly observed from the graph: 0.3% (val), 0.4% (test) [n_estimators]; 0.5% (val), 0.5% (test) [max_features]; 0.2% (val), 0.3% (test) [min_samples_split].
- Therefore, the model is stable w.r.t values of these three parameters and the accuracies are not affected by much. Still out of the three, maximum difference in accuracy values is seen in *Figure 1.10* (max_features). Hence, in a way, we can say that max_features is the most sensitive parameter out of the three. But, still this may not be true in the strict sense, as the variation is very small. Also, such a variation may not even exist for different random forests, as they are all random.
- Lastly, observe that the *Out-of-Bag* accuracy is the highest for the optimal set of parameters in each plot. This is expected as the parameters were selected based on OOB score.

Chapter 2

Neural Networks

Some points:

- Both input (X) and output (Y) is one-hot encoded in all the parts below (2.1-2.4)
- Θ parameters are used for each layer of NN. These consist of both weights (W) and bias (b) parameters
- In all parts except *section 2.3 (sigmoid)*, Θ parameters are initialized using `np.random.normal` multiplied with $\sqrt{\frac{2}{layers[i]+layers[i-1]}}$
- In *section 2.3 (sigmoid)*, Θ parameters are initialized by just using `np.random.normal`

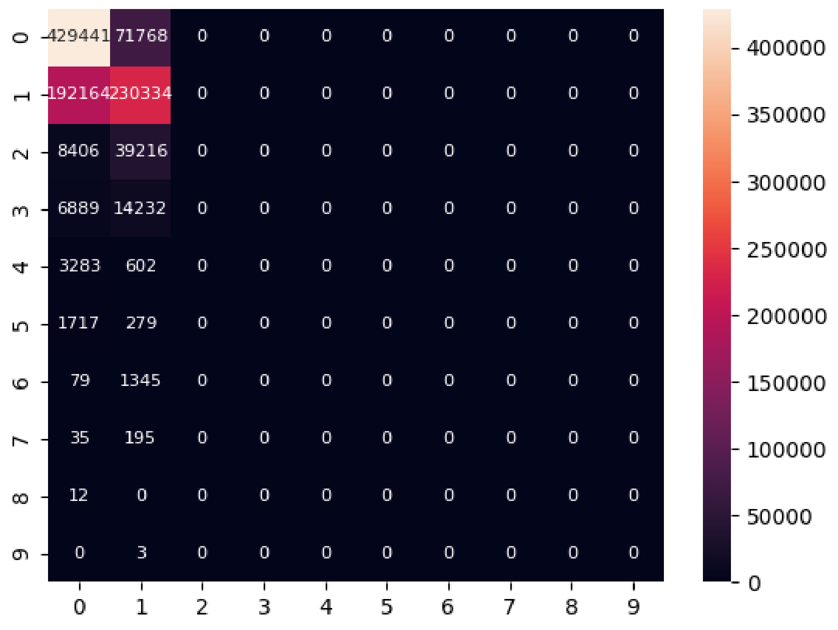
2.1 Single Hidden Layer - Constant η

I used the generic neural network implementation to train five neural networks with one hidden layer (5, 10, 15, 20 and 25 units respectively), using sigmoid activation function for hidden layer and constant learning rate. Below, I specify the NN architecture more concisely:

- Number of features: 85
- Number of Classes: 10
- Hidden Layers: [5] or [10] or [15] or [20] or [25]
- Loss Function: Mean Squared Error (MSE)
- Solver: Stochastic Gradient Descent
- Mini-batch size: 100
- Base Learning Rate: 0.1
- Learning Rate: Constant
- Maximum Iterations: 6000
- MSE difference threshold: 10^{-6}
- Stopping Criteria: Maximum Iterations or difference in average mini-batch MSE (training set) over an epoch is less than 10^{-6} for 10 consecutive epochs

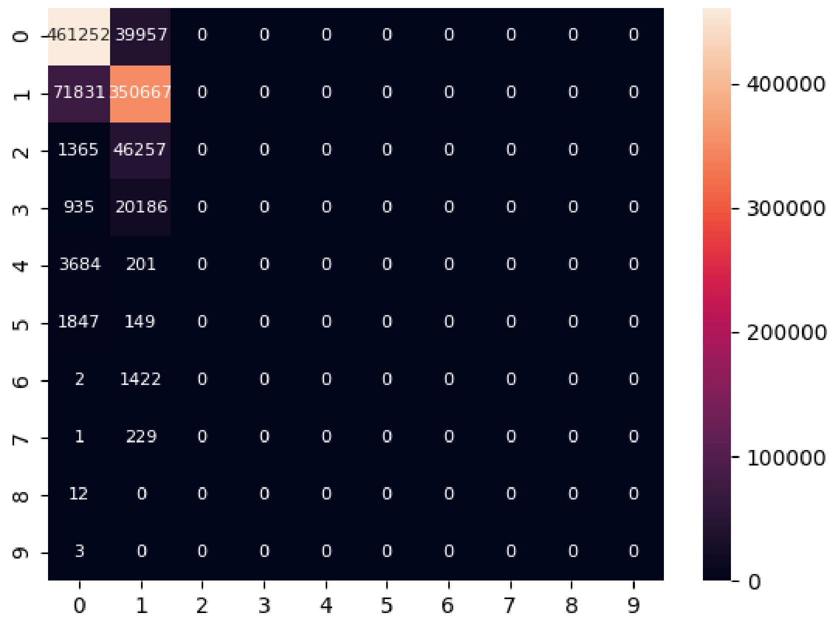
Using the above NN architecture, I trained five networks: each using 5, 10, 15, 20 and 25 units in the hidden layer respectively. Below, I report my observations:

5 hidden units
Training time (in s): 320.8039879798889
Training Accuracy (in %): 67.44102359056377
Test Accuracy (in %): 65.9775



	0	1	2	3	4	5	6	7	8	9
0	429441	71768	0	0	0	0	0	0	0	0
1	192164	230334	0	0	0	0	0	0	0	0
2	8406	39216	0	0	0	0	0	0	0	0
3	6889	14232	0	0	0	0	0	0	0	0
4	3283	602	0	0	0	0	0	0	0	0
5	1717	279	0	0	0	0	0	0	0	0
6	79	1345	0	0	0	0	0	0	0	0
7	35	195	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	0	3	0	0	0	0	0	0	0	0

Figure 2.1: Confusion Matrix (actual vs predicted) (5 hidden units)



	0	1	2	3	4	5	6	7	8	9
0	461252	39957	0	0	0	0	0	0	0	0
1	71831	350667	0	0	0	0	0	0	0	0
2	1365	46257	0	0	0	0	0	0	0	0
3	935	20186	0	0	0	0	0	0	0	0
4	3684	201	0	0	0	0	0	0	0	0
5	1847	149	0	0	0	0	0	0	0	0
6	2	1422	0	0	0	0	0	0	0	0
7	1	229	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Figure 2.2: Confusion Matrix (actual vs predicted) (10 hidden units)

10 hidden units
Training time (in s): 348.49889373779297
Training Accuracy (in %): 83.50659736105558
Test Accuracy (in %): 81.19189999999999

15 hidden units
Training time (in s): 360.0292429924011
Training Accuracy (in %): 91.93922431027589
Test Accuracy (in %): 91.2873

20 hidden units
Training time (in s): 406.30319809913635
Training Accuracy (in %): 92.33106757297081
Test Accuracy (in %): 92.19659999999999

25 hidden units
Training time (in s): 463.5871419906616
Training Accuracy (in %): 92.31507397041183
Test Accuracy (in %): 92.00500000000001

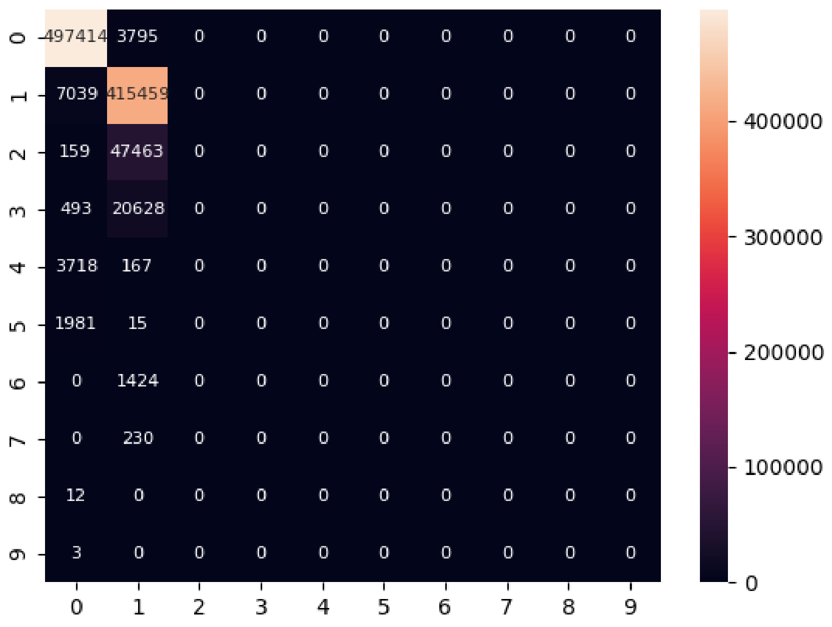
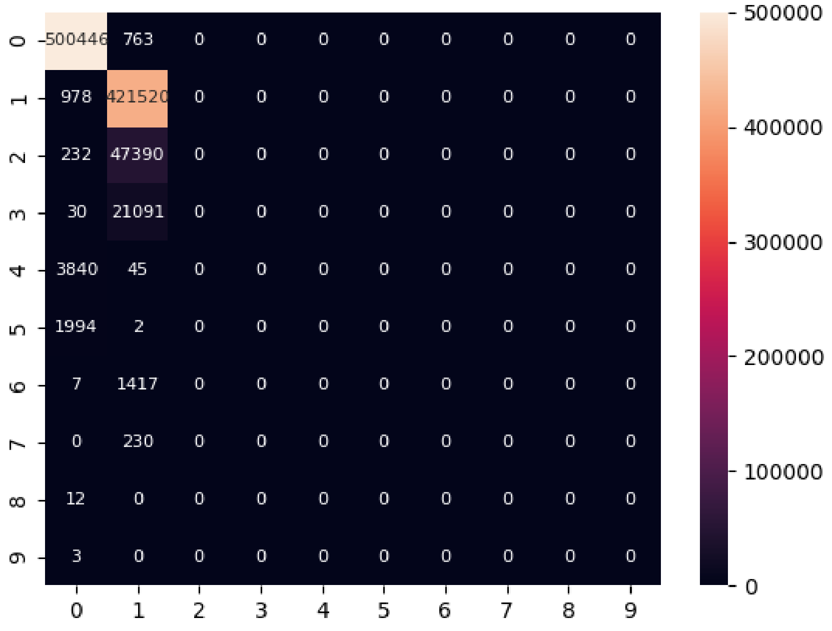
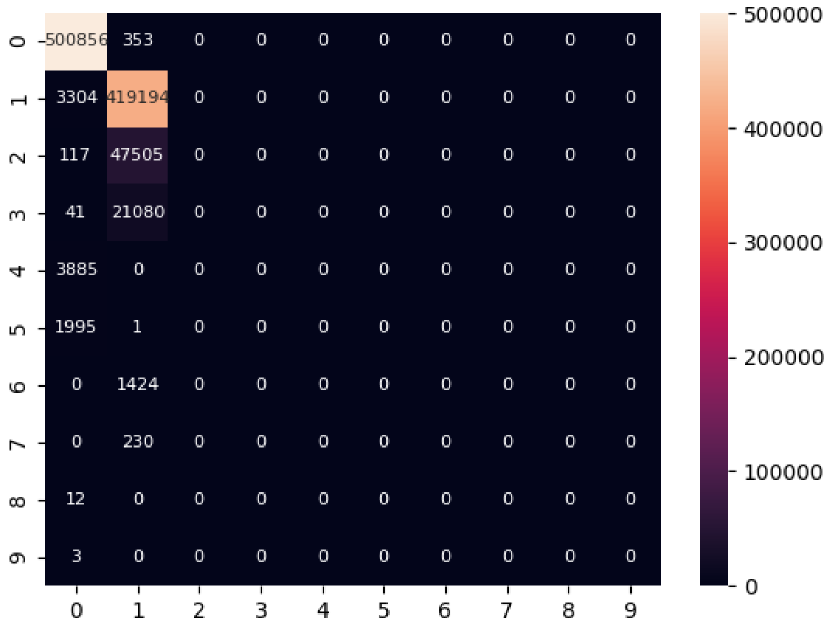


Figure 2.3: Confusion Matrix (actual vs predicted) (15 hidden units)



	0	1	2	3	4	5	6	7	8	9
0	500446	763	0	0	0	0	0	0	0	0
1	978	421520	0	0	0	0	0	0	0	0
2	232	47390	0	0	0	0	0	0	0	0
3	30	21091	0	0	0	0	0	0	0	0
4	3840	45	0	0	0	0	0	0	0	0
5	1994	2	0	0	0	0	0	0	0	0
6	7	1417	0	0	0	0	0	0	0	0
7	0	230	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Figure 2.4: Confusion Matrix (actual vs predicted) (20 hidden units)



	0	1	2	3	4	5	6	7	8	9
0	500856	353	0	0	0	0	0	0	0	0
1	3304	419194	0	0	0	0	0	0	0	0
2	117	47505	0	0	0	0	0	0	0	0
3	41	21080	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1995	1	0	0	0	0	0	0	0	0
6	0	1424	0	0	0	0	0	0	0	0
7	0	230	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Figure 2.5: Confusion Matrix (actual vs predicted) (25 hidden units)

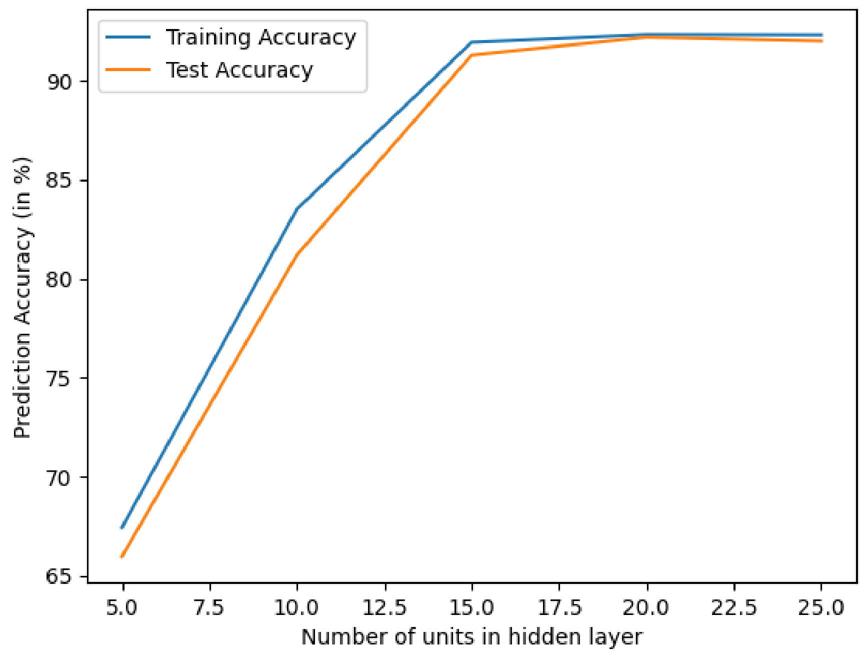


Figure 2.6: Prediction accuracy vs number of hidden units

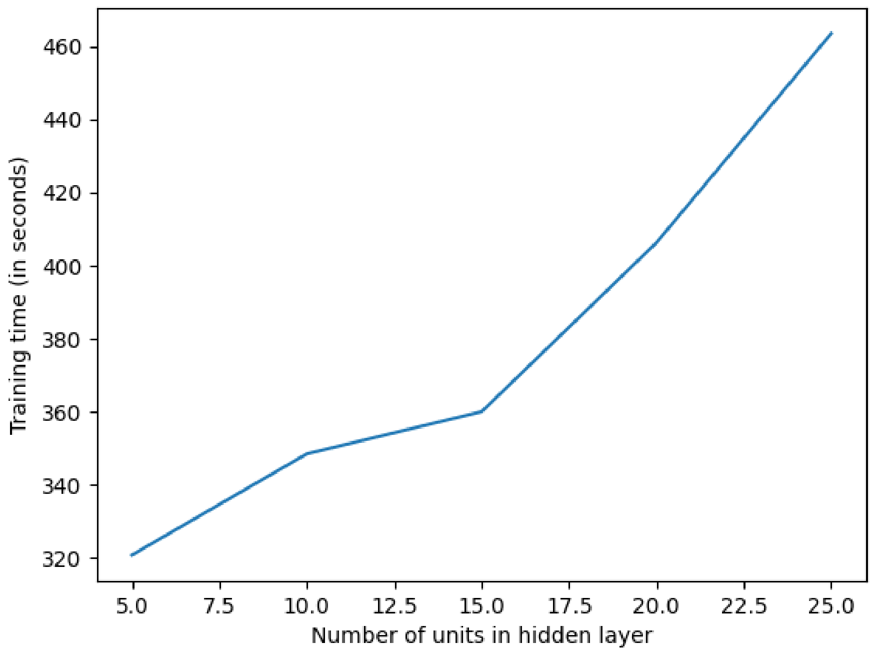


Figure 2.7: Training time vs number of hidden units

Comments:

- From *Figure 2.7*, it is clear that the training time of NN increases monotonically with increase in number of hidden units. This is explainable as more units mean more parameters per training example. Hence, determining and doing a single update of SGD will be more time consuming as we increase the number of hidden units.
- From *Figure 2.6*, it is clear that both training and test set accuracy increase with increase in number of hidden units (till 20). From 20 to 25 units, there is a minor decrease in both training and test set accuracy (0.1%). This is explainable, as more units mean more representational power. Hence, we are able to better classify the data. Also, the increase in accuracy is significant from 5 to 10 and from 10 to 15. 15 units onwards the increase in accuracy is not much (no significant improvement in learning). This is possibly a constraint of just using a single hidden layer (and possibly using MSE loss function). Thus, we are unable to represent the actual hypothesis function accurately by just using a single hidden layer.
- The confusion matrix for each case is very similar in the sense that the predictions lie entirely in the first two columns (class 0 and 1). Hence, the model is unable to effectively learn the features corresponding to other classes, possibly because the data is heavily skewed towards the first two classes, and the fact that a single hidden layer NN is not powerful enough to represent the multi-class classification problem at hand. Other thing, I observe is that the sum of diagonal entries increases as we increase the number of hidden units (till 20). From 20 to 25, there is a minor decrease in the sum of diagonal entries (difference = 1916). This observation is equivalent to the observation regarding prediction accuracy.

2.2 Single Hidden Layer - Adaptive η

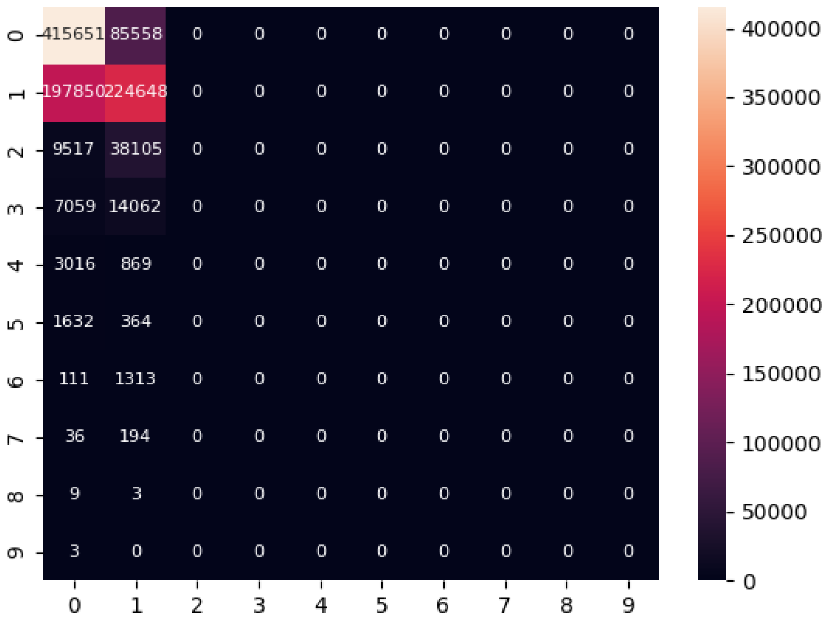
I used the generic neural network implementation to train five neural networks with one hidden layer (5, 10, 15, 20 and 25 units respectively), using sigmoid activation function for hidden layer and adaptive learning rate. Below, I specify the NN architecture more concisely:

- Number of features: 85
- Number of Classes: 10
- Hidden Layers: [5] or [10] or [15] or [20] or [25]
- Loss Function: Mean Squared Error (MSE)
- Solver: Stochastic Gradient Descent
- Mini-batch size: 100
- Base Learning Rate: 3
- Learning Rate: Adaptive ($\frac{3}{\sqrt{epoch}}$)
- Maximum Iterations: 6000
- MSE difference threshold: 10^{-6}
- Stopping Criteria: Maximum Iterations or difference in average mini-batch MSE (training set) over an epoch is less than 10^{-6} for 10 consecutive epochs

Using the above NN architecture, I trained five networks: each using 5, 10, 15, 20 and 25 units in the hidden layer respectively. Below, I report my observations:

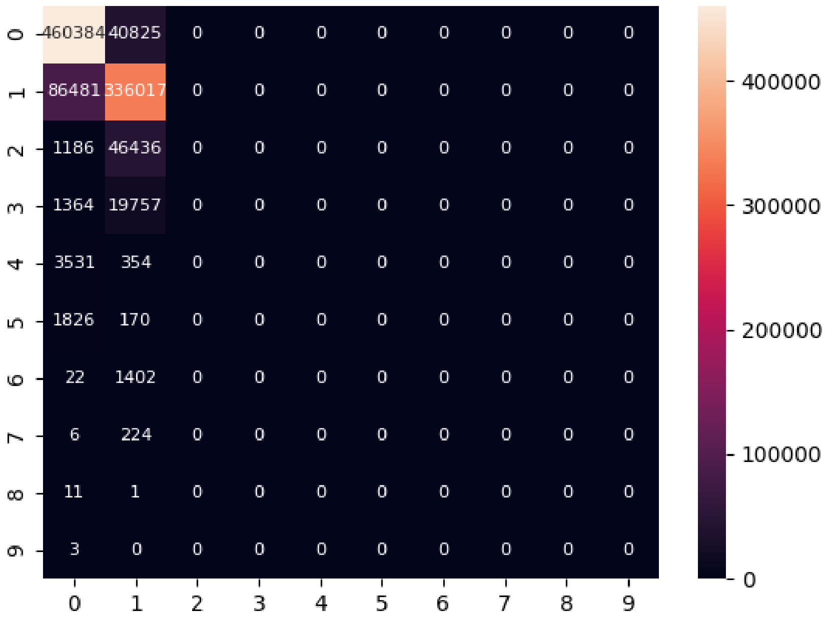
5 hidden units
Training time (in s): 169.59088730812073
Training Accuracy (in %): 65.38984406237505
Test Accuracy (in %): 64.0299

10 hidden units
Training time (in s): 349.59820580482483
Training Accuracy (in %): 81.98720511795283
Test Accuracy (in %): 79.6401



	0	1	2	3	4	5	6	7	8	9
0	415651	85558	0	0	0	0	0	0	0	0
1	197850	224648	0	0	0	0	0	0	0	0
2	9517	38105	0	0	0	0	0	0	0	0
3	7059	14062	0	0	0	0	0	0	0	0
4	3016	869	0	0	0	0	0	0	0	0
5	1632	364	0	0	0	0	0	0	0	0
6	111	1313	0	0	0	0	0	0	0	0
7	36	194	0	0	0	0	0	0	0	0
8	9	3	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Figure 2.8: Confusion Matrix (actual vs predicted) (5 hidden units)



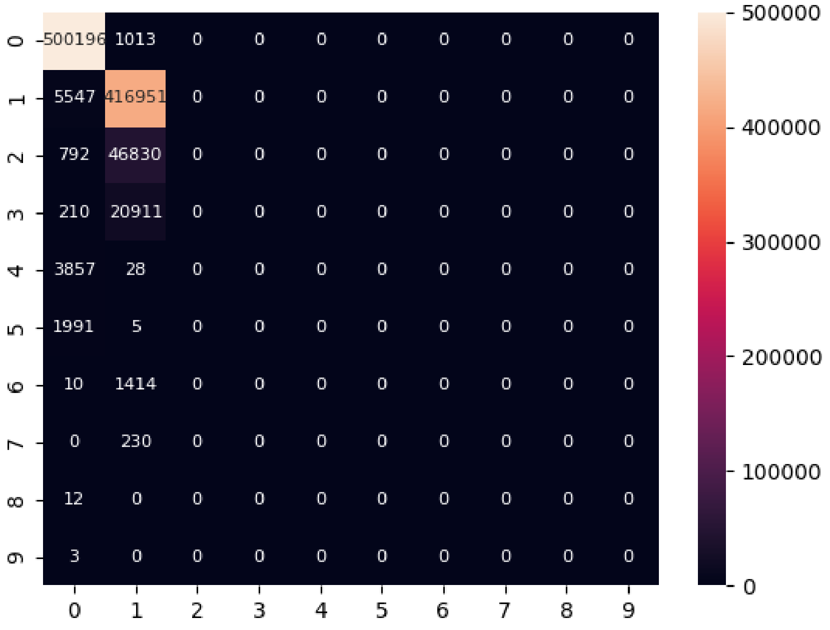
	0	1	2	3	4	5	6	7	8	9
0	460384	40825	0	0	0	0	0	0	0	0
1	86481	336017	0	0	0	0	0	0	0	0
2	1186	46436	0	0	0	0	0	0	0	0
3	1364	19757	0	0	0	0	0	0	0	0
4	3531	354	0	0	0	0	0	0	0	0
5	1826	170	0	0	0	0	0	0	0	0
6	22	1402	0	0	0	0	0	0	0	0
7	6	224	0	0	0	0	0	0	0	0
8	11	1	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Figure 2.9: Confusion Matrix (actual vs predicted) (10 hidden units)

15 hidden units
Training time (in s): 361.51237297058105
Training Accuracy (in %): 91.9312275089964
Test Accuracy (in %): 91.71470000000001

20 hidden units
Training time (in s): 414.5130271911621
Training Accuracy (in %): 92.28708516593362
Test Accuracy (in %): 91.9735

25 hidden units
Training time (in s): 430.644388198852546
Training Accuracy (in %): 92.29908036785287
Test Accuracy (in %): 92.038



	0	1	2	3	4	5	6	7	8	9
0	500196	1013	0	0	0	0	0	0	0	0
1	5547	416951	0	0	0	0	0	0	0	0
2	792	46830	0	0	0	0	0	0	0	0
3	210	20911	0	0	0	0	0	0	0	0
4	3857	28	0	0	0	0	0	0	0	0
5	1991	5	0	0	0	0	0	0	0	0
6	10	1414	0	0	0	0	0	0	0	0
7	0	230	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Figure 2.10: Confusion Matrix (actual vs predicted) (15 hidden units)

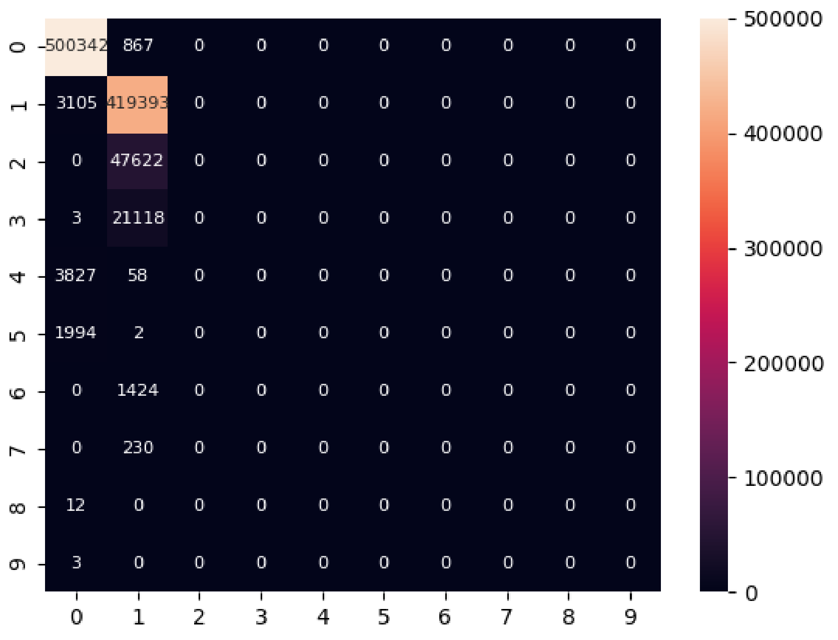


Figure 2.11: Confusion Matrix (actual vs predicted) (20 hidden units)

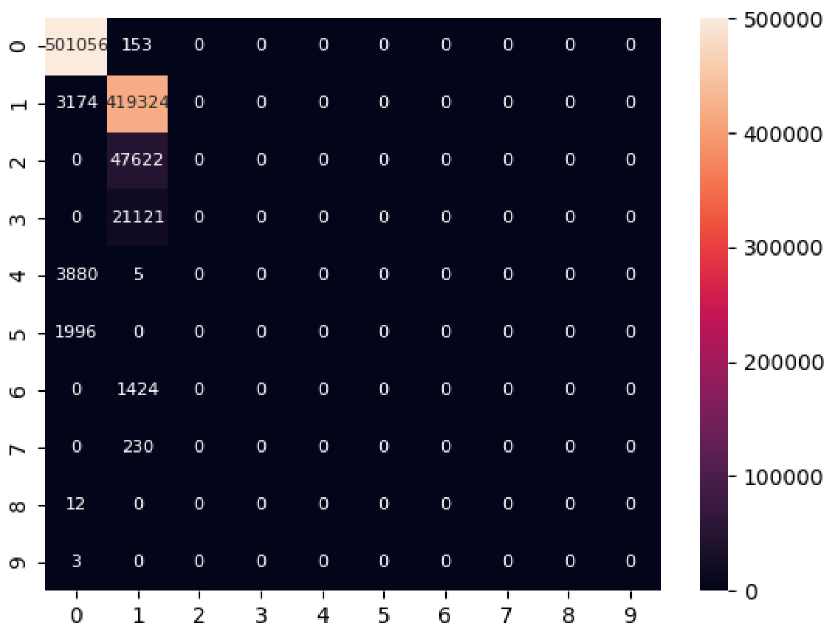


Figure 2.12: Confusion Matrix (actual vs predicted) (25 hidden units)

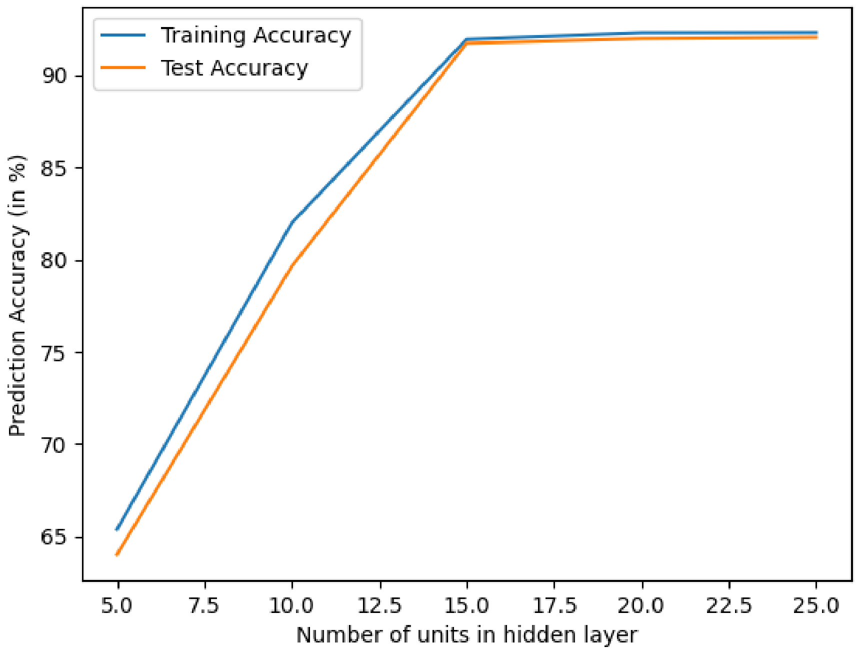


Figure 2.13: Prediction accuracy vs number of hidden units

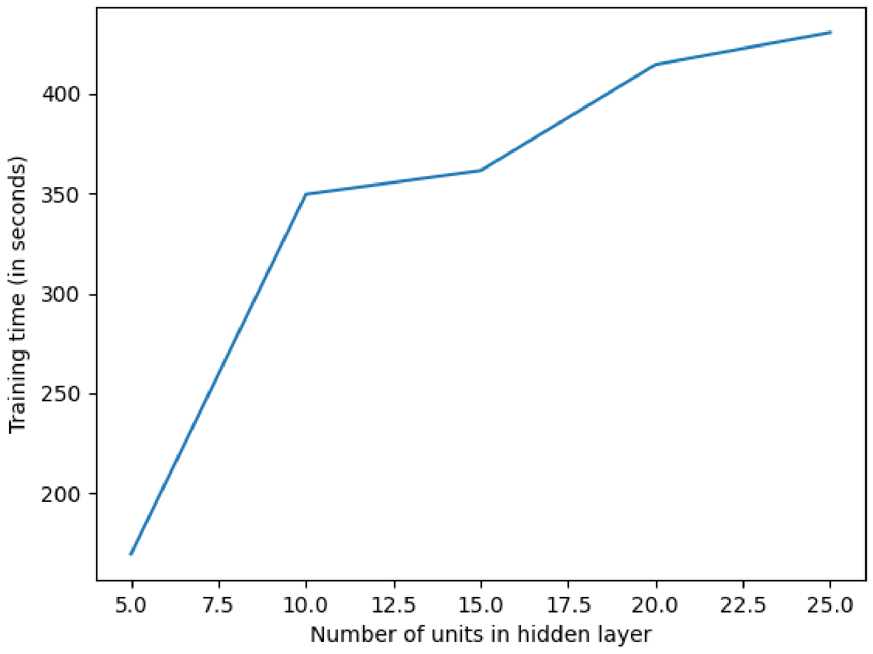


Figure 2.14: Training time vs number of hidden units

Comments:

- From *Figure 2.14*, it is clear that the training time of NN increases monotonically with increase in number of hidden units. This is explainable as more units mean more parameters per training example. Hence, determining and doing a single update of SGD will be more time consuming as we increase the number of hidden units.
- From *Figure 2.13*, it is clear that both training and test set accuracy increase with increase in number of hidden units. This is explainable, as more units mean more representational power. Hence, we are able to better classify the data. Also, the increase in accuracy is significant from 5 to 10 and from 10 to 15. 15 units onwards the increase in accuracy is not much (1%, no significant improvement in learning). This is possibly a constraint of just using a single hidden layer (and possibly using MSE loss function). Thus, we are unable to represent the actual hypothesis function accurately by just using a single hidden layer.
- The confusion matrix for each case is very similar in the sense that the predictions lie entirely in the first two columns (class 0 and 1). Hence, the model is unable to effectively learn the features corresponding to other classes, possibly because the data is heavily skewed towards the first two classes, and the fact that a single hidden layer NN is not powerful enough to represent the multi-class classification problem at hand. Other thing, I observe is that the sum of diagonal entries increases as we increase the number of hidden units. This is in fact equivalent to increase in accuracy as noted in point above.

Comparison:

- The accuracy values obtained in *section 2.1* and *section 2.2* are very similar and there is no significant difference when comparing the models using same number of hidden layer units. This is expected, as the only change in NN is that of the learning rate, and the learning rate has no effect whatsoever on the optimal values of Θ parameters and loss function. Hence, the final accuracy values do not change. (the minor difference is possibly existing because of the fact that parameters are initialised randomly, every time the program is run)
- The confusion matrices obtained are also similar when looking at individual entries (with minor difference due to randomness)
- The training time is similar when using 10 and 15 hidden units. When using 5 units or 25 units, the training time when using adaptive learning rate is lesser than when using constant learning rate (2 min and 30 seconds respectively). The training time when using 20 hidden units is 8 seconds more than before. In total adaptive learning rate is leading to faster training. The slight variations are possibly due to randomness in learning parameters. This is explainable, as using adaptive learning rate allows us to pick higher seed learning rate (3), which enables faster learning without overshooting the minima (as the rate is suitably reduced with epochs).

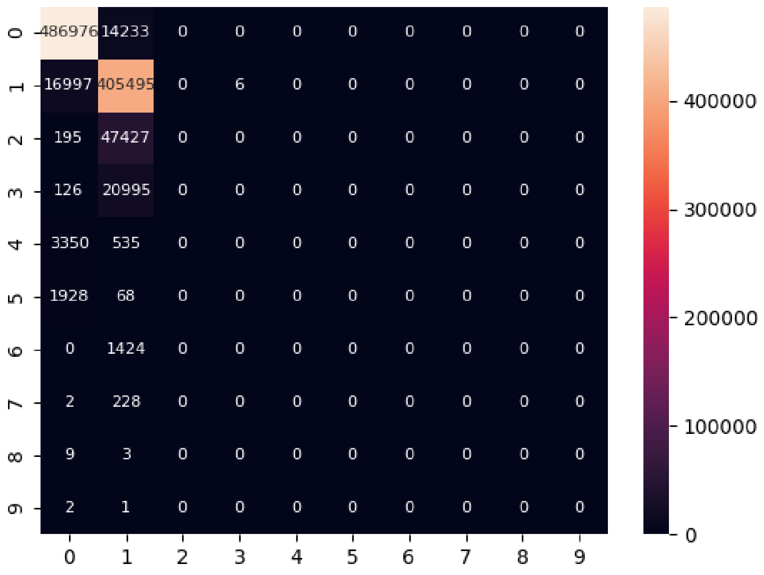
2.3 Two Hidden Layers - Adaptive η

I used the generic neural network implementation to train a neural network with two hidden layers (100 units each), using both sigmoid and ReLU activation function for hidden layers. Below, I specify the NN architecture more concisely:

- Number of features: 85
- Number of Classes: 10
- Hidden Layers: [100, 100]
- Loss Function: Mean Squared Error (MSE)
- Solver: Stochastic Gradient Descent
- Mini-batch size: 100
- Base Learning Rate: 3
- Learning Rate: Adaptive ($\frac{3}{\sqrt{epoch}}$)
- Maximum Iterations: 3000
- MSE difference threshold: 10^{-6}
- Stopping Criteria: Maximum Iterations or difference in average mini-batch MSE (training set) over an epoch is less than 10^{-6} for 10 consecutive epochs

Using the above NN architecture, I trained two networks: one using ReLU as the activation function for hidden layer units, and other using sigmoid as the activation function for hidden layer units. Below, I report my observations:

Sigmoid
Training time (in s): 859.7374517917633
Training Accuracy (in %): 92.32706917233106
Test Accuracy (in %): 89.2471



	0	1	2	3	4	5	6	7	8	9
0	486976	14233	0	0	0	0	0	0	0	0
1	16997	405495	0	6	0	0	0	0	0	0
2	195	47427	0	0	0	0	0	0	0	0
3	126	20995	0	0	0	0	0	0	0	0
4	3350	535	0	0	0	0	0	0	0	0
5	1928	68	0	0	0	0	0	0	0	0
6	0	1424	0	0	0	0	0	0	0	0
7	2	228	0	0	0	0	0	0	0	0
8	9	3	0	0	0	0	0	0	0	0
9	2	1	0	0	0	0	0	0	0	0

Figure 2.15: Confusion Matrix (actual vs predicted) (Sigmoid)

ReLU
Training time (in s): 52.95625400543213
Training Accuracy (in %): 92.33106757297081
Test Accuracy (in %): 92.3695

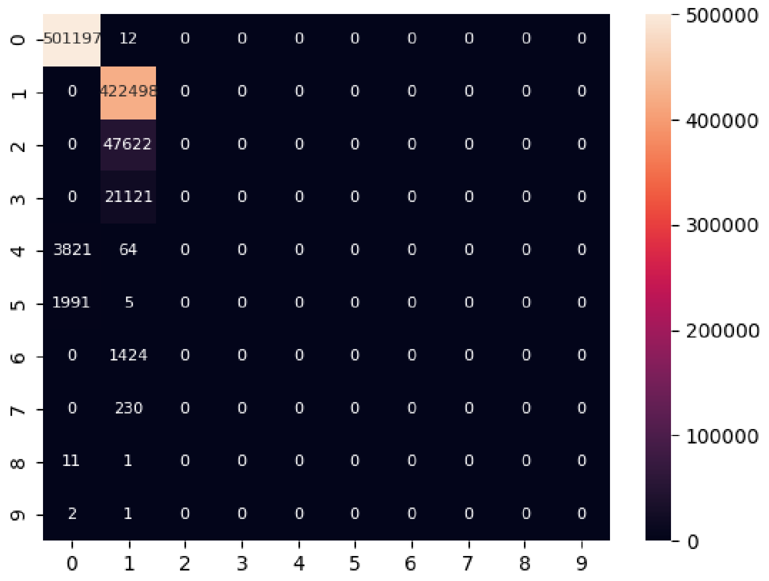


Figure 2.16: Confusion Matrix (actual vs predicted) (ReLU)

Relative Comparison:

- The training accuracy of NN using ReLU is slightly better than the one using sigmoid in the hidden layers ($92.33 > 92.32$). The test accuracy of NN using ReLU is however, much better than the one using sigmoid (3% difference, $92.36 > 89.24$). Still both models are performing poorly when it comes to making predictions of examples with classes other than 0 and 1. This is also evident from the confusion matrices, where the last 8 columns are empty for both NNs.
- The training time of NN using ReLU is much much less than the training time of NN using sigmoid in hidden layer units ($52.95 < 859.73$). Hence, when training a 2-layer NN, it is a better choice to use ReLU in hidden layers, as it leads to more accuracy (point above) and significantly less training time. (This slow training in case of sigmoid can be attributed to the fact that if 'x' is large (> 4) or small (< -4), then the gradient of sigmoid(x) becomes very small and close to *zero*. This leads to small updates and slow convergence)

Comparison:

- There is a slight increase in accuracy over single hidden layer model (20/25 units) trained in *section 2.1*, however, the difference is not significant.
- Also, the confusion matrices obtained are similar in the sense that predictions lie entirely in the first two columns, thereby, having a low *recall* for other classes and low *precision* for

classes 0 and 1.

- This lack of significant increase in accuracy/F1-score is possibly because of the limitations of MSE loss function when it comes to multi-class classification problem at hand (heavily skewed data). Hence, a single hidden layer NN with 25 units is performing almost equally well as compared to two-hidden layer NN with 100 units each.

2.4 MLPClassifier - Cross-Entropy Loss

In this section, I used MLPClassifier (scikit-learn) to train a neural network with two hidden layers (100 units each), using both sigmoid and ReLU activation function for hidden layers. Below, I specify the NN architecture more concisely:

- Number of features: 85
- Number of Classes: 10
- Hidden Layers: [100, 100]
- Loss Function: Cross-Entropy Loss
- Solver: Stochastic Gradient Descent
- Mini-batch size: 100
- Base Learning Rate: 0.1
- Learning Rate: Adaptive (sklearn 'Adaptive' is different from inverse square root)
- Maximum Iterations: 1000

Note: All other attributes/parameters were set to default.

Using the above NN architecture, I trained two networks: one using ReLU as the activation function for hidden layer units, and other using sigmoid as the activation function for hidden layer units. Below, I report my observations:

Sigmoid
Training Accuracy (in %): 99.9360255897641
Test Accuracy (in %): 98.33070000000001

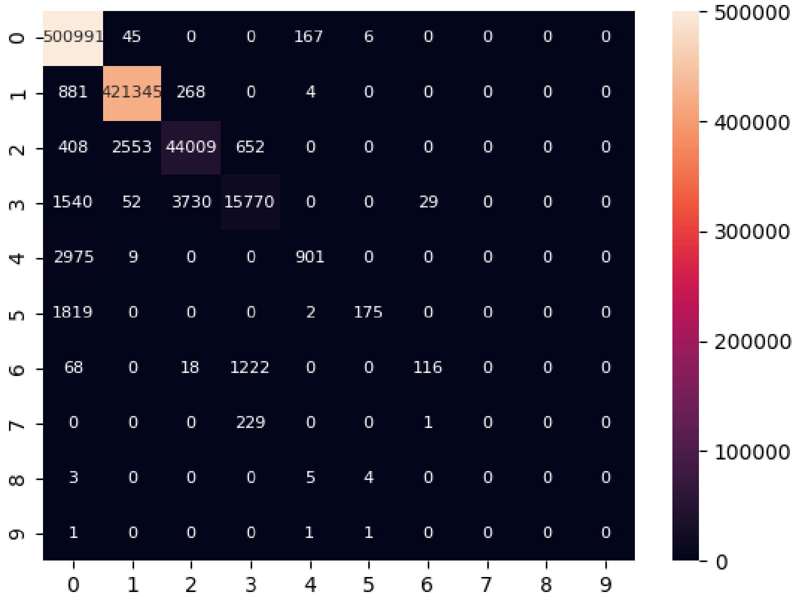


Figure 2.17: Confusion Matrix (actual vs predicted) (Sigmoid)

ReLU
Training Accuracy (in %): 100.0
Test Accuracy (in %): 99.2876

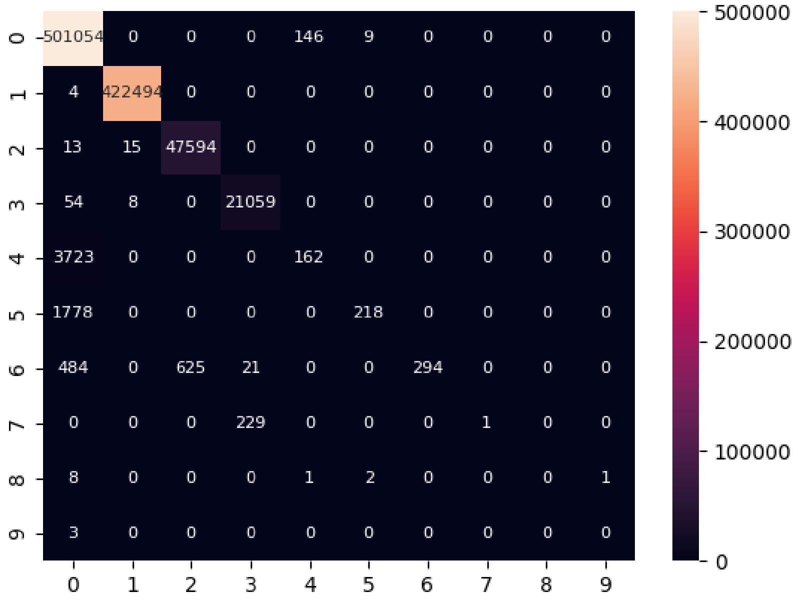


Figure 2.18: Confusion Matrix (actual vs predicted) (ReLU)

Comparison:

- Comparing NNs using ReLU as the activation function, I observe that the training and test set accuracies obtained using MLPClassifier are much much greater than the ones obtained in *section 2.3* (100 > 92.33 and 99.28 > 92.36), even though both of them are using the same NN architecture. This is possibly because of the different loss functions. In *section 2.3*, the NN uses MSE loss to optimize its set of parameters, wherease MLPClassifier uses the cross-entropy loss, which is, at least for classification purposes, a much better loss function. Since, the problem at hand is a classification problem, it makes sense that MLPClassifier will have a better performance than my NN, when it comes to making accurate predictions.
- The confusion matrix obtained in this case is also a lot better, especially when it comes to making predictions for classes other than 0 and 1. It can be seen that confusion matrix (*Figure 2.18*) has a better *recall* when it comes to other classes (2-9), and a better *precision* when it comes to making predictions of classes 0 and 1. Hence, the MLPClassifier's NN has a much better macro-F1 score along with having a much better accuracy. Hence, it is a better model.
- Comparing NNs using sigmoid as the activation function, I observe that the training and test set accuracies obtained using MLPClassifier are much much greater than the ones obtained in *section 2.3* (99.93 > 92.32 and 98.33 > 89.25), even though both of them are using the same NN architecture. This is possibly because of the different loss functions. In *section 2.3*,

the NN uses MSE loss to optimize its set of parameters, wherease MLPClassifier uses the cross-entropy loss, which is, at least for classification purposes, a much better loss function. Since, the problem at hand is a classification problem, it makes sense that MLPClassifier will have a better performance than my NN, when it comes to making accurate predictions.

- The confusion matrix obtained in this case is also a lot better, especially when it comes to making predictions for classes other than 0 and 1. It can be seen that confusion matrix (*Figure 2.17*) has a better *recall* when it comes to other classes (2-9), and a better *precision* when it comes to making predictions of classes 0 and 1. Hence, the MLPClassifier's NN has a much better macro-F1 score along with having a much better accuracy. Hence, it is a better model.

Hence, training using existing library and modified loss function is leading to much better classification models.