CLASSIFICATION TECHNIQUES FOR COMMERCIAL PRODUCTS
COMP9417 Machine Learning Project

Name	**ZID**	
Aniket Chhillar	z5205119	Boosted Trees
Hui Chuan Tan	z5106339	Decision Trees
Daniel Pham	z5209600	Nearest Neighbour
Jet Young Lim	z5195170	Neural Networks

1. Introduction

As modern society continues to bloom in technological advances, the increasing demand for different classification systems have become apparent. Machine learning classification algorithms and techniques have become widely adopted to solve these problems. A small subset of these problems include labelling objects of complex features, counting the number of items of a desired object and recognizing handwriting of different languages. In this report, the breakdown of many different machine learning classification techniques will be used to understand the underlying effectiveness of each algorithm in identifying commercial products.

The different classification techniques that will be looked at in this report include *Decision Trees, Nearest Neighbour, Boosted Trees* and *Neural Networks*. As each algorithm is explored, the aim is to achieve a high accuracy score. To further increase the classification capacity of each machine learning algorithm, the algorithms will be ensembled to see whether the accuracy improves.

2. Kaggle

In this report, we will attempt the Otto Group Product Classification Challenge on Kaggle (https://www.kaggle.com/c/otto-group-product-classification-challenge/). Otto Group is an e-commerce company. The dataset 93 features and more than 200,000 rows that represent products. The objective of the challenge is to build a predictive model that can distinguish between the 9 product categories that are given.

We have chosen this particular competition because it presents an interesting and practical real-world problem. The requirements are quite straightforward and clear, and the dataset is of a reasonable size. Therefore it is very suitable for our intention of comparing multiple classification techniques.

2.1 Kaggle Loss Function

Model performance will be evaluated by minimizing the multiclass logarithmic loss.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\, log(p_{ij})$$

Where

N	Number of products in the test set.
log	Function of the natural logarithm.
y_{ij}	1 if observation i is in class j else 0.
p_{ij}	Probability that observation i belongs to class j .

3. Machine Learning Classification Model

3.1.1. Multilayer Perceptron Neural Network

A multilayer perceptron neural network is a supervised learning technique that utilises backpropagation for training with the end goal of predicting the output or target variable through feeding the neurons with a set of input data that consists of the target's features. Below we will discuss and explore how a MP neural network can be used to predict the probability distribution of a multi-class classification problem.

3.1.1.1 Data Preprocessing & Constructing Our Neural Network

Since we are provided with a custom dataset we have to first convert the labels and features into a float tensor form in order to fit it into the TensorDataset function so that we are able to train and validate our data properly. Note that the provided dataset has been split into a 80/20 training/validation set. The processed training set from the TensorDataset function will be passed into the DataLoader function to split our data into mini batches of size 256 each for training later on. The reason we used a batch size of size 256 is because a lower batch size would lead to a longer training time whereas a larger batch size would result in the model producing a lower accuracy. All functions used for the data preprocessing are imported from packages such as scikit-learn and torch.

To build our multilayer perceptron network, we have to import the nn package which by default is provided by the torch package as well. The network's architecture consists of an input layer with 93 nodes that represents the number of unique features, 3 hidden layers with tunable number of hidden nodes for each layer and finally an output layer that outputs 9 nodes that represents the number of unique classes. Batch normalization and dropouts were used to ensure that we don't overfit our model during training. The tanh activation function is used over other activation functions as it was able to provide a lower loss and higher accuracy score. The loss function that we will be using is CrossEntropyLoss which is suited for a multi-class classification problem and it is provided by the pytorch/torch nn package.

3.1.1.2 Network Performance

By default the network is trained on the given hyperparameter settings:

- Learning rate $\eta = 0.001$
- Dropout rate = 0.5
- Hidden nodes = 100

It is also trained in 15 epochs as any number greater than 15 would result in the model overfitting while providing diminishing returns such as obtaining a lower accuracy score on the validation model and a higher loss value. Training on the given hyperparameters and values above would result in the model achieving a training accuracy of 82.5% and validation accuracy of 79% with the training loss resulting in 0.48 which seems to be good as it indicates that our trained model is not overfitting while producing desirable results.

The trained network is then used to run on the test set provided which has to be preprocessed with reasons similar as above and the results produced is then saved into a .csv file for the Kaggle submission. By submitting our given output dataset on Kaggle, we are able to obtain a private score of 0.51970 and a public score of 0.51377 which puts us around position 1570 and 1550 on the leaderboard respectively.

3.1.1.3 Hyperparameter Tuning

In order to further improve/optimize the model, hyperparameter tuning is needed as sometimes small changes to these parameters can lead to an increase in the model's performance. The three main hyperparameters that have been tested on are the number of hidden nodes per hidden layer, the learning and dropout rate. As shown in the models below, we can observe the best number or value for each hyperparameter that allows our model to achieve the best accuracy.

By training our model with our newly obtained hyperparameters, the results obtained after a submission to Kaggle along with the training/validation accuracy shows that it is slightly worse than the original model. But by tweaking with the learning rate parameter specifically from 0.03 (optimal value from the model below) to 0.003 while retaining the other hyperparameters with their optimal value we were able to achieve a better score than the original model. This is significant as it shows how tuning and testing around with different hyperparameter values can impact our models performance. This slight change brought our private and public score to 0.51762 and 0.51230 respectively which puts us around position 1518 and 1545. Note that the optimal values for each hyperparameter will be different each time we run the code but ultimately it should result in us achieving a consistent score as above.

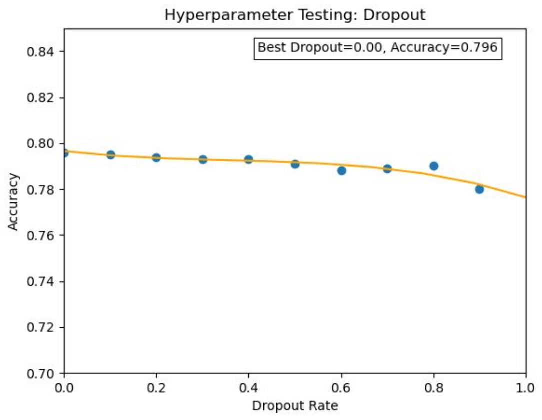

Figure 3.1.1.3.1 Dropout Rate Testing with Validation Set

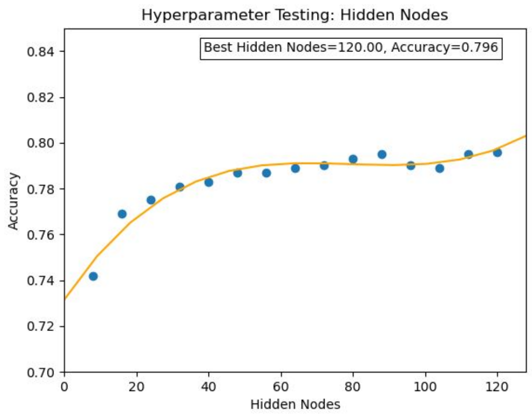

Figure 3.1.1.3.2 Hidden Nodes Testing with Validation Set

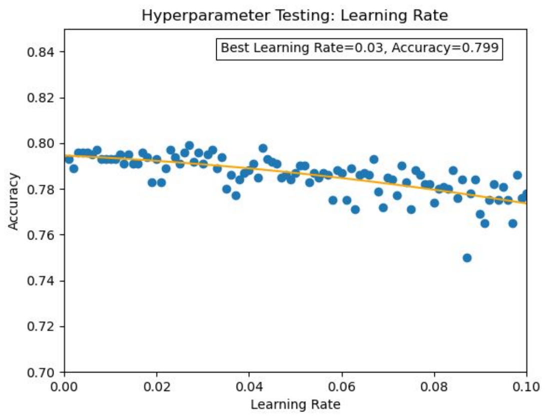

Figure 3.1.1.3.3 Learning Rate Testing with Validation Set

3.1.2. Decision Trees

The decision trees algorithm (DT) is a non-parametric supervised learning method that predicts the value of a target variable using decision rules inferred from the features of the data. In this report, we will explore how DT can be used to predict the probability distribution of a multi-target classification problem.

3.1.2.1 Setup

We use an 80/20 training/validation split with stratification on the dataset. Cross validation is performed with 5 folds on on the training set. The numbers below are generated with the random state set to 42.

We use the scikit-learn DecisionTreeClassifier with the default hyper-parameters unless specified otherwise. Cross validation was performed to investigate the effect of changing the criterion from *gini* to *entropy*, and of changing the splitter from *best* to *random*. It was found that using the defaults of *gini* and *best* resulted in better metric scores.

In the given dataset, we observe that the range of values for the features vary by an order of magnitude, from [0, 10] to [0, 352]. We compared the performance of a model trained on a dataset without feature scaling to another on a dataset scaled using a MinMaxScaler. The resulting metrics were near-identical; within 0.1% of each other. We conclude that DT is not sensitive to variance in the data and proceed to train the model without scaling the features.

3.1.2.2 Baseline Performance

To start with, we train our model without restricting the tree depth or number of nodes, and we do not perform any pruning. This gives us a mean accuracy of 0.72 and a log loss of 9.80 when predicting on the validation set.

While our accuracy seems reasonable, a naive classifier that simply predicts a uniform probability distribution has a log loss of 2.19, which is significantly better than the baseline score we have achieved.

Inspecting our class probability predictions shows that for each row, we predict 1 for some class and 0 for all others. This indicates that the tree has fully grown to fit the data and each leaf node has zero impurity. We can possibly improve by pruning the tree somehow.

3.1.2.3 Post-pruning

To improve our model, we utilise the cost complexity pruning mechanism of the DecisionTreeClassifier. This is controlled by a hyper-parameter alpha (α), where a higher α results in more pruning of the tree, which increases the total impurity of its leaves.

We proceed by generating the list of alphas at each step of the pruning process, and then training/validating our model using each of those values of α. The following result is obtained.

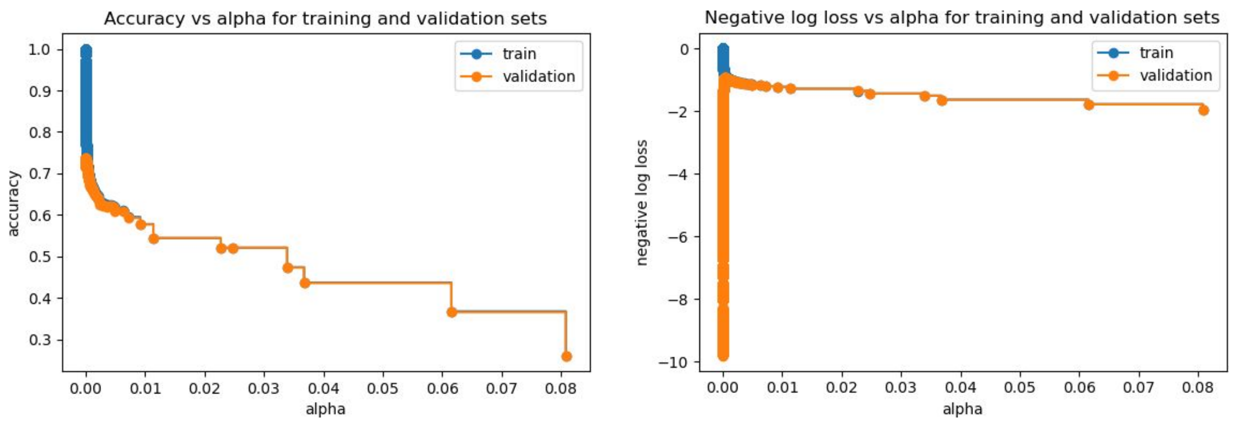

Figure 3.1.2.3.1 Relation between α and accuracy/log loss of the model

We observe that the optimal α for the training set is smaller than that for the validation set. Both are at the very low end of the range, below 0.01.

We now pick 2 values for α. alpha accuracy (α_{acc}) = 0.00007015 where the maximum accuracy of the validation set occurs, and alpha loss (α_{loss}) = 0.00043908 where the minimum log loss of the validation set occurs.

Performing a cross validation using α_{acc} gives us a mean accuracy of 0.73 and a mean log loss of 2.74. Using α_{loss} instead, we get a mean accuracy of 0.70 and a mean log loss of 0.98. We trade a slight decrease in accuracy for a significant improvement in the log loss score.

The log loss scores for the final test set when submitting to Kaggle show a similar improvement; 1.21 when using a model trained using α_{acc} vs 0.94 for α_{loss}.

3.1.2.4 Bagging

As a quick attempt to observe the effects of ensemble learning using decision trees, we use the scikit-learn BaggingClassifier with the default hyper-parameters. This further improves the cross validation mean log loss to 0.79 and Kaggle score to 0.80, which shows that introducing some randomness improves the model. We will take this idea further by exploring XGBoost.

3.1.2.5 eXtreme Gradient Boosted Trees

XGBoost is an implementation of a decision tree ensemble. The model is trained additively - wherein each step the previous generated trees are considered fixed and we add new trees based on optimising our objective function at each step using the following equation(although in our case the loss function will be different as we are doing a classification task, however for the purposes of understanding the following equation is helpful).

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

Fig 3.1.2.5.1 Loss Function at t th iteration. Tianqi C. (2016)

We are representing the loss for adding the t-th tree. For more complex objective functions like the log-loss function the implementation approximates them using a 2nd order Taylor expansion[1]. The algorithm then splits each new tree based on optimising the objective function while also penalizing model complexity.

3.1.2.6 Fitting Default Implementation

Initially we fitted the default XGBoost Classifier to our dataset, using all features. The model was fit with the objective 'multi:softprob' and eval-metric as 'logloss'. This fits the model to predict probabilities and evaluates the model on a log loss basis when calculating CV scores. The model attained a score of 0.47791 on the Kaggle leaderboard.

3.1.2.7 Hyper Parameter Tuning

We felt that our score was not the best we could do with the default implementation, so for the next step we attempted to optimise key parameters using cross validation to get better scores.
From the XGBoost documentation[2] we found the following parameters were essential to

[1] Tianqi Chen, Carlos Guestrin, 16 August 2016,
XGBoost: A Scalable Tree Boosting System, accessed 7/8/2020 <https://arxiv.org/pdf/1603.02754.pdf>

[2]Tianqi Chen XGBoost Documentation, accessed 7/8/2020< https://xgboost.readthedocs.io/en/latest/parameter.html>

controlling how well our model fit the target function:
Within classification tasks there are several parameters to control model complexity:

- max_depth(default = 6) - the maximum depth an individual tree can reach
- min_child_weight(default = 1) - Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning.
- gamma(default = 0) - Minimum loss reduction required to make a further partition on a leaf node of the tree
- n estimators(default = 100) - the number of trees to fit in the ensemble

And also several parameters to introduce randomness to make the model more robust to minimise overfitting:

- subsample(default = 1) - subsample ratio of the training instances
- colsample_bytree(default = 1) - subsample ratio of columns(features) when constructing the tree, occurs once for every tree constructed
- eta(0.3) - step size shrinkage used in updates to prevent overfitting, i.e. "Shrinkage scales newly added weights by a factor η after each step of tree boosting. Similar to a learning rate in stochastic optimization, shrinkage reduces the influence of each individual tree and leaves space for future trees to improve the model"[3]

Visualisation of Parameter tuning on 3-Fold log-loss(other parameters held at default values):

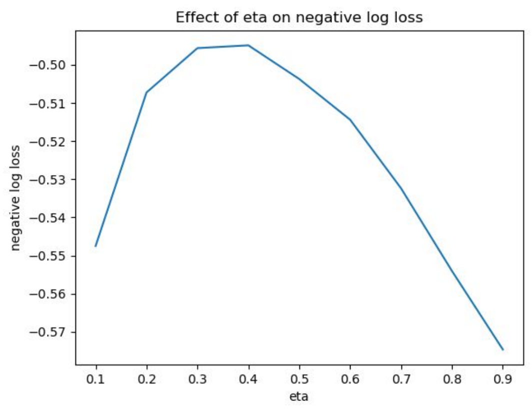

Figure 3.1.2.7.1
Effect of eta on negative log loss

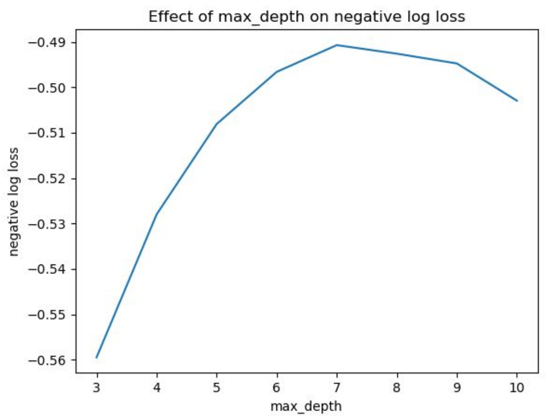

Figure 3.1.2.7.2
Effect of max depth on negative log loss

[3] Tianqi Chen, Carlos Guestrin, 16 August 2016, *XGBoost: A Scalable Tree Boosting System*, accessed 7/8/2020 <https://arxiv.org/pdf/1603.02754.pdf>

Grid Search
For the grid search we tuned eta, max_depth and min_child_weight, however due to issues surrounding computational time we were unable to extensively search the parameter space. Hence we searched only the following values:

Max Depth	Min Child Weight	eta
4, 7, 10	0.3, 0.6, 1	0.3, 0.5, 0.1

We found the optimal parameters to be:

Max Depth	Min Child Weight	eta
0.3	1	0.3

This model attained a score of 0.46827 on the Kaggle scoring.

Bayesian Optimisation

Since we were bottlenecked in our search of the parameter space by the speed of our computers, we now attempt Bayesian Optimisation of parameters and search a much wider parameter space using the python library : bayes_opt[4].
Parameter Space:

- Learning Rate(eta) : (0.01, 1.0),
- Min Child Weight : (0.5, 5.0),
- Max Depth: (3,10),
- Subsample : (0.1,1),
- Colsamplebytree: (0.1,1)
- N estimators : (50,400)

Please note that max_depth is an integer parameter but it is casted to an integer to use the package. Note that casting to an integer can affect the optimisation since the Bayesian optimisation process assumes real

[4] Fernando Noguira, 2014 *Bayesian Optimization Open source constrained global optimization tool for Python*
, accessed 7/8/2020 <https://github.com/fmfn/BayesianOptimization>

parameters[5], however casting was still used. Bayesian Optimisation optimisation treats the objective function as a random function and tries to optimise by assuming a prior distribution and then using the objective values in each iteration of optimisation to try and build a posterior distribution. By using information from previous iterations Bayesian optimization can potentially find global maxima and minima with minimal computation effort[6].
So we apply Bayesian optimisation over the above parameter space setting n_iter(the number of iterations in the algorithm) to 100 and init_points(the number of random starts used to help prevent finding local maxima or minima) to 2.

The returned parameters were:

3-Fold Log Loss	Colsample bytree	Learning Rate	Max Depth	Min Child Weight	subsample	n estimators
0.4728	0.4263	0.1312	9	3.048	0.7148	378

When we fit the model and submitted to Kaggle our score was 0.44389, so it performed significantly better than the grid search, suggesting the Bayesian was able to find a global minima. This model probably performed significantly better than the grid search since we searched over more parameters and a wider parameter space.

3.1.2.8 Feature Creation

Then we tried to improve the performance by adding in new features. We added the sum of features and also the number of non-zero features.
Bayesian Optimisation
We again tried to optimize the hyperparameters using these 2 new features but not over n estimators due to computational constraints. Our ideal parameter set was:

[5] Eduardo C. Garrido-Merch´an, Daniel Hern´andez-Lobato, 30 June 2017, *Dealing with Integer-valued Variables in Bayesian Optimization with Gaussian Processes,* accessed 8/8/2020 <https://arxiv.org/pdf/1706.03673.pdf>

[6] Ruben Martinez-Cantin, 30 May 2014,
BayesOpt: A Bayesian Optimization Library for Nonlinear Optimization, Experimental Design and Bandits, accessed 7/8/2020 <https://arxiv.org/pdf/1603.02754.pdf>

3-Fold Log Loss	colsample_bytree	learning_rate	max_depth	Min Child Weight	subsample
0.4811	0.5954	0.2219	9	2.419	0.8425

When submitting to kaggle our score was 0.46781, which did not beat the model without the extra features, this was likely due to a combination of factors: not optimising over n estimators and the fact that these features did not deliver significant information to the model.

3.1.2.9 Bagging

Bagging a model involves resampling the training dataset and training a model, then combining many models trained using this resampling process and ensembling the models with equal weights. We bagged the model from the original Bayesian parameter search 10 times using the full training dataset. Hoping that by doing so we could improve the score by increasing stability. In doing so we obtained a kaggle score of 0.44248 which marginally beat our previous score and would have achieved a rank of 531/3507.

3.1.2.10 XGBoost Conclusion

Overall we have found that our parameter tuning has been extremely effective, yielding gains in performance and bagging was somewhat effective, yielding only a modest gain in performance.

3.1.3.Nearest Neighbour

Nearest Neighbour (NN) is an algorithm used to identify the label of an object by comparing data points of similar attributes. Identifying objects in NN is achieved by finding data points of close proximity using a distance heuristic such as the *Euclidean Distance* or *Manhattan Distance* of all the points to a point's label being predicted. In this report, we will be looking at K-Nearest Neighbour (K-NNB) which predicts the value of some data points by comparing the closest k data points in the training set.

3.1.3.1 Implementation

Kaggle submission score requires the result of the labels to be in class probabilities. A problem arises when using the scikit-learn module for the K-NNB algorithm as scikit does not output based on class probabilities but rather outputs a single discrete value estimating the correct class probability based on the mode of the labels of the k nearest data points. Thus, to resolve this issue, K-NNB was implemented from scratch utilising pytorch tensors which have GPU support.

To predict the correct label for a given test set, the closest k-points were determined using a distance heuristic. Afterwards, the labels were extracted for the k-points and tallied up for each of the 9 classes. The probabilities were then converted using the following formula.

$$P(Class_i) = Class_i / \Sigma_k Class_k$$

During preprocessing, the training data was normalised using MinMaxScalar from scikit and then the dataset was randomised to prevent bias from occurring.

3.1.3.2 Finding the better heuristic

To evaluate the more effective heuristic in this classification problem, a dataset of size 10,000 is sampled from the otto training set and evaluated with both *Euclidean Distance* and *Manhattan Distance* heuristics under 10-fold cross evaluation. A size of 10,000 data points sampled as a lower dataset would cause higher variance while a larger dataset would be too slow as computing manhattan distance for multiple k-values was extremely slow.

The validation set performance is measured using a custom loss function which resembles the description of the kaggle's loss function.

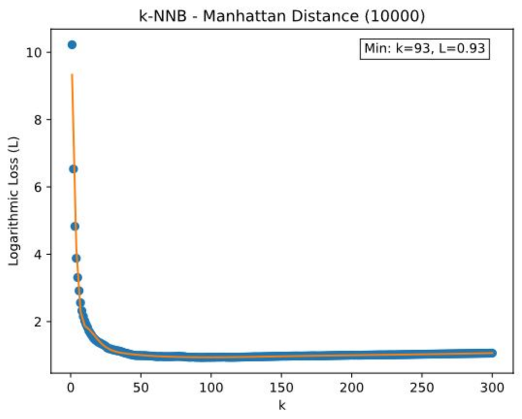

Figure 3.1.3.2.1 K-NNB Manhattan Distance

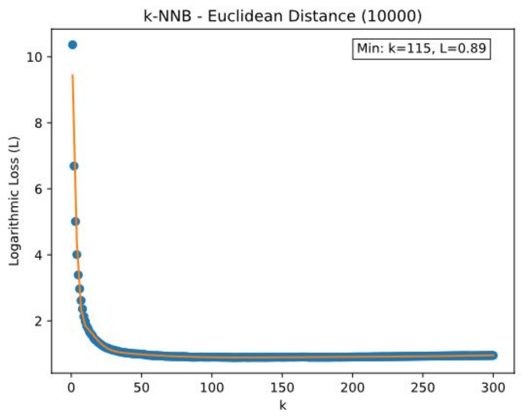

Figure 3.1.3.2.2 K-NNB Euclidean Distance

Observing the results of both the euclidean distance and manhattan distance heuristics, the euclidean distance minimises the logarithmic loss better than manhattan distance being at 0.89 compared to 0.93. However, euclidean distance has a higher k value when the losses are at its minimum being at 115 compared to manhattan's k value of 93, thus having a higher bias. But when comparing the performance of both algorithms, manhattan distance takes 1-3 hours to run with a dataset of 10000 while euclidean distance takes 1-3 minutes. Using a larger dataset, the nearest neighbour performs better as there would be a reduction in variance. Hence, euclidean distance is much better in consideration of time, when utilising much larger training sets.

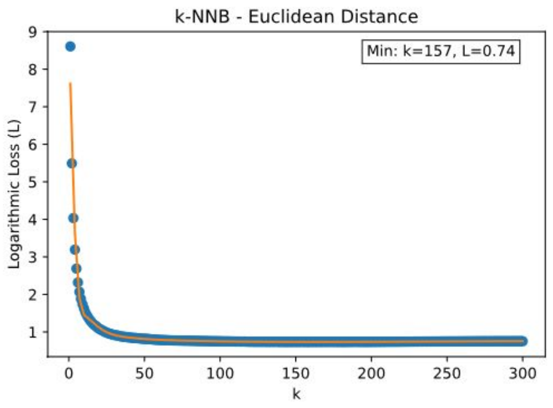

Figure 3.1.3.2.3 K-NNB Euclidean Distance with whole training set

Observing the loss value of euclidean distance with a dataset of 61879, the loss has significantly reduced from 0.89 to 0.74. When running the K-NNB in kaggle, the submission score yielded was 0.72992.

4. Conclusion

FROM OTTO GROUP

Classifier	Score (lower is better)
Decision Tree	0.93749
Decision Tree With Bagging	0.80650
K-NNB	0.72992
Neural Network (Multilayer Perceptron)	0.53332
Boosted Trees	0.44389
Bagged Boosted Trees	0.44248

The decision tree model did not perform as well as expected, even after pruning the tree to avoid overfitting. It could possibly be improved further by tuning the various hyper-parameters that affect the splitting or by weighting the classes. It was observed that introducing randomness in the form of a bagging classifier improved performance considerably.

Surprisingly K-NNB performed exceptionally well despite having a curse of dimensionality. It performed better than decision trees. One consideration for this might be due to the large quantity of training data which improved K-NNB significantly. Furthermore, most features for every class were clustered together very closely thus allowing K-NNB to predict the class labels more accurately.

We found that eXtreme Gradient Boosted Trees and Neural networks were best suited to the task. We found that even with varied parameters Boosted Trees were extremely effective in prediction on the test set, suggesting good model selection. By optimisation of parameters for Boosted Trees we were able to improve the performance of the model. We found that eXtreme Gradient Boosted Trees were the strongest model for the task, this was probably since the boosted model offers great flexibility and capitalizes on the advantage of ensemble learning which we further took advantage of by bagging on top of it.

Overall, Bagged Boosted Trees performed exceptionally well, having a ranking of 531 in kaggle.

5. References

Eduardo C. Garrido-Merch´an, Daniel Hern´andez-Lobato, 30 June 2017, Dealing with Integer-valued Variables in Bayesian Optimization with Gaussian Processes, accessed 8/8/2020 <https://arxiv.org/pdf/1706.03673.pdf>

Fernando Noguira, 2014 *Bayesian Optimization Open source constrained global optimization tool for Python,* accessed 7/8/2020 <https://github.com/fmfn/BayesianOptimization>

Ruben Martinez-Cantin, 30 May 2014,BayesOpt: A Bayesian Optimization Library for Nonlinear Optimization, Experimental Design and Bandits, accessed 7/8/2020 <https://arxiv.org/pdf/1603.02754.pdf>

Simon Low, August 15 2019, XGBoost hyperparameter tuning with Bayesian optimization using Python, accessed 20/7/2020, <https://aiinpractice.com/xgboost-hyperparameter-tuning-with-bayesian-optimization/>

Tianqi Chen, Carlos Guestrin, 16 August 2016, XGBoost: A Scalable Tree Boosting System, accessed 7/8/2020 <https://arxiv.org/pdf/1603.02754.pdf>

Tianqi Chen, XGBoost Documentation, accessed 7/8/2020 < https://xgboost.readthedocs.io/en/latest/parameter.html>