COMP0081: Applied Machine Learning Coursework

Group members:
Rahul Trada, student number: 1018650
Guoliang HE, student number: 19046274
Michail Koupparis, student number: 14062973
Marilena Lemonari, student number: 19045031
Myrto Papakonstantinou, student number: 110025679

April 1, 2020

1 Otto Group Product Classification Challenge

For the first challenge we chose the Otto Group Product Classification Challenge[1]. This challenge is about classifying products into categories (fashion, electronics etc). The evaluation metric for this challenge and thus the one used throughout this section is the Multi-class Logarithmic Loss, Equation 1.

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

where N is the number of products in the set, M is the number of class labels, $y_{ij} = 1$ if observation i is in class j and 0 otherwise and p_{ij} is the probability that observation i came from class j.

1.1 Data

Each row of the dataset represents a different product and to identify the product there is an 'id' attribute which takes unique numerical values. The dataset consists of 93 nu-

[1] https://www.kaggle.com/c/otto-group-product-classification-challenge

merical features, each corresponding to counts of different events. These features are represented as $feat\_1, feat\_2, ..., feat\_93$. Note that there is no definition as to what each feature represents. Finally, there are 9 categories in total for all products represented as $Class\_0, Class\_1, ..., Class\_8$ and defined by the target attribute in the dataset.

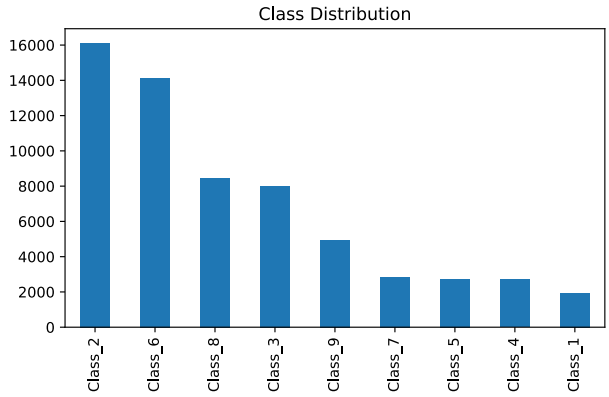


Figure 1: Category distribution of Otto train data.

The challenge creators provide two different datasets (csv files), one for training and one for testing. The train dataset consists of 61,878 products and the test dataset consists of 144,368 products. We used a stratified split of the train dataset into train (80%) and validation (20%) subsets. We chose a stratified split to keep the portion of data from each class the same in each set since the dataset is unbalanced, as can be seen in Figure 1. The validation set was used during training to find the best hyperparameters and to prevent overfitting by keeping track of the validation loss. The best combination of hyperparameters was the one that produced the lowest validation loss.

Moreover, we used PCA (Principal Component Analysis) and TSNE [8] (T-distributed Stochastic Neighbor Embedding) techniques to reduce dimensionality and thus to try to visualise the data (Figure 2). Another reason was to try and see if these dimensionality reduction techniques help to separate the data, and thus if it makes sense to use their results as features, either as additions or by themselves. As can be seen from the figure, TSNE is better at separating the classes and hence could be used as additional features. In the end we did not use these additional features because the second layer (Section 1.3.2) architecture, where TSNE features were going to be added, performed much worse than the first.

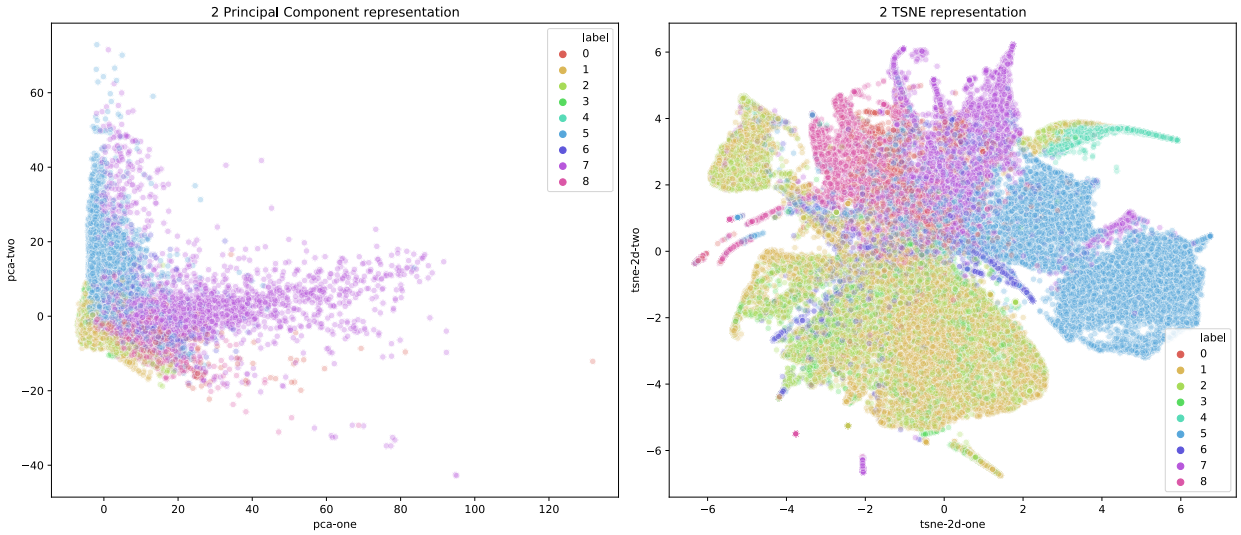


Figure 2: 2-D PCA and TSNE reduction.

1.2 Methods

Some of the best performing models for this challenge, including the winning solution, used a multi-layer-model architecture to tackle this problem[2]. We decided to attempt a simpler version of the architecture referenced above, namely a two-layer architecture with model outputs as input features to the next layer. More specifically, we first trained 22 different models (8 types described below) using the original 93 features of the dataset. As mentioned in Section 1.1, there is no description of the features and thus we did not have a strong intuition for additional features that would be helpful. Although we did not add new features, there are some general features such as number of non-zero elements that people tend to use even without domain knowledge. We appended the outputs of the first layer models to the original 93 features and used these 291 (93 + 22x9) features as inputs into the second layer. The second layer consisted of 3 models, namely AdaBoost, XGBoost, and a Neural Network. The final layer was a weighted average of the predictions of these 3 models. The architecture of the model is shown in Figure 3.

Random Forests:

The random forest model for classification is a method for assigning classes to data points. It operates as an ensemble of multiple, relatively uncorrelated decision trees. This gives an advantage to this method, making it more robust. Each decision tree gives a prediction and the random forest assigns to a data point the class predicted by the majority of trees.

[2] https://www.kaggle.com/c/otto-group-product-classification-challenge/discussion/14335

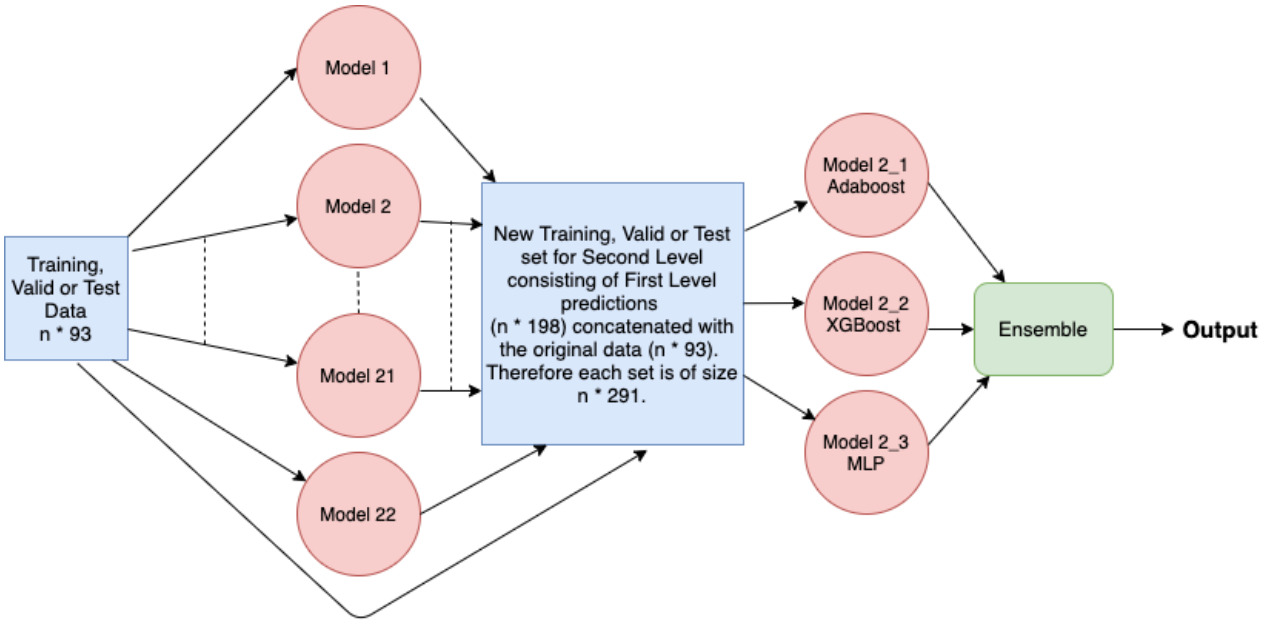


Figure 3: Original two layer architecture

ExtraTrees:

The Extra Trees classifier works similar to the Random Forest classifier with some differences. To build the multiple decision trees, you need multiple datasets. Random Forests use bootstrapping with replacement to accomplish this by picking a certain percentage of the data each time for each tree. Extra Trees draw observations without replacement, hence there is no repetition as in Random Forests. Random Forests select the best split (based on some measure) at each split whereas in Extra Trees a random split is selected. So the randomisation part is in the split at each node rather than in the selection of which data to be used. So, Extra Tress have lower variance and their execution is faster. In the end we used both models to capitalise on both ideas.

Logistic Regression:

Logistic Regression is used for classification in the case of dichotomous dependent variables (two outcomes). The model is an extension of linear classification with the difference that the equation we model is the probability of the outcome and the function (1/1+exp(-x)) is applied to the features. Essentially, we model the relationship between the log odds and the features.

K-Nearest Neighbour (KNN):

KNN is a model for test point classification that involves calculating the distance between

that point and each data point in the training set. The prediction assigned to the test point is the prediction corresponding to the weighted average of the k nearest train points, according to a pre-chosen distance and weight metric.

Neural Networks:

Multi-layer perceptrons (MLPs) or deep neural networks (DNNs) are feedforward networks that consist of at least 3 layers: the input layer, hidden layer and the output layer. MLPs are successful at distinguishing data that is not necessarily linearly separable through the use of non-linear activation functions that transform the data from one layer to the next. They are optimised through gradient descent with backpropagation. Neural Networks are famously adept at automatically extracting features without the need for heavy manual feature engineering.

Boosting:

Boosting is a general method of collecting individual 'votes' from weak learners, and achieving better results.

- Adaboost is a particular way to implement the boosting process. Adaboost [3] helps combine many weak classifiers into a single strong classifier. The core of this algorithm is that every new weak leaner will concentrate on the most difficult examples, which have been misclassified by the previous learners, and the new weak learner is weighted before combining their 'votes' together. For multi-class classification Adaboost can be implemented using the `SAMME.R` algorithm[4].
- Gradient boosting is another boosting method that is based on decision tree models. The objective function of a typical machine learning model can be described as:

$$Obj = \underbrace{\mathcal{L}(\Theta)}_{\text{training loss}} + \underbrace{\Omega(\Theta)}_{\text{regularisation}} \tag{2}$$

  For decision tree models, if we assume there are m examples and K trees, this can be written as:

$$Obj = \sum_{i=1}^{m} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{3}$$

  Upon predicting the example i with tree t, gradient boosting models define the 'gradient' being:

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i) \tag{4}$$

Also, imitating the *taylor expansion*, the objective function becomes:

$$Obj^{(t)} = \sum_{i=1}^{m}(\mathcal{L}(y_i, \hat{y_i}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)) + \Omega(f_t) + const \tag{5}$$

Where g_i and h_i are first order and second order derivatives of $\mathcal{L}(y_i, \hat{y_i}^{(t-1)})$ with respect to $\hat{y^{(t-1)}}$ respectively. By further defining tree models as an index function to map an instance to a leaf and a set of vectors to score each instance in leaves, gradient boosting models optimise the objective function. We used several gradient boosting models namely XGBoost, LightGBM, and CatBoost.

- XGBoost [1] is an optimised distributed gradient boosting library that has been used in Kaggle competitions of the past to great success. It is built to be fast and efficient, and has shown great performance on tabular datasets for classication probelms similar to this Otto challenge.
- CatBoost [10] and LightGBM [7] are other implementations of the gradient boosting algorithm that have shown great success in Kaggle competitions. They both provide support for dealing with categorical features in a dataset.

Ensemble Learning:

The reason why we ensemble the results from individual models comes from the fact that the 'vote' of the crowds may often be correct. A paper has empirically investigated the outcomes of ensemble learning [9], and the findings are that in many cases ensemble learning has a better performance compared with individual models. However, subjected to the dataset, ensemble results can also be zero gain or even negative. We thus choose the ensemble methods based on the best result from the validation set.

Bayesian Optimisation for Hyperparameter Tuning:

Apart from traditional hyperparameter tuning methods such as grid search and random search, we also tried to justify our hyperparameter tuning process via Bayesian Optimisation, which allows us to quickly address the desirable values of hyperparameters.

It is believed that the values of hyperparameter and the performance of the model such as loss, have a black-box function relationship, that is:

$$y = f(\theta) \tag{6}$$

Where y is the loss of the machine learning model, θ is the value of hyperparameters, and f

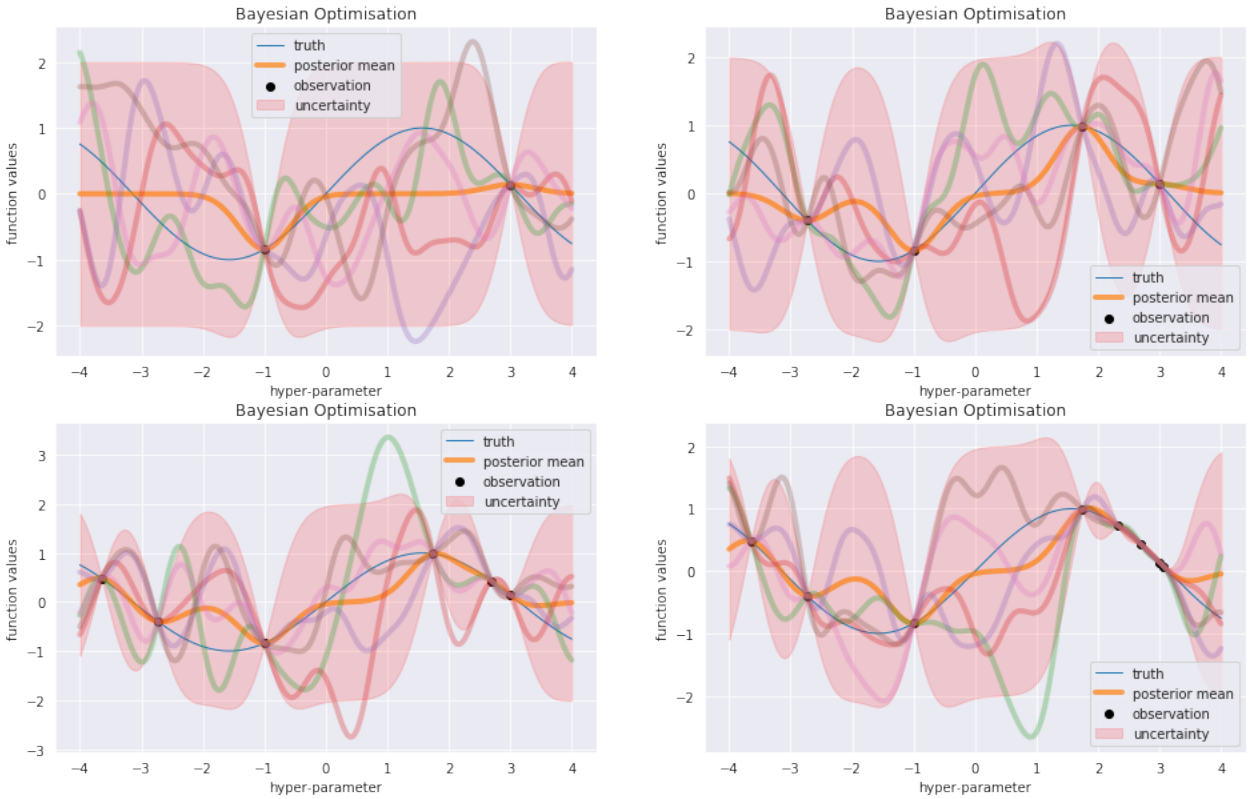


Figure 4: Visualisation of Bayesian Optimisation

is the black-box function. In this way, finding the optimal hyper-parameter value x^* can be rephrased as an optimisation problem:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, f(\theta) \tag{7}$$

Bayesian Optimisation is thus introduced to implement this black-box optimisation process [2].

As shown in Figure 4, a Gaussian process prior distribution with Gaussian kernel is placed on the interval that corresponds to the range of the hyperparameter. To determine the position of the next observation points, Bayesian Optimisation constructs an acquisition function, which is differentiable, and uses the observation point that maximises this acquisition function as the next observation point. The acquisition function used in our case is obtained via expected improvement, as shown by Equation 8.

$$EI_n(x) = max(\triangle_n(x), 0) + \sigma_n(x)\varphi(\frac{\triangle_n(x)}{\sigma_n(x)}) - |\triangle_n(x)|\,\Phi(\frac{\triangle_n(x)}{\sigma_n(x)}) \tag{8}$$

Where $\triangle_n(x) = \mu_n(x) - f_n^*$, f_n^* is the best observation thus far, n is the number of observed points thus far, $\mu_n(x)$ is the posterior Gaussian process mean, $\sigma_n(x)$ is the posterior variance, φ and Φ are *pdf* and *cdf* of standard normal distribution respectively.

The next observation point is then given by:

$$x^* = \underset{x}{\operatorname{argmax}}\, EI_n(x)$$

In this way, optimising the black-box function f becomes optimising Equation 8, which can be done by gradient optimisation methods, such as Quasi-Newton method. The detail implementation of Bayesian Optimisation is shown in the following section.

1.3 Experiments and Results

1.3.1 First Layer

We identified the hyperparameters of each model and tuned them with a variety of techniques such as grid, random, Bayes optimisation and/or manual search. Using the obtained best hyperparameter values, we trained the models and calculated the prediction probabilities for the training, validation and test sets. Then we passed those probabilities to the second layer models concatenated with the original data. Below we show the results for all 22 first layer models separated by the the specific Classification technique they belong to.

Random Forests (1 model):

For the Random Forests model, no manipulation of the dataset is necessary. We chose to incorporate calibration into the model. The model parameters chosen to investigate are five, namely `bootstrap`, determines whether the sample drawn for each tree is with or without replacement, `max_depth`, `max_features`, the number of features to be considered, `min_samples_split` and `min_samples_leaf`. The values considered for each of the five are shown in Figure 5.

Another parameter to be investigated is the number of trees in the forest. This parameter seems to affect the model's performance the most. So the best model, to our knowledge, has `n_estimators=500` and the rest of the parameters have default values (`bootstrap`= True, `max_depth`= 'None', `max_features`= 'auto', `min_samples_split`=2, `min_samples_leaf`=1).

```
base_estimator__bootstrap = [True, False]
base_estimator__max_depth = [None,3,5,7]
base_estimator__max_features = ['auto', 'sqrt', 'log2']
base_estimator__min_samples_split = [1,2,3,4,5,6,7,8,9,10]
base_estimator__min_samples_leaf = [0,1,2,3,4,5,6,7,8,9]
```

Figure 5: Grid Search Random Forests

Extra Trees (1 model):

For Extra Trees the hyperparameters we tuned were the number of estimators-trees, the maximum depth of each tree, bootstrapping and the criterion of the split at each node. The rest of the parameters were the default values of the Sklearn Extra Tress classifier[3]. Also, we tried some pre-process methods such as Oversample to have balanced classes, Standard-Scaled, Log(X+1)-Scaled and PCA. To find the best parameters we used exhaustive linear search. The order of tuning is as shown in Table 1. So we tuned first the number of trees using everything else as the default values, then tuned the maximum depth of each tree using everything as the default value except the number of trees in which we use the best value found before, and so on. The reason we didnt use grid search is the limitation in execution time. The range of values used along with what yielded the best validation loss are summed up in Table 1.

Parameter	**Values**
#Trees	10,50,100,250,500,1000,**1500**
MaxDepth	10,50,**100**,None
Bootstrapping	**False**,True
Criterion	**'gini'**, 'entropy'
Pre-process	**None**, Oversample, Standar-Scaled, Log(X+1)-Scaled, PCA
Train Loss	5.6753 x 10^{-4}
Valid Loss	0.56591

Table 1: Values used to tune Extra Trees with best values in bold

Logistic Regression (1 model):

Firstly, the dataset was transformed from X to log(X+1). Amongst the parameters for that model, we tried to find the best values for the penalty and parameter C (Figure 6). The penalty parameter specifies the norm of penalisation and C is a value representing the inverse

[3] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

of regularisation strength. The combination of the best two values for these two parameters is shown in Table 2.

```
base_estimator__C=uniform(loc=0, scale=4)
base_estimator__penalty = ['l1', 'l2']
```

Figure 6: Grid Search Logistic Regression

Parameter	Values
Penalty	'l2'
C	2.23241
Train Loss	0.62243
Valid Loss	0.61980

Table 2: Best Parameters for Logistic Regression

K-Nearest Neighbour (12 models):

For K-Nearest Neighbour, we use models for different number of neighbors as displayed in Table 3. These models are referenced as **KNN_1**, ..., **KNN_1024** in the rest of this section. The results are summed up in Table 3. We used every model's output for the second layer to capture different distance connections.

#Neighbors	Train Loss	Valid Loss
1	8.1046 x 10^{-15}	8.0430
2	0.16136	4.9706
4	0.28895	2.8874
8	0.38633	1.7095
16	0.46083	1.1254
32	0.52443	0.84923
64	0.58250	0.72303
128	0.63645	0.69740
256	0.69206	0.71082
512	0.75788	0.76381
1024	0.84538	0.84510

Table 3: KNN classifier results with various neighbors.

KNN-128_2:

We did further tuning of the KNN classifier to create another more tuned model. For this model the dataset was transformed from X to log(X+1). The hyperparameters tuned were

the number of neighbours, distance metric and the choice for weights. More specifically, Figure 7 shows the choices for each of those parameters and Table 4 sums up the best values.

```
n_neighbors=[2,8,32,128]
weights=['uniform','distance']
metric=['euclidean','manhattan','chebyshev','seuclidean','minkowski']
```

Figure 7: Grid Search KNN

Parameter	Values
#Estimators	128
Metric	'euclidean'
Weights	'distance'
Train Loss	8.1046 x 10^{-15}
Valid Loss	0.65171

Table 4: Best parameter values for KNN

Neural Networks (3 models):

The input to the MLP was the original 93 features. The hyperparameters tuned were the learning rate, batch size, number of hidden units, number of layers and dropout rate. Batch normalisation was used at every layer. A combination of grid and manual search was performed using values as shown in Figure 8.

```
lr_rates = [0.001,0.01,0.1,0.005]
p = [0.4,0.5,0.6]
batch_sizes = [256,512,1024,2048]
layers_list = [[512,256],[256,128],[256,128,64]]
best_params = grid_search(layers_list,lr_rates,batch_sizes,transformations)
```

Figure 8: Grid Search MLP

Additionally, we employed early stopping at 30 epochs, i.e. we stopped training and recorded the best epoch if the validation loss had not decreased in the last 30 epochs. Moreover, we used the Adam optimiser which is an adaptive learning rate optimisation algorithm making use of momentum and weight decay. In addition, we trained another MLP model using the same settings without batch normalisation. We also tried certain transformations such as $Log(X+1)$ on the input data to produce another MLP model (MLP_3 Log(X+1)) using the best values found for MLP_2. Thus in total we trained 3 MLP models. The best parameters for all models found are displayed in Table 5.

Parameter	Values		
	MLP_1	**MLP_2**	**MLP_3**
Learning Rate	0.001	0.001	0.001
No. Hidden Layers	2	1	1
Hidden Layer Units	[512, 256]	500	500
Batch size	1024	5000	5000
Dropout	0.5	0	0
Preprocess	-	-	Log(X+1)
Train Loss	0.32504	0.36614	0.37661
Valid Loss	0.46589	0.53746	0.49376

Table 5: Best Parameters for all MLP models

CatBoost (1 model):

We mostly adhered to the default values for CatBoost and only manually tweaked learning rate, l2 leaf regularisation and depth. The best parameters are shown in Table 6.

Parameter	Value
Learning Rate	0.1
L2 Leaf Regularisation	9
Depth	10
Train Loss	0.21613
Valid Loss	0.45585

Table 6: Best Parameters CatBoost

XGBoost (2 models):

We first tried various data preprocessing methods on the model with default parameters. These include the square root transform, the log transform, the standard scaling transform, and the Anscombe transform. However, all transforms resulted in the same validation loss of 0.641. Therefore we proceeded from this point without any data preprocessing.

We found that using a GPU massively improved execution time of the model, down from 1 hour to under 5 minutes, without sacrificing too much in the validation loss. This increased efficiency helped in hyperparameter tuning. We decided to do a randomised search as XGBoost has a large number of parameters that could be tuned. The search was performed over the range of values shown in Figure 9. The best parameters found by the randomised search are found in Table 7.

```
params = {'min_child_weight': [1, 3, 5, 10],
          'gamma': [0,1,2],
          'subsample': [0.7, 1.0],
          'colsample_bytree': [0.7, 1.0],
          'max_depth': [4,6,8],
          'n_estimators': [100,200,400,525,600]}
```

Figure 9: Random search XGBoost

Parameter	Value
Subsample	0.7
Num estimators	525
Min child weight	3
Max depth	8
Gamma	0
Colsample bytree	0.7

Table 7: Best Parameters XGBoost

	XGBoost_1	XGBoost_2
Train Loss	0.12839	0.18270
Valid Loss	0.43857	0.43428

Table 8: XGBoost Losses

After that we trained with those parameters using the CPU as it is more accurate, yielding 2 XGBoost models. We tried the same transformations mentioned above with the best one being the Standard-Scaled. Their losses are summed up in Table 8, where XGBoost_1 is GPU-trained and XGBoost_2 is CPU-trained with Standard-Scaled date.

LightGBM (1 model):

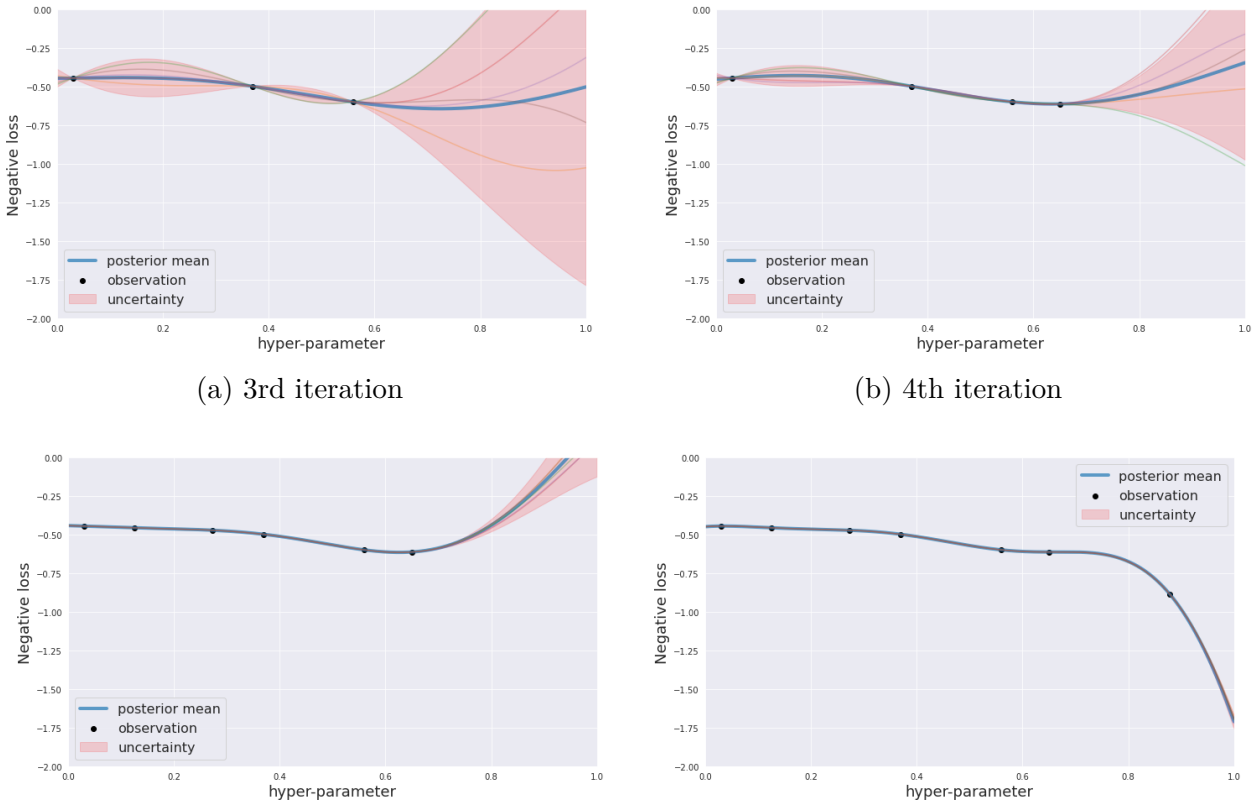
The input to the LGBM classifier was the original 93 features. We performed a combination of random and manual search for hyperparameters with 3Fold Cross-Validation (Figure 10). Early stopping was implemented at 30 rounds of no validation loss improvement. The best hyperparameters of the model are shown in Table 9.

```
param_test ={'num_leaves': range(6, 50),
             'max_bin': range(100, 255),
             'bagging_fraction': uniform(loc=0.2, scale=0.8),
             'feature_fraction': uniform(loc=0.4, scale=0.6),
             'max_depth': range(1, 11, 2)}
```

Figure 10: Random Search LightGBM

This hyperparameter tuning process is also justified by Bayesian Optimisation.

For example, after recording three values of learning rates and the corresponding model



(a) 3rd iteration

(b) 4th iteration

(c) 6th iteration

(d) 7th iteration

Figure 11: Tuning hyperparameter via Bayesian Optimisation

losses, the uncertainty region is quantified by a Gaussian Process (GP), as per Figure 11 (a).

The next suggestion observation point is then given by the Expected Improvement acquisition function 8, being around 0.65. Setting the learning rate to be 0.65, we run the model again, and input the corresponding model loss to GP. This then gives the following figures:

We can see since the model performance is not as good as the previous one, the posterior mean of Gaussian Process decreases around the fourth observation point and uncertainty shrinks to the observation point when learning rate is 0.65.

Iterating the above process, we confirm our choice of learning rate is near optimal as per Figure 11.

Parameter	Value
Learning Rate	0.03
Num Leaves	150
Max Bin	230
Feature Fraction	0.9
Bagging Fraction	0.8
Bagging Frequency	5
Max Depth	9
Train Loss	0.16456
Valid Loss	0.44269

Table 9: Best Parameters LightGBM

Model	Train Loss	Valid Loss
XGBoost_1	0.12839	0.43857
XGBoost_2	0.18270	0.43428
Random Forests	9.8033×10^{-2}	0.47191
Logistic Regression	0.62243	0.61980
LightGMB	0.16456	0.44269
CatBoost	0.21613	0.45585
ExtraTrees	5.6753×10^{-4}	0.56591
MLP_1	0.32504	0.46589
MLP_2	0.36614	0.53746
MLP_3 Log(X+1)	0.37661	0.49376
KNN-1	8.1046×10^{-15}	8.0430
KNN-2	0.16136	4.9706
KNN-4	0.28895	2.8874
KNN-8	0.38633	1.7095
KNN-16	0.46083	1.1254
KNN-32	0.52443	0.84923
KNN-64	0.58250	0.72303
KNN-128	0.63645	0.69740
KNN-128_2	8.1046×10^{-15}	0.65171
KNN-256	0.69206	0.71082
KNN-512	0.75788	0.76381
KNN-1024	0.84538	0.84510

Table 10: **Layer 1** Model Losses

1.3.2 Second Layer

At this stage, we concatenated the original 93 features and the prediction output probabilities of the first layer models to feed into the second layer models. However, contrary to expectations, the models overfitted very early in the training, yielding higher losses than the first layer.

For Adaboost we used Decision Trees as base estimators and used a grid search on the number of base estimators (number of trees), maximum depth for each tree and learning rate. The parameters searched and the losses of the best model are shown in Table 11.

Parameter	Values
#Trees	**1**, 3, 5, 10
MaxDepth	10, 100, **500**
LearningRate	0.1, **0.5**, 0.8, 1
Train Loss	4.8387×10^{-3}
Valid Loss	0.86601

Table 11: Values used to tune Adaboost with best values in bold

Parameter	Value
Colsample bylevel	0.25
Colsample bytree	0.05
Gamma	0
Learning rate	0.05
Max depth	6
Min child weight	6
N estimators	300
Reg alpha	0
Reg lambda	1
Scale pos weight	1
Subsample	0.8
Train Loss	4.9109×10^{-2}
Valid Loss	0.49840

Table 12: Second layer XGBoost parameter values and losses

Parameter	Value
Learning Rate	0.001
No. Hidden Layers	2
Hidden Layer Units	[512, 256]
Batch size	1024
Dropout	0.5
Train Loss	0.27909
Valid Loss	0.49489

Table 13: Second layer MLP parameter values and losses

We also used an XGBoost model in the second layer, starting with the same parameters as that in the first layer. However, we again observed the validation loss being worse, and the model overfitting extremely quickly. We tried manually tuning the parameters, including tuning more parameters than we did in the first layer, but it did not make a significant difference on the result. The best parameters found and the losses of the model are shown in Table 12.

The last second layer model was an MLP. We once again observed ovefitting very early in the training. The same hyperparameter tuning with grid search was performed as shown in Figure 8. We experimented with using only the mean of the first layer models (9 extra features) along with the 93 features and even though there was some improvement, the loss was still higher than in the first layer. The best parameters from the grid search are shown in Table 13.

Despite this rise in validation loss, we adhered to the original strategy and tried weighted mean variations of the resulting second layer outputs as our final predictions. Table 14

shows the losses achieved by the best second layer weighted average trial. As we can clearly see, the validation loss was higher than losses achieved in the first layer models shown in Table 10. The best validation loss of first layer models was 0.43428 coming from XGBoost_2 whereas even the ensembling of second layer models is worse than this (0.47166). This is when we turned to ensembling outputs of the first layer as described in Section 1.3.3, instead of ensembling outputs of the second layer.

Model	**Weight**
XGBoost	0.44509
MLP	0.55491
Train Loss	0.13576
Valid Loss	0.47166

Table 14: Best Results Obtained from Second Layer Ensemble

1.3.3 Ensemble

Since our attempts with the second layer were not successful, we tried ensembling of first layer outputs. This yielded promising results, allowing us to pass the baseline.

We first tried a simple mean over all outputs but found that this was not as good as a weighted average. To find the optimal weights for the models, we used scipy's 'optimize.minimize' method [4]. This function minimises a scalar objective function (in our case log loss) of one or more variables. We randomly selected between 2 to 22 (total number of models from the first layer) each time (for 2000 rounds) and performed the minimise method in order to come up with the best models to use in the ensemble and also their optimal weights. The results are shown in Table 15. This gave us a validation loss of 0.41979 which was an improvement on the best validation loss (0.43428) we had so far.

1.4 Conclusion

Even though the original plan did not yield the expected results, we adjusted the strategy and successfully passed the baseline with the ensemble of Layer 1 models. Our submission is shown in Figure 12.

[4] https://www.kaggle.com/hsperr/finding-ensamble-weights

Model	**Weight**
XGBoost_1	4.7275×10^{-1}
XGBoost_2	1.2377×10^{-1}
Random Forest	5.0630×10^{-2}
Logistic Regression	3.5101×10^{-17}
LightGMB	3.3776×10^{-2}
CatBoost	1.4790×10^{-3}
ExtraTrees	1.0152×10^{-2}
MLP_1	2.2621×10^{-1}
MLP_3 Log(X+1)	7.5461×10^{-3}
KNN-1	2.9119×10^{-2}
KNN-2	2.3303×10^{-2}
KNN-4	5.2817×10^{-3}
KNN-8	1.5984×10^{-2}
KNN-32	3.9302×10^{-18}
KNN-128	1.6371×10^{-17}
KNN-256	3.2797×10^{-18}
KNN-1024	7.5894×10^{-18}
Valid Loss	0.41979
Final Test Loss	0.43121

Table 15: Otto Final Ensemble

Figure 12: Otto Submission

References

[1] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[2] Peter I. Frazier. "A Tutorial on Bayesian Optimization". In: *arXiv e-prints*, arXiv:1807.02811 (July 2018), arXiv:1807.02811. arXiv: 1807.02811 [stat.ML].

[3] Yoav Freund, Robert E Schapire, et al. "Experiments with a new boosting algorithm". In: *icml*. Vol. 96. Citeseer. 1996, pp. 148–156.

[4] Trevor Hastie et al. "Multi-class adaboost". In: *Statistics and its Interface* 2.3 (2009), pp. 349–360.

[5] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[6] Gao Huang et al. *Densely Connected Convolutional Networks*. 2016. arXiv: 1608.06993 [cs.CV].

[7] Guolin Ke et al. "Lightgbm: A highly efficient gradient boosting decision tree". In: *Advances in neural information processing systems*. 2017, pp. 3146–3154.

[8] Laurens van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE". In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605.

[9] R. Maclin and D. Opitz. "Popular Ensemble Methods: An Empirical Study". In: *arXiv e-prints*, arXiv:1106.0257 (June 2011), arXiv:1106.0257. arXiv: 1106.0257 [cs.AI].

[10] Liudmila Prokhorenkova et al. "CatBoost: unbiased boosting with categorical features". In: *Advances in neural information processing systems*. 2018, pp. 6638–6648.