COMP0081: Applied Machine Learning Coursework Report

Agnieszka Dobrowolska*
MSc Machine Learning
16034489

Tom Grigg*
MSc Computational Statistics & Machine Learning
19151291

Kelvin Ng*
MSc Machine Learning
707184

Oliver Slumbers*
MSc Computational Statistics & Machine Learning
19027699

Ritwick Sundar*
MSc Machine Learning
19125221

Abstract

We present this report as a comprehensive review of work done towards the completion of the given coursework, with separate sections dedicated to analyses of methods used in each of the Kaggle competitions that we participated in. In **Part 1**, we detail the methods used in developing a competitive model in Amazon's Employee Access Challenge, while **Part 2** focuses on the steps undertaken for the Otto Group Product Classification Challenge.

1 Amazon.com - Employee Access Challenge

Here, we describe in detail our approaches towards implementing a model that obtained an AUROC score of **0.92358** on the Private Leaderboard.

1.1 Introduction

When an employee at any company starts work, they first need to obtain the computer access necessary to fulfil their role. This access may allow an employee to read/manipulate resources through various applications or web portals. It is assumed that employees fulfilling the functions of a given role will access the same or similar resources. It is often the case that employees figure out the access they need as they encounter roadblocks during their daily work (e.g. not able to log into a reporting portal). A knowledgable supervisor then takes time to manually grant the needed access in order to overcome access obstacles. As employees move throughout a company, this access discovery/recovery cycle wastes a nontrivial amount of time and money.

There is a considerable amount of data regarding an employee's role within an organisation and the resources to which they have access. Given the data related to current employees and their provisioned access, models can be built that automatically determine access privileges as employees enter and leave roles within a company. These auto-access models seek to minimise the human involvement required to grant or revoke employee access.

*Equal contribution

1.2 Objective

The objective of the competition was to develop a model that would be able to determine an employee's access needs, such that manual access transactions (grants and revokes) are minimised as the employee's attributes change over time. The model would take as input an employee's role information and a resource code, and would return whether or not access should be granted. We were provided with historical data upon which to train.

1.3 Data Description

The data consisted of real historical data collected from 2010 & 2011. Employees were manually allowed or denied access to resources over time. We had to come up with an algorithm capable of learning from this historical data to predict access approval/denial for an unseen set of employees. The data was split into the seen and unseen data as `train.csv` and `test.csv`.

The training set `train.csv` contained rows which each had an ACTION column, (ground truth), RESOURCE column, and data about the employee's role at the time of approval. Each row corresponds to whether an employee having the listed feature values was given or denied access to the listed resource. Table 1 describes the columns present in the dataset. The `test.csv` file was the test set for which predictions were to be made.

Table 1: Column Description

Column Name	Description
ACTION	ACTION is 1 if the resource was approved, 0 if the resource was not
RESOURCE	An ID for each resource
MGR_ID	The EMPLOYEE ID of the manager of the current EMPLOYEE ID record; an employee may have only one manager at a time
ROLE_ROLLUP_1	Company role grouping category id 1 (e.g. US Engineering)
ROLE_ROLLUP_2	Company role grouping category id 2 (e.g. US Retail)
ROLE_DEPTNAME	Company role department description (e.g. Retail)
ROLE_TITLE	Company role business title description (e.g. Senior Engineering Retail Manager)
ROLE_FAMILY_DESC	Company role family extended description (e.g. Retail Manager, Software Engineering)
ROLE_FAMILY	Company role family description (e.g. Retail Manager)
ROLE_CODE	Company role code; this code is unique to each role (e.g. Manager)

1.4 Initial Role-Resource Analysis

It was clear that for the given task at hand, we were trying to find relationships between roles and resources that helped to determine the action outcome. Hence, we postulated that the training dataset would provide information related to such role-resource relationships. However, after some exploration, we observed the following:

There seemed to be a low number of action values that corresponded to a given resource, as depicted in Figure 1. More than half the resources had just a single observation. Another interesting observation (as shall be seen in the following section) was that a very high percentage (94.2%) of "access requests" resulted in approvals (ACTION = 1).

Any implemented model would have to be careful to regularize against the very small amounts of data per role-resource relationships to avoid overfitting to the small number of points in each subspace. This was corroborated by the numerous discussions on this project in the Kaggle competition forums. Throughout the project, we prioritised investigating higher-dimensional role attribute feature spaces that could not only better represent the relationships between roles and resources, but also allow examples from different resources to help in our model's prediction.

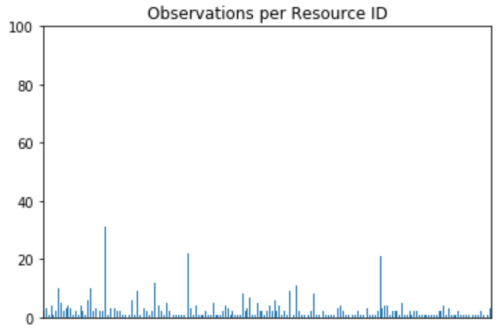

Figure 1: Role-Resource Relationships

1.5 Data Preprocessing

We split the data preprocessing task into three steps: balancing the dataset, applying transformations/feature engineering and feature selection.

Balancing the Dataset The given dataset was very imbalanced with the number of classes denoting `ACTION=0` (access not granted) much less prominent compared to the alternative. As depicted in Figure 2, the ratio of `ACTION=0` to `ACTION=1` in the training set was observed to be approximately 1:16.25. Various approaches exist to correct for the imbalanced classification problem in order to make the dataset suitable for standard machine learning models. Here, we decided to use *Random Oversampling and Undersampling*.

We balance the data distribution by sampling the minority class with replacement (i.e. oversampling). Although performance was improved using this sampling method, the shortcomings of oversampling are clear: oversampling will potentially result in overfitting due to the repetition of training datapoints. We accounted for this with implicit regularization provided by using a large number of estimators in our ensembling models (discussed later).

Figure 2: Imbalance observed in the ACTION class

1.5.1 Feature Engineering

One Hot Encoding Since the original features were discrete category values, we could not directly train models on the given dataset. In order to do so, we had to make use of *one-hot encoding*; where we replace categorical values with binary-valued dummy variables. It is clear that this encoding method will expand the feature space, potentially increasing computation time - however, since most of the sample values were '0', we used a sparse matrix to represent the newly encoded feature space.

Creating New Feature Sets After going through past discussions on the Kaggle Forum, we found that leading solutions made fairly extensive use of dataset transformations. Our strategy was to

produce new feature sets after model performance plateaued on the original features. We ended up producing two new feature sets. One feature set would be categorical and was to be modeled with decision tree based approaches, and the other feature set was to be a sparse matrix of binary features. The first feature set made use of the original categorical features, with some features merged (due to them being a subset of other feature variables). We created the second feature set by binarizing all categorical values along with their respective second and third degree interaction terms. This second feature set was to be used within our Logistic Regression model. Elaborating on the manner in which we created our first feature set, we merged ROLE_TITLE with ROLE_FAMILY as well as ROLE_ROLLUP_1 with ROLE_ROLLUP_2, as we noticed in the data that these features were subsets of the other. A noticeable speed up in code execution was noticed by merging the features in this manner. In addition to this, the counts of the remaining five features, the percentage of a categorical feature's total requests coming from each particular resource, and the number of unique resources that a MGR_ID received requests for were part of our first feature set, and this is depicted in Table 2.

Table 2: Engineered Feature Set Description

Column Name	Description
ACTION	ACTION is 1 if the resource was approved, 0 if the resource was not
RESOURCE	An ID for each resource
MGR_ID	The EMPLOYEE ID of the manager of the current EMPLOYEE ID record; an employee may have only one manager at a time
ROLE_ROLLUP_2	Company role grouping category id 2 (e.g. US Retail) Merged ROLE_ROLLUP_1 with this feature as it was a subset
ROLE_DEPTNAME	Company role department description (e.g. Retail) Retail Manager)
ROLE_FAMILY_DESC	Company role family extended description (e.g. Retail Manager, Software Engineering)
ROLE_FAMILY	Company role family description (e.g. Retail Manager) Merged ROLE_TITLE with this feature as it was a subset
COUNTS	Counts of the remaining 5 features (excluding merged features)
REQ_PERCENT	Percentage of a feature's total requests coming from a particular RESOURCE
NUM_UNIQUE	Number of unique resources that a MGR_ID received requests for

Feature Importance Another aspect to the features that we explored throughout were their relative *importance*, particularly with respect to feature selection later on. We quantified feature importance using a simple Random Forest Classifier. Each time a split is made on a feature that increases separation/reduces entropy between the two classes in feature space, we attribute this reduction to the given feature. We then aggregate all such attributions across the many (high-variance) decision trees in our random forest, producing a statistic that gives a good measure of a feature's overall ability to classify the dataset, taking into account interactions with other features at different levels in a tree-based classifier. (This is the default implementation of `RandomForestClassifier.feature_importances` in `sklearn`.)

From our dataset, there was no guarantee that a feature that was more important by this measure would necessarily make better decisions, but these statistics could be used to guide our modelling process. Some feature importances are depicted in this manner in Figure 3.

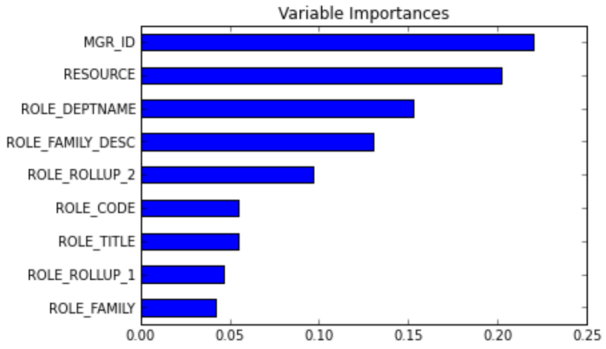

Figure 3: Importance of Feature Variables

Feature Selection We performed a greedy forward selection on the one-hot encoded data by training the model on ever increasing subsets of the original transformed columns. Here, we selected groups of encoded binary features, and not the individual binary feature bits. We also decided to group all sparse features together for each type of feature (just this one change resulted in an improvement of our score by 0.008). Getting rid of a couple of feature combinations that were redundant also helped to speed things up a bit and reduce complexity. For instance, ROLE_TITLE was a subset of ROLE_FAMILY, so there wasn't any reason to look at a higher order feature that combined the two.

1.6 Description of Our Models

Given the problem and the dataset, we initially thought of proceeding with implementing two models to see how they fared, with one being a *Naive Bayes* model, and the other being a *Logistic Regression* model.

Naive Bayes For this model, we make the vastly oversimplifying assumption that all features are mutually independent. This "naive" independence assumption would allow us to apply the Naive Bayes algorithm in the given categorical data. Also, Laplace smoothing would be applied to those features never seen during training. We used `sklearn`'s implementation for this model.

Logistic Regression Commonly used linear classifier, which is not only simple, but also efficient. In it's basic form, uses a logistic function to model a binary dependent variable. Our idea was to make use of the first set of sparse features (as detailed in the section above) to be used with Logistic Regression and other sparse classifiers. A second set of non-sparse features was made for ensemble methods. We used `sklearn`'s implementation for this model.

1.6.1 Ensemble Methods

Ensemble Methods are meta-algorithms that combine several machine learning techniques into one predictive model in order to decrease variance (bagging), bias (boosting), or improve predictions (stacking). For our model, we used the sparse binary matrix feature set in the *Random Forest*, *Extra Trees* and *Gradient Boosting Methods*. Here, we shall elaborate a bit on each of these classifiers.

Random Forest Classifier Consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest gives a class prediction, with the class having the most votes becoming the prediction of the model. The reason the *Random Forest Classifier* works well is due to the large number of relatively uncorrelated models (trees) operating as a committee that outperforms any of the individual constituent models. The low correlation between models is key, with the reason for this being that the individual trees "protect" one another from individual errors.

Extra Trees Classifier A type of ensemble learning. technique which aggregates the results of multiple de-correlated decision trees collected in a "forest" to output it's classification result. It is similar to a *Random Forest Classifier*, and only differs from it in the splitting behaviour of the decision trees in the forest (see our Otto Group report for the exact nature of this difference).

Each decision tree in the *Extra Trees Forest* is constructed from the original training sample. Then, at each test node, each tree is provided with a random sample of *k* features from the feature set from which each decision tree must select the best feature to split the data based on some mathematical criteria (typically the Gini Index). This random sample of features leads to the creation of multiple decorrelated decision trees.

Gradient Boosting Methods Boosting builds models from individual "weak learners" in an iterative manner. The individual models are not built on completely random subsets of data and features but sequentially by putting more weight on instances with wrong predictions and high errors. The idea behind this is that instances which are hard to predict correctly will be focussed on during learning, so that the model learns from past mistakes.

The gradient is used to minimise a loss function. In every round of training, the "weak learner" is built and it's predictions are compared to the correct outcome that we were expecting. The distance between the prediction and truth represents the error rate of our model, with these errors being used to calculate the gradient. In *Gradient Boosting*, we combine the predictions of multiple models, so we are not optimising the model parameters directly, but instead, the boosted model predictions. In this manner, the gradients will be added to the running training process by fitting the next tree to these values.

1.6.2 Model Implementation

All of our models were implemented using *scikit-learn*. Table 3 contains the values of the various parameters of the Random Forest, Extra Trees and Gradient Boosting classifiers, with each of the parameters described as follows:

- n_estimators : The number of trees in the forest.
- max_features : The number of features to consider when looking for the best split.
- max_depth : The maximum depth of the tree.
- min_samples_split : The minimum number of sample required to split an internal node.

Table 3: Classifier Implementation Details

Model	n_estimators	max_features	max_depth	min_samples_split	learning_rate
Random Forest	999	sqrt	None	9	–
Extra Trees	999	sqrt	None	8	–
Gradient Boosting	50	–	20	9	0.20

The hyperparameters for our models were obtained via a combination of Kaggle Leaderboard discussions and our Grid Search implementation. Our Logistic Regression model was implemented using a 80/20 Train-Validation split. A L2 penalty was used to regularize. We do not need to weight our classes because we have already performed random oversampling. and the class weight being set to balanced. We didn't select the best performing Logistic Regression model based purely on raw score, but also on the standard deviation between the 10 folds.

As mentioned earlier, we used the sparse feature set in the Ensemble Methods described above, and then combined them with our Logistic Regression model, after having transformed the sparse features in the Ensemble Methods with an inverse sigmoid function. The blend of our Random Forest Classifier, Extra Trees Classifier and Gradient Boosting Method was in the ratio $0.25 : 1 : 0.10$ (this idea was borrowed from the Kaggle Discussions Forum). Initially, our ensemble methods blend obtained a score of 0.9043 on the Private Leaderboard as compared to a score of 0.9139 for our Logistic Regression model. Using a $\frac{2}{3} : \frac{1}{3}$ ratio split of our Logistic Regression model and Ensemble Methods enabled us to score much higher on the Private Leaderboard.

We feel that the reason our model was able to score quite high was due to the fact that we tinkered a bit with the 2nd and 3rd degree interaction terms. We merged together certain feature categories based on how frequently they occurred, with our greedy forward selection (detailed in Feature Selection) selecting these combinations.

Table 4: Amazon.com - Employee Access Challenge Results

Model	Private Leaderboard	Public Leaderboard
Leader	0.92360	0.92964
(RF : ET : GBM) + Logistic Regression	0.92358	0.92700
Baseline	0.91696	–

1.7 Model Performance

The performance of our model, which consisted of a combination of the Ensemble Methods with the Logistic Regression model, can be seen in Table 4. Figure 4 depicts the scores obtained by our model on the Private and Public Leaderboards after submission of the csv file to Kaggle.

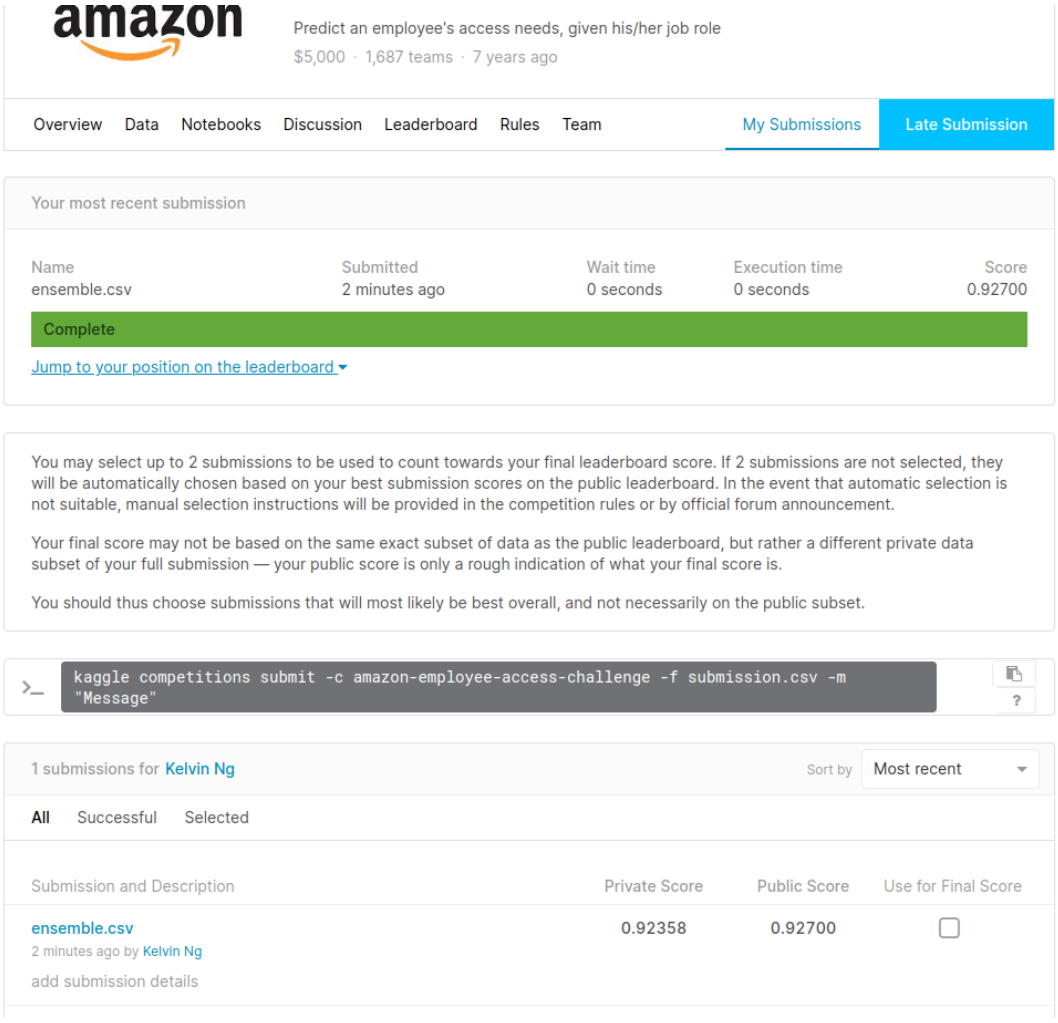

Figure 4: Scores obtained on Private and Public Leaderboards

2 Otto Group Product Classification Challenge

2.1 Introduction

For big commercial companies, such as Otto, analysis of product performance is of crucial importance. However, the process of automatic categorisation of individual products poses a significant challenge, and often results in products which are identical being assigned to different categories. Thus, improving product classification has a direct impact on downstream tasks, such as quality assessment and product satisfaction analysis. For this competition, we were provided a dataset with 93 feature columns and more than 200,000 rows, each corresponding to a product.

2.2 Objective

Our objective was to build a model that, given the 93 feature values, classified a product into one of the 9 possible categories.

Metrics

The metric chosen by *Otto* to evaluate models was the multiclass log loss - this metric correlates somewhat with a model's discriminatory power, and so seemed appropriate for the task at hand.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log\left(p_{ij}\right)$$

In particular, using log-loss results in heavy penalties being applied where a low probability is assigned by the model for the correct underlying class. In this project, we began by building individual models that performed well, and then combined them using model stacking to obtain a low final loss.

2.3 Data Description

The dataset consisted of 93 features, labelled only as $feat_1, feat_2, \ldots, feat_{93}$, thus giving no intuition for interpreting what the feature represents. This challenge was unusual in that the test set was much larger than the training dataset, each containing 144,268 and 61,878 instances, respectively. This meant that particularly good validation practises were necessary in order to ensure generalisation to unseen data.

All of the features were discrete features that had already numericalised, with the majority of them being binary - i.e. corresponding to some unknown boolean property of the product.

2.4 Exploratory Data Analysis

Imbalanced Data We found that the categories were not balanced, as shown in Figure 5. The most represented class was class 2, with 16122 observations, while the least represented class was class 1, with 1929 observations. The observed extreme differences in class size were initially a cause for concern, and we experimented with various methods for balancing our dataset, such as randomized oversampling and undersampling (discussed earlier). We note that we did not try to reweight the loss function relative to target class size as this did not make sense given that the loss function calculation was fixed by *Otto*. In our final model, we found that rebalancing the dataset via sampling had little effect on the performance of our models with respect to the significant increase in training time.

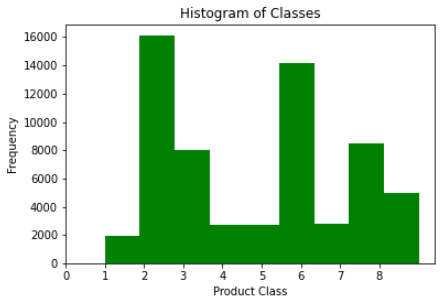

Figure 5: Distribution of product classes in the training dataset.

To visualise the separability of the classes we produced some t-SNE visualisations of the raw training data, as seen in figure 6.

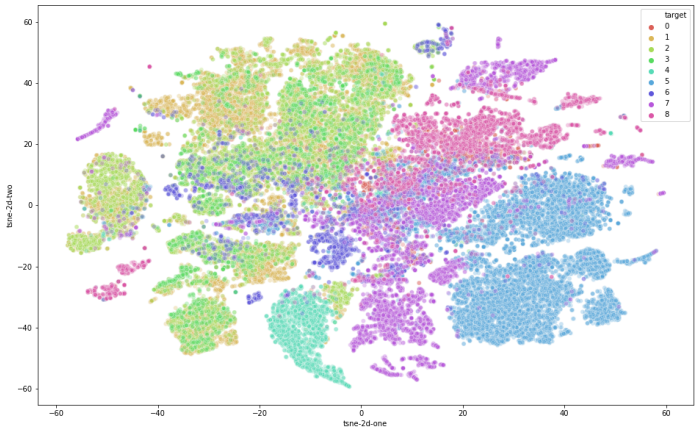

Figure 6: T-SNE visualisation of the raw training data.

From our t-SNE plot, it is clear in particular the classes 3, 5, and 7 are distinct from the others, and that generally the dataset is quite separable through this particular dimensionality reduction technique. This perhaps gives some intuition as to why the class imbalance did not significantly affect our final model's performance: easily separable clusters mean that the model is less likely to find an "optimal" point in parameter space corresponding to consistently guessing the majority class.

Correlations We also performed some correlation analysis on the dataset for use in feature selection (since we had little intuition to go on based on the anonymous feature names). We highlighted large magnitude feature correlations using the heat/size-map visualisation technique shown below in Figure 7.

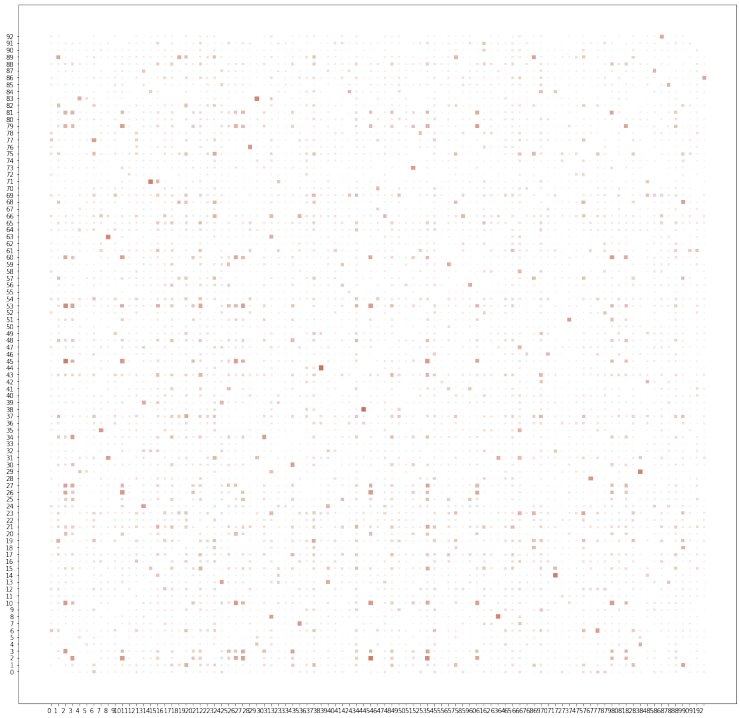

Figure 7: Correlation between feature columns, bigger squares correspond to larger absolute correlation values. Redder squares correspond to more positive, bluer to more negative.

2.5 Data Preprocessing

Transformations In this project, we plateaued early in terms of our model score when building up ensemble models using only the raw data. Despite being able to create multiple individual models that performed well on the test set, past a certain point, more models in our ensemble did not appear to lead to increased performance on the test set. We surmised that our models were not diverse enough, and confirmed this by assessing our model output's correlations to one another. We began to experiment with different data transformations to capture alternative representations of the underlying feature space. In order to find suitable diversifying transformations, we analysed the relationships between features in the training data, and tried to come up with transformations that tended to increase the observable variance so as to hopefully capture more information. An example of this intuition is demonstrated below: Figure 8 visualises the relationship between two features in the raw dataset, and Figure 9 shows the increased variance after performing a simple log transformation.

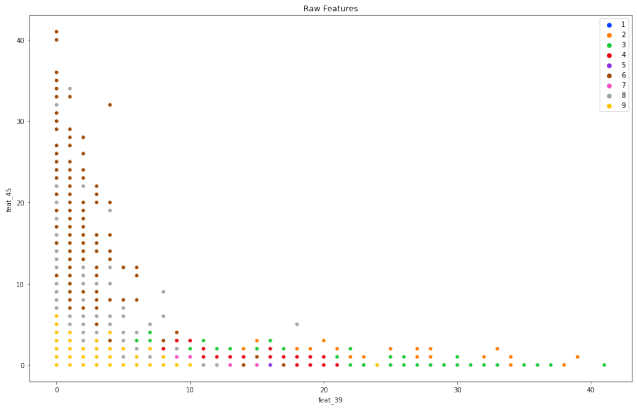

Figure 8: Visualisation of the relationship between two raw features from the training set.

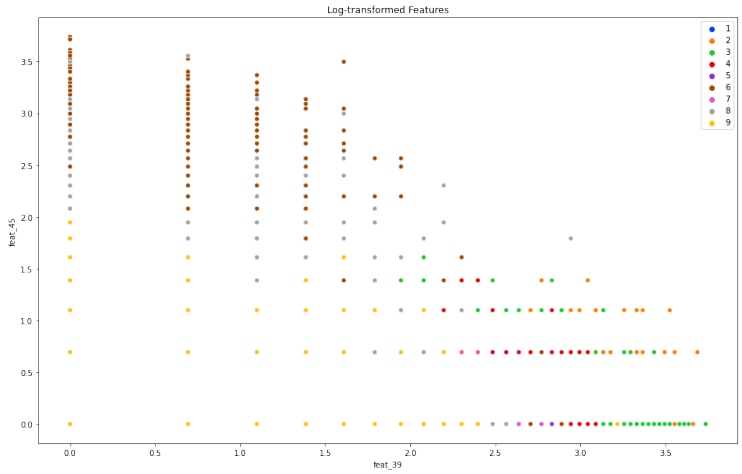

Figure 9: Visualisation of the log-transformed relationship between two features from the training set.

Figure 10: t-SNE visualisation of the log-transformed training data.

We experimented with many different transformations guided by experiments including visualisations such as the ones above. After identifying a potentially useful transformation, we would more rigorously examine the transformation by retraining a subset of our existing individual models on the transformed dataset to determine the effectiveness of the data transformation. We did this in two steps: first evaluating the final loss value of the classifier to make sure it wasn't too high, and then observing the correlation to the outputs of the same model trained on the raw data. This ensured that these data transformations lead to predictive and diverse models. Ultimately, across the different models that were included in our final ensemble, we used 3 different forms of the dataset:

- The raw dataset
- Log-transformed - $log(x + 1)$ applied to all cells.
- TF-IDF dataset - term frequency-inverse document frequency applied column-wise to the raw data.

Dimensionality Reduction We also tried to apply dimensionality reduction techniques to the dataset in the hopes that this would allow us to train more diverse models - however, for any reasonable reduction we experienced a significant increase in model loss and seemingly no useful additions to our ensemble. In hindsight, this perhaps makes some sense as we did not find many large correlations in our dataset, suggesting that each feature dimension contained some measure

of independent information. Looking at Figure 11, we can see that using unsupervised reduction techniques such as PCA does not appear to lead to data that is easy to cluster, this is visually true in 2 dimensions, and seemed to be the case in practice with more dimensions.

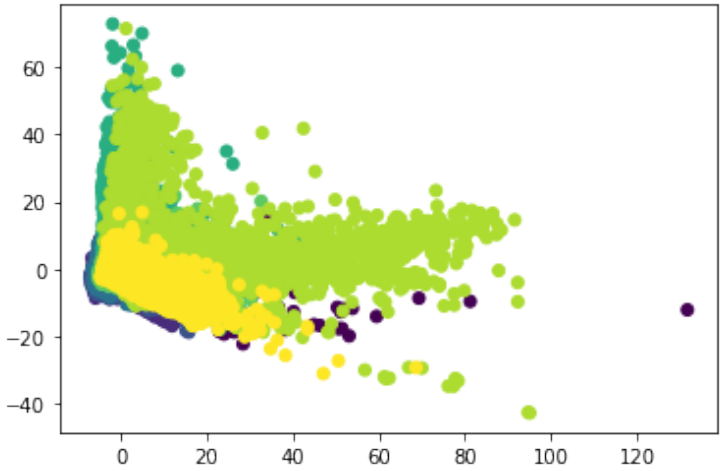

Figure 11: Dimensionality reduction on the raw training data using PCA.

2.6 Models

After researching past solutions and discussions of the competition on Kaggle, we found that the majority of the leading solutions used some form of model stacking. Hence, from the outset we set up a general stacking framework (implemented by sticking to a fixed directory structure on Drive), which allowed us to have arbitrarily many levels of stacking. Given that we would be stacking and ensembling models, we were conscious from the beginning that model diversity as well as individual model performance was important in order to get the benefits of high variance ("Wisdom of the Crowd"). We therefore focused our efforts on effectively training four sufficiently different model classes to include in our final ensemble:

- Neural Networks
- KNN
- Gradient Boosted Trees
- Bagged Trees

Below we outline some of our findings regarding each of these model classes, while more explicit model parameterisations for the individual models found in our final ensemble can be found in the appendix. Following this, we discuss our approach to ensembling and our final best-performing model.

2.6.1 Neural Networks

Architectures We defined n-layer perceptrons using PyTorch that were parameterizable in terms of the number layers n, the number of neurons in each hidden layer, and the activation functions used on each hidden layer's output. We generally found that neural networks did not perform particularly well individually on this dataset, perhaps due to the fact that the features were discrete and so the networks were slightly more prone to overfitting in feature space. Normalization and the use of rectified linear unit activation functions helped, but our neural networks were not able to add to the predictive power of our ensemble until we used them on the normalized log transformed dataset. Every neural network included in our final ensemble was trained on this version of the dataset. To obtain our final neural network architectures and hyperparameters, we initially ran a large and sparse architecture search using PyTorch's default SGD parameters, followed by a round of manual tuning in order to identify more narrow ranges of hyperparameters once usable architectures had been discovered. We finished

with a more narrow and dense grid search over optimization hyperparameters on a small subset of architectures that we had identified as most performant. In Figure 12 we show a 3-dimensional graph corresponding to our grid search over 3-layer architectures.

Optimizers and Regularization Experimenting with different optimisers, we found that SGD performed better than Adam on this dataset. Inspired by forum discussions, we manually implemented a function which allowed us to linearly decrease the learning rate with increasing number of epochs. Using dropout was a crucial strategy for preventing overfitting and forcing our individual networks to be more powerful. We found that a dropout value of 0.5 combined with a small amount of $l2$ regularization generally performed well on the normalized log dataset.

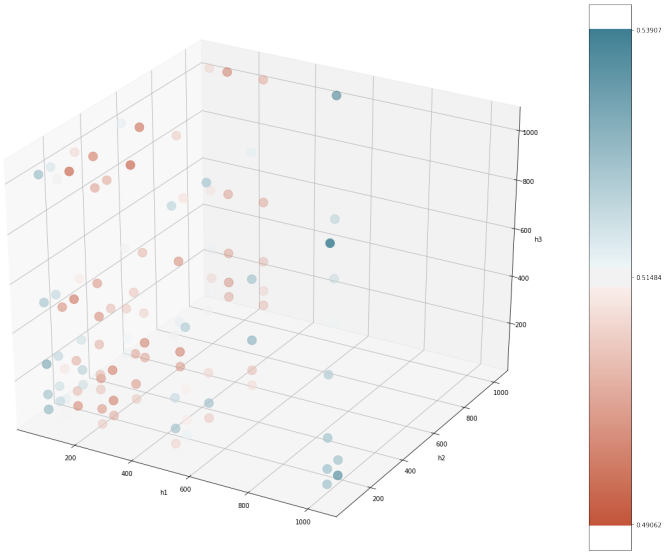

Figure 12: Visual aid used to identify good architectures for a simple 3-layer feed-forward neural network trained on the normalized log-transformed dataset. Colour corresponds to loss value after 20 epochs using SGD (redder = lower).

2.6.2 K-Nearest Neighbours

As stand-alone models KNNs did not perform very well in terms of loss; we found that adding a set of KNNs with k varying from 2 to 1024 into the stacked model led to a significant reduction in log loss. We surmise that this is because we were able to generate a large number of varied predictions this way, especially by experimenting with different distance measures, including: *manhattan*, *euclidean*, *minkowski* and *braycurtis*. We found that *braycurtis* almost always produced the strongest results both individually and as an ensemble member for this particular objective. Below we plot a correlation heatmap for the out-of-fold predictions for each KNN for varied values of k. We found this class of models to have the most intra-class variability in terms of their final predictions. This can be visualised in the below Figure 13, where we plot the correlation of each of our final KNN's predictions to one another for different values of K. We note that we primarily experimented with `sklearn`'s implementation of KNN, which was sufficiently parameterizable.

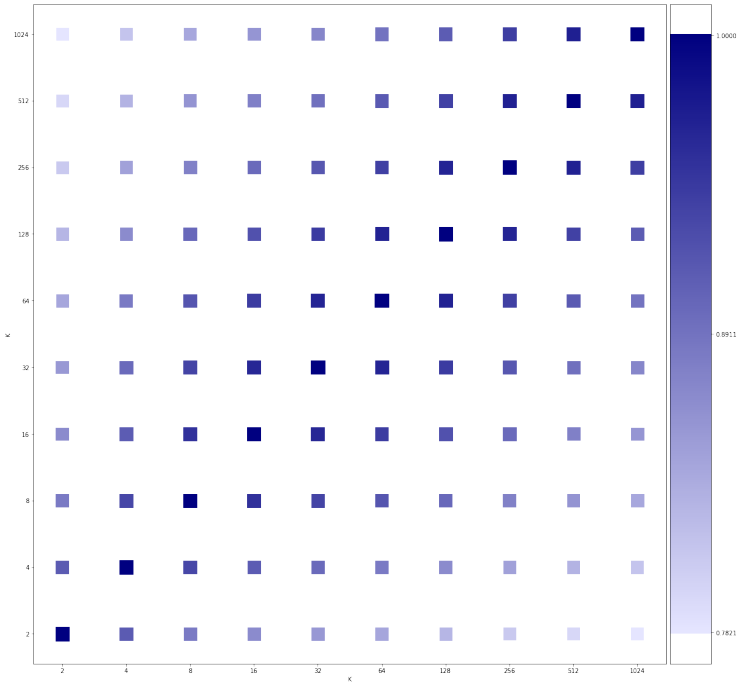

Figure 13: Correlations between the outputs of each of our KNN models. We can see significant model diversity within this class of models.

2.6.3 Gradient Boosted Decision Trees

We utilised two libraries to create our gradient boosted decision trees: *LightGBM* and *XGBoost*, and we found that both performed significantly well as stand-alone models. With some hyperparameter tuning both models were able to achieve an individual test score of below 0.45, with our best XGBoost models typically performing better than our best LGBM models.

The primary difference between XGBoost and LGBM is in how they calculate the threshold value upon which to split a feature. XGBoost uses a histogram-based algorithm, dynamically binning feature values in order to speed up the line search necessary to find the optimal split. LGBM uses a faster but less exact technique called GOSS (Gradient-based One-Side Sampling). This makes Boosted Trees much faster to train using LGBM, but this does usually lead to a decrease in predictive power. Therefore, as our XGBoosts were able to train in a reasonable amount of time on a GPU, the speed benefits of LightGBM were outweighed by the increased performance and variability of XGBoost. We therefore opted to train the majority of our individual Gradient Boosting models with XGBoost, with one LightGBM model as it did improve overall test performance. We did however opt to use an LGBM model as a second-level stacked model, taking all the other models outputs as inputs, as well as the raw training data. This was necessary due to the increased size of the data in this second-layer: XGBoost was not able to train in a reasonable amount of time in this case.

2.6.4 Bagged Decision Trees

We primarily used `RandomForestClassifier` and `ExtraTrees` from `sklearn.ensemble`. Both RandomForest and ExtraTrees perform notably well on their own, both being able to achieve loss of 0.45 and below. We additionally utilised two meta techniques on top of both types of model discussed here: namely bagging and calibration. Since these models are themselves ensembles, we observed no benefit in including multiple different versions of each in our collection of models, and instead sought to make them as predictive as possible.

Random Forests In random forests each tree in the ensemble is built from a bootstrapped sample (i.e. sampled with replacement) of the same size as the original training set. Furthermore, the best

split is calculated only on a randomly generated subset of size `max_features`. This ensures variance of the individual trees, yielding decision trees with somewhat decoupled prediction errors. The trees probabilistic predictions are averaged and Random forests overall achieve a reduced variance by combining these diverse trees, (potentially at the cost of a slight increase in bias). We also used Random Forests to calculate feature importances when selecting models to go into our initial stacked layer in a similar manner to the previous competition. Our strongest random forest model was obtained by meta-training using `sklearn`'s `BaggingClassifier` meta classifier to further increase each subforest's variance with respect to the training data. This greatly helped generalization to the unseen test set.

Extremely Randomized Trees In extremely randomized trees (i.e. ExtraTrees), splits are computed even more randomly. As in random forests, a random subset of candidate features is used, but instead of explicitly looking for the optimal threshold among these features, a threshold value is drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the new split. This greatly increases variance and reduces bias in the individual trees, leading to lower variance in the ensemble - though the tradeoff is that with weaker individual models, many more estimators are required.

In both cases, we allowed the individual trees to grow until they reached purity, or until they had only 2 samples at their leaves. More estimators typically leads to less variance (though more bias), and a particularly strong model on the validation data - however, we had to be careful to balance this in order to get good generalisation to the unseen test data. The main parameters we considered as variables were `max_features`, for the number of candidate features at each split, and `n_estimators`, which controls the number of trees to include in the model. We found that, presumably due to its increased randomness, the ExtraTrees model needed many more trees in order to be effective than the RandomForest model, but was able to achieve a lower loss before observing diminishing returns.

A Note on Calibration Since our final model score was the multi-log loss, and thus we were essentially aiming to accurately predict class probabilities, we had to be careful when using these forest models as they are not very well calibrated out-of-box. A model that predicts class probability is well calibrated if, for example, approximately 80% of the values it gives a class prediction of 0.8 for actually belong to that class. These ensembled trees generally return a biased probability estimate due to the high variance in the individual trees which are trained on random subsets of the features. Essentially, in order for a random forest to predict a probability of 0 it would require all the base-trees to predict a probability of 0 as well, which is unlikely due to the variance. Using `CalibrationClassifierCV` was key to correcting for this calibration error.

2.6.5 Final Model Stacking

As discussed earlier, the individual model classes above were investigated with the intention of training a number of strong but diverse individual models for use in a stacking framework. We experimented with using multiple layers of stacking, but eventually opted for a single layer of submodels fed into a final model that we refer to as the "stacked layer" or stacked model. We experimented with different model types here, and found that a gradient-boosted (LGBM) decision tree performed gave the best balance between reasonable training time and results on the validation and test sets. We found that additional layers/models did not increase performance sufficiently for the required increase in computation time. Our model stacking process is outlined more explicitly below.

Out-of-fold predictions We had to create the training data to input into our stacked model. To do this, we performed 5-fold cross validation on each of our individual models, and recorded their predictions on each validation fold. We aggregated these out-of-fold predictions until we have an out of fold prediction for the given model on each datapoint in the raw training dataset. These out-of-fold predictions could then be used as training data for the stacked model, since this mitigated leakage from the upstream raw training data into the final stacked model. This was absolutely necessary to avoid poor generalisation to unseen data. The predictions by individual models on the unseen data were generated by first training the individual model on all of the available training data after performing a separate cross-validation for hyperparameter tuning.

Validation Throughout, our models were validated using 5-fold cross-validation (i.e. both individual and stacked models). We found that averaging the model's validation loss across each of the 5 folds lead to a measurement that was well-correlated with a model's test loss, which saved us a lot of time submitting our predictions to Kaggle.

Model Selection After training 29 individually strong but diverse models (and discarding many more), we had to find a winning combination. The large size of the combinatoric search space, coupled with the significant training time, meant that we primarily had to rely on our intuition to find a good combination, enforcing model diversity using correlation plots such as Figure 16. We combined this manual effort with more rigorous and extensive searches to find good combinations. We give each of these final 29 individual models a unique identifier in the appendix, and note here that the final LGBM ensemble had the raw data as features, as well as the softmaxed class probabilities for each of the following models: 1, 2, 3, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28 and 29. Please see the appendix (Section 3) for explicit details on our final model. Below is Figure 14 demonstrating the graphical results of one such combinatoric search.

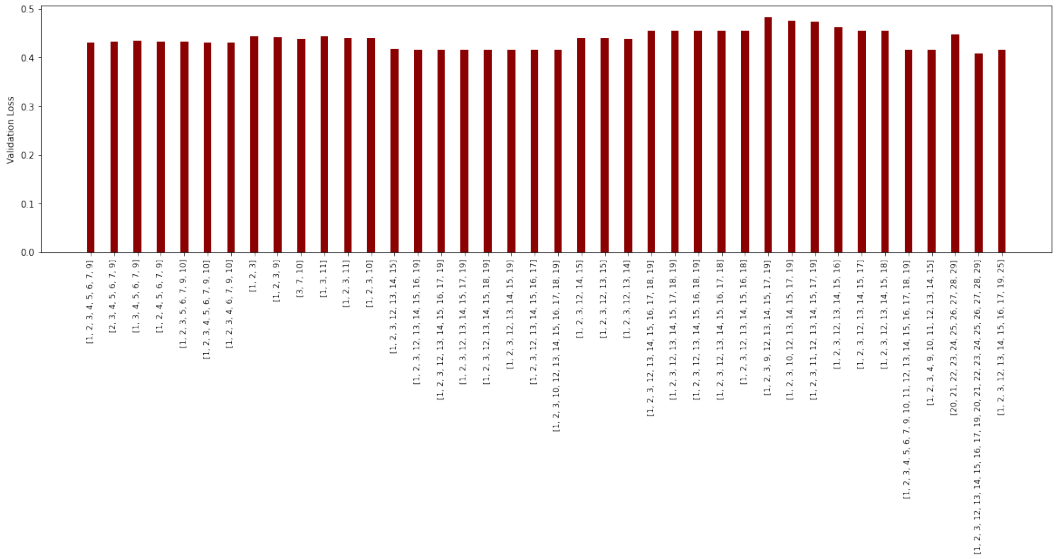

Figure 14: Different combinations of submodels final output on cross-validation.

2.6.6 Final Model Performance

The final ensemble model achieved a validation loss of 0.40753 and a test loss of 0.39693 on the Public Leaderboard, placing us in 8^{th} place. We include the final model parameters in the appendix (we refer the reader to the default parameters of the relevant libraries for unlisted parameters of interest).

3 Appendix

3.1 Source Code

The code for the Otto Competition can be found at `https://drive.google.com/drive/folders/1uFgfIwM7t8iLEZCqXfuYfFsW-JXtRw8f?usp=sharing`

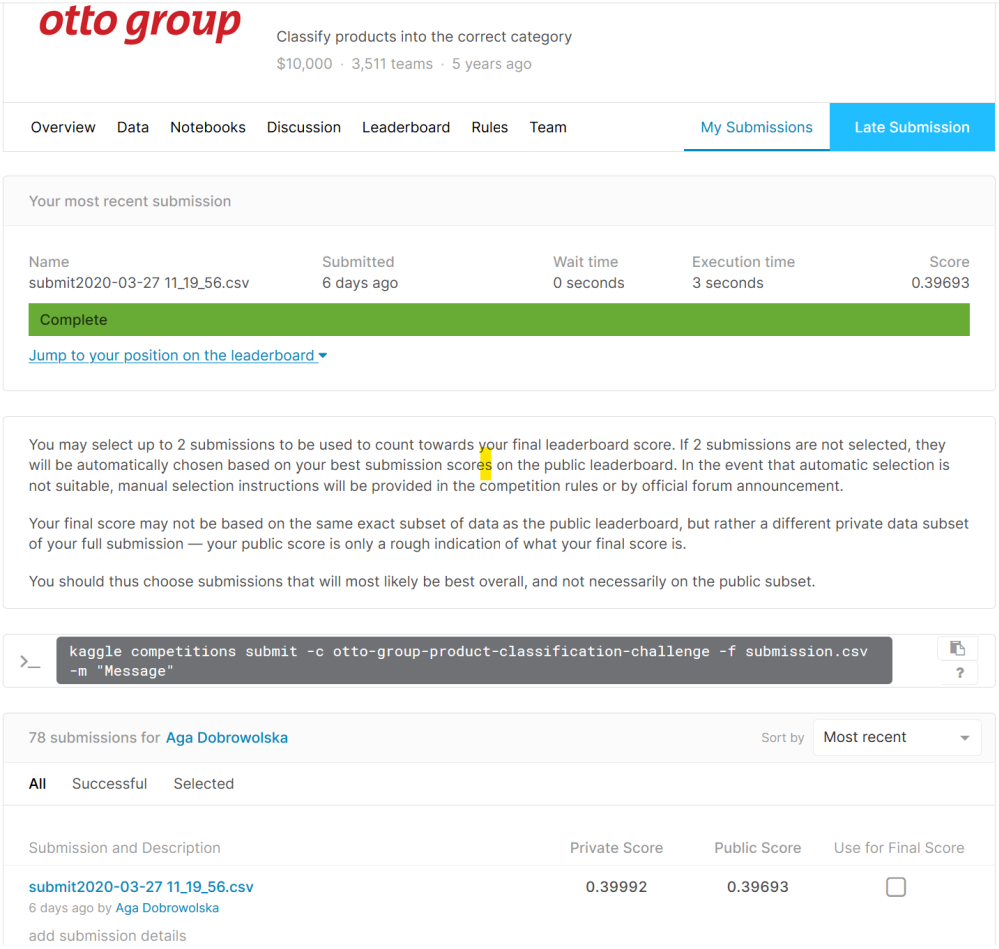

Figure 15: The performance on the test set of the final ensemble model.

3.2 Final Model Parameters

ID	Model	Data Set	Parameters
1	LGBM	Raw	Rounds=30000, Num Leaves=139, Lambda1 = 0.026, Lambda2 = 9.6
2	Neural Network, PyTorch	Log-transformed and Standardized	[512,512], bs= 64 d=0.5, SGD, lr=0.01, m = 0.9, wd=5e-4
3	KNN	Raw	Number of neighbours: 128, Distance: Minkowski
4	XGB	Raw	Rounds=2000, Max Depth = 11, Min Child Weight = 1.42
5	XGB	7 KMeans Clusters, Sum X = 0	Rounds=2000, Max Depth = 11, Min Child Weight = 1.42
6	XGB	KMeans Clusters of Log(X+1)	Rounds=2000, Max Depth = 11, Min Child Weight = 1.42
7	XGB	KMeans Clusters of MinMaxScale(X)	Rounds=2000, Max Depth = 11, Min Child Weight = 1.42
8	Neural Network, PyTorch	Log-transformed and Standardized	[128, 512, 128] bs=64 d=0.0, SGD*, lr=0.0025, m=0.9, wd=5e-4
9	Neural Network, PyTorch	Log-transformed and Standardized	[128, 1024, 128] bs=64 d=0.2, SGD*, lr=0.005, m=0.9, wd=5e-4
10	Neural Network, PyTorch	Log-transformed and Standardized	[512, 1024, 512] bs=128 d=0.4, SGD*, lr=0.005, m=0.9, wd=5e-4
11	Neural Network, PyTorch	TF-IDF Features	[2500, 1300, 40] bs=256 d=0.3, SGD*, lr=0.02, m=0.9, wd=5e-4
12	Extra Trees Classifier	TF-IDF Features	N-Estimators=1000, Max Features = 80
13	Bagged Trees Classifier	Raw	N-Estimators=100, Bagging Estimators=100
14	Calibrated Trees Classifier	Raw	N-Estimators=300
15	Bagged Random Forest	Raw	N-Estimators=40, Bagging-Estimators=100
16	XGBoost	Raw	Max Depth = 9, Min Child Weight = 0.0010, Eta = 0.125
17	XGBoost	Raw	Max Depth = 13, Min Child Weight = 0.0078, Eta = 0.0625
18	XGBoost	Raw	Max Depth = 8, Min Child Weight = 0.1, Eta = 0.011
19	XGBoost	Raw	Max Depth = 9, Min Child Weight = 0.1, Eta = 0.0113
20	KNN	Raw	Number of neighbours = 2
21	KNN	Raw	Number of neighbours = 4
22	KNN	Raw	Number of neighbours = 8
23	KNN	Raw	Number of neighbours = 16
24	KNN	Raw	Number of neighbours = 32
25	KNN	Raw	Number of neighbours = 64
26	KNN	Raw	Number of neighbours = 128
27	KNN	Raw	Number of neighbours = 256
28	KNN	Raw	Number of neighbours = 512
29	KNN	Raw	Number of neighbours = 1024

Table 5: Parameter Table for our collection of final submodels

ID	Model	Data Set	Parameters
1001	LGBM	Submodels + Raw	Rounds=30000, early_stopping_rounds = 300

Table 6: Final Single-layer Stacked Model (LGBM) parameters.

Abbreviations for Neural Net Parameters

- bs: batch size
- d: dropout probability
- lr: learning rate
- m: momentum
- wd: weight decay
- SGD*: SGD with linear learning rate decay

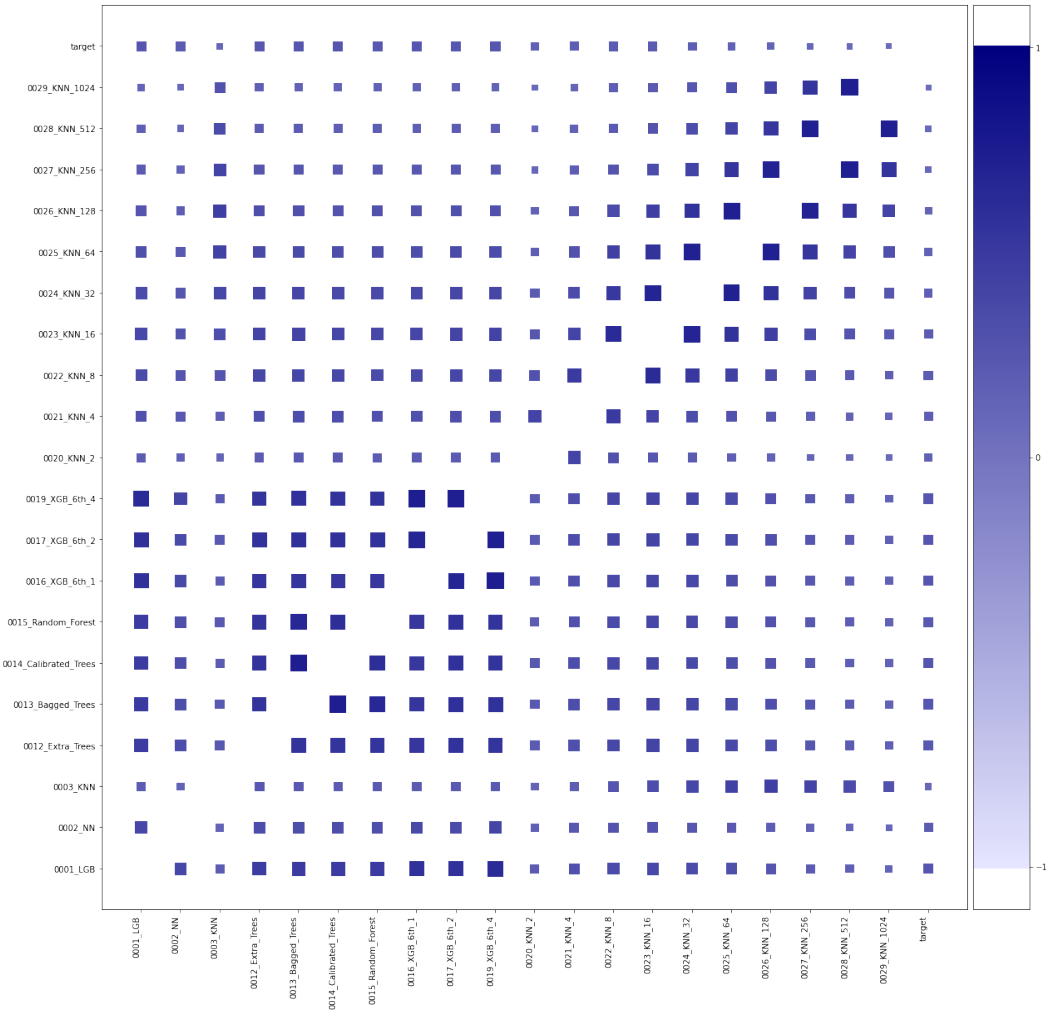

Figure 16: Correlations visualised between each model in the final ensemble, as well as the target class. Note that correlations have been raised to the power 10 in order to emphasise differences using a linear size/colour scale.