Predicting Customer Behaviour Using KDD Cup 2009 Small Dataset With A Variety Of Classifiers

Jon-Paul Boyd, Henry Lidgley
School of Computer Science and Informatics
De Montfort University
United Kingdom

***Abstract*— This report presents our approach to addressing the challenges of the KDD Cup 2009 ("small" dataset) on classifying the probability of churn, appetency and upselling of mobile phone customers. We were drawn to this challenge as it remains a very relevant classification problem in that understanding customer behavior is critical to optimization of marketing strategy with effort and budget focused on the most valuable or at-risk customers. The availability of a real-world, noisy and unbalanced dataset gave added incentive as an opportunity to learn from working through several interesting data science and machine learning technical challenges. Our first phase concentrated on data preparation with a range of methods trialed. The second phase was dominated by modelling a variety of classifiers looking to solve the problem. In the third, extensive tuning of the classifier hyperparameters sought to extract the greatest possible performance in predicting the probabilities using the AUC scoring method. Finally, we evaluated our classifier performance and compare results against the KDD Cup 2009 entries. Our gradient boosting classifier outperformed other tested models, with a final AUC average of 0.8118, placing it 34th overall.**

Keywords— Customer Relationship Management, Classification, Churn, KDD Cup, Machine Learning, Data Mining, Random Forest, Decision Trees, AdaBoost, Gradient Boosting, Bagging, Artificial Neural Networks, Keras, ReLU, Adam, Swish

I. INTRODUCTION

Customer Relationship Management (CRM) is a key element of modern marketing approaches and in a recent poll was revealed to be the most popular field in 2017 for the application of analytics, data science and machine learning [1]. It can be extremely useful for organisations to predict the likelihood of their customers to act in certain ways, such as their likelihood to switch providers. Such knowledge can then be used, for example, to identify what relevant factors may be correlated with such a prediction in order to resolve them, better target marketing campaigns, or improve the personalisation of a customer relationship [2, p. 4].

The KDD Cup 2009 customer relationship prediction challenge [3] offered the opportunity to explore a variety of machine learning classifiers using a real-world customer database made available online by the French telecom company Orange. In total, 7865 valid entries were submitted by 453 different teams. The task was to "*predict the propensity of customers to switch provider (churn), buy new products or services (appetency), or buy upgrades or add-ons proposed to them to make the sale more profitable (up-selling)*" [3]. We follow the same predictive model evaluation criteria stipulated in 2009, namely the Receiver Operating Characteristic (ROC) Area Under Curve (AUC) score, a common method to "*accurately evaluate the performance of a system that learns by being shown labelled example*" [2]. The ROC curve is a plot of the true positive rate (sensitivity) against the false positive rate (specificity) for the different possible cut points of a diagnostic test. In effect, it measures the accuracy of prediction probabilities for every possible decision threshold that can be used to round probabilities to binary predictions. AUC scores can range between 0 and 1, with the higher the value the more accurate the model's predictions. As a benchmark, the challenge included AUC scores from a basic naïve bayes classifier for competitors to beat in order to qualify for a prize [1]. The overall AUC benchmark score is 0.6711 (Churn problem: AUC = 0.6468; Appetency problem: AUC = 0.6453; Up-selling problem: AUC = 0.7211).

Our team consisted of two active, distance learning students of the Applied Computational Intelligence module within the MSc Intelligent Systems course. It was agreed to each explore the full end-to-end process of data preparation, classification model architecting and results evaluation separately, but with continuous theoretical and practical exchanges of knowledge and corroboration. Lidgley selected a Neural Network approach to the problem, whilst Boyd looked at a variety of tree-based classifiers. This "ways of working" design provided both members with the opportunity to gain individual hands-on experience of each step, yet collectively share research and provide support whilst working to own time availability. One shared understanding was splitting the data into a training set of 40000 records and test set of 10000 that facilitated a common mechanism for comparison of approaches. We also agreed on using the Python programming language and some select APIs to provide a common development environment for our own applied computational intelligence classifiers to surpass and significantly improve upon the KDD Cup 2009 benchmark.

This paper is organized as follows. Section II details initial dataset exploratory analysis. In Section III Boyd covers the approach with a focus on tree-based classifiers. In Section IV Lidgley details the neural networks solution. Section V elaborates on our team ways of working, with Section VI closing with final evaluation and thoughts.

II. DATASET

The competition organisers made two datasets available – the "large" dataset with 50000 observations and 15000 features, and the "small" dataset with the same 50000 observations but with fewer features, of which 190 are numerical and 40 categorical. We briefly considered the large dataset before erring on the side of risk mitigation and opting for the small, particularly given some of the unknowns including required

computing resources and model training time. The dataset content was scrambled, with random feature ordering and no column labels available to identify what values represent. Binary label sets for the 3 predictions for all 50000 observations were available as downloads and used as target variables to train and validate our developed binary classifier models. A label with value 1 from the churn file indicated positive churn behaviour for the observation, and -1 negative behaviour.

Initial dataset analysis highlighted some interesting characteristics. 18 features contained absolutely no data. 5 features were constant with 1 value only, and 48 columns had between 2 and 10 values. Many observations had missing feature values, for example feature "Var92" with only 169 recorded values over 50000 observations. Some features looked to be correlated, for example "Var41", "Var50" and "Var53" which are recorded with non-null values in the same 702 observations. 6 features had over 30000 distinct values. Positive target binary values were very sparse, with only 7.34% (3672) of observations labelled positively for churn, 1.78% (890) for appetency and 7.36% (3682) for upselling.

III. Tree-Based Classifier Approach

A. Aims and Objectives

Predict the 3 binary target variables for churn, appetency and upselling using the Orange small dataset, testing a variety of tree-based models that includes the standard decision tree, bagging and boosting of decision tree ensembles and finally a majority voting ensemble. The two main aims are achieving an AUC score surpassing the KDD Cup 2009 benchmark and ranking at least mid-table on "*Full Results: Slow Track*" submissions [4]. Note the organization of sub objectives as hypothesis (Hn) grouped by research questions (RQn).

1) Sub Objectives

RQ1 Feasibility Can a competitive solution be fielded by a non data scientist/statistician with no budget?

- **H1** Discover available open source classifiers with KDD Cup scoring performance "out of box"

RQ2 Performance Can tree-based classifiers predict targets?

- **H2** Determine the importance of data preparation on tree-based classifier predictive performance
- **H3** Determine if predictive performance increases with bagging and further with boosting
- **H4** Ascertain if classifier tuning leads to improved predictive performance
- **H5** Determine if a tree-based classifier AUC score average can beat the KDD Cup benchmark
- **H6** Determine if a tree-based classifier AUC score average can rank at least mid-table

B. Methodology

A quantitative after-only study design will be used to evaluate the research, with empirical measurements collected from the classifier development and test framework which constitutes the main research instrument. All observations will be quantitatively measured by ratio scale in the form of AUC scores for each of the 6 concepts outlined in "Sub Objectives".

RQ1 H1 will be measured by ranking the baseline AUC scores of classifiers against KDD Cup benchmark and existing KDD Cup submissions "*Full Results: Slow Track*" [4]. Baseline means classifier with full default settings (no tuning applied).

RQ2 H2 is measured by comparing classifier baseline AUC scores when a variety of data preparation methods are applied.

RQ2 H3 will compare the predictive performance of each classifier to determine if "*Boosting > Bagging > Single Tree*" as stated by Leo Breiman still holds true (and quoted by Trevor Hastie at a Stanford University talk in 2003 [5]).

RQ2 H4 will compare the baseline AUC scoring of each classifier with corresponding final score after hyperparameter tuning applied.

RQ2 H5 will be measured by ranking the AUC scores of hyperparameter tuned classifiers against KDD benchmarks.

RQ2 H6 will be measured by ranking the AUC scores of hyperparameter tuned classifiers against existing KDD submissions "*Full Results: Slow Track*" [4].

C. Software and Hardware Components

The following software and hardware component specification was used throughout the exploration of this tree-based machine learning challenge:

Software Components - Windows 10 Pro, Python 3.6, PyCharm Professional 2018.1

Hardware Components - Intel i7-5820K 4.6Ghz, 16GB RAM

Main Python Modules – Scikit-learn (sklearn), a data mining module that includes collection of classifiers, pandas v0.22.0 (data structure and analysis toolkit)

Python was selected as programming language for the solution given its prevalence in machine learning and other AI fields, with this view supported by the authors [6, p. 1] "*The Python programming language is establishing itself as one of the most popular languages for scientific computing. Thanks to its high-level interactive nature and its maturing ecosystem of scientific libraries, it is an appealing choice for algorithmic development and exploratory data analysis. Yet, as a general-purpose language, it is increasingly used not only in academic settings but also in industry*". It is a sensible choice given further hands-on exposure will return carry-over value into professional work within business analytics. In 2017 IEEE.org ranked Python number 1 in popularity from over several dozen programming languages [7].

Scikit-learn is a well-established datamining toolkit module for Python, with a "*rich environment to provide state-of-the-art implementations of many well known machine learning algorithms, while maintaining an easy-to-use interface tightly integrated with the Python language*" [6, p. 1], a broad variety of text book references and a very active developer community over at Stackoverflow [8].

D. Solution Key Features

The goal of exploring the development, training, tuning, scoring and comparing of several tree-based classifiers predicting 3 different target variables necessitated an elegant software design with the following key features:

Multi classifier analysis with decision tree, random forest, gradient boosting, AdaBoost, bagging and majority voting.

Highly configurable runtime environment facilitating any combination of data preparation profile, target variable scoring, baseline/final classifier scoring and hyperparameter tuning tasks to be executed. Maintaining the runtime arguments allows only baseline scoring (default hyperparameter setting) or only final scoring (tuned models) to be executed, or alternatively only execution of grid search and scoring of multi-hyperparameter permutations, for any combination of target variable and classifier. This was key given the collection of models explored and extended runtimes for training, allowing focused, efficient analysis especially during tuning and retesting.

Multiple dataset preparation profiles aid exploration of various data transformation approaches and their effect on model predictive performance.

Visualization built in for graphical illustration of dataset features, data preparation methods and classifier scoring.

Extensive file system logging of classifier feature importance, baseline scoring, final tuned model scoring and hyperparameter tuning results, aiding detailed further analysis and reporting.

E. Key Custom Modules

Provided below is an outline of the five custom-developed modules, in total comprising ~1200 loc (lines of code).

The main module responsible for orchestrating the overall solution according to runtime environment settings.

The file handler module takes care of saving/loading of datasets and target variable files, in addition to output of scoring and feature importance data.

The pre-processor module is responsible for data wrangling, with imputation of null values, feature dropping, label encoding and one hot encoding.

The modeler module scores the baseline and final variant of each classifier. The module includes hyperparameter scoring on an individual and combination parameter basis.

The visualizer module graphs key stages within the classification process that includes dataset feature nullity, feature correlation and classifier scoring.

F. Classifiers

It was appropriate to research a variety of tree-based classifiers as this algorithm "family" is known to effectively handle datasets with a large mix of numerical, categorical and redundant values. Ensembles of trees have high predictive power and are resistant to over-fitting. According to [9, p. 1345], "*It is well known that trees and especially ensembles of trees can provide robust and accurate models in "real-life" data settings. They handle mixed and noisy data, and are scale insensitive*". All selected tree-based classifiers in this study leverage supervised learning by looking for patterns between the many independent features (numerical and categorical) and the target dependant variable (churn, appetency, upselling) during a training phase, using a specifically allocated partition of the dataset. This is known as model fitting, with the model built and ready for the evaluation step, which involves scoring it on a sample of previously unseen test data where correct target variable labels are known and thus used in assessing the predictive power of the model. What follows is a brief overview of each tree-based classifier explored. The technical implementation of each of classifier is sourced from the well-known python module Scikit-learn (sklearn) [8].

The first classifier tested is the classical decision tree (DTC), which takes a divide and conquer approach to a given classification problem, starting at the root node of a tree with the whole dataset and splitting it left and right into lower nodes according to questions asked. This process of question and split continues until the predicted class for each observation is pure (of the same value) or until some configured threshold such as tree depth is reached. Visualizing the end result will show a tree-like structure with a main root, decision nodes (where a question is asked of the observation subset passed to it) and terminal nodes containing class predictions made for each observation. There are a variety of decision tree algorithms such as C4.5 [10] and CART (Classification and Regression Trees) [11]. The Scikit-learn module uses an optimized version of the CART algorithm [12]. [13, p. 121] confirms the decision tree is a satisfactory starting point, with "*Decision-tree learning has been extensively used in many application areas of machine learning, especially for classification, because the algorithms developed for learning decision trees represent a good compromise between comprehensibility, accuracy and efficiency*". Fig. 1 shows sample architecture of a decision tree from this study solution predicting churn, with the data being split based upon a series of questions.

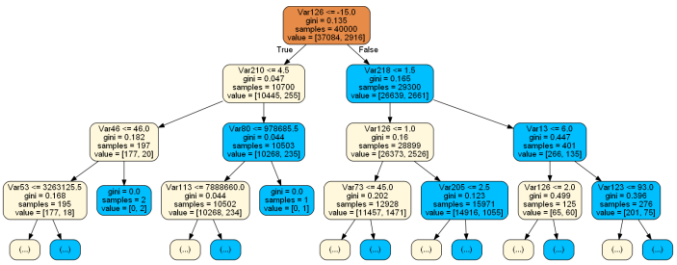


Fig. 1. Example implemented decision tree predicting churn, to depth of 3

To illustrate the complexity of the dataset in terms of number of features and data variance, Fig. 2 shows the scale and breadth of the full tree from which snapshot in Fig. 1 was taken.

Fig. 2. Scale of full decision tree predicting churn, each dot a tree node

It was appropriate to then review performance of the random forest classifier (RFC) [14], as it improves upon standard decision tree performance by bagging samples of data and averaging the output of many trees. The algorithm was first introduced by Tin Kam Ho [15] in 1998, with the phrase "*random forest*" and concept of bagging elements for random selection of observations and features reported by Leo Breiman [16]. It is a supervised learning method that uses decision tree classifiers (DTC) as "*weak*" base learners. "*Forest*" comes from the algorithm's use of multiple DTC, and "random" as each DTC in the ensemble works on a random subset of both the dataset observations and features in order to make a class prediction for its sampled observations. Aggregation of each

decision tree observation binary classification determines the final prediction, known as majority voting. [9, p. 1346] claims "*Random forest (RF) is an exemplar for parallel ensembles (Breiman, 2001). It is an improved bagging method (Breiman, 1996) that extends the "random subspace" method (Ho, 1998). It grows a forest of random decision trees on bagged samples showing excellent results comparable with the best known classifiers*".

An alternative to the RFC is sklearn's bagging classifier (BGC) [17], another ensemble method randomly sampling data into multiple weak base learners, in this case the decision tree, as it makes for relevant comparison with RFC performance.

The AdaBoost, or adaptive boosting classifier (ABC) [18] introduced by Freund and Schapire in 1995 "*calls a given weak or base learning algorithm repeatedly in a series of rounds. Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased so that the weak learner is forced to focus on the hard examples in the training set*" [19, p. 2]. According to [9, p. 1346], "*Boosting showed dramatic improvement in accuracy even with very weak base learners (like decision stumps, single split trees)"*. Hyperparameter tuned CART decision trees are used as base weak learners, given that DTC is the default estimator for ABC and again provides for relevant comparison with RFC and BGC.

The Gradient Boosting Classifier (GBC) is an algorithm utilizing an ensemble of regression trees [20]. Like RFC the predicative classification probability from each tree is averaged but boosting comes from a method that re-weights at each step in order to resolve deficiencies from prediction errors in previous tree steps, and hence the ensemble is grown in an adaptive fashion. Gradient boosting is described by Friedman as "*competitive, highly robust, interpretable procedures for both regression and classification, especially appropriate for mining less than clean data*" [21, p. 1].

Given IBM, winners of the KDD Cup 2009, used an ensemble classifier comprising a range of learning algorithms that satisfied "*learning algorithms that were efficient enough to handle a data set of this size in the allotted time, while still producing high performing models*" [22, p. 25], a simple ensemble of 3 classifiers - random forest, bagging, and gradient boosting – was assembled to test whether the committee of 3 base yet powerful models combining their predictive output through majority voting could outperform a single classifier alone. The Voting Classifier (VTC) [23] was used to build the ensemble.

G. Data Preparation

Initial analysis was done to understand the quality of information content and consider options to deal with what is obviously an imbalanced dataset. As a short reminder, only 7% of observations labelled positive for churn, 1.8% for appetency and 7.4% for upselling. Acknowledging "*Class imbalance plays a major role in affecting the reliability of a classifier. The major issue existing due to class imbalance is that the minority class is not well represented and hence the classifier is undertrained on the minority classes*" [24, p. 216] several further modelling strategies including under sampling, oversampling and weighting are tested as detailed in sub-sections H and I.

5 features hold constant value and are therefore redundant. The number of unique categorical feature values ranges from 1 to 15413 with mean of 10, standard deviation of 4150 and positive skewness of 2.56. 18 features have no data. 150 of the 230 total features had 95% or greater observations with no data.

The nullity matrix in Fig. 3 shows a high-level overview of the data density across all 50000 observations and clearly illustrates features like Var1 in the leftmost column of the chart have very little recorded data as represented by a majority of whitespace. The matrix also illustrates patterns in the dataset, for example what appears to be correlated variables in Var21 and Var24 where blocks of black (data) and white (no data) are comparable.

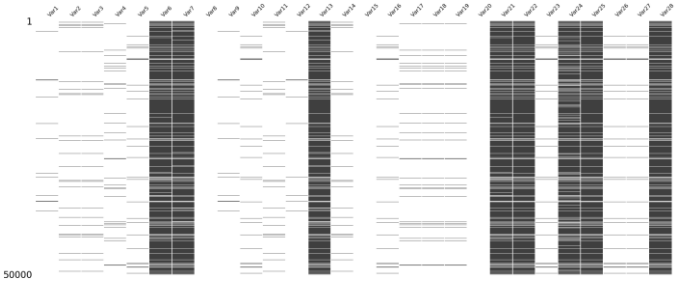


Fig. 3. Nullity matrix of first 28 numerical features

Checking availability of feature data by observation with counts indicates significant missing data. Fig. 4 shows Var1 provisions data for only 702 of the total 50000 observations. As the data was scrambled, randomized and provided without feature labels there was no opportunity to leverage any level of domain expertise or insight when attempting to understand the dataset further. It is not possible to determine for example whether groups of features relate specifically to a mobile phone or internet subscription, and therefore whether a customer has 1 or more subscription "packages" with Orange. However, given the patterns and data correlations observed this data seems not missing at random, but rather systematically not present. This can be corroborated by [24, p. 215] "*due to the requirement of structural representation of the data, all the instances are bound to contain all the properties corresponding to a generic customer in the organization. This leads to data sparseness, since customers will be associated with only a few properties and not all the properties pertaining to the organization. The hugeness of data and sparsity acts as the major difficulties in the process of churn prediction*".

Fig. 4. Nullity comparison count by colmn for first 28 numerical features

Exploring feature correlation further, a subset of numeric feature nullity correlations is given on the heatmap in Fig. 5. The nullity correlation ranges as follows: -1 (one variable has value, other does not), 0 (no correlation between nullity of variables, orange blocks), 1 (both variables have value, dark blue blocks).

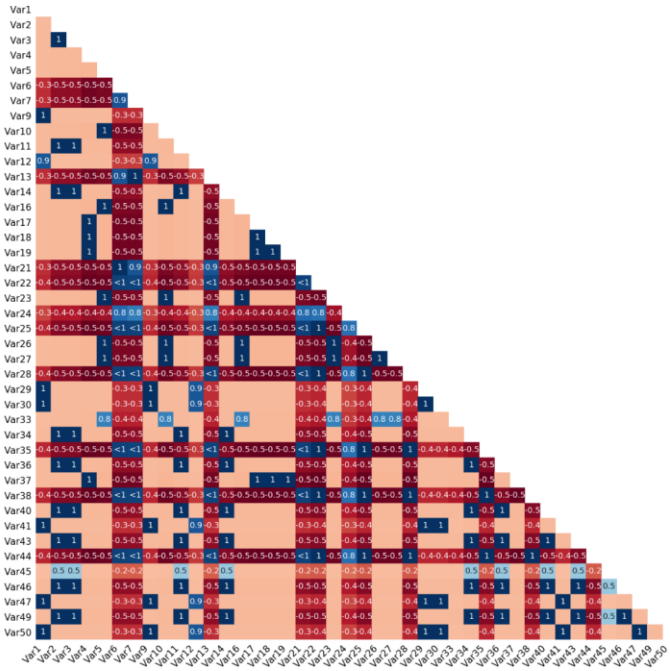


Fig. 5. Nullity comparison count by colmn for first 28 numerical features

Features that are fully devoid of value or fully represented throughout all observations are excluded, as the focus is on nullity comparison. For example, Var8 is removed as it contains null for all 50000 observations. The heatmap clearly tells us that where we have a value in Var1 there is always a value in Var9, Var29, Var30, Var41 and Var47, suggesting some form of grouping of data features, perhaps an address, subscription package detail or customer event, but currently unknown as unscrambling will not be performed. Taking one example correlation "<1" between Var6 and Var44, these features very closely correlate excepting a few observations where Var6 is null and Var44 has value. This may indicate same grouping or event indicator commonality, but again without availability of labels for feature description the relationship is not yet understood. At present this correlation insight has not been leveraged, but future work could use this information to identify redundant features and aid simplicity and runtime performance of the model or cross-referenced with feature importance weightings to analyse how closely features work together predictively.

After initial bare minimum cleansing (resolving null values) to validate the end-to-end solution framework design and feasibility in making sensible AUC predictions comparable with the KDD benchmark, three data preparation approaches were finalised, as summarized in Table I. All approaches included label encoding (transforming categorical feature values from string to numeric value in range 0 n-1 classes) with sklearn's LabelEncoder method, as the random forest classifier is unable to handle nominal values.

Several methods are available for imputing null numeric features, including taking the mean or median. 3 variations of numeric imputation were tested, setting null to zero, an arbitrary negative "-9876" value to preserve the missing "signal", or the mean of the feature, following the recommendation "*In the case of normal distribution, the sample mean provides an optimal estimate of the most probable value*" [25, p. 1002]. There still however remains a concern that introducing synthetic values to an already noisy and imbalanced dataset may dampen valuable signals. As [26] states, "*The standard errors and test statistics can also be underestimated and overestimated respectively. This imputation technique works well with when the values are missing completely at random*". However, as the nullity matrix and heatmap visualisations have suggested, all data missing due to randomness is believed unlikely.

TABLE I
DATASET PREPARATION STRATEGIES

Action	**Dataset**		
	DS01	***DS02***	***DS03***
Drop Numeric Features	Constant	Constant	Constant
Drop Categorical Features	Constant	Constant	Constant > 10 unique
Impute Numeric	Null by mean	Null by 0	Null by -9876
Impute Categorical	Null by "missing"	Null by "missing"	Null by "missing"
Label Encode	Yes	Yes	Yes
1 Hot Encode	No	No	Yes
Standard Scaling	Yes	No	No
Numerical Features Removed	Var8, Var15, Var20, Var31, Var32, Var39, Var42, Var48, Var52, Var55, Var79, Var141, Var167, Var169, Var175, Var185	Var8, Var15, Var20, Var31, Var32, Var39, Var42, Var48, Var52, Var55, Var79, Var141, Var167, Var169, Var175, Var185	Var8, Var15, Var20, Var31, Var32, Var39, Var42, Var48, Var52, Var55, Var79, Var141, Var167, Var169, Var175, Var185
Categorical Features Removed	Var209, Var230	Var209, Var230	Var209, Var230, Var192, Var193, Var195, Var197, Var198, Var199, Var200, Var202, Var204, Var206, Var207, Var212, Var214, Var216, Var217, Var219, Var220, Var222, Var226, Var228
Remaining Feature Count	212	212	245

Where stated categorical features were one-hot encoded with sklearn's OneHotEncoder method. This generates additional features with a binary value for each original categorical feature value. Standard normal distribution scaling is performed with sklearn's StandardScaler.

Note DS03 removes features with greater than 10 unique values before one hot encoding, and hence why, despite removing more features than other strategies, results in a final dataset with a greater number of features. The threshold of 10 was purposely set as many categorical features had a high number of distinct values, for example Var214 with 15412, and attempts to one hot encode caused memory issues due to the explosion in dimensionality with number of additional features created.

These 3 strategies were arrived at after exhaustive preparation permutation testing, including incremental setting of feature value thresholds, then scored with baseline classifiers to benchmark them. The custom pre-processor module handles all data transformation and the finalised dataset output to the file system for later processing by the main and modeller modules.

Table II summarizes classifier comparative performance in predicting the 3 target variables using their baseline (default) configuration consuming each dataset variant. It is useful to present baseline scores here to observe the impact on AUC score by data preparation strategy. The dataset consumed when

achieving the highest target variable AUC score for each given classifier is highlighted in bold. The "Diff" column shows the increase in best score over second best.

TABLE II
BASELINE CLASSIFIER INDIVIDUAL TASK PERFORMANCE ACROSS DATASETS

Task	Classifier	Dataset			
		DS01	**DS02**	**DS03**	**Diff**
Churn	Decision Tree	0.5381	0.5322	**0.5449**	0.0068
	Random Forest	0.6074	**0.6133**	0.6060	0.0059
	Bagging	0.6270	0.6167	**0.6436**	0.0166
	AdaBoost	0.6475	0.6206	**0.6553**	0.0078
	Gradient Boosting	0.7368	0.7373	**0.7412**	0.0039
	Voting	0.6890	**0.6943**	0.6875	0.0053
Appetency	Decision Tree	0.5298	0.5132	**0.5337**	0.0039
	Random Forest	**0.6182**	0.5947	0.5967	0.0215
	Bagging	**0.6538**	0.6421	0.6201	0.0117
	AdaBoost	**0.6816**	0.6699	0.6309	0.0117
	Gradient Boosting	0.8110	0.8125	**0.8237**	0.0112
	Voting	0.7808	0.7808	**0.7910**	0.0102
Upselling	Decision Tree	0.6870	0.6914	**0.6934**	0.0020
	Random Forest	0.7773	**0.7812**	0.7529	0.0039
	Bagging	0.7964	0.8013	**0.8052**	0.0039
	AdaBoost	0.8101	**0.8228**	0.8208	0.0020
	Gradient Boosting	0.8716	**0.8726**	0.8687	0.0010
	Voting	0.8345	**0.8379**	**0.8379**	0.0034
Dataset Win Count		3	6	10	

Overall performance advantage of any given dataset preparation strategy is marginal, especially in prediction of upselling (max 0.0039) where the target label for this behaviour is less sparse than it is for appetency which sees a max of 0.0215. DS03 records most "wins" with 10. Reviewing scores across the entire suite of baseline classifiers suggest DS03 generally lends itself to the churn (4/6) and tied with DS01 on appetency (3/6). DS02 does best for upselling (4/6). DS01 seems only to be effective in the appetency task and interestingly where the greatest performance margins are shown, and the only strategy to impute numerics by mean and scale value distribution.

DTC responds best to DS03 for all 3 tasks which could suggest the effectiveness of preserving the numeric null signal by imputation with arbitrary "-9876". RFC has preference for DS02 (2/3), BGC for DS03 (2/3), ABC is split (1/1/1/), GBC best with DS03 (2/3) and VTC for DS03 (2/3). Interestingly, despite DS01 only winning 3 individual classifier tasks the data suggests this strategy performs better than DS02 as a single overall data preparation approach for all classifiers in predicting the 3 behaviours, as illustrated in Table III.

As expected, the ensemble classifiers outperform the stand-alone CART decision tree in all predictive tasks using any dataset by quite a margin and therefore worth any additional computational cost. BGC always achieves better performance than RFC, with boosting improving on bagging, indicating RQ2 H3 holds true. Unfortunately, VTC as an ensemble of RFC, BGC and GBC classifiers performs poorer than expected, suggesting the majority voting mechanism allows untuned, default RFC and BGC classifiers to outvote the effective GBC with inaccurate voting predictions resulting in poorer AUC score. A later review will determine if the same holds true after analysis of scores from tuned models.

Note that DS03 is the only strategy to employ one hot encoding. It is not clear whether being constrained in only being able to one hot encode features with less than 10 unique values (due to high dimensionality causing memory issues) is negatively impacting this approach with some predictive signals being lost. Whilst the preference is less constraint in number of features hot encoded it is suggested that hot encoding categorical features is not critically important for classification problems addressed by tree-based algorithms where any feature value is a question without linear consideration and therefore not influenced by outlier values.

The objective of RQ2 H5 is improving upon the KDD benchmark - churn 0.646, appetency 0.645 and upselling 0.721. At this point only considering baseline configuration, DTC fails the benchmark for all 3 tasks. RFC only surpasses the benchmark in the upselling task whilst BGC improves on benchmark in appetency and upselling. ABC, GBC and VTC beat benchmarks for all tasks with baseline configuration.

TABLE III
BASELINE CLASSIFIER AVG AUC PERFORMANCE ACROSS DATASETS

Classifier	DS01	DS02	DS03	KDD Cup Rank
Decision Tree Avg	0.5849	0.5789	**0.5906**	84
Random Forest Avg	**0.6676**	0.6630	0.6518	82
Bagging Avg	**0.6924**	0.6867	0.6896	79
AdaBoost Avg	**0.7130**	0.7044	0.7023	78
Gradient Boosting Avg	0.8064	0.8074	**0.8112**	**34**
Voting Avg	0.7681	0.7710	**0.7721**	71

Table III presents average AUC scores for each baseline classifier with KDD rank. DTC performs poorly in untuned form. There is a significant performance divide between GBC and all other classifiers, with default baseline configuration satisfying both RQ1 H1 "*scoring performance out of box*" and RQ2 H6 "*rank at least mid-table*".

H. Sampling

Given the dataset is extremely unbalanced with all 3 positive behaviours in the minority class (churn 7.34%, appetency 1.78%, upselling 7.36%), experimentations on oversampling the minority class and under sampling the majority class in observations within the training set were done using the imbalanced-learn python package [27] with a tuned variant of the RFC predicting churn with dataset strategy DS02.

Unfortunately, only performance degradation compared with a "no sampling" test was observed, as highlighted in Table IV. ROS (Random Over Sampling) in generating additional duplicate minority class samples performed better than more complex oversampling methods but outperformed by RUS (Random Under Sampling) which randomly removes majority class samples.

TABLE IV
COMPARISON OF SAMPLING METHODS

Category	Method	Score
No sampling	Tuned RFC	0.7286
Over sampling	SMOTETomek	0.6527
Over sampling	SMOTE	0.6527
Over sampling	ADASYN	0.6554
Over sampling	ROS	0.6896
Under sampling	RUS	0.6912

These results are disappointing and lead to agreement with [28, p. 63] that this KDD Cup challenge is a difficult one, making the right data preparation strategy decision paramount, as they comment "*On the one hand, we would like to deal with balanced data. On the other hand, we wish to exploit all available information.*".

I. Weighting

A further attempt to address the dataset imbalance used RFC hyperparameter *class_weight*, which allows additional emphasis to be put on a class and penalizes the model for misclassification during training. A range of weighting schemes were tested using a tuned RFC consuming dataset DS02. The "balanced" scheme computes weighting inversely proportional to class frequencies in the training data, with "balanced_subsample" like "balanced" except weights are computed based on the bootstrap sample for every tree grown [14]. The scheme with 0.67340 and 0.5401 uses weighting calculated by sklearn method compute_class_weight. The other 3 schemes are arbitrary. Sadly, the weighted models suffer significant AUC score performance degradation compared with the non-weighted model, as shown in Table V.

TABLE V
COMPARISON OF WEIGHTING METHODS WITH RFC

Minority Class Weight	Majority Class Weight	Score
N/A	N/A	0.7286
balanced		0.6968
balanced Subsample		0.6969
2	1	0.7020
5	1	0.7021
10	1	0.6991
6.7340	0.5401	0.6968

J. Classification Thresholds

In Scikit-learn the threshold is set to 0.5 for positive binary classification. When predicting churn an observation class probability of less than 0.5 is classed as "no churn" with binary value -1, a probability 0.5 and greater classed as "churn" with binary value 1. Fig. 6 plots the confusion matrix, a method to represent the classification types and error rate of the model, when the threshold for positive churn uses default 0.5 and then adjusted to 0.29.

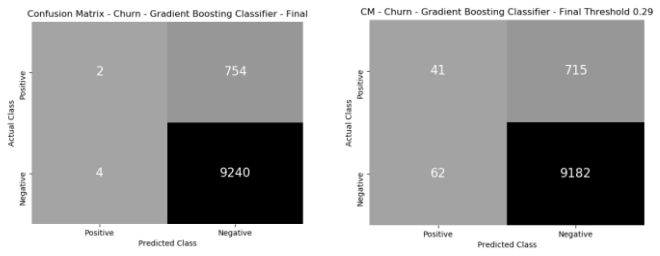


Fig. 6. Confusion matrix for GBC prediciting churn, threshold 0.5 and 0.29

The top left cell represents true positives, observations correctly classified as "churn". With the default threshold of 0.5, a disappointing 2 observations are correctly classified as positive churn. In real-world scenarios these are the classifications we are truly interested in, as they can directly support actionable subscriber communication in trying to prevent churn. This view is supported by [29, p. 935], with "*In binary classification, it is typically the minority (positive) class that the practitioner is interested in. Imbalance in the class distribution often causes machine learning algorithms to perform poorly on the minority class. In addition, the cost of misclassifying the minority class is usually much higher than the cost of other misclassifications.*".

The number of true positive classifications can be increased by adjusting the positive threshold, as evident in Table VI, yet this comes at varying cost, especially to false positives which directly impacts the AUC score. As the objectives of this study are based on highest achievable AUC score the threshold remains at default 0.5. However, in a real-world scenario the ultimate goal would more likely be best possible identification of positive churners, therefore serious consideration would be given to optimisation of classification threshold value to facilitate this.

TABLE VI
THRESHOLD WITH CONFUSION MATRIX AND SCORE

Threshold	True Neg	False Neg	True Pos	False Pos	Score
0.09	7416	351	405	1828	0.6689
0.19	8915	614	142	329	0.5761
0.29	9182	715	41	62	0.5237

K. Feature Selection

Feature ranking is a widely used method to select a subset of the overall features available for the purposes of increasing generalization (avoiding overfitting) and reducing dimensionality. An added benefit is reduced model complexity and training times. Feature importance can be extracted from an sklearn instanced classifier to review the ranked importance of each feature to a classification and use this ranking to control which features are used in the model prediction. Summing importance for all features results in value of 1.

Fig. 7 graphs the scored contribution of the top 10 features (from total of 213) when predicting churn, appetency and upselling using RFC. Var113 dominates all 3 customer behaviours with an importance of 0.337 (churn), 0.610 (appetency) and 0.738 (upselling). The graph suggests the first 7 features provide predictive value to the random forest classification model and from there the remaining features contribute very little predictive information.

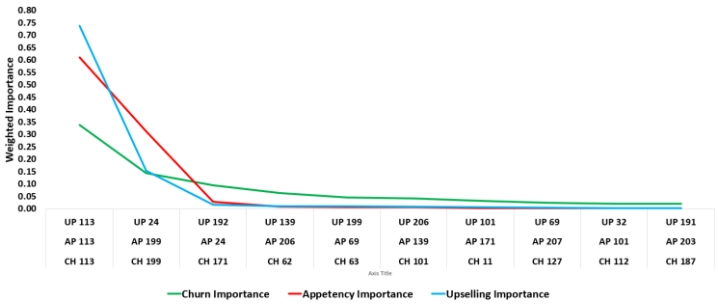


Fig. 7. Top 10 random forest feature importance

Table VII presents combined importance for a range of groupings. The top 20 features account for 0.9 importance for churn, 0.99 for appetency and 0.97 for upselling. A combined 50 features ranked 31-80 provide very little especially for appetency and upselling, demonstrating the noise of this dataset.

TABLE VII
COMBINED FEATURE IMPORTANCE WEIGHTINGS

Features	Churn	Appetency	Upselling
Top 10	0.822698	0.981884	0.954647
Top 20	0.900744	0.991004	0.9715
Top 30	0.94755	0.995846	0.982366
31-80	0.051049	0.004125	0.016367
Features with 0 importance	96	145	101

Significant numbers of features provide no predictive value to the RFC. Experimentation in configurating the classification model to only use the top 20 features could not yield improved predictive performance in AUC scoring. This does raise the question again whether some feature signals are not sufficiently weighted, amplified or binned in the data preparation phase and information is being lost. Indeed, another KDD Cup entrant comments on weighting, confirming that "*In order to handle this imbalance problem we tried a few initial weighting scheme*" and "*We found that the best-performing weighting scheme was when both classes received half of the total weight, which meant that the instances from the positive class had higher initial weights than the instances form the negative class.*" [30, p. 117].

L. Classification Model Design, Build & Tune

The main.py python module orchestrates the solution with the following technical steps examined in detail.

1) Map technical runtime environment

30 configurable system parameters direct what tasks are performed, from determining dataset transformation strategy to indicating what behavioural predications are required, if hyperparameter tuning is to be run and whether baseline or final scores are needed. This configuration comes into the main.py module as system argument variables referenced by integer index. To simplify the program flow that heavily references this configuration the technical system arguments are mapped to a more verbose form via a Python dictionary with access using key name. The code snippet gives example of dictionary definition:

```
envparm = {'PrepEnabled':False, 'ProcessDS01':False,
```

With mapping of technical system argument to dictionary

```
if sys.argv[1] == 'PrepEnabled=1':
   envparm['PrepEnabled']= True
```

And access to the verbose environment parameter

```
if envparm['PrepEnabled']: Preprocessor(envparm)
```

Given the complexity and configurability of the runtime environment, and accounting for lengthy runtimes especially in combined hyperparameter tuning, a comprehensive console log is maintained which documents each execution step, making execution progress transparent.

2) Preprocessor instantiation

If configured via system environment variable 'PrepEnabled=1', instantiates the preprocessor class which transforms the original Orange dataset and outputs a prepared .csv file. The transformation strategy determines what imputation is done and whether one hot encoding is performed. The strategy is indicated by environment variables 'ProcessDS01', 'ProcessDS02' and 'ProcessDS03'. Configuration with 'PrepEnabled=0' allows bypassing of the data transformation step, used when satisfied with a previously prepared dataset file.

3) Load prepared dataset and target labels

Loads the last prepared dataset according to required dataset strategy, with individual file paths mapped to environment variables 'ProcessDS01', 'ProcessDS02' and 'ProcessDS03'. The target binary labels for churn, appetency and upselling are also uploaded, dependent on variables ''PredictChurn', 'PredictAppetency' and 'PredictUpselling'.

4) Dataset split into training and test

Program flow is now transferred to the modeller.py module. Using sklearn's model_selection.train_test_split class the previously prepared and uploaded dataset and target label (churn, appetency or upselling) are divided into training and test sets. Team discussion agreed to randomly partition out the dataset into 80% training and 20% test.

```
dataset_train, dataset_test, target_train,
target_test = train_test_split(dataset,
target.values.ravel(), test_size=0.2)
```

5) Score baseline classifiers

It is beneficial to score each implemented classifier on the prediction tasks using default settings. This provides an immediately available understanding of "out-of-box" performance regardless of classification task and will be later used in assessing how sensitive a given classifier is to tuning. Before any model implementation the classifier definitions are loaded from sklearn.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
```

The example below illustrates the method definition for scoring the baseline random forest classifier (RFC).

```
def score_baseline_rtc(self, filehandler):
  clf = RandomForestClassifier()
  clf.fit(dataset_train, target_train)
  predict = clf.predict_proba(dataset_test)[:, 1]
  score_auc = roc_auc_score(target_test, predict)
```

An initial instance of the classifier class is instantiated into clf. The model is trained by fitting the training set and associated target training labels. The prediction is then made, with the AUC score calculated using the actual known target labels from the holdout test set against the predicted labels for the test set. The score is logged both in a Python list for later output to the file system and in the Python console. As previously noted, in Scikit-learn the threshold is set to 0.5 for positive binary classification. Fig. 8 shows observation 158 negatively classified, as it's probability of churn is calculated at 0.2, whereas observation 163 is positively classed as it's probability of 0.6 crosses the threshold.

Reconsidering the real-use case example of a directed marketing campaign with a limited budget, you may have several hundred positively classified customers but budget sufficient only for contacting a subset. The probability value can be used to prioritize and focus a customer relationship exercise to only the top-n probabilities, often more useful than binary values in real world applications.

This sentiment is shared by [3, p. 2] with "*The most practical way to build knowledge on customers in a CRM system is to produce Scores*" and "*Scores are then used by the information system (IS), for example, to personalize the customer relationship. The rapid and robust detection of the most predictive variables can be a key factor in a marketing application*". This is further supported by [31, p. 532] that comment "*For such an application, the goal is not only to predict whether or not a subscriber would switch from one carrier to another, it is also important that the likelihood of the subscriber's doing so be predicted. The reason for this is that a carrier can then choose to provide special personalized offer and services to those subscribers who are predicted with higher likelihood to churn.*".

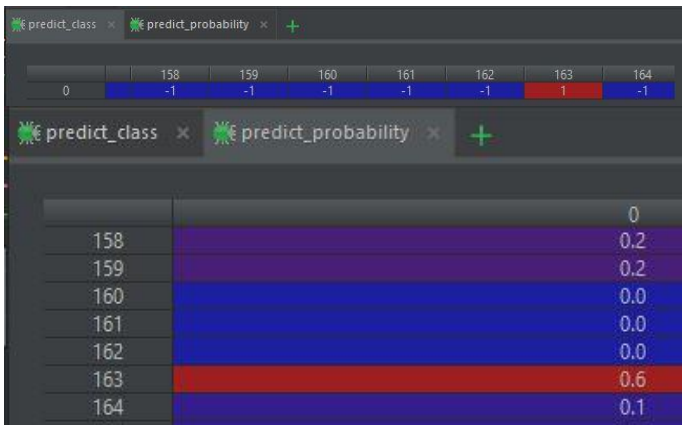


Fig. 8. Observation binary classification and probability

It is therefore important to understand the output of a given classifier when designing a solution, as not all can predict probability scores for each observation, only a binary classification. Decision trees are incapable of providing a probability, whereas random forests can.

6) Classifier hyper-parameter tuning (single)

Baseline AUC scores are calculated using the "out-of-box" default sklearn settings. To extract best predictive performance from each model we look to discover its optimal configuration.

This tuning exercise begins by proposing a range of values for the more important parameters, which includes n_estimators to designate the number of trees to grow and max_depth as the maximum depth of a tree.

```
from sklearn.model_selection import GridSearchCV

def fit_predict_score(clf, clf_name, gs_param):
  clf.fit(dataset_train, target_train)
  predict = clf.predict_proba(dataset_test)[:,1]
  score_auc = roc_auc_score(target_test, predict)

def get_gridsearch_single_abc(param_grid):
  grid_search = GridSearchCV(
      estimator=AdaBoostClassifier(
         DecisionTreeClassifier(
            random_state=20, criterion='gini',
            max_depth=5, max_leaf_nodes=20,
            min_samples_leaf=44,
            min_samples_split=46),
         random_state=20, n_estimators=500),
         param_grid=param_grid, scoring='roc_auc',
         n_jobs=8, iid=False, cv=3, verbose=0)
  return grid_search
```

The code snippet shows single hyperparameter tuning of an ensemble AdaBoost classifier using decision trees as base learners. The decision tree is a predefined, tuned model with parameters like max_depth configured as "sweet spot" values determined in earlier iterations of broader single parameter tuning of the DTC.

For AdaBoost the important parameters are n_estimators and learning_rate. An instance of class GridSearchCV fits and scores the classifier (estimator) for each individual parameter permutation passed in param_grid. The AUC score for each parameter value is output. The last instance of grid_search is always the best performing parameter permutation, which is passed to method fit_predict_score for scoring and logging.

```
def gridsearch_single_abc():
  gs_param = 'n_estimators'
  param_grid = {gs_param: range(50, 1550, 50)}
  grid_search =
    get_gridsearch_single_abc(param_grid)
  fit_predict_score(grid_search, self.abcname,
    gs_param)

  gs_param = 'learning_rate'
  param_grid = {gs_param: [0.1, 0.2, 0.5, 1.0]}
  grid_search =
    self.get_gridsearch_single_abc(param_grid)
  fit_predict_score(grid_search, self.abcname,
    gs_param)
```

Fig. 9 presents output from GridSearchCV scoring of the AdaBoost classifier predicting churn using DS02, with a range of values for hyperparameter n_estimator (number of tree base learners grown). AUC score tapers off at 1450 trees with score of 0.6382. Interestingly, 50 trees returns a score of 0.6273 then sharply drops, only surpassed when using 850 trees and above. For information baseline configuration score was 0.6205.

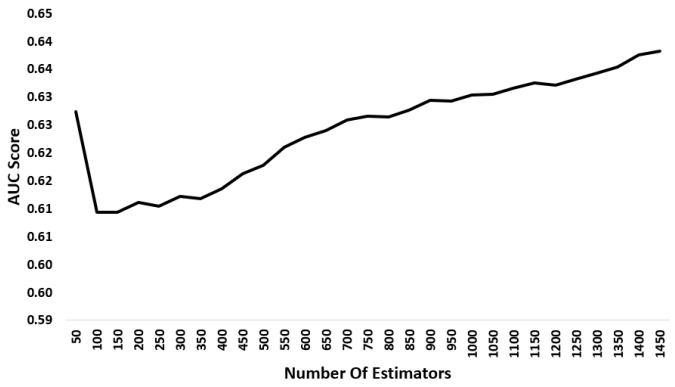


Fig. 9. AdaBoost churn AUC score with number of estimators

The overall purpose of the gridsearch_single methods are to test a broad range of individual values, and for each hyperparameter return which value scored highest. Having this information helps limit as much as possible the permutations tested in multi-parameter tuning which is more computationally expensive given the explosion in multi-parameter value variations. Fig. 10 shows example logged grid search information.

0	AdaBoost Classifier	roc_auc_score	GS Best	{'n_estimators': 1450}	churn	0.660173537
1	AdaBoost Classifier	gs	n_estimators	50	churn	0.627359323
2	AdaBoost Classifier	gs	n_estimators	100	churn	0.609325063

Fig. 10. Hyperparameter tuning output to .csv

7) Classifier hyper-parameter tuning (multi)

We've seen how the tuning of individual parameters can boost model performance over baseline settings. However, often

the very best predictive performance comes from a tuned combination of several parameters. The difficulty lies in finding the best combination of values, especially if the tuning exercise is time or computing power constrained as testing many combinations of parameter values can be prohibitively expensive in both. The optimal approach found was to leverage exploring of single parameter tuning over all 3 predictive tasks to suggest best or closest parameter values observed to work well for all 3 tasks, such as *max_features=None*, so that there was no need to explore variations of this parameter further in multi hyperparameter tuning. This speeds up the tuning process by reducing multi-parameter search space. The code snippet for the gradient boosting classifier is given below:

```
def gridsearch_multi_gbc():
  param_grid = {'max_depth': [2, 3, 4, 5],
      'min_samples_leaf':  range(11, 17, 3),
      'max_leaf_nodes': [None, 5, 10],
      'min_samples_split': [8, 24, 48]}
  grid_search = get_gridsearch_multi_gbc(param_grid)
  fit_predict_score(grid_search, gbcname,
      'IsMulti')
```

The parameter grid is defined in method gridsearch_multi_gbc, with 4 parameters specified, and a total of 108 (4x3x3x3) permutations to test. We now enhance the GridSearchCV template in method get_gridsearch_multi_gbc, no longer using full default settings for the estimator but adding value specifications for n_estimators, max_features, loss and subsample. These values were derived from the earlier single parameter testing and deemed sufficiently performant and not requiring adding to the multi hyperparameter search space. What is key, and found through first-hand experience, is strict restraint over the number of parameter/value combinations specified in param_grid, otherwise permutation counts climb, dramatically increasing runtimes. Best value permutations from searching the multi hyperparameter space are logged for later analysis.

As stated earlier, the objective of the two-step tuning exercise is extraction of best possible model performance. Fig. 11 shows difference between baseline vs tuned scores for churn. All models respond positively to tuning, the greatest increase over baseline configuration score with the decision tree (DTC) classifier (0.1709), demonstrating its receptiveness to tuning. There are some surprises, with AdaBoost (ABC) registering unexpectedly poor performance, being surpassed by DTC and RFC, contrary to RQ2 H3 "*Boosting* > *Bagging* > *Single Tree* ". GBC shows minimal improvement of just 0.0039, underlining just how strong this classifier is "out of box" with well optimised default settings and beating VTC. This is further supported with evidence in Table III showing GBC with baseline settings consuming DS03 achieves best AUC score in study with 0.8112.

Turning to the appetency problem a similar story is illustrated with all classifiers responding to tuning, albeit with varying degree of success, with DTC increasing by 0.2847, RFC by 0.2139 and GBC only by 0.0015. DTC puts in a strong performance, outperforming ABC and only 0.0107 behind RFC. Excluding ABC, RQ2 H3 "*Boosting* > *Bagging* > *Single Tree*" is satisfied. This problem sees the VTC ensemble (RFC, BGC, GBC) beating GBC, due to the observed individual strong performance of the base classifiers RFC and BGC.

Classifiers applied to the third problem of upselling show smaller AUC score margin between baseline and final than observed for the other two problems, with baselines performing well out of box, suggesting upselling is an easier "nut to crack" than churn and appetency. Again, DTC shows the greatest improvement following hyperparameter tuning with an increase of 0.1738, beating ABC. GBC beats VTC by only 0.0010.

The performance of GBC is less variable across all 3 problems, even with its baseline configuration placing the best AUC score of any classifier excepting VTC. ABC is more variable, suggesting its more receptive to tuning in addressing the appetency problem, quite likely due to using a tuned decision tree base learner which performs exceptionally well in appetency.

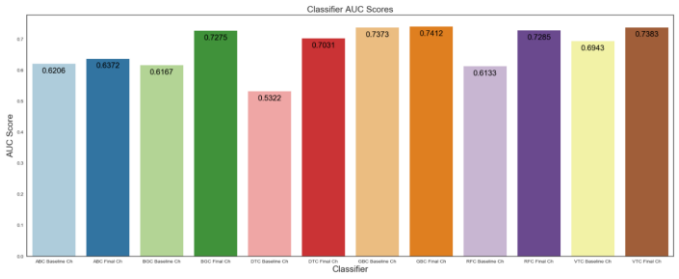


Fig. 11. Classifier baseline vs final tuned, predicting churn (dataset DS02)

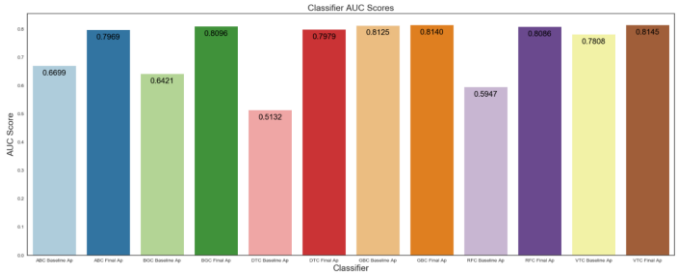


Fig. 12. Classifier baseline vs final tuned, predicting appetency (dataset DS02)

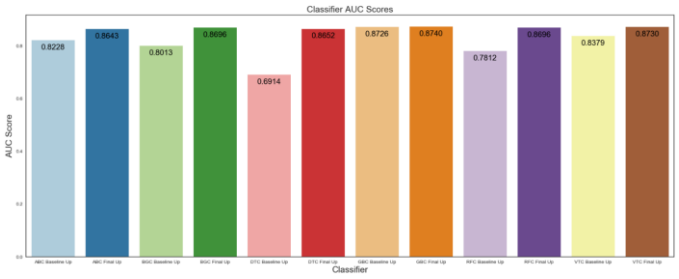


Fig. 13. Classifier baseline vs final tuned, predicting upselling (dataset DS02)

Tuning has been shown to be highly effective for all classifiers except GBC, with value return for the effort and leading to improved KDD Cup ranking, as discussed in Section M evaluation.

8) Score final classifiers

Following the two-stage hyperparameter tuning exercise, where parameter values are tested for performance effect both in isolation and in combination, the final configuration of the classifier is made, adopting the recommendations from grid-search tuning. First the baseline instantiation of DTC, scoring 0.5132 for appetency:

```
clf = DecisionTreeClassifier(
  random_state=self.random_state)
```

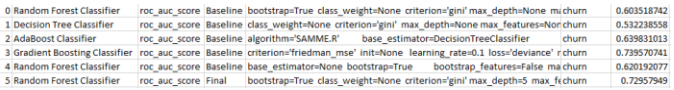
Final hyperparameter settings for DTC, scoring 0.7979 with increase over baseline of 0.2847 for appetency. Tuning is proven to be effective, satisfying RQ2 H4.

```
clf = DecisionTreeClassifier(
  random_state=self.random_state,
  criterion='entropy',
  max_depth=5,
  max_leaf_nodes=10,
  min_samples_leaf=43,
  min_samples_split=46)
```

The final configuration overrides the following default values: criterion from gini to entropy, max_depth from None to 5, max_leaf_nodes from None to 10, min_samples_leaf from 1 to 43 and min_samples_split from 2 to 46.

9) Output results

All scores from baseline, grid search and final tasks are output to the file system as a .csv for follow-up analysis and reporting (Fig.14)



0	Random Forest Classifier	roc_auc_score	Baseline	bootstrap=True class_weight=None criterion='gini' max_depth=None mi	churn	0.603518742
1	Decision Tree Classifier	roc_auc_score	Baseline	class_weight=None criterion='gini' max_depth=None max_features=Nor	churn	0.532238558
2	AdaBoost Classifier	roc_auc_score	Baseline	algorithm='SAMME.R' base_estimator=DecisionTreeClassifier	churn	0.639831013
3	Gradient Boosting Classifier	roc_auc_score	Baseline	criterion='friedman_mse' init=None learning_rate=0.1 loss='deviance' r	churn	0.739570741
4	Random Forest Classifier	roc_auc_score	Baseline	base_estimator=None bootstrap=True bootstrap_features=False ma	churn	0.620192077
5	Random Forest Classifier	roc_auc_score	Final	bootstrap=True class_weight=None criterion='gini' max_depth=5 max_fe	churn	0.72957949

Fig. 14. Classifier scoring output to .csv

M. Evaluation

Table VIII presents final AUC scores for all 6 classifiers. Following KDD Cup convention, the 3 individual predictive behaviour scores are averaged then used to rank the solution. The best performing experiment variant was the gradient boosting classifier consuming DS03, setting an average AUC score of 0.8118, ranking 34th of 89 total challenge submissions. Note the baseline variant of GBC predicting appetency (0.8237) outperformed best effort tuned model (0.8122). The voting classifier is a close second in this study, 0.0009 behind GBC, ranking 35th. Both classifiers satisfy hypothesis H5 as results are placed above mid table, and H1 with freely available open source classifiers from the sklearn module proving they are competitive. Overall results are satisfactory given teams from well-respected institutions such as IBM (1st, 0.8521) and University of Melbourne (2nd, 0.8484) participated.

Bagging (0.8023, 43rd) just outperforms random forest (0.8022, 43rd). On individual tasks RFC is slightly better than BGC in all 3 tasks when disregarding dataset, however BGC finds the better average with DS02. Decision trees claim 5th spot in this study with a score of 0.7887 and ranking 61st, unable to outperform any of the ensemble methods excepting AdaBoost. Poorest performing is AdaBoost, scoring 0.7877, ranking 63rd.

TABLE VIII FINAL CLASSIFIER RESULTS

Classifier	Data	Churn	Appetency	Upselling	AUC Avg	KDD Cup Rank
Decision Tree	DS01	0.7022	0.7939	**0.8680**	0.7880	62
	DS02	**0.7029**	0.7978	0.8653	**0.7887**	**61**
	DS03	0.6926	**0.8035**	0.8631	0.7864	68
Random Forest	DS01	0.7240	**0.8094**	**0.8708**	0.8014	45
	DS02	0.7286	0.8084	0.8697	**0.8022**	**43**
	DS03	**0.7300**	0.7930	0.8693	0.7974	54
Bagging	DS01	0.7246	0.8075	**0.8699**	0.8007	46
	DS02	0.7280	**0.8090**	0.8698	**0.8023**	**43**
	DS03	**0.7291**	0.7923	0.8691	0.7968	54
AdaBoost	DS01	0.6972	**0.8001**	**0.8657**	**0.7877**	**63**
	DS02	**0.6983**	0.7969	0.8643	0.7865	67
	DS03	0.6420	0.7885	0.8458	0.7587	72
Gradient Boosting	DS01	0.7394	0.8124	0.8723	0.8080	36
	DS02	**0.7410**	0.8139	**0.8740**	0.8096	**34**
	DS03	0.7408	**0.8237***	0.8710	**0.8118**	**34**
Voting	DS01	0.7357	0.8146	0.8717	0.8073	37
	DS02	0.7382	0.8144	**0.8728**	0.8084	**35**
	DS03	**0.7412**	**0.8147**	0.8703	**0.8087**	**35**

Given the strong "out-of-box" baseline performance of GBC it really is the preferred classifier, benefitting from simplified configuration over VTC (1 classifier configuration vs 3) and reduced computing runtime (GBC 22s vs VTC 397s). GBC is superior to BGC, RFC and DTC. The best gradient boosting churn score of 0.7410 ranks 28th (IBM 0.7651), best GBC appetency of 0.8237 ranking ~36th (IBM 0.8819), and best GBC upselling score of 0.8723 ranking ~34th (IBM 0.9092). This indicates our gradient boosting churn prediction lifts the average, helping to improve final ranking and suggests further effort should be focused on appetency and upselling tasks. The poorer ranked appetency score also illustrates how challenging appetency is to correctly classify, particularly when there is no improvement gain from any tuned version which is outperformed by baseline. What is apparent is the density of submissions. Only 0.0096 separates our gradient boosting and random forest scores, and yet this makes a difference of 9 in the KDD ranked order. As proven in this study, even minor performance extracted from adjusted data preparation and/or model tuning can prove significant.

Such poor performance from AdaBoost was unanticipated, approximately on par with decision tree performance, but as an ensemble method growing several hundred trees better was expected. Although efforts were invested in tuning and tweaking the AdaBoost classifier, such as adjusting the number and depth of trees grown in the decision tree base learner, the final score could not be improved. This suggests alignment with "*Consistent with theory, boosting can fail to perform well given insufficient data, overly complex weak hypotheses or weak hypotheses which are too weak. Boosting seems to be especially susceptible to noise*" [19, p. 10]. Reviewing the baseline configuration churn scores for bagging (0.6167) and AdaBoost (0.6206) shows AdaBoost the better default performer with the bagging classifier responding better to tuning. Random forest ranks joint 3rd in this suite, proving its ability to handle many features with noisy and unbalanced data, with low computing resource demands, completing the appetency prediction task in 3s. Despite poor performance from AdaBoost, I believe hypothesis H3 can be confirmed, with bagging better than single decision trees and gradient boosting better than bagging.

Given the difference in ranking for each classifier according to dataset used, for example ABC (63rd DS01, 72nd DS03) and BGC (43rd DS02, 54th DS03), I can confirm hypothesis H2 "*Determine the importance of data preparation on tree-based classifier predictive performance*". With more time research would include binning the many unique feature values into smaller sub-groups and further investigate both weighting and sampling to understand why they don't help remedy the dataset imbalance. I would agree with [28, p. 100] that suggests "*based on the results of our post-challenge submission, we can conclude that significant progress may be achieved using more advanced pre-processing and feature selection techniques*". One further area worthy of additional research is improving the voting classifier ensemble which currently utilizes our 3 best tree-based algorithms - gradient boosting, bagging and random forest, both exploring this approach and taking an alternative with a significantly greater number of "just better than random" models to increase the voting population.

With achieving a final rank of 34th we have satisfied one goal set by the organizers of the KDD Cup 2009, proving "*they could handle a very large database, including heterogeneous noisy data (numerical and categorical variables), and unbalanced class distributions*" [3, p. 2]. With data preparation and classifier build, tune and test technically implemented with only 2 python packages - sklearn and pandas - we can see how the vision of

IBM from back in 2009 is tangible when they remarked "*It is notable that this performance was obtained through fairly standard techniques without much need for human expertise or intervention, or tailoring to the particular problems addressed in this competition. One can easily imagine all these techniques being incorporated in a general purpose push-button application*" [22, p. 27]. In an industrial context gradient boosting could be deployed to productive use with very little effort and return acceptable predictive performance with low computing resource demand.

IV. Neural Network Classifier Approach

A. Introduction

Recent progress in artificial neural networks (ANNs) has enabled the surpassing of other machine learning algorithm benchmarks in a wide variety of applications, including image and speech recognition, natural language understanding and bioinformatics [32] [33] [34]. Essentially, an ANN consists of a hierarchical architecture with layers that contain individual information processing units, or neurons (Fig. 15). It generally has three types of layers: input layer, hidden layer(s), and output layer. Furthermore, the number of neurons within each layer can vary depending on the number of input variables, the complexity of the problem being modelled, and the required nature of the predictions. For example, ANNs can represent functions with higher complexity by increasing the numbers of hidden layers and units within them [32].

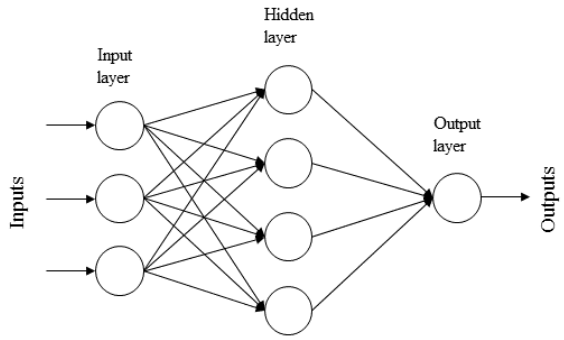


Fig. 15. Example of a feed-forward neural network.

In their simplest form, a neuron in a standard feed-forward, densely-connected ANN (also known as a Multi-Layered Perceptron) consist of an activation function, which takes an input value, applies a function to it, and outputs the new value [2, p. 288]. The input value for a neuron in the first layer is simply the observed value of a particular variable for a given observation. In the other layers, the input values are the outputs of other neurons. The output of the neuron(s) in the final layer are the predicted values of the model, usually a continuous value for a regression problem or a binary prediction value for a classification problem. What affects the values passed through the model are weights. Each neuron assigns a weight to each of its inputs before summing them and this weight is how the network learns from input data.

There are a variety of techniques for how weights are updated, with one of the most common methods involving backpropagation, an efficient gradient descent method for supervised learning models (those for which previous true output values are known) [2, p. 299]. First, a cost (or loss) function, such as binary cross entropy, must calculate an error value by comparing the true training outputs against the current predictions. Then the backpropagation algorithm calculates the partial-derivative, or gradient, of the cost function with respect to the weight in question. Finally, an optimization algorithm, such as stochastic gradient descent (perhaps the most widely used optimizer [34]), is used to try to find the global minimum of the cost function and thereby minimise the error. The result, once this process has been repeated numerous times, is ideally a neural network that is able to accurately predict the outputs of the training data and, more importantly, is able to make future predictions based on new input data.

ANNs, then, are relatively complex algorithms taken as a whole, although there is great simplicity in their individual elements and elegance in their construction, and when scaled and applied to appropriate domains they have been shown to achieve unrivalled results. What's more, as their use has become more popular, user-friendly tools to support their implementation have been developed. Software such as Keras allows the relatively rapid construction and training of ANNs, by providing simple methods for combining model components [35], and automating features such as backpropagation. Keras is a high-level neural network API, written in Python and capable of running on top of TensorFlow, an API for numerical computation using data flow graphs. TensorFlow [36] was originally developed by members of the Google Brain team for their deep learning research, and it has been used across many areas of computer science, including speech recognition, computer vision, robotics, and natural language processing.

Neural networks can be powerful prediction algorithms, however, they are blackboxes in so much as it is extremely difficult to understand why they make the predictions they do. This may be a problem for some uses, particularly in CRM if a business, for example, wants to understand which key variables correlate with a customer deciding to switch providers. Knowing that there is a strong probability that a customer will switch without knowing how to stop the customer doing so may not fulfil a business need. Conversely, if behavioural economics has taught us anything, it is that human behaviour can often be less than rational in an objective sense, and as deep neural networks are adept at modelling non-linearities without prior judgements being made about which features are important, it could be concluded that they have utility in predicting the future actions of customers using behavioural data [37].

Regardless, less than 20% of the classifiers used to submit predictions in the original KDD Cup 2009 were neural networks, and they were not the best performing classifiers [3]. This could be because they are not well suited to modelling categorical variables, especially those with a large number of unique values, nor data that is not well understood, and therefore cannot be properly prepared for modelling [2]. Despite this, a new attempt is made here using methods that have been developed or used more prevalently since 2009, and which therefore were not necessarily readily available to competitors at the time. This includes the TensorFlow API (2015), the Keras API (2015), the ReLU activation function and its variants (tanh or sigmoid were the standard activation functions as recently as 2012) [38], the Swish activation function (2017), and the Adam optimizer and its variants (2015).

B. Development environment

Software Components – MacOS High Sierra 10.13.1, Python 3.6, Spyder 3.2.6;
Hardware Components – MacBook Air, 1.6 GHz Intel Core 2, 2GB RAM;
Python APIs – Keras (neural networks), TensorFlow (numerical computation), Hyperopt (hyperparameter search), Sci-kit learn (machine learning), Pandas (data structures and analysis), NumPy (mathematics).

C. Methodology

To an extent, the architecture of an ANN for the KDD Cup 2009 is dictated by the challenge. It is made up of three classification problems, and even though one multi-class model could have three targets representing three different classes (with a fourth for a nil class, representing a customer who is predicted to neither churn, be upsold, or purchase another product), a decision was made to use three different binary-classification models. This was influenced by the KDD Cup 2009 organisers suggesting this approach [39] and by the possibility of a single customer being inclined towards more than one class, e.g. both upselling and appetency. Therefore, the output layer of the ANN is made up of one neuron with binary output, and a sigmoid activation function fulfils this criteria following the transformation of target labels from ([1,-1]) to ([1,0]). Similarly, the input layer is made up of a neuron for each independent variable (feature), or 387 neurons. The problem then becomes deciding on the following hyperparameters:

1) Number of hidden layers

The options originally decided upon for this project were for there to be either two or three hidden layers. Canonically, the greater the depth of the neural network the better they are at recognising certain features within data, particularly those at various levels of abstraction, and so the better they are at generalising [40]. However, following a stagnation of results, a bottom up approach was taken with regards to building up the model from a small architecture, and it was found that one or two layers achieved better accuracy than three. Therefore, the final hyperparameter search included only one and two hidden layers.

2) Number of neurons in each hidden layer

Similarly, the choice of number of neurons in each hidden layer was originally decided to be either 50, 100, 200, 300, 400, 700, 900, or 1100, with each hidden layer able to have a different value. More neurons allow more features to be learnt by the model [38] and, as with the number of layers, there is no agreed method with which to anticipate the optimal number of neurons for a given problem. This is why a fairly wide range around the number of input neurons was chosen. Nevertheless, a good deal of research has been conducted on this topic. For example, Jinchuan and Xinzhe [40] have developed a formula tested on 40 cases: $N_h = (N_{in} + \sqrt{N_p})/L$ where L is the number of hidden layers, N_{in} is the number of input neurons, and N_p is the number of input samples. Applying this, we obtain 560 hidden units for one hidden layer, 280 for two hidden layers, and 187 for three hidden layers. This lends support to a range of 50-1100 units for hyperparameter search. However, as with the number of hidden layers, in a bottom up construction of the model it was found that reducing the number of neurons to 50 for each hidden layer (a combination not necessarily found through random hyperparameter search of limited evaluations), improved results. This is discussed further in section 3.6.1. Therefore, the final hyperparameter search included 20, 30, 40, 50, 60, 70, 80, and 90 hidden neurons.

3) Activation function in first and hidden layers

Traditionally, non-linear activation functions, such as sigmoid and tanh, were used in neural networks to give them the ability to capture non-linearities in data. However, recent advances in methodologies have supplemented this approach with piece-wise linear functions due to a few key disadvantages of the non-linear functions [41]. For example, the sigmoid activation has a steady state regime around 1/2, therefore, after initializing with small weights, all neurons fire at half their saturation regime. This is biologically implausible and hurts gradient-based optimization [42]. Furthermore, as the sigmoid and tanh curves approach their limits, a gradient vanishing effect is created, slowing optimization.

ReLU, a rectified linear unit [38], is now the most popular activation function in deep learning, and has also been identified as essential for state-of-the-art neural networks [33] [43]. Because ReLU treats all input less than zero as zero, it allows a network to obtain greater sparsity in its representations. Furthermore, computational gains are made [33]. Therefore, even though they can still take advantage of semi-supervised setups with extra-unlabelled data, deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labelled datasets [33]. Experimental results show successful training behaviour of this activation function, especially for deep architectures, i.e., where the number of hidden layers in the neural network is 3 or more [44]. This is because any non-linearity must be represented by a greater network complexity in which non-linearities are approximated. Potential problems could theoretically arise from the hard saturation at 0, which could harm optimization by backpropagation (the learning mechanism implemented automatically by Keras). To remedy, this a variation of ReLU has been developed, called Leaky ReLU, which has a small and fixed negative gradient a, as shown in Fig. 16. Parametric Rectified Linear Unit (PReLU) takes this idea further by making a an adaptive parameter that can vary as the network is trained [43].

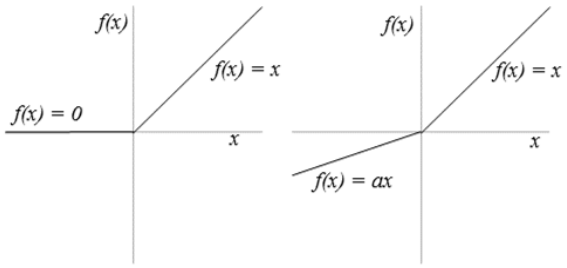


Fig. 16. ReLU and Leaky-ReLU/PReLU activation functions

Given that ReLU and variants have been so prevalent over recent years, the search is on for the next type of activation function. Although various hand-designed alternatives to ReLU

have been proposed, none have managed to replace it due to inconsistent gains [45]. To meet this challenge a team at Google Brain used automatic search techniques, a combination of exhaustive and reinforcement learning-based search, to discover new ones [45]. Their best discovered activation function is $f(x) = x \cdot sigmoid(\beta x)$, which they name Swish (see [45]). Its shape, though unique, is similar to ReLU, which makes it suitable for practitioners to replace ReLU with Swish in any neural network. The team's extensive experiments purport to demonstrate that Swish consistently matches or outperforms ReLU on deep networks applied to a variety of challenging domains, such as image classification and machine translation.

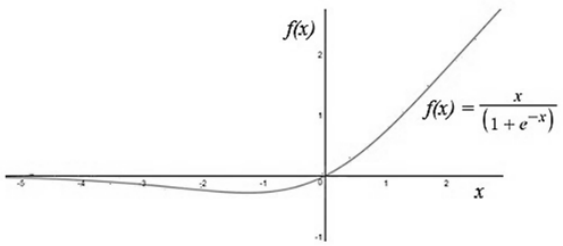


Fig. 17. The Swish activation function.

Given that Swish was only identified as a successful activation function in October 2017, Keras does not yet support it. However, as it is so simple, it is easy to customise its implementation using open source examples, which when called by specifying the activation function as being `'swish'` in Keras, it is successfully incorporated into the network.

```
def swish(x):
     return (K.sigmoid(x) * x)

get_custom_objects().update({'swish':
Activation(swish)})
```

In summary, given their pervasiveness, the ReLU and PReLU activation functions, are used as possible options for the hyperparameter search, along with one of the most recent activation function additions to the field of artificial neural networks, Swish.

4) Optimizer

Stochastic Gradient Descent (SDG) has proved itself as an efficient and effective optimization method that has been central in many machine learning success stories, such as recent advances in deep learning [46]. SGD with momentum is used here, which incorporates momentum to accelerate SGD in the relevant direction of the gradient and thus potentially 'roll' out of local minima, and Nesterov accelerated momentum, which 'looks ahead' to avoid rolling out of global minima once found [47]. A more recently developed optimization function is Adam [48]. It is a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. The method is designed to combine the advantages of two recently popular methods: AdaGrad, which works well with sparse gradients [48], and RMSProp [49] which works well in on-line and non-stationary settings. Some of Adam's advantages are that its step sizes are approximately bounded by a step size hyperparameter, it works with sparse gradients, and it naturally performs a form of step size annealing [48], which enables it to 'jump' out of local minima. Furthermore, a variant of Adam has been developed called Nadam, which is essentially Adam with Nesterov momentum. Therefore, the final search for the best optimizer includes SGD with momentum and Nesterov accelerated momentum, Adam and Nadam. All parameters for the three optimizers were set as the Keras recommended default, except that a learning rate decay of 0.004 was included with Adam (default=0), in order to provide parity with the other two optimizers and with the intention of achieving greater accuracy.

5) Batch size

Batch size is the number of observations that are used to train the ANN in a single iteration, with its size often dependent on the amount of memory available to run the network. Experimentation has found that larger batch sizes result in higher test accuracies [50]. However, the higher the batch size the greater the computational cost and so the learning may be slow and fail to converge. Conversely, a small batch size will result in more stochastic training that may diverge from the gradient of the entire dataset and also be slow to train. Typically batch size are 16, 32, 64, 128, 256, or 512. Given hardware constraints, the final hyperparameter search includes batch sizes of 16, 32, 64, and 128.

6) Dropout probability

Overfitting is a serious problem in ANNs. It occurs when the network learns the training data well, but is unable to generalise and make accurate predictions for new data. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time [51]. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training, which encourages a number of neurons to represent a given feature as opposed to a single neuron [51]. The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data. Using this technique, a neuron is temporarily removed from the network. The choice of which units to drop is random and in the simplest case, each unit is retained with a fixed probability p independent of other units. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has weights reduced using the probability that a given neuron was removed from the network. A probability p of 0.5 has been found to be close to optimal for a wide range of networks and tasks [51], however, with the intention of providing the possibility of small rates of dropout in some sample networks, p = 0.2 was considered a good value to provide a range. Therefore, the final hyperparameter search included independent dropout probabilities for each hidden layer taken from a random uniform distribution between 0.2 and 0.5.

7) Automated hyperparameter search

The above hyperparameters are selected randomly using the Hyperopt API [52], which makes a selection of each, runs the model, evaluates it using a bespoke criteria, and then returns the parameters of the best performing model for future use. As a

very brief and incomplete overview, first a hyperparameter space is defined with the different hyperparameter choices and values, for example:

```
space = {'optimizer': hp.choice('optimizer',['nadam', 'adam', 'sgd']), …}
```

Then a method to construct and train a neural network using a choice of hyperparameters is defined, using the object *params*:

```
def f_nn(params):
    model = Sequential()
    model.add(Dense(input_dim = 387, units=params['units1'], kernel_initializer = "glorot_uniform"))
    …
```

The method also enables the trained network to make predictions on the test data and return an evaluation metric. Here we calculate the AUC score and return it as a negative value, because when `f_nn` is called the `fmin` method returns the model parameters that best *minimize* the specified loss.

```
best = fmin(f_nn, space, algo=tpe.suggest, max_evals=evaluations, trials=trials)
```

The best parameters are then used to automatically re-construct and re-train a new network to provide a final set of predictions, a resultant AUC score, relevant plots of training history, and print out what the chosen parameters are for future use.

D. Further methodological considerations

1) Overfitting – early stopping and check points

Other considerations must also be made for other features of a neural network. The first relates to the problem of overfitting already highlighted, and is how to decide when to stop training the model. A common way to help alleviate the problem of overfitting is to stop training as soon as the performance on a validation dataset starts to deteriorate [53]. This can be achieved using a callback function in Keras called EarlyStopping, which allows the user to specify which metric to use to stop the training, usually either the validation set accuracy or loss, and whether any leniency should be given to allow temporary lapses in validation performance (using a patience value). Different patience values were experimented with but a value of 10 epochs was generally used. Another useful feature in Keras, called CheckPointer, is the ability to save a 'snap shot' of the model during an earlier epoch (capturing the value of each weight), again using a validation set performance metric. This allows training to continue beyond an optimal epoch, and then be recovered, to ensure that the model is as trained as it can be and no more. The best weights found can then be loaded into the model and used to make the final predictions on a test set, or future input data. This feature has been incorporated into these ANN classifiers along with EarlyStopping.

2) Batch normalisation

A further consideration for a neural network, especially one using a rectifier activation function that can return any positive value, is the use of batch normalization. Training neural networks can be complicated by the distribution of each layer's inputs varying significantly as the previous layer's inputs change. This can hamper the speed of training as values vary wildly, which in turn requires that learning rates are lowered [54]. Batch normalization, however, reflects the benefits of normalising a dataset pre-training, except that here it is also performed on each input before it is passed to the activation function. It is also important that a neural network should be initialised through the setting of its weights to random, non-zero values, otherwise they are simply unable to learn. The standard Keras Glorot initialisation is utilised [55].

3) Loss function

The appropriate loss (or cost) function must also be selected for a neural network. As already noted, a neural network whilst it is being trained attempts to minimise a loss function. Given that the goal of the classifiers developed for this project is to predict a binary outcome, the loss function used here is binary cross entropy. Cross entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1 with respect to the target label. The sigmoid activation function used in the output unit provides this probability. Cross entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be undesirable and result in a high loss value.

4) Class imbalance – weighting and oversampling

Finally, early runs of neural network classifiers revealed a problem caused by the target values being highly unbalanced, as referred to in the introduction. They would frequently reach very high accuracy scores of circa 98% but produce mediocre AUC scores. This is because they would learn to make every single prediction negative, which would be correct in circa 98% of observations. However, a classifier which makes no predictions of which customers might churn, be sold another product, or be upsold, is clearly of no business use, regardless of how accurate it is. This highlights why AUC is a superior evaluation metric for this dataset. A method to remedy this problem, to some extent, is to weight the positive target values during training. This was achieved using the Scikit-learn API method `compute_class_weight` (which has the effect of balancing the frequency of the classes), as shown:

```
class_weight = compute_class_weight('balanced', np.unique(Y_train), Y_train)
class_weight_dict = dict(enumerate(class_weight))
```

`class_weight_dict` is then a standard input of the Keras `model.fit` method.

Another approach to dealing with unbalanced classes is to oversample the less frequent class, i.e. to duplicate less frequent observations. This can be achieved using a Scikit-learn API called imbalanced-learn. The following is the Synthetic Minority Oversampling Technique (SMOTE) [56].

```
sm = SMOTE(random_state=7, ratio = 1.0)
X_train_res, Y_train_res = sm.fit_sample(X_train, Y_train)
```

`X_train_res, Y_train_res` are then standard inputs of the Keras `model.fit` method in place of `X_train and Y_train`. The ratio value ([1.0, 0.0]) can be set from 1.0 (to make the two classes balanced) to 0 to leave the data unchanged. However, it

was found that oversampling in place of weighting the classes significantly and adversely affected the performance of the neural network classifiers. This was true for ratio values of 1.0 to 0.2, and even when weighting was also included for low ratio values, indicating unsuitability for this dataset.

E. Data Preparation

1) Numerical features

The KDD Cup 2009 'small' dataset was prepared for modelling using the development environment already described, principally using the NumPy and pandas APIs. First the 190 numerical variables were separated and their unique value frequencies were calculated by grouping and then sorting them in ascending order. By then looking across the top few frequency rows it was apparent which variables only contained a single value and those that had only a few values (8) with low frequencies. To simplify the data, the variables that corresponded to the these categories were deleted: 1, 3, 7, 10, 14, 18, 19, 25, 26, 28, 30, 31, 33, 38, 41, 47, 48, 51, 53, 54, 66, 78, 81, 86, 89, 92, 99, 109, 115, 117, 121, 129, 137, 140, 141, 142, 146, 166, 168, 172, 174, 184. The dataset also contained a number of variables with missing data. Two canonical approaches to missing data were adopted, the first being to replace all missing values with the variable's mean, the second being to replace all missing values with zero. Following later model comparisons, and as corroborated by this paper's other classifiers, the latter option was generally found to confer greater performance. Both can be achieved with a single line of code using the pandas method *`fillna`*.

2) Categorical features

Next, the 40 categorical variables were prepared. Again the value frequencies were calculated using the methods described above. This allowed the identification of a value threshold for each variable that would retain the 8 highest frequency values, and the subsequent grouping of all the other values into a new value called '*uncommon'*, as follows:

```
 def groupCatFeatures(df, threshold, column, prefix,
normalize=False):

 freqencies = df[column].value_counts( sort=False,
    normalize=normalize)
 idx = freqencies[freqencies < threshold].index
 tmp = df

 if idx.shape[0] > 0:
   tmp[column] = df[column].replace(idx, 'Uncommon')
   else:
   tmp = df
  d   =   pd.get_dummies(tmp,   columns=[column],
prefix=prefix, dummy_na=True)
  return d
```

This reduced the number of values for each categorical variable to 9 in total. This is a valuable step for neural networks given that any categorical variables must be transformed to numerical values. Furthermore, given that no information about what each variable represents is available for this dataset, the creation of dummy variables for each categorical variable was preferred to any speculative ordinal numerical encoding. This was achieved in the code shown above through the use of the pandas `get_dummies` method, in which each value is transformed into a unique binary variable. Usefully, each category was found to include NaN values, which enabled the easy deletion of this dummy variable for each category in order to avoid the dummy variable trap of multicollinearity.

3) Splitting and normalising dataset

Finally, the numerical and new categorical variables were concatenated to create the final 387 variables used for modelling and exported to a .csv file for future use. When imported for the training and evaluation of a model, a custom `data()` method was called that split the dataset into training (0.6), validation (0.2) and test data (0.2) using the Scikit-learn's `train_test_split` method, with the test ratio the same as used for the evaluation of the other classifiers developed in this paper. The *`data()`* method also normalized the dataset using Scikit-learn's *`fit_transform`*, which enables the training, validation and test data to be normalized using the training data's distribution alone. This avoids the problem of the training data 'learning' anything about the validation and test data. The scaling of data is a crucial step for the development of a neural network to avoid variables with greater ranges dominating its training even if they have little underlying explanatory power.

4) Principal Components Analysis

Of note, Principal Components Analysis was explored briefly but found to adversely and significantly affect model performance. This corroborates an observation by the KDD Cup 2009 organisers that very few teams incorporated it [3]. However, for completeness, principal components of the dataset were calculated using the following Scikit-learn method:

```
pca = decomposition.PCA(n_components=387)
pca.fit(X)
X = pca.transform(X)
```

Fig. 18 was produced to identify an appropriate number of principal components to use. A model tested incorporated 250 components, which as can be seen accounts for just under 100% of variance.

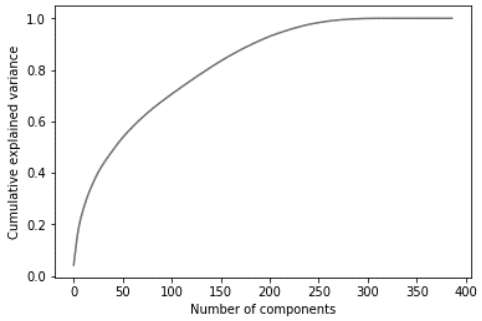


Fig. 18. A plot of principal components against cumulative variance.

5) General observations

Observations on the dataset, in addition to those already made in this paper's introduction, are that it is a likely the KDD Cup 2009 dataset could be prepared for modelling much more accurately if information on what the variables represent were available. As already stated, some categorical variables could well be ordinal, in which case they would not need to be turned into dummy variables, increasing the model's complexity. Some numerical variables could also be ordinal categories (those that increase by a factor of 7, for example). These may, therefore, benefit from any missing values being transformed to

zero, when other continuous numerical values would benefit from missing values assuming the mean. So transforming all numerical missing values to either the mean or zero is perhaps less than optimal. But, again, without knowing what the variables are, or performing a grid search on all possible data preparation options for each variable, it is difficult to know how best to proceed. Given the time available and current development environment a best guess must suffice.

F. Results

As stated in this paper's introduction, Area Under the Curve (AUC) score was chosen as the evaluation metric for our classifiers. This tests for both sensitivity and specificity, using the prediction probabilities generated by the model. However, these probabilities can also be rounded using a threshold to provide binary predictions, an example threshold might be 0.5 (as used here for all accuracy metrics). The total number of correct binary predictions can then be divided by the total number of predictions to give a simply accuracy, which is used to provide greater insight into the performance of the neural network classifiers generated when compared against the AUC score. It should be noted, however, that for classification problems with very unbalanced classes, accuracy alone is not the best evaluation metric. For example, with the positive target label for upselling only amounting to 7.36% of all targets, the classifier could make all predictions zero and achieve an accuracy of 92.64%. This can create model training problems that will be discussed below.

1) Initial results

As already mentioned in the methodology, the initial hyperparameter search with 40 different evaluations for each classifier was centred on models later found to be too complex for the training data. Using a choice of two or three hidden layers and 50-1100 hidden units produced the following initial results using an early stopping patience value of 50. (It should also be noted that initial ranges for drop size and batch size were also different to the final parameters available for selection.). Table IX shows best initial classifier hyperparameters of the 40 randomised combinations tested, with the resultant initial best AUC scores. All results are given to four significant figures.

TABLE IX
INITIAL BEST HYPERPARAMETERS

Params \ Classifier	**Upselling**	**Appetency**	**Churn**
Layers	3	3	2
Units1	1100	50	50
Units2	1100	300	50
Units3	100	300	N/A
Drop Out1	0.7438	0.6182	0.4913
Drop Out2	0.2933	0.4283	0.6717
Drop Out3	0.3877	0.7012	N/A
Act. Func.	PReLU	ReLU	ReLU
Opt. Func	SGD	SGD	Nadam
Batch Size	40	60	40
AUC Score:	0.7892	0.7193	0.6569
Accuracy:	0.8439	0.9552	0.7410

It is interesting to note that although the AUC score for *upselling* is significantly higher than for *appetency*, its accuracy is significantly lower. This demonstrates why accuracy is not an ideal metric in this instance. Upon no further improvement in performance, a bottom up approach to construct a neural network with a small starting architecture was made, with the view to increase complexity until no further performance improvements were made. Indeed, a reduced architecture of only one hidden layer with only 50 units resulted in an immediate improvement in AUC scores, which when increased to 100 units was not the case. Therefore, a subsequent hyperparameter search was made, with a choice of only one or two hidden layers and 20-90 hidden units.

2) Final results

The best and final classifier hyperparameters of the 25 randomised combinations for each target class tested are shown in Table X.

TABLE X
FINAL BEST HYPERPARAMETERS

Params \ Classifier	**Upselling**	**Appetency**	**Churn**
Layers	2	1	2
Units1	30	70	40
Units2	20	Nil	30
Drop Out1	0.4296	0.392	0.4991
Drop Out2	0. 4169	N/A	0.4354
Act. Func.	PReLU	Swish	PReLU
Opt. Func	Nadam	Nadam	Nadam
Batch Size	16	16	16

Following the selection of the best hyperparameters for each classifier, the classifiers were re-trained using various early stopping patience values for the reasons discussed in the evaluation section below. (See Fig 19-21 for accuracy plots of the training histories with a patience value of 50 epochs.) Additionally, in order to test the robustness of a smaller architecture in comparison to those originally trialled, a final run was conducted with an early stopping patience of 200 epochs and two hidden layers of 387 and 200 units respectively, but otherwise the same parameters for each classifier.

As can be seen in Table XI, significantly greater (and perhaps more intuitively and theoretically appropriate [40]) complexity in the neural networks, with training times in excess of 7 hours, resulted in worse or insignificant increases to AUC scores, but significantly better accuracies. This despite a checkpoint being made of the best weights based on validation data accuracy to guard against overfitting.

The final results of the three best classifiers are shown in Table XI.

TABLE XI
FINAL RESULTS

Classifier:		**Best params**	**Best params**	**Best params**	**Best var.***
Patience:		20	50	200	200
Upselling	AUC	0.8118	0.8166	0.8125	0.7806
	Accuracy	0.8234	0.8539	0.852	0.9298
	Epochs	28	101	350	325
Appetency	AUC	0.7155	0.7355	0.7164	0.7197
	Accuracy	0.884	0.9704	0.9705	0.9797
	Epochs	31	90	250	292
Churn	AUC	0.6891	0.6808	0.6681	0.6475
	Accuracy	0.7176	0.7326	0.8512	0.8889
	Epochs	68	95	480	928
AUC mean		0.7404	0.7427	0.7323	0.7159

* Best var. is a variant on the best params in which layers=2, units1=387 and units2=200

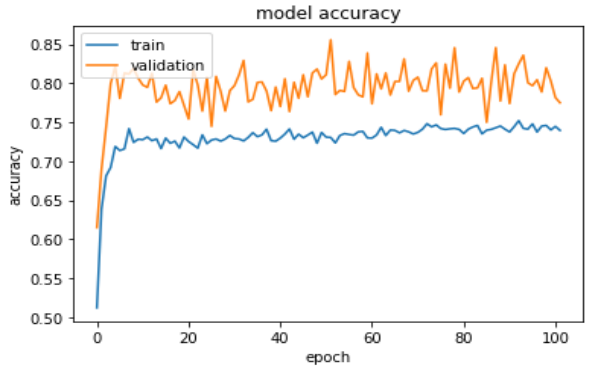


Fig. 19. Upselling accuracy training history with patience of 50 epochs.

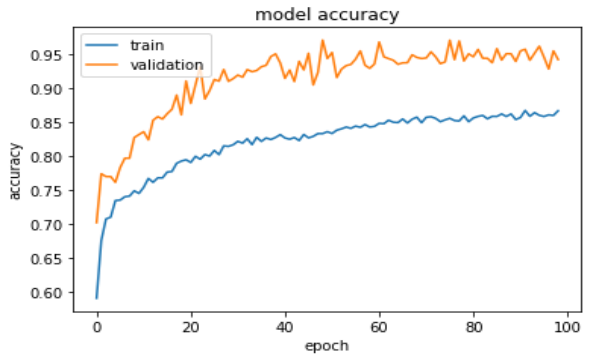


Fig. 20. Appetency accuracy training history with patience of 50 epochs.

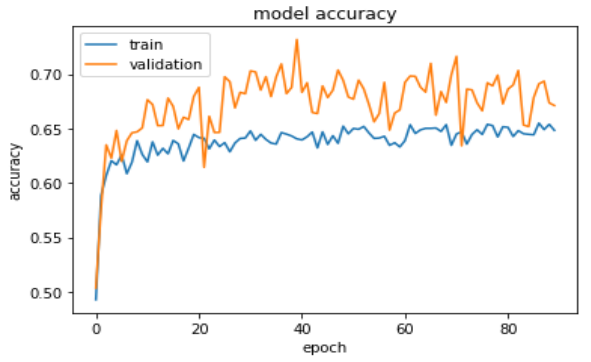


Fig. 21. Churn accuracy training history with patience of 50 epochs.

G. *Evaluation*

The best *upselling* AUC score is 0.8166, the best *appetency* AUC score is 0.7355, and the best *churn* AUC score is 0.6891, with a combined average of 0.7471, comfortably surpassing the naïve bayes benchmark and putting the attempt at 77th position on the original KDD Cup 2009 slow track results list. As is apparent from the results, a much simpler architecture than was initially assumed necessary, given the relatively large number of samples and variables, resulted in a higher overall AUC score. This is perhaps a reflection of a large amount of redundancy in the Orange dataset, i.e. it contains a large amount of data that has little correlation with the target variables. In this way, a small neural network is able to model the key independent variables without introducing unnecessary complexity that may adversely affect performance.

Nadam has gained the most empirical support as an optimizer in this relatively small project, at least for small network architectures. PReLU has outperformed the simpler ReLU, although not for larger networks (providing support for ReLU's use in generating network sparsity). Swish also makes one appearance in the best and final models selected, indicating its early utility as an activation function to rival the dominant linear rectifiers. A further observation is that a batch size of 16, the smallest batch size, sweeps the board across the best models, perhaps indicating its role in providing greater stochastic learning. However, given the small sample size with which these observations are made, and lack of statistical rigour, no firm conclusions can be drawn about how well each component mentioned above outperforms their competitors.

The most significant evaluation of the results perhaps relates to the unbalanced nature of the target classes. It is suggested that using the binary cross entropy loss function for a classification problem in which target labels are significantly unbalanced is not entirely optimal, even when using weighting to balance the classes in training. This is perhaps because each loss function error contributes equally to the overall accuracy of the model at the end of each training epoch. And when one class of target prediction, in this case zero, contributes more to the accuracy of the model, the effect is to focus on sensitivity over specificity, or, in this example, accuracy over the AUC score. This is borne out through an increase to the number of training epochs (by increasing early stopping patience), and a subsequent improvement to validation accuracy, but the result being a *lower* AUC score when final predictions are compared to the test set. This is apparent by an increased accuracy from 0.8234 to 0.8539 for the *upselling* classifier when going from 20 to 50 epochs, but a corresponding reduction in AUC from 0.8166 to 0.8118. The same effect can be observed for the *churn* classifier, and although the AUC score for the *appetency* classifier increases in line with accuracy, it increases by a much smaller margin.

Furthermore, by increasing the training time by at least 200 epochs for each classifier, by going from an early stopping patience of 20 to 200, the AUC scores either got worse or only improved marginally (by only 0.0009 for *appetency*). This despite significant accuracy increases. Clearly, using validation accuracy as an early stopping metric for this dataset is not optimal, and unfortunately Keras does not currently support AUC score for this use. But what might be more important than being able to 'reign in' the training of a neural network pursuing

greater accuracy, by stopping training at the point the AUC score stops improving, would be to design a neural network that actually pursued the maximization of the AUC score directly. A means to solve this problem could be the incorporation of the AUC score into a loss function. For instance, Zhao et al [57] have developed two complex algorithms for online AUC maximization for applications where data are unevenly distributed among different classes. They address this challenge by exploiting the reservoir sampling technique, and present extensive experimental studies that confirm the effectiveness and the efficiency of their proposed algorithms for maximizing AUC [57]. However, the further investigation of such an approach is not currently supported by either Keras or TensorFlow, and the customization and incorporation of a novel loss function is beyond the scope of this project and can instead be considered for future research. It seems apparent that further refinement of the neural network architecture for the classifiers used in this challenge is misguided when the networks are fundamentally not optimizing the desired outcome. Further training, without over fitting, only serves to decrease the AUC score. This perhaps explains why the top scoring teams in the original KDD Cup 2009 competition succeeded with other ensemble machine learning algorithms.

V. WAYS OF WORKING

Both team members are distance learning students, one based in the UK and the other in Switzerland. We used the team communication platform Slack [58] extensively from day one to open a simple and accessible channel of communication, where we discussed daily the problem domain, shared updates, exchanged information and discussed final editing. Bi-weekly Skype calls started with a round-table task activity update followed by more informal discussions on our ideas in tackling the challenge. We maintained a list of tasks with assigned resources and deadlines, which made very transparent who was working on what and by when completion was targeted.

Appendix A shows our list of project activities in Gantt chart format.

VI. CONCLUSION

Our highest performing solution for the benchmarking 2009 KDD Cup Orange Challenge with the 'small' dataset was to use a gradient boosting classifier on a dataset with missing numerical values imputed with an arbitrary -9876 value, missing categorical values imputed with 'missing', creation of categorical binary dummy variables, and no pre-training normalization. In descending order of AUC score performance, the other solutions were voting classifier, boosting, bagging, random forest, decision trees, AdaBoost, and, finally, neural networks. Although the classifiers performed well, they fall short of the winning entries in the original competition. The reasons for this are likely varied, but undoubtedly rest on a different level of prior experience with machine learning techniques applied to large, messy datasets. It is clear that the ensemble selection approaches used by the winning team from IBM Research [3], in which a library of 500-1000 classifiers were used to generate and ensemble a wide variety of models, is going to lead to greater performance than the few types of solutions used here, regardless of the amount of hyperparameter tuning and implementation of varying components. Both authors are new to the APIs used (one to Python entirely) and the vast majority of methods employed in this project. So from the perspective of this as a learning experience, we both agree that it has been an extremely beneficial undertaking to have collaborated on.

REFERENCES

[1] "Analytics, Data Science, Machine Learning Poll 2017," [Online]. Available: https://www.kdnuggets.com/2018/04/poll-analytics-data-science-ml-applied-2017.html. [Accessed 12 April 2018].

[2] G. L. a. M. Berry, Data Mining Techniques, 3rd Edition ed., Indianapolis, IN: Wiley, 2011.

[3] I. Guyon, V. Lemaire, M. Boullé, G. Dror and D. Vogel, "Design and analysis of the KDD cup 2009: fast scoring on a large orange customer database," *ACM SIGKDD Explorations Newsletter,* vol. 11, no. 2, pp. 68-76, 2009.

[4] V. L. M. Labs, "KDD Cup 2009 Submission Results," [Online]. Available: http://www.vincentlemaire-labs.fr/kddcup2009/#results.

[5] T. Hastie, "Boosting," 2003. [Online]. Available: http://web.stanford.edu/~hastie/TALKS/boost.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher and M. Perrot, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* pp. 2826-2830, 2011.

[7] "Interactive: The Top Programming Languages 2017," [Online]. Available: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017.

[8] "Scikit-learn,org," [Online]. Available: http://scikit-learn.org/stable/.

[9] E. Tav, A. Borisov, G. Runger and K. Torkkola, "Feature Selection with Ensembles, Artificial Variables, and Redundancy Elimination," *Journal of Machine Learning Research,* vol. 10, pp. 1341-1366, 2009.

[10] R. Quinlan, C4. 5: Programs for machine learning, Morgan Kaufmann, 1993.

[11] L. Breiman, Classification and Regression Trees, Chapman & Hall, 1984.

[12] "scikit-learn DecisionTreeClassifier," [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[13] C. Ferri, P. A. Flach and J. Hernández-Orallo, “Improving the AUC of Probabilistic Estimation Trees,” in *European Conference on Machine Learning*, 2003.

[14] “scikit-learn RandomForestClassifier,” [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[15] T. K. Ho, “The Random Subspace Method for Constructing Decision Forests,” *IEEE Transactions On Pattern Analysis And Machine Intelligence,* vol. 20, no. 8, 1998.

[16] L. Breiman, “Random Forests,” *Springer Machine Learning,* vol. 45, no. 1, pp. 5-32, 2001.

[17] “scikit-learn BaggingClassifier,” [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html.

[18] “scikit-learn AdaBoostClassifier,” [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html.

[19] Y. Freund and R. E. Schapire, “A Short Introduction to Boosting,” *Journal of Japanese Society for Artificial Intelligence,* vol. 14, no. 5, pp. 771-780, 1999.

[20] “scikit-learn GradientBoostingClassifier,” [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html.

[21] J. H. Freidman, “Greedy Function Approximation: A Gradient Boosting Machine,” in *IMS 1999 Reitz Lecture*, 199.

[22] A. Niculescu-Mizil, C. Perlich, G. Swirszcz, V. Sindhwani, Y. Liu, P. Melville, D. Wang, J. Xiao, J. Hu, M. Singh, W. X. Shang and Y. F. Zhu, “Winning the KDD Cup Orange Challenge with Ensemble Selection,” in *JMLR: Workshop and Conference Proceedings 7*, 2009.

[23] “scikit-learn VotingClassifier,” [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html.

[24] A. A. Q. Ahmed and M. D, “Churn prediction on huge telecom data using hybrid firefly based classification,” *Egyptian Informatics Journal,* vol. 18, pp. 215-220, 2017.

[25] I. Myrtveit, E. Stensrud and U. H. Olsson, “Analyzing Data Sets with Missing Data: An Empirical Evaluation of Imputation Methods and Likelihood-Based Methods,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 27, no. 11, pp. 999-1013, 2001.

[26] W. Akinfaderin, “Missing Data Conundrum: Exploration and Imputation Techniques,” [Online]. Available: https://medium.com/ibm-data-science-experience/missing-data-conundrum-exploration-and-imputation-techniques-9f40abe0fd87.

[27] “imbalanced-learn,” [Online]. Available: http://contrib.scikit-learn.org/imbalanced-learn/stable/over_sampling.html#a-practical-guide.

[28] V. Nikulin and G. J. McLachlan, “Classification of Imbalanced Marketing Data with Balanced Random Sets,” in *JMLR: Workshop and Conference Proceedings 7*.

[29] J. V. Hulse, T. M. Khoshgoftaar and A. Napolitano, “Experimental Perspectives on Learning from Imbalanced Data,” in *Proceedings of the 24th international conference on Machine learning*, Corvalis, Oregon, USA, 2007.

[30] R. Busa-Fekete and B. Kegl, “Accelerating AdaBoost using UCB,” in *JMLR: Workshop and Conference Proceedings 7*, 2009.

[31] Wai-Ho Au, K. C. C. Chan and X. Yao, “A Novel Evolutionary Data Mining Algorithm With Applications to Vhurn Prediction,” *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION,* vol. 7, no. 6, pp. 532-545, 2003.

[32] W. L. e. al., “A survey of deep neural network architectures and their applications,” *Neurocomputing,* vol. 234, no. 11–26, December 2016.

[33] Z. T. e. al, “A joint residual network with paired ReLUs activation for image super-resolution,” *Neurocomputing,* no. 273, pp. 37-46, 2018.

[34] e. a. Y. LeCun, “Deep Learning,” *Nature (London),* no. 7553, pp. 436-444., May 2015.

[35] F. Chollet, “Keras,” GitHub Repository, [Online]. Available: https://github.com/keras-team/keras.

[36] e. a. M. Abadi, “TensorFlow,” 2015. [Online]. Available: www.tensorflow.org.

[37] A. Bojanowska, “Application of neural networks in CRM systems,” in *CMES’17*, Lublin, Poland, 2017.

[38] A. K. e. al, “ImageNet Classification with Deep Convolutional Neural Networks,” in *NIPS*, 2012.

[39] “KDD Cup 2009 - Data,” KDD, 2009. [Online]. Available: http://www.kdd.org/kdd-cup/view/kdd-cup-2009/Data.

[40] K. J. a. L. Xinzhe, “Empirical analysis of optimal hidden neurons in neural network modeling for stock prediction,” in *Proceedings of the Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, 2008.

[41] J. Schmidhuber, “Deep Learning in Neural Networks: An Overview,” *Neural networks : the official journal of the International Neural Network Society,* vol. 61, pp. 85-117, January 2015.

[42] L. B. Y. B. a. P. H. Y. Lecun, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[43] “PReLU,” [Online]. Available: https://arxiv.org/pdf/1502.01852.pdf.

[44] X. G. e. al, “Deep Sparse Rectifier Neural Networks,” in *AISTATS*, ort Lauderdale, FL, USA, 2011.

[45] P. R. e. al, “Swish: A Self-Gated Activation Function,” Google Brain Residency program, 2017. [Online]. Available: https://arxiv.org/pdf/1710.05941.pdf.

[46] P. T. e. al, “Learning Multi-Instance Deep Discriminative Patterns for Image Classification,” *IEEE Transactions on Image Processing,* vol. 26, no. 7, 2017.

[47] Y. Nesterov, “A method for unconstrained convex minimization problem with the rate of convergence,” *Doklady ANSSSR,* vol. 269, no. http://www.cis.pku.edu.cn/faculty/vision/zlin/1983-A%20Method%20of%2, pp. 543-547, 1983.

[48] D. K. a. J. Ba, “Adam: A Method for Stochastic Optimization,” in *International Conference on Learning Representations*, 2014.

[49] G. H. -. U. o. T. T Tieleman, “RMSProp: Lecture 6.5-RMSProp, Technical Report,” COURSERA: Neural networks for machine learning, 2012. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[50] L. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1,” Mar 2018. [Online]. Available: https://arxiv.org/pdf/1803.09820.pdf.

[51] N. S. e. al, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research,* vol. 15, pp. 1929-1958, 2014.

[52] J. B. e. al, “Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures,” in *Proc. of the 30th International Conference on Machine Learning*, 2013 .

[53] S. N. a. G. Hinton, “Simplifying Neural Networks by Soft Weight-Sharing,” *Neural Computation,* vol. 4, pp. 473-493, 1992.

[54] S. L. a. C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” in *In ICML*, 2015.

[55] X. G. a. Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *13th AISTATS 2010*, Chia Laguna Resort, Sardinia, Italy., 2010.

[56] N. C. e. al, “SMOTE: Synthetic Minority Over-sampling Technique,” *Journal of Artificial Intelligence Research,* vol. 16, p. 321–357, 2002.

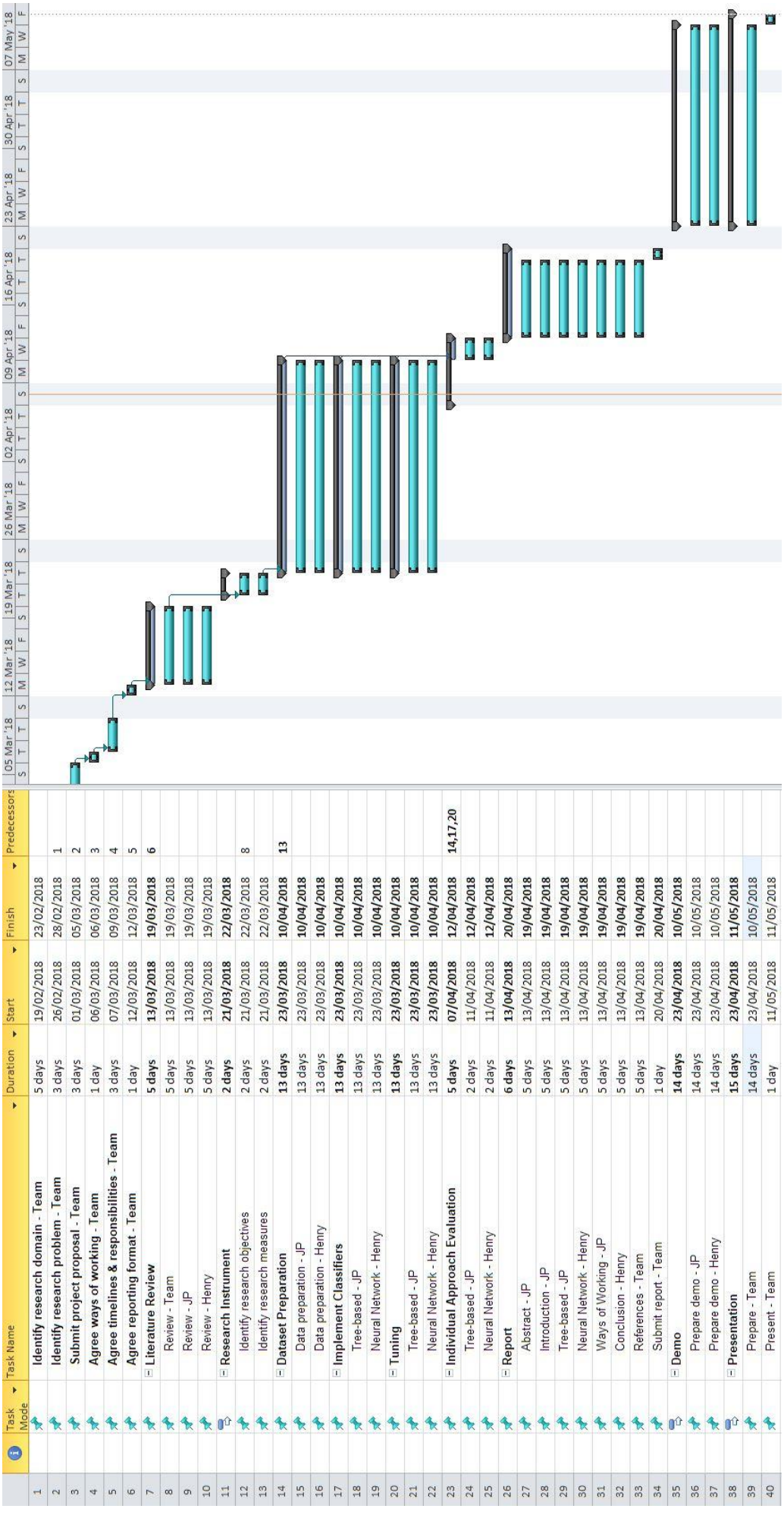
[57] P. Z. e. al, “Online AUC Maximization,” in *28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011.

[58] “Slack,” [Online]. Available: http://slack.com.

[59] A. P. Bradley, “The use of The Area Under The Curve in the evaluation of machine learning algorithms,” *Pattern Recognition,* vol. 30, no. 7, pp. 1145-1159, 1997.

[60] J. D. e. al, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” *Journal of Machine Learning Research ,* vol. 12, pp. 2121-2159, July 2011.

APPENDIX A – PROJECT ACTIVITIES GANTT CHART



	Task Mode	Task Name	Duration	Start	Finish	Predecessors
1		**Identify research domain - Team**	5 days	19/02/2018	23/02/2018	
2		**Identify research problem - Team**	3 days	26/02/2018	28/02/2018	1
3		**Submit project proposal - Team**	3 days	01/03/2018	05/03/2018	2
4		**Agree ways of working - Team**	1 day	06/03/2018	06/03/2018	3
5		**Agree timelines & responsibilities - Team**	3 days	07/03/2018	09/03/2018	4
6		**Agree reporting format - Team**	1 day	12/03/2018	12/03/2018	5
7		**Literature Review**	**5 days**	**13/03/2018**	**19/03/2018**	**6**
8		Review - Team	5 days	13/03/2018	19/03/2018	
9		Review - JP	5 days	13/03/2018	19/03/2018	
10		Review - Henry	5 days	13/03/2018	19/03/2018	
11		**Research Instrument**	**2 days**	**21/03/2018**	**22/03/2018**	
12		Identify research objectives	2 days	21/03/2018	22/03/2018	8
13		Identify research measures	2 days	21/03/2018	22/03/2018	
14		**Dataset Preparation**	**13 days**	**23/03/2018**	**10/04/2018**	**13**
15		Data preparation - JP	13 days	23/03/2018	**10/04/2018**	
16		Data preparation - Henry	13 days	23/03/2018	**10/04/2018**	
17		**Implement Classifiers**	**13 days**	**23/03/2018**	**10/04/2018**	
18		Tree-based - JP	13 days	23/03/2018	**10/04/2018**	
19		Neural Network - Henry	13 days	23/03/2018	**10/04/2018**	
20		**Tuning**	**13 days**	**23/03/2018**	**10/04/2018**	
21		Tree-based - JP	13 days	**23/03/2018**	**10/04/2018**	
22		Neural Network - Henry	13 days	**23/03/2018**	**10/04/2018**	
23		**Individual Approach Evaluation**	**5 days**	**07/04/2018**	**12/04/2018**	**14,17,20**
24		Tree-based - JP	2 days	11/04/2018	**12/04/2018**	
25		Neural Network - Henry	2 days	11/04/2018	**12/04/2018**	
26		**Report**	**6 days**	**13/04/2018**	**20/04/2018**	
27		Abstract - JP	5 days	13/04/2018	**19/04/2018**	
28		Introduction - JP	5 days	13/04/2018	**19/04/2018**	
29		Tree-based - JP	5 days	13/04/2018	**19/04/2018**	
30		Neural Network - Henry	5 days	13/04/2018	**19/04/2018**	
31		Ways of Working - JP	5 days	13/04/2018	**19/04/2018**	
32		Conclusion - Henry	5 days	13/04/2018	**19/04/2018**	
33		References - Team	5 days	13/04/2018	**19/04/2018**	
34		Submit report - Team	1 day	20/04/2018	**20/04/2018**	
35		**Demo**	**14 days**	**23/04/2018**	**10/05/2018**	
36		Prepare demo - JP	14 days	23/04/2018	10/05/2018	
37		Prepare demo - Henry	14 days	23/04/2018	10/05/2018	
38		**Presentation**	**15 days**	**23/04/2018**	**11/05/2018**	
39		Prepare - Team	14 days	23/04/2018	10/05/2018	
40		Present - Team	1 day	11/05/2018	11/05/2018	