Poker Hand Induction: Multi-Class Classification of Extreme Imbalanced Data with Machine Learning

Thomas Angeland (151433)[a], Kim-Andre Engebretsen (151471)[a], Mustafa Numic (151867)[a]

[a]*Østfold University College, B R A Veien 4, Halden 1783, Norway*

Abstract

Class imbalance is a crucial problem in machine learning and is found in many domain-specific applications. More specifically, the binary class imbalance problem (i.e. datasets with two classes) has received interest from researchers in recent years, and various solutions have been proposed for real-world tasks like credit card fraud detection, oil spill detection and tumour discovery, amongst others. On the other hand, less research has been done in regards for handling varying degrees of class imbalance in datasets consisting of more than two classes. For imbalanced multi-class datasets, the classification model often miss-classifies minority classes (i.e. classes with less samples) as majority classes (i.e. classes with more samples), leading to poor predictive accuracy. While there exists proposed method and techniques for multi-class classification of imbalanced data, there is still a lack of research on extreme cases of class imbalance (e.g. one every half a million others). In this paper, we attempt to try out a variety of different methods where some are known for handling imbalanced data pretty well. These methods will initially be tested on the rather popular Poker Hands dataset from UCI Machine Learning Repository. We will explain how this dataset is not good for improving the performance of our selected techniques and how that motivated us to create a new poker hands dataset. Our goal for the new dataset will be to have a more balanced set of data containing more of the rare occurring classes and less of the more common occurring classes. This new dataset will be used in an attempt to improve the methods we will be using in this project.

Keywords: machine learning, deep learning, multi-class classification, imbalanced data, poker

1. Introduction

Substantial research has been conducted in order to find new, more intuitive ways to deal with imbalanced data in imbalanced machine learning [1]. In binary learning, imbalanced data is a common issue in many practical applications such as medical diagnosis [2], product defect identification [3], credit card fraud detection [3], among others [1]. To make matters more complex, multi-class imbalanced data classification is shown to be more challenging compared to binary imbalanced learning [4].

In a traditional five hand poker game, each player will be assigned 5 cards from a shuffled card deck of 52 unique cards. Based on what kind of cards the player receives, a type of hand is then naturally assigned based on multiple criterion. There are 10 distinct, ranked types of hands. The player with the best type of hand wins the game.

When order matters, there are 311 875 200 unique poker hands. The probabilities of each type of hand are quite imbalanced. The most likely type of hand a player will have at hand when cards are dealt is either nothing (50.12%, also known as "high card") or a single pair (42.26%), while the least likely type of hand is a royal flush (0.000154%). Probabilistically, that is one royal flush for every 649 740 hand dealt.

The probabilities of poker and it's extreme imbalanced nature is the main focus of our work. The challenge is not really about creating a system that can learn how to rank poker hands as this could be done with simple boolean expressions. The challenge is to create a system that recognises something it sees only a few times over millions of samples.

In this paper, we compare six well-established decision tree algorithms, two types of neural networks and three different machine learning algorithms provided by the sci-kit learning kit for multi-class classification. These classifiers are tasked to assess poker hands of 5 cards from a 52 card deck. The poker hand set is naturally extremely imbalanced and therefore an interesting problem to use in our testing. In Table 15, we summarise the top score we were able to achieve with each of these methods. In addition, we will also provide a new poker hand dataset that contains more samples and that is accurately depicting the real world. Lastly, we give our recommendations for future work.

2. Related work for multi-class imbalance

Various methods and techniques have been proposed to address imbalanced data classification [1, 5, 6, 7, 8, 9]. These can be categorised into several groups:

1. Data collection: The ways in which data is collected.
2. Data augmentation methods: In which the dataset is modified in a way to reduce class imbalance.

Email addresses: thomas.angeland@hiof.no (Thomas Angeland (151433)), kim.a.engebretsen@hiof.no (Kim-Andre Engebretsen (151471)), mustafa.numic@hiof.no (Mustafa Numic (151867))

3. Class and sample weights: In which the samples are given a level of importance (i.e. a weight) and prioritisation.
4. Loss functions: A function which is to be minimised (or maximised) when training a model.
5. Validation metrics: A score value used to validate the predictive performance of the model for imbalanced data classification.
6. Hyperparameter tuning: In which existing algorithm is modified and tuned with a set of provided parameters in order to achieve the best loss value for imbalanced data.

2.0.1. Data augmentation

Data augmentation methods for imbalanced data are mainly divided into three approaches: Oversampling technique, undersampling technique and hybrid technique. These approaches can be used to alter the class distribution of the dataset. The oversampling approach increases the amount of samples in the minority classes by creating new samples (e.g. by random duplication) in order to balance the dataset. The undersampling approach on the other hand, randomly deletes samples from the majority classes in order to obtain class balance. The hybrid technique combines oversampling and undersampling by applying oversampling to one or more minority classes, as well as applying oversampling to one or multiple majority classes.

While these methods have proven to increase performance in imbalanced class classification [5], they still have some drawbacks. For undersampling, the disadvantage is that it discards potentially useful data. For oversampling, duplicate data provides no more detail to the dataset (i.e. the representation of the population) and may result in overfitting. To address these issues, Synthetic Minority Oversampling Technique (SMOTE) [6] is used. SMOTE is a powerful algorithm that is specifically made to deal with imbalanced datasets, and has shown success in various domain-based applications like web spam classification [7] and credit evaluation [8]. The SMOTE algorithm is an oversampling technique that adds synthetic minority class samples to the original dataset in order to reduce class imbalance. This has shown to be very usable in the case when the minority classes have limited amounts of samples [1].

Depending on the oversampling required, a number of nearest neighbours are randomly chosen for each minority class. The synthetic data is generated by interpolation. For subset ϵS, consider K nearest neighbours for each sample ϵX. The K-nearest neighbours are the K elements whose Euclidean distance between itself and have smallest weight along the n-dimensional feature space of X. The samples are generated simply as, randomly select one of the K-nearest neighbours and multiply corresponding Euclidean distance with random number between $[0, 1]$. Finally, this value is added to the original value. The mathematical formula is defined as

$$Synthetic(X_i) = X_i + X\left(\hat{X}_i - X_i\right) \times \delta, \tag{1}$$

where X_i is the sample from minority class used to generate synthetic data. $\hat{X}_i$ is the nearest neighbour. δ is a random number between $[0, 1]$. The generated synthetic data is a point on line fragment between under consideration and k-nearest neighbours for X_i.

2.0.2. Weights

Another method for accounting for class imbalance in imbalanced data classification, is assigning weight-values (cost) to each sample (or each class). Weights, or penalised classification, imposes an additional cost on the model for making classification mistakes on the minority class during training, making the model pay more attention to the minority class [10]. These weight-values indicate a level of importance or influence a sample or class has on the loss value. The higher the weight value, the more emphasis the classifier puts on predicting the class correctly. In other words, weights change the way loss values are calculated. A weighted loss function multiplies the loss output with the weight value for a given sample or class. Truic and Leordeanu [9] reports that using weights yielded good results with imbalanced data. This was tested on three different decision trees where all benefited from using weights. In addition, Anand et al. [5] proposed a method that used both data augmentation and weights which also returned good results.

2.0.3. Loss functions

In combination with weights, Categorical Cross Entropy (CCE) [11] is an accessible class probability metric and loss function for both binary and multi-class classification that is easy to interpret and sometimes a default option for various multi-class classifiers. While there has been proposed different loss functions for class imbalance recently, like Focal Loss by Zhang and Sabuncu [11], we will limit our research to CCE for now. The equation for CCE is

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} 1_{y_i} \in C_c \log\left(P_{\text{model}}\left[y_i \in C_c\right]\right) \tag{2}$$

where the double sum is over the observations i, whose number is N, and the classes c, whose number is C. $1_{y_i \in C_c}$ is a therm for the indicator function of sample i^{th} belonging to class c^{th}. The $p_{model}[y_i \in C_c]$ is the probability predicted by the classifier for the ith sample to belong to the cth class. For multi-class classification (i.e. for more than two classes), the CCE outputs a vector C probabilities, where each component contains a probability as well as an index which indicates which class (by integer, $\{0, 1, 2, \ldots, C\}$) the probability is for. The sum of every probability component adds up to 1.0.

Conventionally, loss functions are applied to sets of predictions generated by the classifier from sets of testing samples, and the probability vectors are returned as matrices of probability components (or as two-dimensional arrays). By using a simple argmax-function over every row of the matrix (or sub-array), which returns the index of the highest probability component, the result is a list of predicted classes which can compared to the list of observed classes. The predicted and observed classes can be used to create a confusion matrix of size $C \times C$ which contains the accuracy for observed classes versus predicted classes.

2.0.4. Validation metrics

According to Li et al. [12], accuracy is not a good indicator for multi-classification on imbalanced data. For instance, a

dataset that contains three classes, where the sample distribution for the first, second and third class is 70%, 20% and 10%, respectively, a classifier that only predict the first class for all samples will score a 70 percent accuracy. On paper, 70 percent does seem like a high amount, but this is not a good indication for how well the classifier can predict every class. Instead of using accuracy as a way to describe the predictive performance of a given classifier, Li et al. [12] suggests using the Geometric Mean (GM)[13] for imbalanced class classification. GM is one of the most frequently used metrics for imbalanced learning [1]. It reflects the ability to classify positive samples and negative samples at the same time. The formula for GM is defined as

$$G_{mean} = \sqrt{R_{t+} \cdot R_{t-}}, \tag{3}$$

where R_{t+} represents true positive rate, which is calculated by $R_{t+} = N_{TP}/N_P \cdot R_{t-}$ and represents true negative rate, which is formulated as $R_{t-} = N_{TN}/N_N$. The higher the GM, the better the ability of the classifier to recognise positive class samples and negative class samples at the same time.

3. Proposed methods for poker induction

To our best effort, we failed to find an appropriate amount of publicly available peer-reviewed literature with proposed methods for the poker hand problem. The single peer-reviewed articles found was published by Jabin [14], who proposed three feed-forward back-propagating network models created with MATLAB. The networks uses the Mean Squared Error (MSE) and are respectively trained with Resilient backpropagation (RBP, trainrp) [15], Levenberg-Marquardt backpropagation (LMBP, trainlm) [16], and scaled conjugate gradient backpropagation (SCGBP, trainscg) [17]. Jabin also proposed a Self-Organising Map (SOM) [18] model with random weight-bias rule, which is a type of neural network that is trained using unsupervised learning to produce a low-dimensional representation of the input space of the training samples, called a map. The poker hand data was augmented with a binary encoding scheme and the best result was achieved with a feed-forward scaled conjugate back-propagating neural network three hidden layers which yielded 95% accuracy.

On the other hand, there have been published multiple web articles by practitioners that address the problem, some with multiple methods with code, different techniques and parameters, as well as performance metrics and claimed test results. In regards to the sparse peer-reviewed literature, we will list some of these that we found interesting for the poker hand problem.

First off is Sihota [19], who used several different techniques to find out which one yielded the best results in terms of accuracy. He attempted to deal with the poker hands dataset using different types of machine learning models: Decision Trees with pruning, Neural Networks, Boosted Trees, Support Vector Machines (SVM) and K-Nearest Neighbours. With decision trees, he managed to get an accuracy of 54% and this was the worst result out of all the techniques that he tried. The tree was created using recursive partitioning and was pruned using complexity parameter with minimum error to shrink it. With neural networks, he achieved 57% accuracy using 6 hidden layers and 100 iterations. With SVM, Sihota managed to score $\approx$ 100% accuracy using two different kernel functions, Radial and Linear with the parameters: 10-fold cross validation and 0.1 gamma cost. With AdaBoost using classification trees and single classifiers he managed to get an accuracy of 65%. And lastly with k-nearest Neighbours he managed to score an accuracy of 70%.

Another method proposed was by Xu [20], who used the poker hand dataset to get a better understanding of genetic algorithms through experiments to build a predictive model on real data. Wenjie mentions that the problem of poker rule inductions is based on the set of simulated poker hands. This was brought up by a prediction competition hosted on Kaggle.com. The competition description states that it is potentially difficult to discover rules that can correctly classify poker hands and the algorithm will need to find rules that are general enough to be broadly useful, without being so broad that they end up being occasionally wrong. The essence of poker rules lies in the ability to recognise and compare poker hands. Wenjie attempted to tackle this problem by developing an algorithm based on the concepts of genetic algorithms to train the computer to predict the class label for a given poker hand as accurately as possible. Since he was interested in discovering rules of poker games, he used decision trees to represent the rules since tree structures are suitable for applying genetic algorithms and can be easily encoded into strings of operations with connected nodes and leaves. Wenjie came to the conclusion that his genetic algorithm may not be a good model in terms of predictive accuracy, with a sample size of 1000 and a population 400 the best result he managed to get was 47.9% accuracy.

Hamel [21] attempted the competition by Kaggle.com using Random Forest and managed to get an accuracy of 76.9% by increasing the number of trees to 2000 in his model. Hamelg concludes that generating new training examples or increasing the number of trees to more than 2000 in his model could improve accuracy even further.

Dişken [22] attempted to produce a neural network system to classify the cards into hands using the poker hands dataset. This was done in MATLAB using the Neural Networks Toolbox. For the first training attempt, the number of hidden neurons Dişken used was 130 as he stated this was suitable for large amount of data. Further, the number of epochs was 1000, 0.01 as learning rate parameter, resilient backpropagation as the training function and transig as the transfer function. The data was split into 80% training, 10% validation, and 10% testing. The result showed that as iterations went on, the error was getting smaller. The overall results showed an accuracy of 92.4% was achieved and by looking at the confusion matrix, one could quickly notice that the system did really well with the first class as it achieved 98.9% correct classification which played a huge role in the overall achieved accuracy. At the end it was concluded that the examples shown in the paper hints that a small number of hidden neurons may not produce acceptable results. For future improve-

ment, the next step would be to change or adjust parameters such as learning rate, neuron numbers etc. and only one at a time to see the effect of the change.

Garg [23] created a neural network system as well to predict poker hands. Akash initially states that a simple network system with 10 inputs, 1 output and a few hidden layers might suffice. Further he explains that, one might expect that 2 hidden layers with 20 neurons would be enough for this task but it turned out to not perform so well. Even by increasing the neurons up to 100 and using 7 layers it still performed very bad. Akash then attempted to tackle the problem with some feature engineering. The rank of a card can be represented by a vector with 13 zeros and ones. Similarly for suits we can have a vector with the length of 4. Each card has 17 features which means a set of cards can be represented by a vector of length 85. Then Akash did the same transformation for the output, the output can take 10 values as input and thus can be represented with a vector of length 10. He then created a neural network using the following parameters: 1 input layer with 85 neurons, 2 hidden layers where the first layer consists of 18 neurons and the second layer consisting of 10 neurons, and lastly, 1 output layer with 10 neurons. With this neural network, Akash managed to score an accuracy of 94%.

Noor Ali [24] did a research project on the poker hand dataset where the main goal was to analyze poker hands and predict strength using a neural network system. In the preperation process, Sahir thought at first to reduce the number of features from the dataset and applied a Random Forest Classifier to print out and get an overview over the importance of each feature. The result clearly showed that each feature is very important as each one contributes to more than 5% to the result. This relates to a practical scenario where we cannot remove a card or ignore the cards suits when determing the strength of the hand. Therefore, attempting to reduce the number of features will heavily damage the dataset. Before executing the model Sahir scaled the data using StandardScalar preprocessing before applying an MLPClassifier. The reason for this is because the MLPClassifier is sensitive to feature scaling. The model for this project consisted of a FeedForwardNet Neural Network using the trial and error approach to converge to good results. The neural network was created with the following parameters: 1 input layer consisting of 10 neurons, 2 hidden layers where both layers consists of 64 neurons and 1 output layer consisting of 10 neurons. Sahir made sure to avoid overfitting by using an equation which calculates the amount of neurons needed in a single hidden layer. Since he didnt achieve a good accuracy using a single hidden layer, he instead went for 2 hidden layers and using far less neurons for the layers than what he had initially calculated. Thus avoiding overfitting. The model was executed 5 times for randomization using a learning rate of 0.02 and a maximum of 2000 iterations. With this, Sahir managed to score an accuracy of 96%. To compare his results with other well known machine learning algorithms, he did a quick test by running the dataset with a few algorithms and managed to get these results: 59% with Bagging, 57% with Random Forest, 49% with AdaBoost, 55% with k-nearest neighbour, 50% with Naive Bayes, 48% with Decision Trees, 50% with Linear Support Vector Machine and Lastly 60% with OutputCodeClassifier using Linear Support Vector Machine.

4. Selected models and theory

We have selected all in all 11 different machine learning algorithms for multi-classification of imbalanced poker hand data. Of these, six are decision trees, namely C5.0 [25], AdaBoost Freund and Schapire [26], Random Forest [27], XGBoost Chen and Guestrin [28], LightGBM [29] and CatBoost [30]; two are neural networks, namely feedforwardnet from MATLAB [31] and a Keras model from TensorFlow [32]; one linear support vector machine approach [33]; one Naive Bayes method [34]; and lastly, one K-Nearest Neighbour algorithm [35].

4.0.1. K-Nearest Neighbour

K-Nearest Neighbour (KNN) is a simple supervised machine learning algorithm, where the raw training instances are used to make predictions. The KNN algorithm assumes that similar objects exists close to each other. The algorithm uses mathematics to determine if two objects are the same, this can be done in many different ways but the most popular are the euclidean distance, Hamming distance, Manhattan distance and Minkowski distance. The advantages to using this simple algorithm is that it is simple and easy to implement and no need for parameter tuning and it can be used for classification or regression and search. But it has one disadvantage, it gets significantly slower as the number of instances or predictors increase.

The Euclidean distance:

$$P(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} \quad (4)$$

The Euclidean distance is the length of the line segment connecting two points.

4.0.2. Naive Bayes

In the sci-kit learning kit we have access to a set of supervised learning algorithms based on applying Bayes theorem [34]. These sets of algorithms are called Naive Bayes methods. Naive Bayes is one of the most efficient and effective inductive learning algorithms for machine learning and data mining. Naive Bayes is the simplest form of Bayesian network, in which all attributes are independent given the value of the class variable. This is called conditional independence. The classification performance of Naive Bayes is surprisingly good despite the fact that the conditional independence assumption on which naive bayes is based upon, is rarely true in real world applications.

From Scikit learn's documentation Bayes theorem states the following relationship:

$$P(y|x_1,\ldots,x_n) = \frac{P(y)P(x_1,\ldots,x_n|y)}{P(x_1,\ldots,x_n)} \quad (5)$$

where y is a class variable and x_1 as a dependent feature vector through x_n. Now we can use the naive conditional independence which assumes the following:

$$P(y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y) \tag{6}$$

For all i, this can be simplified to:

$$P(y|x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, \ldots, x_n)} \tag{7}$$

As $P(x_1, \ldots, x_n$ is constant given the input, the following classification rule can be used:

$$P(y|x_1, \ldots, x_n)\alpha P(y)\prod_{i=1}^{n} P(xi|y) \tag{8}$$

$$\hat{y} = \underset{y}{argmax} P(y)\prod_{i=1}^{n} P(xi|y) \tag{9}$$

According to sci-kit learning documentation, Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods and is famously known for document classification and spam filtering.

In a written report by Zhang [34] where he studied the supposedly superb classification performance of Naive Bayes, he came to the conclusion that, in a given dataset, two attributes may depend on each other but The dependence may distribute evenly in each class. In his case however, even though the conditional independence assumption was violated, Naive Bayes still proved to be the best classifier. There may exist strong dependence between two attributes that affects the classification. However, if the dependencies among all attributes work together, they may cancel each other out and wont affect the classification anymore. From this, we could argue that what eventually affects the classification of Naive Bayes, is the distribution of dependencies among all attributes over classes, and not the dependencies themselves.

In this project, we will create two models using the sci-kit learn toolkit. One model based upon Gaussian Naive Bayes, and one model base upon Complement Naive Bayes. See chapter 5.

4.0.3. Support Vector Machines

Support Vector Machines (SVMs) [33] are supervised learning models used for classification, regression and other tasks. The algorithm learns by constructing a hyper-plane or set of hyper-planes in an infinite dimensional space, and it differentiates from general linear regression algorithms in the sense that it tries to find the hyper-planes that have the largest distance to the nearest training data. In other words, with maximum margin and minimum miss-classification error. In a multi-class classification task, the classifier separates multiple classes with multiple hyper-planes.

In the following, we describe the formal, binary definition of SVMs. Specifically, a training sample x_i is paired with an integer $y_i \in \{+1, 1\}$ as its class. A positive and negative instance is a training instance with the class of +1 and -1, respectively. Given a set χ of n training samples, the goal of training a SVM classifier is to find a hyperplane that separates the positive and negative classes in χ. The training is equivalent to solving the following optimisation problem:

$$\begin{aligned} \underset{\omega,\xi,b}{\arg\max} \quad & \frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i\left(\omega \cdot x_i - b\right) \geq 1 - \xi_i \\ & \xi_i \geq 0, \forall i \in \{1, \ldots, n\} \end{aligned} \tag{10}$$

where ω is the normal vector of the hyperplane, C is the penalty parameter, b is the bias of the hyperplane, and ξ are variables to tolerant some training instances falling in the wrong side of the hyperplane.

The classifier uses a mapping function that maps the training samples from the original data space to a higher dimensional data space, which makes non-linear data become linearly separable. Equation 10 can be rewritten to a dual form where the dot products of two mapped training samples become replaceable by a kernel function. Because only dot products are involved, this avoids explicitly defining the mapping functions. The optimisation problem in the dual form is defined as

$$\begin{aligned} \max_{\alpha} \quad & F(\alpha) = \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\alpha^T Q\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \forall i \in \{1, \ldots, n\} \\ & \sum_{i=1}^{n} y_{ii} = 0 \end{aligned} \tag{11}$$

where $F(\alpha)$ is the objective function. $\alpha \in \mathbb{R}^n$ is a weight vector, where α_i is the *weight* of x_i. C is the penalty parameter. Q is a positive matrix, where $Q = \left[Q_{ij}\right]$, $Q_{ij} = y_i y_j K\left(x_i, x_j\right)$ and $K\left(x_i, x_j\right)$ is a kernel value computed from a kernel function. Collectively, every kernel value forms an $n \times n$ kernel matrix.

4.0.4. AdaBoost

AdaBoost is a boosted tree algorithm proposed by Freund and Schapire [26] in 1997. It can solve multi-class classification problems, but in a restricted way by reducing the problem to multiple two-class problems (binary classification), comparing one class against all the others. The amount of classes in a given dataset is equal to the amount of one-versus-the-rest (OVR) binary classifiers for solving the problem. In response to this issue, Hastie et al. [36] proposed an extension of the AdaBoost algorithm that does not reduce the multi-class case to multiple two-class problems. They propose two versions called *SAMME* and *SAMME.R* Hastie et al. [37] which combines weak classifiers that requires the predictive performance of each classifier to be better than randomised guessing.

The AdaBoost algorithm approximates the Bayes classifier $C*(x)$. It builds a decision tree with unweighted training samples that produces the class labels for each sample. If the classifier missclassify a given training sample, the weight of that sample is increased (i.e. boosting). Then a second classifier is built based

on the new weights. This repeats for I iterations. Algorithm 1 shows each step in the AdaBoost algorithm proposed by Freund and Schapire [26].

Algorithm 1: AdaBoost (Freund and Schapire [26])

Input: X, Training data with $\{\{x_1, c_1\}, \{x_2, c_2\}, \{x_n, c_n\}\}$
Initialise the observation weights $w_i = 1/n, i = 1, 2, \ldots, n$
for $m = 1, 2, \ldots, M$ **do**
  Fit a classifier $T^{(m)}(x)$ to the training data using weights w_i
  Compute
  $\epsilon(m) = \sum_{i=1}^{n} w_i \mathbb{I}\left(c_i \neq T^{(m)}(x_i)\right) / \sum_{i=1}^{n} w_i$
  Set
  $w_i \leftarrow w_i \times \exp\left(\alpha^{(m)} \mathbb{I}\left(c_i \neq T^{(m)}(x_i)\right)\right), i = 1, 2, \ldots, n$
  Re-normalise w_i
end
Output: $C(x) = \arg\max_k \sum_{m=1}^{M} \left[\alpha^{(m)} \mathbb{I}\left(T^{(m)}(x) = k\right)\right]$

4.0.5. Random Forest

Random Forest (RF) is an ensemble learning method used for classification and regression [27]. It was developed by Breiman [27] in 2001, combining his bagging sampling approach, as well as the random selection of features, introduced independently by [38], [39] and [40], as a way to construct a set of decision trees with controlled variation. The decision trees in the ensemble are created using samples with replacement from the training set (i.e. *bagging*). Acting as a base classifier, each tree in the ensemble determine the class to a given sample through majority voting, where the most voted class is used to classify the sample. Algorithm 2 summarises the RF algorithm where N is the number of training samples and M is the number of classes in dataset.

Algorithm 2: Random Forest

Result: A vector of trees $\vec{RF}$
Input: N, S
for $i = 2, 3, 4, \ldots, N$ **do**
  Create an empty tree T_i
  repeat
    Sample S out of all features F using Bootstrap sampling
    Create a vector of the S features $\vec{F_S}$
    Find best split feature $B(\vec{F_S})$ Create a new node using $B(\vec{F_S})$ in T_i
  until *No more instances to split on*;
  Add T_i to the $\vec{RF}$
end

In the ensemble, each tree is created with some randomisation applied (e.g. with $\sqrt{F}$, where F is the number of predictors) to the selection of best nodes to split on. The classification and regression trees (CART) technique is used to add additional randomness to the construction of the trees, where the Gini index [41] is used to evaluate the subset of predictors selected in each node. The predictor with the highest Gini index, in our case the suit or rank of poker card, is selected as the split predictor in the node. In its general form, Gini is defined as

$$\text{Gini}(t) = 1 - \sum_{i=1}^{N} P(C_i|t)^2, \tag{12}$$

where t is a condition, N the total number of classes, and C_i is the i^{th} class label in the data set. The resulting index is used to measure the impurity of data (i.e. the level of uncertainty of the determination of the class label).

4.0.6. XGBoost

XGBoost, short for "eXtreme Gradient Boosting", was introduced by Chen and Guestrin [28] in 2014. XGBoost starts out by trying to determine the step directly by minimising the objective function. To prevent overfitting, a regularisation term is used to control the complexity of the model. It is defined as

$$\Omega = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{13}$$

where T is the number of leaves, and w_j^2 is the score at the j^{th} leaf. It is added to the loss function $\mathcal{L}$, which in multi-class classification is the multi logloss (i.e. CCE), and the result is the objective of the model defined as

$$Obj^{(t)} = \sum_{i=1}^{N} \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{i=1}^{t} \Omega\left(f_i\right) \tag{14}$$

where the loss function $\mathcal{L}$ (in our case, Equation 2) controls the predictive power, and the regularisation Ω controls the simplicity. Gradient descent is used to optimise the objective, which is an iterative technique that calculate

$$\delta_{\hat{y}} Obj\,(y, \hat{y}) \tag{15}$$

at each iteration. $\hat{y}$ is improved along the direction of the gradient to minimise the objective. The performance can be improved by using both the first and second order gradient defined as

$$\begin{aligned} &\delta_{\hat{y}^{(t)}} Obj^{(t)} \\ &\delta^2_{\hat{y}^{(t)}} Obj^{(t)} \end{aligned} \tag{16}$$

We then calculate the second order taylor approximation since there is not a derivative for every objective function. This is defined as

$$Obj^{(t)} \simeq \sum_{i=1}^{N} \left[\mathcal{L}\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t\left(x_i\right) + \frac{1}{2} h_i f_t^2\left(x_i\right)\right] + \sum_{i=1}^{t} \Omega\left(f_i\right)$$

where

$$\begin{aligned} g_i &= \delta_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right) \\ h_i &= \delta^2_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}_i^{(t-1)}\right) \end{aligned} \tag{17}$$

The objective at the t^{th} step can be defined as

$$Obj^{(t)} \simeq \sum_{i=1}^{N}\left[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\right] + \sum_{i=1}^{t}\Omega(f_i) \tag{18}$$

where the constant terms have been removed. The goal is to optimise (in our case, minimise) it by finding the f_t.

In order to build the tree structure, the way to assign prediction scores need to be formulated. We can mathematically define a tree as

$$f_t(x) = w_{q(x)} \tag{19}$$

where $q(x)$ is a function that assign every sample to the $q(x)^{th}$ leaf. The predictive process assigns the data point x to a leaf by q, as well as the corresponding score $w_{q(x)}$ on the $q(x)^{th}$ leaf to the sample. We then define the index set as

$$I_j = \{i|q(x) = j\} \tag{20}$$

which contains the indices of samples assigned to the h^{th} leaf. The objective can now be redefined as

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^{N}\left[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\right] + \gamma T + \frac{1}{2}\lambda\sum_{i=1}^{t} w_j^2 \\ &= \sum_{j=1}^{T}\left[\left(\sum_{i\in I_j} g_i\right)w_j + \frac{1}{2}\left(\sum_{i\in I_j} h_i + \lambda\right)w_j^2\right] + \gamma T \end{aligned} \tag{21}$$

which sums the prediction by leaves. The leaf score is then defined as

$$w_j = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda} \tag{22}$$

where g and h are the first and second order of the loss function, respectively. λ is a the regularisation parameter.

Each split is found by determining the best splitting point for optimising the objective. The *exact greedy algorithm* is described in Algorithm 3. Every time a split is done, the leaf is changed into a internal node. The best value of the objective on the j^{th} leaf is defined as

$$Obj^{(t)} = -\frac{1}{2}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} g_i h_i + \lambda} + \gamma \tag{23}$$

Trees are constructed by finding the best splitting points recursively until the maximum depth is reached. The pruning is used in a bottom-up order to remove nodes with a negative gain. The gain of the split is defined as

$$gain = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} g_i h_i + \lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} g_i h_i + \lambda} - \frac{\left(\sum_{i\in I_I} g_i\right)^2}{\sum_{i\in I_I} g_i h_i + \lambda}\right] - \gamma \tag{24}$$

Algorithm 3: Exact greedy algorithm for split finding [28].

Result: Split with max score
Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i\in I} g_i, H \leftarrow \sum_{i\in I} h_i$
for $k = 1, 2, \ldots, m$ **do**
  $G_L \leftarrow 0, H_L \leftarrow 0$
  for j *in sorted*(I, *by* x_j **do**
    $G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$
    $score \leftarrow max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}$
  end
end

4.0.7. Neural networks

An Artificial Neural Network (ANN) is a system of many connected nodes called neurons [42, 43]. ANN are vaguely inspired by biological neural networks that we find in brains of animals, and can be trained to perform specific tasks without being adjusted to any task-specific rules. There are two types of ANNs; single layered ANN, which is the simplest and oldest model; multilayered ANN, which is more sophisticated than a single-layered ANN, and is capable of solving more complex classification and regression tasks. The architecture of an ANN can be divided into three types of layers: input, output and hidden. The input layer receive the information to be processed, which then goes through a network of weighted neurons called the hidden layer until it reaches the output layer. Fig. 1 shows a simple single layered neural network.

The training of a neural network consists of two parts: forward propagation and backward propagation. For notations, we use w_{jk}^l for the weight w for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. We use a similar notation for biases and activations: b_j^l for the bias of the j^{th} neuron in the l^{th}, and x_j^l for the activation of the j^{th} neuron in the l^{th}.

The activation values by each layer is calculated The activation x_{lj} of the j^{th} neuron in the l^{th} layer is related to the activations in the $(l-1)^{th}$ layer by the equation

$$x_j^l = \sigma\left(\sum_k w_{jk}^l x_k^{l-1} + b_j^l\right) \tag{25}$$

where the sum is of every neuron k in the $(l-1)^{th}$ layer. For simplicity, we rewrite this expression in a matrix form by defining a weight matrix W^l for each l^{th} layer, whose components are all weights connecting to the l^{th} layer of neurons. For instance, weight w_{jk}^l is the weight component at the j^{th} row and k^{th} column in matrix W^l. Similarly, biases are redefined in matrix form as B^l for each layer l. The components of the bias matrix are the values b_j^l, for the j^{th} neuron in the l^{th} layer. At last, we define an activation matrix X^l whose components are the activations x_j^l. The final matrix version of the formula is defined as

$$Z^l = W^l X^{l-1} + B^l, \tag{26}$$

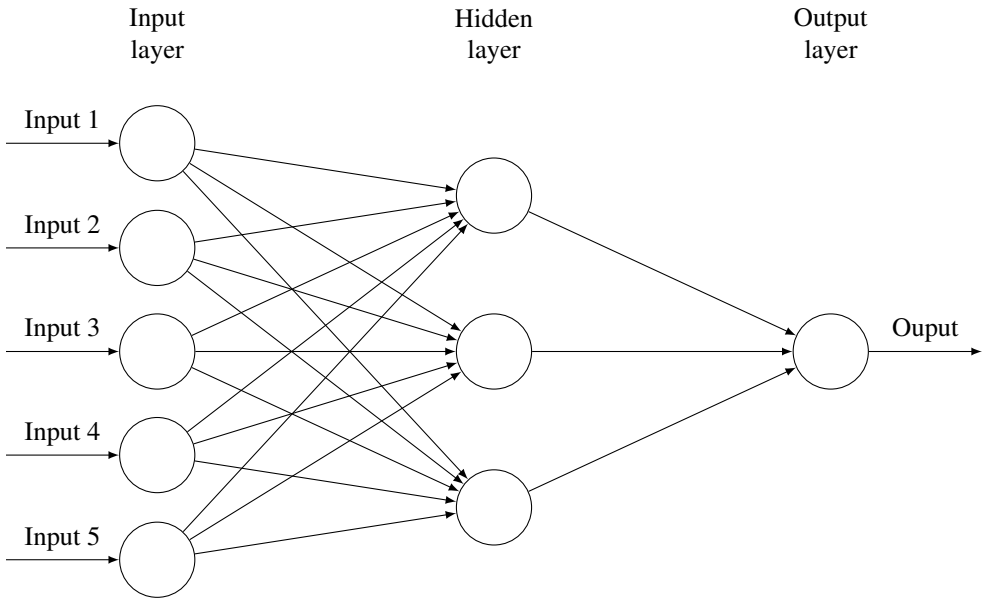

Figure 1: A simple feed forward neural network with 5 input neurons, one hidden layer with 3 neurons and a output layer with one output neuron.

$$X^l = \sigma\left(Z^l\right), \tag{27}$$

where the weight matrix W^l is applied to the input matrix X^{l-1}. The bias matrix is added to the dot product of $W^l X^(l-1)$. Finally, the activation function σ is applied element-wise to the components in the matrix.

With respect to any weight w or bias b in the network, the goal of backpropagation is to calculate the partial derivatives $\delta C/\delta W$ and $\delta C/\delta B$ of the cost function C. The cost function C as CCE (Equation 2) can be defined for weights and biases for single neurons as

$$\frac{\delta C}{\delta w_j} = \frac{1}{n}\sum_x x_j\left(\sigma\left(z\right) - y\right) \tag{28}$$

$$\frac{\delta C}{\delta b} = \frac{1}{n}\sum_x \left(\sigma\left(z\right) - y\right) \tag{29}$$

The C for all neurons in multiple layers can be generalised as

$$C = -\frac{1}{n}\sum_x\sum_j\left[y_j \ln\left(x_j^L\right) + \left(1 - y_j\right)\ln\left(1 - x_j^L\right)\right] \tag{30}$$

The equation for the error in the output layer, δ^L, is given by

$$\delta^L = \nabla_X C \odot \sigma^{'}\left(Z^L\right) \tag{31}$$

Here, $\nabla_X C$ is a vector that contains the partial derivatives $\delta C/\delta x_j^L$. Another way to describe $\nabla_X C$ is the rate of change of C with respect to the output activations.

An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}, is given by

$$\delta^l = \left(\left(W^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(Z^l\right) \tag{32}$$

where $(W^{l+1})^T$ is the transpose of the weight matrix W^{l+1} for the $(l+1)^{th}$ layer. We then take the Hadamard product $\odot\sigma'(Z^l)$. The transpose of the matrix W^{l+1} in order to move backwards through the network and obtain the error at the l^{th} layer.

The equation for the rate of change of the cost with respect to any bias in the network is defined as

$$\frac{\delta C}{\delta b_j^l} = \delta_j^l \tag{33}$$

which states that the error δ_j^l is exactly equal to the rate of change $\delta C/\delta b_j^l$. The equation for the rate of change of the cost with respect to any weight in the network is defined as

$$\frac{\delta C}{\delta w_{jk}^l} = x_k^{l-1}\delta_j^l \tag{34}$$

which shows how compute the partial derivatives $\delta C/\delta w_{jk}^l$ in terms of the quantities δ^l and x^{l-1}.

These formulas are needed in order to calculate the gradient of the cost function that is used to adjust the weights and biases in the neural network. Algorithm 4 summarises the mathematical steps in forward and backward propagation for a matrix-based neural network.

Algorithm 4: Forward and backward propagation in a neural network.

Result: The gradient of the cost function

Set the corresponding activation a^1 for the input layer;

forall $l = 2, 3, 4, \ldots, L$ **do**

$Z^l = W^l X^{l-1} + B^l$

$X^l = \sigma\left(Z^l\right)$

end

$\delta^L = \nabla_X C \odot \sigma'\left(Z^L\right)$

forall $l = L-1, L-2, L-3, \ldots, 1$ **do**

$\delta^l = \left(\left(W^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(Z^l\right)$

end

Output: $\frac{\delta C}{\delta w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\delta C}{\delta b^l_j} = \delta^l_j$

Table 1
Poker hand dataset information and statistics.

Class	Poker hands	Frequency	Probability
9	480	649740	0.0000015390771693
8	4,320	72193	0.0000138516945240
7	74,880	4165	0.0002400960384154
6	449,280	694	0.0014405762304922
5	612,960	509	0.0019654015452335
4	1,224,000	255	0.0039246467817896
3	6,589,440	47	0.0211284513805522
2	14,826,240	21	0.0475390156062425
1	131,788,800	2	0.4225690276110440
0	156,304,800	2	0.5011773940345370
Total	311,875,200		1.0

5. Methodology

Like any machine learning task, some essential steps are needed in order to train a classifier to predict classes from imbalanced data. First of, we collect poker hand data and create a dataset that represents the population, which in this case is every single combination of poker hands (unordered). Secondly, we select samples for training, validation and testing that still accurately depicts the population. Thirdly, different methods of data augmentation are considered. In the fourth step, the selected classifiers are prepared for multi-class classification. Lastly, the selected classifiers are subjected to different methods for solving the imbalance problem as well as being tuned with hyperparameters.

5.1. Creating the dataset

Initially, we started working with a publicly available poker hand dataset from UCI Machine Learning Repository by Cattral and Oppacher [44]. While this dataset contains over one million poker hand samples, the data distribution by class was not reflecting the full population of poker hands. The least represented class was still over-represented by almost two times. This was not too surprising; The dataset would have needed to be bigger for the distribution of samples per class to be more accurate.

This inspired us to create a new dataset, but with the same structure proposed by Cattral and Oppacher [44]. By keeping the same structure, we make sure that all methods created for the original dataset will also work with this new dataset. Because poker is a game with a set of rules and finite values, we were able to generate every poker hand combination that exist with object-oriented C# programming in the .Net developer platform. This was made possible by defining every distinct card as class objects and put them in a list. The program used this list to create every possible combination of cards and save them to a separate list as a hand class object. A hand assessing algorithm would then go over each hand in the list of hands and assign a rank to each. At last, the list with hand objects was randomly shuffled and saved to a CSV file that ended up being 7.5 GB large. This file contains every possible poker hand combination.

In the dataset, there are 311,875,200 unique poker hand samples. Table 1 shows dataset information and statistics by each class. For data structure, each card in a given poker hand is denoted by a suit-value that can be between 1 and 4, and a rank-value that can be between 1 and 13. Each poker hand is assigned a CLASS (rank) label between 0 and 9 which indicates how good the hand is and what kind of hand it is. Table 2 lists each predictor and a few examples. Table 3 lists each class and their description.

This dataset is quite large, so loading all of it into computer memory was challenging on our available hardware. With every sample in this dataset, we used a stratified random sampling technique to create a final sample that we could use for training, validation and testing. We did this for three reasons: Firstly, to lower the computational requirement to process all of this data in order to lower the training time needed. Secondly, to find out which methods, techniques and parameters are needed with a limited amount of imbalanced data to create a generalised model. By generalised, we mean a model that is not over-fitted on limited amount of data. Thirdly, to more realistically depict a real scenario where data is obtained, not generated. Training with almost every combination of a population should not be needed in order to create a predictive model for imbalanced data.

5.2. Creating sets for training, validation and testing

2% of the original dataset were used to create the stratified sample, and it ended up containing 6,237,504 unique samples. These were used to create three different sample sets: one training set, one validation set and one testing set. A training set contains the samples that the classifiers use to learn from. It is important that the training set accurately depicts the population in order for the classifier to not be biased as well as being able to predict poker classes in a generalised fashion. The validation set is a set of samples used for providing a unbiased evaluation of how well a model fit on the training set. It is used to see which methods and model hyperparamters that improves the predictive performance and to which degree these need adjustment. The testing set cannot be used for validation as this would give inaccurate, biased results. Instead, this portion is used to measure

Table 2
Dataset predictors.

Predictor	Description	Values
S1	Suit of card #1	Ordinal {1, 2, 3, 4} (Hearts, Spades, Diamonds, Clubs)
C1	Rank of card #1	Numerical {1, 2, ..., 13} (Ace, 2, 3, ... , Queen, King)
S2	Suit of card #2	Ordinal {1, 2, 3, 4} (Hearts, Spades, Diamonds, Clubs)
C2	Rank of card #2	Numerical {1, 2, ..., 13} (Ace, 2, 3, ... , Queen, King)
S3	Suit of card #3	Ordinal {1, 2, 3, 4} (Hearts, Spades, Diamonds, Clubs)
C3	Rank of card #3	Numerical {1, 2, ..., 13} (Ace, 2, 3, ... , Queen, King)
S4	Suit of card #4	Ordinal {1, 2, 3, 4} (Hearts, Spades, Diamonds, Clubs)
C4	Rank of card #4	Numerical {1, 2, ..., 13} (Ace, 2, 3, ... , Queen, King)
S5	Suit of card #5	Ordinal {1, 2, 3, 4} (Hearts, Spades, Diamonds, Clubs)
C5	Rank of card #5	Numerical {1, 2, ..., 13} (Ace, 2, 3, ... , Queen, King)
CLASS	Poker Hand	Ordinal {0, 1, ..., 9}

Table 3
Dataset classes.

Class	Description
0	Nothing in hand; not a recognised poker hand
1	One pair; one pair of equal ranks within five cards
2	Two pairs; two pairs of equal ranks within five cards
3	Three of a kind; three equal ranks within five cards
4	Straight; five cards, sequentially ranked with no gaps
5	Flush; five cards with the same suit
6	Full house; pair + different rank three of a kind
7	Four of a kind; four equal ranks within five cards
8	Straight flush; straight + flush
9	Royal flush; Ace, King, Queen, Jack, Ten + flush

Table 4
Stratified sample information Training, validation and testing set sizes.

Class	Sample size	Training set size	Validation set size	Testing set size
0	3,126,096	625,219	312,610	2,188,267
1	2,635,776	527,155	263,578	1,845,043
2	296,525	59,305	29,652	207,567
3	131,789	26,358	13,179	92,252
4	24,480	4,896	2,448	17,136
5	12,259	2,452	1,226	8,581
6	8,986	1,797	899	6,290
7	1,498	300	150	1,048
8	86	17	9	60
9	10	2	1	7
Total	6,237,504	1,247,501	623,750	4,366,253

the predictive performance of a given classifier.

20% (1,247,501) samples were used for training, 10% (623,750) samples were used for validation and the remaining 70% (4,366,253) samples were used for testing. Table 4 shows the distribution of poker hands by class for all of these sets. Consider the size of the stratified sample, we used this advantage to pick a big testing set. This decreases the chance that we will do a type I error. In other words, this makes the results from the testing more accurate and we can draw conclusions from it with a higher level of confidence.

5.3. Data augmentation

Empirically, a training set consisting of different numbers of representatives from the different classes could lead to a majority-biased classifier. With a similarly imbalanced test set, the classifier may show an optimistically high predictive performance. The validation and testing sets should be representative of the population, which is why we chose stratified sampling to begin with. Applying changes to these (primarily the testing set) will only increase the risk that we will draw the wrong conclusions about the results.

On the other hand, training set augmentations will not affect the final conclusions drawn from the testing set. One of the augmentations that we will test is SMOTE [6]. which will be applied to all minority classes (i.e. all classes except the majority class) and generate enough synthetic samples so every class has the same amount of samples. In theory, this will increase the number of samples in the training set from 1,247,501 to 6,252,190 samples.

Because machine learning algorithms are iterative processes, we suspect that bigger training sets increase the total training time. For extreme imbalanced datasets, SMOTE generates a lot of synthetic samples in order to achieve balance. We think this could be unnecessary, so we set out to test if there is a need to generate synthetic samples for frequently represented minority classes. In order to test this issue, we defined a custom SMOTE approach called *Weighted SMOTE* or *WSMOTE*, were we use

Table 5
Number of samples when applying WSMOTE to each class in the training set.

Class	Training set	New samples	Augmented set
9	2	129,948	129,950
8	17	14,438	14,455
7	300	833	1,133
6	1,797	138	1,935
5	2,452	101	2,553
4	4,896	50	4,946
3	26,358	9	26,367
2	59,305	4	59,309
1	527,155	0	527,155
0	625,219	0	625,219
Total	1,247,501	145,521	1,393,022

the class weights (i.e. the class frequency, see Table 3) to decide the amount of synthetic samples is to be generated. The formula is defined as

$$Samples(n_c) = n_c + \left\lfloor \frac{N}{n_c} a \right\rfloor \tag{35}$$

where n_c is the number of samples of class c from a given dataset. $\lfloor \cdots \rfloor$ indicates the floor value. N is the total number of samples in the dataset. a is an adjustable parameter that is recommended to be between $[0, 1]$. The function output is the new number of total samples for the specified class. We decided to test with $a = 0.2$ for each class, which added up to 145,521 new samples. Table 5 shows the number of new samples by class, and the new total number of samples in the augmented training set.

5.4. Model implementation

5.4.1. K-Nearest Neighbour

When writing the KNN algorithm, We choose to use the Euclidean distance for determining neighbours. Which is the length of the line segment connecting two points. K-nearest neighbour is a lazy supervised learning algorithm that is simple to use, and much more optimal on smaller datasets. We had no expectations for how good or bad results we would get from the algorithm, since we could not find any related work articles that has tried running a KNN algorithm on the poker hand dataset.

5.4.2. Naive Bayes

In this project we will look at two Naive Bayes algorithms provided by the sci-kit learning tools: Gaussian Naive Bayes and Complement Naive Bayes. Even though these methods are quite known for their text classification, we weren't able to find a lot of related work regarding using Naive Bayes for this Poker hands dataset. Therefore, before running the dataset with these methods we didnt quite know what to expect the results would look like.

Gaussian Naive Bayes is according to Schultebraucks, the simplest and most popular one out of the available methods. With this method, the likelihood of features is assumed to be gaussian (normal distribution). Gaussian Naive Bayes is especially recommended when the features are continuous.

As mentioned earlier, the problem with our dataset is the occurrences of the different classes being heavily imbalanced. A Naive Bayes algorithm which seems to be in favour of this is the Complement Naive Bayes (CNB) algorithm [45]. According to the sci-kit learning documentation, CNB is an adaption of the standard multinomial naive bayes algorithm, which is well known for text classification, and is particularly suited for imbalanced data sets. In a written article by Rennie M D, Shih, Teevan and Karger R, CNB was mentioned as a modified version Naive Bayes. Their goal was to investigate Naive Bayes supposedly poor performance in text classification. They mention that one systematic problem with Naive Bayes is when one class has more training examples than another, it will select poor weights for the decision boundary. This is due to a bias effect which shrinks weights for classes with few training examples. They attempted to balance the amount of training examples used per estimate by introducing a "complement class" formulation of Naive Bayes. CNB calculates the odds of a certain feature appearing in other classes by using statistics from the complement of each class to compute the model's weights

In our case, assuming we got the logic correctly, some examples for this model would be: After having drawn an Ace of hearts, what are the chances that we will end up with Four Of a Kind? Or, after having drawn an Ace of hearts and Ace of Diamonds, what are the chances we will end up with Two pair? Technically we can just tell the machine the answer to each probability since the amount of different hands are limited. But as mentioned before, simply telling the program how the rules works isn't the goal of this project.

So what exactly makes ComplementNB different from Normal Naive Bayes? When we drawn a card, using normal Naive Bayes, we would ask ourselves, what is the likelihood of this card appearing within a given class? Using ComplementNB we would rather ask ourselves, what is the likelihood of this card appearing in any class other than the class that we're "looking" at.

5.4.3. SVM

In order to implement SVM, we chose the Support Vector Classifier (SVC) package from scikit-learn [35] (version 0.20.3). According to the documentation made available, SVC supports multi-class classification, but it is handled according to a one-vs-one scheme, meaning the algorithm trains one classifier for every pair of classes. In our case, this results to training $2^{10} = 1024$ individual classifiers. Early on, we had a hard time getting any results out of SVC, as it would not complete in a reasonable time frame. We tried to set the kernel to *linear*, which should be faster, but this did not have a great enough effect on the training time. We tried a different SVM method called Lightning [46], as well as a GPU-optimised method called ThunderSVM [47], but they were still too slow for our dataset. In the end, we ended up

Table 6
SVM model parameters.

Parameter	Value
random_state	42
penalty	12
loss	squared_hinge
C	1.0
dual	TRUE
fit_intercept	FALSE
intercept_scaling	1
max_iter	10000
multi_class	ovo
tol	0.00001

replacing these methods with LinearSVC from scikit-learn [35], as it has a linear kernel optimised for Liblinear [48], instead of Libsvm [49] that SVC is based on.

LinearSVC should be faster for larger datasets, but it could also create a model with poor predictive performance if samples are not linearly separable. Early tests seemed to confirm that the poker hand samples were not that easy to separate, and we think that a non-linear SVM approach could have been better i terms of predictive performance. Table 6 lists the parameters used for LinearSVC.

5.4.4. AdaBoost

Due to easy implementation, our AdaBoost [26] model was created and trained with scikit-learn [35] (version 0.20.3). Table 7 shows the parameter values chosen for our final model, obtained with a simple grid search. We used a decision tree classifier as base estimator. We found that the SAMME.R algorithm with a learning rate of 0.5 yielded the best results.

criterion, was set the default value "gini", and is the function to measure the quality of a split (see Equation 12). The split strategy was set to "best" for the *splitter* parameter, which means that the algorithm should use the best split instead of a random best split. The *max depth* was set to 10 and is the maximum tree depth (i.e. the maximum number of nodes along a path). The default value is 3, but we noticed that deeper trees yielded better scores, but this also increased the training time drastically. *Minimum sample split* was set to 2, which is the minimum number of samples required to split an internal node. *minimum samples leaf* was set to 1, and is the minimum number of samples required to be at a leaf node. 0 was used for *minimum weight fraction leaf*, which is the minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. We set the number of *max features* to None, so that there are no limit to the number of predictors to consider when looking for the best split. Similariy, *max leaf nodes* was also set to None, so that the tree grow with unlimited number of leaf nodes by best impurity.

Table 7
AdaBoost model parameters.

Parameter	Value
n_estimators	500
learning_rate	0.5
algorithm	SAMME.R
class_weight	None
criterion	gini
max_depth	10
max_features	None
max_leaf_nodes	None
min_impurity_decrease	0.0
min_impurity_split	None
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0.0
presort	FALSE
random_state	42
splitter	best

5.4.5. Random Forest

Similarly to AdaBoost [26], our random forest [27] model was made with scikit-learn [35] (version 0.20.3). Table 8 shows the parameter values used for model initialisation. Some of the parameters in random forest share the same meaning as the ones in AdaBoost. See Section 5.4.4 for the meaning of these.

The parameter *number of jobs* is equal to the amount of CPU threads to use, which dictates the number of jobs to run in parallel. We set it to (-1) so it uses all available threads (in our case, 8). Max features was set to "auto", which means the max number of features equals to $\sqrt{n_{\text{predictors}}} = \sqrt{10} \approx 3$. We ended up not using *Bootstrap* samples and *out-of-bag samples* as this yielded worse results.

5.4.6. XGBoost

Table 9 lists the various parameters used for our final XGBoost [28] model. Some of these parameters share the same meaning with the parameters available in AdaBoost and RandomForest. Please review Section 5.4.5 and 5.4.4 for the meaning of these.

For *booster*, we used the tree based model, "gbtree" (i.e. gradient boosting tree), which is the default option. The step size shrinkage, i.e. *learning rate*, was set to 0.5. We also used 0.5 as the *gamma* value, which is the minimum loss reduction required to continue to split a leaf nodes in the tree. The *minimum child weight* was set to 1.0, which is the minimum sum of hessian needed in a child node. Bigger child weights gives a more conservative algorithm. The *maximum delta step* was set to 0, which means there are no constraint to the maximum delta step allowed for each leaf output. We set *alpha regulation* (i.e. *L1*) and *lambda regulation* (i.e. *L2*) to 0 to make the algorithm less conservative. The *objective* (i.e. loss function) used is "softmax" which is highly related to CCE (Equation 2). We used the multi logloss (also highly related to CCE) and the geometric mean as

Table 8
Random forest model parameters. (∗) The number of trees is set to an arbitrary high number as we are using early stopping.

Parameter	Value
n_jobs	-1
random_state	42
n_estimators	100000∗
max_depth	None
min_samples_leaf	3
min_samples_split	2
min_weight_fraction_leaf	0
max_features	auto
max_leaf_nodes	None
min_impurity_decrease	0
bootstrap	FALSE
oob_score	FALSE

evaluation metrics.

5.4.7. LightGBM

The parameters made available for LightGBM [29] share close similarities with AdaBoost, RandomForest and XGBoost. Please refer to Section 5.4.4, 5.4.5 and 5.4.6 for the meaning of these. Refer to Table 10 for the parameters used in creating the LightGBM model.

Like XGBoost, the boosting type used was the gradient boosting decision tree. The maximum number of leaves in base learners was increased to 60. The *subsample ratio* and *subsample frequency* was set to the standard 1 and 0, respectively. A frequency of 0 means it is not enabled. The *column subsample ratio* was set to 1, which is the ratio of columns when constructing each tree.

5.4.8. CatBoost

The last decision tree implemented is CatBoost [30]. The parameters used for initialising the CatBoost model share similarities with LighGBM and XGBoost. These are listed in Table 11. Because CatBoost does not support geometric mean natively, we created a custom geometric mean metric for evaluation, but that meant we had to use the CPU-optimised version of CatBoost instead of the much faster GPU-optimised version. After some testing, we decided to switch to GPU optimisation because of two reasons. Firstly, because of the amount of time saved in training the model and do predictions on the evaluation set. This benefited the tuning process as we could get more done within a smaller time-frame. Secondly, being able to do more in less time meant we could find the close-to-optimal parameters faster, which directly benefited the predictive performance. We used the multiclass (closely related to CCE in Equation 2) metric for loss as well as evaluation metric. We also used F1 and accuracy for evaluation as these had a small effect the training time.

5.4.9. TensorFlow

One of the tools used to model and train a neural network is TensorFlow [32] developed by Google AI. Keras is a high-

Table 9
XGBoost model parameters. (∗) The number of trees is set to an arbitrary high number as we are using early stopping. (∗∗) We used a custom geometric mean function as a secondary evaluation metric.

Parameter	Value
n_jobs	-1
random_state	42
num_class	10
booster	gbtree
n_estimators	100000∗
learning_rate	0.5
gamma	0.5
max_depth	10
min_child_weight	1
max_delta_step	0
reg_aplha	0
reg_lambda	0
objective	multi:softmax
eval_metric	mlogloss
eval_metric	gmean∗∗

Table 10
LightGBM model parameters. (∗) The number of trees is set to an arbitrary high number as we are using early stopping. (∗∗) We used a custom geometric mean function as a secondary evaluation metric.

Parameter	Value
boosting_type	gbdt
n_jobs	-1
num_class	10
n_estimators	100000∗
num_leaves	60
max_depth	10
learning_rate	0.1
min_split_gain	0
min_child_samples	20
subsample	1
subsample_freq	0
colsample_bytree	1
reg_alpha	0
reg_lambda	0
objective	multiclass
metric	multi_logloss
metric	gmean∗∗

Table 11
CatBoost model parameters. (*) The number of trees is set to an arbitrary high number as we are using early stopping.

Parameter	Value
task_type	GPU
thread_count	8
classes_count	10
num_trees	100000*
learning_rate	0.1
max_depth	10
l2_leaf_reg	1
objective	MultiClass
eval_metric	TotalF1
custom_metric	Accuracy

Table 12
Layers of the neural network model created with TensorFlow.

Layer (type)	Output Shape	Parameters
input (Dense)	(None, 1024)	11264
hidden1 (Dense)	(None, 1024)	1049600
hidden2 (Dense)	(None, 1024)	1049600
hidden3 (Dense)	(None, 1024)	1049600
hidden4 (Dense)	(None, 1024)	1049600
output (Dense)	(None, 10)	10250
Total params		4,219,914

level API provided with TensorFlow to build, train and test deep learning models. It is used for rapid prototyping, advanced research and production. Keras is a flexible and contains a lot of functionality which allows us to quickly build and test simple or very complex neural networks within minutes as it requires minimal lines of code.

The neural network model created for the poker imbalance problem, as shown in Table 12, contains four hidden layers. Each hidden layer consists of $2^{10} = 1024$ neurons. The input layer contains 10 neurons (one for each predictor). The output layer consists of 10 neurons (one for each poker class) with a Softmax activation function (for CCE). This resulted in a network that has 4,219,914 trainable parameters. We also implemented some classifier-specific parameters.

First of is the activation function used for the four hidden layers. Neural network models without activation functions are essentially just linear regression models. The activation function does the non-linear transformation over the input making it able to learn and perform more complex tasks. There exists many different activation functions, but we ended up using the Rectified Linear Unit (ReLU) [50]. Glorot et al. [51] showed in 2011 that rectifying neurons are a better model of biological neurons and yielded equal or better performance compared to traditional activation functions like sigmoid [52] and hyperbolic tangent [53]. More recently proposed activation functions like Leaky Rectified Linear Unit (LReLU) [54] and Parametric Rectified Linear Unit (PReLU) [55] also show good results in testing [56], but these were not explored in our model. ReLU is defined as

$$f(x) = x^{+} = max(0, x) \tag{36}$$

where x is the input to a neuron.

Second of is the loss function. In imbalanced multi-class classification, loss functions, also called objective functions or error functions, are used to measure the difference between predicted classes and ground truth classes. The output of the function, called the loss value, helps guide the model optimisation. The loss function we ended up using was Sparse Categorical Cross Entropy (Equation 2).

Next is the optimisation algorithm, which helps to minimise (or maximise) the loss value calculated by a given loss function. After some initial testing, we found the optimisation algorithm, Adam [57], to yield good results. The amount of change to be applied to a given network model during each step, or the step size, is called the *learning rate*. This parameter is an essential part of tuning the network training in order to obtain better performance. The higher the learning rate, the faster the model will converge towards the local or global minima, but too high learning rates will make the learning unstable. We ended up using Adam with a learning rate of $10^{-5} = 0.00001$.

Lastly is the batch size, which is the number of training samples in one forward and backward pass. Higher batch size requires more computer memory, but if the computer supports it, it may speed up the training time. Keskar et al. [58] showed that higher batch sizes used with Stochastic Gradient Descent (SGD) can degrade the model, we found that a batch size of 4096 worked well with our problem. As our dataset is extremely imbalanced, higher batch sizes may be better suited compared to lower batch sizes where there is a high chance the minority classes won't be present in every batch.

5.4.10. FeedForwardNet

The second model is created and trained with FeedForwardNet in MATLAB [31]. The best evaluation score was achieved using the following parameters:

- 1 input layer.
- 1 hidden layer with 130 nodes.
- 1 output layer.
- 20% train/10% validation/70% Test split ratio.
- Epochs: 1000
- Network training function: trainscg

5.5. Shared training method across models

There are several general steps to training machine learning classifiers for multi-class classification. While these models learn poker hand classes differently, there still are some shared parameters and techniques implemented that we will go over.

Table 13
Loss functions by classifier.

Classifier	Parameter value	Loss type
LinearSVM	*squared_hinge*	Squared Hinge
Naive Bayes		
K-Nearest Neighbours		Euclidean distance
RandomForest	-	Gini Impurity
AdaBoost	*gini*	Gini Impurity
XGBoost	*softmax*	CCE (log loss)
LightGBM	*multiclass*	CCE (log loss)
CatBoost	*MultiClass*	CCE (log loss)
TensorFlow	*softmax*	CCE (log loss)

5.5.1. Loss functions and evaluation metrics

Each classifier has it's own set of defined loss functions and evaluation metrics made available through parameters. Table 13 lists the different loss functions used for each classifier. We tried to limit us to CCE loss (e.g. multiclass, mlogloss, softmax) for most classifiers, but there where some that did not support it.

There is also room for creating custom metrics that can be used during training. We decided to add a geometric mean metric to each classifier as a way to monitor the change in predictive performance that is not influenced by class imbalance. After some testing we discovered that using a custom metric sometimes resulted in longer training times. Especially for GPU-optimised algorithms. This made it difficult to test the some of the models on the large poker hand dataset in a appropriate time frame. While we believe that evaluating classifiers on metrics that are highly relevant to the problem at hand is appropriate, we decided to use a different approach for the models that converges slow when using a custom metric or simply do not support it.

5.5.2. Iterations

Another part of training is finding out how many iterations to train for. Iterations is a general term for how many times the learning algorithm repeats. For decision trees and boosted trees, the word "iterations" is an alias for the number of trees, usually shortened to *n_trees* or *n_estimators*. For neural networks, the number of *epochs* is used instead, stating how many times the weights in the networks should be adjusted by the cost.

These are adjustable parameters and an essential part of the model tuning process. More iterations does not always equal to better predictive performance, but it will increase the training time. We decided that the training time should not be influencing the training process and the final scores, so we implemented *Early Stopping* for the models that support it.

5.5.3. Early stopping

Early stopping is a approach for ending the training when a certain evaluation metric criterion is met (i.e. not by the number of iterations). A common way to set this criterion is to define a level of *patience*, a score *tolerance* and to specify which evaluation metric to monitor. The patience equals to the number of iterations the model will train for while not registering any improvement made to the evaluation metric. If the model does not improve more than the set tolerance value, the training stops. The idea is that the model will train for as long as there are no more improvements to be gained. We found that using a patience level between 25 and 100 (depends on the learning rate) iterations worked well for all of the models.

While ending the training at the right time may help in finding the optimal model, the training still ends too late for when the optimal evaluation metric was registered. The higher the patience level, the further away the model is from the optimal state registered. In order to solve this issue, we implemented model snapshots; Every time a new best evaluation metric score is registered, the model is saved to either a file or to a variable in computer memory.

For random forest, AdaBoost and TensorFlow, we created a custom early stopping class for calculating the evaluation metrics, saving the best model and for stopping the training when no improvements were made. For XGBoost, CatBoost and LightGBM, we did not need to create an early stopping class as these were already natively implemented, with an option for saving the best model.

Table 14
Test scores from training and testing with the old, UCI [44] poker hand dataset. **Best score** is in bold and underlined, while next best is just underlined.

Classifier	*Accuracy*	*Gmean*
K-Nearest Neighbours (Euclidian)	**0.614000**	0.325000
FeedForwardNet ()	0.845000	

6. Results

Table 15 shows all the results. The results are presented in confusion matrices, and we also have included graphs for the algorithms that supported evaluation metric history. For comparison, we will look at some different outcomes for some techniques to see how well it performed. Most of the results will be presented with: First a simple run-through with the needed parameters to run the selected algorithm. Second, A rerun of the selected algorithm using cross-validation or alternatively a validation set with the exception of neural networks where a validation set is required to create the system. Third, a rerun of the selected algorithm by examining differing costs or weighting the classes in case of imbalance. Lastly, a rerun of the selected algorithm with changes in the hyper-parameters or adding other techniques to improve accuracy such as SMOTE to improve a unbalanced dataset. This is not the case for every method that we have tried but we will cover everything we have done regardless.

6.1. C5.0 decision trees

The dataset used for creating decision trees using C5.0 is the poker hands data set which can be found at UCI. The reason for this was that the tests which we will cover in the next few chapter

Table 15
Test scores from training and testing with the new, proposed poker hand dataset. **Best score** is in bold and underlined, while next best is just underlined.

Classifier	Iterations	Train duration (sec)	Test duration (sec)	*Accuracy*	*Gmean*
Naive Bayes (CNB)				0.389370	0.311200
Naive Bayes (CNB) + SMOTE				0.151650	0.370580
Naive Bayes (GNB)				0.501640	0.300000
Naive Bayes (GNB) + SMOTE				0.198770	**0.425510**
Scikit-learn K-Nearest Neighbours (Euclidian)	-	540	529.3153	0.525747	0.448091
Scikit-learn K-Nearest Neighbours (Manhatten)	-	539	853.7490	0.539228	0.443559
LinearSVM	1000	2229	0.6193	0.501570	0.300000
LinearSVM + weights	1000	373	0.5116	0.472716	0.301166
AdaBoost	500	6560	1038.5779	0.660446	0.493086
AdaBoost + weights	500	6667	1034.2841	0.646699	0.489990
AdaBoost + WSMOTE	500	7900	1021.0474	0.648325	0.490894
AdaBoost + SMOTE	500	56498	1020.1530	0.622537	0.486451
RandomForest	12	560	55.4995	0.777490	0.427241
RandomForest + weights	262	8283	266.7266	0.803760	0.581121
RandomForest + WSMOTE	59	611	63.4938	0.767144	0.463455
RandomForest + SMOTE	54	1508	69.3454	0.752334	0.504722
XGBoost	176	1674	136.7631	0.998604	0.804550
XGBoost + weights	190	1855	151.7264	0.998789	0.826327
XGBoost + WSMOTE	191	1977	159.1160	0.998580	0.806843
XGBoost + SMOTE	75	5721	74.4658	0.947933	0.746881
LightGBM	28	76	13.9719	0.655304	0.436780
LightGBM + weights	1622	1352	2137.4680	0.974637	0.824555
LightGBM + WSMOTE	120	156	61.4912	0.714826	0.503972
LightGBM + SMOTE	140	514	75.3114	0.789806	0.594836
CatBoost	2110	170	41.9965	0.999836	0.877041
CatBoost + weights	1344	118	34.2797	**0.999944**	0.888083
CatBoost + WSMOTE	2773	253	59.3957	0.999785	0.872277
CatBoost + SMOTE	2647	713	55.6256	0.993443	0.743192
TensorFlow	47	1086	89.3652	0.993267	0.659498
TensorFlow + weights	290	4832	90.7784	0.999281	**0.912772**
TensorFlow + WSMOTE	59	1389	94.7261	0.994595	0.783682
TensorFlow + SMOTE	305	22394	89.0478	0.988981	0.815630

	0	1	2	3	4	5	6	7	8	9
0	111841	651								
1	3714	6845	12							
2	335	700	171							
3	122	342	9	40						
4	62	16			15					
5	40	6				8				
6	5	30		1						
7		6								
8	3				1				1	
9	2	2								1

Figure 2: Confusion matrix results using Decision trees. Errors: 6032(24.3%)

Table 16
Poker hand dataset scores using the C.5 tool with the dataset from UCI

Type	Error rate Train	Error rate Test
Decision Tree	6032(24.3%)	419166(42.0%)
Rules	4919(20.5%)	259913(26.8%)
DT with Costs	419166(27.4%)	419166(40.6%)

where we use the C5.0 tools, was done before we decided to create our own dataset. We decided to not do the tests again with the new dataset that we generated.

First we downloaded the dataset and added it to the school directory at Østfold University College which allows for Linux commands at school or through VPN from home. The first Decision Tree was created using the C.5 tools which is accessible through Linux. The files were altered following the C.5 guidelines and produced Decision Tree with an error rate of 24.3% or more specifically, 6032 errors. By looking at figure 2 we can see that the result shows a clear sign to imbalance in the dataset. We can see that the result is heavily carried by classifying the lowest and second lowest hand ranks and didn't find any of the rank 6 and 7 hand classes. However, a rare occurrence which we rarely saw while going through all the other different algorithms, is the chosen classifier being able to find any of the rank 8 and 9 classes. The decision tree managed to find one rank 8 class and one rank 9 class and classify them correctly.

6.1.1. Decision Trees vs Rules

By running the dataset with Rules in C5.0, we can see a slight improvement in the error rate as it has decreased down to 20.5% or rather, 4919 total errors. However, much like the decision trees, the result here is also heavily carried by the lower ranked classes. As we can see in figure 3, the confusion matrix shows that Rules didn't find any of the rank 3, 5, 6, 7, 8, 9 classes at all, yet we managed to get a lower error rate. For getting a better overview of why the results look the way they are, we tried looking at some of the rules generated by C5.0. However, as there are a total of 803 rules (840 with the costs file) generated through c5.0 and a good chunk of these describes a single class, it is hard to find out if one single rule is sensible or not as it only specifies either the rank, suit or both for 1 to 3 cards. But if we look at the last 5 rules described below we can see that many of these rules doesn't make much sense as one of this rules, at best, was 50% correct:

- Rule 1: If Rank1 = 3 (card 1 is an Ace), and Rank2 = 2 (card 2 is a 2), and Suit3 = 3 (card 3 is Diamond), and Rank4 = 4 (card 4 is a 4), then those cards belong in class 4. If the order matters then there is some errors in here. Class 4 doesn't have any Aces, card 2 is not a 2 of any kind, card 3 is diamond which is correct and card 4 is not a 4. All in all this rule is 25% correct.
- Rule 2: If Rank2 = 6 (card 2 is a 6), and Rank3 = 7 (card 3 is a 7), and Suit4 = 4 (card 4 is a Club), and Rank4 = 4 (card 4 is a 4), then those cards belong in class 4. Card 2 is a 6 of Club which is correct, card 3 is a 7 which is correct, card 4 is a spade so the rule is incorrect here, card 4 is a 8 of a club so the rule is wrong here too. All in all this rule is 50% correct.
- Rule 3: If Rank3 = 9 (card 3 is a 9), and Rank4 = 7 (card 4 is a 7), and Suit5 = 3 (card 5 is a diamond), and Rank5 = 8 (card 5 is a 8), then those cards belong in class 4. The third card of class 4 is a 7 of diamonds so the rule is incorrect here, the 4th card is a 8 of spades so the rule is incorrect here too, the fifth card is 9 of hearts so the rule is incorrect here too. All in all this rule is 0% correct.
- Rule 4: If Rank2 = 4 (card 2 is a 4), and Suit4 = 4 (card 4 is a Club), and Rank4 = 9 (card 4 is a 9), and Rank5 = 12 (card 5 is a queen), then those cards belong in class 5. For class 5 card is 2 of hearts so the rule is correct on this one, card 4 is a heart of 8 so the rule is wrong here too, card 5 is king of hearts so the rule is wrong here too. All in all this rule is 0% correct..
- Rule 5: If Rank2 = 11 (card 2 is a Jack), and Rank4 = 6 (card4 is a 6), and Suit5 = 4 (card 5 is a Club), and Rank5 = 1 (card 5 is a 1), then those cards belong in class 5. In class 5 the 2nd card is 2 of hearts so the rule is wrong here, the 4th card is a 8 of hearts so the rule is wrong here too, the 5th card is king of hearts so the rule is wrong here too. All in all this rule is 0% correct.

6.1.2. Generalising ability in C5.0 using cross-validation or a testing set.

For this specific task we chose to go for adding an additional test set which can be downloaded together with the training set on the UCI website. A new decision tree was created and we could see an almost double increase in error rate as it increased

	0	1	2	3	4	5	6	7	8	9
0	11287	1123	83							
1	2021	8574	4							
2	83	1119	4							
3	2	511								
4	79	6	1		7					
5	41	7	3			3				
6		36								
7		6								
8	4	1								
9	4	1								

Figure 3: Confusion matrix results using Rules. Errors: 4919(20.5%)

to 42.0% from 24.3% which we saw earlier. However, since the test file is pretty big, the number of instances has increased from 25010 cases to 1000000 cases. The increase in amount of cases may be the reason to the nearly double increase in error rate and we can see from the confusion matrix in figure 4 that overall, the classification has performed worse. Even though it manages to recognise many of the different ranked classes, it fails to classify a lot of them correctly as it totally misses the rank 5, 8 and 9 classes and didn't manage to find any of the rank 6 and 7 classes at all.

6.1.3. Costs in C5.0

We can see in all the examples earlier that the algorithm have a really hard time at finding any combinations of the classes from 6 to 9, and if there are any then they are classified wrong in most cases. We want to try having the algorithm know classifying the classes between 6 and 9 is very bad while 1 to 5 is not really that bad even though a lot of the ranked classes between 1 and 3 were classified wrong. Figure 5 shows the confusion matrix after running the tool again with an attached cost file. In short, what we tell the program when using this cost file is that making an error in classifying for example any of the cards that belongs to class 6 wrong is really expensive. This is represented by a number where 0 is not expensive and 5 is really expensive and are defined like this in the cost file: 6, 0: 5.

We can see from the results that with the costs file we get slightly better error rate comparing to previous examples. Though other values defining the costs may improve it even further, more experimenting with costs values should be done if the goal is to decrease the error rate even further.

6.1.4. Winnowing

We ran the winnowing option in the C5.0 tool which was used for creating the decision trees earlier and ended up with an error rate of 27.1% using the training data (25010 cases) and

	0	1	2	3	4	5	6	7	8	9
0	399386	99743	1049	206	439	340			20	26
1	237321	179838	4178	747	227	165			12	10
2	16131	30340	1027	117	4	3				
3	5586	15092	302	137	4					
4	3013	793			69	2			4	4
5	1592	399	4	1						
6	100	1279	42	3						
7	11	216	1	2						
8	8	4								
9	3									

Figure 4: Confusion matrix results using Decision trees with test data. Errors: 419166(42.0%)

	0	1	2	3	4	5	6	7	8	9
0	455491	43185	968	250	896	373			20	26
1	275307	137218	2999	6273	506	173			12	10
2	19343	26114	825	1314	23	3				
3	6837	13387	178	709	10					
4	3379	399			97	2			4	4
5	1827	161	4	1	3					
6	151	1187	34	52						
7	18	203	1	8						
8	8	4								
9	3									

Figure 5: Confusion matrix results using Decision trees with test data and attached cost file. Errors: 419166(40.6%)

	0	1	2	3	4	5	6	7	8	9
0	11755	738								
1	4293	6299	6		1					
2	356	739	111							
3	133	338	4	38						
4	64	14			15					
5	45	3				6				
6	5	30		1						
7		6								
8	3				1				1	
9	3	1								1

Figure 6: Confusion matrix results using Winnowing with Training data in C.5. Errors: 419166(27.1%)

	0	1	2	3	4	5	6	7	8	9
0	416420	83861	143	58	440	241			20	26
1	247618	171605	2405	518	204	126			12	10
2	17064	29789	679	85	5					
3	5936	14902	138	140	5					
4	3118	684			73	2			4	4
5	1658	337		1						
6	130	1262	29	3						
7	14	212	2	2						
8	8	4								
9	3									

Figure 7: Confusion matrix results using Winnowing with Test data in C.5. Errors: 419166(41.1%)

41.1% error rate using the test data (1000000 cases). Comparing with the decision trees that we made earlier, we can see that error rate has only slightly improved and as we can see from the confusion matrices in figure 6 and figure 7 it still makes the same mistakes at the same places from earlier.

6.2. K-nearest neighbour

We first tried to run the 2% sample dataset but it was too big to run a KNN algorithm. When we tried to run the 2% sample we got memory error, so we had to use the poker hand dataset from UCI database.

K-nearest neighbour algorithm took around 15 hours to run on the UCI dataset, that is because the KNN algorithm is not optimal to run on bigger datasets. We made a KNN algorithm in plain python and also tried to use sci-kit's implementation of the KNN algorithm. the KNN algorithm from the sci-kit were a lot faster but we got lower accuracy, and better geometric mean. Also on the scikit-learn algorithm we tried calculating neighbours with using manhatten in addition to euclidean distance, but it yielded worse results.

We ran five tests with the KNN algorithm in python to find the best value of k that gave the best accuracy and geometric mean. We got the best result when we ran the algorithm with k = 3, where we got an accuracy of 0.614 and geometric mean of 0.325 in plain python and accuracy of 0.525 and geometric mean of 0.448 in scikit-learn's implementation of KNN.

6.3. Naive Bayes

To see how well Naive Bayes performed with our dataset, a script were created in python which ran both Gaussian and Complement Naive Bayes. The dataset used for the script were initially the 2% sampled dataset which we created ourselves as mentioned in chapter 5.1. Other different sample percentage datasets where tested as well but all in all, the 2% dataset proved to yield the best results and is what we will cover here.

6.3.1. Gaussian Naive Bayes

With the first run-through with the Gaussian Naive Bayes, we managed to score an accuracy of 50% with a 70/30 split. We tried to alter this accuracy by changing around the parameters but it seemed to not improve beyond the 50% mark. Then we decided to look at the confusion matrix and as we can see by the results at figure 8, the classification was a bit out of the ordinary compared to what we had seen using other algorithms. 50% of what the classifier found was the class 0, 42% class 1, 5% class 2, 2% class 3 and was only able to classify them as class 0, hence the 50% accuracy. This seemed extremely over-fitted and we decided to try out applying SMOTE to the program to see if that would help. Figure 9 shows the confusion matrix after applying SMOTE and as we can see, now it at least managed find something of every class. However, the accuracy was reduced to over half of what it was before as the new accuracy was 19.8%, but as suggested in chapter 3.0.5, Geometric mean would be a better indicator for what we are doing. By calculating the Geometric mean in both cases (with and without SMOTE) we can see that the performance has actually improved. Geometric mean without SMOTE: 30%, Geometric mean with SMOTE: 42%. We also tried doing a cross-validation and managed to land the best scores using 5-folds. See Table 17 for a full overview of scores using Naive Bayes.

6.3.2. Complement Naive Bayes

We ended up seeing similar results the first time we ran Complement Naive Bayes, after first run-trough we ended up with an accuracy of 34% and after playing around with the parameters a bit we managed to increase it to 38.9% with a 70/30 split. By looking at the confusion matrix (figure 10) we came to the same conclusion that this looked extremely over-fitted too, which seemed to be a theme with Naive Bayes and the poker

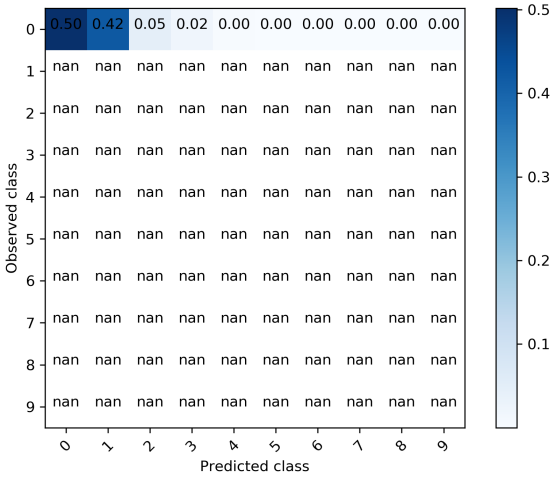

Figure 8: Confusion matrix results using Gaussian Naive Bayes.

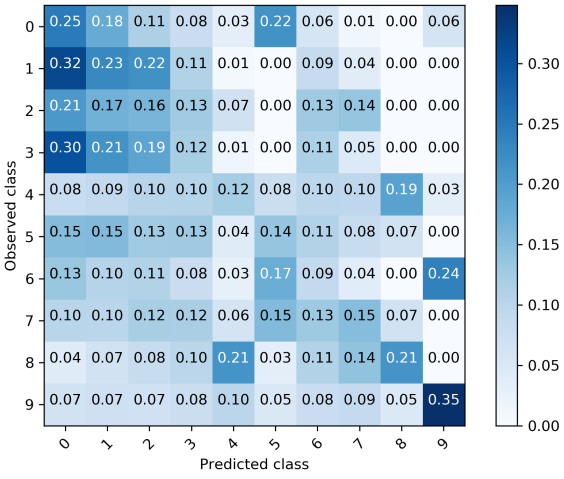

Figure 9: Confusion matrix results using Gaussian Naive Bayes with SMOTE.

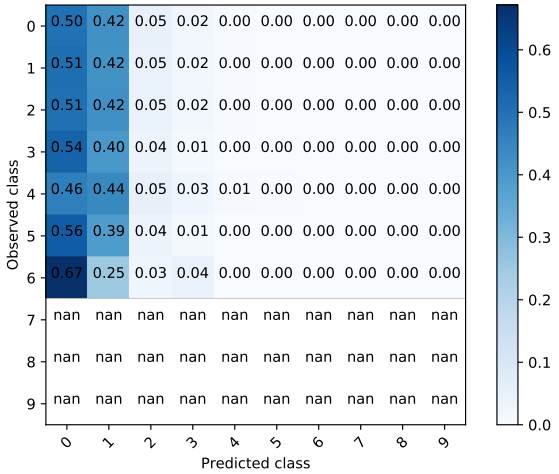

Figure 10: Confusion matrix results using Complement Naive Bayes.

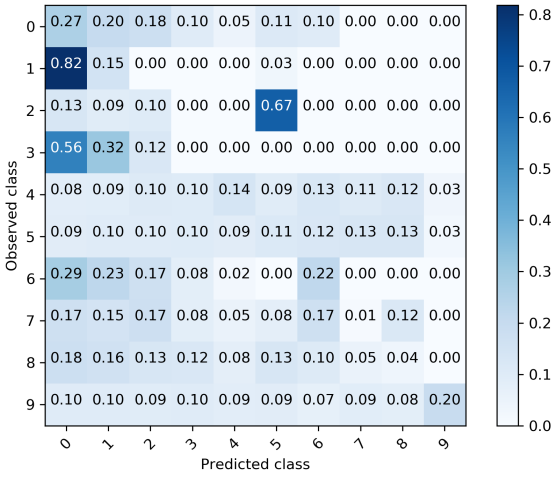

Figure 11: Confusion matrix results using Complement Naive Bayes with SMOTE.

hands dataset. It was for the most part only able to find the first two classes and much of it was classified wrong as well. As we can see by the confusion matrix at figure 11, by applying SMOTE here as well we could see similar results to Gaussian Naive Bayes with SMOTE. The accuracy has decreased drastically (new accuracy: 15.1%) but at least now it was able to find something of each class. Geometric mean was calculated here too and by comparing it to the geometric mean calculated without applying SMOTE we could see that it actually improved. By using Geometric mean as an indicator to performance here as well, we were able to improve it using SMOTE. Geometric mean without SMOTE: 31.1%, Geometric mean with SMOTE: 37%. We tried doing a cross-validation here as well and managed to land the best scores using 5-folds here too. See Table 17 for a full overview of scores using Naive Bayes.

Table 17
Poker hand dataset scores using Naive Bayes.

NB	Accuracy	GM	CV
CNB	0.38937	0.3112	0.35(+-0.04)
GNB	0.50164	0.30000	0.50(+-0.00)
CNB SMOTE	0.15165	0.37058	0.15(+-0.00)
GNB SMOTE	0.19877	0.42551	0.20(+-0.00)

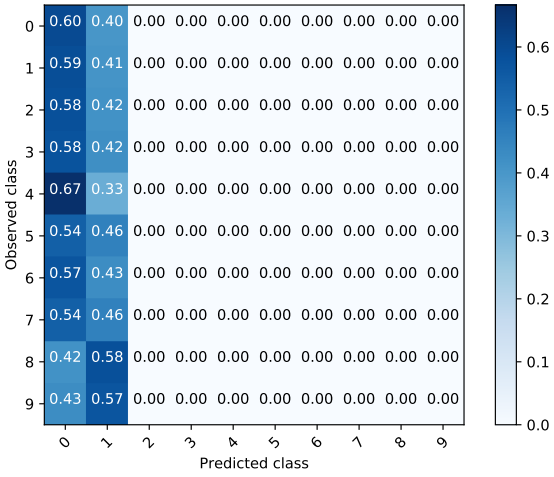

Figure 12: Confusion matrix results using SVM

6.4. Support Vector Machines

Fig. 12 shows the results from the best model in a confusion matrix. The highest score we could achieve with linear SVMs was by using weights, which yielded a geometric mean of 0.301166 geometric mean and 0.472716 accuracy. The geometric mean increased +0.001166 when using weights, but the accuracy decreased by -0.028854. The highest accuracy achieved was without weights at 0.501570 accuracy. This score could be improved with radial basis function kernel, as it seems that the poker data is not that easily separable with a linear kernel, but the time needed to train the model would increase drastically.

6.5. AdaBoost

AdaBoost is one of the oldest machine learning algorithms tested. It managed to score better than SVMs in our testing, but it still failed to tackle the class imbalance as seen in Fig. 13. The best result was achieved without any data augmentation or weights, yielding a geometric mean of 0.493086 and accuracy of 0.660446 from 500 iterations of training. Weights or different levels of augmentation did not affect the results that much as seen in Table 15.

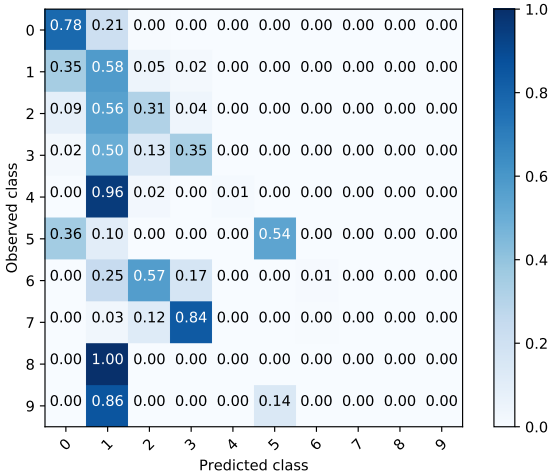

Figure 13: Confusion matrix results using AdaBoost boosted trees.

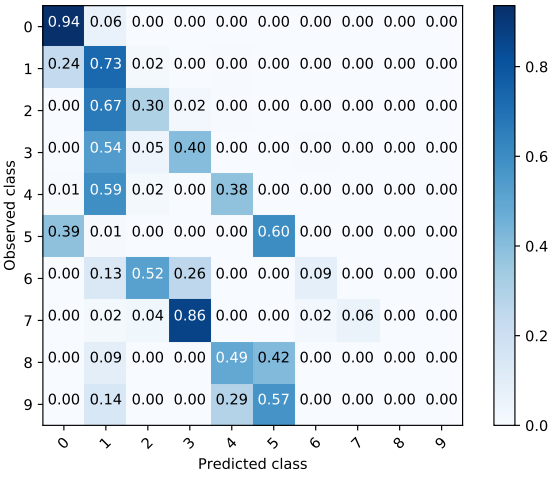

Figure 14: Confusion matrix results using Random Forest bagged trees.

6.6. Random forest

As seen in Fig. 14 and Fig. 15, random forest struggled to classify the minority classes in the dataset (class 6 to 9). The best model trained used weights and managed to achieve an geometric mean of 0.581121 and an accuracy of 0.803760. Compared to AdaBoost, this score was an increase of +0.088034 and +0.143313 in geometric mean and accuracy, respectively. Weights also had a big impact on Random Forest compared to the other methods as seen in Table 15. It is also clear that WSMOTE trained almost 3 times as fast compared to SMOTE, while still achieving better results. The results from using WSMOTE and SMOTE could be affected by the selected parameters, so a more conclusive test must be conducted in order to draw accurate conclusions.

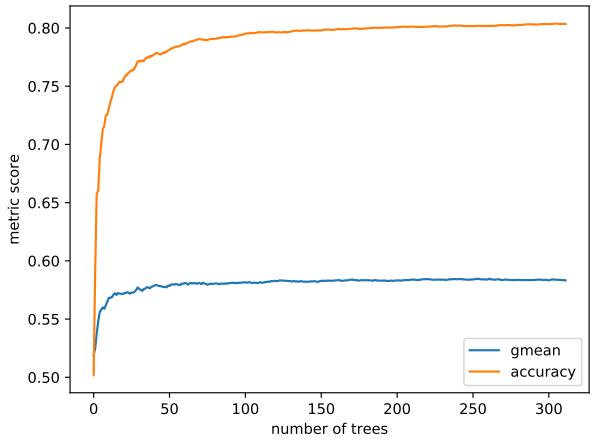

Figure 15: Plot results using Random Forest bagged trees.

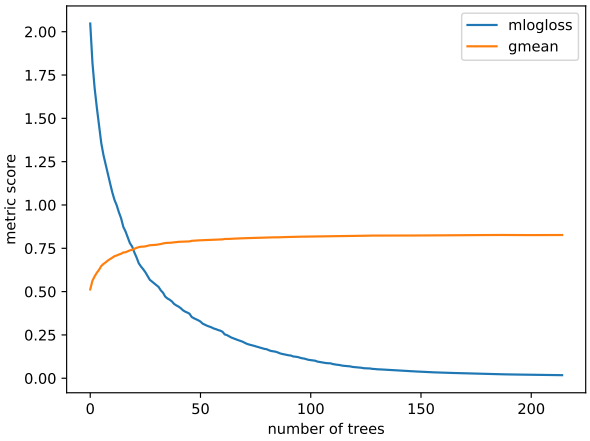

Figure 17: Plot results using XGBoost boosted trees.

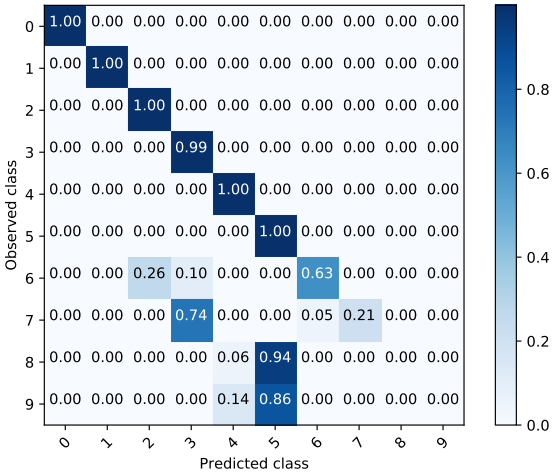

Figure 16: Confusion matrix results using XGBoost boosted trees.

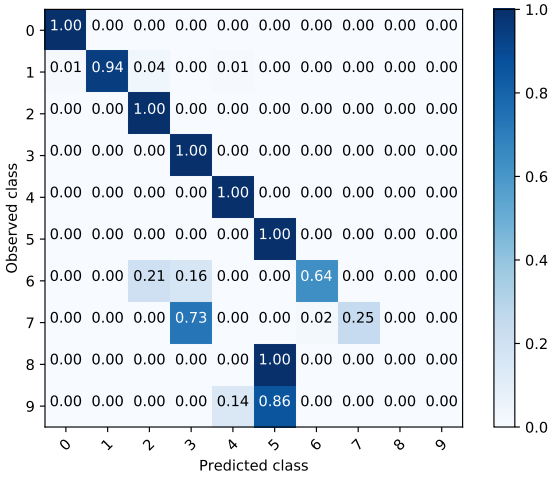

Figure 18: Confusion matrix results using LightGBM boosted trees and x training samples.

6.7. XGBoost

XGBoost [28] is a relatively recent proposed model for classification, compared to AdaBoost and Random Forest. With weights, it managed to score higher than the older algorithms in both geometric mean and accuracy, yielding 0.826327 and 0.998789, respectively. This is an +0.245206 and +0.195030 increase in geometric mean and accuracy, compared to Random Forest, and Fig. 16 clearly shows that it is successful in classifying the first 6 classes. It managed to classify 63% and 21% of the samples belonging to class six and seven correctly, but failed in classify minority class eight and nine. Fig. 17 shows that the model converged towards optimal geometric mean for the model around 100 iterations, but did not gain any large improvements from there, despite the improvement made to the multi logloss.

6.8. LightGBM

LightGBM [29] managed to score marginally better than XGBoost, with an geometric mean of 0.824555 and a accuracy of 0.974637. While the accuracy was slightly worse compared to XGBoost (-0.023967) it increased the geometric mean by 0.020005. Like XGBoost and Random Forest, using weights yielded the best result as seen in Table 15. Fig. 18 shows that it classified the different classes similairy like XGBoost did in as seen in Fig. 16. Fig. 19 shows that LightGBM used quite more iterations and to achieve the results, and it is also clear that the multi logloss has potential improvement. Despite the number of iterations and slow convergence, LightGBM used less time per iteration.

6.9. CatBoost

Like XGBoost, CatBoost [30] is one of the most recently proposed decision tree algorithms used in our testing. It managed to achieve the best result among the different decision trees (AdaBoost, Random Forest, XGBoost and LightGBM). Using weights, it scored a geometric mean of 0.888083, which is the second best across all models. It also managed an accuracy

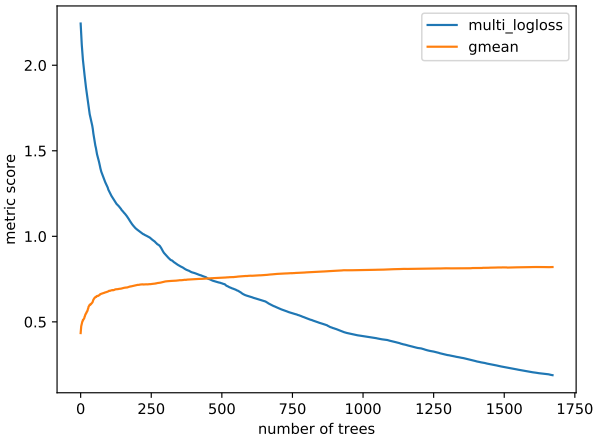

Figure 19: Plot results using LightGBM boosted trees.

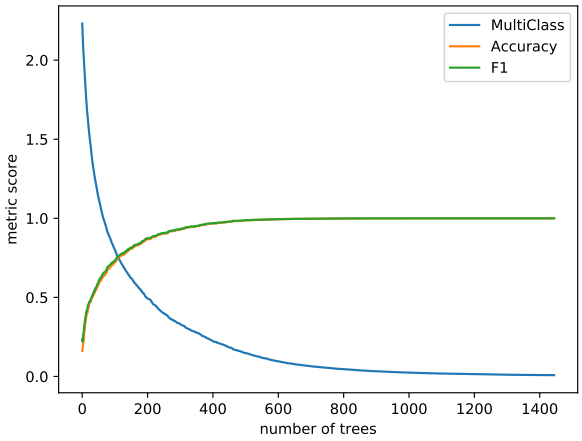

Figure 21: Plot results using CatBoost boosted trees.

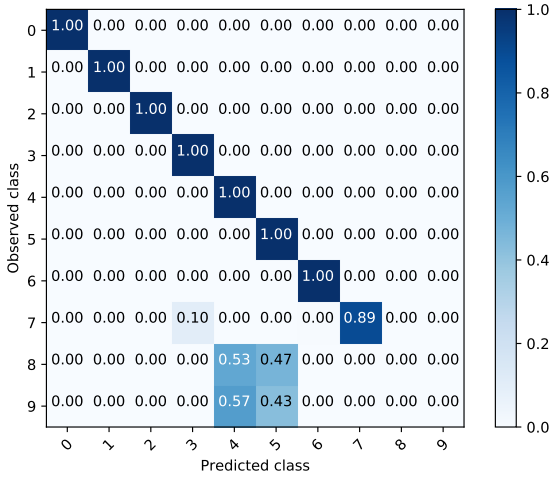

Figure 20: Confusion matrix results using CatBoost boosted trees.

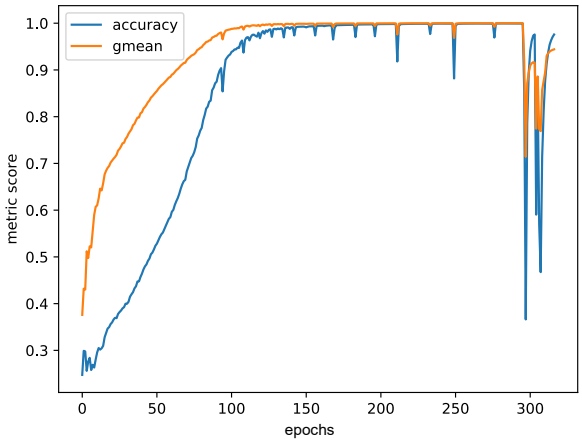

Figure 22: Training progression for the best results achieved with a five layer neural network created with TensorFlow.

of 0.999944, which is the highest accuracy achieved across all models.

Fig. 21 shows the training progress of our CatBoost model. It stopped training after 1344 iterations, which took only 118 seconds because it uses GPU instead of the CPU. Predictions was also fast, completing in 34.28 seconds, i.e. 18196 poker hand samples each second. As seen in Table 15, the next best result achieved with our CatBoost model was with WSMOTE (without weights), beating plain SMOTE both by score and execution time. Fig. 20 show that CatBoost successfully classified the first 7 classes and got 89% of the samples from class 7 correct. Despite its high accuracy and geometric mean, it completely missed the two last minority classes.

6.10. TensorFlow

Our TensorFlow [30] model ended up achieving the best score across all models tested for imbalanced multi-class classification of poker hand data. The best neural network model was created with weights, scoring a geometric mean of 0.912772 and an accuracy of 0.999281. The geometric mean improved by +0.024688 compared with the best model trained with decision trees (CatBoost). Fig. 22 shows the predictive performance per epoch measured in geometric mean and accuracy. The training ended after 340 epochs as the progression degraded quite heavily and unsteadily after the 290 mark. While the model needed 80.53 minutes to finish training, the testing time was still kept short to only 90.78 seconds. This is just the advantage of neural networks compared to decision trees, as the size of the network model is left unchanged no matter how many epochs the network needs.

As seen in Fig. 23, our model near perfectly classified every class except the two last minority classes. While the best decision tree model made with CatBoost failed to classify any of the last two classes, the TensorFlow model succeeded in classifying 34% of the samples from class 8. This is how it scored the best geometric mean, as these samples heavily

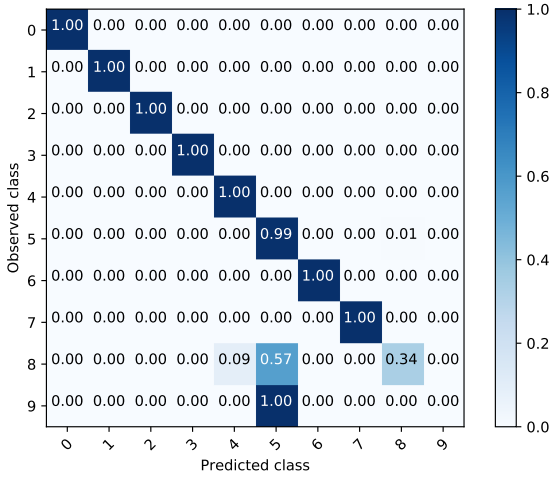

Figure 23: Confusion matrix of the best results achieved with a five layer neural network created with TensorFlow.

affected the final geometric mean.

Please refer to Table 15 for a full comparison across all models.

6.11. FeedForwardNets

We ran the MATLAB code without any restrictions in amount of epochs and ended up achieving the ”best” result landing at 1000 epochs, see figure 25. As we can see from the results (see confusion matrix at figure 24), the outcome of the feedforward neural network is very similar to the results achieved by other methods like the decision tree and early testing of methods such as boosting. The performance is heavily carried by the lower classes, which, as mentioned earlier is more common to come across. 49.5% of the score comes from classifying class 1 (the lowest hand, also known as ”no hand”) correctly, and 31.8% from classifying class 2 correctly. It managed to find and classify a bit of the other classes like class 3, class 4, class 6 and barely finding and classifying anything of class 7. Lastly, as we can see it didn't manage to find and/or classify correctly any of the classes: 5, 8, 9 and 10. Note that the classes here are labelled from 1 to 10 unlike how it is labelled in the dataset, which is 0 to 9.

7. Conclusion

In this project we have explored several methods for poker hand induction. We discovered early that the poker hands dataset in general is very imbalanced since the amount of combinations of the lowest most common class, otherwise known as ”no hand”, is 325,635 times bigger than the amount of combinations of the highest rare class, otherwise known as ”Royal Flush”. We initially attempted to handles this imbalance by performing a stratified sampling so we could get a dataset with equal amount of each class. However, we decided that it was not a good idea to perform this stratified sampling on the dataset provided by UCI. The reason for this was that the dataset from UCI only had

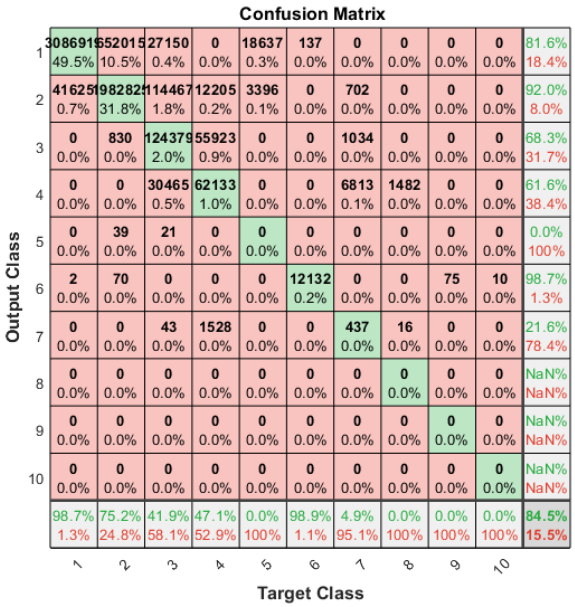

Figure 24: Confusion matrix results using Feedforward Neural Network.

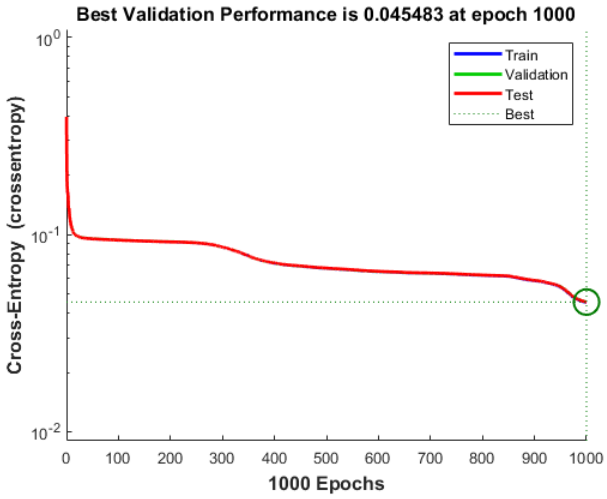

Figure 25: Neural Network performance using FeedForwardNet.

1000000 instances, thus it doesn't contain every combination of poker hands which is what we wanted to ensure a good sample. A new dataset were created using c programming in visual studio which covers every possible poker hand combination and we used this dataset to perform the sample. Several sample percentages were tried out such as 2%, 5%, 10% samples etc. and we seemed to land on a sweet spot with a 2% sample. Any higher or lower sample percentage proved to yield worse result on the methods we have tried in this project. Overall we scored the best result with Neural Networks, which achieved very high percentage in both accuracy and geometric mean. Some honorable mentions would be CatBoost, XGBoost and LightGBM which also achieved high percentages in accuracy and geometric mean with the right parameters. We eventually chose to use geometric mean as performance indicator and is recommended for imbalanced datasets which we were trying to handle in this project. The results were presented in confusion matrices and we are happy with the overall results, which provides a good overview over selected techniques to handle poker hand induction.

8. Future work

One thing we haven't mentioned in our project yet is our heavy bias for testing and "runtime". We performed significantly more tests on CatBoost than any other boosting method as it allows for the use of the GPU. Using the GPU, otherwise known as Graphic Processing Unit, makes the duration of training and testing significantly shorter than the standard of using a single core of the CPU. Performing more or equally as much tests on the other methods could possibly have given us an even better result than what we have got with CatBoost and Neural Networks. We also feel like we didn't perform as much tests as we should have with Neural Networks, especially with using feedforwardnet. By looking at a lot of previous work done arround neural networks and the poker hand data set, we dare say that almost everyone used different parameters when it comes to the amount of hidden layers and amount of neurons in each hidden layer. Simply running three to five tests with feedforward neural network trying different parameters, is in our opinion, not enough to land at a final conclusion that "these parameters" provides the "best" performance for poker hand induction using feedforward neural networks. We also agreed that we should have looked into other alternatives when choosing a Naive Bayes algorithm or maybe use an entirely different dataset for testing it. Even though Complement Naive Bayes is known for handling imbalanced datasets, it is generally built arround handling naive bayes poor performance as a text classifier. Further, Gaussian Naive Bayes supports features that are continous which are not the case for the poker hands dataset. If we wanted to continue testing the poker hands dataset with naive bayes we may have to look into other alternatives such as Hoeffding Naive Bayes Tree or Hoeffding Naive Bayes Perceptron Tree [59], which have performed well with the poker hands dataset before, are more advanced methods and more difficult to implement compared to the classifiers which we have used in this project.

References

[1] G. Haixiang, L. Yijing, J. Shang, G. Mingyun, H. Yuanyue, G. Bing, Learning from class-imbalanced data: Review of methods and applications, Expert Systems with Applications 73 (2017) 220–239.

[2] S. Shilaskar, A. Ghatol, P. Chatur, Medical decision support system for extremely imbalanced datasets, Information Sciences 384 (2017) 205–219.

[3] K. Oh, J.-Y. Jung, B. Kim, Imbalanced classification of manufacturing quality conditions using cost-sensitive decision tree ensembles AU - Kim, Aekyung, International Journal of Computer Integrated Manufacturing 31 (2018) 701–717.

[4] S. Wang, X. Yao, Multiclass Imbalance Problems: Analysis and Potential Solutions, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 42 (2012) 1119–1130.

[5] A. Anand, G. Pugalenthi, G. B. Fogel, P. N. Suganthan, An approach for classification of highly imbalanced data using weighting and undersampling, Amino Acids 39 (2010) 1385–1391.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, SMOTE: Synthetic Minority Over-sampling Technique, Journal Of Artificial Intelligence Research (2011).

[7] Y. Li, X. Nie, R. Huang, Web spam classification method based on deep belief networks, Expert Systems with Applications 96 (2018) 261–270.

[8] J. Sun, J. Lang, H. Fujita, H. Li, Imbalanced enterprise credit evaluation with DTE-SBD: Decision tree ensemble based on SMOTE and bagging with differentiated sampling rates, Information Sciences 425 (2018) 76–91.

[9] C.-O. Truic, A. A. Leordeanu, Classification of an Imbalanced Data Set Using Decision Tree Algorithms, U.P.B. Sci. Bull., Series C 79 (2017).

[10] G. Solon, S. J. Haider, J. M. Wooldridge, What Are We Weighting For?, Journal of Human Resources 50 (2015) 301–316.

[11] Z. Zhang, M. R. Sabuncu, Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels (2018).

[12] F. Li, X. Zhang, X. Zhang, C. Du, Y. Xu, Y.-C. Tian, Cost-sensitive and hybrid-attribute measure multi-decision tree over imbalanced data sets, Information Sciences 422 (2018) 242–256.

[13] M. Kubat, S. Matwin, Addressing the Curse of Imbalanced Training Sets: One-Sided Selection, in: In Proceedings of the Fourteenth International Conference on Machine Learning, Morgan Kaufmann, 1997, pp. 179–186.

[14] S. Jabin, Poker hand classification, in: 2016 International Conference on Computing, Communication and Automation (ICCCA), IEEE, 2016, pp. 269–273.

[15] M. Riedmiller, H. Braun, A direct adaptive method for faster backpropagation learning: the RPROP algorithm, in: IEEE International Conference on Neural Networks, pp. 586–591.

[16] W. W. M. H. Gill, P. R.; Murray, The Levenberg-Marquardt Method, Practical Optimization (1981) 136–137.

[17] M. F. Møller, A scaled conjugate gradient algorithm for fast supervised learning, Neural Networks 6 (1993) 525–533.

[18] T. Kohonen, The self-organizing map, Proceedings of the IEEE 78 (1990) 1464–1480.

[19] J. S. Sihota, Poker Rule Induction, 2015.

[20] W. Xu, Genetic Algorithms and Poker Rule Induction, Technical Report, 2015.

[21] G. Hamel, Kaggle: Poker Rule Induction, 2014.

[22] G. Dişken, Predicting Poker Hands With Artificial Neural Networks, Technical Report, Çukurova Üniversitesi, Adana, 2010.

[23] A. Garg, Predicting Poker hand using neural networks, 2016.

[24] S. Noor Ali, Analysis and Prediction of Poker Hand Strength (2007).

[25] S. Weston, M. Culp, N. Coulter, R. Quinlan, M. Kuhn, Package 'C50' (version 0.1.2), 2018.

[26] Y. Freund, R. E. Schapire, A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting, Journal of Computer and System Sciences 55 (1997) 119–139.

[27] L. Breiman, Random Forests, Machine Learning 45 (2001) 5–32.

[28] T. Chen, C. Guestrin, XGBoost: A Scalable Tree Boosting System (2016).

[29] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, LightGBM: A Highly Efficient Gradient Boosting Decision Tree, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems 30, Curran Associates, Inc., 2017, pp. 3146–3154.

[30] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, A. Gulin, CatBoost: unbiased boosting with categorical features (2017).

[31] MATLAB (version R2019a), 2015.

[32] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: A system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283.

[33] C. Cortes, V. Vapnik, Support-Vector Networks, Machine Learning 20 (1995) 273–297.

[34] H. Zhang, The Optimality of Naive Bayes, in: FLAIRS Conference, American Association for Artificial Intelligence, 2004.

[35] 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, M. Perrot, E. Duchesnay, Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[36] T. S. U. Hastie, S. T. A. U. Rosset, J. U. o. M. Zhu, H. U. o. M. Zou, Multi-class AdaBoost, Statistics and Its Interface 2 (2009) 349–360.

[37] T. S. U. Hastie, S. T. A. U. Rosset, J. U. o. M. Zhu, H. U. o. M. Zou, Multi-class AdaBoost (2006) 0–20.

[38] T. K. Ho, Random decision forests, in: Proceedings of the third international conference, Montreal, Quebec, Canada, volume 1, New York City, pp. 278–282.

[39] T. K. Ho, The random subspace method for constructing decision forests, IEEE Transactions on Pattern Analysis and Machine Intelligence 20 (1998) 832–844.

[40] Y. Amit, D. Geman, Shape Quantization and Recognition with Randomized Trees, Neural Computation 9 (1997) 1545–1588.

[41] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification And Regression Trees, volume 7, Routledge, New York, 1th edition, 2017.

[42] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Networks 61 (2015) 85–117.

[43] M. Nielsen A., Neural Networks and Deep Learning, Determination Press, 2015.

[44] R. Cattral, F. Oppacher, Poker Hand Data Set, 2007.

[45] J. D. M. Rennie, L. Shih, J. Teevan, D. R. Karger, Tackling the Poor Assumptions of Naive Bayes Text Classifiers, in: Proceedings of the Twentieth International Conference on International Conference on Machine Learning, ICML'03, AAAI Press, 2003, pp. 616–623.

[46] M. Blondel, F. Pedregosa, Lightning: large-scale linear classification, regression and ranking in Python, 2016.

[47] Z. Wen, J. Shi, Q. Li, B. He, J. Chen, ThunderSVM: A Fast SVM Library on GPUs and CPUs, Journal of Machine Learning Research 19 (2018) 1–5.

[48] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, C.-J. Lin, LIBLINEAR: A Library for Large Linear Classification, J. Mach. Learn. Res. 9 (2008) 1871–1874.

[49] C.-C. Chang, C.-J. Lin, LIBSVM: A library for support vector machines, ACM Transactions on Intelligent Systems and Technology 2 (2011) 27:1–27:27.

[50] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, H. S. Seung, Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit., Nature 405 (2000) 947–951.

[51] X. Glorot, A. Bordes, , Y. B. B. T. P. o. t. F. I. C. o. A. I. Statistics, Deep Sparse Rectifier Neural Networks, 2011.

[52] J. Han, C. Moraga, The influence of the sigmoid function parameters on the speed of backpropagation learning BT, in: J. Mira, F. Sandoval (Eds.), International Workshop on Artificial Neural Networks, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 195–201.

[53] I. Abramowitz, Milton;Stegun, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, Dover Publications, New York, 9th edition, 1965.

[54] A. L. Maas, A. Y. Hannun, A. Y. Ng, Rectifier Nonlinearities Improve Neural Network Acoustic Models, in: Proceedings of the 30 th International Conference on Machine Learning, volume 28, p. 6.

[55] K. He, X. Zhang, S. Ren, J. Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (2015).

[56] B. Xu, N. Wang, T. Chen, M. Li, Empirical Evaluation of Rectified Activations in Convolutional Network (2015).

[57] D. P. Kingma, J. Ba, Adam: A Method for Stochastic Optimization (2014).

[58] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, P. T. P. Tang, On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima (2016).

[59] A. Bifet, G. Holmes, B. Pfahringer, E. Frank, Fast Perceptron Decision Tree Learning from Evolving Data Streams, in: Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part II, PAKDD'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 299–310.