# Classification with tree based methods

Written by R.S

# Abstact

This article presents a exploration of tree-based machine learning algorithms, specifically focusing on Boosting, Random Forests, and Classification Boosting. The discussion begins with a theoretical overview, offering a understanding of the fundamental principles behind these algorithms.The exploration involves practical application through simple simulation. This simulation serve as a means to bridge theory and application, enabling readers to observe how these algorithms perform for a classification task on "Chess (King-Rook vs. King)" dataset from the USI dataset repository [BH94].FFNN will be used as a benchmark to see how a none tree based method does compared tree methods.

# Contents

**Long Section Title** **i**

**Abstact** **ii**

**Contents** **iii**

**1 Introduction** **1**

**2 Theory** **2**
2.1 Bagging . . . . . 4
2.2 Random Forests . . . . . 5
2.3 Boosting . . . . . 6
2.4 Gradient Boosting . . . . . 8
2.5 Variable importance . . . . . 10
2.6 Tree ensemble algorithms . . . . . 11

**3 Simulation study** **13**
3.1 Accuracy . . . . . 14
3.2 Precision and Recall . . . . . 15
3.3 F1 macro . . . . . 15
3.4 F1 micro . . . . . 16
3.5 Result using scoring function accuracy in RandomizedSearchCV 17
3.6 Result using scoring function F1 macro in RandomizedSearchCV 21
3.7 Result using scoring function F1 micro in RandomizedSearchCV 25
3.8 Result for FFNN . . . . . 29

**4 Discussion** **31**

**5 Conclusion** **33**

**Bibliography** **34**

# CHAPTER 1

# Introduction

Tree-based methods are essential tools in machine learning, providing a practical way to understand relationships between different factors. These methods are known for being quick to learn, easy to interpret, scalable, and capable of handling various types of input. Due to these qualities, tree-based methods have become widely used in different fields.

In this article, we explore the theories behind tree-based methods, starting with decision trees and moving on to concepts like bagging, random forests, and AdaBoost. Our goal is to build a theoretical foundation before diving into a detailed simulation study.

Motivated by the strategic and logical nature of chess, we aim to evaluate how well tree-based methods perform using the "Chess (King-Rook vs. King)" dataset from the USI dataset repository. We believe that the complexities of chess make it an interesting test for tree-based methods.

As we examine tree-based methods, we also introduce a comparison by including a Feedforward Neural Network (FFNN) in our study. The FFNN acts as a benchmark, helping us assess how effective tree-based methods are compared to a more complex neural network approach.

Guided by theoretical insights and motivated by the chess dataset, our exploration goes beyond theory to practical validation. The following sections will unfold the results obtained from the simulation study, blending them with the foundational theories presented earlier. Our main reference will be [HTF01][The Elements of Statistical Learning] which provides additional context to the topic.

In the conclusion of this article, we present a straightforward summary that combines theoretical insights with real-world findings. This approach aims to shed light on the practical aspects of tree-based methods and contribute to a broader understanding of their usefulness in different fields.

The simulations used in this article can be found in the following link:
https://github.com/gituser1234566/some_coding/tree/main/project3fys-stk4155

# CHAPTER 2

# Theory

Tree-based methods partition the feature space into regions, with each region modeled by a simple model as part of the global tree model. The tree is constructed through a recursive binary partition of the feature space. The splitting procedure involves splitting a region based on a variable $X_j$ and a split condition $s$, resulting in regions $R_1(j,s) = \{X|X_j \leq s\}$ and $R_2(j,s) = \{X|X_j > s\}$. At each internal node, an existing region is split into two new regions. The process continues, subject to the criteria of minimizing the loss under some loss function. The leaf nodes represent the final regions $R_1, ..., R_M$. In the regression setting, each region is modeled with a constant $c_m$.

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m) \tag{2.1}$$

For a classification problem, this constant represents region-specific class probabilities $p_{mk} = p(G = k|X \in R_m)$,the probability for class $k$ in region $m$.For every region $R_m$ we have a vector $[p_{mk}]_{k=1}^{K}$.We estimate the class probabilities as:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(g_i = k) \tag{2.2}$$

$N_m$ being the total number of data-points in region $R_m$.

We can classifying the observations in region $m$ to a class by majority vote:

$$k(m) = \arg\max\{\hat{p}_{mk}\} \tag{2.3}$$

The splitting procedure could be done by making a split if the decrease in the cost function exceeds a threshold. The cost function value before a split is then compared to the cost function after the split. Each time a split happens, the cost function is divided into two. When the splitting procedure is done, the final expression becomes:

$$C(T) = \sum_{m=1}^{|T|} N_m Q_m(T) \tag{2.4}$$

where $|T|$ is the number of terminals.

For classification, the $Q_m$ used are:

Misclassification error:

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} I(g_i \neq k(m)) = 1 - \hat{p}_{mk}(m) \tag{2.5}$$

Gini index:

$$Q_m(T) = \sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{2.6}$$

Entropy:

$$Q_m(T) = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk} \tag{2.7}$$

It is recommended to use the Gini index and the entropy for growing the tree because the misclassification error does not distinguish between splits that leads to pure nodes i.e a split leading to regions only containing one class and classes who do not.

The process of building the tree often leads to overfitting. Therefore, what is often done is to grow the tree and then prune it by collapsing internal nodes dependent on a tuning parameter $\alpha \geq 0$.

The procedure of pruning the tree can be summed up as follows: define a subtree $T \subset T_0$ as any tree that can be obtained by collapsing some of its internal nodes. Also, define $m = 1, ..., |T|$ to index the leaf nodes. Define the cost-complexity criterion as:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T| \tag{2.8}$$

For each chosen tuning parameter $\alpha \geq 0$, the goal is to find the tree that minimizes:

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T| \tag{2.9}$$

$\alpha$ controls the tradeoff between tree size and fit to the training data.

One approach to find the optimal tree $T_\alpha$:

Once the tree is grown, we can collapse the internal node that gives the smallest increase in $C(T)$. This strategy is applied recursively until only the root node is left. It has been shown that among the sequence of trees, we can find $T_\alpha$.

For most classification problems, the evaluation metric used is the confusion matrix and the ROC curve.

## 2.1 Bagging

One method that extends the use of decision trees is bagging. Bagging is a technique that combines the common approach of bootstrap sampling with decision trees.

Bagging is a method for improving estimations by averaging over estimates from a collection of bootstrap samples:

The algorithm can be summarized as follows:

Let $Z = \{(x_1, y_1), ..., (x_N, y_N)\}$ be our training data, then:

for $b = 1, ..., B$:

Create bootstrap sample $Z^{*b}$.

Fit model $\hat{f^{*b}}(x)$, in our case, a classification tree.

Once all models are fitted, compute the bagging estimate as:

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f^{*b}}(x) \tag{2.10}$$

In the classification setting, assuming a classification rule on an underlying k-class response vector, such that for a specific $x$, $\hat{f}(x)$ is used to classify the response at $x$ as:

$$\hat{G}(x) = \arg \max_k \hat{f}(x) \tag{2.11}$$

The bagged estimate of the same response vector is given by the vector $\hat{f}_{\text{bag}}(x) = (q_1(x), ..., q_K(x))$, where $q_k(x)$ is the proportion of trees predicting class $k$ for the response vector for input observation equal to $x$. We can then either use consensus votes or select the most probable class. The consensus vote is given as:

$$\hat{G}(x) = \arg \max_k q_k(x) \tag{2.12}$$

And the most probable class sums up all probability vectors for the bag of trees for classes $1, ..., k$ at input $x$, normalizes the summation with the total number of trees before choosing the class with the highest probability among the resulting vector:

$$\hat{G}(x) = \arg \max_k \frac{1}{B} \sum_{b=1}^{B} \hat{p^{*b}}_k(x) \tag{2.13}$$

Intuitively, Bagging can reduce the variance of unstable prediction procedures since we are taking an average of multiple predictors based on datasets that are generated from the same distribution.

## 2.2 Random Forests

Random Forests build on the idea of bagging, offering a simple and effective enhancement to the method of creating a bag of predictors. The bagging procedure implies that each tree in the bag is identically distributed, and the expectation of the average $\hat{f^{*b}}(x)$ is the same as the expectation of each individual tree $\hat{f^{*b}}(x)$. Therefore, any improvement in the fitting procedure comes from variance reduction.

The variance of the average of B identically distributed (id) random variables, each with variance $\sigma^2$, with positive pairwise correlation $\eta$ is given by:

$$\eta\sigma^2 + \frac{1-\eta}{B}\sigma^2 \tag{2.14}$$

We can mitigate the second part of the equation by increasing the number of bootstrap samples. The first part is the main concern since we are using trees fitted on bootstrap samples, and these trees will be highly correlated.

Random Forests aim to address this problem. Instead of allowing each split in the trees to be chosen based on all possible variables, we restrict the options of variables to split on. The intuition is that if we have a strong predictor, and if we let the tree choose from the full list of features for each split, the strong predictor will be part of most trees, thereby increasing the correlation between trees. By restricting the space of choices, the predictor will choose different relevant features, reducing pairwise correlation between any two trees in the bag.

Random Forest is likely to perform poorly if the total number of features $p$ is large, but the fraction of relevant variables is small when we choose $m$ to be small. The reason is that the probability is high that no relevant variable will be selected.

In addition to controlling the pairwise correlation between any pair of trees in the Random Forest, we can also control the variance of the distribution of trees.

To illustrate the following, we write:

$$\sigma^2(x) = \mathrm{Var}_{\Theta,Z}[T(x;\Theta(Z))] \tag{2.15}$$

Where $T(x;\Theta(Z))$ is a specific tree composed of a set of nodes $Z$, and $\Theta()$ is a function describing the structural attributes of the tree, split variable at each internal node, split point at each internal node, and output value at each terminal node.

Using the law of total variance on this quantity, we get:

$$\mathrm{Var}_{\Theta,Z}[T(x;\Theta(Z))] = \mathrm{Var}_Z[E_{\Theta|Z}[T(x;\Theta(Z))]] + E_Z[\mathrm{Var}_{\Theta|Z}[T(x;\Theta(Z))]] \tag{2.16}$$

The first term represents the sampling variance of the Random Forest estimator, and it decreases as $m$ decreases. The second term represents the cumulative variance that comes from the variation of trees that can be constructed from $Z$, for each tree in the bag of trees.

For Random Forest, the bias is the same as the bias of the individually sampled trees $T(x;\Theta(Z))$.

## 2.3 Boosting

The core concept of boosting is to sequentially construct an additive model using weak learners. At each step, a weak learner is added based on observation weights, emphasizing the importance of observations that were misclassified by previously added weak learners. This principle is fundamental to the first boosting algorithms, such as AdaBoost and gradient boosting, which we will explore later.

Let's begin by examining AdaBoost, the first boosting algorithm introduced by Freund and Schapire (1997). AdaBoost can be formulated as forward stagewise additive modeling under the exponential function: $L(y, f(x)) = \exp(-yf(x))$.Forward Stagewise Additive Modeling is a step-by-step approach to building predictive models. It starts with a basic model and adds simple models one at a time, adjusting each addition to improve predictions without changing the previously added models.More general :

From a statistical point of view.Boosting can be viewed as a technique for fitting an additive basis expansion of the form:

$$f(x) = \sum_{m=1}^{M} \beta_m b(x;\gamma_m) \tag{2.17}$$

where,

$\beta_1, ..., \beta_M$ are the expansion coefficients.

$b(x; \gamma) \in R$ are simple basis functions of $x$ defined by $\gamma$.

These methods are fitted by minimizing some loss function using the training data.

$$argmin_{\{\beta_m, \gamma_m\}_{m=1}^{M}} \sum_{i=1}^{N} L(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m)) \tag{2.18}$$

The above optimization problem is computationally difficult for many loss functions and basis function,but can be approximated through heuristic optimization if one can decompose the minimization problem into smaller minimization problems that sequentially updates the global minimization problem and for each step solve:

$$min_{\{\beta, \gamma\}} \sum_{i=1}^{N} L(y_i, \beta b(x_i; \gamma) \tag{2.19}$$

For Adaboost we do the following:

For each step $m$, we need to solve:

$$argmin_{\{\beta_m, G_m\}} \sum_{i=1}^{N} \exp\left(-y_i \left(f_{m-1}(x_i) + \beta_m G_m(x_i)\right)\right) \tag{2.20}$$

In the case of binary classification, where $G_m(x) \in \{-1, 1\}$, and setting $w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$, we get:

$$argmin_{\{\beta_m, G_m\}} \sum_{i=1}^{N} w_i^{(m)} \exp\left(-\beta_m y_i G_m(x_i)\right) \tag{2.21}$$

The solution involves minimizing:

$$G_m = argmin_G \left\{ \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)) \right\} \tag{2.22}$$

By substituting $G_m$ back into the expression, we can derive the following expression for $\beta_m$:

$$\beta_m = \frac{1}{2} \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right) \tag{2.23}$$

Here, $\text{err}_m$ is the minimized weighted error rate:

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G_m(x))}{\sum_{i=1}^{N} w_i^{(m)}} \tag{2.24}$$

The model is then updated at step $m$ as follows:

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x) \tag{2.25}$$

For the next step $m+1$, the weights are updated before applying the procedure again:

$$w_i^{(m+1)} = w_i^{(m)} \exp(-\beta_m y_i G_m(x_i)) = w_i^{(m)} \exp(\alpha_m I(y_i \neq G_m(x))) \exp(-\beta_m) \tag{2.26}$$

Since $\exp(-\beta_m)$ is a constant for all weights, AdaBoost is equivalent to forward stagewise additive modeling under the exponential loss function.

Looking back at the setting of $G(x) \in \{-1, 1\}$ classification under the classification rule $G(x) = sign\{f(x)\}$,we have that observations where the sign of $y_i$ the target is equal to the sign of $f(x)$ we get $y_i f(x_i) > 0$ signalling correct classifications,and observations leading to $y_i f(x_i) < 0$ signals misclassification.The goal of the classifier is to penalize misclassification more then correct classifications.

Another loss function with the same population minimizer is cross-entropy, which is more robust than exponential loss, especially in chaotic datasets.

## 2.4 Gradient Boosting

Adopting a more general concept from Adaboost, gradient boosting allows the use of any differentiable loss function. It employs numerical optimization to find tree structures under such loss functions. The objective is to find:

$$\hat{f} = argmin_f\{L(f)\} \tag{2.27}$$

Here, $f \in \mathbb{R}^N$ represents the values of the function $f(x)$ for each training data point:

$$f = \{f(x_1), ..., f(x_N)\} \tag{2.28}$$

The general boosting idea still holds, and the final model is a sum of weak learners:

$$f_M = \sum_{m=0}^{M} h_m, \quad h_m \in \mathbb{R}^N \tag{2.29}$$

Starting with $f_0 = h_0$ as an initial guess, each $f_m$ is fitted based on $f_{m-1}$.

One method to implement gradient boosting is through steepest descent. Scikit-learn, for instance, implements gradient descent boosting through forward stagewise tree boosting which we will look at later.

Steepest descent is a greedy procedure where $h_m = -\eta_m g_m$, $\eta_m$ is a scalar step, and $g_m \in \mathbb{R}^N$ is the gradient of $L(f)$ evaluated at $f = f_{m-1}$.

The gradient components are given by:

$$g_{i,m} = \frac{dL(y_i, f(x_i))}{df(x_i)}\Big|_{f(x_i)=f_{m-1}(x_i)}, \quad i = 1, ..., N \tag{2.30}$$

The step length $\eta_m$ is optimized with respect to the loss:

$$\eta_m = argmin_\eta\{L(y, f_{m-1} - \eta g_m)\} \tag{2.31}$$

After computing the gradient and step length, the current solution is updated:

$$f_m = f_{m-1} - \eta g_m \tag{2.32}$$

Another method for gradient tree boosting is similar to forward stagewise tree boosting. The difference lies in the constraint; forward stagewise tree boosting is constrained to produce predictions with trees of a given depth.

We aim to solve:

$$\hat{\Theta_m} = argmin_{\Theta_m} \left\{ \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)) \right\} \tag{2.33}$$

For:

$$\Theta_m = \{R_{jm}, \eta_{jm}\}_{j=1}^{J_m} \tag{2.34}$$

given the current model $f_{m-1}(x)$.

To achieve this, we first find $R_{jm}$ before determining $\eta_j$ by solving:

$$\hat{\eta_{jm}} = argmin_{\eta_{jm}} \left\{ \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \eta_{jm}) \right\} \tag{2.35}$$

This is done by fitting a tree at the $m$-th iteration such that its predictions $t_m$ are as close as possible to the negative gradient. This is achieved using the squared error loss:

$$\tilde{\Theta_m} = argmin_{\Theta_m} \left\{ \sum_{i=1}^{N} (-g_{im} - T(x_i; \Theta_m))^2 \right\} \tag{2.36}$$

After finding the tree that solves the above problem, $\eta_{jm}$ is fitted with respect to the target loss function.

We find $\tilde{R_{jm}}$ instead of optimal $\hat{R_{jm}}$ due to computational reasons. In practice, $\tilde{R_{jm}}$ and $\hat{R_{jm}}$ are sufficiently similar.

Another aspect of implementing boosting trees is to limit the tree size, often done by setting the size of the tree the same, $J_m = J$, for each step in the boosting algorithm. The size is defined as:

$$N = 2L - 1 \tag{2.37}$$

Here, $L$ is the total number of leaf nodes in the tree.

$J$ is a tuning parameter for the algorithm. The number of boosting iterations $M$ is another tuning parameter. Running the algorithm for too many iterations can lead to overfitting. One can find an optimal $M^*$ for the specific dataset using cross-validation or early stopping based on a validation set, a common technique used in neural networks.

## 2.5 Variable importance

In addition to evaluating performance of the model we for tree based methods also understand what variables are more important for the prediction then others,using a common procedure of looking metrics based on evaluating how well the performance of including the variable was.

In context of a single decision tree the following is often used:

$$I_j^2(T) = \sum_{t=1}^{J-1} \hat{i}_t^2 \; l(v(t) = j) \tag{2.38}$$

where $j$ indicate the variable that the measure is for. $t = 1, ..., J-1$ are the indices of the internal nodes in the tree.

$v(t)$ is the splitting variable at node $t$.

$\hat{i}_t^2$ is the impurity improvement after splitting on node $t$.

$l()$ is function that measure impurity.Impurity of a node is a measure of how often a randomly chosen element would be incorrectly classified. The impurity decrease at each split is then aggregated to evaluate the overall impact of each variable's contribution to the model.

For methods based on bagging the method can be extended by taking the average over the trees:

$$I_j^2 = \sum_{b=1}^{B} \frac{1}{B} I_j^2(T_b) \tag{2.39}$$

This has been shown to be more reliable than its original single-tree counterpart do to the average.

## 2.6 Tree ensemble algorithms

**Algorithm 1** Bagging (Bootstrap Aggregating)

1: **Input:** Dataset $D$ with $N$ samples.
2: **for** $i$ from 1 to $B$ (number of base models) **do**
3: Sample $N$ instances with replacement from $D$.
4: Train a base model $M_i$ on the sampled dataset.
5: **end for**
6: **Output:** Ensemble model $E$ comprising base models $\{M_1, M_2, ..., M_B\}$.

Algorithm .1 is a pseudo code for the Bagging method.

**Algorithm 2** Random Forests

1: **Input:** Dataset $D$ with $N$ samples and $M$ features.
2: **for** $i$ from 1 to $B$ (number of trees in the forest) **do**
3: Sample $N$ instances with replacement from $D$.
4: Randomly select $m$ features.
5: Train a decision tree $T_i$ on the sampled dataset using the selected features.
6: **end for**
7: **Output:** Random Forest model $RF$ comprising decision trees $\{T_1, T_2, ..., T_B\}$.

Algorithm .2 is a pseudo code for the Random Forest method.

**Algorithm 3** Gradient Boosting

**Input:** Dataset $D$ with $N$ samples, loss function $L(y, F(x))$, and learning rate $\eta$.
Initialize target values: $F_0(x_i) = \text{argmin}_\eta \sum_{i=1}^{N} L(y_i, \eta)$.
**for** $m$ from 1 to $M$ (number of observations) **do**
    **for** $i$ from 1 to $N$ (number of observations) **do**
        Compute negative gradient: $r_{mi} = -\frac{\partial L(y_i, F_{t-1}(x_i))}{\partial F_{t-1}(x_i)}$ for all $i$.
        Train a base model $tree_m$ to the target $r_{mi}$ given terminal regions $R_{jm}$.
        **for** $j$ from 1 to $J_m$ (number of regions) **do**
            Learn optimal learning rate:
            $\eta_{jm} = \text{argmin}_\eta \sum_{x_i \in R_{jm}} L\left(y_i, F_{t-1}(x_i) + \eta\right)$.
        **end for**
    **end for**
    Update target values: $F_m(x_i) = F_{m-1}(x_i) + \eta_m \cdot tree_m(x_i)$.
**end for**
**Output:** Gradient Boosting model $GB$ comprising sum of base models.

Algorithm .3 is a pseudo code for the Gradient Tree Boosting method.

# CHAPTER 3

# Simulation study

Our simulation study begins by comparing the accuracy of decision trees, bagging, random forest, and boosting.

We will be utilizing the "Chess(King-Rook vs. King)" dataset from the USI dataset repository for our investigation. This dataset focuses on the endgame scenario of a rook king vs. king chess match. It encompasses six features, two for the column and row positions of each piece. Our primary objective is to perform classification between a draw and checkmate, categorized from 0 to 16. The response variable is hence categorical, encompassing 18 values ranging from 0 to 18, where 0 represents a draw, and 1 signifies checkmate in 0 up to 16 moves.

This dataset was specifically selected due to its suitability for tree structures. Our goal is to explore and validate this characteristic by employing a Feedforward Neural Network (FFNN) with varying regularization parameters and the number of hidden layers.

Our analysis begins by executing decision tree, bagging, random forest, and XGBoost— a widely used optimized gradient boosting method—using built-in packages from scikit-learn. We perform this analysis by exploring multiple hyperparameter configurations through 5-fold cross-validation with the RandomizedSearchCV package on the training set. Subsequently, we calculate the confusion matrix, accuracy, and variable importance plot for each of these tree ensemble methods.

To accommodate the substantial dataset, consisting of more than 28056 data points, we partition it into test and training sets a 80-20 split. The RandomizedSearchCV package includes a "score" parameter, allowing us to specify the metric for selecting the best model. We assess accuracy, f1 macro, and f1 micro metrics, conducting 20 iterations for RandomizedSearchCV. In each iteration, the package tests random predefined hyperparameters, selected by us, and identifies the best model for that specific run through 5-fold cross-validation on the randomly chosen configuration.

Our predefined hyperparameters include:

```
# Define parameter distributions for XGBoost
param_dist_xgb = {
    'learning_rate': [0.01, 0.1, 0.2,0.5,0.7],
    'n_estimators': [ 30,40,20,40,60,80,200,300,400],
    'max_depth': [2,5,8,10,15],
    'subsample': [0.8,  1.0],
    'colsample_bytree': [0.8,  1.0],
    "lambda":[0.001,0.1,0.5,0.9, 1,1.1],

}

# Define parameter distributions for RandomForest
param_dist_rf = {
    'n_estimators': [10,20,40,60,80,100, 150,200,250,500],
    'max_depth': [25, 30,20,30, 50,100,250],
    'min_samples_split': [4, 5,10,30,50,70],
    'min_samples_leaf': [1, 2,4,10,30,70],
    'criterion': ['gini', 'entropy'],  # Experiment with 'entropy'
    'class_weight': ['balanced', None]  # Adjust class weights
}

# Define parameter distributions for DecisionTree
param_dist_dt = {
    'criterion': ['gini', 'entropy'],
    'splitter': ['best', 'random'],
    'max_depth': [10,20,40,60,80,90,100,130,150,250,350],
    'min_samples_split': [2,3,4, 5, 6,10,30,70 ],
    'min_samples_leaf': [1, 2,4,10,30,70]
}

# Define parameter distributions for Bagging
param_dist_bagging = {
    'n_estimators': [10,20,40,60,80,100,150,200,300,400,500],
    'max_samples': [0.8, 1.0],
    'max_features': [1.0],

     # Adjust class weights
}
```

Figure 3.1: Hyperparameters for tree-based methods

Furthermore, we compute confusion matrices for the Feedforward Neural Network (FFNN). For each method, we report the best confusion matrix. We explore regularization parameters of 0, 0.3, and 0.7. The hidden layers we consider are (64, 32), (64, 64, 64, 32), (64, 64, 64, 64, 64, 32), and (128, 128, 128, 128, 128, 128, 32).

Further we will compute confusion matrices for the FFNN and for each method we will report the best confusion matrix.The regularization parameter we will test is 0,0.3 and 0.7.The hidden layers we will test is: (64,32),(64, 64,64,32),(64, 64,64,64,64,32) and (128, 128,128,128,128,128,32).

A minor simulation error occurred, wherein the images in the report were generated from a different simulation than the Jupyter Notebook script. However, it's essential to note that they still accurately depict the same rankings of the methods and the complexity of the models.

For the sake of brevity in the report, we have opted not to include all results. Instead, we will provide references to the Jupyter Notebook scripts when discussing specific sections that are not covered in the report.

## 3.1 Accuracy

We will use the accuracy score for our classification tasks, representing the proportion of correctly classified instances over the total number of instances. It

is calculated as the ratio of true positives and true negatives to the sum of true positives, true negatives, false positives, and false negatives. In mathematical terms, the accuracy score ($Acc$) can be expressed as:

$$Acc = \frac{\text{True Positives} + \text{True Negatives}}{\text{True Positives} + \text{True Negatives} + \text{False Positives} + \text{False Negatives}}$$

The accuracy score ranges from 0 to 1, where 1 indicates perfect classification accuracy, and 0 indicates no correct classifications. When we have imbalanced data i.e one class is highly represented compared to the others the accuracy will be high regardless of whether the other classes are predicted well or not.In such cases, additional metrics like precision, recall, or F1 score.

## 3.2 Precision and Recall

Before we introduce the f1 micro score and f1 macro score we will start by looking at the components of these scores which is precision and recall.

**Precision:** Precision measures the accuracy of the positive predictions made by the model. It is the ratio of true positive predictions to the total number of positive predictions (true positives and false positives). Precision is calculated using the formula:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

A high precision value indicates that the model has a low rate of false positives, meaning that when it predicts a positive outcome, it is likely to be correct.

**Recall:** Recall, also known as sensitivity or true positive rate, measures the ability of the model to capture all the positive instances in the dataset. It is the ratio of true positive predictions to the total number of actual positive instances (true positives and false negatives). Recall is calculated using the formula:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

A high recall value indicates that the model can identify a significant proportion of the actual positive instances in the dataset.

## 3.3 F1 macro

The F1 macro score is a metric commonly used in classification tasks to assess a model's performance by considering both precision and recall across multiple classes.From section 3.1 we noticed that the accuracy are not well suited for imbalanced classes,which motivate the exploration of the f1 score types. The F1 macro score is calculated as the unweighted average of the F1 scores for each class. The F1 score for a single class is the harmonic mean of precision and recall:

$$F1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 macro score (*F1 Macro*) is then computed as the average of the F1 scores across all classes. Like accuracy, the F1 macro score ranges from 0 to 1, where 1 indicates perfect classification performance.F1 score penalizes the case where either the precision or the recall is low.Since we are computing the F1 score for each class separately before taking the average it is strict in its interpretation of what a good model is.It puts emphasis on each class individually to be predicted with better accuracy.

## 3.4 F1 micro

The F1 micro score is the other version we will use to evaluate the overall performance of a model across all classes. It is calculated by considering the total true positives, false positives, and false negatives across all classes.

**Calculation:** The F1 micro score is computed using the following formula:

$$\text{F1 Micro} = \frac{2 \times \text{Total True Positives}}{2 \times \text{Total True Positives} + \text{Total False Positives} + \text{Total False Negatives}}$$

The F1 micro score takes into account the contribution of each class to the overall performance, treating the classification problem as a collective task. The F1 micro score is less strict and depending on the class counts for the imbalanced data case,its happy to give a higher score as long as the classes with the highest counts gets predicted well.

This interpretation has to be looked through the lens of the fact that bought f1 macro and f1 micro has the goal of reducing bought the recall and the precision.

## 3.5 Result using scoring function accuracy in RandomizedSearchCV

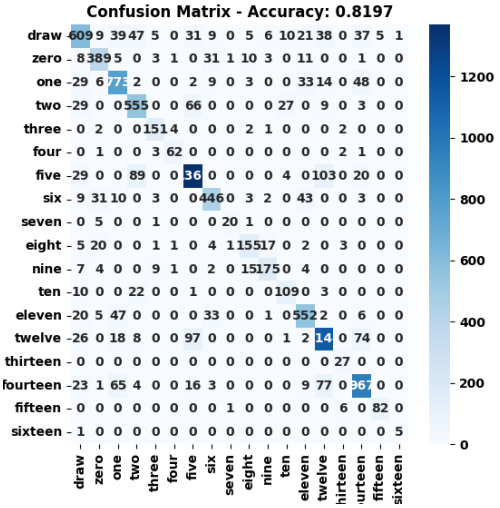


Figure 3.2: Best model result for DecisionTreeClassifier selection with accuracy

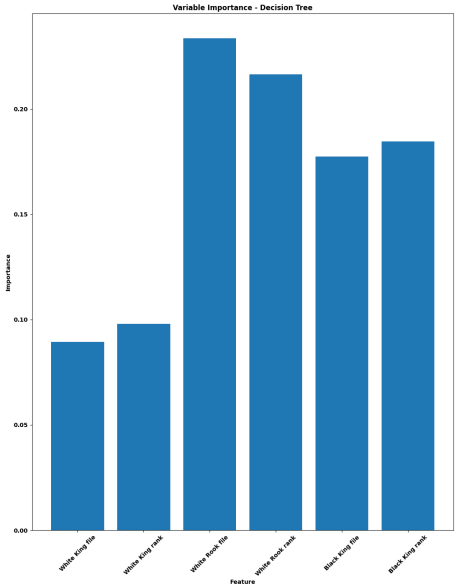


Figure 3.3: Best model informative variable result for DecisionTreeClassifier selection with accuracy

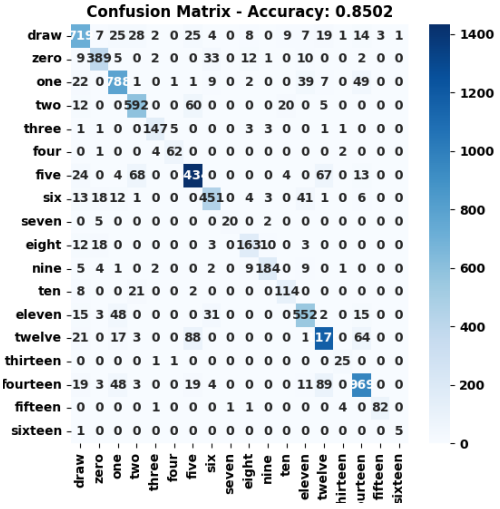


Figure 3.4: Best model result for BaggingClassifier selection with accuracy

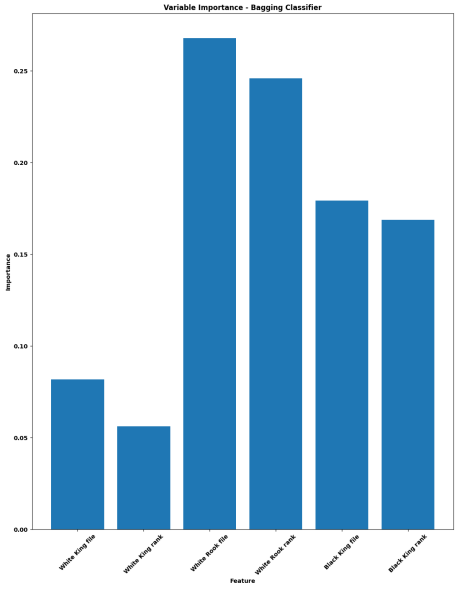


Figure 3.5: Best model informative variable result for BaggingClassifier selection with accuracy

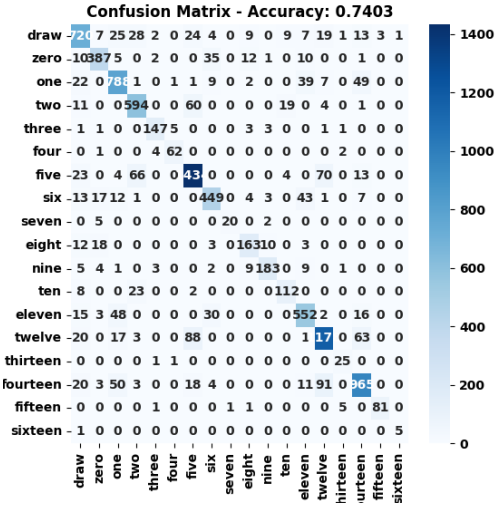


Figure 3.6: Best model result for Randomforestclassifier selection with accuracy

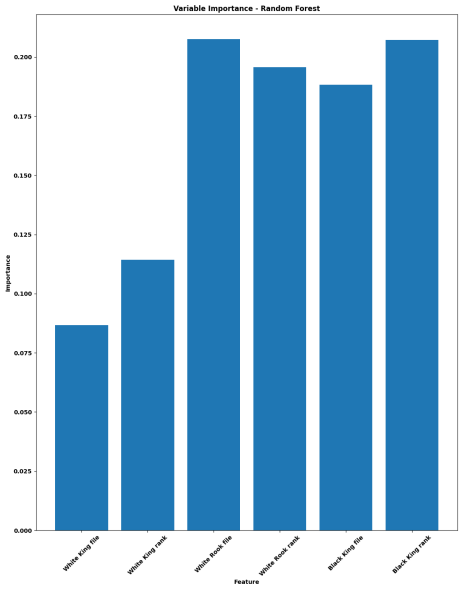


Figure 3.7: Best model informative variable result for RandomForestClassifier selection with accuracy

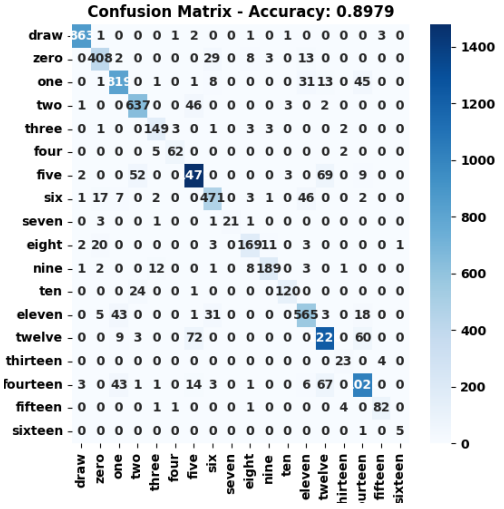


Figure 3.8: Best model result for XGBClassifier selection with accuracy

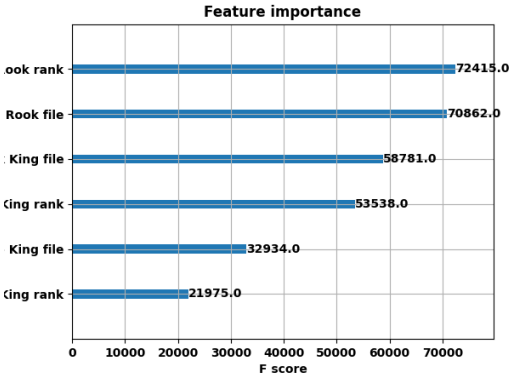


Figure 3.9: Best model informative variable result for XGBClassifier selection with accuracy

Rest of result can be found at: https://github.com/gituser1234566/some_coding/blob/main/project3fys-stk4155/tree_simualtions/krkopt_accuracy.ipynb

## 3.6 Result using scoring function F1 macro in RandomizedSearchCV

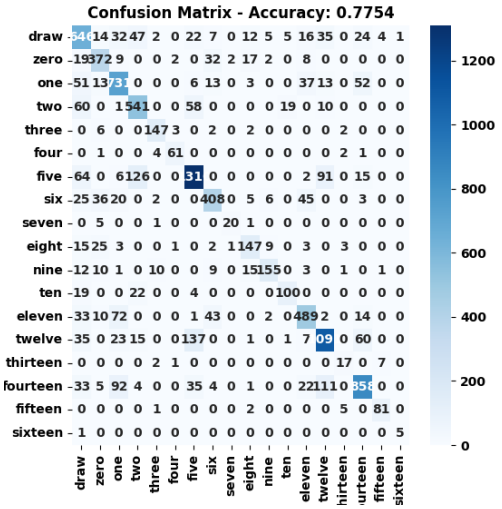


Figure 3.10: Best model result for DecisionTreeClassifier selection with F1 macro

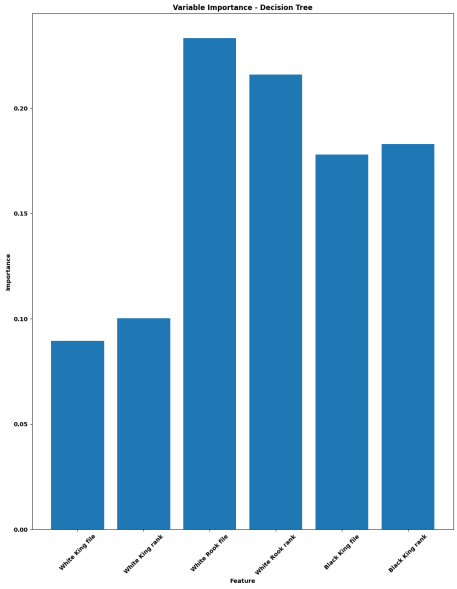


Figure 3.11: Best model informative variable result for DecisionTreeClassifier selection with F1 macro

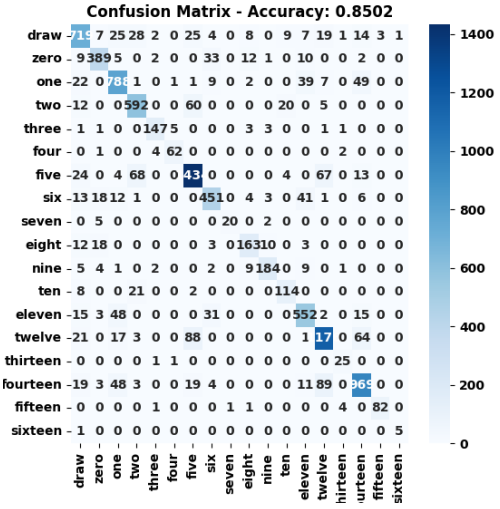


Figure 3.12: Best model result for BaggingClassifier selection with F1 macro

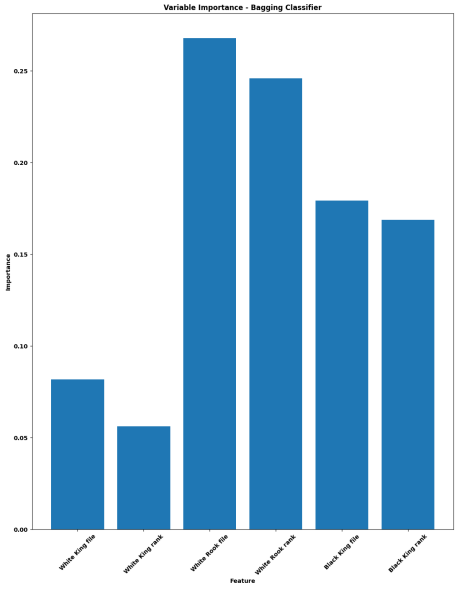


Figure 3.13: Best model informative variable result for BaggingClassifier selection with F1 macro

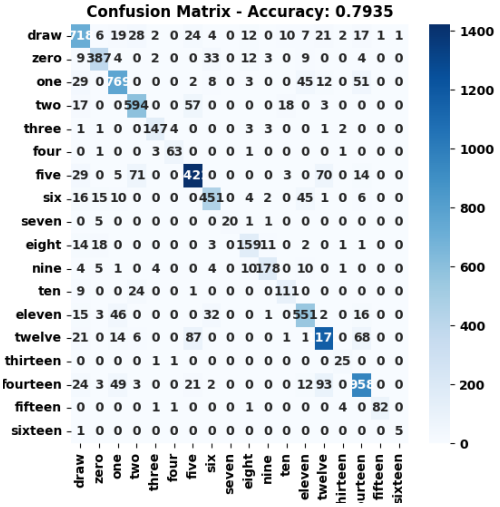


Figure 3.14: Best model result for Randomforestclassifier selection with F1 macro

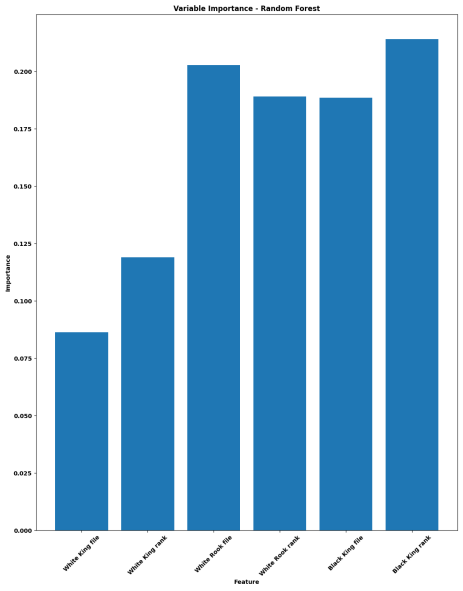


Figure 3.15: Best model informative variable result for RandomForestClassifier selection with F1 macro

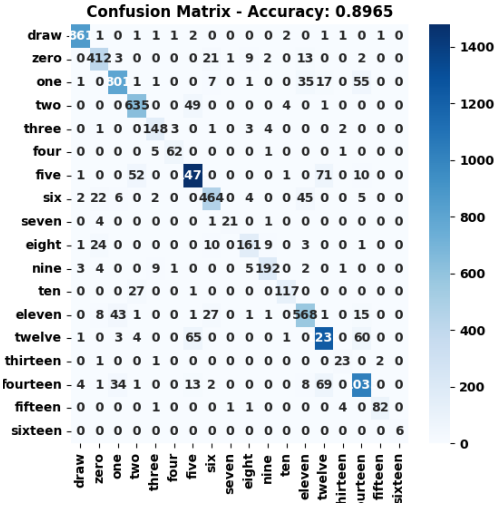


Figure 3.16: Best model result for XGBClassifier selection with F1 macro

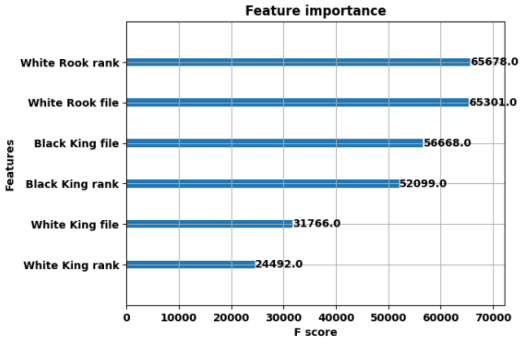


Figure 3.17: Best model result for XGBClassifier varaible importance plot selection with F1 macro

Rest of result can be found at: https://github.com/gituser1234566/some_coding/blob/main/project3fys-stk4155/tree_simualtions/krkopt_f1_macro.ipynb

## 3.7 Result using scoring function F1 micro in RandomizedSearchCV

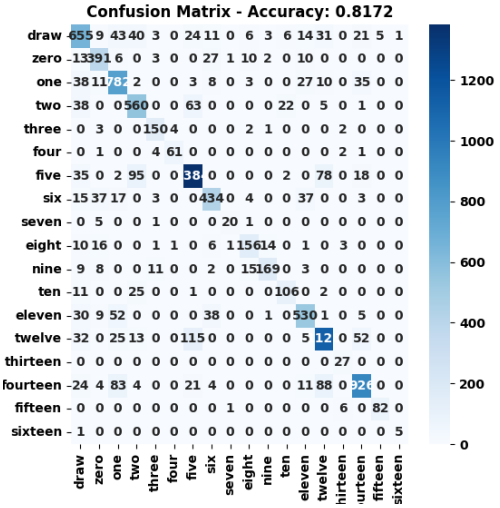


Figure 3.18: Best model result for DecisionTreeClassifier selection with F1 micro

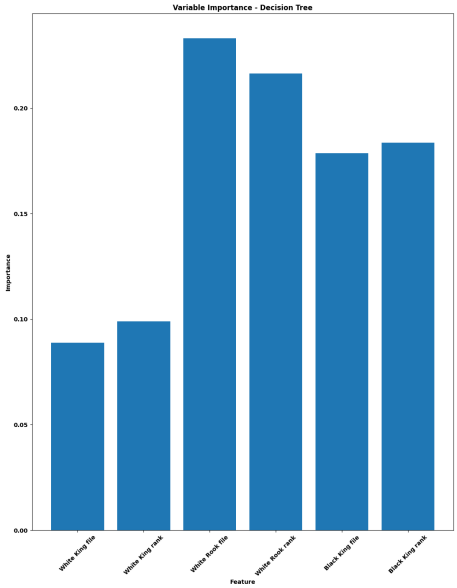


Figure 3.19: Best model informative variable result for DecisionTreeClassifier selection with F1 micro

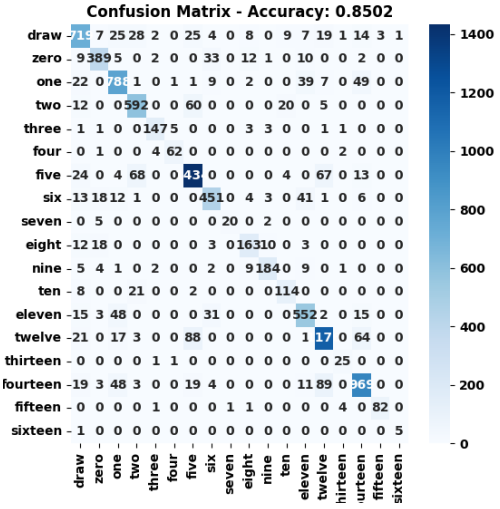


Figure 3.20: Best model result for BaggingClassifier selection with F1 micro

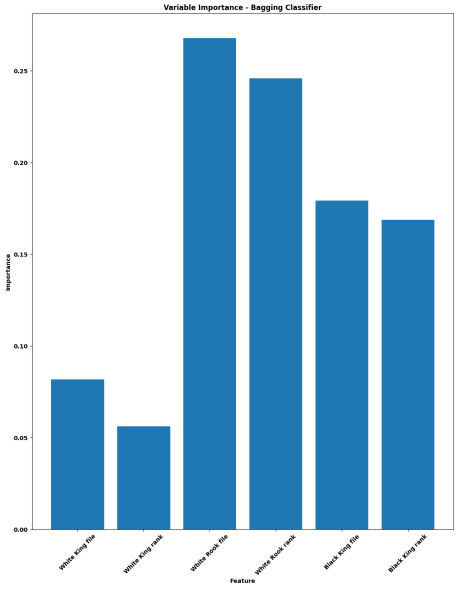


Figure 3.21: Best model informative variable result for BaggingClassifier selection with F1 micro

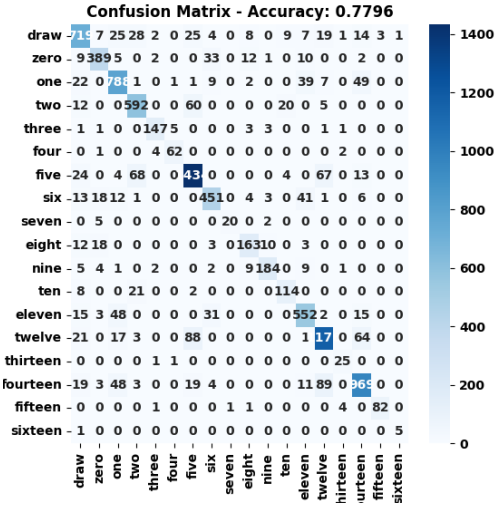


Figure 3.22: Best model result for Randomforestclassifier selection with F1 micro

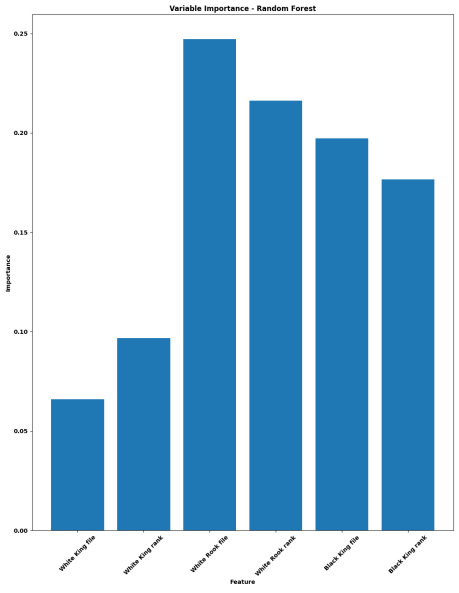


Figure 3.23: Best model informative variable result for RandomForestClassifier selection with F1 micro

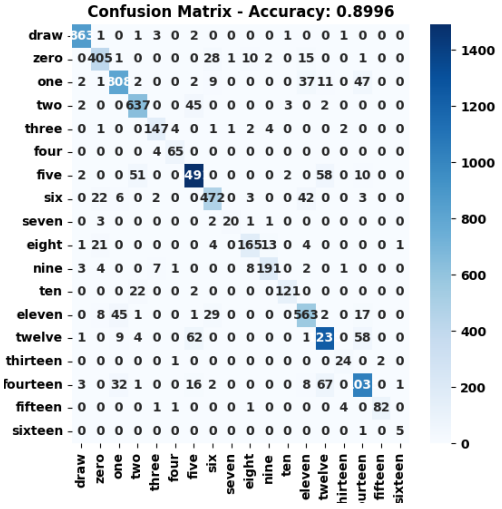


Figure 3.24: Best model result for XGBClassifier selection with F1 micro

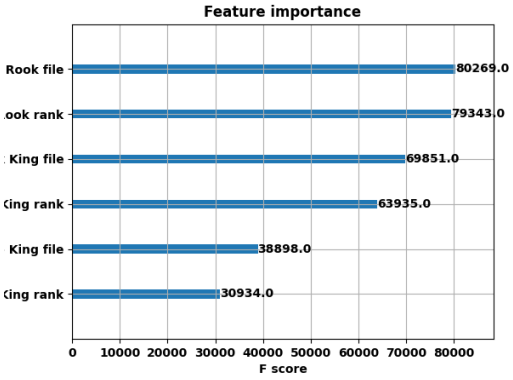


Figure 3.25: Best model informative variable result for XGBClassifier selection with F1 micro

Rest of result can be found at: https://github.com/gituser1234566/some_coding/blob/main/project3fys-stk4155/tree_simualtions/krkopt_f1_micro.ipynb

## 3.8 Result for FFNN

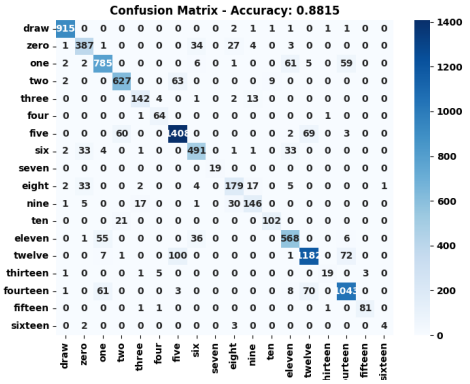


Figure 3.26: Best model result for FFNN selection with regularization parameter 0

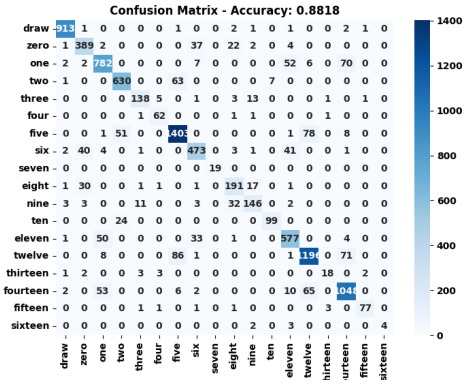


Figure 3.27: Best model result for FFNN selection with regularization parameter 0.3

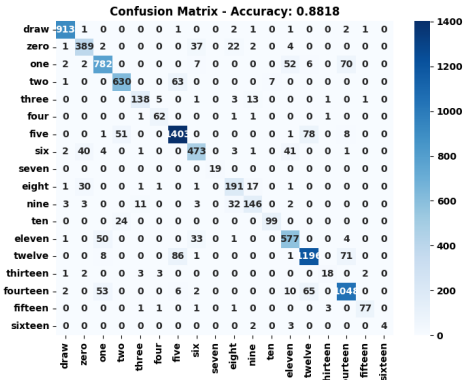


Figure 3.28: Best model result for FFNN selection with regularization parameter 0.7

Rest of result can be found at: https://github.com/gituser1234566/some_coding/blob/main/project3fys-stk4155/test_with_FFNN/krkopt-ffnn.ipynb

# CHAPTER 4

# Discussion

Analysis of the confusion matrices reveals that the top-performing Decision Tree model lags behind Bagging and XGBoost across all scoring criteria in RandomizedSearchCV, including *Acc*, $F1_{macro}$, and $F1_{micro}$. The best Decision Tree model outperforms Random Forest in terms of Acc and F1 macro scores. The optimal model, identified through scoring, exhibits a small $min\_samples\_split$, a small $min\_samples\_leaf$, and a $max\_depth$ exceeding 120.

Bagging demonstrates stability concerning the main parameter $n\_estimators$, with accuracies consistently surpassing 0.8 across all scoring criteria in RandomizedSearchCV. It stands as the second-best model when evaluated against the best model.

In Random Forest, setting $max\_features = None$ (the default) halts splitting when further divisions fail to yield improvements. The suboptimal performance may be attributed to insufficient correlated features. In a gaming context, omitting the most crucial feature can significantly impact predictions. Hyperparameters include a small $min\_samples\_split$, a modest $min\_samples\_leaf$, $n\_estimators = 250$ for F1 macro, and $n\_estimators = 60$ for F1 micro and accuracy.

Boosting emerges as the superior classifier, achieving an accuracy exceeding 0.89 in the test set simulation, with a Jupyter Notebook accuracy surpassing 0.9 for F1 micro scoring. The optimal model has $n\_estimators = 400$ and $max\_depth = 10$ for F1 micro,while there is a model in the same simulation scoring 0.885 for $n\_estimators = 40$ and $max\_depth = 8$. This pattern is apparent for XGboost for all score criteria,where the some more complex models are a little better then the more simple one.

The marginal improvements suggest that the data is not improper in the sense of number of data points in each class compared to other classes.

The most favorable outcomes are observed with F1 micro scores for XGBoost and Decision Tree accuracy. Depending on the simulation, F1 macro and F1 micro yield optimal accuracy for RandomForest. Boosting appears stable across different score criteria and $n\_estimators$.

For F1 micro, all important variable plots, excluding Decision Tree, indicate

that the most critical predictors for predicting game outcomes are the position of the white rook and, subsequently, the position of the black king.

For F1 macro, all important variable plots, excluding Decision Tree and RandomForest, highlight the significance of the white rook's position and the subsequent black king's position in predicting game outcomes.

For accuracy, all important variable plots, excluding RandomForest, underscore the importance of the white rook's position and the subsequent black king's position in predicting game outcomes.

Across all scoring criteria, Bagging and XGBoost assign diminished importance to the white king's position, a trend less evident in RandomForest and Decision Tree plots.

The stability of Bagging and the nature of the problem—a game with strong assumptions for victory—imply that selecting the wrong feature results in a loss, emphasizing hard constraints on importance rather than the freedom to vary variables in the model. Methods that naturally deviate from choosing the wrong combination may perform poorly.

The only method surpassing the FFNN, with an accuracy of 0.8818 for hiddenlayer config $(128, 128, 128, 128, 128, 128, 32)$, was XGBoost, with an accuracy of 0.8996. The Jupyter script includes the training procedure duration, revealing that FFNN, with 700 epochs for regularization parameters 0, 0.3, 0.7, for a specific hidden layer configuration, took 24 minutes. In contrast, all fits for tree ensembles required less time, highlighting one of the advantages of tree methods.

# CHAPTER 5

# Conclusion

In this study, we examined various tree ensemble methods within the realm of classification. Prior to delving into a small simulation that compares these methods and assesses their performance against an FFNN implementation in Keras, we explored the theoretical underpinnings of these methods. Notably, XGBoost and Bagging demonstrated a preferred assigning variable importance to the same factors when predicting the outcome of the endgame—specifically, the position of the white rook and the location of the black king.

Among the methods considered, XGBoost emerged as the most effective, while Bagging proved to be the most robust across different hyperparameters. In contrast, Decision Trees exhibited subpar performance compared to the aforementioned methods. The theoretical explanation for this discrepancy could lie in the inherent variance in tree fitting. Random Forest, despite its capabilities, performed less optimally, possibly due to the omission of relevant features in its selection process.

It's noteworthy that both XGBoost outperformed our implementation of the FFNN for the tested dataset and models. Additionally, the tree-based models demonstrated quicker evaluation times, further solidifying their efficacy.

# Bibliography

[BH94] Bain, M. and Hoff, A. *Chess (King-Rook vs. King)*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C57W2S. 1994.

[HTF01] Hastie, T., Tibshirani, R. and Friedman, J. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.