Water Potability Prediction

Binary Classification
using
Deep Learning and Ensemble Methods

Bendik Nyheim, Marcus Moen, & Mira Mors

https://github.com/benyhh/FYS-STK4155

Department of Physics
University of Oslo, Norway
December 2021

Abstract

The objective of this work is to predict the potability of water studying a binary classification dataset [6]. The methods used for prediction are feed forward neural network, decision tree, decision tree with bagging, boosting and random forest. The classifiers of scikit-learn are utilized for the implementation, and the results are obtained by k-fold cross validation with $k = 5$ folds. After an intensive grid search, all methods yield similar results.

Bagging performs best on the validation data with a validation accuracy of 70.9%. Here, the optimal parameters are 100 decision trees with a max depth of 14. The decision tree performs the worst with an validation accuracy of 65.2% with a max depth of 6. Notably, the optimal architecture of the neural network is small with one hidden layer and 10 neurons. Random forest reaches its optimum with a max depth of 14 as the least complexity. The boosting method also obtains its maximum with a max depth of 14 with estimators ranging from 40 to 100.

The F1 value across the methods is also very similar. For the non-potable class, bagging has the best F1 value of 0.78, while the decision tree gives the worst value which is 4% smaller. Since bagging is the best model, this method is chosen for prediction against the test set and gives an accuracy of 65.5%. That is a decrease by 8% compared to the validation accuracy. Accordingly, the F1 score dropped by 7%.

From the similar results of the different methods it can be deduced that the given data does not allow a better prediction of water-potability. Moreover, our results are consistent with previous studies [6].

Contents

1 Introduction **3**

2 Methods **3**

2.1 Feed Forward Neural Network . . . 3
2.2 Gradient Based Optimization . . . 5
2.3 Decision Tree . . . 7
2.4 Ensemble methods . . . 9
2.4.1 Bagging . . . 9
2.4.2 Boosting . . . 9
2.5 Random Forest . . . 9
2.6 Metrics . . . 10

3 Data **10**

3.1 Data Features . . . 11
3.2 Splitting of the Data . . . 13

4 Results **13**

4.1 Feed Forward Neural Network . . . 13
4.2 Decision Tree, Random Forest, Boosting, and Bagging . . . 15
4.3 Comparing Methods . . . 16
4.4 Test Data . . . 18

5 Discussion **18**

5.1 Feed Forward Neural Network . . . 18
5.2 Decision tree, Random Forest, Bagging, and Boosting . . . 19
5.3 Comparing Methods . . . 20
5.4 Test data . . . 20

6 Conclusion **20**

1 Introduction

Water is an essential source for life. About 70% of the Earth's surface is covered by water, most of which is seas and oceans. Only 3.5% of this is fresh water, while about 1.77% is bound up in the form of ice at the poles, glaciers and permafrost soils, and is therefore unavailable in the short run. Hence, fresh water is a limited resource. It is estimated that about two-thirds of the world's population face serious water shortages for at least one month a year. Almost half of them come from India and China [8]. As living standards are expected to rise, the resources of about three planets Earth will be needed to meet the demand for drinking water [5]. Meanwhile, drinking water sources are constantly polluted by natural and anthropogenic activities. More than four-fifths of the world's wastewater is released untreated into the environment [1]. The consequence, "dirty water causes the death of a human being every 10 seconds" [14]. It is therefore not far-fetched to develop methods that reliably determine the potability of water. In this paper we will use machine learning approaches to predict the potability of water. The dataset studied can be found here [6]. This binary set contains nine features plus the target feature. Earlier studies have shown a prediction accuracy of $\sim 70\%$ [6].

The method section first introduces the principles of a feed-forward neural network for classification. It is explained how to build a network using the MLP classifier from scikit-learn. For gradient-based optimization, the stochastic gradient descent and the adam's algorithm are explained in more detail. Decision trees with the concepts of Gini index and entropy are introduced. This is followed by the expansion to ensemble methods such as bagging and boosting. Then, we look at random forest. Also here the models are built with scikit-learn's classifier. The method section ends with the definition of important metric terms.
In the section 'data', the water-potability dataset is presented. In particular, the features and their distribution are shown. It is also explained how the data are processed.
The results include a grid search for the hyperparameters used to train the neural network. A similar grid search is performed for the decision tree, random forest, bagging and boosting. With the help of confusion matrices the different methods are compared. For the best model, a prediction is made based on the test data.
In the discussion, the results are evaluated. Concluding remarks are included at the end.

2 Methods

We will use different machine learning algorithms to analyze the dataset including feed forward neural network, decision trees, random forest, boosting and bagging.

2.1 Feed Forward Neural Network

The structure of an artificial neural network reminds of a biological nervous system insofar as it is made up of layers of interconnected neurons. We will use a feed forward neural network where the connection between the nodes is solely forward and thus they do not form a cycle. The simplest form is given by a single layer perception and is often used for classification tasks. The input data is fed into the layer, where the are multiplied by the weights and added with a bias. The strength of a particular node is given by the weights and with the bias value the activation function curve can be shifted up or down. The multiplied values are referred to as a weighted sum. The weighted sum is then applied to an activation function which is needed to map the input between required values like $(0, 1)$. A common activation function is the sigmoid.

By comparing the neural network's output with the true values, the weights and biases are trained to optimize the performance. This model performance is specified by the loss function. The training is based on gradient optimization such as stochastic gradient descent. If there are multiple layers, the training is extended using backpropagation. In principle, the computed output values in the final layer serve to adjust each hidden layer within the network. [15]

Equations of Backpropagation For a more detailed derivation of the backward propagation, we refer to our previous work where we also used a feed forward neural network [11]. We define w^l_{jk} as the weight for the connection from the kth neuron in the (l-1)th to the jth neuron in the (l)th layer. Analogously, b^l_k describes the the bias of the jth neuron in the (l)th layer. The corresponding activation of that node and layer, a^l_j, is given by applying the weighted sum to an activation function

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) \tag{1}$$

where σ represents the activation function and the sum runs over all neurons k in the (l-1)th layer. As we analyze a binary classification problem, the sigmoid function in the last layer is used since we want to map the outcome to be potable (0) or not (1). In matrix notation, the respective weights of each neuron and layer are stored in a weight matrix w^l at the entry w^l_{jk}. The biases for a layer are stored in a bias vector, b^l. Lastly, the activation vector a^l stores the the activations a^l_j. We get

$$a^l = \sigma\,\underbrace{(w^l a^{l-1} + b^l)}_{z^l} \tag{2}$$

where z^l is the weighted input.

Let C represent the cross entropy cost as the loss function (which is commonly used for binary classification problems)

$$C(\boldsymbol{w}) = -\sum_{j=1}^{n}\left(t_j \log a^L_j + (1-t_j)\log\left(1-a^L_j\right)\right) \tag{3}$$

where t_j are the targets. The derivative of the cost function with respect to the output a^L_j then is

$$\frac{\partial C(\boldsymbol{w})}{\partial a^L_j} = \frac{a^L_j - t_i}{a^L_j(1-a^L_i)} \tag{4}$$

It follows the equations of the backpropagation. Starting with the error in the output layer, δ^L

$$\delta^L = \Sigma^{'}(z^L)\nabla_a C \tag{5}$$

where $\Sigma^{'}(z^L)$ is a square matrix. It's only non-zero values are along the diagonal which are given by the values of $\sigma'(z^L_j)$. The error in an arbitrary hidden layer l is

$$\delta^l = \Sigma^{'}(z^l)(w^{l+1})^T \sigma^{l+1} \tag{6}$$

The change for a particular weight and bias is then given by

$$\Delta w^l_{jk} = -\eta a^{l-1}_k \delta^l_j \tag{7}$$
$$\Delta b^l_j = -\eta \delta^l_j \tag{8}$$

where η is the learning rate. [10]
The expressions for the change in weights and bias can now be substituted into the gradient descent equation (9) to update the weights and biases . Often the size of the weights is measured using the so-called L2 norm. This will add an additional shrinking parameter λ such that equation (7) has an additional $-\eta\lambda w^l_{jk}$ term. This is based on adding the term $\frac{\lambda}{2}\sum_j^n w_j^2$ to the cost function. The gradient descent algorithm thus favours smaller weights as constrained weights make the L2 term smaller (which then reduces the cost). Consequently, the network will use all of the weights as none of the weights will particularly dominate, which prevents overfitting. For more details about weight, bias, activation functions and initialization please see [11].

MLP Classifier We build our feed forward neural network with scikit-learn's multi-layer perceptron classifier. Among the parameters to set, there is the

- hidden_layer_sizes which defines the number of layers and the number of nodes
- max_iter which denotes the number of epochs
- activation which defined the activation function for the hidden layers
- solver which sets the algorithm for weight optimization across the nodes
- random_state for setting a seed for reproducing the same result

For a complete list visit the website of sklearn's MLP Classifier.

Scikit-learn initializes its weight with Glorot. That is, the weights are initialized uniformly within the range of $\left[-\sqrt{\frac{A}{f_{in}+f_{out}}}, \sqrt{\frac{A}{f_{in}}+f_{out}}\right]$ where is $A = 2$ if the activation function is a sigmoid and 6 otherwise. f_{in} and f_{out} denote the number of weights coming into going out of the layer [7]. The point is to replace the "small random value" approach by a more principled set of values thereby taking into account the architecture of the network.

Pseudocode The following algorithm shows a simple set up for scikit-learn's multi-layer perceptron classifier. For the actual implementation visit GitHub.

Algorithm 1: Building MLPClassifier

```
Require: Training data: X (design matrix), Y (true values)
Require: Validation data: X (design matrix), Y (true values)
/* Importing MLClassifier */
from sklearn.neural_network import MLPClassifier
/* Initializing the MLClassifier */
classifier = MLPClassifier(hidden_layer_sizes=(150,100,50), max_iter=300,
  activation='relu',solver='adam', radom_state=1)
/* Fitting the training data to the network */
classifier.fit($X_{train}$, $Y_{train}$)
pred = classifier.predict($X_{val}$)
```

We will also implement the Keras Classifier as it supports even more settings and a wider choice of parameters. For the actual implementation we refer to our code on GitHub. Additional settings are the dropout rate and weight constraint. The dropout specifies the probability of setting each input to the a given layer to zero. It prevents overfitting. Another method to reduce overfitting is done by applying weight constraints. Using the maxnorm, the weights are forced to have a magnitude at or below a given limit.

In order to find the optimal hyper-parameters, we both implement our own grid search and use scikit-learns's GridSearchCV.

2.2 Gradient Based Optimization

The idea of gradient-based optimization is that the derivative of the cost (which is as a function of a parameter $C(\theta)$) indicates how θ must be changed to reduce the error. Minimization of the error is achieved by moving in the direction of the negative gradient. The general formula for updating the parameter from an initial starting point θ_j is

$$\theta_{j+1} = \theta_j \underbrace{-\eta\nabla_\theta C(\theta)}_{\Delta\theta} \tag{9}$$

where $\Delta\theta$ represents the optimizing shift. η is the learning rate, a positive scalar which allows to scale the size of the step.

Stochastic Gradient Descent An extension to the simple gradient descent is the stochastic gradient descent. At the name implies, the method randomly uses only a subset of the given training data to compute the gradient. The subset is referred to as a mini-batch, B_k. It can be shown that the equation (9) can be written as a sum over the randomly selected mini-batches

$$\theta_{j+1} = \theta_j - \eta \sum_{i \in B_k}^{n} \nabla_\theta c_i(\mathbf{x}_i, \theta) \tag{10}$$

where η_j is the learning rate and n denotes the number of iterations. The method can be extended by adding a momentum parameter which is meant to increase the gradient in the right directions in order to reach faster convergence. We have

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta C(\theta) \tag{11}$$
$$\theta_{t+1} = \theta_t - v_t, \tag{12}$$

where γ is the momentum parameter which takes values bween 0 and 1. The pitfall of this method is that we do not know whether we miss a local minima if the applied momentum is too much. Thus, one would oscillate back and forward between the local minima.
Sklearn's MLP classifier uses **Nesterov momentum** which is a small change to normal momentum. The gradient is not computed from the current position θ_t, but from an intermediate position $\theta_{\text{intermediate}} = \theta_t + \gamma v_t$. It considers that the momentum may point in the wrong direction while the gradient term always points in the right direction. In other words, if the momentum term overshoots or points in the wrong direction, the gradient can make up for the mistake and "go back" in the same update step. Summarized the formula reads

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta C(\theta + \gamma v_{t-1}) \tag{13}$$
$$\theta_{t+1} = \theta_t - v_t \tag{14}$$

The main advantage of the Nesterov accelerated gradient is that it allows a larger learning rate for the same choice of γ [4].

Adam The adaptive moment estimation (adam) optimization algorithm can be used instead of the classical stochastic gradient descent. It is said to perform best on average. Both properties of AdaGrad and RMSprop are used in order to converge faster. **AdaGrad** allows to change the learning rate over time. It can thus adapt to differences in the datasets by allowing smaller or lager updates according tho how the learning rate is defined. The equation is given by

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\epsilon + \sum_{\tau=1}^{t} (\nabla C(\theta_\tau))^2}} \nabla_\theta C(\theta_t) \tag{15}$$

ϵ makes sure that the denominator never becomes zero and usually has a value of 10^{-8}. The sum runs over all the gradients squared for all the times-steps τ up to the current time step t. As time passes, the sum gets larger as more gradients are added and the learning rate decreases over time; the learning rate adapts. **RMSprop** uses an exponentially decaying average instead of a sum of gradients. The formula states

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\epsilon + \mathbb{E}[g^2]_t}} g_t \tag{16}$$

where we have defined $\nabla C(\theta_\tau))^2 = g(\theta_\tau)^2$. (If there are multiple paramters θ_i, g_i is the gradient with respect to each parameter θ_i). $\mathbb{E}[g^2]_t$ is the expectation of the gradient and is responsible for the decaying average. The so-called running average of the squared gradients depends only on the previous average and the current gradient

$$\mathbb{E}[g^2]_t = (1 - \gamma) g_t^2 + \gamma \mathbb{E}[g^2]_{t-1} \tag{17}$$

where γ is the momentum parameter introduced earlier. In AdaGrad, the learning rate decreases monotonously, while in RMSprop, the learning rate can adapt up and down in value. With these definitions, the update rule for adam algorithm comprises the decaying average of past squared gradients and past gradients

$$\theta_{t+1} = \theta_t - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{18}$$

$$\text{where} \quad \hat{m}_t = \frac{m_t}{1-\beta_1^t} \text{ and } \hat{v}_t = \frac{v_t}{1-\beta_2^t} \qquad \beta_1, \beta_2 \in [0,1) \tag{19}$$

$$\text{with} \quad m_t = (1-\beta_1)g_t + \beta_1 m_{t-1} \tag{20}$$

$$v_t = (1-\beta_2)g_t^2 + \beta_2 v_{t-1} \tag{21}$$

where β_1 and β_2 are exponential decay rates for the moment estimates and are similar to the γ term. Commonly $\beta_1 = 0.9$ and $\beta_2 = 0.999$. β^t simply means to compute β to the power of t. m is the decaying average of past gradients and v is the decaying average of past squared gradients. Initially, m and v are zero. $\hat{m}$ and $\hat{v}$ are bias-corrected terms [13].

Pseudocode The following algorithm shows how the adam algorithm can be implmented [2].

Algorithm 2: Adam algorithm

```
Result: Optimized parameter $\theta$ for minimizing a cost described by $C(\theta)$
Require: Initial guess of $\theta$
Require: Cost function $C(\theta)$
Require: Exponential decay rates $\beta_1$, $\beta_2$
Require: Learning rate $\eta$
$m_0 \leftarrow 0$ // initialize first moment vector
$v_0 \leftarrow 0$ // initialize second moment vector
$t_0 \leftarrow 0$ // Initialize time step
while $\theta_t$ not converted do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta C_t(\theta_{t-1})$
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$
    $\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$
    $\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$
    $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon$ // update parameters
end
```

2.3 Decision Tree

Decision trees can in principle be used for both regression and classification. They are based on finding those descriptive features that carry the most information related to the target feature. The dataset is then divided along the values of these features such that the target feature values for a given dataset are as pure as they can be. A decision tree is made up of a root node, interior nodes until it branches into the final leaf node(s). The leaf yields the classification outcome while a node defines a test of some attribute of the instance. A connecting branch gives the possible value of the attribute. Starting from the root node, an instance is classified by testing the attribute assigned to that node while moving down the tree branch and picking up the value of the attribute reaching the next node.

Thus, given our dataset described by its nine features and its target feature, the model is trained by continuously partitioning the target features along the values of the descriptive features using a measure of information gain. The tree grows until a stopping criteria is reached. The last node, the leaf node, is then used for prediction [7].

At every node, the best rule, that is the best attribute, for the current set of information available is selected. Here, best means the best way to separate the classes into two groups. Simply brute force is used to choose the best rule out of the possible candidate rules. While running though all possible combination of features and values, the algorithm makes continuous values discrete by binning. Then, the rule is applied to the training set, evaluating the purity of the splitted set.

Let the pdf p_{mk} be defined as fraction of a class k in a region R_m with N_m observations. The likelihood function is then given by the occurrence of the observations which belong to class k in the region R_m

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k) \tag{22}$$

p_{mk} is set to symbolize the majority class of obserations in region m. The splitting of a node is then determined by for example the **Gini index**

$$\text{Gini} = \sum_{k=i}^{K} p_{mk}(1 - p_{mk}) \tag{23}$$

where in a binary case $k = 1, 2$. A Gini index of 0 will mean a perfect split between the classes. Thus, the aim it so minimize the Gini index by choosing the candiate rule which gives the smallest Gini index.
Another splitting method is given by the information **entropy**

$$\text{entropy} = -\sum_{k=1}^{K} p_{mk} \log p_{mk} \tag{24}$$

Scikit-learn uses the CART algorithm which splits the dataset into two subsets using a single feature k and a threshold t_k.
The aim is to find the pair (k, t_k) which gives the purest subset defined by for example the Gini factor, G. The cost function which has ot be minimized is then

$$C(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}} \tag{25}$$

where $G_{\text{left/right}}$ determines the impurity of the left/right subset. $m_{\text{right/left}}$ denotes the number of instances in the left/right subset.
After successfully splitting the training set in two, the obtained subset is split analogously. The splitting is stopped until a maximum depth (given by the max_depth hyperparameter) has been reach or if the impurity cannot be further reduced. [3]

Desicion Trees often overfit the training data. Using a random forest can remedy this [7].

Pseudocode We use scikit-learn's decision tree classifier. The basic set up is given below.

Algorithm 3: Building DecisionTree Classifier

```
Require: Training data: X (design matrix), Y (true values)
Require: Validation data: X (design matrix), Y (true values)
/* Importing DecsionTree */
from sklearn.tree import DecisionTreeClassifier
/* Initializing the classifier */
classifier =DecisionTreeClassifier(max_depth=5)
/* Fitting the training data to the network */
classifier.fit($X_{\text{train}}$, $Y_{\text{train}}$)
pred = classifier.predict($X_{\text{val}}$)
```

2.4 Ensemble methods

Ensemble methods are a group of techniques where you use multiple machine learning models to make predictions. Here we will discuss the two main, and mostly used, ensemble methods, **bagging**, also known as bootstrap aggregation and **boosting**. As we will use scikit-learn's classifier, their implementation is analogous to the previous shown algorithms. See GitHub for the actual implementation.

2.4.1 Bagging

Selection with replacement is the random selection of a new training dataset from the original training dataset, where the same training dataset sample can be selected again. The newly selected dataset is called the bootstrap sample, which is used to train the respective decision tree. Bagging is when we take each of these new datasets and make models with them. All these models are made simultaneously. When you then get a new datapoint you give this datapoint to every model and let them vote on what label to give it. Letting the different models vote reduces the variance compared to a single model. Bagging can be used by various different models, but in this project we will only use bagging with decision trees as the base estimator. Examples of other models that can be used are support vector machines and regression models.

2.4.2 Boosting

Boosting is another commonly used ensemble method. While we in bagging made n new datasets using bootstrap resampling and thus made n models in parallel, boosting is done by making models sequentially, each new model "learns" from the previous model. Boosting is done by creating one model, then give weight to the data based on this model before making a new with this data. Observations that are misclassified are thus punished. The goal here is that the next model will hopefully learn from the mistakes made by the previous, and hence be better. Slow grown models, as the boosting method is, generally perform well.

2.5 Random Forest

The connection from decision trees to random forest can be summarized in three concepts

- Ensembles of classifiers and voting between them
- Resampling of the training set by selecting samples with replacement
- Selection of random feature subsets

Ensemble methods combine multiple decision trees to achieve better predictive performance than with a single decision tree. Selection with replacement is the random selection of a new training dataset from the original training dataset, where the same training dataset sample can be selected again. The newly selected dataset is called the bootstrap sample, which is used to train the respective decision tree. Bagging then refers to the collection of these newly obtained datasets. Using the average of all predictions from different trees reduces the variance compared to a single decision tree.

The models based on the collection of the newly sampled datasets forms a random forest. Features with high predictive power may dominate and cause the trees to be similar. Consequently, they have common weaknesses. This is mitigated by randomly selecting a subset of the features that a tree uses for training. This stops the correlation between trees and makes the forest more robust in general. Hence the word random in random forest. With n features, it is common that each tree uses only randomly selected $\sqrt{n}$ features for training. Again, the CART algorithm can be used to pick the best split among the given features.

2.6 Metrics

Metrics are used to measure and monitor the performance of a model during training and test.

Accuracy In this context accuracy refers to classification accuracy which is defiend as the ratio of number of correct predicitions to the total number of input samples

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions made}} \tag{26}$$

The disadvantage of accuracy score is that it can give false sense of achieving high accuracy. For example, if there are 98% samples of potable water and the rest is non-portable. The model can then easily get 98% training accuracy by predicting every training be potable.

Precision If the class distribution is imbalanced, it is wise to look at class specific performance as well. Precision is defined as

$$\text{Precision} = \frac{\text{True positive}}{(\text{True positive} + \text{False positive})} \tag{27}$$

Precision can be interpreted as a classifier's exactness. Many false positives can be indicated by a low precision.

Recall Recall is defined as the number of true positives divided by the number of true positives and the number of false negatives

$$\text{Recall} = \frac{\text{True positive}}{(\text{True positive} + \text{False negative})} \tag{28}$$

If the recall is low, the model predicts many false negatives.

F1 score The F1 score is the harmonic mean of recall and precision

$$\text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{29}$$

The score allows to find the optimal combination of the metrics recall and precision. The harmonic mean is used in order to punish extremes.

Confusion Matrix A confusion matrix summarizes the overall performance of the model by visualizing the model predictions versus the actual truth labels. There are four important terms for our binary classification problem. A True positive describes the case in which the model predicts that the water is potable which is actually true. Similarly, given a true negative, the model correctly predicts the water to be non-potable. A false negative (positive) is the case in which we predict the opposite of what is true; hence the false. [9]

3 Data

For this paper we are looking at ways to predict water potability. Potable water means water which is safe to drink. The data is split into two categories, the water is either potable or non-potable, either 1 or 0. Hence, we have a binary classification problem. This section will give a quick overview of how that data is distributed, which features we are looking at, and a quick discussion of the suitability of the data. The data is taken from this kaggle page [6]. There are a total of 3276 samples in this set, but 1265 of these samples have at least one missing feature value. We have, for this project, decided to just remove these values since we will still have a total of 2011 samples.

There are multiple other ways to handle missing values, filling them with the mean of the feature for example. There are both advantages and disadvantages to do this, all depending on how the data is distributed. However, this is not part of the curriculum and since the number of samples is still relatively large we think it will not hurt the analysis by removing these samples. Another important part of a classification task is to look at how the target values are distributed. As we can see in figure 1 the data is reasonably distributed. 59.7% of the data is non-potable while the remaining 40.3% are potable. This essentially means that the most basic model we can make, always predict the one with most values, will give us around 60% accuracy. Hence, we should hope our models to do a better job then this.

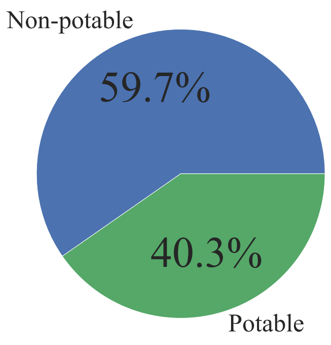


Figure 1: Pie chart to show the percentage of potability of the given data

3.1 Data Features

The dataset is characterized by nine features: pH, hardness [mg/L], solids [ppm], chloramines [ppm], sulfate [mg/L], conductivity [μS/cm], organic carbon [ppm], trihalmoethanes [μg/L], and turbidity [NTU]. Apparent from figure 2, all features have a very similar distribution of their values.. A description of the features of the dataset is given below.
The **pH** value indicates the acid-base balance of the water. According to the WHO, the maximum permissible range is a pH of 6.5 to 8.5. The given data has pH values ranging from 2.8 to 11.20. Calcium and magnesium salts which are dissolved from geologic deposits are the main factor influencing the **hardness** of water. The mean value of the hardness is around 200 mg/L which is considered very hard. Total dissolved **solids** (TDS) include all inorganic and organic minerals or salts that can be dissolved in water. They often cause a change in the color of the water and contribute to an undesirable taste. An optimum TDS limit should be less than 500 mg/L and no more than 1000 mg/L. Here, the average is about 20000 mg/L which is not realistic. **Chloramines** are commonly used for disinfection in public water systems. The concentration should be no more than 4 mg/L. The chloramines in the given data are as high as 13 ppm. **Sulfates** is a substance commonly found in minerals, soil and rocks. The safety limit is set to 500 mg/L. The majority of samples has a value of 300 mg/L. Substances solved in water are the reason for **conductivity**. The WHO standards state that the electrical conductivity should not be grater than 400 μS/cm. About half of the samples cross this limit. Total **organic carbon** (TOC) represents the total amount of carbon in organic compounds in pure water. The US environmental protection agency recommends a concentration less than 2 mg/L for drinking water. Here, the values are mostly around 14. If water is treated with chlorine, **Trihalomethanes** (THM) might be found as well. THM levels under 88 ppm are considered safe in drinking water. **Turbidity** describes the light emitting properties of water and is used to indicate the quality of wate discharge with respect to colloidal matter. [12]

Since some of the sample values do not correspond to the real safety limits, none of the samples should be drinkable. The model created is intended only as an exercise and should not be used for practice. Due to the wide range of the values for the different features it will be necessary to scale the data in order compare distributions.

The correlation matrix in figure 3 shows that the nine features are not correlated with one another. Thus, we can use all of the features as input.

All the boxplots of figure 4 show that the data is distributed as good as normal and not skewed as we already saw in figure 2. Figure 4c illustrates that the average score for solids has a much higher unit and that the data is much more dispersed. There are also quite a few outliers. This plot gives us a good reason for scaling the data since the values of the different features have different scales. Apparent from figure 4a, pH, chloramines and turbidity have the densest data. Overall, all the box plots show that the data is well distributed and that we do have some outliers.

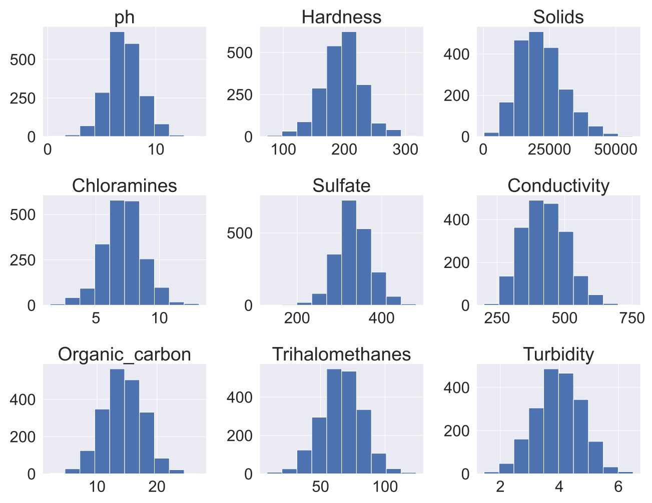


Figure 2: Features of water-potability set with their respective counts on the y-axis: pH, hardness [mg/L], solids [ppm], chloramines [ppm], sulfate [mg/L], conductivity [μS/cm], organic carbon [ppm], trihalmoethanes [μg/L], and turbidity [NTU].

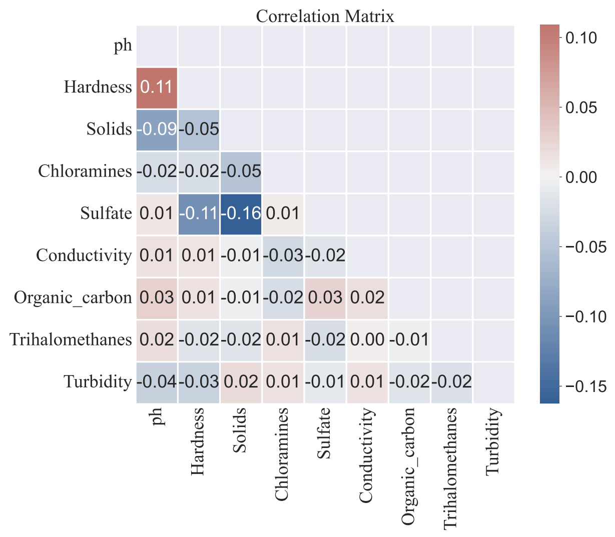
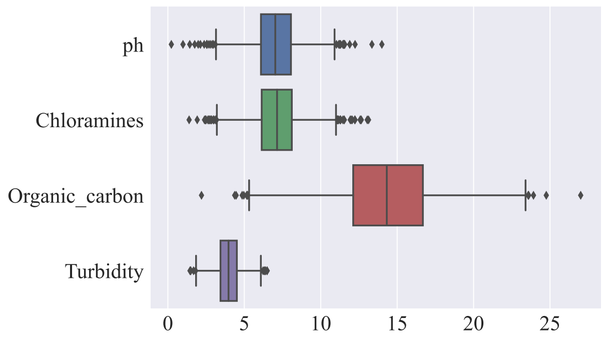
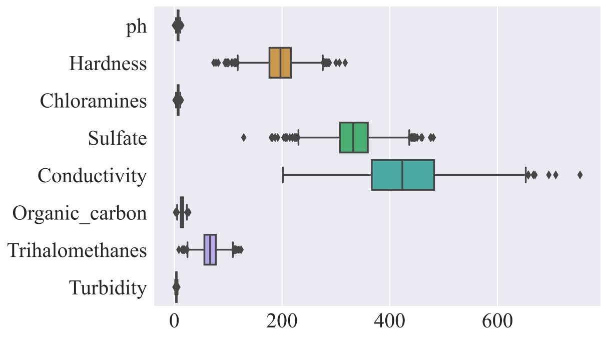


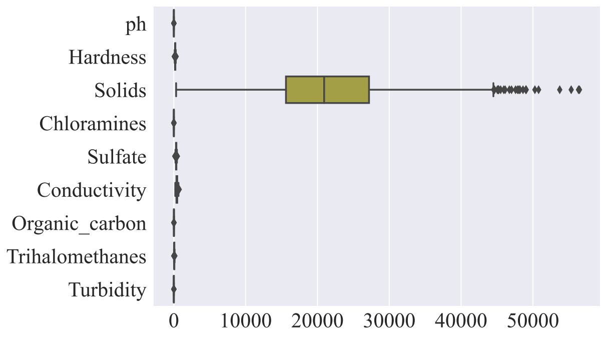
Figure 3: Correlation matrix of the features of water-potability data



(a) Closer look at ph, chloramines, organic carbon and trubidity



(b) Closer look at hardness sulfate, conductivity and trihalomethanes



(c) Closer look at the solids

Figure 4: Boxplots for the nine features in water-potability data

3.2 Splitting of the Data

For this project we have decided to split the data into three parts; training, validation, and test sets. These three sets have all different use cases. The train set is the biggest containing 60% of the data. This set is used to train the different models. The validation set is used when comparing the different models to each other. We want to both test different parameters and different models, to find the best of these we use the validation set and find the accuracy of the different alternatives. The test set is only used once. This set is unseen by all the different models until we have chosen a final model. This final model is then tested against the test set to obtain an unbiased estimate of how good the model is. Since we have used the validation set a lot and based our selection on the best model here this will definitely not be unbiased and hence, we need an unseen part of the data. The data is scaled using scikit-learn's standard scaler.

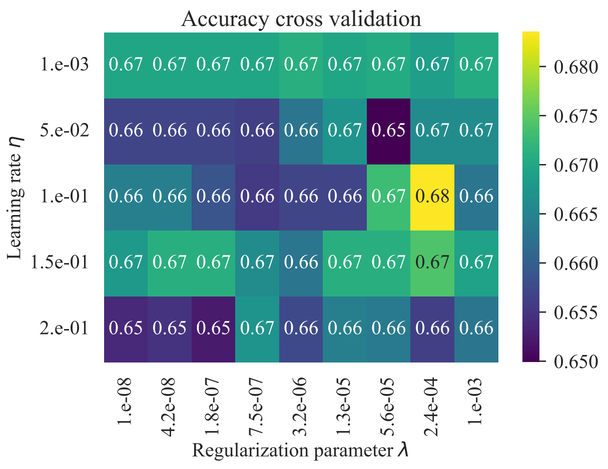
Note: When we are using K-fold cross validation we combine the training and validation sets.

4 Results

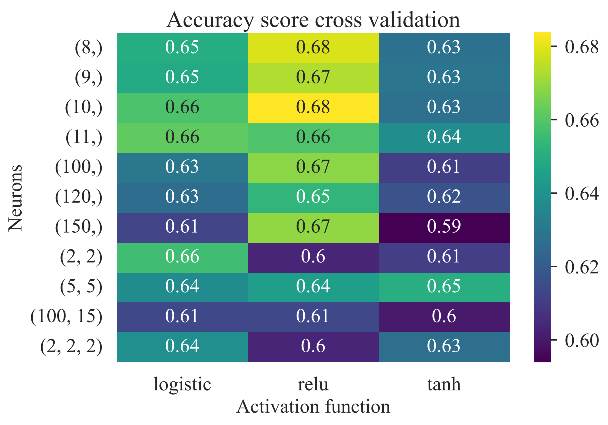
4.1 Feed Forward Neural Network

First, we present the results using scikit-learn's MLP Classifier (see section 2.1) with our own grid search. Computations were done using stratified k-fold with $k = 5$ folds thus preserving the percentage of samples for each class. Figure 5a shows the accuracy after running over the learning rate, $\eta = \{0.001, 0.05, 0.1, 0.15, 0.2\}$ and regularization parameter, $\lambda \in [10^{-8}, 10^{-3}]$ for a fixed number of epochs (150), batch size (50) and one hidden layer with 10 neurons. For figure 5b, the number of epochs is 150, batch size 50 and the regularization parameter is the optimal parameter of figure 5a with $\lambda = 2.4 \cdot 10^{-4}$. The number of hidden layers and the activation functions were varied. For example $(2, 2, 2)$ denotes three hidden layers with two nodes per layer. The logistic function is the same as the sigmoid function. Figure 5c illustrates the dependence of the accuracy on the number of epochs and the learning rate, $\eta = \{0.001, 0.05, 0.1, 0.15\}$ for a batch size of 50, optimal hidden layer size and activation function according to figure 5b ((10,) and relu) and optimal regularization parameter $\lambda = 2.4 \cdot 10^{-4}$. For figure 5a, 5b and 5c the solver for the gradient descent algorithm was adam. Figure 5d shows the accuracy as a function of the solver using either adam or stochastic gradient descent with and without Nesterovs momentum $\gamma \in \{0.3, 0.9\}$.

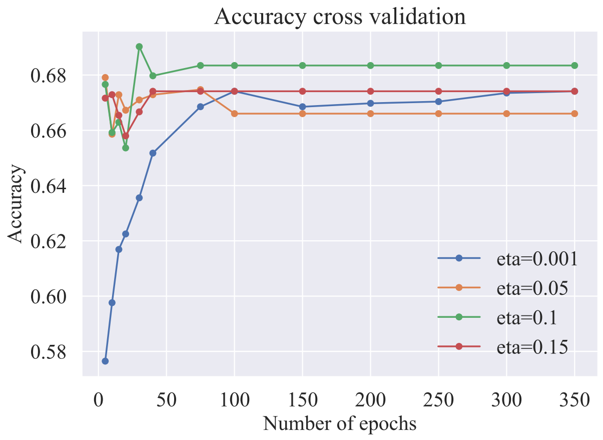
As the adam alogrithm was best, we performed a grid search using sklearn's GridSearchSV using cross validation with 5 folds varying the exponential decay rates ($\beta_1 = \{0.75, 0.8, 0.85, 0.9, 0.95\}, \beta_2 = \{0.9, 0.95, 0.999\}$) getting an accuracy of 0.69. The optimal parameters where $\beta_1 = 0.8$ and $\beta_2 = 0.9$. For comparison, the worst result in this grid search was 0.66. Note that 150 epochs were used in order to ensure convergence. Performing an intensive grid search with the keras classifier over, among others, epochs, batch size, number of neurons, learning rate, type of solver, kernel initialization, activation function, dropout rate and weight constraint does not yield to better but similiar results. These runs are summarized in tables in the support material.



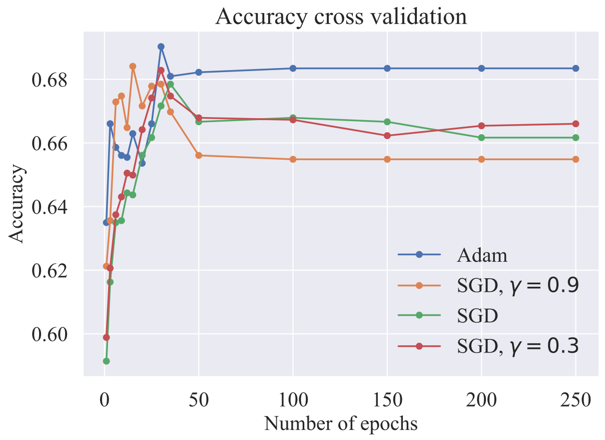
(a) epochs=150, batch size=50



(b) $\eta = 0.1$, epochs=150, batch size=50



(c) batch size = 50



(d) $\eta = 0.1$, solver adam: batch=50, solver sgd: batch=42, solver sgd with $\gamma = 0.3$: batch = 50, solver sgd with $\gamma = 0.9$: batch = 80

Figure 5: Mean accuracy score using cross validation with $k = 5$ folds on water-potability training data using a feed forward network. Besides for a), the regularization parameter $\lambda = 2.4 \cdot 10^{-4}$. Besides for b) relu is used as the activation function in the only hidden layer with 10 neurons. Besides for d) adam is used as solver.

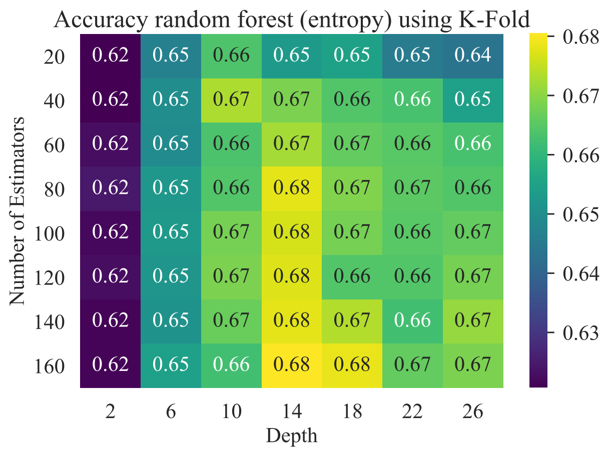
4.2 Decision Tree, Random Forest, Boosting, and Bagging

We can now go over to look at decision tree, random forest, bagging using decision tree, and boosting. Here, we have tested numerous models with different hyperparameters. All the results in this subsection is gained using K-Fold cross validation with $k = 5$. The implementations used in this section are sklearn's DecisionTreeClassifier, RandomForestClassifier, BaggingClassifier, and GradientBoostingClassifier.

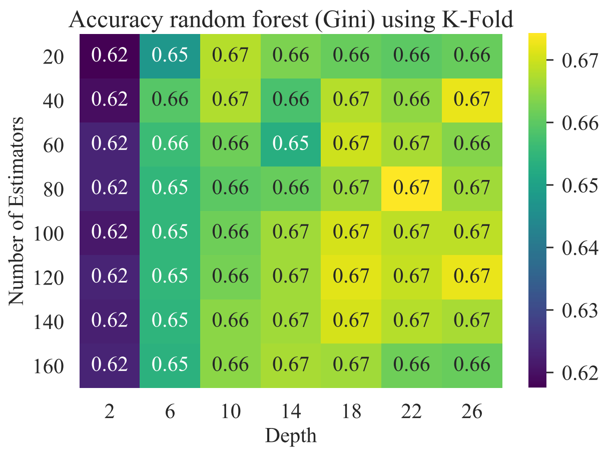
Figure 6 shows a grid search for two different random forest hyperparameteres, max depth and number of trees. Figure 6a shows for random forest with entropy, while figure 6b shows for random forest using Gini.

Figure 7 shows how the accuracy of different decision trees and random forests evolves with the max depth of the trees. We have tested decision tree and random forest using both the Gini index and entropy. The red and green line shows how the accuracy of random forest evolves with the max depth, for entropy and Gini respectively. The orange and blue shows the same for decision tree. We have done a search with depths ranging from 2 to 32.

After testing random forest and decision tree we went over to test bagging and boosting. We have used bagging with decision trees as the base estimator. Figure 8a shows grid search of different number of estimators and different max depths. The number of estimators means how many decision trees are we making with different bootstrap samples, while the max depth is the maximum depth for each tree. Figure 8b shows the same as figure 8a except here we have the results from using boosting. We have used boosting with a learning rate of 1, "deviance" as loss, and mean square error as criterion.



(a) Random forest using entropy



(b) Random forest using Gini

Figure 6: Heat maps for random forest with either entropy or Gini using training data. We are looking at both max depth and number of trees/estimators.

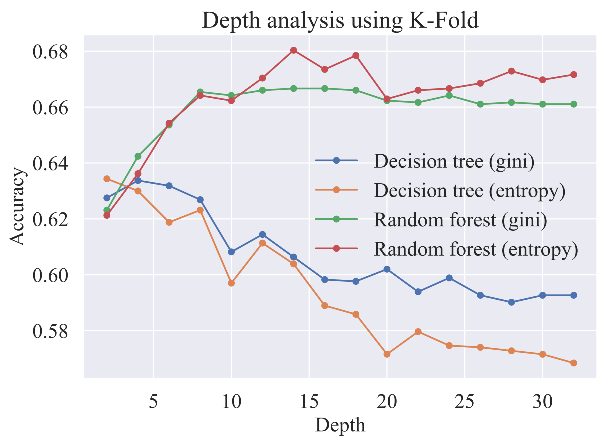
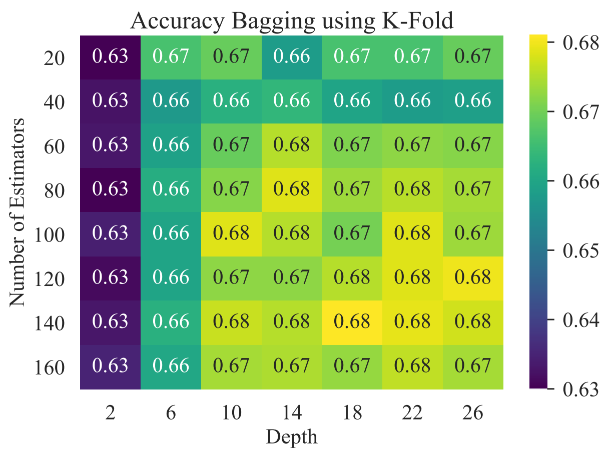
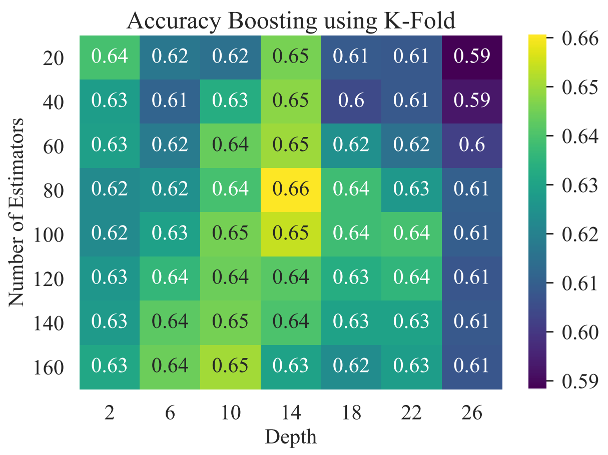


Figure 7: Analysis of different max depth for decision tree and random forest with 160 estimators on training data.



	2	6	10	14	18	22	26
20	0.63	0.67	0.67	0.66	0.67	0.67	0.67
40	0.63	0.66	0.66	0.66	0.66	0.66	0.66
60	0.63	0.66	0.67	0.68	0.67	0.67	0.67
80	0.63	0.66	0.67	0.68	0.67	0.68	0.67
100	0.63	0.66	0.68	0.68	0.67	0.68	0.67
120	0.63	0.66	0.67	0.67	0.68	0.68	0.68
140	0.63	0.66	0.68	0.68	0.68	0.68	0.68
160	0.63	0.66	0.67	0.67	0.67	0.68	0.67

(a) Bagging with decision trees as base estimators



	2	6	10	14	18	22	26
20	0.64	0.62	0.62	0.65	0.61	0.61	0.59
40	0.63	0.61	0.63	0.65	0.6	0.61	0.59
60	0.63	0.62	0.64	0.65	0.62	0.62	0.6
80	0.62	0.62	0.64	0.66	0.64	0.63	0.61
100	0.62	0.63	0.65	0.65	0.64	0.64	0.61
120	0.63	0.64	0.64	0.64	0.63	0.64	0.61
140	0.63	0.64	0.65	0.64	0.63	0.63	0.61
160	0.63	0.64	0.65	0.63	0.62	0.63	0.61

(b) Boosting with learning rate = 1, loss = "deviance", and mean square error as criterion.

Figure 8: Heat maps for bagging and boosting using training data. We are looking at both max depth and number of trees/estimators.

4.3 Comparing Methods

Figure 9 shows the confusion matrix for a network build by the MLPClassifier with 150 epochs, a batch size of 50, a learning rate of 0.1, a regularization parameter of $2.4 \cdot 10^{-4}$ with relu as the activation function with one hidden layer of 10 neurons. Predictions were done with the validation set.

Figure 10 shows the confusion matrices from decision tree, random forest, bagging, and boosting. These results are obtained by using the best models in the previous section on the validation data.

Table 1 shows various performance measures calculated from the confusion matrices. Here we see a clear comparison between all the five different models studied in this project. We see the accuracy for each model. Precision, recall, and the F1 score for the class non-potable is also added.

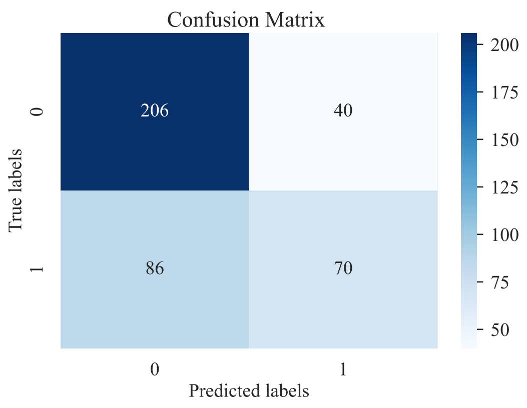
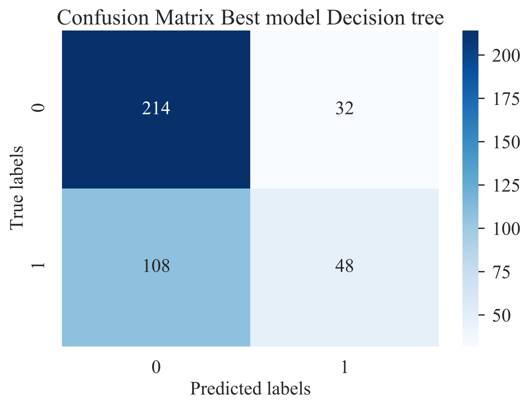
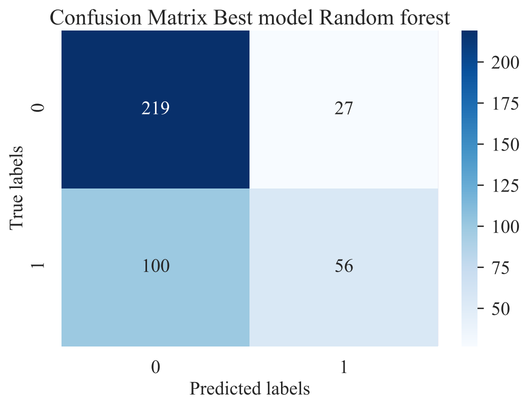


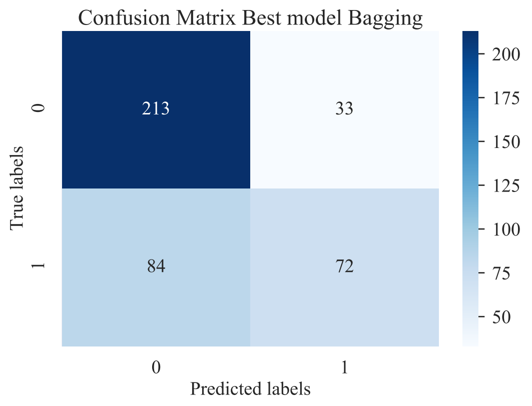
Figure 9: Confusion matrix after predicting water-potability on validation data using a feed forward neural network with epochs =150, batch size = 50, learning rate =0.1, regularization parameter = $2.4 \cdot 10^{-4}$, relu as activation function the only hidden layer with 10 neurons and adam as the solver.



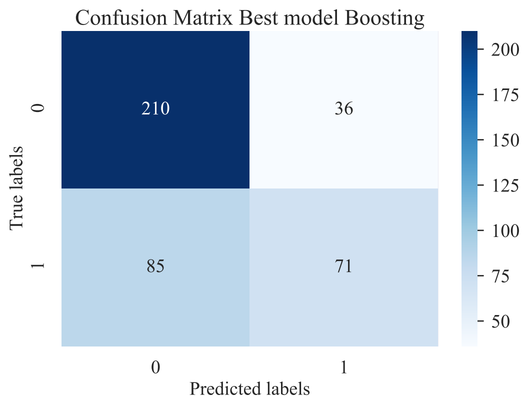
(a) Performance of decision tree with max depth 6 using the Gini index.



(b) Performance of random forest with max depth 14 using entropy.



(c) Performance of bagging using 140 decision trees with max depth 18.



(d) Performance of boosting using 40 estimators and a max depth of 14

Figure 10: Confusion matrices for decision tree, random forest, bagging, and boosting on the validation data.

Table 1: Performance on water potability validation data using neural network, decision tree, random forest, bagging, and boosting. Precision and recall are for the non-potable class.

	Accuracy	Precision	Recall	F1 Score
Neural Network	0.687	0.705	0.837	0.766
Decision Tree	0.652	0.665	0.870	0.754
Random Forest	0.684	0.687	0.890	0.775
Bagging	0.709	0.717	0.866	0.784
Boosting	0.699	0.712	0.854	0.777

4.4 Test Data

After running all the grid searches and testing the models against the validation data it is time to test the best model against the test set. Figure 11 shows the confusion matrix for the result after running the bagging model on the test data. While, table 2 shows the different performance results extracted from figure 11.

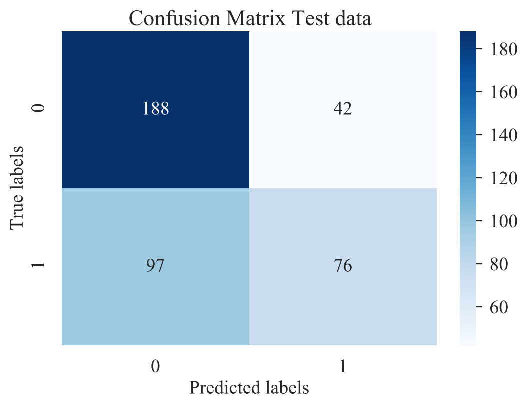


Figure 11: Confusion matrix after predicting water-potability of test data using bagging with 140 decision trees with a max depth of 18.

Table 2: Performance on water potability validation data using neural network, decision tree, random forest, bagging, and boosting. Precision and recall are for the non-potable class.

	Accuracy	Precision	Recall	F1 Score
Bagging (test)	0.655	0.660	0.817	0.730

5 Discussion

5.1 Feed Forward Neural Network

The general overview shows that the neural network does not approach 70% accuracy besides an intensive grid search.

Figure 5a shows that varying the learning rate and regularization parameter has a small influence on the accuracy from 65% to 68%. There is no direct trend discernible in the heatmap, but for higher learning rates a larger regularization parameter seems better. The best result is obtained with $\eta = 0.1$ and $\lambda = 2.4 \cdot 10^{-4}$.

Apparent from figure 5b, the neural network can be quite simple. The best result is obtained with one hidden layer that has 10 neurons when the activation function in the hidden layer is the relu function. A more complex architecture can quickly lead to overfitting. Figure 5a and figure 5b agree will in their results. Using the logistic function as the activation, the results are slightly worse with an accuracy of 66%. Now the optimal number of neurons is 11. The tanh function performs worst with an accuracy of 65%, but now the network consist of two layers with two node per layer. In general, the change in accuracy is minor and could partly be due to the randomness implied by the algorithm.

Looking at figure 5c it becomes clear that the number of epochs must be large enough such that the accuracy converges; especially if η is small. At a number of epochs equal to 100, all the learning rates beside $\eta = 0.001$ have converged. Again, $\eta = 0.1$ performs best and $\eta = 0.05$ worst which agrees with figure 5a. The peak at epoch $= 30$ for $\eta = 0.1$ can be explained by the fact that a minimum has been reached, but the algorithm jumpes out of it, oscillating forward and back as the iterations progress. This can be explained by the "high" learning rate. Whereas, for $\eta = 0.001$ at epoch 100 the algorithm hops out of a minimum, but as the learning rate is small enough, it is able descent down again into a minimum (maximising accuracy) at around epoch $= 300$.

However, $\eta = 0.1$ still yields the best result and the difference in terms of accuracy is small with about one percent. That $\eta = 0.1$ is best makes also sense as the regularization parameter is now fixed, using the optimal parameter from figure 5a.

Since only the adam solver has been used, figure 5d shows the accuracy also for the stochastic gradient descent with and without momentum. Still, the adam algorithm performs best. Indeed, the blue curve in figure 5d matches the green curve well in figure 5c. Again, since $\eta = 0.1$ for all the solvers, a jump at epochs $= 30$ can be observed. The cause is the same as already explained. Interestingly, using stochastic gradient descent without momentum performs better in the long run. At least until epochs $=$150. However, as the number of epochs is small, the stochastic gradient descent with a high γ performs better. This makes sense as the momentum parameter allows faster convergence. However, the danger is to hop out of a minimum. In the long run, the stochastic gradient solver with the high momentum performs worst with an accuracy slightly less than 66%. Additionally, depending on the solver used, the optimized batch size varies. The stochastic gradient descent with $\gamma = 0.9$ has the highest batch size with 80 and the simple sgd the smallest with 40.

As stated in the results section, an intensive grid search using keras has been done as well. The accuracy still did not reach the 70% accuracy mark.

5.2 Decision tree, Random Forest, Bagging, and Boosting

For the decision tree, random forest, bagging, and boosting we see some quite interesting results. We can start by looking at decision tree and random forest. As one can expect random forest does a better job then decision tree. This is expected since random forest is multiple decision trees, and made to reduce the variance of the model. In figure 7 we can see that, for decision tree, the accuracy decreases with an increase in the max depth. This can be explained by overfitting. As a tree gets deeper and deeper it also gets more and more complex.

Figure 6 shows the two heat maps for random forest. If we first look at 6a we see that when using entropy max depth 14 seems to stand out. In general, using entropy with max depth of around 14 to 18 with over 140 trees seems to give the best results. For 6b we see that we need fewer trees but more depth.

In figure 7 we see that one single tree gets to complex for the test data very quickly. The best results of decision tree seems to be around a max depth of $2 - 8$. Another thing to note here is that using entropy is overall worse then using Gini.

For random forest we see a little different picture. Here we see that as the max depth grows, the accuracy stabilizes. Using Gini or entropy seems to be of little importance, but Gini gives slightly better result. We can see that the maximum accuracy is when using Gini for random forest with a max depth of 14. What we see here makes sense as when we reach a given number of max depth, we will not get any more information and the accuracy should stabilize. When this happens we choose the model which gives us the best result with least complexity.

Figure 8 gives us the results from bagging and boosting. In figure 8a we can clearly see that we need a depth of over 6 to get decent results. It seems that the best area for bagging is by using around 100 estimators with a max depth between 14 and 26. In general, it seems like bagging gets decent results. Since we use bagging with decision trees it is very close, and comparable, to random forest. As we can see, it gives very close results to random forest.

In figure 8b (showing the results for boosting) we see that the best results seems to be at max depth 14 with the number of estimators ranging from 40 to 100. In this heatmap there is more variance than in 8a. It seems like the boosting method is more sensitive to the hyperparameters than the bagging method.

5.3 Comparing Methods

After finding the best hyperparameters for each model we made confusion matrices and found the performance scores for all of them on the validation data. Just by looking at the five confusion matrices we can see that the models does not do a great job, but overall decent. Table 1 shows a nice comparison of the different models. We can see that all models range between 0.65 and 0.71 accuracy and precision. However, recall is definitely the best for all models with values ranging from 0.84 to 0.89.

As the methods used yield similar results it is to be assumed that the given data does not allow a better prediction of water-potability. The model constructed simply served as practice and should not be used for real life application.

5.4 Test data

After doing multiple tests with different models and different hyperparameteres we found that the best model was bagging with 140 decision trees of max depth 18. We saw in table 1 that this model gave us an accuracy of 0.709, precision of 0.717 and recall of 0.866 on the non-potable class on the validation data. However, we can fairly assume that this model will perform worse on the test data. As discussed in section 3.2 we have set aside 20% of the data as test data. This data has, until now, been unseen and the model is therefor unbiased against it. That is not the case with the validation data. This data has been used by several models and we have chosen the model based on how well it performer. In the real-world, where we do not know the correct label, we can not make different models and see which ones performs best. Therefor, we have tested the model against a unseen dataset. The results in table 2 gives us a unbiased estimate of how well we can expect the model to perform against new data. We see, as expected, that the model perform worse. The unbiased estimate of the accuracy is 0.655. That is a decrease of 8%. The F1 score decreased by 7%.

6 Conclusion

Despite an intensive grid search, the feed forward neural network obtains an accuracy not better than 69%. Using scikit-learn's MLP Classifier, the best results with an accuracy of 0.687 were obtained for $\eta = 0.1$, $\lambda = 2.410^{-4}$, with relu as the activation function one hidden layer of 10 neurons and adam as solver. Thus, a small architecture holds. As the learning rate is relatively high, convergence is reached after ~ 100 epochs.

Overall we can see that the models perform in more or less the same range. Decision tree, is as expected, a little worse then the rest. This model gave an accuracy of 65.2% on the validation data with max depth 6 using the Gini index. We saw that decision tree quickly got to complicated and overfitted the validation data. Random forest and boosting, which is meant to combat this problem in decision tree, gave better results. Random forest gave an accuracy of 68.4% on the validation data, with a max depth of 18, 160 trees and entropy, while boosting gained 69.9%, with 40 estimators and a max depth of 14.

Bagging with decision trees was the best model, giving us 70.9% accuracy on the validation data. Here we used 140 decision trees with a max depth of 14. However, our unbiased estimation of the accuracy, using the test data, gave us an accuracy of 65.5%. This is better then guessing, but not by much.

The recall was quite good for all models, so if one thinks that it is important to not miss-classify non-potable water as potable the model is good.

Since it seems to be difficult to associate the features with the given target features, the application of the feature development strategy could help to increase the correlation. This includes handling outliers, as can be seen in figure 4 that the dataset contains some outliers. Moreover, the dataset consists of 1265 samples that have at least one missing feature value. We simply dropped the values, but they could be replaced with estimates instead. This can be done by, for example, adding the mean value of each feature to the missing values. Another aspect is that we used 60% of the data for training. One could investigate how the training size affects the accuracy. Regarding the method used, one could extend the ensemble methods by for example applying XGBoost; or potentially try convolutional neural networks. One could also explore other methods like logistic regression and support vector machines.

Similar studies found here [6] also show a prediction accuracy of $\sim 70\%$. It can therefore be assumed that the given data does not allow a better prediction. It is clear that models with this accuracy should not be used in reality. Additionally, it was found that the available data showed unrealistic values with respect to the safety limits. Thus, it could well be that a new dataset would provide a better prediction. However, the goal here was to compare the methods themselves (on any given dataset).

References

[1] *2017 UN World Water Development Report, Wastewater: The Untapped Resource.* URL: http://www.unesco.org/new/en/natural-sciences/environment/water/wwap/wwdr/2017-wastewater-the-untapped-resource/. (accessed: 10.12.2021).

[2] Casper Hansen. *Optimizers Explained - Adam, Momentum and Stochastic Gradient Descent.* Oct. 2019. URL: https://mlfromscratch.com/optimizers-explained/#/. (accessed: 12.12.2021).

[3] Morten Hjorth-Jensen. *Applied Data Analysis and Machine Learning: Decision Trees.* https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter6.html. Nov. 2021. (accessed: 14.12.2021).

[4] Morten Hjorth-Jensen. *Week 40: From Stochastic Gradient Descent to Neural networks.* https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html. Nov. 2021. (accessed: 06.11.2021).

[5] Abul Hussam. "11 - Potable Water: Nature and Purification". In: *Monitoring Water Quality.* Ed. by Satinder Ahuja. Amsterdam: Elsevier, 2013, pp. 261–283. ISBN: 978-0-444-59395-5. DOI: https://doi.org/10.1016/B978-0-444-59395-5.00011-X.

[6] Aditya Kadiwal. *Water Quality: Drinking water potability.* https://www.kaggle.com/adityakadiwal/water-potability. Apr. 2021. (accessed: 24.11.2021).

[7] Ron Kneusel. *Practical Deep Learning : A Python-Based Introduction.* eng. New York: No Starch Press, 2021. ISBN: 1-7185-0075-0.

[8] Mesfin M. Mekonnen and Arjen Y. Hoekstra. "Four billion people facing severe water scarcity". In: *Science Advances* 2.2 (2016), e1500323. DOI: 10.1126/sciadv.1500323.

[9] Aditya Mishra. *Metrics to Evaluate your Machine Learning Algorithm.* Feb. 2018. URL: https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234. (accessed: 12.12.2021).

[10] Michael Nielsen. *Chapter 2: How the backpropagation algorithm works.* http://neuralnetworksanddeeplearning.com/chap2.html. Dec. 2019. (accessed: 02.12.2021).

[11] Bendik Nyheim, Marcus Moen, and Mira Mors. *Classification and Regression: From linear and logisitc Regression to neural networks.* Nov. 2021. URL: https://github.com/benyhh/FYS-STK4155/tree/main/Project2/Report. (accessed: 10.12.2021).

[12] *Secondary Drinking Water Standards: Secondary Standards Contaminants Table.* https://www.knowyourh2o.com/indoor/secondary-standards-contaminants-table. (accessed: 14.12.2021).

[13] Gaurav Singh. *From SGD to Adam.* May 2020. URL: https://medium.com/mdr-inc/from-sgd-to-adam-c9fce513c4bb. (accessed: 12.12.2021).

[14] *The World Counts.* URL: https://www.theworldcounts.com/challenges/planet-earth/freshwater/deaths-from-dirty-water/story. (accessed: 10.12.2021).

[15] *What is a Feed Forward Neural Network?* https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network. Nov. 2021. (accessed: 02.12.2021).