Project 3 FYS-STK4155

Sigurd Holmsen, Philip Sommerfelt, Øystein Høistad Bruce

November 2021

UiO : Universitetet i Oslo

Department of Mathematics

Contents

Abstract

When given a data set to study, there is a plethora of methods in machine learning to choose from. In this project, we will compare some of them to find which one works best for a classification data set. We will find the parameters which give the best results in terms of accuracy, and study which algorithm that has the highest accuracy in the end. We will also discuss the pros and cons of each method in terms of simplicity and training time. As an additional problem, we will study the bias-variance trade-off for different machine learning approaches for a regression problem, and explore what impact this analysis has when choosing a machine learning method. For the classification problem, we observed similar results in accuracy among the algorithms. Meanwhile, there were some vast differences in training time. For the regression problem, when increasing the complexity of each method, both bias and variance behaved differently.

1 Introduction

This report will cover both the main- and optional task, that is both the general study of different machine learning methods and the bias-variance problem. Firstly, we will look into the data sets we have used in the respective problems, which is both a classification- and a regression data set. The classification problem studies a dry beans data set 2.1 containing seven classes of beans, where the purpose is to classify correctly. The regression data set for the bias-variance problem contains housing data 2.2, where the goal is to estimate the average price of a block group. It is important to understand both the features of the data and what we want to predict before we can set up and interpret the machine learning models.

To be able to interpret the results, it is also important to understand the fundamental properties and mathematics of the methods we are performing. We will look into the methods that we have used, and will give a deeper explanation of the additional methods for the optional project. We will elaborate more on multinomial classification problems and regression, and on the details of the following machine learning methods: neural networks, logistic regression, decision trees and random forests. We will study some of the mathematical concepts behind them, and give a motivation for why the models work. We will also give a brief explanation of how we measure the performance of the models, as well as details on bias-variance trade-off.

The goal of this project is to evaluate the models we have chosen on specific data sets, so it is important to do thorough testing and analyze the results. When testing our models, we will firstly look at the classification data set, where we used a neural network, logistic regression, decision tree and random forest. All methods have pros and cons when it comes to simplicity, computation time and interpretability. In each machine learning technique, we will optimize the parameters for each model by making grid searches before we can evaluate the performance of the model on the data sets.

Then we will look at the regression data set. As in project 1 [15], we are again interested in doing the bias-variance trade-off analysis, but now with different machine learning methods. We will look at neural network, decision tree and random forest. Before we do the bias-variance analysis, we need once again to optimize the different parameters used in the methods. We will study how complexity can be interpreted in each model, and how it affects the error propagated from both bias and variance.

Lastly, we will summarize our results, compare the performances of the models and state their pros and cons, both for the main problem and the additional one. We will give a critical assessment and link with existing literature, before concluding with our main findings. This research is important because it shows that picking and training a machine learning method is no simple task, and every model has its pros and cons. It is also worth noting that in the field of machine learning it is often difficult to predict how well models work a priori, which bolsters the importance of thorough

testing with trials and errors before any conclusions can be made. The bias-variance trade-off is relevant because it explains not only how large the errors in our models are but where the errors come from, something that helps our understanding of machine learning.

In this project, we have used Tensorflow and "keras" [21] when creating and training the neural network, scikitlearn for decision trees and random forest-algorithms, and our own logistic regression implementation from project 2 [16]. The logistic regression code has some new features, since it will now handle multiple target values (more information in section 3.7).

The code is stored in the Github repository attached at the last page of the report (section 6.1). The main code in the repository is the **test_project_3** file, which contains all the tests we are running to achieve the results and figures in the result sections. In the appendix (section 6.2 and 6.3), we have added enough information to reproduce the figures and results we uses in the result section.

2 Data sets

2.1 Beans data set

We have chosen to use a classification data set provided by The University of California at Irvine containing data on different beans (UCI) [14]. We are trying to find a great machine learning model for this data set in section 4.2. The data set is interesting to look at, since the beans are almost inseparable for the human eye. Therefore it is interesting to see if a machine learning algorithm will be able to do.

The data consist of 13611 observations and 15 features. Some of the features included in the data set are shape, compactness and solidity of the bean. We will be using all the data in our analysis, and split the data according to section 3.1. We have transformed the classification data set to targets that are either $0, 1, ..., 6$ and represents the different beans shown below:

Figure 1: Picture of the beans in the data set, with the picture source.
From upper left: Seker [6], Barbunya [17], Bombay [23], Cali [6].
From lower left: Horoz [6], Sira [3], Dermason [1]

Now, we have a data set with 7 different numerated classes. The following table shows us how the targets are distributed, and the class number of the different beans (information gained from **Result 1** in the appendix section (6.2)):

Class (num)	distribution of targets	Class(name)
0	0.149	SEKER
1	0.097	BARBUNYA
2	0.038	BOMBAY
3	0.120	CALI
4	0.142	HOROZ
5	0.194	SIRA
6	0.261	DERMASON

2.2 Housing data set

We have chosen to use a housing (regression) data set provided by the Sci-kit learn package *sklearn.datasets* [18]. We will use this data set when studying bias-variance trade-off of different machine learning algorithms in section 4.3.

The data consist of 20640 observations and 8 features. We reduced the data to 2000 observations for the sake of computing power and time consumption. The target values lie between 0.15 and 5 and represent an average house value in units of $\$100,000$. The average is being taken by a block group which typically has a population of 600 to 3000 people) [18].

We want to predict average house value in a block group, therefore many of the input features will be a average of all the housings in the block. The input features, that shall predict the average block price are:

- median income and age in the block
- average number of rooms, bedroom and household members in the block group
- total population in the block
- latitude and longitude of block group

3 Method

Here, we explain the functions we have programmed and/or used and some concepts behind them. All methods assume some input and output data X, y that have been extracted or generated, and they are all supervised machine learning methods. The methods which will be discussed below are neural networks, logistic regression, decision tree and random forest.

3.1 Train-test splitting

For both data sets, we have done a train-test split. We do not want to evaluate our model with the same data we used to train it, as this will not reveal overfitted models. An overfitted model can fit the data it trained with very well, but it is often so high in variance that it loses accuracy when being used on new data, hence the name. Therefore, we only use 75% of each data set for training, while the remaining 25% is used for testing the quality of the model after it is trained. It's the result from the testing data we are most interested in when evaluating the performance of the algorithms. We will study both a classification data set and a regression data set for the main problem and additional problem, respectively. The details of the data sets are explained in the sections (2.1 and 2.2).

3.2 Bootstrap as resampling

We do not have a probability distribution of the outputs, and therefore we will use the independent bootstrap method. The bootstrap method goes as follow:

1. Pick n numbers for the observed variables with replacement, and save the result.
2. Now, use the saved result from last step to compute the desired estimate.
3. Repeat the process (1 and 2) k times.

The method provided is inspired by [7].

3.3 Principal component analysis (PCA)

The principal component analysis deals with the problem of high dimensional data set. The algorithm will reduce the dimension of the data, without loosing (much) information of importance. How this is done, can be found here: [10]. We will be using scikit-learn's functionality for computing the principal components in our data sets [19].

3.4 Scaling the data

In both of our data set, we scale by the maximum of each input feature, so that the largest input value will be 1 for each feature. This is to adjust for different units for each feature, which will help all our models train. Some units may be of an entirely different order than others, which would make it difficult for the models to adjust equally for all the features.

3.5 Stochastic Gradient Descent

The goal of gradient descent is to find parameters that minimizes a cost function. We have written a gradient descent function that takes in initial values of the parameters that will be tuned, a cost function, the learning rate η, a hyper-parameter λ and the number of iterations, as well as input and output data X, y. The learning rate η decides the length of each step in the search direction, while the hyper-parameter λ adds a term to the cost function to avoid overfitting in the following manner:

$$C(\beta, X, y, \lambda) = cost(\beta, X, y) + \lambda \|\beta\|_2^2 \tag{1}$$

Here, *cost* is the cost-function given as a parameter, β are the parameters for a given model that we want to tune. $\lambda \|\beta\|_2^2$ shows how large values of β are penalized in the final cost C. This will help prevent overfitting because large β values implicates a complex model which may fit the training data better than the testing data. The algorithm for updating the β parameter is explained below.

3.5.1 Momentum

We have used a momentum based gradient descent, which means the length of each step in the gradient descent depends on the size of the previous gradient. The point is to do longer steps when gradients are steeper, and smaller steps when the gradients are flatter. In this way, the gradient descent adjusts to the "terrain" of the cost function. The algorithm can be expressed as follows (inspired by [8]):

$$v_t = \gamma v_{t-1} + \eta_t \frac{dC}{d\beta} \tag{2}$$

$$\beta_{t+1} = \beta_t - v_t \tag{3}$$

where the parameter $\gamma \in [0, 1]$ is the amount of momentum we want to use ($\gamma = 0$ means no momentum). v_t is the amount of tuning of the parameter at time t. β is the parameter we want to tune, and $\frac{dC}{d\beta}$ is the change in cost with respect to β.

3.5.2 Stochastic

The gradient descent method is stochastic, meaning we do not necessarily compute the gradients for the entire data set X, y, but for a small, randomly chosen batch X_k, y_k. If we let the size of the mini-batches be M, we will have m mini-batches, where $m = \frac{\text{number of data}}{M}$ (rounded down to nearest integer). We then select a random integer $k \in [0, 1, ...(m-1)]$, and select the k-th mini-batch:

$$X_k = X[k \cdot M : (k+1) \cdot M]$$
$$y_k = y[k \cdot M : (k+1) \cdot M]$$

In this way, we have produced a randomly chosen mini-batch from X, y, and computed the gradient $\frac{\text{number of data}}{M}$ times. We also need to decide the number of epochs, that is how many times we will repeat this process. The goal of the stochastic gradient descent is to save computational power and time, as we compute lower dimensional gradients.

To summarize, Stochastic gradient descent takes the following parameters: model-parameter to be tuned β, a cost function, the learning rate η, the regularization parameter λ, the mini-batch size and the number of epochs, and its goal is to tune the parameters of a model until the cost function reaches a minimum. It will be utilized both in the Neural Network method and in Logistic regression.

3.6 Neural Network

A neural network is inspired by how a biological brain processes signals. In a feed forward neural network, the input signal is fed through hidden layers before it reaches the output layer. Each layers consists of nodes which take in signals of nodes from the previous layer, multiplies it with a weight matrix, adds a bias parameter, then sends the signal through an activation function. When the input signal reaches the output layer, the model will have made a prediction. We will further elaborate on the details and some of the mathematics behind a neural network.

3.6.1 Architecture

Regarding the network architecture, a neural network has the following parameters: number of input nodes, number of output nodes, number of hidden layers and number of nodes per hidden layer. The number of input nodes and output nodes are fitted to the dimension of the input and output data, respectively. The number of hidden layers and nodes per hidden layer can be interpreted as the complexity of a neural network, because if these parameters are increased, we will need to tune a higher amount weight and bias parameters, as will be explained below. We have decided to look at a constant number of nodes per hidden layer even though it would be possible for this to vary. This is mainly because there are a lot of parameters to tune in a neural network, and without this simplification we would have to run an unfeasible amount of tests to find the optimal network in our given time-frame.

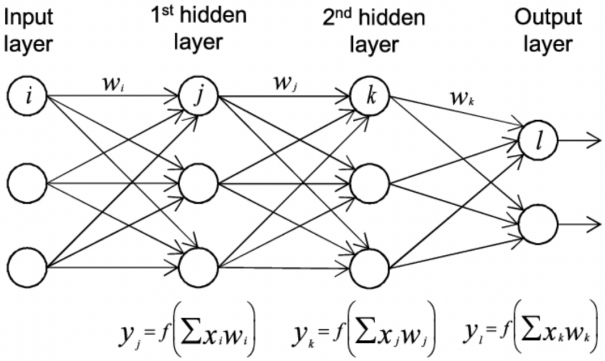

Figure 2: An example of architecture in a feed forward neural network [8]

In fig. 2, we can see an example of an architecture of a neural network with 3 input nodes, 2 hidden layers with 3 nodes each and 2 output nodes. This example provides just the weights parameters and not the biases. If we had added the biases to the architecture, it would look like $y_j = f(\sum x_i w_i + b_i)$, where x, w and b are the input from last layer, weights and biases (respectively). The function f is an activation function, which will be more explained in section 3.6.4.

3.6.2 Feed Forward

When the input-data is given to the input nodes, it is "fed forward" to the hidden layers. Each node in the hidden layers have its own weights and biases. When data is fed forward from one layer to the next, each node in the next layer receives signals from each of the previous nodes multiplied with each weight and then added with the bias parameter. This can be written mathematically as follows:

$$z_i^l = \sum_{j=1}^{M} w_{ij}^l x_j^{l-1} + b_i^l \tag{4}$$

The left hand side z_i^l is the information passed into node i in the layer l, and the right hand side is the sum of the signals from the previous layer x^{l-1} multiplied with the corresponding weight values w_i^l for each previous node, and finally added with the corresponding bias b_i^l. The value i in the equation indicates that this happens for every node in layer l. Next, each such signal is passed through an activation function:

$$a_i^l = f(z_i^l) \tag{5}$$

Here, the previously computed z_i^l is passed through the activation function f (more on this in 3.6.4), and the resulting value is the signal a_i^l which is passed on to the next hidden layer. An activation function has the purpose of deciding whether a signal should yield an output to the next layer or not. After the signals have been passed through all the layers, it reaches the output layer where the signals are passed through the final activation function. The output layer does not have to use the same activation function as the hidden layers, or it may not have an activation function at all. This all depends on what values we are expecting our model to return. If we want output values to be anything on the real line, we do not need an activation function here, but if we would like the output to be for instance a probability, we would use a function that takes values on $[0, 1]$. Examples of common activation functions will be given below.

The data should be a matrix of dimension $(\#data, \#features)$. This means we are approximating several output values simultaneously. The feed forward algorithm is further elaborated here: [8]. Before the training of the model can begin, the weights are initialized according to some distribution, for instance the standard normal distribution and the biases as small float numbers. That includes the learning rate η, the regularization parameter λ, batch size and number of epochs in the SGD, and the hyper-parameter γ.

3.6.3 Back Propagation

To tune our weight and bias parameters, we use a momentum stochastic gradient descent (SGD) as explained above. To find gradients for both the weights and biases, we perform the back propagation algorithm. First, we choose a cost-function as a measure of model quality, then we are interested in finding the gradients for the cost function C with respect to the weights and biases for a given layer l:

$$\frac{\delta C}{\delta w_{ij}^l}, \frac{\delta C}{\delta b_j^l} \tag{6}$$

According to the back propagation algorithm, this can be written as:

$$\frac{\delta C}{\delta w_{ij}^l} = \delta_j^l a_k^{l-1}, \frac{\delta C}{\delta b_j^l} = \delta_j^l \tag{7}$$

where

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{8}$$

Here, f is the activation function for the given layer, δ_j^l is the error term for layer l and node j, a_k^l is the value for layer l and node k after it has been sent through the activation function for the hidden layers, and z_j^l is the value of node j in layer l before being sent through the activation function. We can compute this if we know the value of δ^{l+1}. We compute the last layer delta (or error) in such way:

$$\delta^L = \frac{1}{2n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{9}$$

So, we are able to find all the δ values starting backwards, which we use to compute the gradients for all weights w^l and biases b^l. After we have found all these gradients, we use this for the momentum based SGD in the same way as described in the section above. When the SGD has iterated through all the given epochs, the training of the model is complete. We have been inspired by [8].

For this project, we look at both a regression problem and a classification problem, and these two problems require slightly different neural networks which will be explained below. We have used "tensorflow" with "keras" to create instances of our neural networks. The documentation is referenced here: [21]

3.6.4 Activating functions hidden layers

We have used both of the following activation functions for the hidden layers:

Sigmoid

The Sigmoid is a bounded and differentiable S-shaped function defined for all real values $x \in \mathbb{R}$. In our case, for use as activation function in a neural network, we define the Sigmoid as the logistic function:

$$f_{sig}(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}.$$

The derivative of the logistic function is:

$$\frac{\partial}{\partial x} f_{sig} = \frac{e^x}{(1+e^x)^2},$$

it is bell shaped, always positive and tends to zero as $x \to \pm\infty$. When training a neural network with the logistic function in the hidden layers this can lead to the infamous problem of a vanishing gradient, causing the network to learn slower for smaller or larger inputs. That is, weights will change little or not at all in each iteration. Different proposals have been made to overcome this problem. One solution is to consider another activation function, e.g the ReLU presented in the following. [24]

ReLU

The Rectified Linear Unit, or ReLU for short, is a piecewise linear function; linear for values $x \in R^+$ and zero for values $x \in \mathbb{R}^-$. The linearity often makes optimization task easier, and the fact that is zero for negative values accounts for non-linearities. The ReLu is defined as:

$$f_{ReLu}(x) = \max(0, x)$$

This simple definition makes it very easy to implement, and although it's derivative is not defined for $x = 0$, for $x \neq 0$ it is well defined as:

$$\frac{\partial}{\partial x} f_{ReLu}(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$$

In practice, one chooses a value (either 0 or 1) for the derivative when $x = 0$. As mentioned, the ReLu activation function does not suffer from the problem of a vanishing gradient (negative weights are put to zero and input is disregarded). This causes models to learn faster and often perform better than models using other activation functions such as the sigmoid. There are however some limitations to the ReLu. In particular, large weight updates can result in the input to the activation to always be negative. This causes the node to always give a value of zero activation, resulting in a phenomena that is often referred to as the dying ReLu. Some proposals has been made to overcome this issue, and an example is the leaky ReLu. [24]

3.6.5 Regression case

For the regression case, we will not need an activation function for the output layer. This is because we may want to fit any number from $-\infty$ to ∞, and then we do not want to put a boundary on the value of the output. We also use the MSE as the cost function:

$$MSE = \frac{1}{2n} \sum_i (y_i - \tilde{y}_i)^2 \tag{10}$$

Here, y_i is the actual target data and $\hat{y}_i$ is the predicted value given the same input data for $i = 1, ..., n$ where n is the number of data points.

3.6.6 Multinomial Classification case

For the first data set, we will utilize multinomal classification, which means the targets in the data are one of n classes. To translate the different classes into usable data, each class instance is represented by it's own one-hot vector [9]. The only non-zero element is placed correspondingly with the class:

$$\begin{aligned} Class_0 &= (1, 0, ..., 0) \\ Class_1 &= (0, 1, ..., 0) \\ &... \\ Class_{n-1} &= (0, 0, ..., 1) \end{aligned}$$

This is done using the "to categorical" function from "keras" [22]. The neural network will need n output nodes to predict each one-hot vector, where the values of each output node can be interpreted as the probability of the input being of the corresponding class. To further separate the multi-class case from the binomial case we did in project 2, we need to introduce a new cost function and an activation for the output layer.

3.6.7 (Cross-entropy) cost function

As a cost function for the neural network, we will be using the cross-entropy cost function:

$$C(\hat{\boldsymbol{y}}; \boldsymbol{y}) = -\sum_{i=0}^{n-1} log(\hat{y}_i) y_i \tag{11}$$

Where $\hat{\boldsymbol{y}}$, $\boldsymbol{y}$ is the predicted target and the actual target (respectively) of the neural network. This function can be interpreted as a maximum log likelihood with a negative sign, meaning the higher the likelihood, the lower the cost.

3.6.8 Softmax activation function

Since we want to restrict our output layer so that it represents probabilities of each class, we need an activation function for the output layer that sums up to 1. We use the Softmax function:

$$f(z_j) = \frac{e^{z_j}}{\sum_{i=0}^{n-1} e^{z_i}}, \qquad j = 0, 1, ..., n-1 \tag{12}$$

After this is applied to every node of the output layer, we have our model prediction. Our final prediction will be the class, or equivalently the output node, with the highest probability.

3.7 Logistic regression

Logistic regression predicts probabilities of given outputs being of a certain class, and uses stochastic gradient descent to tune its model parameters. The model is given as follows:

$$p(y_j = 1; \boldsymbol{x}, \boldsymbol{\beta}) = \frac{e^{(x_j)^T \boldsymbol{\beta_c}}}{\sum_{i=0}^{n-1} e^{(x_i)^T \boldsymbol{\beta_i}}} \tag{13}$$

Where $p(y_j = 1)$ is the probability that a given input data will be part of class $class_j$, and each class $j = 0, ..., n-1$ will have its own $\boldsymbol{\beta}$ parameter $\boldsymbol{\beta_j}$. To find the best $\boldsymbol{\beta}$ parameter values for a given data set, we use stochastic gradient descent given a cost function.

3.7.1 Cost function

The cost function used in the gradient descent for the logistic regression algorithm is the categorical cross-entropy:

$$C(\boldsymbol{\beta}, \boldsymbol{x}, \boldsymbol{y}) = -\sum_{i=0}^{n-1} log\left(p(y_i = 1; \boldsymbol{x}, \boldsymbol{\beta})\right)\ y_i \tag{14}$$

Again, the optimal $\boldsymbol{\beta}$ parameters will minimize the negative log likelihood function. The final predicted output given input data will be the class which the highest value, i.e. the highest probability. For instance if $(y_j = 1; \boldsymbol{x}, \boldsymbol{\beta}) = 0.90$, the predicted output will be $class_j$.

3.8 Decision trees

The explanation of the decision tree algorithm is inspired of a lecture given by Riccardo De Bin, Fall 2021 [5].

Decision trees are maybe the most used machine learning algorithm outside the field of machine learning. This is due to its quite simple implementation, low training times and high level of interpretability. The method is, not very surprisingly, about building a tree like structure of decision rules. It works fine for both regression and classification problems and can be applied to problems with both numerical and categorical features. One downside to the decision tree method is that it generally is very unstable (high variance).

The tree itself is a recursive binary partition of the feature space. In each iteration, a region is split into two or more regions. This splitting process is repeated until a stopping criterion is satisfied, and the end result is M regions $R_m, m = 1, ..., M$. In regression problems a constant value c_m is assigned to each region, in classification problems we need to assign each region to a class. In regression problems the final prediction is

$$\hat{f}(X) = \sum_{m=0}^{M-1} \hat{c}_m \mathbb{1}(X \in R_m). \tag{15}$$

where X is the feature matrix, $\hat{c}_m$ is the predicted value of region R_m (e.g. $ave(y_i|x_i \in R_m)$) and $\mathbb{1}(\cdot)$ is the indicator function.

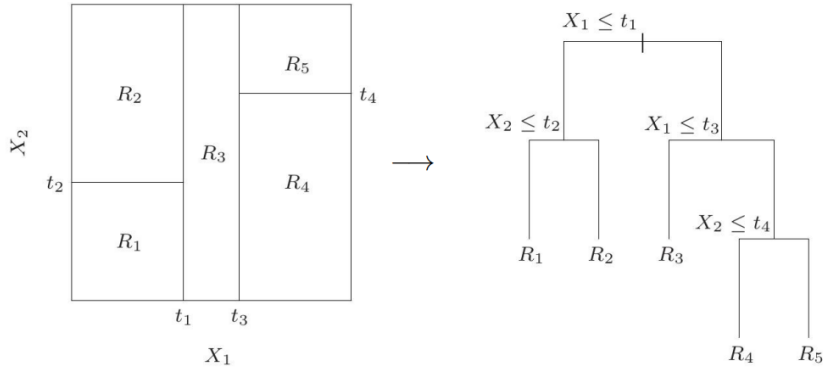

Figure 3: Source: [13]

The splitting points t can be interpreted as a junction of the tree where all observations are assigned to a *branch* and at the end of each branch an end node, or *leaf*. The variable to split and where to split it needs to be automatically chosen by our algorithm. That is, the algorithm chooses

the variables and splits points that minimizes a cost function at each iteration.

The problem of finding the optimal number of splits is however not trivial. Finding the best tree only in terms of minimizing total cost is generally not feasible, and if we split the tree until there are no more splits we have only one observation in every region and the tree will heavily overfit the training data. Therefore, we will have to choose a criterion of when to stop the algorithm. One could stop when the pre-specified tree size is reached, or whenever the next split does not reduce the total cost more than some threshold. However, the best approach is in general to build a complete tree with only one observation in each region and then *prune* it to find the optimal tree size for the problem at hand. We will look at this soon. First, we focus on growing the tree.

Growing a regression tree

- Firstly, we need to choose a cost function $C(\hat{y}; y)$ that takes the correct target value y and the predicted value y_hat.
- For each input variable X_j, find the best splitting point $\hat{s}$. Define the two hyperplanes:
  - $R_1(j, s) = \{X | X_j \leq s\}$
  - $R_2(j, s) = \{X | X_j > s\}$
- For each j and s, solve

$$\min_{j,s} \Big\{ \min_{c_1; x \in R_1(j,s)} C(c_1; y) + \min_{c_2; x \in R_2(j,s)} C(c_2; y) \Big\} \tag{16}$$

When the cost function is the usual squared error $C(\hat{y}; y) = \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2$, the solution to the inner minimization problem is feasible and the solution is

- $\hat{c}_1 = ave(y_i | x_i \in R_1(j, s))$
- $\hat{c}_2 = ave(y_i | x_i \in R_2(j, s))$

The index j corresponding to the variable X_j to be split and split point $hats$ is then the pair (j, s) that minimizes the sum $\hat{c}_1 + \hat{c}_2$

Classification trees

There are no major differences between regression and classification problems. However, we will need a different cost function and instead of predicting a value we need to predict the class to assign each region.

- Define the class $k \in \{1, \ldots, K\}$ for each region. i.e. choose the class that contains the most observations in each region:
  - $k_m = argmax_k \, \hat{p}_{mk} = argmax_k \Big\{ N_m^{-1} \sum_{x_i \in R_m} \mathbb{1}(y_i = k) \Big\}$
- Spilt the tree where the predicted value, $\hat{p}_{mk}$, minimize the cost.

Some standard cost functions for classification problems include

- 0-1 loss: $N_m^{-1} \sum_{x_i \in R_m} (y_i \neq k)$
- Gini index: $\sum_{k=0}^{K-1} \hat{p}_{mk}(1 - \hat{p}_m k)$
- Entropy: $\sum_{k=0}^{K-1} \hat{p}_{mk} log \hat{p}_{mk}$

In the classification problem later in this project we chose entropy to be our cost function.

Pruning

There are some different algorithms for tree pruning, but we will here focus on the method of cost complexity pruning:

Let the complete tree be T_0. Then, the goal is to find the optimal tree $T_\alpha \subset T_0$. The *cost complexity criterion* is:

$$C_\alpha(T) = \sum_{m=1}^{M_T} N_m Q_m(T) + \alpha M_T \tag{17}$$

where N_m is the number of observations in each region, $Q_m(T)$ is loss in each region, M_T is number of leaves (tree size) in tree T and $\alpha \geq 0$ is a tuning parameter that represents the trade-off between tree size and level of fit. Smaller values of α gives larger trees and larger values smaller trees. Typically, one finds the optimal value $\hat{\alpha}$ by iteratively collapsing the complete tree at the split point that corresponds to smallest increase in loss until no more splits and in the end choose the tree T_a with smallest total loss withing the sequence of iterations. Or, one could simply use cross validation to find the optimal value for α and with it the optimal tree $T_{\hat{\alpha}}$ with optimal tree size M_{opt}.

3.9 Random forest

The method of random decision forests was first introduced by Ho (1995) [11] and later improved by Breiman (2001) [2]. The general idea proposed by Ho was to train many decision trees, each time on randomly selected features, and aggregate over the results. This was done to prevent overfitting of training data. In 1996, Breiman introduced the concept of bagging and applied this to decision trees later. His method of random forest with bagging is widely used and will be presented here.

Bagging, inspired by [12]

Bootstrap aggregating, or bagging for short, is an ensemble learning method. An ensemble method in machine learning is a method that uses a finite set of models to increase the predictive performance. In bagging, the addition of more than one model is used to reduce the variance in a dataset by using weak (high bias / low variance) learning models. The method follows three steps:

- Sample multiple samples with replacement from the observations (bootstrap).
- Train a model on each sample.
- Aggregate over results.

In regression problems the aggregation is typically done by taking the average over predictions. In classification the standard approach is to choose the class with the majority of predictions. This is known as soft and hard *voting* respectively.

Decision trees with bagging

When constructing a random forest, the models that are trained and aggregated over are decision trees. This model is much harder to interpret than the single decision tree as the final prediction does not follow some very specific decision rules but is the result of many decision rules combined.

The trees in a random forest all have the same depth but are not trained on the same samples and will therefore be a little random. Generally, one would not benefit from having large trees as this will lead to overfitting and also more correlation between the trees. Instead, one would want the trees to have less correlation (higher variance between them), be more biased and therefore span more of the data. This is known as the concept of weak learners often used in ensemble methods (this is maybe more important in other methods such as boosting). In addition to the tree depth there is another parameter to consider; the number of trees. When adding more trees to the model

one will get a better prediction in the end. When aggregating over many weak learners, one will end up with a stronger learner. This is famously known as Wisdom of Crowds; the collective opinion of a large group is generally better than the opinion of a single expert [25].

3.10 Bias-variance trade-off

This part will be specifically for the additional problem. When we study how the complexity affects the accuracy of a model, it is useful to look at the bias-variance trade-off. Firstly, we will explain what it means when we say that a model is complex. Even though it is a bit subjective to each model, it can generally be interpreted as how many parameters in a model that need tuning, or in other words how many degrees of freedom are in play. Complex models generally require large data sets to train properly, while simple models can train on smaller sets. The optimal complexity in a model requires heavily on the nature of the data set.

We often measure the mean square error of a model to determine how well it performs. The mean square error can be re-written as follows [15]:

$$\mathbb{E}[(\boldsymbol{y} - \tilde{\boldsymbol{y}})^2] = \frac{1}{n}\sum_i (y_i - \mathbb{E}[\tilde{\boldsymbol{y}}])^2 + \frac{1}{n}\sum_i (\tilde{y}_i - \mathbb{E}[\tilde{\boldsymbol{y}}])^2 + \sigma^2 \tag{18}$$

where $\boldsymbol{y}$ and $\tilde{\boldsymbol{y}}$ represents the actual target values and predicted values, according to our model. σ^2 is the irreducible error of the model.

Here, the first term is the expected square difference between the observed data and the expected model $E(\tilde{\boldsymbol{y}})$. This can be interpreted as the bias in the model, as this is the error of the expected, "perfect" model $E(\tilde{\boldsymbol{y}})$. The second term is the expected square difference between the actual model $\tilde{\boldsymbol{y}}$ and the expected model $E(\tilde{\boldsymbol{y}})$, which is the variance of the model. If the bias is high, it means the model is not complex enough to predict the target data, and this would likely be an underfit. If the variance of the model is high, we likely have an overfit, as small changes in data should not lead to large changes in the model itself. Though it appears that you can always get rid of variance in the model by adding more data points, this will not necessarily be possible in real life scenarios, and this insures the importance of the bias-variance trade-off.

To be able to get the estimates above, we need to use a resampling method. We have chosen to go for the bootstrap method for the resampling process.

3.11 Measure the quality of a model

3.11.1 R2-score

The coefficient of determination, or R2 score, is a goodness-of-fit measure often used in regression problems. It is interpreted as the proportion of variance in the dependent variable that is explained by the model. The R2 score is generally a number between 0 and 1, with higher values indicating a better fit. However, it can also be negative if the model fits the data really poorly, e.g. worse than a straight line in two dimensional problem. The formula for the R2 score is

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2} \tag{19}$$

That is, SS_{res} is the residual error as a sum of squares. SS_{tot} is the total error, also as a sum of squares. $\hat{y}_i$ and $\bar{y}$ is the model prediction for the target value y_i and the average of the y_i's respectively. The R2-score will be most relevant when we look at the regression problem.

3.11.2 Accuracy score

We will look at the accuracy score to measure how well the adapted model is, when the target values are discrete (classification problems). The accuracy is the proportion of correct predictions

made by the model.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} \mathbb{I}(\hat{y}_i)}{n} \tag{20}$$

where $\mathbb{I}$ is the indicator function:

$$\mathbb{I}(\hat{y}_i) = \begin{cases} 1 & \hat{y}_i = y_i \\ 0 & \hat{y}_i \neq y_i \end{cases}$$

$\hat{y}_i$ is the predicted target value of the model, y_i is the actual value and n is the number of data points.

3.11.3 Confusion matrix

The confusion matrix shows the ways in which your classification model is confused when it makes predictions [4]. The confusion matrix presents the number of correct and incorrect predictions, within a matrix where the entries are different classes. This measuring tool is a great way to get to know your model prediction, and will provide more information than the accuracy score which will only tell you a true-false statement of the prediction.

Figure 4 is an example of a confusion matrix, where the actual values are along the y-axis and the predictions along the x-axis. The diagonal refer to correct predictions, the other entries shows the predictions that were wrongly classified. The entries above the diagonal are falsely predicted not to belong to the class when they were in fact a member of the class (false negative), the entries below the diagonal are predicted to belong to a class in which they did not belong (false positive).

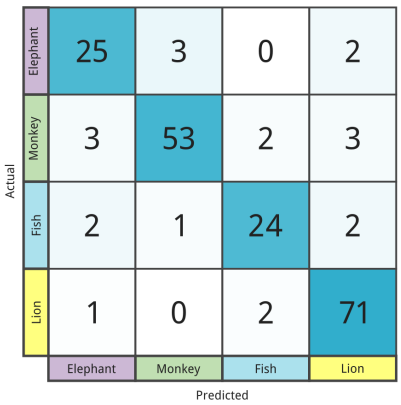

Figure 4: confusion matrix example

Analysis of the confusion matrix could be by looking at the predictions of the wrong predicted values, and see if the target values have highly correlated properties, shape etc. For instance, if the target values are of a ranking from $1 - 10$, then a smaller "predicting error" distance is better than a greater. In such cases, there will probably be some good models even though the accuracy score is low.

4 Results and discussion

4.1 Introduction to the results

In this section, we will refer to the tests we have made in the **test_project_3.py** file which will be found in the **project3** folder inside the github repository, attached in the appendix section (section 6.1). To have a pleasant experience with the testing file, we recommend you to read the **README.md** file inside the **project3** folder.

We will provide the necessary parameters for reproducing the figures/ results, and what test to run, in the appendix (section 6.2 and section 6.3).

4.1.1 General comments on the results

Here is a list of things to be aware of related to the results we are presenting:

1. When we are comparing two parameters, we will look at the R2-score and accuracy score for the regression- and classification problem (respectively). The tests inside the **test_project_3.py** file will often contain the result for both training and testing data, but we have provided the testing data results in this report (since this is the most important one).

2. In our analysis, we had to initialize some parameters to be able to start tuning the first parameters. Whenever we achieved some results from a previous testing, we added the best parameters to the new testing.

3. The number of epochs we have chosen in the analysis isn't optimized for the best model. More iterations will, very often, achieve a better model. As a result we have chosen to go with some *friendly* number of epochs to save computing expenses.

4. We need to make some initial guess of the parameters we want to tune with the SGD-algorithm. The initial guess for the logistic regression case is $\beta = [0.1, ..., 0.1]$.

5. While optimizing the batch size for the neural network, we have decided to set the number of epochs to a scalar times the batch size. This adjustment is to make the number of iterations in the SGD algorithm equal for each batch size.

6. We will often provide only the result from the testing data when we are optimizing the parameters. The training data figures will be showed when running the different tests in the code, but will not be included in the report.

4.2 Exploring machine learning algorithms (classification problem)

In this section, we will study four different machine learning algorithms on the beans data set (2.1)). We will try to find the best model among neural networks, logistic regression, decision trees and random forests.

In the appendix (6.2) under **Result 1** we achieve information of the total explained variance ratio of the principal component analysis (section 3.3) on the beans data set, which is close to 100% for only one feature. We will use 3 features utilized by the PCA. We have explored the usage of 1 feature, since it returned such a high total explained variance ratio, but that would require more iterations in the fitting process. We are scaling the data according to section 3.4.

4.2.1 Neural network

Since the data set has seven classes, we need a multinomial classification neural network as described in the method section 3.6. We will start the analysis by studying the RELU function as an activation function. We will in the following test find parameters that give the best results

(accuracy score) for the neural network.

First, we need to find the optimal architecture of the neural network

We will produce a heatmap with the number of layers and nodes at the x- and y-axis respectively. To be able to create such a plot, we need to initialize some parameters (as shown in 6.2). The plot (fig. 5) tells us how we should build up our neural network by picking some architecture, a combination of the amount of nodes and layers, that gives us the highest accuracy score.

By looking at figure 5, we can see a region with an accuracy of about 0.9 and also some cells with a poor accuracy. The architecture that received such a poor accuracy was probably too simple to be accurate as a prediction model. We tried to find a region where the architecture gave stable results, and amoung the highest accuracy score. We picked 14 hidden nodes and 4 hidden layers for the further analysis.

A too high complexity of the model will be more computationally expensive, and may also result in an overfit, while a too simple model seems to give an underfit. So, we are picking the architecture with that in mind. The tendencies of underfit can be understand in fig. 5 by looking at the results with 2 hidden nodes, which are poor accuracy. The overfitting can be seen in the right lower corner of the plot, where the score will be worse by adding more hidden layers/ nodes to the network. The last interpretation can may be wrong, as the model need more training when having a more complex network (this gridsearch was with the same amount of training iterations).

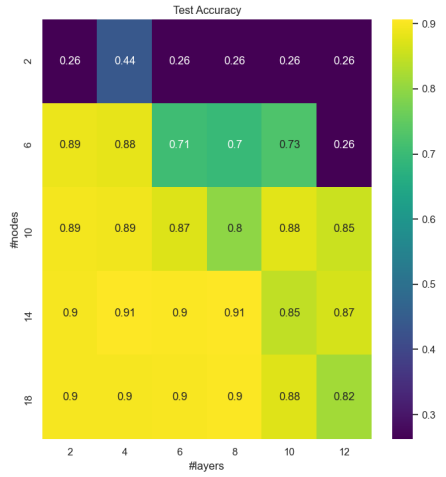

Figure 5: Data: testing, accuracy of different architecture of the neural network

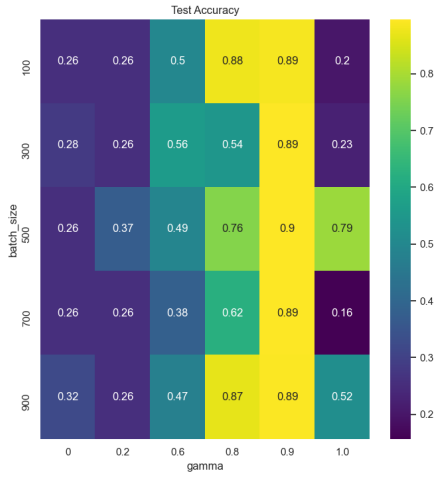

Figure 6: Data: testing, accuracy score of different batch size and gamma values

Batch size and momentum parameter

Now we want to find some optimal parameters for the stochastic gradient descent algorithm inside the neural network, namely the optimal batch size and momentum (gamma) parameter. Since we have optimized the architecture of the neural network (by last interpretations), we will use those in the new grid search. The plot (fig. 6) shows us the accuracy score of different momentum- and batch size values. The model seems most accurate at a gamma value of 0.9. Such a large momentum parameter can indicate that the number of epochs (or iteration of training the algorithm) are too low, such that a greater momentum will converge faster to a better solution. The accuracy looks kind of independent of the batch size, so we will choose 100 to save compute expenses.

The hyperparameters learning rate and lambda
Now we want to find the optimal hyperparameter λ and learning rate η by making a grid search, and find the parameters such that the accuracy score is as large as possible. With the parameters we found for the batch size, momentum (for the SGD) and the architecture of the neural network, we will use these optimized values to find the optimal λ and η values.
The figure 7 tells us that the best learning rate, η, is around 10^{-2} or 10^{-3} and the most stable lambdas are between 10^{-3} and 10^{-7}.

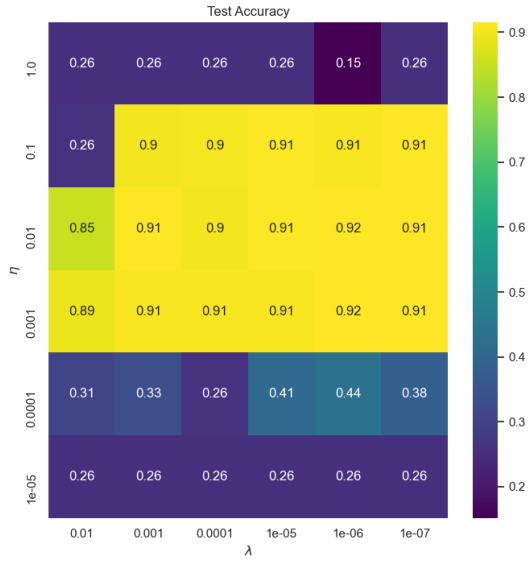

Figure 7: data: testing, accuracy score of different lambda and eta values

We can see from fig. 7 that the results are almost independent of the λ value, so we will go for a $\lambda = 10^{-5}$. If the learning rate exceeds 10^{-2}, we observe that the model is not as stable. With a too low learning rate, we observe that in some cases the training have gone too slow to reach well tuned parameters. This could be fixed by setting up the number of epochs, but this would require more computing power which is something we want to avoid if possible.

We want to find the optimal activation function for the hidden layers. So far we have tested with the RELU function. In plot 8 we have switched to the sigmoid activation function where all other parameters stay the same. We observe that it does not achieve a better result than the RELU function. Hence we will choose RELU as the optimized activation function for the hidden layers.

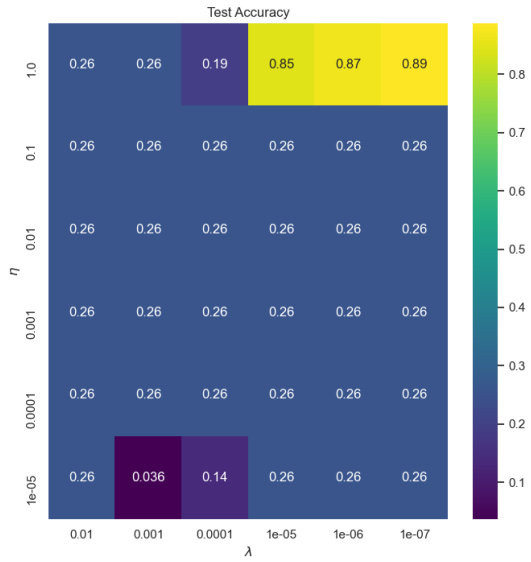

Figure 8: data: testing, accuracy score of different lambda and eta values (activation function: sigmoid)

Summary

We conclude from the optimization routine that the optimal parameters are:

parameter	value
learning rate, η	10^{-2}
regularization, λ	10^{-5}
momentum γ	0.9
batch size	100
hidden nodes	14
hidden layers	4

We want to look at the accuracy by running the optimal parameters. How this is done, will be shown in the appendix section (6.2, underneath **Result 2**). The accuracy of the training- and testing data are 0.911 and 0.917 (respectively).

Our network is able to predict 91% of the testing data, that is predicting the correct bean of data that were unknown for the model when it was trained. The confusion matrix for the prediction, with the neural network algorithm, on the testing data is shown in the figure below (figure 9).

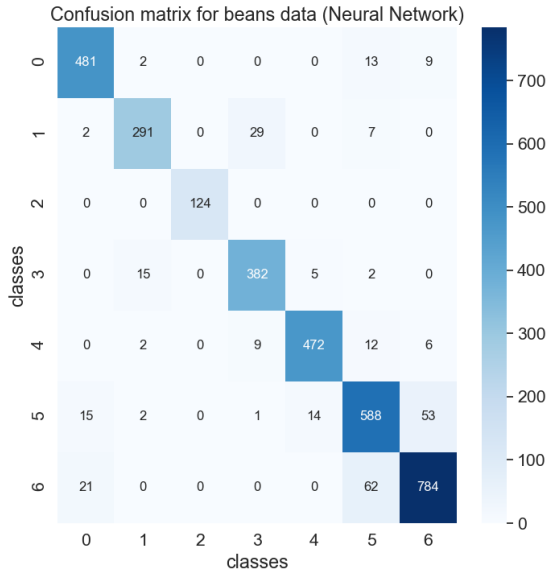

Figure 9: data: testing, confusion matrix

The confusion matrix shows what the model predicts on the testing data. The confusion matrix gives more information than just the accuracy score since the accuracy score provides just true or false on the prediction, while the confusion matrix tells us what the wrong predictions were. By fig. 9, it looks like it predicts (more often) wrong between the classes 5 and 6, which are the beans *Sira* and *Dermason*. The reason of a bad prediction on these nuts, can be if they have many similarities, like shape. In addition, the neural network mixes some of the observations from classes 1 and 3. Class 2 is distinguishable from the rest, where no prediction including this class are wrong. So, when the model predicts the class 2 then it has correct every time (on the testing data set).

4.2.2 Logistic regression

The second method we have chosen to study for optimizing a model to predict beans is the logistic regression. We will, as in the neural network algorithm, optimize the parameters used in the SGD-algorithm.

Batch size and momentum parameter

Now, we need to initialize some parameters to start optimizing - the initialized parameters are shown in the appendix section.

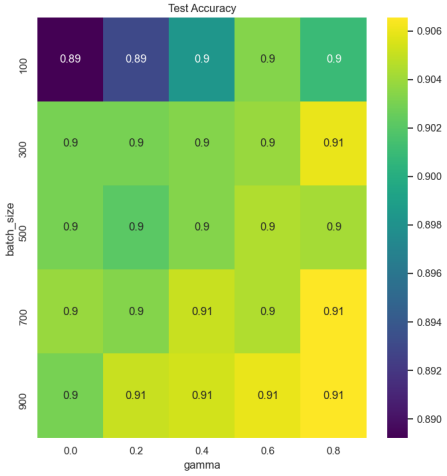

Figure 10: data: testing, accuracy score of different batch size and gamma values

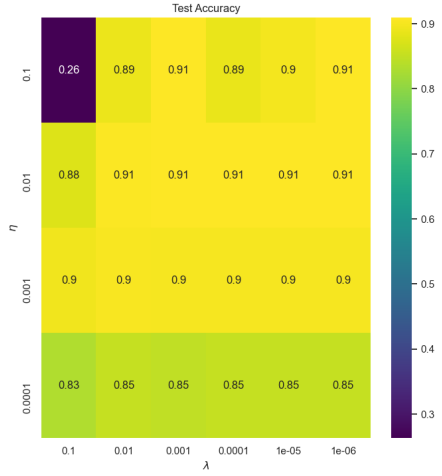

Figure 11: data: testing, accuracy score of different lambda and eta values

The figure 10, shows us that the accuracy of the model is kind of indifferent regarding to both the gamma and batch size parameter. We are going to go further with gamma = 0.8 and batch size of 500.

The learning rate and lambda

We continue the optimization by looking over the learning rate and lambda.
In fig. 11, we see that the accuracy does not depend that much on the regularization parameter lambda, but the learning rate get the best score with a value of 0.01.

Summary

Now, we want to see the accuracy and the confusion matrix with the optimized logistic regression model. The accuracy of the training- and testing data are 0.907 and 0.906 (respectively). The results can be reproduced (see the appendix section 6.2, underneath **Result 3**).

The logistic regression achieves an accuracy score of around 91%. The confusion matrix below, figure 12, is generated from prediction of the testing data with the logistic regression model.

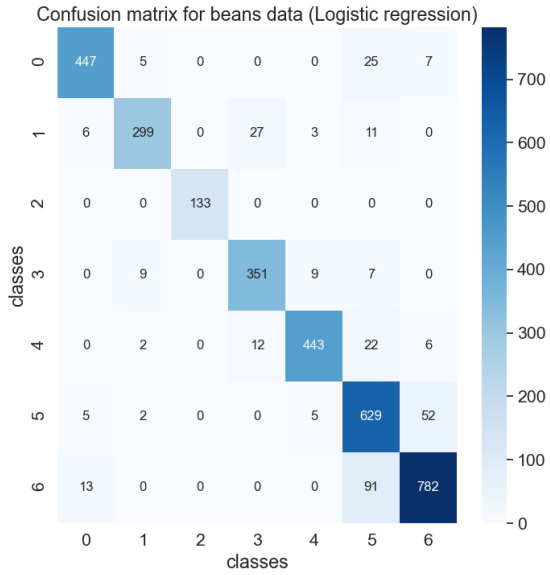

Figure 12: data: testing, confusion matrix

The confusion matrix looks similar in this case, as for the neural network method. However, the logistic regression does even worse when it comes to the very similar beans in class 5 and 6. Although it does equally good when it comes to the true positive rate of class 5, it predicts many more 6's to be class 5. In general, we see a heavy overprediction to class 5. Regarding the mix up between class 1 and 3, the logistic regression does a little better than the neural network. We can also see the same results, as in neural network model, with respect to the second class.

4.2.3 Decision tree

There is just one parameter to optimize here, the depth of the decision tree. We increase the depth in the model until we reach an accuracy of 1 for the training data. The plot underneath, figure 13, will show the accuracy score of the training- and testing data on all trained models.

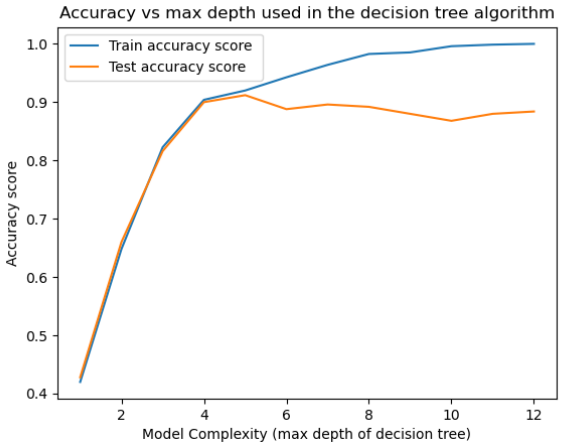

Figure 13: data: testing, accuracy vs. depth (decision tree)

In fig. 13 we see that the depth giving the highest accuracy is around 5. The accuracy for training data will converge towards 100% as the complexity increases (as told in section 3.8). We observe that a larger depth will cause an overfit.

Summary

Now, we want to see the accuracy and the confusion matrix. From the appendix (**Result 4**), the accuracy of training- and testing data are 0.904 and 0.902 (respectively).
The decision tree algorithm seems to also work well for this data set, we get an accuracy score of around 90%. We also get the following figure (14) the confusion matrix for testing data with the decision tree algorithm.

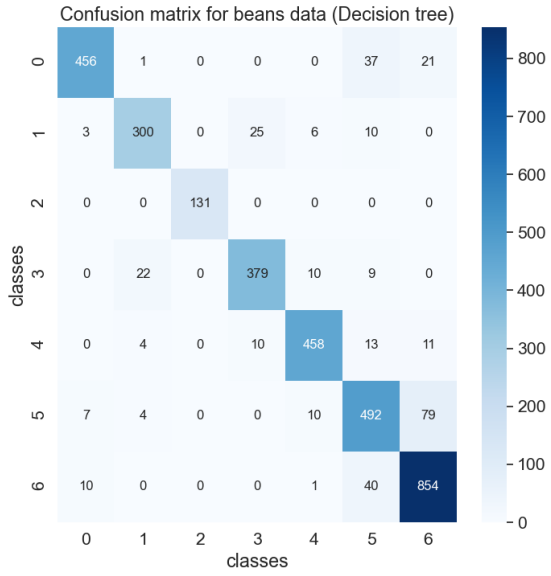

Figure 14: data: testing, confusion matrix

The confusion matrix for decision trees does, as expected with almost same accuracy, look similar to the two previous methods. However, while logistic regression did worse than the neural network when predicting classes 5 and 6, the decision tree makes as many mistakes as the neural network. The decision tree does in contrast to logistic regression predict fewer 5's to 6 and more 6's to 5. We also see a slightly worse prediction with respect to class 1 and 3 than the previous methods.

4.2.4 Random forest

Lastly, we will apply a random forest algorithm to our data. we will only optimize one parameter, namely the depth of each decision tree. We increase the tree depth until we reach an accuracy of 1 for the training data, and then we compare with the accuracy for testing data to identify the optimal depth. We have not tested for different numbers of trees, as we have set this to be equal to 100, which we assume is sufficiently high enough for good results.

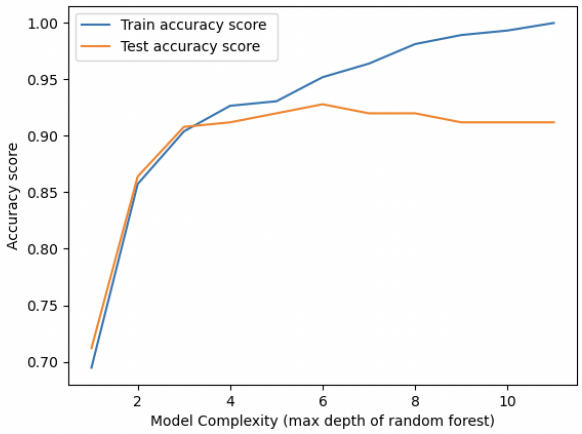

Figure 15: data: testing, accuracy vs. depth (random forest)

In fig. 15, we see that the depth giving the highest accuracy is around 6. The random forest also performs a little better than the decision tree in terms of accuracy. This is very much as expected as the random forest aggregates over many trees, see section 3.9.

Summary

Now, we want to see the accuracy and the confusion matrix for the random forest algorithm. In the appendix, **Result 5**, will give us accuracy scores of 0.914 and 0.913 to the training- and testing data set (respectively).
We get an accuracy score of over 91%, slightly higher than the other methods. And we also get the following figure (16), the confusion matrix for testing data with the random forest algorithm.

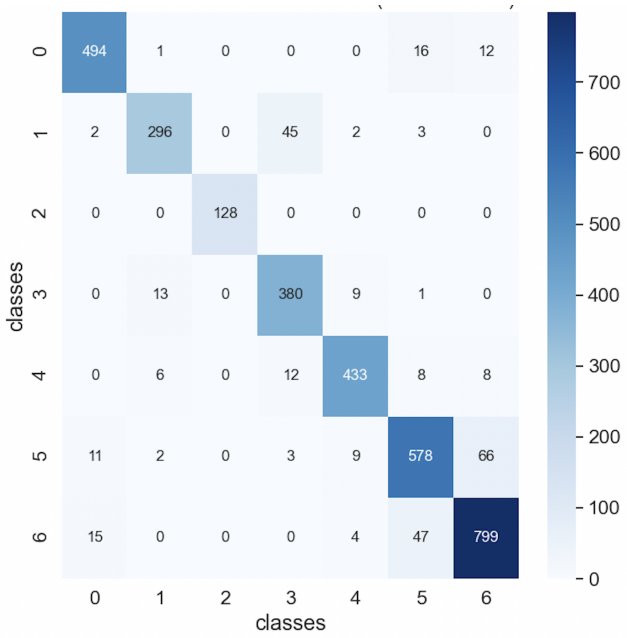

Figure 16: data: testing, confusion matrix for beans data (random forest)

The confusion matrix again looks similar to the other algorithms we have tested. Most of the errors comes from misclassifying class 6 to 5, class 5 to 6 or class 1 to 3. Also in this case, the bean in class 2 does not contribute at all to the error.

4.2.5 Comparison and discussion

In this part of the project we have used four different machine learning algorithms to classify bean types in the dry beans dataset. The methods all performed very similarly in terms of accuracy. The worst performer was the decision tree classifier with reported test accuracy of 90.6% and the best was the neural network with test accuracy of 91.7%. By comparing the confusion matrices we learned that the most indistinguishable beans seems to be Dermason (class 5) and Sira (class 6) as trying to classify these resulted in the most wrong predictions. However, as reflected by the high accuracy scores, a large majority of the predictions corresponding to these beans was also correct.

Comparing at the training and test accuracies of the different algorithms we see that these are very similar for all the models. This can be due to the nature of the dataset. The highly distinguishable classes makes the predictions simpler even for less complex models and our final choice of models are therefore generally of low complexity. This leads to less variation in our models, less overfitting and therefore similar train and test errors.

The random forest did a little better in terms of accuracy than the single decision tree. However, we did expect the random forest to do even better. We again assume that the similarities in results is due to the nature of the dataset.

As our results over all methods are very similar in terms of accuracy and prediction errors are not very critical in this case, one could make an argument to recommend using the method with lowest computation time. In the appendix section 6.2, underneath **Result 6** we have presented the training time for all the methods. The decision tree algorithm (0.03s) is significantly faster than the others, while the random forest (0.56s) is also quite fast compared to the neural network (1.9s) and logistic regression (3.6s). However, none of the algorithms has particularly slow training time in our problem.
A lot of publications has been made by various people analysing this dataset. Comparing our accuracy scores with a paper published by a student at the Univerity of Technologies and Economics in Warsaw, Grzegorz Słowiński [20], we find that our results are not as good as the one he presented. Perhaps some of our models have missing implementation. For instance, we did not implement a learning schedule for our neural network, something that might have help the model give higher accuracy. Also perhaps, we could have increased the model complexities even further, though this could have lead to overfitting. It is also possible that we did not pick the best models for the relevant data set. We could also have looked at algorithms such as boosting or support vector machine.

4.3 Bias-variance trade-off (regression case)

Now, we are going to look at the bias-variance trade-off for three different machine learning algorithms; a neural network, decision tree and random forest. The bias variance trade-off analysis can be understood in section 3.10. We will look at some errors against the complexity of the models, where the complexity parameter is somewhat different from one method to another. We are studying the housing data set (2.2). We will use all the features utilized by the PCA (3.3), and scale the data as described in section 3.4.

4.3.1 Neural network

We need to optimize the neural network parameters before we perform the bias-variance trade-off analysis.

Batch size and momentum parameter

We have chosen to go for the hidden activation function RELU in our initial optimization. Now, we want to find the optimal batch size and momentum parameter by looking at a heatmap with the

momentum parameter and batch size at the x- and y-axis (respectively). The plot below (figure 17) shows us the R2-score for testing data by different batch sizes and momentum parameters. By looking at the plot, it seems like the optimal values are on the right hand side, so a greater momentum parameter and it is closer to indifferent to the batch_size. We have chosen to continue with a batch size of 120, and a gamma value of 0.8 since those give one of the best R2-scores and a lower momentum are more safer/stable than a higher one.

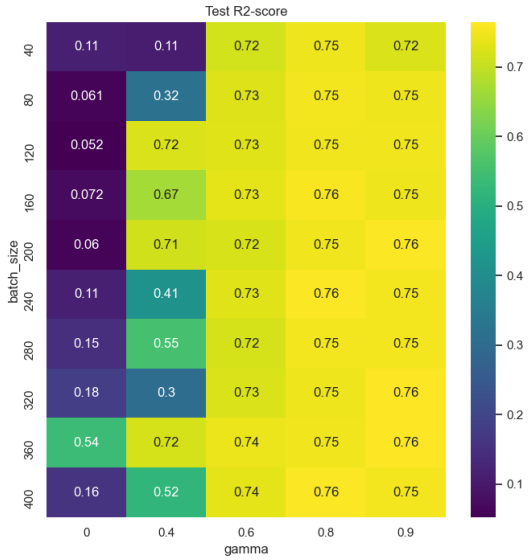

Figure 17: Data: testing, R2-score of different batch sizes and gamma values

Hyperparameters learning rate and lambda

Now we want to find the optimal hyperparameter learning rate and lambda by making a grid search. Now, by the parameters we tuned for the batch size, momentum (in the SGD) and initial values of the architecture of the neural network, we are able to use those values for finding the optimal lambda and eta values. We will try to find the parameters, η and λ, with the highest R2-score. The figure, 18 tells us that the best learning rate, η, is around 10^{-2} and 10^{-3} and the most stable lambdas are between 10^{-4} and 10^{-5}.

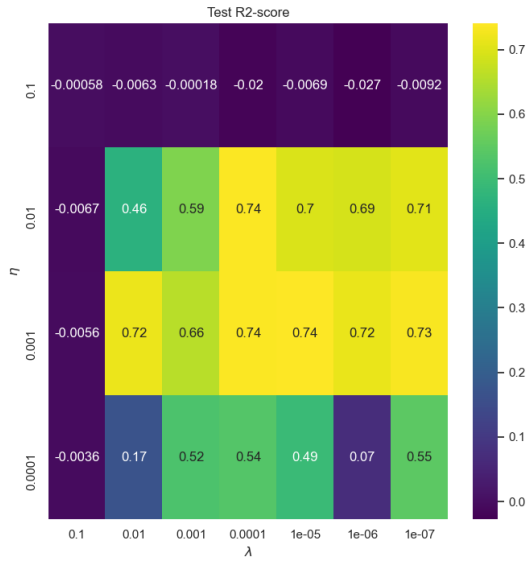

Figure 18: data: testing, R2-score of different lambda- and eta values

We can see at fig. 18 that for a higher λ value, the model will not be as overfitted as it would be without such a parameter. Also if the learning rate exceeds 10^{-2}, we observe occurrences of exploding- or vanishing gradients, which leads to a poor model. With a too low learning rate, we observe that in some cases the training has been too slow to reach well tuned parameters. This could be fixed by setting up the number of epochs, but this would require more computing power which is something we want to avoid if possible.

We conclude from this optimizing that the optimal parameters are: $\eta = 10^{-2}$, $\lambda = 10^{-4}$, $\gamma = 0.8$, batch size $= 120$.

Now, we are ready for doing the bias-variance trade off with the optimal values for the neural network

The complexity of a neural network can be regarded as the number of hidden layers, since we introduce more parameters to tune with each new layer, yielding more degrees of freedom. Number of nodes per hidden layer could also represent the complexity, but we are content with studying number of layers. We get the following plot, when studying the error vs. complexity of the optimized neural network.

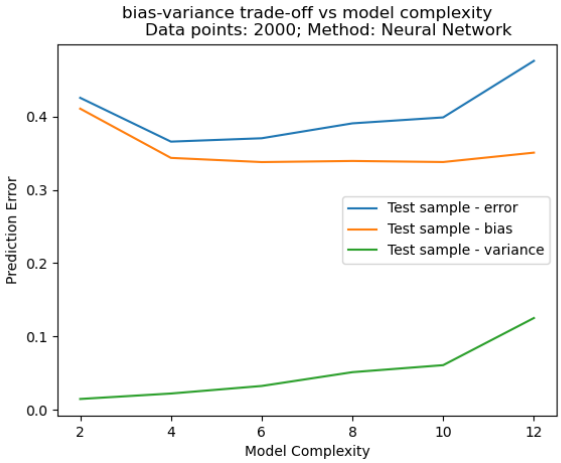

Figure 19: data: testing; bias-variance trade-off (neural network)

Figure 19 shows us the bias, variance and total MSE for the model as a function of complexity where we use the bootstrap resampling method for the predicted values. In the plot, we recognize that the bias will decreases when the complexity increases. This makes sense, as the model will not always be able to fit a complicated data set using a neural network of few hidden layers, the variance for such models is always low in the results of low complex models. When the complexity increases, we usually see the variance increase as well. These two effects makes it such that the MSE will often have a minimum when the complexity is not too high or too low, and hence there is an optimal trade-off between bias and variance.

4.3.2 Decision tree

In the decision tree algorithm, the depth is the parameter that can be used to represent the complexity of the model. A deep tree requires many branches which means many degrees of freedom. The depth is the most important when tuning this algorithm, and therefore we do not need any pre-tuning before we look at the bias-variance trade-off.

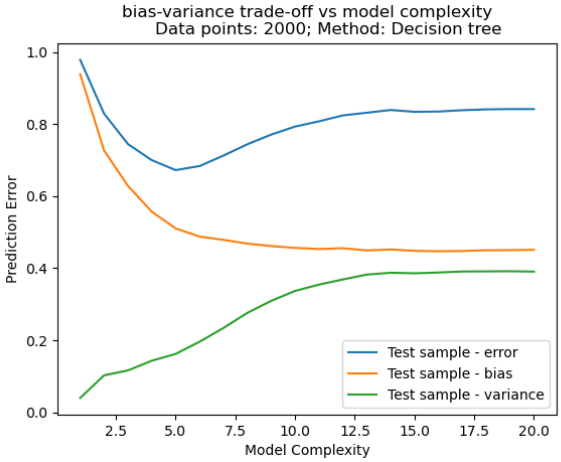

Figure 20: data: testing; bias-variance trade-off (decision tree)

Figure 20 shows us the same tendencies as in figure 19. The more complex model will result in an increasing variance and decreasing bias, and the optimal mean squared error is clearer in this plot.

We can see that a tree depth of 5 gives the lowest MSE, meaning this value gives the best trade-off between bias and variance. We observe that the total mean square error increases somewhat dramatically when we move away from the optimal complexity, meaning the error is sensitive to the complexity parameter.

4.3.3 Random forest

In the random forest algorithm, we study both the tree depth and number of trees as complexity parameters. The tree depth represents the complexity of each tree, while an increase in number of trees will make the random forest's complexity increase since the total complexity is number of trees times complexity in each tree. We will first look at how the bias/variance trade-off behaves when increasing the depth of each tree with the number of trees in the model is set to Scikit's default value of 100 trees. Later, we fix the tree sizes and study the behaviour when changing the number of trees.

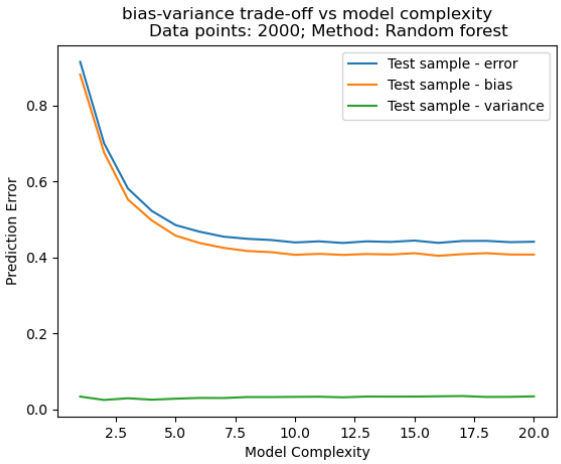

Figure 21: data: testing; bias-variance trade-off (Random forest)

Figure 21 shows us that the random forest algorithm does not suffer much in the variance for the increase of depth. The almost constant (low) variance is an expected result as the method of a random forest aims to decrease the variance of the dataset 3.9. On the other hand, we see that the bias is larger in the beginning, decreasing with complexity before it converges. This behaviour is similar to that of a single tree indicating that if we increase the maximum tree depth, we end up with many trees that are similar to the single decision tree. The optimal tree depth is not that easy to state as we expect that even weak learners with high bias will produce good predictions. It is however important to choose a depth were there is not too much bias in each tree as the bias in the random forest will be the same as the bias in each tree. We conclude that in this case the optimal tree depth is somewhere around 6 or 7.

Now, we let the maximum depth of each tree be constant at 7 while changing the number of trees in the model. The results is the following plot:

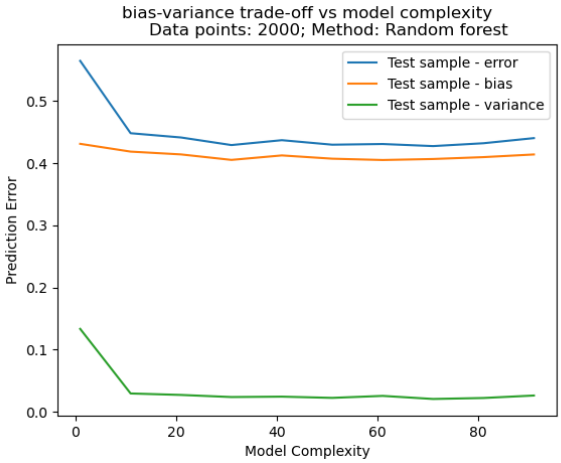

Figure 22: data: testing; bias-variance trade-off (Random forest)

It is evident from this plot that too few trees in the model results in much more variance. However, the variance decreases really fast when adding more trees to the model, before converging at a value close to zero. On the other hand, the bias is almost constant at a value around that of single tree with max depth 7. In this case, the optimal number of trees is much lower than 100 that we used before. A number around 40 trees seems to be sufficient.

4.3.4 Comments and discussion of the methods

In this part of the project we have looked at how the bias/variance trade-off is different for a neural network, a decision tree and a random forest in a regression problem. In general, the neural network had the lowest bias and the random forest the lowest variance. As for the decision tree, we observed that this algorithm is more unstable than the others with significantly more variance as the complexity increases. This is expected behavior as it is known that the decision trees have high variance in general.

Looking at the variance of the random forest, we see that this is the lowest amongst the three methods, no matter level of complexity. This is however no big surprise as the random forest algorithm improves the single decision tree only in terms of variance reduction (3.9). In a random forest, the bias in the final model is the same as for any of the trees within the model. Since these trees are weak learners and has a high bias by construction, we had expected the bias of the single decision tree model to be a little lower than that of the random forest. However, from the plots we showed earlier it seems that the bias of the decision tree and the random forest are very much the same. Due to the random forest's low variance, it performs clearly better than the decision tree in terms of error.

Comparing the random forest to the neural network it is evident that the neural network is able to give somewhat lower mean square error at optimal complexity. However, the lower variance of the random forest results in a more stable and easy to train model. In other words, the random forest does not really suffer from overfitting for high complexity models due to its low variance, while the neural network is more dependant on choosing just the right complexity as it is prone to both over- and underfitting.

5 Conclusion

First of all, we have gained some interesting insight about the beans in the dry beans data set when looking at the different confusion matrices (figure 9, 12, 14 and 16). The Dermason and Sira beans showed to be the most correlated, as trying to classify these beans gave the most confusion. On the other hand, the Bombay bean did not contribute at all to the prediction error indicating that this bean is uncorrelated with the other beans in the data set, and has some feature(s) that separates it from the others.

We learned that after we applied PCA to the data sets, we did not need to use every component in the input data for the machine learning methods to be accurate. As we got almost 100% explained variance ratio for only 3 components, we saved training time by only using these three and not the entire input data set. Although the PCA works great for prediction on the data, it will lead to some loss of interpretability of final models. Especially for the decision tree method, where the algorithm is often easy to understand by the splitting regarding the input features.

In the first part of the project, we tried to find the optimal algorithm for the classification data set about dry beans 2.1. We did this by optimizing the different machine learning algorithms by minimizing a cost function. The algorithms in this case were neural network, logistic regression, random forest and decision trees. Neural network has the most parameters to be tuned with its unique architecture. Both the neural network and logistic regression need to optimize the parameters in the stochastic gradient descent (SGD) algorithm, where the learning rate has a huge impact on the result. A large learning rate will probably jump over a minimum of the cost-function, while a too small learning rate will have a hard time converging. The momentum parameter in the SGD will help a lower learning rate converge faster, and the batch size will help the computational expenses since the model do not need to take the whole data set in consideration of the training at all times. For the method decision tree, the depth of the tree is the only parameter of importance. For a random forest, one also has consider the number of trees. Since the two latter machine learning methods have a smaller number of parameters to consider, it was easier to implement and tune these.

For the bias-variance analysis, we applied three methods to a regression data set and studied the bias and variance of each method as a function of model complexity. Model complexity is subjective to the specific model in question, and one has to interpret what parameters are most essential to represent the complexity. For the neural network, we chose number of hidden layers as the complexity parameter, and we got expected results: for low complexity, the model was not able to fit the data well which was shown in the high bias, while the variance was low. Meanwhile, for high complexity, the bias decreased as the neural network got more parameters to fit the data, but the variance increased as the model started overfitting the data. We found that there existed a perfect complexity where the total mean square error is lowest, meaning it is the perfect trade-off between bias and variance.

In the case of the decision tree, the depth was the most natural parameter to represent complexity as this decides how many branches the tree will split into. The decision tree showed similar results as the neural network: the bias decreased with complexity while the variance increased, yielding a perfect tree depth which trades off bias and variance well. Compared to the result for neural network, the mean square error increased more dramatically when moving away from the optimal complexity, meaning the total error was more sensitive to the complexity parameter.

In the final case with the random forest, the analysis was different. First, we looked at tree depth as model complexity. We then as unusual saw a decrease in bias against complexity, but the variance was always low due to the high number of trees. We then looked at the number of trees as complexity parameter, and this time the variance decreased instead of increased, while the bias was close to constant. This means no matter if we interpret tree depth or number of trees as

complexity parameter, the total mean square error decreases, but in the first case it is because of decreasing bias while in the other it is due to decreasing variance. This analysis would in fact result in: The higher the complexity, the better the model. We can obviously not increase the complexity to infinity as this will take infinite computing power, but we know that we can always expect accurate results if we choose both tree depth and number of trees to be high.

Our time with this project was limited, so it was necessary to restrict the project in some way. Further analysis we could have done would be to look at more types of machine learning algorithms, both for finding the best model regarding the accuracy score and to get a deeper insight in several methods.

6 Appendix

6.1 Github repository

Github REPO - project 3

6.2 How to reproduce the results/ figures (classification problem)

In this section, we will provide enough information to reproduce the results/ figures we are using in this project. Under (Result, Figure) X, we will give the necessary parameters and which test to run to achieve the correct figure. The tests we will refer to, is the tests inside **test_project_3.py**, attached in the github repository (section 6.1).

Result 1

Run **test 9**, and get the following results to the terminal:

```
>> RUNNING TEST 9:
>> The distribution of the targets
0:  0.149 (SEKER)
1:  0.097 (BARBUNYA)
2:  0.038 (BOMBAY)
3:  0.120 (CALI)
4:  0.142 (HOROZ)
5:  0.194 (SIRA)
6:  0.261 (DERMASON)
Total explained variance ratio (of 1 component):  1.000
Total explained variance ratio (of 2 component):  1.000
Total explained variance ratio (of 3 component):  1.000
```

```
>> Time: 1.9454131126s (Neural network)
>> Time: 3.6890549660s (Logistic regression)
>> Time: 0.0280399323s (Decision tree)
>> Time: 0.5625009537s (Random forest)
```

Result 2

Run **test 1** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 30
batch_size = 100
lmbda = 1e-5
eta = 1e-2
gamma = 0.9
hidden_nodes = 14
hidden_layers = 4
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 1 <<
Accuracy for training: 0.9114420062695925
Accuracy for testing: 0.9174258007640317
```

Result 3

Run **test 11** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 200
batch_size = 500
gamma = 0.8
eta = 1e-2
lmbda = 1e-5
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 11 <<
Training accuracy: 0.9065438871473355
Testing accuracy: 0.9062591830737584
```

Result 4

Run **test 5** with the following parameters:

```
n_components = 3
m_observations = 13611
max_depth = 5
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 5:
Accuracy_train = 0.9042907523510971
Accuracy_test = 0.9021451660299735
```

Result 5

Run **test 7** with the following parameters:

```
n_components = 3
m_observations = 13611
max_depth = 6
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 7:
Accuracy_train = 0.9141849529780565
Accuracy_test = 0.9133117837202468
```

Result 6

Run **test 10**

Terminal output:

```
>> RUNNING TEST 10:
>> Time: 1.9454131126s (Neural network)
>> Time: 3.6890549660s (Logistic regression)
>> Time: 0.0280399323s (Decision tree)
>> Time: 0.5625009537s (Random forest)
```

Figure 5

Run **test 2** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 80
batch_size = 500
gamma = 0.8
eta = 1e-2
lmbda = 1e-4
act_func = 'relu'
```

Figure 6

Run **test 3** with the following parameters:

```
n_components = 3
m_observations = 13611
eta = 1e-2
lmbda = 1e-4
act_func = 'relu'
hidden_nodes = 14
hidden_layers = 4
```

Figure 7

Run **test 4** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 80
gamma = 0.9
batch_size = 100
```

```
act_func = 'relu'
hidden_nodes = 14
hidden_layers = 4
```

Figure 8

Run **test 4** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 80
gamma = 0.9
batch_size = 100
act_func = 'sigmoid'
hidden_nodes = 14
hidden_layers = 4
```

Figure 9

Run **test 1** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 30
batch_size = 100
lmbda = 1e-5
eta = 1e-2
gamma = 0.9
hidden_nodes = 14
hidden_layers = 4
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 1 <<
Accuracy for training: 0.9114420062695925
Accuracy for testing: 0.9174258007640317
```

Figure 10

Run **test 12** with the following parameters:

```
n_components = 3
m_observations = 13611
eta = 1e-3
lmbda = 0
```

Figure 11

Run **test 13** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 100
batch_size = 500
gamma = 0.8
```

Figure 12

Run **test 11** with the following parameters:

```
n_components = 3
m_observations = 13611
n_epochs = 200
batch_size = 500
gamma = 0.8
eta = 1e-2
lmbda = 1e-5
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 11 <<
Training accuracy: 0.9065438871473355
Testing accuracy: 0.9062591830737584
```

Figure 13

Run **test 6** with the following parameters:

```
n_components = 3
m_observations = 13611
```

Figure 14

Run **test 5** with the following parameters:

```
n_components = 3
m_observations = 13611
max_depth = 5
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 5:
Accuracy_train = 0.9042907523510971
Accuracy_test = 0.9021451660299735
```

Figure 15

Run **test 8** with the following parameters:

```
n_components = 3
m_observations = 13611
```

Figure 16

Run **test 5** with the following parameters:

```
n_components = 3
m_observations = 13611
max_depth = 6
confusion_result = True
```

Terminal output:

```
>> RUNNING TEST 7:
Accuracy_train = 0.9141849529780565
Accuracy_test = 0.9133117837202468
```

6.3 How to reproduce the figures (regression problem)

In this section, we will provide enough information to reproduce the figures we are using in this project. Under Figure X, we will give the necessary parameters and which test to run to achieve the correct figure. The tests i will refer to, is the test inside **test_project_3.py**, attached in the github repository (section 6.1)

Figure 17

Run **test 22** with the following parameters:

```
n_components = 8
m_observations = 2000
eta = 5e-4
lmbda = 1e-5
hidden_nodes = 20
hidden_layers = 3
act_func = 'relu'
```

Figure 18

Run **test 23** with the following parameters:

```
n_components = 8
m_observations = 2000
gamma = 0.8
batch_size = 120
```

```
hidden_nodes = 60
hidden_layers = 7
act_func = 'relu'
epochs = 500
```

Figure 19

Run **test 21** with the following parameters:

```
n_components = 8
m_observations = 2000
gamma = 0.8
batch_size = 120
eta = 1e-2
lmbda = 1e-4
hidden_nodes = 60
epochs = 500
act_func = 'relu'
```

Figure 20

Run **test 24** with the following parameters:

```
n_components = 8
m_observations = 2000
max_degree = 20
n_bootstrap = 1000
```

Figure 21

Run **test 25** with the following parameters:

```
n_components = 8
m_observations = 2000
max_degree = 20
n_bootstrap = 100
```

Figure 22

Run **test 26** with the following parameters:

```
n_components = 8
m_observations = 2000
max_degree = 100
n_bootstraps = 10
```

7 Bibliography

[1] Alamy. *Dermason beans*. URL: https://c8.alamy.com/comp/BJ7GP5/whole-dried-white-dermason-beans-close-up-full-frame-top-shot-phaseolus-BJ7GP5.jpg.

[2] L Breiman. *Random Forests*. URL: https://link.springer.com/article/10.1023%2FA%3A1010933404324.

[3] Coban Brothers. *Sira beans*. URL: http://www.cobanbros.com/index.php/product/white-beans-sira-type/.

[4] Jason Brownlee. *Confusion matrix*. URL: https://machinelearningmastery.com/confusion-matrix-machine-learning/.

[5] Riccardo De Bin. *Lecture 8 2021, STK-IN4300*. URL: https://www.uio.no/studier/emner/matnat/math/STK-IN4300/h21/slides/lecture_8.pdf.

[6] Alleshier GmbH. *Picture of several beans*. URL: https://alleshiergmbh.com/en/our-products/legumes-2/.

[7] Morten Hjort-Jensen. *Week 37 Lecture notes*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week37/html/week37.html.

[8] Morten Hjort-Jensen. *Week 40: From Stochastic Gradient Descent to Neural networks*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html.

[9] Morten Hjort-Jensen. *Week 41 Lecture notes*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html.

[10] Morten Hjort-Jensen. *Week 43 Lecture notes*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week43/html/week43.html.

[11] Tin Kam Ho. *Random Decision Forests*. URL: https://web.archive.org/web/20160417030218/http://ect.bell-labs.com/who/tkh/publications/papers/odt.pdf.

[12] IBM. *Bagging*. URL: https://www.ibm.com/cloud/learn/bagging.

[13] Xiaoxiao Lei. *Decision Trees*. URL: https://dark417.github.io/MachineLearning/sv_trees/.

[14] Ilker Ali OZKAN Murat KOKLU. *Dry Bean Dataset*. URL: https://archive.ics.uci.edu/ml/datasets/Dry+Bean+Dataset.

[15] Sigurd Holmsen Øystein Bruce. *Project 1 report*. URL: https://github.com/oystehbr/FYS-STK4155/tree/main/project1/report.

[16] Sigurd Holmsen Øystein Bruce. *Project 2 report*. URL: https://github.com/oystehbr/FYS-STK4155/tree/main/project2/report.

[17] Picture. *Barbunya beans*. URL: https://cdn.shopify.com/s/files/1/0561/2834/0156/articles/IMG_2284-1080x700.jpg?v=1618481365.

[18] Scikit-learn. *sklearn.datasets.fetch_california_housing*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_california_housing.html#sklearn.datasets.fetch_california_housing.

[19] scikit-learn. *PCA by scikit*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[20] Grzegorz Słowiński. *Dry Beans Classification Using Machine Learning*. URL: http://ceur-ws.org/Vol-2951/paper3.pdf.

[21] Tensorflow. *Tensorflow Keras documentation*. URL: https://www.tensorflow.org/api_docs/python/tf/keras.

[22] Tensorflow. $to_categoricaldocumentation$. URL: https://www.tensorflow.org/api_docs/python/tf/keras/utils/to_categorical.

[23] Buy In Turkey. *Bombay beans*. URL: https://www.buyinturkey.com/index.php?route=product/product&product_id=24067277/.

[24] Sommerfelt Nevland Murugesu Vollan. *Project 2*. URL: https://github.uio.no/erlingpv/Project2/blob/master/report_project2.pdf.

[25] Wikipedia. *Wisdom of the crowds*. URL: https://en.wikipedia.org/wiki/Wisdom_of_the_crowd.