Machine Learning Methods for Diabetes Diagnosis

Justina Grønbekk,
Sophia Gaupp,
Mina Piasdatter Walderhaug

UNIVERSITETET I OSLO

Project 3
of FYS-STK4155 – Applied Data Analysis and Machine Learning.
at the University of Oslo
Oslo, Norway

December 2024

Abstract

In a rapidly evolving world, modern lifestyles have made it easier than ever to live comfortably with minimal physical activity, with conveniences like fast-food delivery at our fingertips. This shift has contributed to a global rise in Type 2 diabetes. Beyond its direct symptoms, diabetes acts as a co-factor for numerous severe diseases further escalating its impact. Thus Type 2 diabetes poses one of the most significant challenges to healthcare systems worldwide.

Efficient and early diagnosis is crucial in addressing these challenges. With sufficient data, machine learning (ML) methods are well-suited to this use case. With ML methods we can accurately classify individuals, thereby assisting healthcare professionals in making well-informed and early diagnoses.

In this study, we evaluate the Pima Indian Diabetes dataset. We implement and compare three distinct machine learning models: logistic regression, a neural network, and decision trees. Additionally, we extend our analysis by applying ensemble methods like extreme gradient boosting and random forest to enhance performances. Our results demonstrate that the neural network achieved the highest accuracy of 81.2%, surpassing several previous machine learning approaches on this dataset.

We believe that further development of this flexible and reliable approach to leverage neural networks in diabetes diagnosis could represent a significant step forward in addressing this global health crisis.

Contents

Chapter 1

Introduction

The prevalence of Diabetes Mellitus Type 2 (T2D) has increased dramatically in recent years. In the year 2022 there were 830 million people living with T2D (World Health Organization 2024) which makes up around 10% of the worldwide population. T2D is a chronic and noncommunicable metabolic disorder characterized by persistent hyperglycemia, primarily due to insulin resistance and a relative deficiency in insulin secretion. Diabetes can cause blindness, kidney failure, heart attacks, strokes and lower limb amputation (World Health Organization 2024).

With proper management through lifestyle modifications, medication, and insulin therapy, individuals diagnosed with T2D can maintain blood glucose levels within a healthy range and significantly reduce the risk of complications. Early detection of T2D is crucial, as timely intervention can prevent or delay the onset of complications, improve long-term outcomes, and enhance the overall quality of life for individuals affected by T2D. Despite this, patients often exhibit few symptoms in the early stages, making diagnosis challenging (Chang et al. 2022).

"The increase in global health expenditure due to diabetes has grown from US $232 billion in 2007 to US $966 billion in 2021 for adults aged 20-79 years–a 316% increase over 15 years." (Health Economics and Research 2023). Developing tools for early-stage detection of T2D is therefore essential to address the increasing global burden for healthcare systems worldwide. A compelling dataset for studying predictive models is the Pima Indian Diabetes Dataset. The Pima Indians, a population indigenous to the southwestern United States and northern Mexico, have been the focus of diabetes research due to their disproportionately high rates of T2D. It is a highly challenging dataset to work with, and achieving good classification results is difficult. Previous research, leveraging machine learning approaches implemented in this study were not

able to achieve higher classification accuracies than 80% (Karatsiolis and Schizas 2012; Chang et al. 2022).

In this study, we aim to outperform existing accuracy benchmarks. To achieve this, we first conduct preprocessing of the dataset to address missing values. We implemented three machine learning techniques: logistic regression, a neural network, and a decision tree model. In addition we boosted the neural network with extreme gradient boosting and created a random forest out of decision trees.

We base our model evaluation on the classification accuracy and ROC AUC curves. Our neural network achieved the highest accuracy of 81.2%. This performance surpasses values from similar methods reported in our reviewed literature.

Chapter 2

Methods

2.1 Dataset

We utilized the Pima Indian Diabetes dataset which has been the focus of many previous studies (Sankar Ganesh and Sripriya 2020; Karatsiolis and Schizas 2012; Lakhwani et al. 2020; Chang et al. 2022). The dataset has consistently proven to be challenging for classification tasks. For benchmarking, we selected two key studies for comparison: Chang et al. 2022 and Karatsiolis and Schizas 2012.

Table 2.1: Literature values of accuracy of different Machine Learning methods.

Method	Accuracy value from literature	Source
Logistic regression	77%	Karatsiolis and Schizas 2012
Neural network	77%	Karatsiolis and Schizas 2012
Random forest	80%	Chang et al. 2022

Utilizing a random forest model reported the highest accuracy of 80% in the literature (Chang et al. 2022). Both logistic regression and neural network yielded accuracy values of 77% (Karatsiolis and Schizas 2012).

The dataset, originally provided by the United States National Institute of Diabetes and Digestive and Kidney Diseases, is publicly available through multiple sources, including Kaggle and GitHub. It comprises 768 cases of women aged 21 years and older. It includes eight features: number of pregnancies, glucose levels, blood pressure, skin thickness, insulin levels, BMI, diabetes pedigree function and age.

We are working with real data as opposed to randomly generated data, to make it more applicable to real-life datasets and applications.

2.2 Data handling

To ensure the reproducibility of our results, we implemented a random seed. We set the seed to 2024 consistently across all methods to enhance the comparability of our outcomes.

2.2.1 Data Peprocessing

To preprocess the data, we addressed values that were clearly implausible. For example, a skin thickness of 0 or a blood pressure of 0 is unlikely to represent real data and is more likely indicative of missing values. To handle this, we first identified all such nonsensical values by determining realistic ranges for each feature. We then replaced all impossible values with "Not a number" (NaN).

One way of dealing with NaN values is to remove all the rows containing NaN values. In our case, this approach would leave us with only about one-third of the original dataset entries. Since this would result in a substantial reduction of data, we opted against this method.

Another approach, which we chose, is to replace all NaN values with the mean value of their respective columns. This ensures that the NaN values do not distort the existing data in the column, while still allowing us to retain information from the other columns and preserve a significant portion of the dataset.

There are also more advanced approaches to handling NaN values, like multiple imputation, k-nearest neighbor or deep learning based imputation. However, we decided against this to keep our project straightforward and comprehensible, avoiding unnecessary complexity.

2.2.2 Splitting the Data

We split the dataset into training and test sets to evaluate how well our models perform. To achieve this, we chose an 80/20 split, using 80% of the data for training and 20% for testing. This ratio was consistently applied across all implemented methods to ensure comparability between them.

2.3 Machine Learning Algorithms

Historically speaking logistic regression was a cornerstone of early statistical modeling. As computational power grew, neural networks emerged to tackle more complex problems by leveraging the hierarchical processing of features. Random forests were

developed to address overfitting and instability in decision trees, becoming a powerful tool for tabular data.

Neural networks extend logistic regression by stacking multiple logistic-like units (neurons) and introducing non-linearity through activation functions and hidden layers. Both neural networks and random forests handle complex, non-linear patterns, but random forests rely on tree-based structures while neural networks use layered representations. Random forests provide a more flexible alternative to logistic regression, especially for non-linear problems, but sacrifice interpretability.

All the methods we implemented operate on labeled data, making all our models supervised learning models.

We are aiming to implement diagnostic models to correctly classify individuals as diabetic or non-diabetic based on a set of features.

2.3.1 Logistic regression

Logistic regression is a binary classification technique used to model the probability of an outcome belonging to one of two classes. Its simplicity aside we reached good results with this method in our last project (Grønbekk, Gaupp, and Walderhaug 2024) which lead us to incorporating it here again.

The model predicts the probability $p(\mathbf{x})$ of the target variable y taking the value 1, given a feature vector $\mathbf{x}$, as:

$$p(\mathbf{x}) = \frac{1}{1 + e^{-z}}, \tag{2.1}$$

where z is a linear combination of the input features:

$$z = \mathbf{w}^T \mathbf{x} + b, \tag{2.2}$$

with $\mathbf{w}$ representing the weights, b the bias term, and $\mathbf{x}$ the feature vector. The predicted class is determined by applying a threshold (typically 0.5) to the predicted probability:

$$\hat{y} = \begin{cases} 1, & \text{if } p(\mathbf{x}) \geq 0.5, \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

Optimization

To optimize the model parameters $\mathbf{w}$ and b, we minimize the cross-entropy loss function:

$$\mathcal{L}(\mathbf{w}, b) = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i)) \right], \tag{2.4}$$

where N is the number of data points, y_i is the true label, and $p(\mathbf{x}_i)$ is the predicted probability for the i-th data point. This is a convex function and any local optimum is the global optimum. (Hjorth-Jensen 2024).

Stochastic Gradient Descent

We used Stochastic Gradient Descent (SGD) to minimize the loss function. SGD calculates the gradient using a random subset of the data (a minibatch). The gradient for a minibatch B_k with size M is approximated as:

$$\nabla \mathcal{L}_{B_k}(\mathbf{w}) = \frac{1}{M} \sum_{i \in B_k} \nabla \mathcal{L}(\mathbf{w}; \mathbf{x}_i, y_i). \tag{2.5}$$

The parameters are updated iteratively as:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \mathcal{L}_{B_k}(\mathbf{w}), \tag{2.6}$$

where η is the learning rate.

Momentum

To improve convergence, we employed SGD with momentum. Momentum helps the algorithm navigate ravines in the cost function landscape by incorporating the direction of the previous updates into the current step. The update rule with momentum is:

$$\mathbf{v} = \gamma \mathbf{v} + \eta \nabla \mathcal{L}_{B_k}(\mathbf{w}), \tag{2.7}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \mathbf{v}, \tag{2.8}$$

where γ is the momentum coefficient (here $\gamma = 0.9$) and $\mathbf{v}$ is the velocity term.

Hyperparameters

Multiple hyperparameters were adjusted when fine-tuning this model in order to achieve the best possible training and test accuracy:

The **learning rate** controls the step size for updating model parameters during gradient descent.
The **number of iterations (epochs)** determines how many times the algorithm passes through the dataset.
The **batch size** is the number of data points used to compute the gradient in each iteration.
The **regularization parameter** λ helps prevent overfitting by penalizing large parameter values.

Implementation

We utilized existing code from our previous report Building a Neural Network and adapted it for our purposes.

Finally, we implemented the Logistic Regression model from SciKit-Learn (Pedregosa et al. 2011) to benchmark our fine-tuned implementation against a widely used library.

2.3.2 Neural Network

"Neural networks (NNs) are computational models inspired by the structure and functioning of biological neural networks, particularly those in the human brain. They consist of interconnected units called neurons, organized into layers. Information flows from the input layer to a number of hidden layers, and finally out through the output layer" (Grønbekk, Gaupp, and Walderhaug 2024).

Neural networks consist of layers of interconnected nodes (or "neurons"), where each node in one layer is connected to every node in the subsequent layer. Training a Neural Network consists of two main steps: feed-forward loop and back propagation. A more descriptive explanation can be found in our report: Building a Neural Network.

We chose to implement a feed-forward neural network (FFNN) with backpropagation as we have experienced it to be effective for classification in Building a Neural Network. We implemented it using logistic functions. It is commonly very adaptive to new unseen data once trained. We also wanted to see if boosting methods would improve the learned classifications from a NN.

Loss Function

"A loss function is a mathematical function that quantifies how well the predictions of a machine learning model match the actual data. In essence, it measures the error or difference between the predicted output from the model and the true target values. The goal of training a machine learning model is to minimize this loss function by adjusting the model's parameters" (Grønbekk, Gaupp, and Walderhaug 2024).

For a classification problem, a natural choice of loss function would be the cross-entropy function. It gives us binary output, which is what we want in our case as we only have two classifications. Look back to Section 2.4.2 for explanation of the cross-entropy function.

Optimization

The choice of optimization algorithm is important for the learning part in a NN. Some common optimizers are RMSprop, Adagrad and Adam. We only implement Adam in this project, and we will explain in more detail what this optimizer does.

"The Adaptive Moment Estimation (Adam) optimizer combines the benefits of both

AdaGrad and RMSprop by maintaining separate adaptive learning rates for each parameter and incorporating momentum into the update process. Adam computes two different running averages: the first is for the gradient (similar to momentum), and the second is for the squared gradient (similar to RMSprop). The optimizer uses these averages to adjust the learning rates for each parameter adaptively.

The update rules for Adam are given by:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t
\end{aligned}
$$

Where:

- m_t and v_t are the running averages of the gradient and squared gradient, respectively,
- β_1 and β_2 are hyperparameters controlling the exponential decay rates for these moving averages, typically set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$,
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected versions of the moment estimates.

Adam adapts the learning rate for each parameter, taking into account both the magnitude and the frequency of the updates. This makes Adam highly effective in scenarios with noisy or sparse data and helps maintain a fast and stable convergence rate" (Grønbekk, Gaupp, and Walderhaug 2024).

Hyperparameters

We have multiple hyperparameters to set for the neural network. Batch size, learning rate, number of epochs, number of layers and number of nodes. These can greatly affect the predictions of our neural network and are important to fine-tune.

Implementation

In this report, we used the Pytorch library to implement the FFNN. We then used the learned features from our trained neural network on an extreme gradient boosting model. This was implemented using the XGBoost library. We then get our final predictions of labels, and compare with the actual labels of the dataset to calculate the accuracy. We also use the Scikit-learn library (Pedregosa et al. 2011) for certain

metrics.

2.3.3 Decision trees

A decision tree is a supervised ML algorithm that can be used for both logistic and linear regression problems. They form the backbone for many ensemble methods like random forest and gradient boost. The name comes from the characteristic tree-like shape that the algorithm takes.

A tree consists of a root node, interior nodes and leaf nodes which are connected by branches (Hjorth-Jensen 2024). The root node represents the entire dataset, and the leaf nodes are the final predictions. When using a decision tree for prediction, the queried cases start in the root node, goes through the interior nodes, and finally ends in a prediction in the leaf nodes.

A decision tree is grown through recursive binary splitting (Hjorth-Jensen 2024). That is, for every node except the leaves, there are two child nodes branching from it. The splits are decided from a set of predetermined features. The algorithm finds the most descriptive features, and splits the tree along them. The most descriptive features are those that most accurately produce the target feature. The process of finding the most descriptive feature is repeated until a stopping criterion is met.

Decision trees are easy to interpret, and can be used to solve many different problems. They do not, however, achieve the same level of accuracy as other more complex algorithms. Decision trees also have a tendency to overfit if they get too large. A too large tree is less interpretable, losing said advantage for very complex tasks.

Building a Decision tree

We start by selecting the predictor x_j and a cutpoint s that split the predictor space into two regions R_1 and R_2, in such a way that we obtain the lowest MSE value (Hjorth-Jensen 2024). In other words, the regions R_1 and R_2 are respectively:

$$\{X|x_j < s\},$$

and

$$\{X|x_j \geq s\}.$$

They are created to minimize

$$\sum_{i:x_i \in R_1} (y_i - \overline{y}_{R_1})^2 + \sum_{i:x_i \in R_2} (y_i - \overline{y}_{R_2})^2.$$

For all j and s $\overline{y}_{R_1}$ is defined to be the mean of all the predictions in R_1(j,s), and $\overline{y}_{R_2}$

the mean predictions in R_2(j,s) (Hjorth-Jensen 2024).

The next split does not concern the entire predictor space, but is rather confined to one of the two regions R_1 or R_2. We choose to split the region that minimizes the MSE. The region is then split in two subregions, following the best predictor and cutpoint. This process is repeated until a stopping criterion is met.

The tree most common ways to split a node are:

- Misclassification error

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i \neq k) = 1 - p_{mk}.$$

- Gini index

$$g = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}).$$

- Information entropy

$$s = -\sum_{k=1}^{K} p_{mk} \log p_{mk}.$$

The gini index is a common choice, so we opted for this method of splitting.

Hyperparameters
We tune our decision tree model on the maximum depth of the tree. Some other parameters that could be tuned are maximum number of leaves or maximum number of features to be considered on each split.

If a decision tree has no **maximum depth**, it can grow very large. This can in turn lead to overfitting, and poor performance on unseen data. A decision tree thus needs to be balanced between having the required complexity to solve the task and avoid overfitting.

2.3.4 Random Forest

A random forest is a ML algorithm consisting of several decision trees on bootstrap training samples made to increase accuracy relative to a single decision tree (Hjorth-Jensen 2024). A chosen number of trees are built. Each time a split in a tree is introduced, the split is considered on a subset of the total number of features. The subset of features is usually chosen to be $\sqrt{total\ number\ of\ features}$, and is chosen anew for each split (Brownlee 2021).

Random forests are less prone to overfitting than decision trees. They are also able to achieve a high accuracy because of the aggregation of results from different trees. They are more computationally costly than simpler models like decision trees or logistic regression, and are therefore not always worth the trouble. Random forests also have the disadvantage of being less interpretable than the simple decision tree.

Hyperparameters

The **maximum depth** is less important for random forests than for decision trees. Because of the nature of the models, random forests are less prone to overfitting than decision trees. This means that there is less of a problem having deep trees for random forests than for decision trees.

Another hyperparameter is the **number of trees** in the forest. Generally more trees in the forest means higher accuracy. There comes, however, a point of diminishing returns. More trees means a higher computational cost, and at some point more trees will not improve the model.

One of the parameters we tune our xgboost random forest model on is called **eta**. This is described by the library's own website, as "Step size shrinkage used in update to prevent overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative". Eta has a range of [0,1]. A higher eta might make the model more prone to overfitting, but speeds up learning. A lower eta means slower learning, and scales down the contribution of each tree in the ensemble.

Implementation

We tried two different random forest models, one from Scikit-learn and one from XGBoost. They are different implementations of the same concept.

We tune our Scikit-learn (Pedregosa et al. 2011) algorithm on the number of trees in the forest and the depth of the tree. We also set bootstrap to true, and criterion to gini, ensuring bootstrapping and the use of the gini index to split nodes.

We tune our XGBoost random forest model on the number of trees in the forest, and eta. We also use a subsample of 0.9, and a colsample_bynode of 0.2, inspired by this article How to Develop Random Forest Ensembles with XGBoost.

2.3.5 Gradient Boost

Gradient boosting is a ML method for improving models by running several weak learners and weighting the learners based on the performance. A weak learner is an algorithm that is only slightly better than random guessing, like decision trees. Based

on a description made in these lecture notes gradient boosting can be described as such:

Suppose we have a cost function

$$C(f) = \sum_{i=0}^{n-1} L(y_i, f(x_i))$$

where y_i is our target and $f(x_i)$ the function meant to model y_i.

Then we:

1. Initialize our estimate $f_0(x)$.
2. For $m = 1 : M$, we
   (a) compute the negative gradient vector $\boldsymbol{u}_m = -\partial C(\boldsymbol{y}, \boldsymbol{f})/\partial \boldsymbol{f}(x)$ at $f(x) = f_{m-1}(x)$;
   (b) fit the so-called base-learner to the negative gradient $h_m(u_m, x)$;
   (c) update the estimate $f_m(x) = f_{m-1}(x) + h_m(u_m, x)$;
3. The final estimate is then $f_M(x) = \sum_{m=1}^{M} h_m(u_m, x)$.

2.3.6 Extreme Gradient Boost

Extreme gradient boost (XGBoost) is a Python library of gradient boosting methods, which require less resources than previous systems (Hjorth-Jensen 2024). The papers we are comparing to don't use XGBoost, but we felt it was a natural choice of algorithm.

Neural Network

There are multiple ways of implementing a NN combined with gradient boosting, such as mixed models or residual learning. In this project, we implemented what is called feature extraction. We chose to extract the features from a trained NN and use this as input to train the gradient boosting model. We wanted to combine these two methods as the boosting method is known to improve accuracies, and we wanted to see how effective they were in combination with a neural network.

2.4 Performance measures

2.4.1 Accuracy

Accuracy measures the proportion of correct predictions made by a model out of the total number of predictions. Calculated as the ratio of correctly predicted instances to the total amount of instances, accuracy is expressed as:

$$Accuracy = \frac{NumberofCorrectPredictions}{TotalNumberofPredictions}$$

When several models have a high accuracy, it might also be necessary to look at the confusion matrices for the models to make a good comparison.

This part was altered from Building a Neural Network.

2.4.2 Confusion Matrix

"A confusion matrix is a table that provides a detailed breakdown of a model's performance in classification tasks by displaying the true and predicted classifications for each class. It helps in identifying not only how often the model is correct (accuracy) but also the types of errors it makes. This matrix is especially useful when looking at model performance in medical use cases since mistakes in the different classes have widely different medical implications.

The confusion matrix consists of four main components in the case of binary classification:

True Positives (TP): Cases where the model correctly predicted the positive class.

True Negatives (TN): Cases where the model correctly predicted the negative class.

False Positives (FP): Cases where the model incorrectly predicted the positive class (also known as Type I error).

False Negatives (FN): Cases where the model incorrectly predicted the negative class (also known as Type II error)." (Grønbekk, Gaupp, and Walderhaug 2024).

2.4.3 AUC ROC

AUC ROC is short for Area under the Curve receiver operating characteristic curve. It plots the true positive rate against the false positive rate. Each point on the curve represents a specific threshold. A perfect classifier would have 100% of TP with 0% of FP. Such a classifier does not exist in reality. Yet we aim to build a classifier that comes as close as possible to the perfect classifier.

The AUC ROC gives even more details than the confusion matrix and insight into how TP and FP rates are interdependent and how they develop.

2.5 Python and Libraries

We use Python version 3.11, scikit-learn version 1.5.1 (Pedregosa et al. 2011), pytorch version 2.5.1+cpu , and xgboost version 2.1.2 (Chen and Guestrin 2016).

Chapter 3

Results

All plots in this section and any additional plots from our source code can be found in our Github repository.

3.1 Logistic regression

Hyperparameters
Our model showed the highest accuracy when choosing the following hyperparameters:

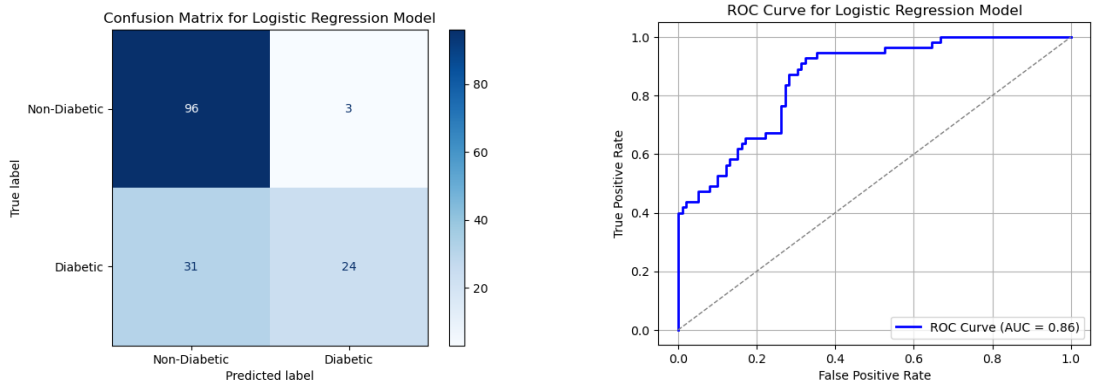
- learning rate = 0.0001
- number of iterations = 50000
- batch size = 30
- lambda = 0.01

The trimmed model left us with an accuracy on the test set of 77.36%.

Confusion Matrix
See below the Confusion matrix with 96 TN, 3 FP, 31 FN and 24 TP.

ROC AUC
The ROC AUC curve rises steeply at the beginning (near the origin) and flattens towards the top-right corner. An AUC of 0.86 was reported with a learning rate of 0.0001.

SciKit Learn
With the SciKit Learn logistic regression Pedregosa et al. 2011 we get an accuracy of 75.97%.



(a) Confusion Matrix

(b) ROC AUC

Figure 3.1: Learning rate was set to 0.0001, number of iterations to 50000, batch size to 30 and lambda to 0.01 in both plots.
(a) Confusion matrix with applied method being logistic regression. 768 cases divided in training test split in a ratio of 8:2 results in a total number of 154 test cases distributed in the confusion matrix.
(b) ROC AUC with applied method being logistic regression. X-axis showing the false positive rate and Y-axis showing the true positive rate.

3.2 Neural Network

Hyperparameters

HL	Number of nodes	Activation functions in the hidden layers	Activation function in the output layer	Batch size	Optimizer algorithm	LR	Epochs	Accuracy
2	8, 16	Sigmoid, Sigmoid	Leaky ReLU	32	Adam	0.001	200	77.3%
2	10, 8	Tanh, Tanh	Leaky ReLU	32	Adam	0.001	350	72.1%
2	12, 8	ReLU, ReLU	Leaky ReLU	32	Adam	0.001	350	78.6%
3	12, 8, 10	ReLU, ReLU, ReLU	Leaky ReLU	32	Adam	0.001	350	81.2%
3	12, 8, 10	ReLU, Sigmoid, ReLU	Leaky ReLU	32	Adam	0.001	350	82.5%
3	12, 8, 10	Sigmoid, Sigmoid, ReLU	Leaky ReLU	32	Adam	0.001	350	77.3%

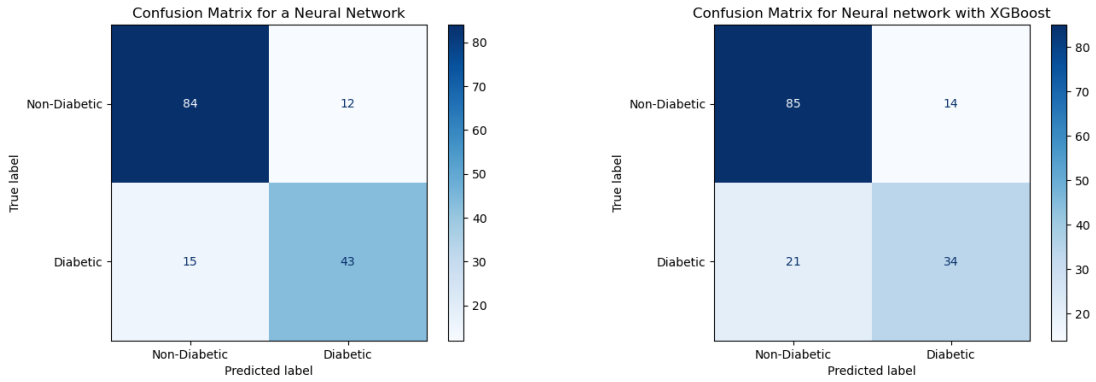
Table 3.1: Different hyperparameters for a neural network and the achieved accuracy. The different hyperparameters are number of hidden layers (HL), number of nodes, activation functions in the hidden layers, in the output layer, batch size, optimizer algorithm, learning rate (LR) and epochs. The initial guess for hyperparameters of our model is the top row. The highest accuracy achieved on the test data was 82.5%, highlighted in blue.

The highest accuracy on the test data was achieved when the hyperparameters were the following:

- number of hidden layers (HL) = 3

- number of nodes in each HL = 12, 8, 10
- activation functions in HL = ReLU, Sigmoid, ReLU
- activation function in output layer = Leaky ReLU
- batch size = 32
- optimizer = Adam
- learning rate (LR) = 0.001
- number of epochs = 350

Confusion Matrix



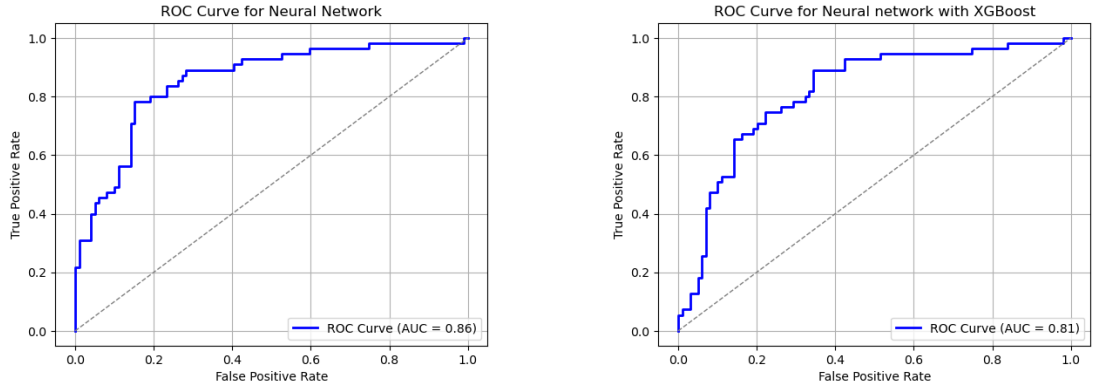
(a) Neural Network

(b) Neural network with XGBoost

Figure 3.2: Side-by-side comparison of the confusion matrix for a neural network and a neural network with XGBoost. The accuracies on the test data are 82.5% and 77.3% respectively.

In Figure 3.2, we see the confusion matrices of our NN without and with gradient boosting. When going from the standard NN to NN with XGBoost, we notice that the false postives go from 12 to 14, but the false negatives go from 15 to 21. The number of true positives goes from 43 to 34, and true negatives from 84 to 84. Just looking at falsely predicted labels, this number goes from 27 to 35 when using XGBoost. This means the NN alone is more accurate at classifyinf. The accuracy on the test data decreases from 82.5% to 77.3%.

ROC AUC



(a) Neural Network

(b) Neural network with XGBoost

Figure 3.3: Side-by-side comparison of the ROC AUC curves for a neural network and a neural network with XGBoost. The accuracies on the test data are 82.5% and 77.3% respectively.

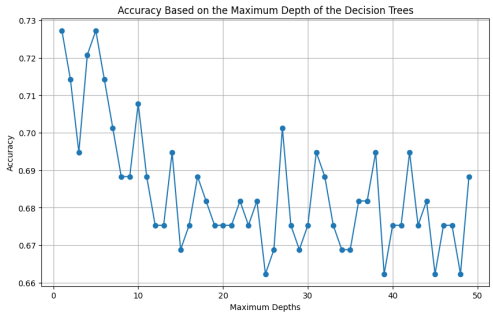
In Figure 3.3 we see that the AUC value is 0.86 for the NN, but decreases to 0.81 with XGBoost. This is not a large change, but it shows that the relationship of true positives to false positives worsen with XGBoost.

3.3 Decision Trees

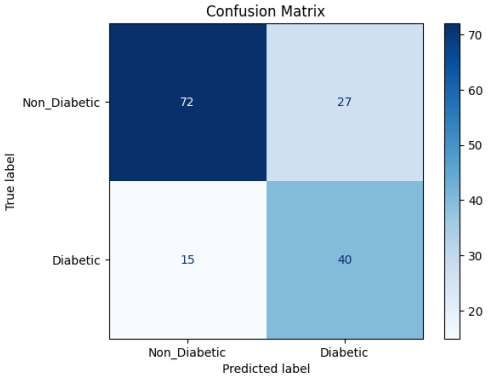
Our model showed the highest accuracy when choosing the following hyperparameters:

- Maximum Depth = 5

The highest accuracy achieved was 72.73%.



(a) Accuracies vs Maximum Depth



(b) Confusion Matrix

Figure 3.4: (a) shows a graph over the accuracy of decision trees on the test data using different maximum depth on the trees. The x-axis is the maximum depths of the trees, and the y-axis is the accuracy. The highest accuracy achieved was reached when the depth is 5.
(b) shows the confusion matrix for this model.

Figure 3.4 shows the graph over different decision tree models and the confusion matrix for the model with the highest accuracy. The maximum depth which achieved the highest accuracy is 5. The accuracy varied, but generally declined as the maximum depth increased. This model classifies 74 TN, 31 TP, 24 FN and 25 FP.

3.4 Random Forest

3.4.1 Scikit-learn Random Forest

Our model showed the highest accuracy when choosing the following hyperparameters:

- Maximum Depth = None,
- Number of trees = 50

The highest accuracy achieved was 79.87%.

Number of Trees

Shown in table 3.2 the highest accuracy achieved using the random forest function was an accuracy of 79.9% and was achieved with 50 trees. The accuracy remained about the same at $\approx$ 76% for 25, 75 and 100 trees, and was slightly higher at $\approx$ 77% for 10 trees.

Maximum Depth

Number of trees	Accuracy
10	77.27%
25	75.97%
50	79.87%
75	75.97%
100	76.62%
125	75.97%
150	76.62%

Table 3.2: A table showing the accuracy for different number of trees in the random forest. The maximum depth of the trees are set to None. The highest accuracy achieved was 79.87% with 50 trees in the forest, highlighted in blue.

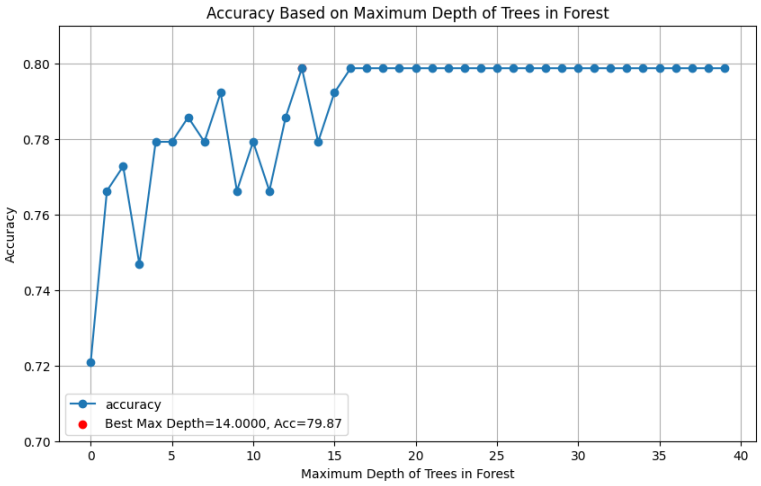


Figure 3.5: The accuracy from Random Forest using different max depths. The number of trees are 50.

From figure 3.5 the highest accuracy achieved when tuning the maximum depth of the trees was 79.87%, with a maximum depth of 14 or ≥ 17. For a maximum depth of $<$ 14, the accuracy was lower, and fluctuated between $\approx$ 72% and $\approx$ 79%.

3.4.2 XG Boost Random Forest

Our model showed the highest accuracy when choosing the following hyperparameters:

- Maximum Depth = None,
- Number of trees = 21,
- eta = 1

Learning Rate

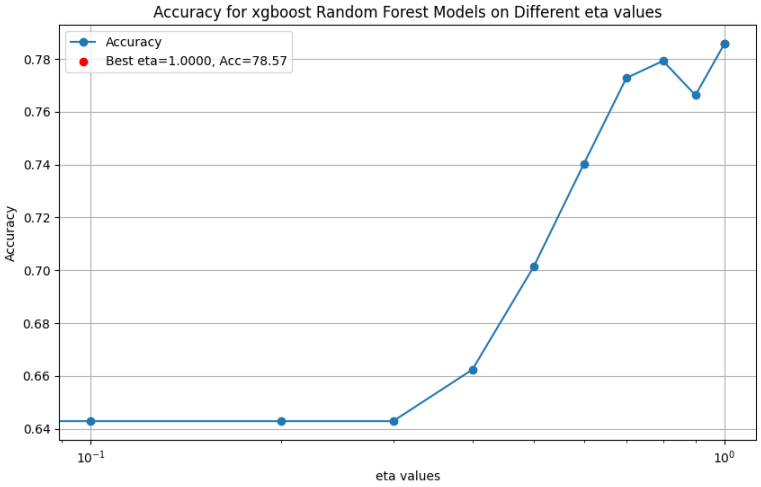
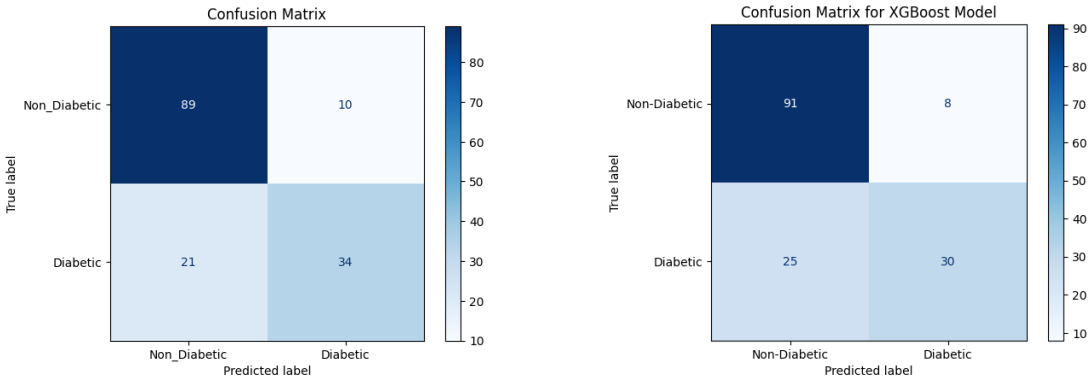


Figure 3.6: The graph showing the relationship between the learning rate (x-axis) and the accuracy achieved by the model (y-axis). The highest accuracy achieved is 78.57%, with a learning rate of 1. The number of trees are equal to 21 and the maximum depth is equal to None.

3.4.3 Comparison

Confusion Matrix



(a) Confusion Matrix for Scikit-learn

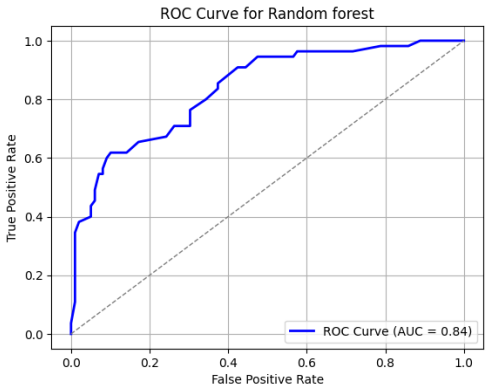
(b) Confusion Matrix for XGBoost

Figure 3.7: The confusion matrices for a Scikit-learn random forest model and an XGBoost random forest model. (a) has 50 random trees and a no maximum depth. (b) has 21 trees in the forest, a no maximum depth and an eta of 1.

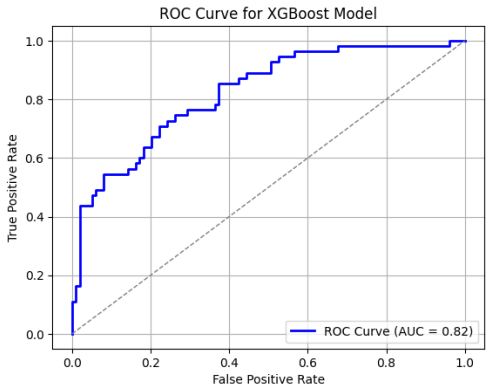
Figure 3.7 (a) shows the confusion matrix for a scikit-learn random forest model with

50 random trees and no maximum depth. This model has 89 TN, 34 TP, 21 FN and 10 FP. Figure 3.7 (b) shows the confusion matrix for an XGBoost random forest model with 21 trees, no maximum depth and an eta of 1. It has 91 TN, 30 TP, 25 FP and 8 FN.

ROC AUC



(a) The ROC AUC for a Scikit-learn random forest.



(b) The ROC AUC for an XGBoost random forest.

Figure 3.8: The ROC AUC for a Scikit-learn and an XGBoost random forest model. (a) has 50 trees and no maximum depth. (b) has 21 trees, a maximum depth of None and an eta of 1.

Chapter 4

Discussion

4.1 Dataset

It is important to acknowledge that the Pima Indians are not representative of the general population. Demographic-specific factors can significantly influence the fitting of a model.

The Pima Indian dataset is well-known for its challenges in classification tasks which raises the question what makes this dataset so challenging. First, the dataset is relatively small, containing only 768 cases. With limited data, it becomes more difficult for models to reliably identify the most important features for accurate predictions and to generalize effectively. Additionally, two-thirds of the dataset contains missing values. While in preprocessing we addressed these gaps by imputing missing values, we can not generate new data points. With preprocessing we primarily ensure that the model can function without being disrupted by incomplete records.

Despite these difficulties, working with such a challenging dataset also offers advantages for model engineering. A simpler dataset might yield higher accuracies even with poorly tuned models, potentially giving a false sense of confidence in their effectiveness. The complexities of the Pima dataset compelled us to focus on optimizing our models. This approach ensures the models are better prepared to handle real-world scenarios. For instance, we observed a compelling 5.2 percentage points increase from initially 77.3% 82.5% in accuracy by fine-tuning the NN.

All this considered, the Pima Indian dataset is well-suited for studying model engineering and optimization. However, it is not ideal for designing models intended for real-world applications, as the Pima Indian population's demographic-specific characteristics limit the dataset's generalizability.

4.2 Models

4.2.1 Logistic Regression

Our logistic regression model achieved an accuracy of 77.36% after fine-tuning. The hyperparameters were perfected through trial-and error since there is no analytical expression to find them. With thoroughly fine-tuning the hyperparameters we were able to outperform the Scikit-learn logistic regression by 2 percentage points. The found accuracy, while modest, is very close to the 77% reported in the literature. This demonstrates that the model is not very effective given the complexity of the dataset and the simplicity of the logistic regression algorithm.

The high rate of false negatives of 20% (31 cases out of 154 test cases) found in the confusion matrix is particularly concerning in a clinical context. False negatives, where a diabetic patient is misclassified as non-diabetic, could lead to missed diagnoses and a lack of timely intervention.

4.2.2 Neural Network

For a Neural Network, there is a lot of fine-tuning of parameters involved. These can affect the results drastically. We initialized the parameters we thought would give acceptable results. See Table 3.1. This gave an accuracy of 77.3%. We wanted to see if it was possible to improve this.

In a process of trial and error, we changed certain hyperparameters, see Table 3.1. The batch size, optimizer and learning rate remained the same as changing these did not improve overall accuracy. The number of epochs was increased to 350. Increasing it further did not change the accuracy on the test data, it only increased computational load. We increased the number of layers to three, finding that a more complex model increased accuracy. An even more complex model was not as well-performing. We ended up with an accuracy of 82.5% on the test data, which is higher than the literature value we found.

In the implementation of a NN with gradient boosting, we kept the basic architecture and hyperparameters of our NN, to make the results comparable. When performing gradient boosting on the neural network predictions, the final accuracy of the test data decreased. See Figure 3.2 and Figure 3.3. This result was surprising as we had hoped the accuracies would increase.

With an accuracy of 81.2% for the NN, XGBoost yielded an accuracy of 78.6%. After tuning, the accuracy for the NN increased even further to 82.5%, and the accuracy of XGBoost decreased further to 77.3%.

The reason for the accuracy for XGBoost being lower than for a better trained NN, could lie in the model being overfitted to the training data. A NN can have diminished informative features that are important for XGBoost's performance. You could also accidentally amplify noise in the dataset, since it is small. Any noisy feature can affect the results greatly.

XGBoost works by learning from previous mistakes, and by creating new models which corrects those mistakes. It might not work to combine a NN and XGBoost because the NN already works reasonably well. There is less to improve with the NN model, leaving the XGBoost functionally useless.

Due to time constraints, we were unable to further explore the optimal accuracy for XGBoost, which could be reached with a less trained NN. This could be an interesting topic for future investigation.

4.2.3 Decision trees

Constricting the maximum depth of the tree to 5 trees, as opposed to having no maximum depth, increased the accuracy from 68.2% to 72.7%. As discussed in Section 2.3.4, trees can overfit if they get too large. Constricting the maximum depth can thus increase accuracy, as supported by our results. An accuracy of 72.7% is still not useful for real-life diagnostics, and we wanted to see if we could improve it by implementing random forest.

4.2.4 Random Forest

The highest accuracy achieved using a random forest model came from using a Scikit-learn model with 50 trees and no maximum depth. The accuracy was at 79.9%, noticeable higher than using just a single decision tree, and comparable to the literature values we found. This is also higher than the highest accuracy achieved when using XGBoost which was 78.6%.

Maximum depth

Constricting maximum depth for either of the random forest models was less effective than when using a single decision tree. In fact, the accuracy did not decrease as the maximum depth increased, contrary to the results from a single decision tree. For the XGBoost model there was no maximum depth that gave the best results. The accuracy was the highest when using no maximum depth. For scikit-learn, having a very low maximum depth of 10 or less decreased the accuracy noticeably.

The finding that too few trees decreased the accuracy is most likely a result of not achieving enough complexity for the task. The reason for us not needing to constrict

maximum depth might come from the fact that random forests are less prone to overfitting than decision trees. The added complexity of larger trees in a forest can thus lead to a higher accuracy, without running the risk of overfitting.

Number of Trees
The best number of trees was found through trial and error for both Sckit-learn and XGBoost. It could be natural to think that increasing the number of trees in the forest would automatically increase the accuracy of the model. In our experience this is not necessarily true. From our results, the Scikit-learn random forest model which achieved the highest accuracy of 79.9% had 50 trees in the forest. With Scikit-learn as well as with XGBoost, it is most likely that more trees made the random forest overfit, and fewer trees made them not complex enough. We assume that given more trees the models stopped increasing in accuracy and started overfitting to the training set instead.

eta
The highest accuracy achieved with XGBoost was 78.57% with an eta of 1, which is the threshold of the range of eta. This suggests that the model was not prone to overfitting with a higher eta, but rather needed the added advantage of better learning from a higher eta.

As seen from the confusion matrix figure 3.7, the most accurate Scikit-learn and XGBoost random forest models were more prone to type II errors than type I. The Scikit-learn model had 10 FP and 21 FN, with a AUC of 0.84. This means that the model has good discriminatory power, effectively distinguishing between the two classes and performing significantly better than random guessing.

4.2.5 Comparison between the models

To make our models more applicable to real-life situations we would need to apply and train our models to different datasets. This is especially important for our well-performing models as they are more likely to be overfitted to the Pima Indian diabetes dataset.

Logistic regression seemed to not give any improved performance from literature values. This is unsurprising, as it still is a very simple model. In the case of classification of this dataset, the model was not a good choice despite being better in other cases.

A NN gave the highest accuracy on its own, with an accuracy of 82.5%. It did need a lot of fine-tuning, but we find that our model did not have too high of a computational demand in this case. This is mostly due to using a small dataset. If we want to use any larger dataset, we would need to fine-tune again.

As we wanted to compare these methods with random forests, we saw that it improved compared to the logistic regression's accuracy. Therefore, it is a bit promising to look into this further. However, we state once again, we do not have information about its generalizability for other datasets.

Putting a Hat on a Hat

Decision trees were improved by the ensemble method random forest, and the highest accuracy increased from 68.2% to 79.9% with Scikit-learn's random forest method. Trying to improve a NN with gradient boosting, however, made the model worse. Ensemble methods like random forest and gradient boosting are usually performed on weak learners like decision trees, and not strong learners like NNs. Gradient boosting works by trying to improve previous models, but there is little to improve in a good NN model. This might be one of the reasons our boosted NN was not improved from our NN. Combining weak learners with gradient boosting works, because there is much to be improved.

Chapter 5

Conclusion

In this paper, we have tried to use different models made from ML algorithms to diagnose Diabetes Type 2 based on the Pima Indian Diabetes dataset. These results have been compared to each other and to values found in the literature.

Further, the best of our models was a feed forward neural network, which achieved an accuracy of 82.5%. We tried improving the neural network by combining it with gradient boosting. Our attempt was unsuccessful, and ended in a lower accuracy. We are not dissuaded by this defeat, and still have high hopes for a future combination of these methods.

For the future, we would want to improve a model using both a NN and XGBoost further. We believe there could be a tradeoff giving us an even higher accuracy on the test data, and saving time from fine-tuning and the computational time needed for a NN for larger datasets.

All our models are prone to type II errors, meaning that anyone using them for actual diagnostic should be wary of false negatives. A false negative diagnosis can lead to devastating health complications. We have found that tuning our models has been crucial for improving the accuracy.

Bibliography

Brownlee, J. (2021). "How to Develop Random Forest Ensembles With XGBoost". In: *Machine Learn Mastery*. URL: https://machinelearningmastery.com/random-forest-ensembles-with-xgboost/.

Chang, V. et al. (Mar. 2022). "Pima Indians diabetes mellitus classification based on machine learning (ML) algorithms". In: *Neural Computing and Applications* 35.22, pp. 16157–16173. ISSN: 1433-3058. DOI: 10.1007/s00521-022-07049-z. URL: http://dx.doi.org/10.1007/s00521-022-07049-z.

Chen, T. and C. Guestrin (2016). "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

Grønbekk, J., S. Gaupp, and M. P. Walderhaug (2024). *Building a Neural Network*. https://github.com/sphia-g/ML24UiO/blob/main/Project2/Report/Report_project2.pdf. Report from project 2, University of Oslo.

Health Economics, J. of and O. Research (2023). "Global Increase in Diabetes Prevalence Imposes a Substantial Health and Economic Burden". In: *Journal of Health Economics and Outcomes Research*. Accessed: 2024-12-12. URL: https://jheor.org/post/1265-global-increase-in-diabetes-prevalence-imposes-a-substantial-health-and-economic-burden.

Hjorth-Jensen, M. (2024). *FYS-STK4155 – Applied Data Analysis and Machine Learning*. https://github.com/CompPhysics/MachineLearning/tree/master/doc. Lecture Notes, University of Oslo.

Karatsiolis, S. and C. N. Schizas (2012). "Region based Support Vector Machine algorithm for medical diagnosis on Pima Indian Diabetes dataset". In: *2012 IEEE 12th International Conference on Bioinformatics Bioengineering (BIBE)*, pp. 139–144. DOI: 10.1109/BIBE.2012.6399663.

Lakhwani, K. et al. (2020). "Prediction of the Onset of Diabetes Using Artificial Neural Network and Pima Indians Diabetes Dataset". In: *2020 5th IEEE International*

Conference on Recent Advances and Innovations in Engineering (ICRAIE), pp. 1–6. DOI: 10.1109/ICRAIE51050.2020.9358308.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Sankar Ganesh, P. V. and P. Sripriya (2020). "A Comparative Review of Prediction Methods for Pima Indians Diabetes Dataset". In: *Computational Vision and Bio-Inspired Computing*. Springer International Publishing, pp. 735–750. ISBN: 9783030372187. DOI: 10.1007/978-3-030-37218-7_83. URL: http://dx.doi.org/10.1007/978-3-030-37218-7_83.

World Health Organization (2024). *Diabetes Fact Sheet*. Accessed: 2024-12-11. URL: https://www.who.int/news-room/fact-sheets/detail/diabetes.

Appendix A

A.1 GitHub Repository

We have all our source code and test runs in our repository on GitHub at this link:

https://github.com/sphia-g/ML24UiO/tree/main/Project3