HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

CAPSTONE PROJECT

Machine Learning & Data Mining - IT3191E

Obesity Risk Prediction

Nguyen Trong Huy - 20210451
Trinh Giang Nam - 20215229
Nguyen Chinh Minh - 20215224

Supervisor:	Associate Professor Than Quang Khoat	_____________
		Signature
Department:	Computer Science	
School:	School of Information and Communications Technology	

HANOI, 06/2024

ACKNOWLEDGMENT

First and foremost, we express our sincere thanks to our project supervisor, Prof. Khoat Quang Than, whose guidance, expertise, and patience were instrumental in steering this project towards its completion.

Special appreciation goes to our classmates and the teaching assistant whose advice helped us a lot in writing this report. Their willingness to contribute their time and thoughts added depth and authenticity to our work.

Lastly, we extend our gratitude to each other, the members of Group 2. This project was a collaborative effort that required dedication, compromise and teamwork. We have grown individually and collectively through this experience, gaining not just knowledge but also friendships that we treasure.

This project report is not only a reflection of our hard work but also a testament to the support and guidance we received from all those mentioned above. Thank you for making this journey memorable and our project a success.

ABSTRACT

The issue of predicting obesity levels is currently of great concerns to many people. Not only doctors, who want to provide the best diagnoses for their patients, but also individuals at home who want a method to assess their current obesity health status. With the advent of artificial intelligence and machine learning, computers can also assist humans in these predictive tasks. Therefore, we chose the topic of classifying obesity levels for this problem with the hope of finding a highly accurate model. In this problem, we conducted research on traditional machine learning models: k-NN, decision tree, random forest, SVM, and multilayer perceptrons. Additionally, we combined some feature selection methods to reduce computational complexity, such as Sequential Feature Selection or methods using Decision Tree and Random Forest. Besides that, we also applied ensemble methods: Voting Classifier, Adaptive Boosting, and Gradient Boosting. From the experimental results, we found that Gradient Boosting provided the best accuracy, reaching up to 90.56%, which is a competitive result that can apply on the real-world application and software.

Student

(Group 2 - Machine Learning and Data Mining)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	**Definition**
ANN	Artificial Neural Network
kNN	k - Nearest Neighbours
SFS	Sequential Feature Selection
SVM	Support Vector Machine

CHAPTER 1. INTRODUCTION

1.1 Problem Statement

Obesity is a major public health concern globally, associated with numerous health conditions including diabetes, cardiovascular diseases, and certain cancers. It is a complex condition influenced by various genetic, environmental, and behavioral factors. Accurately predicting obesity can help in early intervention and the implementation of preventive measures. Traditional methods of predicting obesity often rely on basic statistical models and demographic data, which may not capture the full complexity of the factors involved.

With the advent of machine learning (ML), there is an opportunity to leverage advanced algorithms to improve the accuracy of obesity prediction. ML models can analyze large and complex datasets, uncovering patterns and interactions among variables that might be missed by traditional methods. Despite the potential benefits, developing effective ML models for obesity prediction presents several challenges. These include handling diverse and often imbalanced datasets, selecting relevant features, and ensuring that the models generalize well to different populations. Additionally, ML models must be interpretable to ensure that their predictions can be understood and trusted by healthcare professionals.

This report addresses these challenges by developing and evaluating various ML models for obesity prediction, aiming to enhance prediction accuracy and model interpretability.

1.2 Capstone Project Objectives

The primary objective of this research is to develop an effective machine learning-based system for predicting obesity. The focus is on leveraging state-of-the-art ML techniques to analyze and interpret complex datasets, thereby improving the prediction accuracy compared to traditional methods. The conceptual framework involves the following key components:

1. Data Collection and Preprocessing: Gathering a comprehensive dataset that includes demographic, behavioral, and genetic factors associated with obesity. The data will be cleaned and preprocessed to ensure quality and relevance.

2. Feature Selection and Engineering: Identifying and engineering relevant features that significantly contribute to obesity prediction. This step involves domain knowledge and exploratory data analysis to enhance the model's performance.

3. Model Development and Training: Developing various ML models, including but not limited to logistic regression, decision trees, random forests, support vector machines, and neural networks. Each model will be trained and validated using the preprocessed dataset.
4. Model Evaluation and Interpretation: Evaluating the models based on their accuracy, precision, recall, and other relevant metrics. Additionally, ensuring that the models are interpretable to provide actionable insights for healthcare professionals.
5. Implementation and Validation: Implementing the best-performing model and validating its performance on an independent dataset. This step ensures that the model generalizes well and can be reliably used in real-world scenarios.

The research aims to contribute to the field by providing a robust, interpretable, and accurate obesity prediction model, thereby aiding in early intervention and better management of obesity-related health risks.

1.3 Organization of Report

There are five main chapters in our report:

CHAPTER 1. INTRODUCTION represents the overview of the problem and proposes the objectives and sketch the outlines of solution.

CHAPTER 2. LITERATURE REVIEW provides the fundamental concepts and foundation theory including the selected datasets, pre-trained large language model, and metrics.

CHAPTER 3. METHODOLOGY includes the main scenario applied to solve problem. For each case, we propose the full pipeline to process the data, train the model and evaluate.

CHAPTER 4. EXPERIMENTAL RESULT shows the result for our model, comparison among all the cases. This also compares the results to the current researches.

CHAPTER 5. CONCLUSIONS sums up all the relevant results and proposes the future works on this project.

CHAPTER 2. LITERATURE REVIEW

2.1 Baseline Classifiers

2.1.1 Decision Tree

A decision tree [1] is a non-parametric supervised learning algorithm for classification and regression tasks. It has a hierarchical tree structure consisting of a root node, branches, internal nodes, and leaf nodes. Decision trees are used for classification and regression tasks, providing easy-to-understand models.

This algorithmic model utilizes conditional control statements and is non parametric, supervised learning, useful for both classification and regression tasks. The tree structure is comprised of a root node, branches, internal nodes, and leaf nodes, forming a hierarchical, tree-like structure. The name itself suggests that it uses a flowchart like a tree structure to show the predictions that result from a series of feature-based splits. It starts with a root node and ends with a decision made by leaves.

a, Decision Tree Terminology

- Root Node: The initial node at the beginning of a decision tree, where the entire population or dataset starts dividing based on various features or conditions.
- Decision Nodes: Nodes resulting from the splitting of root nodes are known as decision nodes. These nodes represent intermediate decisions or conditions within the tree.
- Leaf Nodes: Nodes where further splitting is not possible, often indicating the final classification or outcome. Leaf nodes are also referred to as terminal nodes.
- Sub-Tree: Similar to a subsection of a graph being called a sub-graph, a subsection of a decision tree is referred to as a sub-tree. It represents a specific portion of the decision tree.
- Pruning: The process of removing or cutting down specific nodes in a decision tree to prevent overfitting and simplify the model.
- Branch / Sub-Tree: A subsection of the entire decision tree is referred to as a branch or sub-tree. It represents a specific path of decisions and outcomes within the tree.

Parent and Child Node: In a decision tree, a node that is divided into sub-nodes is known as a parent node, and the sub-nodes emerging from it are referred to as child nodes. The parent node represents a decision or condition, while the child nodes represent the potential outcomes or further decisions based on that condition.

Decision trees are upside down which means the root is at the top and then this root is split into various several nodes. Decision trees are nothing but a bunch of if-else statements in layman terms. It checks if the condition is true and if it is then it goes to the next node attached to that decision.

b, ID3 Algorithm

The ID3 algorithm is a popular decision tree algorithm used in machine learning. It aims to build a decision tree by iteratively selecting the best attribute to split the data based on information gain. Each node represents a test on an attribute, and each branch represents a possible outcome of the test. The leaf nodes of the tree represent the final classifications. In this section, we will explain how to use the ID3 algorithm to build a decision tree to predict the output in detail.

Algorithm 1: ID3 Algorithm

```
Input: Training data $D$, Target attribute $C$, Attribute set $A$
Output: Decision tree
Initialize: Create an empty tree
while $D$ is not empty and $A$ is not empty do
    Calculate information gain $(IG(D, A))$ for each attribute $a \in A$
    Find the attribute $a_g$ with the highest information gain
    Add a new decision node labeled $a_g$ to the tree
    for value $v$ of attribute $a_g$ do
        $D_v \leftarrow$ subset of $D$ where $a = v$
        Recursively call ID3 on $D_v$, $C$, $A \setminus \{a_g\}$ (remove $a_g$ from attributes)
        Add the resulting subtree a child of the current node labeled with $v$
        if $D$ is empty then
            Add a leaf node with the most frequent class in remaining
              examples
        else
            if all examples in $D$ have the same class value $c$ then
                Add a leaf node labeled $c$
```

Information gain of an attribute in S measures the reduction of entropy if we divide S into subsets according to that attribute. Information gain of attribute A in S is defined as:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

In which, $\text{Values}(A)$ is the set of all values of A, and $S_v = \{\mathbf{x} | \mathbf{x} \in S, \mathbf{x}_a = v\}$.

Meaning of **Gain(S,A)**: the average amount of information is lost when dividing S according to A.

And Entropy measures the impurity or inhomogeneity of a set. Entropy of a set S with c classes can be defined as:

$$\text{Entropy}(S) = -\sum_{i=1}^{x} p_i \log_2 p_i$$

2.1.2 k-NN

kNN [2], or the k-nearest neighbor algorithm, is a machine learning algorithm that uses proximity to compare one data point with a set of data it was trained on and has memorized to make predictions. This instance-based learning affords kNN the "lazy learning" denomination and enables the algorithm to perform classification or regression problems. kNN works off the assumption that similar points can be found near one another.

The kNN algorithm works as a supervised learning algorithm, meaning it is fed training datasets it memorizes. It relies on this labeled input data to learn a function that produces an appropriate output when given new unlabeled data. This enables the algorithm to solve classification or regression problems. While kNN's computation occurs during a query and not during a training phase, it has important data storage requirements and is therefore heavily reliant on memory.

For classification problems, the KNN algorithm will assign a class label based on a majority, meaning that it will use the label that is most frequently present around a given data point. In other words, the output of a classification problem is the mode of the nearest neighbors.

a, Distance metrics

The distance measure plays a very important role in KNN. It indicates how we assume/suppose the distribution of our data and can only be determined once. It does not change in all prediction later. Some geometric distances that we might explore while using kNN to our problem:

Minkowski:

$$d(x,z) = \left(\sum_{i=1}^{n} |x_i - z_i|^p\right)^{1/p}$$

Manhattan:

$$d(x,z) = \sum_{i=1}^{n} |x_i - z_i|$$

Euclid:

$$d(x,z) = \sqrt{\sum_{i=1}^{n} (x_i - z_i)^2}$$

b, k value

To choose the best k value — the number of nearest neighbors considered — we can experiment with a few values to find the k value that generates the most accurate predictions with the fewest number of errors. Note that: low k values make predictions unstable and high k values are noisy.

2.1.3 SVM

Support vector machine (SVM), first time proposed in [3], as a learning machine has shown a good learning ability and generalization ability in classification, regression and forecasting. SVM shows its significant advantages on both separable problems and non-separable problems. Separable problems include linear separable problems and non-linear separable problems. Thus, this section will present the theory of SVM from three cases.

a, Linearly Separable

SVM extends the two-dimensional linear separable problem to multidimensional, and aims to seed the optimal classification surface, also called as optimal hyperplane. The optimal hyperplane can be defined as:

$$w^T x + b = 0$$

in which w refers to the weight vector and b is the threshold. Thus, the relation between x_i and $f(x_i)$ can be defined as:

$$f(x) = w^T x + b$$

Seeding the optimal hyperplane is equivalent to maximal the distance between the closest vectors to the hyperplane. Define the Euclidean distance between the nearest points and the hyperplane $f(x)$ as:

$$r = |\frac{f(x)}{||w||}|$$

where $f(x)$ refers to functional margin.
The assumption was accompanied with a constraint condition $y_i(w^T x_i + b) \geq 1, i = 1, 2, ..., n$, which implies that all training data are on the two hyperplane or behind them and the training data on the hyperplane are called support vectors (SVs). Thus, the margin between the parallel bounding planes d can be defined as: $d = 2r = \frac{2}{||w||}$. Thus the objective function can be presented as:

$$\max \frac{2}{||w||} \qquad s.t. y_i(w^T x_i + b) \geq 1, i = 1, 2, ..., n$$

For ease of calculation, translate equation into:

$$\min \frac{1}{2}||w||^2 \qquad s.t. y_i(w^T x_i + b) \geq 1, i = 1, 2, ..., n$$

The solution to this optimization problem is given by the saddle point of the Lagrange functional:

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i [y_i(w^T x_i + b)]$$

where α_i are Lagrange multipliers, subject to $\alpha_i \geq 0$, and equation has to be minimized with respect to w, b and maximized with respect to α_i. The objective function has been translated as:

$$p^* = \min_{w,b} \theta(w) = \min_{w,b} \max_{\alpha_i \geq 0} L(w, b, \alpha)$$

The classical Lagrange duality enables the primal problem to be transformed into its dual problem, when satisfying the Karush-Kuhn-Tucker (KKT) condition. In hence, as for equation (7), when it fulfills the KKT condition of $\sum_{i=1}^{n} \alpha_i [y_i(w^T x_i + b) - 1]$, equation is equal to its dual function:

$$d^* = \max_{\alpha_i \geq 0} \min_{w,b} L(w, b, \alpha)$$

It is the KKT condition that determines only the SVs work on the extremum of the objective function. Then the minimum with respect to w and b of the Lagrange, L, is given by:

$$\frac{\partial L(w,b,\alpha)}{\partial b} = 0 \Rightarrow w = \sum_{i=1}^{n} \alpha_i y_i x_i$$

$$\frac{\partial L(w,b,\alpha)}{\partial b} = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0$$

Hence the dual problem d^* can be changed into:

$$\theta(\alpha) = L(w,b,\alpha) = \sum_{i=1}^{n} \alpha_i + \frac{1}{2} \sum_{i=1,j=1}^{n} \alpha_i \alpha_j y_i y_j (x_i)^T x_j$$

which is a two convex optimization problems. Supposing that α_i is the result of equations (10), Xr is a support vector belonging to the positive class, while Xs belongs to the negative class. Then the optimal solution w and b can be expressed as follows:

$$\overline{w} = \sum_{i=1}^{n} \overline{\alpha_i} y_i x_i, \overline{b} = -\frac{1}{2}\overline{w}|X_r + X_s$$

Hence the final decision function is:

$$f(x) = sgn[\sum_{SVs} \overline{\alpha_i} y_i (x_i)^T x + \overline{b}]$$

b, Non-linearly Separable

In addition to the linearly separable problems, there are much more non-linearly cases in real life when there is no line can classify the two classes well, only curves. SVM shows its superiorities for non-linearly problems, which takes the way of mapping the input vectors in low dimension feature space to a high dimension feature space to hunt for a linear optimal separating hyperplane. And it has been demonstrated that it did not account for large storage and computational expense. The following is an example of mapping process. Define of the curve in two-dimensional space as:

$$g(x) = C_0 + C_1 x + C_2 x^2$$

Map it into a four-dimensional space as follows:

$$y = \begin{bmatrix} y_3 \\ y_2 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}, a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} C_0 \\ C_1 \\ C_2 \end{bmatrix}$$

Then, the primal function is translated into:

$$g(x) \Rightarrow f(x) =< a, y > ay$$

It's a linear function, which means the primal non-linearly problem has been converted to a linearly separable problem. SVM employs the empathy, mapping the w and x into high dimensional space, then gets new w' and x'.

$$f(x) =< w, x > +b \Rightarrow g(x') =< w', x' > +b$$

In hence, new decision function can be defined as follows:

$$g(x') = \sum_{i=1}^{n} \alpha_i y_i < x_i', x_i > +b$$

where dot product is core operation. According to the existence theorem of kernel function: for any sample set K, there is a map of F, and in this map, f(k) (in the high dimensional space) is a linear separable. Based on this theorem, SVM employs the kernel function during mapping process, and realizes the dot product in low dimensional space, which is supposed to be done in high dimensional space. Decision function with kernel function is shown as follows:

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i K < x_i, x > +b$$

Different kernel functions can be choose to construct different types SVM, the following are a few commonly used kernel functions:

1. Polynomial $K(x, x_i) = (x \cdot x_i)^q \qquad q = 1, 2, ...$
2. Gaussian Radial Basis Function $K(x, x_i) = exp(-\frac{(x-x_i)^2}{2\sigma^2})$

c, Non-separable

Last circumstance, some points belong to positive class may be classified into negative class. Another way to say, there is no hyperplane can separated the points of different categories accurately. Usually these error points are regarded as noise which can be ignored. However, machines can't deal with the error points as human do. Thus, Cortes introduced slack variables $\xi_i \geq 0$ (a measure of the misclassification error) and punishment coefficient C (the emphasis on the outlier) to offset the effect of noise. And the constraint $y_i(w^T x_i + b) \geq 1, i = 1, 2, ..., n$ is modified to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i, i = 1, 2, ..., n$$

C is a given value and subjects to above constraint. Too small ξ may lead to over learning, while too large may lead to owe learning. The equations are changed into:

$$\Phi(x,\xi) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i$$

$$L(w,b,\alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n}\alpha_i[y_i(w^T x_i + b) - (1-\xi)]$$

While after derivation, the decision function for this case is still the same. In summary, SVM is a powerful classifier, which is suitable for any case of classification with the same decision function, high classification accuracy and small computation.

2.1.4 Multilayer Perceptrons

Multilayer Perceptrons (MLP) are a class of feedforward artificial neural networks (ANN) that consist of multiple layers of nodes, typically organized into an input layer, one or more hidden layers, and an output layer. Each node (or neuron) in one layer is connected to every node in the subsequent layer, making MLPs fully connected networks. MLPs are capable of modeling complex non-linear relationships and are widely used for both classification and regression tasks.

An MLP consists of an input layer with n neurons, each representing an input feature, one or more hidden layers, and an output layer. The connections between the nodes are weighted, and the network learns by adjusting these weights through a process called backpropagation. The activation function introduces non-linearity into the network, enabling it to capture complex patterns in the data.

The output of a neuron j in a hidden layer can be expressed as:

$$h_j = \phi\left(\sum_{i=1}^{n} w_{ij}x_i + b_j\right)$$

Where:

- ϕ is the activation function (For this problem, we use tanh),
- w_{ij} is the weight associated with the connection between input i and neuron j,
- x_i is the input feature i,
- b_j is the bias term for neuron j.

The formula for tanh activation function is defined as:

$$\tanh(x) = f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The output layer then combines the outputs from the last hidden layer, typically using a different activation function depending on the task (e.g., softmax for classification, linear for regression).

a, Training with Backpropagation

The training process of an MLP involves adjusting the weights to minimize a loss function, typically the mean squared error (MSE) for regression tasks or cross-entropy loss for classification tasks. In this multi-class classification problem, we use sparse categorical crossentropy. This is done using the backpropagation algorithm, which consists of two main steps: forward pass and backward pass.

In the forward pass, the input data is propagated through the network to generate predictions. The loss function L is then computed based on the difference between the predicted and actual values:

$$L = -\sum_{i=1}^{n} t_i \log(p_i)$$

In which, t_i is the truth label and p_i is the softmax probability of the i^{th} class

In the backward pass, the gradients of the loss function with respect to the weights are calculated using the chain rule. These gradients are then used to update the weights through Adam optimization [4].

MLPs are versatile and can be applied to a wide range of tasks, including image recognition, natural language processing, and financial forecasting. The performance of an MLP depends on various factors, including the number of layers, the number of neurons in each layer, the choice of activation function, and the optimization algorithm. Proper tuning of hyperparameters and regularization techniques such as dropout are essential to prevent overfitting and ensure good generalization.

b, Regularization

There are two popular methods for regularization of multilayer perceptrons:

One of them, early stopping is a regularization technique used in machine learning, to prevent overfitting during model training. It works by monitoring the model's performance on a validation set during training and stopping the process when performance begins to degrade. By halting training before overfitting occurs, early stopping helps improve the model's ability to generalize to unseen data.

Implementation involves dividing the data into training, validation, and test sets. The model is trained on the training set, and its performance is evaluated on the validation set after each epoch. If the performance fails to improve for a predefined number of epochs (known as patience), training is stopped, and the model parameters from the best epoch are retained.

Early stopping is widely used in various machine learning tasks, including deep learning, and continues to be an essential tool for improving model performance.

The other method for regularization is using dropout layer. Dropout is a regularization technique used in neural networks to prevent overfitting. During training, dropout randomly "drops out" a fraction of the neurons in the network by setting their output to zero with a certain probability. This forces the network to learn more robust features, as it cannot rely on the presence of any particular neuron. By doing so, dropout helps to reduce the co-adaptation of neurons, leading to improved generalization on unseen data. When the network is used for inference, all neurons are used, but their outputs are scaled to account for the dropout rate during training, ensuring that the overall prediction remains consistent.

2.2 Ensemble Classifiers

2.2.1 Random Forest

When building a decision tree, it is important to consider the depth. If the depth is set arbitrarily, the tree may classify everything correctly in the training set, but perform poorly on the test set, indicating overfitting. On the other hand, the Random Forests [5] algorithm addresses this issue by using multiple decision trees with random factors. Each decision tree randomly selects data and attributes to build a model. While individual decision trees may not predict well on their own, the Random Forests algorithm combines the information from all the trees, resulting in a model with good prediction results and low bias and variance. Random Forest is a popular supervised learning algorithm that can be used for both Classification and Regression problems in Machine Learning, based on the concept of ensemble learning.

Suppose the training data set D consists of n data samples and each data has d attributes. We learn K decision trees as follows:

[1]https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/

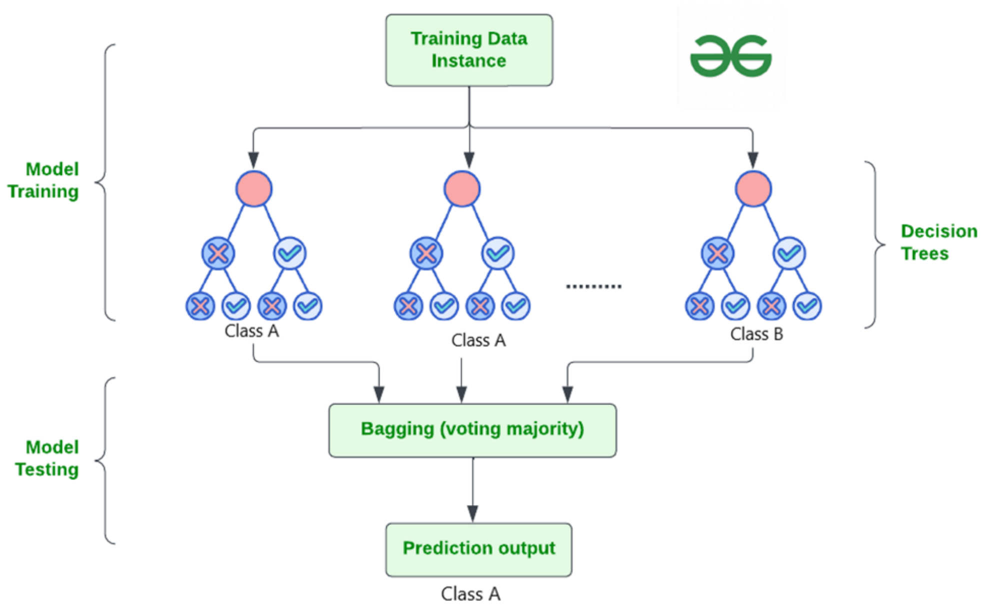


Figure 2.1: Random Forest Algorithm in Machine Learning [1]

Algorithm 2: Random Forest Algorithm in Machine Learning

```
Input: Dataset $D$
Output: Ensemble of $K$ trees
for $i \leftarrow 1$ to $K$ do
    Construct a subset $D_i$ by randomly taking (with duplicates) from $D$;
    Learn the $i$-th tree from $D_i$ as follows:
    for each node during tree construction do
        Randomly select a subset of attributes;
        Branch the tree based on that set of attributes;
        This tree will be grown to full size without pruning;
Each subsequent judgment is obtained by averaging the judgments from all
  trees;
```

2.2.2 Voting Classifier

a, Majority Voting

Majority voting [6] is the most popular and intuitive combination method in classification and regression tasks. In classification problems, the predictions for each class are summed and the class with the majority vote is returned as the ensemble prediction. Meanwhile, the majority vote is achieved in regression tasks by computing the average predictions from the various base learners. Assuming the decision of the $t-th$ classifier is $d_{t,c} \in \{0, 1\}$, $t = 1, ..., T$ and $c = 1, ..., C$, where T and C represents the number of classifiers and the number of classes, respectively.

Then, using majority voting, class is selected as the ensemble prediction, if

$$\sum_{t=1}^{T} d_{t,c} = \max_{c} \sum_{t=1}^{T} d_{t,c}$$

Meanwhile, the ensembles individual base models often do not have equal performance. Hence, considering them equally in the summation might be inappropriate. A more suitable solution is to weigh the performance of the individual models using the weighted majority voting technique.

b, Weighted Majority Voting

The weighted majority voting [7] assumes that some classifiers in the ensemble are more skilful than others, and their predictions are given more priority when computing the final ensemble prediction. The conventional majority voting assumes that all the base models are equally skilled and their predictions are treated equally when calculating the final ensemble prediction. However, the weighted majority voting assigns a specific weight to the base classifiers, which is then multiplied by the models output when computing the final ensemble prediction. Assuming the generalization ability of each base model is known, then a weight W_t can be assigned to classifier h_t according to its estimated generalization ability. Therefore, using the weighted majority voting, the ensemble classifier selects class c^*, if

$$\sum_{t=1}^{T} w_t d_{t,c^*} = \max_{c} \sum_{t=1}^{T} w_t d_{t,c}$$

Usually, the voting weights are normalized so that their sum is equal to 1.

2.2.3 Adaptive Boosting

The AdaBoost algorithm [8] is a type of boosting algorithm capable of using weak learners to obtain a robust classifier. It was developed in 1995 by Freundand Schapire and is among the most robust ML algorithms. AdaBoost was the first successful boosting algorithm, and the base learners are decision trees having a single split. Because the decision trees are short, they are usually called decision stumps. The most successful AdaBoost implementation is the AdaBoost M1, used for binary classification tasks.

The AdaBoost learning process involves training a base classifier using a base algorithm, usually a decision tree. The sample weights are adjusted with respect to the classifier's predictions, and the adjusted samples are employed for training the subsequent classifier. Therefore, the mis-classified samples are assigned larger weights and correctly classified instances are assigned lesser weights, ensuring that subsequent classifiers give more attention to the mis-classified samples.

The different base learners are added sequentially and weighed to obtain the strong classifier. At every iteration, the AdaBoost algorithm assigns weights to each instance in the training set. Given m labelled training instances $S = \{(x_1, y_1), ..., (x_i, y_i), ..., (x_m, y_m)\}$ where y_i is the target label of sample x_i and $y_i \in Y =$ {-1, +1}, the weight D_1 of the sample x_i and the weight update D_{t+1} are computed as:

$$D_1(i) = \frac{1}{n}, i = 1, 2, ..., m \tag{2.1}$$

$$D_{t+1}(i) = \frac{D_t(i)}{z_t} exp(-\alpha_t y_i h_t(x_i)), i = 1, 2, ..., \tag{2.2}$$

where $h_t(x)$ is the base classifier, $t = 1, ..., T$ is the number of iterations, Z_t denotes a normalization factor, while α_t is the weight of the classifier $h_t(x)$. The weight α_t measures the importance of the classifier $h_t(x)$ when obtaining the final classifier prediction. The instances that are wrongly predicted in $h_t(x)$ are assigned larger weights in the $t + 1$ training round. Furthermore, Z_t is selected such that D_{t+1} will be a distribution. Z_t and α_t are obtained using:

$$Z_t = \sum_{t+1}^{n} D_t(i) exp(-\alpha_t y_i h_t(x_i)) \tag{2.3}$$

$$\alpha_t = \frac{1}{2} ln \frac{1 - \epsilon_t}{\epsilon_t} \tag{2.4}$$

where ϵ_t represents the error rate of the classifier, and it is obtained using:

$$\epsilon_t = P[h_t(x_i) \neq y_i] = \sum_{i=n}^{n} D_i I(i)[h_t(x_i) \neq y_i] \tag{2.5}$$

When the given number of iterations have been completed, the final strong classifier is computed using:

$$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x)) \tag{2.6}$$

The AdaBoost algorithm is summarized in Algorithm 3. The AdaBoost is easy to implement with little need to tune its hyperparameters. Furthermore, the AdaBoost is flexible and can use a variety of algorithms as the base learner; hence, an algorithm suitable for a specific application can be used as the base learner, and the AdaBoost can enhance its performance. However, a limitation of the AdaBoost is that it is sensitive to noisy data and outliers because of its iterative learning approach, causing overfitting.

```
Algorithm 3: AdaBoost.M1 Algorithm
  Input: Training data $S$, The base algorithm $L$, The number of iterations $T$
  Output: Apply (2.6) to combine the predictions of the base classifiers to
          obtain the final strong classifier $H(x)$
  Initialize the weight $D_1$ of sample $x_i$
  for $t = 1, ..., T$ do
      Train the base classifier $h_t(x)$ by minimizing $\epsilon_t$
      Calculate the weight $\alpha_t$ of the classifier
      Update the samples weight
```

2.2.4 Gradient Boosting

Gradient boosting [9] is a machine learning algorithm that uses the boosting technique to create strong ensembles. It mainly uses decision trees as the base learner to produce a robust ensemble classifier, and it is also called gradient boosted decision tree (GBDT). The gradient boosting technique was first introduced by Breiman, who noted that boosting can be represented as an optimization technique on an appropriate loss function.

Subsequently, an extended version of the gradient boosting algorithm was developed by Friedman. The learning process of his algorithm involves sequentially training new models to obtain a robust classifier. It is developed in a step by-step manner similar to other boosting techniques, but its core idea is to develop base learners that are highly correlated with the negative gradient of the loss function related to the entire ensemble.

Given a training set $S = \{x_i, y_i\}_1^N$, the gradient boosting technique aims to find an approximation, $\hat{F}(x)$, of a function $F^*(x)$ that maps the predictor variables x to their response variables y, by minimizing the specified loss function $L(y, F(x))$. The GBDT creates an additive approximation of $F(x)$ through a weighted sum of functions:

$$F_m(x) = F_{m-1}(x) + \rho_m h_m(x)$$

where $\rho_m h_m(x)$ represents the weight of the m^{th} function, $h_{m(x)}$. These functions are the decision tree models in the ensemble. The algorithm performs the approximation iteratively. Meanwhile, a constant approximation of $F^*(x)$ is achieved using:

$$F_0(X) = argmin_\alpha \sum_{i=1}^{N} L(y_i, \alpha)$$

Successive base learners aim to minimize

$$(\rho_m h_m(x)) = argmin_{\rho,h} \sum_{i=1}^{N} L(y_i, F_{m-1}(x_i) + \rho h(x_i))$$

Meanwhile, rather than performing the optimization task directly, every h_m can be considered a greedy step in a gradient descent optimization for F^*. Therefore, every h_m is trained with a new training set $D = \{x_i, r_{mi}\}_{i=1}^{N}$, where r_{mi} represents the false residuals, and it is the difference between the output of an individual base classifier and the actual label. The false residual is also called pseudo residuals, and it is computed as:

$$r_{mi} = [\frac{\delta L(y_i, F(x)}{\delta F(x)}]_{F(x)=F_{m-1}}$$

Subsequently, the value of ρ_m miscalculated by performing a line search optimization. Meanwhile, this algorithm could overfit if the iterative task is not regularized appropriately. For certain loss functions, such as quadratic loss function, if h_m fits the false residuals perfectly, then in subsequent iteration the false residuals would be zero and the iteration ends early.

Furthermore, many regularization hyperparameters have been studied to optimize the GBDTs additive learning method. However, the intrinsic approach to regularize the GBDT is by using shrinkage to limit every gradient decent step $F_m(x) = F_{m-1}(x) + v\rho_m h_m(x)$, where v is normally assigned a value of 0.1. The gradient boosting procedure is summarized in Algorithm 4

Algorithm 4: Gradient Boosting

```
Input: Training data $S$, A differential loss function $L(y, F(x))$, The number
       of iterations $T$
Output: Return the final model $F_m(x)$
Initialize the model with a constant value using $F_0 = \arg\min_\alpha \sum_{i=1}^{N} L(y_i, \alpha)$
for $m = 1, \ldots, M$ do
    Calculate the pseudo-residuals $r_{mi} = -\left.\frac{\partial L(y_i,F(x_i))}{\partial F(x_i)}\right|_{F(x)=F_{m-1}(x)}$
    Train a base learner using the training set $D = \{(x_i, r_{mi})\}_{i=1}^{N}$
    Obtain $\rho_m$ by performing the line search optimization
    Update the model: $F_m(x) = F_{m-1}(x) + \rho_m h_m(x)$
```

The main advantage of gradient boosting is that, like other boosting algorithms, it can learn complex patterns from the input data since it is trained to correct the errors of the previous model. However, a model built using this algorithm can overfit and model noise if the input data is noisy. This algorithm is optimal for applications with small datasets.

2.3 Hyperparameter Tuning with Grid Search

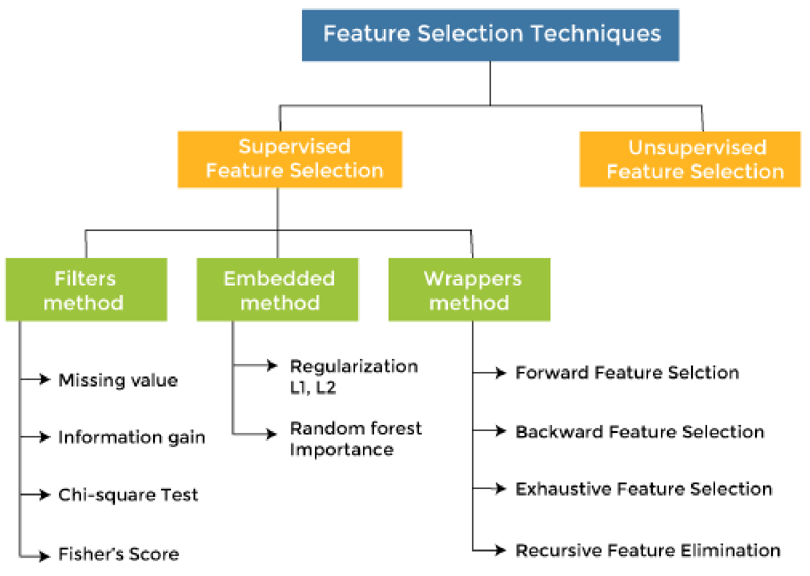
Grid search is an optimization technique used to brute force through all possible combinations of a set of variables. Fundamentally it's one of the most basic and simple optimization algorithms, but it's still quite powerful and guarantees finding the most optimal solution to your problem.

It works by creating a grid of all possible combinations of parameter values and testing each combination to find the best one. This grid of parameters is defined before the optimization/search step, hence the name grid search.

One application of grid search is in hyperparameter tuning, a commonly used technique for optimizing machine learning models. Machine learning models have several parameters that can be adjusted, known as hyperparameters. These hyperparameters have a significant impact on model performance making it crucial to find an optimized combination so that we can build good models.

In our problem, we will use Grid Search combined with Stratified Cross Validation to fine-tune the hyperparameters.

2.4 Feature Selection



There are mainly three techniques under supervised feature Selection:

Figure 2.2: Feature Selection

There are many feature selection methods at this time. One figure demonstrates the development of feature selection approaches is in the figure 2.2:

In this project, we only experiment with Sequential Feature Selection and Embedding Methods using Decision Tree and Random Forest.

2.4.1 Sequential Feature Selection (SFS)

Sequential feature selection (SFS) [10] is a greedy algorithm that iteratively adds or removes features from a dataset in order to improve the performance of a predictive model. SFS can be either forward selection or backward selection.
The process is as follows:

Algorithm 5: Feature Selection Process

```
Input: Predictive model, Number of features to select $N$, Scoring metric,
       Tolerance for improvement
Output: Selected set of features
Initialize the selector with the predictive model, the number of features to
  select $N$, the scoring metric, and the tolerance for improvement;
Fit the predictive model on the full set of features;
Evaluate the model on the training set using the scoring metric;
while desired number of features has not been selected do
    Determine the feature that most improves the model's cross-validation
      score or the feature that least reduces the model's cross-validation
      score;
    Update the selected features set by adding or removing the feature
      based on the greatest improvement in the scoring metric;
    Fit the predictive model on the updated set of features;
    Evaluate the model on the training set using the scoring metric;
```

The process is reversed if the selector is doing backward selection. During backward selection, selector starts with the entire set of features and iteratively removes the feature that has the least impact on the predictive model's performance. The process is repeated until the required number of features is chosen or until no additional features can be eliminated without significantly decreasing the model's performance.

2.4.2 Embedded Method

Among the various feature selection methods, embedded methods that integrate feature selection within the model training process have gained significant attention. This literature review explores the use of embedded methods, particularly focusing on Random Forest and Decision Tree algorithms, for feature selection in machine learning.

a, Decision Trees for Feature Selection

Decision Trees are intuitive and interpretable models that make decisions by splitting data into subsets based on feature values. They inherently perform feature selection during the training process by choosing splits that maximize information gain (or minimize impurity). This property makes Decision Trees a natural candidate for embedded feature selection.

Gini Importance and Information Gain: Decision Trees evaluate the importance of features based on metrics like Gini impurity or information gain. Each feature's importance can be quantified by the total reduction in these metrics it achieves across all splits in the tree. Studies have demonstrated that this embedded selection process effectively highlights significant features [1].

Advantages and Limitations: Decision Trees are easy to interpret and can handle both numerical and categorical data. However, they tend to overfit on training data and are sensitive to small changes in the data, which can affect the stability of feature selection.

b, Random Forests for Feature Selection

Random Forests, an ensemble learning method that constructs multiple Decision Trees, improve the robustness and accuracy of individual trees by averaging their predictions. The feature selection capability of Random Forests is derived from the aggregation of feature importance scores across all trees in the forest.

Mean Decrease Impurity (MDI): Random Forests use the average reduction in impurity (Gini or entropy) attributed to each feature, providing a comprehensive measure of feature importance [5].

Empirical Success: Random Forests are highly effective for feature selection due to their ability to handle large datasets, their robustness to overfitting, and their capability to capture complex interactions between features [11].

2.5 Model Assessment

When assessing a classification model's performance, a confusion matrix is essential. It offers a thorough analysis of true positive, true negative, false positive, and false negative predictions, facilitating a more profound comprehension of a model's recall, accuracy, precision, and overall effectiveness in class distinction. When there is an uneven class distribution in a dataset, this matrix is especially helpful in evaluating a model's performance beyond basic accuracy metrics.

Actual	Predicted	
	Positive	Negative
Positive	TruePositive	FalseNegative
Negative	FalsePositive	TrueNegative

Table 2.1: Confusion Matrix

2.5.1 Accuracy

Accuracy measures the proportion of correct predictions to the total number of predictions. It's a widely-used metric, as it reports the overall predictive performance of a model. The formula is as following:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

2.5.2 F1-Score

Precision measures the proportion of true positives (TP) to the total number of positive predictions made. It's a useful metric when false positives are more costly than false negatives.

$$Precision = \frac{TP}{TP + FP}$$

Recall is the ratio of the number of true positives (TP) to the sum of true positives (TP) and false negatives (FN). This metric measures the percentage of all positive instances in the dataset that are correctly classified by the model.

$$Recall = \frac{TP}{TP + FN}$$

To evaluate model performance comprehensively, we should examine both precision and recall. The F1 score serves as a helpful metric that considers both of them.

$$F1_{score} = \frac{2 * Precision * Recall}{Precision + Recall}$$

In multi-class classification, we calculate the F1 score for each class in a One-vs-Rest (OvR) approach instead of a single overall F1 score as seen in binary classification. In this OvR approach, we determine the metrics for each class separately, as if there is a different classifier for each class. However, instead of having multiple per-class F1 scores, it would be better to average them to obtain a single number to describe overall performance.

The macro-averaged F1 score (or macro F1 score) is computed by taking the arithmetic (unweighted) mean of all the per-class F1 scores. This method treats all classes equally regardless of their support values.

The weighted-averaged F1 score is calculated by taking the mean of all per-class F1 scores while considering each class's support. Support refers to the number of actual occurrences of the class in the dataset.

Micro averaging computes a global average F1 score by counting the sums of the True Positives (TP), False Negatives (FN), and False Positives (FP). We first sum the respective TP, FP, and FN values across all classes and then plug them into the F1 equation to get our micro F1 score.

CHAPTER 3. METHODOLOGY

3.1 Datasets

The dataset we use for this problem is driven from Kaggle competition [1]. For this project scope, we only utilize the train dataset, which is a table-type dataset, including 18 columns and 20758 rows. Full description of this dataset is outlined as below:

Feature	**Datatype**	**Description**
ID	Categorical	Unique identifier
Smoke	Categorical	Smoker or not
Weight	Numerical	Weight (Float)
Age	Numerical	Age (Float)
Height	Numerical	Height (Float)
Gender	Categorical	Gender
Family_history_with_overweight	Categorical	Family history with overweight
FAVC	Categorical	Frequent consumption of high-caloric food items
FCVC	Numerical	Frequency of consuming vegetables (Float)
NCP	Numerical	Number of main meals consumed per day (Float)
CAEC	Categorical	Frequency of consuming food between meals
CH20	Numerical	Amount of water consumed daily (Float)
CALC	Categorical	Frequency of alcohol consumption
SCC	Categorical	Monitoring of calorie consumption
FAF	Numerical	Frequency of engaging in physical activity (Float)
TUE	Numerical	Time spent using technology devices (Float)
MTRANS	Categorical	Mode of transportation used

[1]https://www.kaggle.com/competitions/playground-series-s4e2/data

NObeyesdad	Categorical	Categorizes individuals obesity categories (TARGET)

Table 3.1: Description of each column in the dataset

The data contains 17 attributes (one attribute is for ID) , the records are labeled with the class variable NObesity (Obesity Level), using the labels of Insufficient Weight, Normal Weight, Overweight Level I, Overweight Level II, Obesity Type I, Obesity Type II and Obesity Type III.

3.2 Data Preprocessing

3.2.1 Data Splitting

We divide the dataset into training and test using 80% of the dataset for training the model and the remaining 20% for validating the model's accuracy. Using the train-test-split technique from the Scikit-Learn package, we accomplish the aforementioned. The feature on which the dataset is split into training and testing is specified by the function's stratify property. The number of training and test samples after splitting are 16606 and 4152, respectively.

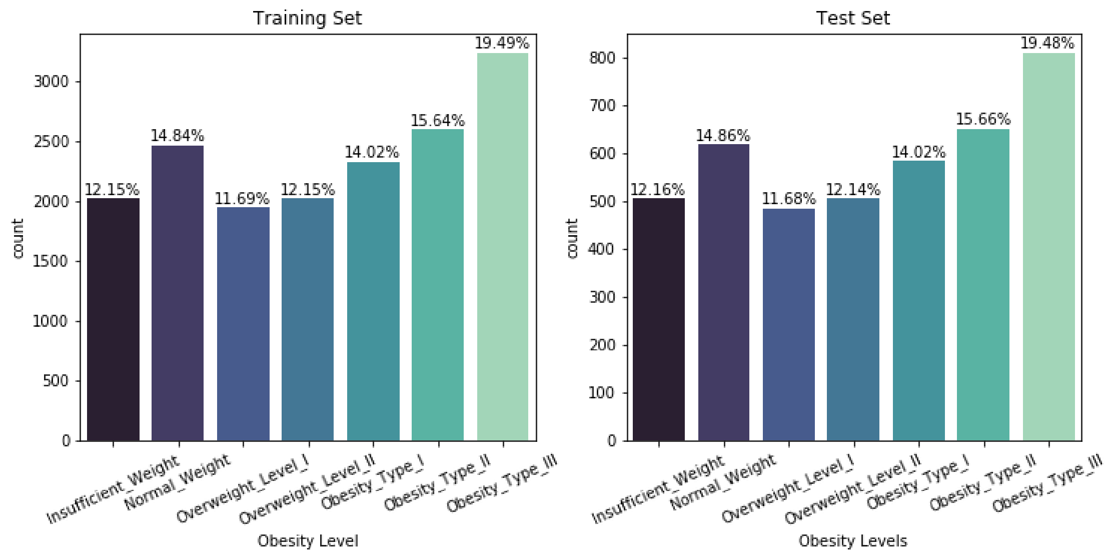


Figure 3.1: Label Distribution on Training and Test set

3.2.2 Exploratory Data Analysis

For our Exploratory Data Analysis (EDA), we take it in two main steps:

1. Univariate Analysis: We focus on one feature at a time to understand its distribution and range.

2. Bivariate Analysis: In this step, we explore the relationship between each feature and the target variable. This helps us figure out the importance and influ-

ence of each feature on the target outcome.

With these two steps, we aim to gain insights into the individual characteristics of the data and also how each feature relates to our main goal: predicting the target variable.

3.2.3 Preprocessing

In this part, we decide to do the following steps: remove irrelevant attribute, handle missing values, encode the categorical features and scale data.

After exploring the data, we found that the "id" attribute is irrelevant so we remove it from the data. There was also no null value of any attribute.

In the next stage, we perform categorical attributes encoding. Here, we decided to apply ordinal encoding to binary-value features (yes/no, true/false), and ordinal-data features. Mean while, for the other case we will apply one-hot encoding (in this case for 'MTRANS' column). And we also define a dictonary for label mapping, which convert each text label of target variables to a specific number.

In the final stage, we intended to apply feature scaling for our training models. Feature scaling is a data preprocessing technique used to transform the values of features or variables in a dataset to a similar scale. The purpose is to ensure that all features contribute equally to the model and to avoid the domination of features with larger values. Models like SVM, KNN and many linear models rely on distances or gradients, making them susceptible to variations in feature scales. Scaling ensures that all features contribute equally to the model's decision rather than being dominated by features with larger magnitudes. But we skipped it during the preprocessing session, since not all algorithms require scaled data. For instance, Decision Tree-based models are scale-invariant. Given our intent to use a mix of models (some requiring scaling, others not), we've chosen to handle standard scaling later using pipelines. This approach lets us apply scaling specifically for models that benefit from it, ensuring flexibility and efficiency in our modeling process.

3.3 Training classification models

During the model training phase, we implement the following model training methods with the following pipelines: (1) Train and tune the hyperparameters of the models: kNN, Decision Tree, Random Forest, and SVC, (2) Train the models with the hyperparameters tuned from (1) along with the feature selection method described in the previous section (using SFS and Ensemble Method by Random Forest and Decision Tree), (3) Train multilayer perceptrons, (4) Train with ensemble methods (including Voting Classifier, AdaBoost, and GradientBoost). The details of each pipeline is defined in each subsection below:

3.3.1 Training kNN, Decision Tree, Random Forest and SVC

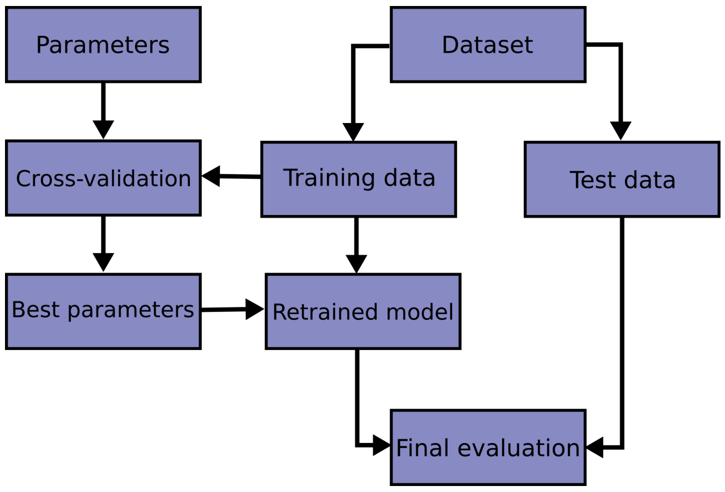


Figure 3.2: Training model with Grid Search workflow [2]

In this pipeline, we tune the hyperparameters of each model using the Grid Search method combined with the Stratified K-Fold method with $k = 5$. The metric used to find the best hyperparameters for each model is ‘accuracy’. After finding the best hyperparameters for each model, the models will be retrained with the original training data to produce the best results. The detailed results of the parameter tuning are recorded as follows:

Model	**Hyperparameters**
k-NN	metric: manhattan, n_neighbors: 17, weights: distance
Decision Tree	criterion: entropy, max_depth: 11, min_samples_leaf: 10, min_samples_split: 2, splitter: best
Random Forest	criterion: entropy, max_depth: None, max_features: sqrt, n_estimators: 900
SVC	C: 5, kernel: linear

Table 3.2: Hyperparameters for each model

3.3.2 Training Models With Feature Selection

In pipeline (2), we train the model with the hyperparameters selected in part 1, then train the model combined with the feature selection method and evaluate it on the test set.

In this case, we using 2 main methods. Firstly, we use SFS method. This method is implemented along with the baseline model with selected hyperparameters in a pipeline, then training model with maximum 10 features selected which drive the best ‘accuracy’ score. For the second method, we training Decision Tree and Random Forest to deliver the best selected combination of 10 features. Then we use these two combination of featured to train again with the model with selected

[2]https://scikit-learn.org/stable/modules/cross_validation.html

hyperparameters.

3.3.3 Training Multilayer Perceptrons

a, Architecture

Our selected multiple perceptron model is a multilayer perceptron (MLP) implemented using the Keras Sequential API [3]. Here's a breakdown of its architecture and configuration:

- Input Layer: The model starts with an input layer specified by 20 input features.
- Hidden Layers: The model consists of three hidden layers, each densely connected to the previous layer:
  - The first hidden layer has 64 neurons with a tanh activation function.
  - The second hidden layer has 64 neurons with a tanh activation function.
  - The third hidden layer has 64 neurons with a tanh activation function.
- Dropout layer: Each hidden layer is connected with a dropout layer of rate 0.01 to avoid overfitting.
- Output Layer: The model ends with a single neuron output layer, which predicts a value (classification task) with a softmax activation function.
- Loss Function: The loss function used for training is sparse categorical cross entropy.
- Optimizer: The Adam optimizer with a learning rate of 0.01 is used to optimize the model parameters during training. Adam is an adaptive learning rate optimization algorithm that combines the advantages of both AdaGrad and RMSProp, and is well-suited for training deep neural networks.

b, Training process

We train the multi-layer perceptron model for up to 50 epochs, with a validation set ratio of 0.2 and a batch size of 32. To prevent overfitting on the training set and enhance the model's generalization, we use Early Stopping with the validation loss as the monitored metric, and a patience of 7 epochs. Additionally, we set the parameter restore_best_weights to True, which saves the model at the checkpoint with the best evaluation score.

[3] `https://keras.io/api/models/sequential/`

3.3.4 Training Ensemble Models

With the ensemble learning methods proposed in the previous chapter 2.2, we implement three ensemble learning methods at this stage: Voting Classifier, Adaptive Boosting Classifier, and Gradient Boosting Classifier.

For the Voting Classifier method, we combine four base models: kNN, Decision Tree, Random Forest, and SVC with the hyperparameters selected from the previous section. The voting method used is soft voting.

For the Adaptive Boosting method, we use the Decision Tree as the base estimator with the hyperparameters chosen from the previous section. Then, we tune the hyperparameters of AdaBoost, including the number of trees and the learning rate. After the hyperparameter tuning process using GridSearch with Stratified K-Fold as mentioned earlier 3.3, we obtain the hyperparameters for AdaBoost as $n\_estimators = 500$ and $learning\_rate = 1$.

For the Gradient Boosting method, we also perform hyperparameter tuning with the parameters: n_estimators, learning_rate, and max_depth. After the tuning process, the values of the hyperparameters are 200, 0.1, and 3, respectively.

To draw the conclusion, the full pipeline for our methodology is presented in this figure below:

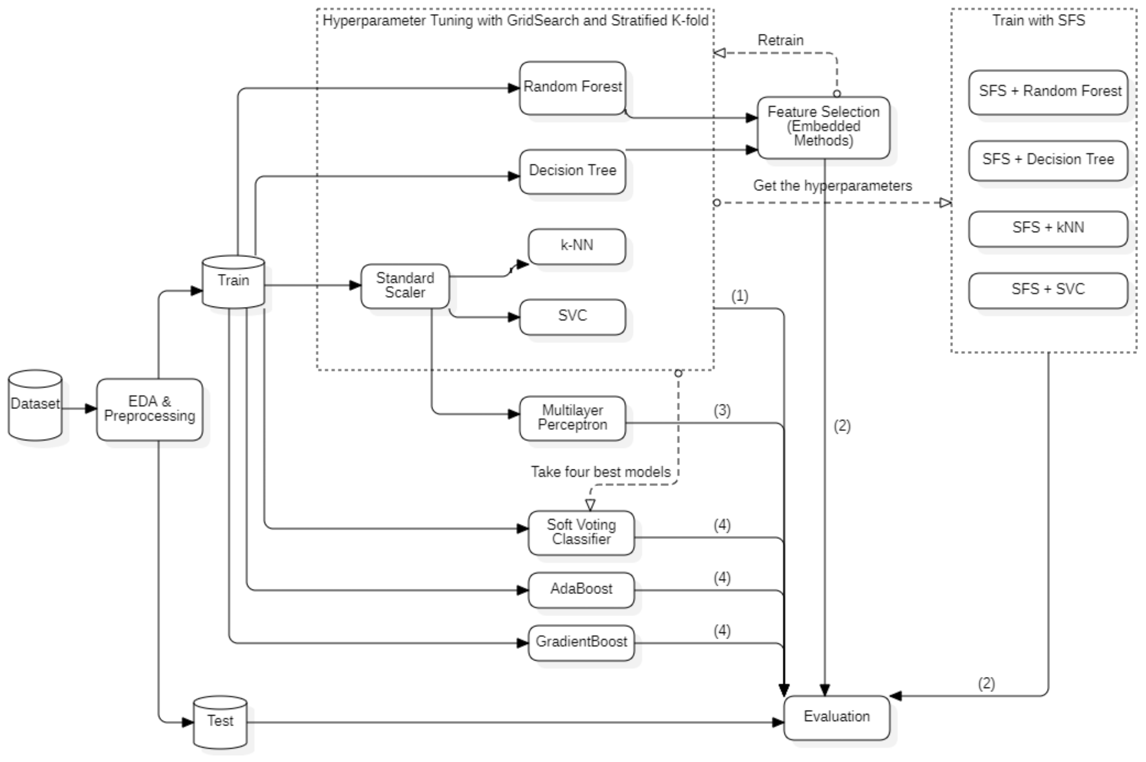


Figure 3.3: Pipeline

CHAPTER 4. EXPERIMENTAL RESULT

In this chapter, we will present the results achieved by the machine learning models for the task of classifying obesity risk levels. We will evaluate these models on a predefined test set 3.2.1, which is used exclusively for assessing the final model and does not appear in the training set. The evaluation metrics include accuracy, macro F1, and weighted F1.

The details results of our experiment is presented below:

Model	**Feature Selection**			**Macro F1**	**Weighted F1**	**Accuracy**
	Decision Tree	**Random Forest**	**SFS**	**(%)**	**(%)**	**(%)**
Decision Tree	-	-	-	86.78	88.00	88.03
	yes	-	-	86.40	87.66	87.67
	-	yes	-	86.12	87.43	87.48
	-	-	yes	86.14	87.34	87.40
Random Forest	-	-	-	89.28	90.31	90.37
	yes	-	-	88.79	89.88	89.93
	-	yes	-	88.36	89.50	89.55
	-	-	yes	89.11	90.13	90.17
k-NN	-	-	-	77.17	79.12	79.36
	yes	-	-	78.15	80.13	80.32
	-	yes	-	77.44	79.63	80.08
	-	-	yes	83.78	85.20	85.24
SVC	-	-	-	85.50	86.86	86.95
	yes	-	-	84.69	86.17	86.27
	-	yes	-	84.93	86.35	86.42
	-	-	yes	84.99	86.37	86.46
Multilayer Perceptron	-	-	-	87.42	88.61	88.66
Voting Classifier	-	-	-	89.22	90.21	90.29
AdaBoost	-	-	-	89.19	90.21	90.20
GradientBoosting	-	-	-	**89.54**	**90.53**	**90.56**

Table 4.1: Experimental Results

Based on the results above, it can be seen that the Gradient Boosting method gives the best results across all three metrics. Additionally, for feature selection methods, Sequential Feature Selection provides features with higher importance compared to the methods using Decision Tree and Random Forest.

CHAPTER 5. CONCLUSIONS

5.1 Summary

In summary, with the given topic, we have implemented methods to build a machine learning model applicable to real-world problems. Specifically, the problem of classifying obesity risk levels has been addressed using basic machine learning models. Machine learning methods, feature selection methods, and tuning methods were applied to build a highly accurate model. In our problem, the Gradient Boosting model yielded the best results, demonstrating that using machine learning models for practical problems can also achieve high accuracy.

Using the Gradient Boosting method and hyperparameter tuning with Grid Search combined with Stratified K-Fold resulted in a model with an accuracy of 90.56. However, we can see that this accuracy can be further improved with more advanced machine learning methods, such as deep learning.

5.2 Suggestion for Future Works

In the future, we hope to address this problem using some advanced machine learning methods, such as deep learning with neural networks of certain complexity, to improve the results.

Additionally, we also aim to deploy these machine learning models as a small module in medical software to help doctors make better obesity diagnoses for patients with the assistance of AI or even a software with GUI helping people to check their health status at their own home.

REFERENCE

[1] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, "Classification and regression trees," *Biometrics*, vol. 40, p. 874, 1984. [Online]. Available: https://api.semanticscholar.org/CorpusID:29458883.

[2] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "Knn model-based approach in classification," in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, R. Meersman, Z. Tari, and D. C. Schmidt, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 986–996, ISBN: 978-3-540-39964-3.

[3] M. Hearst, S. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their Applications*, vol. 13, no. 4, pp. 18–28, 1998. DOI: 10.1109/5254.708428.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:6628106.

[5] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:89141.

[6] D. Ruta and B. Gabrys, "Classifier selection for majority voting," *Information Fusion*, vol. 6, no. 1, pp. 63–81, 2005, Diversity in Multiple Classifier Systems, ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2004.04.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1566253504000417.

[7] N. Littlestone and M. Warmuth, "The weighted majority algorithm," *Information and Computation*, vol. 108, no. 2, pp. 212–261, 1994, ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540184710091.

[8] Y. Freund, R. Schapire, and N. Abe, "A short introduction to boosting," *Journal-Japanese Society For Artificial Intelligence*, vol. 14, no. 771-780, p. 1612, 1999.

[9] L. Breiman, "Arcing classifiers," *Annals of Statistics*, vol. 26, pp. 123–40, 1996.

[10] T. Rückstieß, C. Osendorfer, and P. van der Smagt, "Sequential feature selection for classification," in *AI 2011: Advances in Artificial Intelligence*, D.

Wang and M. Reynolds, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 132–141, ISBN: 978-3-642-25832-9.

[11] D. Cutler, T. Edwards, K. Beard, *et al.*, “Random forests for classification in ecology,” *Ecology*, vol. 88, pp. 2783–92, Dec. 2007. DOI: `10.1890/07-0539.1`.

APPENDIX

A. DETAILED RESULTS

A.1 Decision Tree

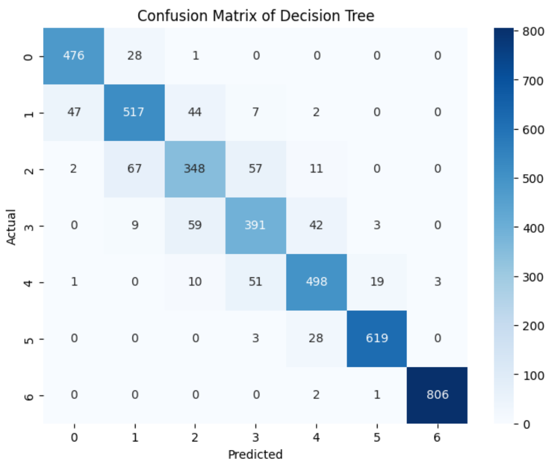


Figure A.1: Decision Tree

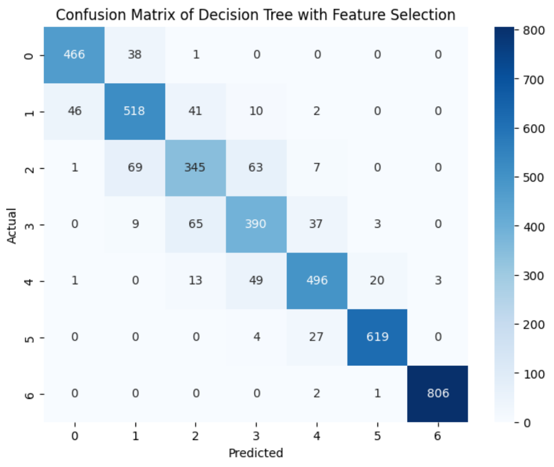


Figure A.2: Decision Tree with Feature Selection

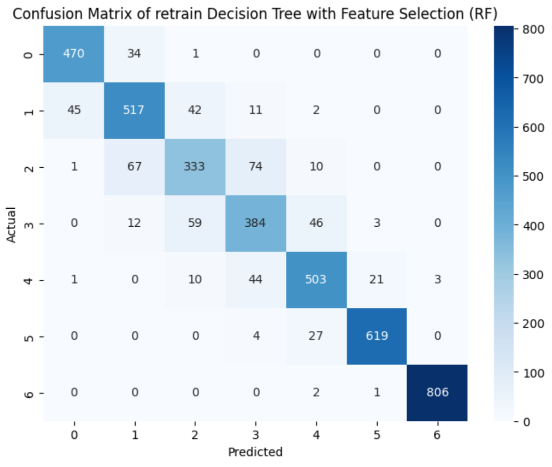


Figure A.3: Decision Tree with Feature Selection (RF)

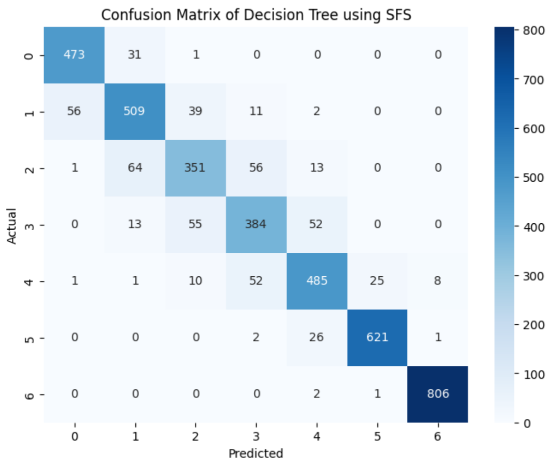
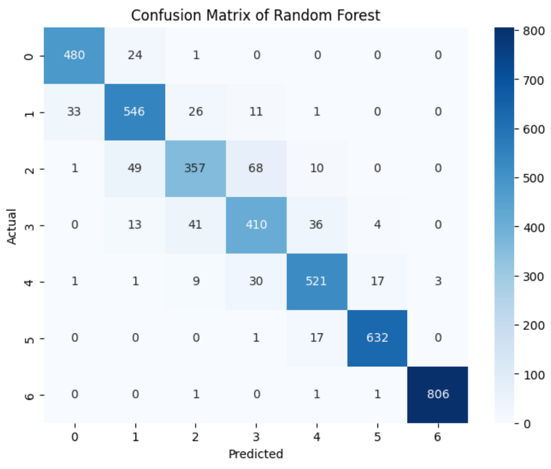


Figure A.4: Decision Tree using SFS

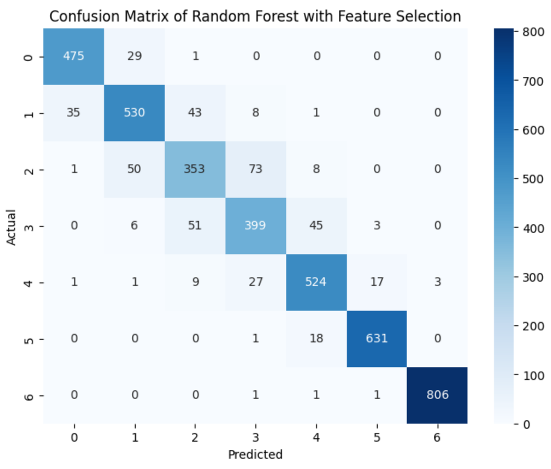
Figure A.5: Confusion Matrix of Decision Tree

A.2 Random Forest



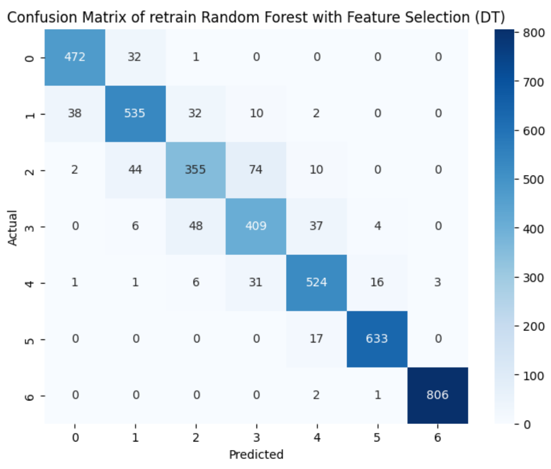
Actual \ Predicted	0	1	2	3	4	5	6
0	480	24	1	0	0	0	0
1	33	546	26	11	1	0	0
2	1	49	357	68	10	0	0
3	0	13	41	410	36	4	0
4	1	1	9	30	521	17	3
5	0	0	0	1	17	632	0
6	0	0	1	0	1	1	806

Figure A.6: Random Forest



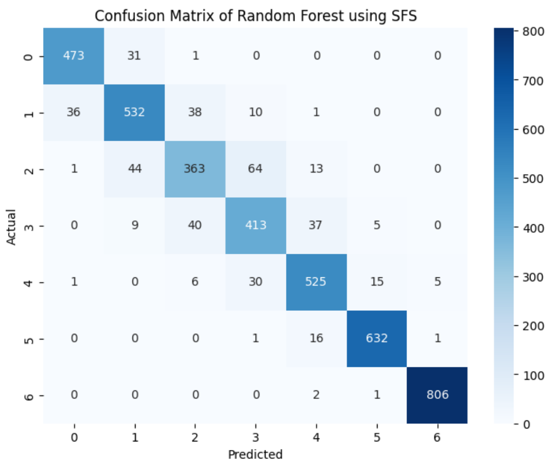
Actual \ Predicted	0	1	2	3	4	5	6
0	475	29	1	0	0	0	0
1	35	530	43	8	1	0	0
2	1	50	353	73	8	0	0
3	0	6	51	399	45	3	0
4	1	1	9	27	524	17	3
5	0	0	0	1	18	631	0
6	0	0	0	1	1	1	806

Figure A.7: Random Forest with Feature Selection



Actual \ Predicted	0	1	2	3	4	5	6
0	472	32	1	0	0	0	0
1	38	535	32	10	2	0	0
2	2	44	355	74	10	0	0
3	0	6	48	409	37	4	0
4	1	1	6	31	524	16	3
5	0	0	0	0	17	633	0
6	0	0	0	0	2	1	806

Figure A.8: Random Forest with Feature Selection (DT)

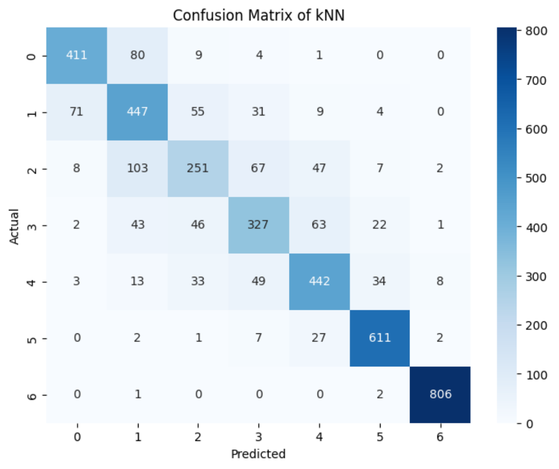


Actual \ Predicted	0	1	2	3	4	5	6
0	473	31	1	0	0	0	0
1	36	532	38	10	1	0	0
2	1	44	363	64	13	0	0
3	0	9	40	413	37	5	0
4	1	0	6	30	525	15	5
5	0	0	0	1	16	632	1
6	0	0	0	0	2	1	806

Figure A.9: Random Forest using SFS

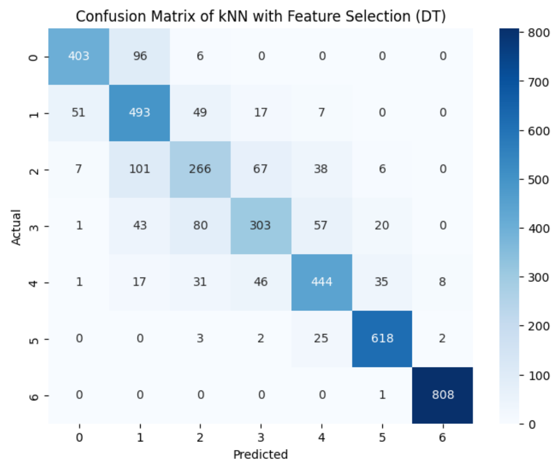
Figure A.10: Confusion Matrix of Random Forest

A.3 k-NN



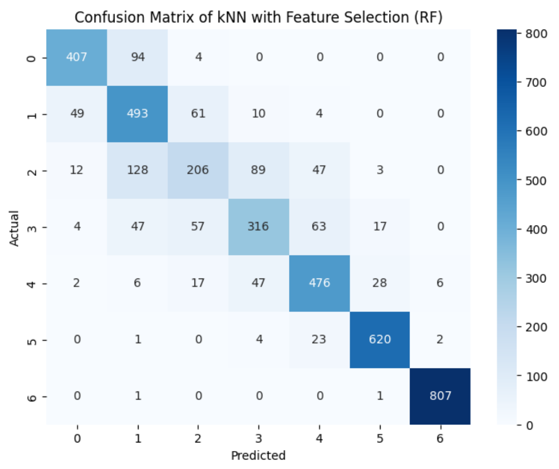
Actual \ Predicted	0	1	2	3	4	5	6
0	411	80	9	4	1	0	0
1	71	447	55	31	9	4	0
2	8	103	251	67	47	7	2
3	2	43	46	327	63	22	1
4	3	13	33	49	442	34	8
5	0	2	1	7	27	611	2
6	0	1	0	0	0	2	806

Figure A.11: k-NN



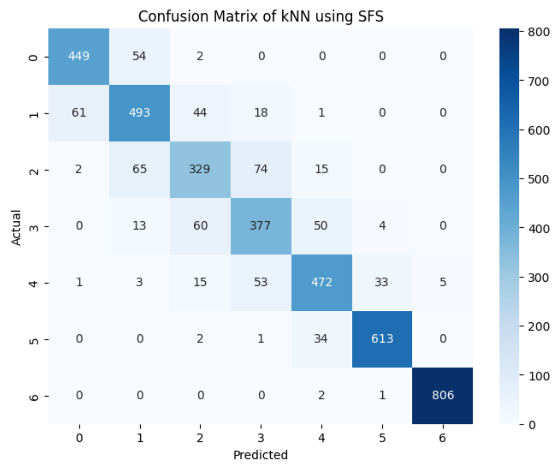
Actual \ Predicted	0	1	2	3	4	5	6
0	403	96	6	0	0	0	0
1	51	493	49	17	7	0	0
2	7	101	266	67	38	6	0
3	1	43	80	303	57	20	0
4	1	17	31	46	444	35	8
5	0	0	3	2	25	618	2
6	0	0	0	0	0	1	808

Figure A.12: k-NN with Feature Selection (DT)



Actual \ Predicted	0	1	2	3	4	5	6
0	407	94	4	0	0	0	0
1	49	493	61	10	4	0	0
2	12	128	206	89	47	3	0
3	4	47	57	316	63	17	0
4	2	6	17	47	476	28	6
5	0	1	0	4	23	620	2
6	0	1	0	0	0	1	807

Figure A.13: k-NN with Feature Selection (RF)



Actual \ Predicted	0	1	2	3	4	5	6
0	449	54	2	0	0	0	0
1	61	493	44	18	1	0	0
2	2	65	329	74	15	0	0
3	0	13	60	377	50	4	0
4	1	3	15	53	472	33	5
5	0	0	2	1	34	613	0
6	0	0	0	0	2	1	806

Figure A.14: k-NN using SFS

Figure A.15: Confusion Matrix of k-NN

A.4 SVC

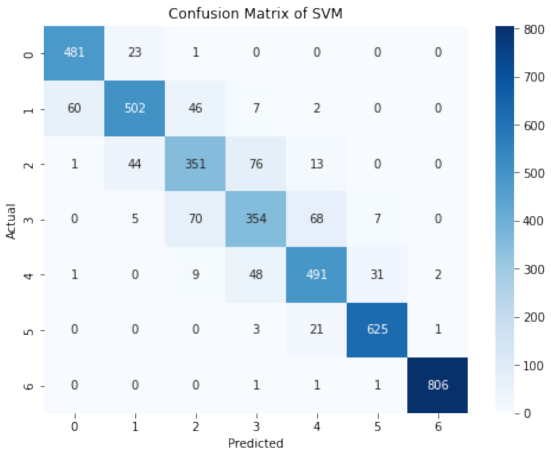


Figure A.16: SVC

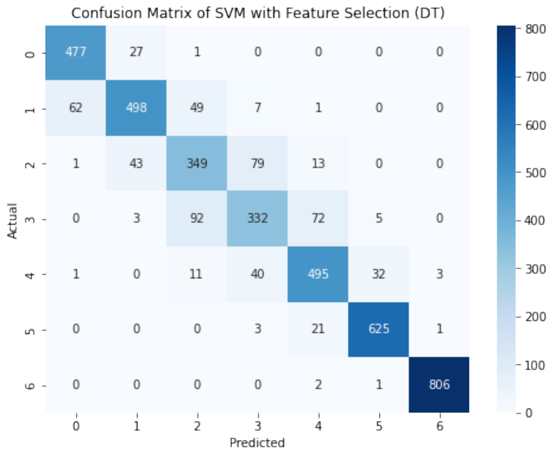


Figure A.17: SVC with Feature Selection (DT)

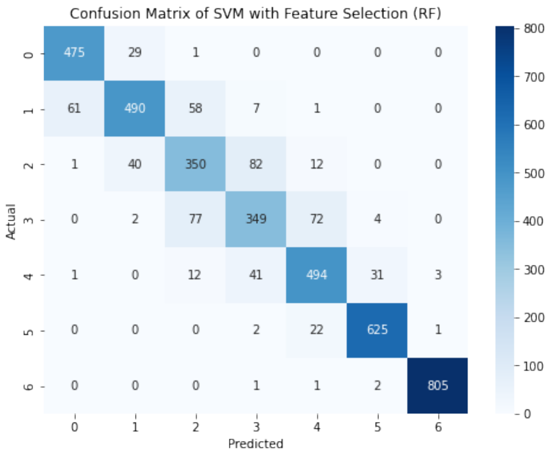


Figure A.18: SVC with Feature Selection (RF)

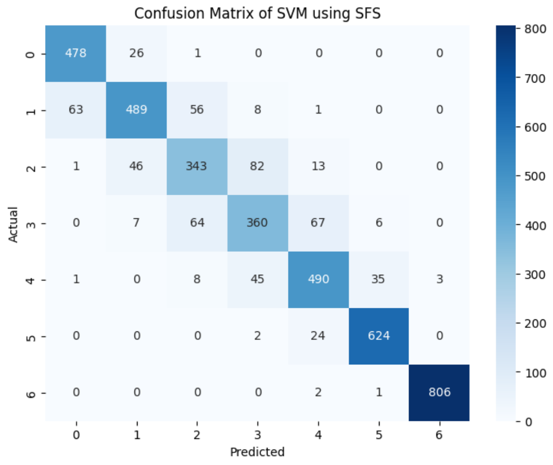
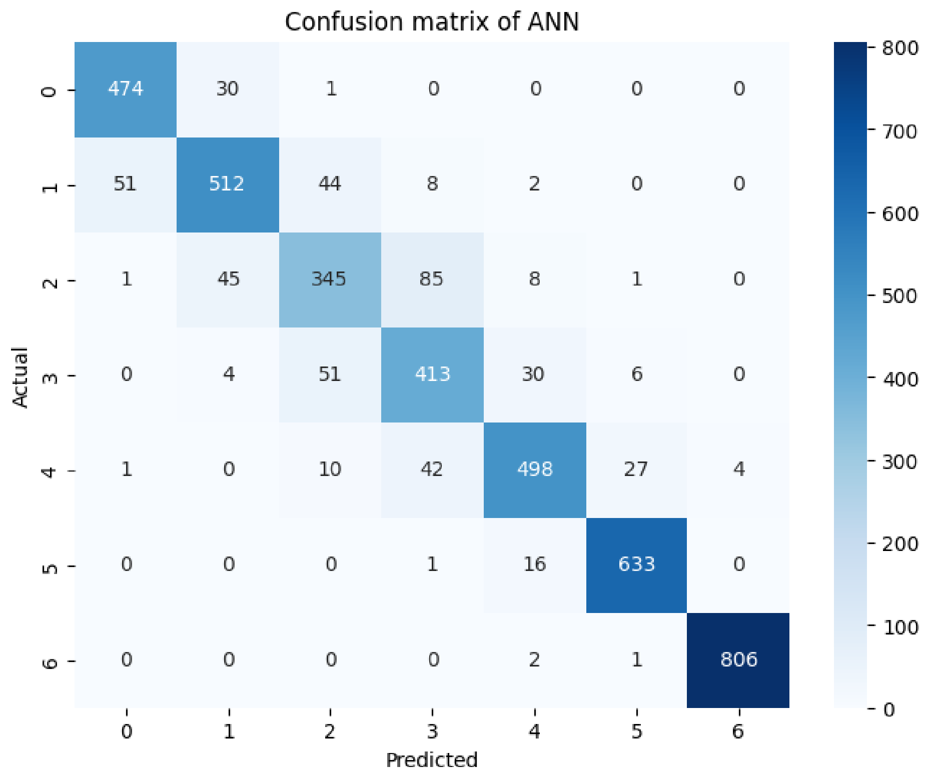


Figure A.19: SVC using SFS

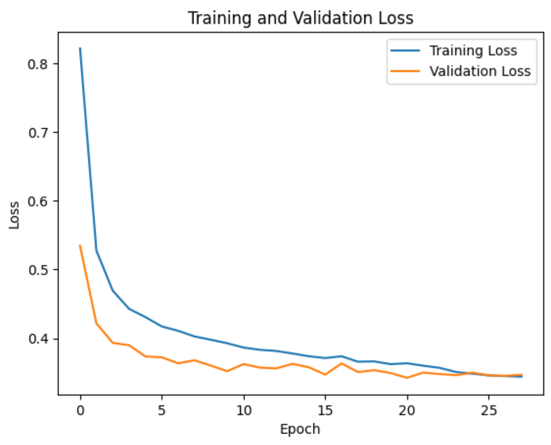
Figure A.20: Confusion Matrix of SVC

A.5 Multilayer Perceptron

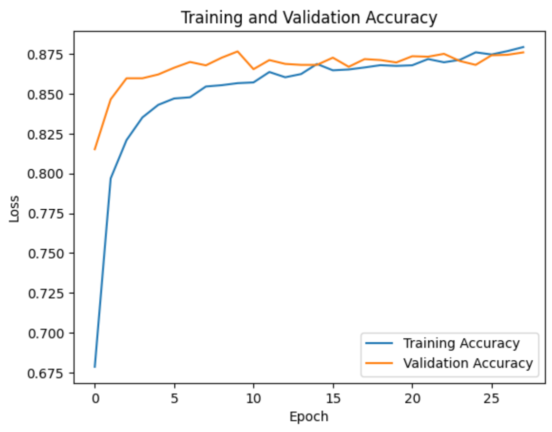


Actual \ Predicted	0	1	2	3	4	5	6
0	474	30	1	0	0	0	0
1	51	512	44	8	2	0	0
2	1	45	345	85	8	1	0
3	0	4	51	413	30	6	0
4	1	0	10	42	498	27	4
5	0	0	0	1	16	633	0
6	0	0	0	0	2	1	806

Figure A.21: Confusion Matrix of ANN



(a) ANN Loss



(b) ANN Accuracy

Figure A.22: Training & Validation Loss and Accuracy of ANN

A.6 Ensemble Method

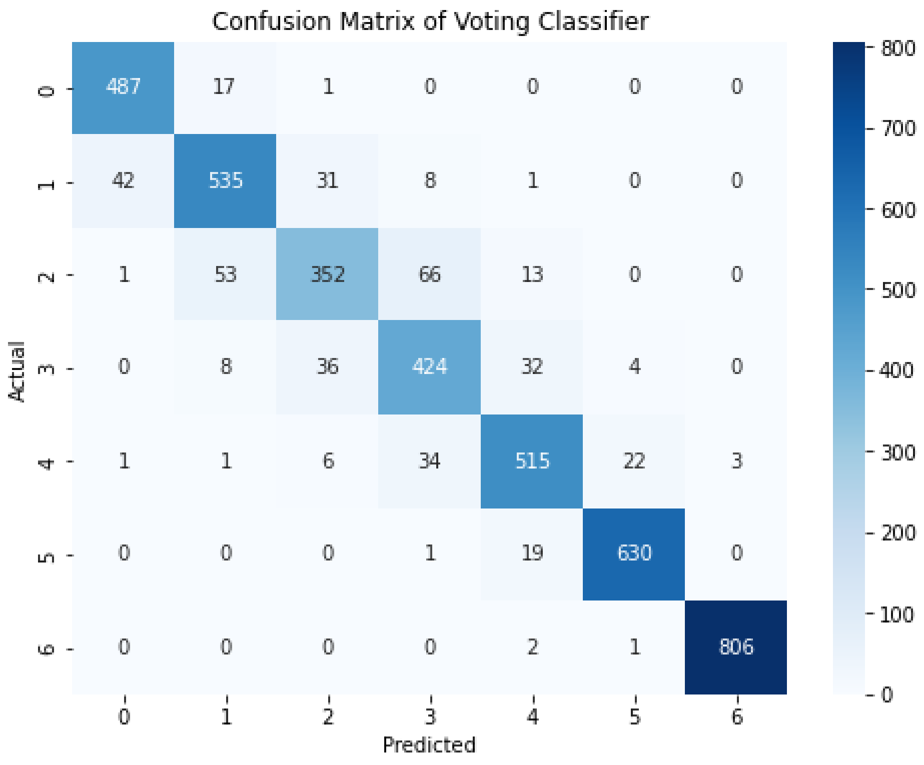
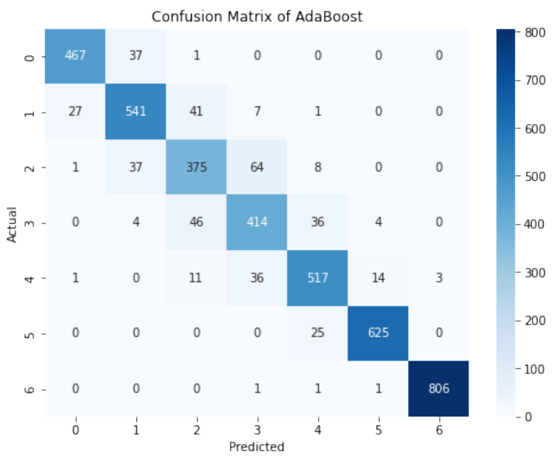
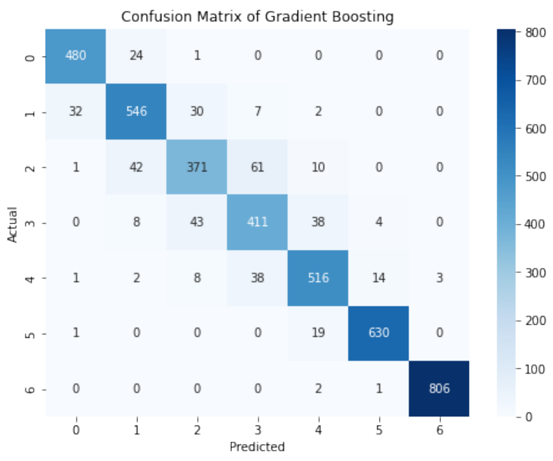


Figure A.23: Confusion Matrix of Voting Classifier



(a) AdaBoost



(b) Gradient Boosting

Figure A.24: Confusion Matrix of Boosting methods

B. CONFIGURATION

B.1 Libraries

This section details the libraries and modules imported for the implementation of the machine learning pipeline and associated functions.

- **seaborn**
  - *Purpose*: Statistical data visualization library based on matplotlib.
  - *Usage*: Creating visually appealing plots for data analysis.
- **matplotlib.pyplot**
  - *Purpose*: Comprehensive library for creating static, animated, and interactive visualizations in Python.
  - *Usage*: Plotting graphs and visualizations.
- **warnings**
  - *Purpose*: Python built-in library to handle warnings.
  - *Usage*: Suppressing warning messages during the execution of the script.
- **numpy**
  - *Purpose*: Fundamental package for scientific computing in Python.
  - *Usage*: Handling arrays and mathematical operations.
- **pandas**
  - *Purpose*: Powerful data manipulation and analysis library for Python.
  - *Usage*: Data structures and data analysis tools.
- **sklearn.preprocessing.StandardScaler**
  - *Purpose*: Standardizes features by removing the mean and scaling to unit variance.
  - *Usage*: Normalizing input features before training the model.
- **sklearn.pipeline.Pipeline**
  - *Purpose*: Sequentially apply a list of transforms and a final estimator.
  - *Usage*: Creating machine learning pipelines for streamlined model training and evaluation.
- **sklearn.metrics.confusion_matrix**

  - *Purpose*: Compute confusion matrix to evaluate the accuracy of a classification.
  - *Usage*: Evaluating the performance of the classification model.

- **sklearn.metrics.classification_report**
  - *Purpose*: Build a text report showing the main classification metrics.
  - *Usage*: Detailed report of precision, recall, F1-score, and support for the classification model.

- **sklearn.neighbors.KNeighborsClassifier**
  - *Purpose*: Classifier implementing the k-nearest neighbors vote.
  - *Usage*: Building and training a k-nearest neighbors classification model.

- **sklearn.feature_selection.SelectFromModel**
  - *Purpose*: Meta-transformer for selecting features based on importance weights.
  - *Usage*: Feature selection using a model with feature importance attributes.

- **sklearn.model_selection.train_test_split**
  - *Purpose*: Utility for splitting data arrays into two subsets: for training data and for testing data.
  - *Usage*: Splitting the dataset into training and testing sets for model validation.

- **sklearn.preprocessing.OrdinalEncoder**
  - *Purpose*: Encode categorical features as an integer array.
  - *Usage*: Transforming categorical data into a format that can be provided to machine learning algorithms to improve predictions.

- **sklearn.ensemble.RandomForestClassifier**
  - *Purpose*: A meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.
  - *Usage*: Building and training a Random Forest classification model.

- **sklearn.tree.DecisionTreeClassifier**
  - *Purpose*: A decision tree classifier.
  - *Usage*: Building and training decision tree models for classification tasks.

- **sklearn.pipeline.Pipeline**
  - *Purpose*: Sequentially apply a list of transforms and a final estimator.
  - *Usage*: Creating machine learning pipelines for streamlined model training and evaluation.
- **sklearn.svm.SVC**
  - *Purpose*: Support Vector Machine classifier.
  - *Usage*: Building and training SVM models for classification tasks.
- **keras**
  - *Purpose*: High-level neural networks API.
  - *Usage*: Building and training deep learning models.
- **tensorflow**
  - *Purpose*: End-to-end open-source platform for machine learning.
  - *Usage*: Backend for Keras and for building custom deep learning models.
- **keras.models.Sequential**
  - *Purpose*: Sequential model in Keras.
  - *Usage*: Building neural network models layer-by-layer.
- **keras.layers.Dense**
  - *Purpose*: Fully connected layer in a neural network.
  - *Usage*: Adding dense layers to a neural network model.
- **keras.layers.Dropout**
  - *Purpose*: Dropout layer for regularization.
  - *Usage*: Preventing overfitting in neural network models.
- **keras.wrappers.scikit_learn.KerasClassifier**
  - *Purpose*: Wrapper for using Keras models with scikit-learn.
  - *Usage*: Integrating Keras models into scikit-learn workflows.
- **sklearn.model_selection.GridSearchCV**
  - *Purpose*: Exhaustive search over specified parameter values for an estimator.
  - *Usage*: Hyperparameter tuning for machine learning models.

- **keras.callbacks.EarlyStopping**
  - *Purpose*: Stop training when a monitored metric has stopped improving.
  - *Usage*: Preventing overfitting by stopping training early.
- **tqdm**
  - *Purpose*: Fast, extensible progress bar for Python.
  - *Usage*: Displaying progress bars for loops and other operations.
- **sklearn.ensemble.AdaBoostClassifier**
  - *Purpose*: An AdaBoost classifier.
  - *Usage*: Building and training an AdaBoost classification model.
- **sklearn.ensemble.GradientBoostingClassifier**
  - *Purpose*: Gradient Boosting for classification.
  - *Usage*: Building and training a Gradient Boosting classification model.
- **sklearn.ensemble.VotingClassifier**
  - *Purpose*: Soft Voting/Majority Rule classifier.
  - *Usage*: Combining multiple classification models.
- **sklearn.model_selection.StratifiedKFold**
  - *Purpose*: Stratified K-Folds cross-validator.
  - *Usage*: Creating K-Folds cross-validator object with stratified folds.
- **sklearn.feature_selection.SequentialFeatureSelector**
  - *Purpose*: Transformer that performs sequential feature selection.
  - *Usage*: Selecting features based on sequential forward or backward selection.
- **sklearn.metrics.accuracy_score**
  - *Purpose*: Accuracy classification score.
  - *Usage*: Computing the accuracy of the classification model.