Predicting diabetes using classification algorithms

1st Bui Quang Phu
Department of Computer Science
University of Information and Technology
20520273@gm.uit.edu.vn

2nd Nguyen Hong Anh Thu
Department of Computer Science
University of Information and Technology
20520313@gm.uit.edu.vn

3rd Truong Thi Thanh Thanh
Department of Computer Science
University of Information and Technology
20520767@gm.uit.edu.vn

4th Nguyen Minh Thuan
Department of Computer Science
University of Information and Technology
20520795@gm.uit.edu.vn

***Abstract*—Nowadays, diabetes disease is becoming a global public health issue. It is important to know who is at a high risk of developing diabetes so that they can receive the right treatment. By implementing classification algorithms on diabetes-related data, we can determine whether a person is at high risk or not, thus identifying all health issues of the patient. This enables necessary changes to be applied to improve the quality of healthcare. This study focused on machine learning techniques such as Support Vector Machine, Logistic Regression, Random Forest and Decision Tree. By using a diabetes dataset that includes medical and demographic data such as age, gender, body mass index (BMI),... from the patient data in hospital. The final results were compared and evaluated to identify the most effective model based on various evaluation criteria. The experimental results show that Random Forest algorithm outperformed than three other algorithms in terms of the accuracy, precision, recall and f1-score with 91% on the diabetes classification task. This application can be a valuable tool to identify patient's health issues.**

1. INTRODUCTION

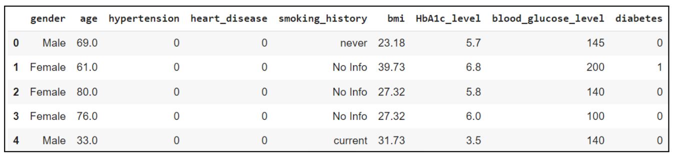
Diabetes is a chronic disease associated with an imbalance in blood sugar levels and nowadays it is becoming a global public health issue. Individuals over the age of 65 as well as those with underlying medical conditions such as cardiovascular disease, diabetes, chronic respiratory disease and cancer which are at a higher risk of developing severe forms of the disease. Previously, diabetes was believed to only affect older adults but now there are signs of a younger age of onset. Diagnosing and treating diabetes early is crucial to provide the best treatment methods and interventions.

For saving costs and time for doctor, we develop a machine learning model to detect diabetes based on basic personal information such as gender, age, BMI or symptoms like hypertension and heart disease can be beneficial. This can help determine whether an individual is likely to have diabetes and needs to seek medical attention. It saves time for individuals and allows for more efficient healthcare management. By analyzing the dataset and building models, we can gain a better understanding of the relationship between risk factors and diabetes. This provides valuable information for preventive measures and improved healthcare for individuals at risk of developing diabetes.

The primary purpose of this research is create a machine learning model that can predict if a diabetes patient is at high risk or not based on the patient's medical and demographic data. We are going to use four classification algorithms including Decision Tree, Random Forest, Logistic Regression and Support Vector Machine for this research. Nguyen Hong Anh Thu will be responsible for Decision Tree algorithm, Nguyen Minh Thuan will do Random Forest algorithm, Bui Quang Phu will take responsibility for logistic Regression algorithm and Truong Thi Thanh Thanh will handle Support Vector Machine.

2. DATASET

The dataset we use in this research is **Diabetes Prediction Dataset**. We had find the dataset on Kaggle. This dataset contains medical and demographic data of patients with along with their diabetes status whether positive or negative. It consist of various features as age, gender, body mass index (BMI), hypertension, heart disease, smoking history, HbA1c level and blood glucose level. The target when using the model is to predict accuracy for as many people as possible who have diabetes.



	gender	age	hypertension	heart_disease	smoking_history	bmi	HbA1c_level	blood_glucose_level	diabetes
0	Male	69.0	0	0	never	23.18	5.7	145	0
1	Female	61.0	0	0	No Info	39.73	6.8	200	1
2	Female	80.0	0	0	No Info	27.32	5.8	140	0
3	Female	76.0	0	0	No Info	27.32	6.0	100	0
4	Male	33.0	0	0	current	31.73	3.5	140	0

Fig. 1: One small part of dataset

The raw dataset includes 20.000 samples of data. The target "diabetes" with label 0 indicates "no" mean that patients not get diabetes, label 1 indicates "yes" mean that patients get diabetes.

Feature	Meaning
gender	Biological sex of the individuals
age	Age of one patient
hypertension	High blood pressure mean blood pressure higher than normal
heart_diasease	Types of heart condition
smoking_history	How often of people are smoking
bmi	Value derived from the mass and height of a person
HbA1c_level	A measure of a person's average blood sugar level over the past 2-3 months
blood_glucose_level	The amount of glucose in blood
diabetes	Target variable being predicted

TABLE I: Meaning Features

3. EXPLORATORY DATA ANALYSIS

3.1. Diabetes

The primary objective of this dataset is to classify people get diabetes or not. The plot above displays the distribution of each label in the target variable "diabetes". We can see that the dataset is lopsided in label 0. From this, we can infer that the dataset is slightly unbalanced between label 0 and 1 which poses a challenge for the model when predicting the target.

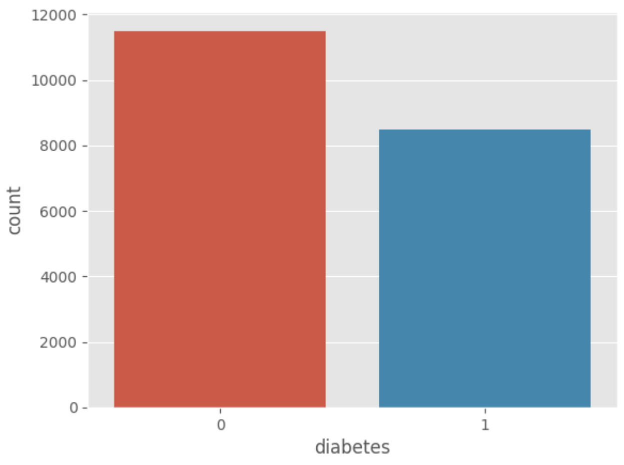


Fig. 2: Diabetes Amounts

3.2. Gender

We can see from plots 1 and 2 that the number of females is higher than males in both the overall dataset and among those with diabetes. This might make us think that females are more likely to have diabetes than males. However, when we look at plot 3, we find that the prevalence of diabetes (the proportion of individuals with diabetes within each gender group) is higher among males. In general, males are more prone to diabetes than females but this difference doesn't seem to help much in distinguishing individuals with diabetes based on their gender.

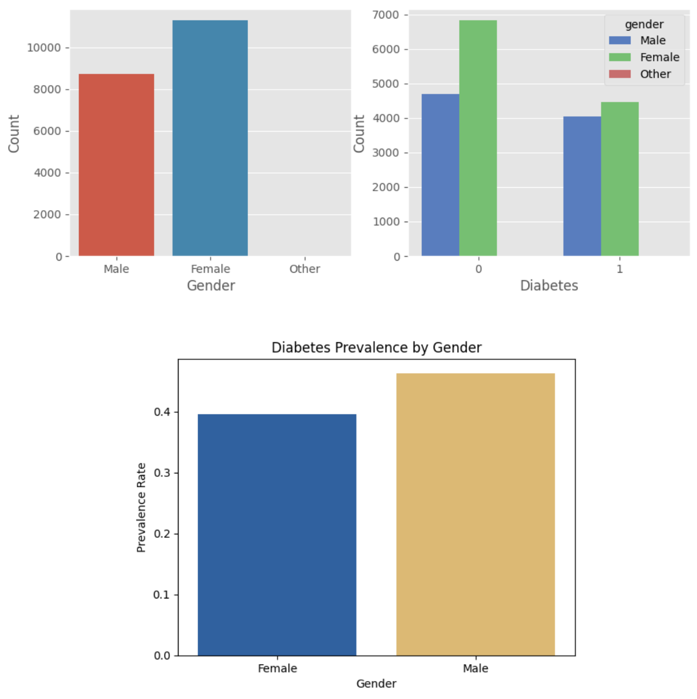


Fig. 3: Gender Categories

3.3. Age

The first plot in Figure 4 shows the distribution of ages in the dataset. We can see that it includes people of different ages, ranging from young to old (from one year old to over 80 years old). However, there is a higher number of individuals surveyed around the ages of 50 to 70. To understand the age better, we divided them into six categories: infants, childs, teenage, adults (between 18 and 40 years old), middle-age(between 40 and 60 years old) and lastly is older adults (more than 60 years old). Looking details at the plot count of age categories, the group has most count is older adults.

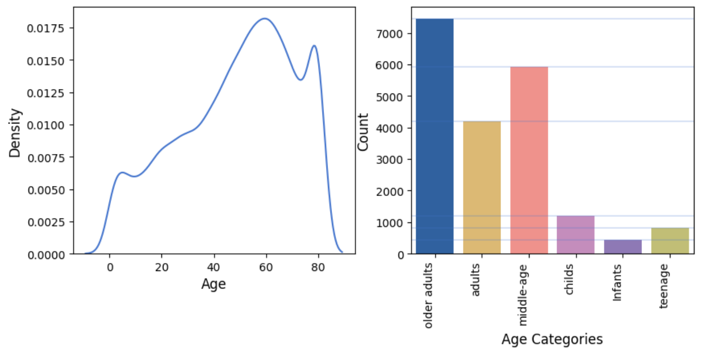


Fig. 4: Age Density and Age Categories

Looking at the percentage and prevalence of different age categories get diabetes in Fig 5, we can observe that a majority of people get diabetes are above 40 years old. Additionally, the data suggests that infants are not prone to diabetes, while teenagers and children have a lower likelihood of developing the condition. This indicates that the risk of diabetes tends to increase the age with older individuals having a higher percentage of being affected. The final plot in Fig 5 further confirms this trend. Due to this observed trend, the age feature appears to be highly valuable in classifying diabetes.

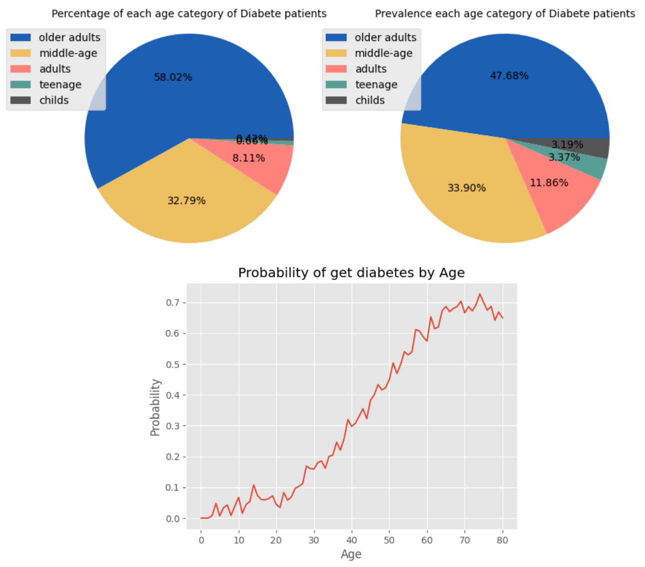


Fig. 5: Age and Diabetes

3.4. Hypertension

The first plot in Fig 6 shows that there are many instances with hypertension value 0 in the dataset, while the occurrences of hypertension value 1 are relatively rare. However, when we examine plots 2 and 3, we can find that around 75% of individuals with hypertension value 1 also have diabetes. This indicates that individuals with hypertension value 1 have a higher likelihood of developing diabetes compared to those with hypertension value 0, even though they are less common. Nevertheless, due to the small number of individuals with hypertension value 1, its role in classifying diabetes is limited to a specific group.

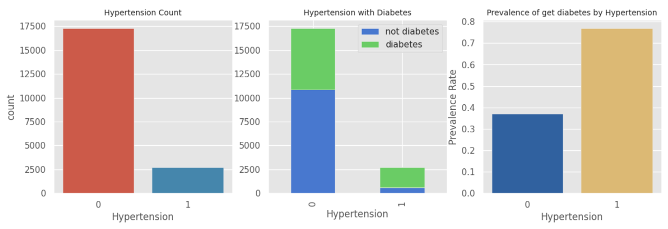


Fig. 6: Hypertension and Diabetes

3.5. Heart Disease

When we look at the plots in Fig 7, we notice that they are quite similar to the hypertension plot. However, heart disease presents a more severe situation compared to hypertension. There is a greater imbalance in the distribution of the "0" values for heart disease compared to hypertension. Additionally, heart disease value "1" is associated with a higher risk of developing diabetes, but it is relatively uncommon, with only around 1600 samples out of the total 20000 samples. Consequently, heart disease only offers meaningful insights for classifying diabetes within a specific subgroup.

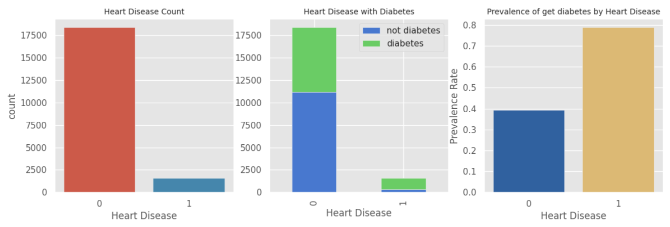


Fig. 7: Heart Disease and Diabetes

3.6. Smoking history

Looking at the first two plots in Fig 8, we can see that the "never" and "no info" smoking history types are the most common in the dataset. However, the "no info" type is particularly problematic for classification because we have no information about it. It could potentially belong to another type or even be a type not included in the dataset. Dealing with this type poses a significant challenge for the model.

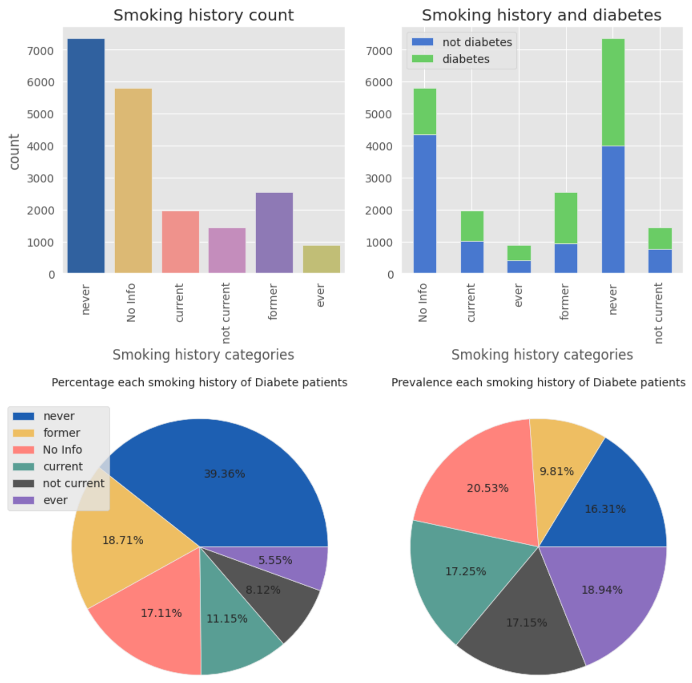


Fig. 8: Smoking and diabetes

Additionally, the two pie plots show that although the smoking history types have different frequencies, their probabilities are relatively similar. This means that the different types have comparable chances of being associated with diabetes. Consequently, smoking history does not provide valuable information for classifying diabetes.

3.7. BMI

The first plot in Fig 9 shows that most people in the dataset have a BMI ranging from 20 to 40. To better understand BMI, we divided the values into four categories: “Underweight” (BMI less than 18.5), “Healthy weight” (BMI between 18.5 and 25), “Overweight” (BMI between 25 and 30), and “Obesity” (BMI greater than 30). The second plot in Fig 9 displays the distribution of individuals in each category. We can observe that the majority of people surveyed fall into the “Overweight” and “Obesity” categories.

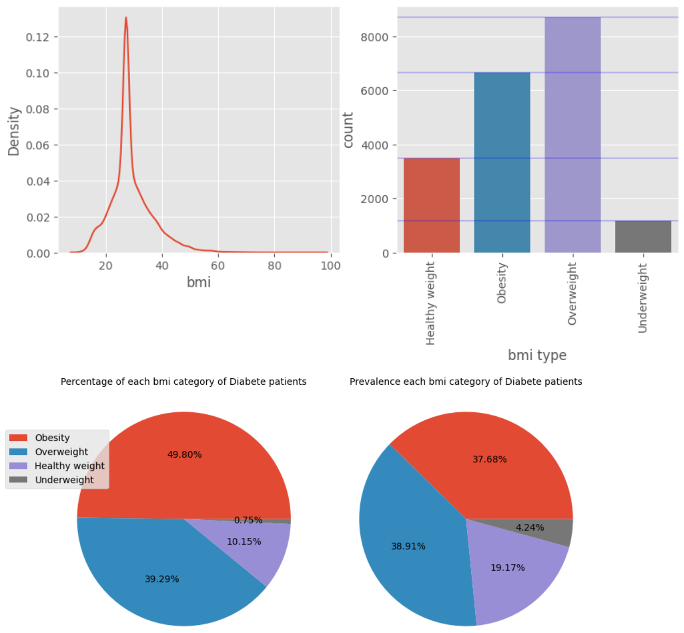


Fig. 9: BMI and diabetes

As BMI is calculated using a person’s height and weight, the third and fourth plots demonstrate that individuals with higher BMI values are more susceptible to diabetes. However, it is important to note that even individuals categorized as “Healthy weight” have a 19% chance of developing diabetes, indicating the influence of other factors. In summary, the plots indicate a relationship between higher BMI values and a greater risk of diabetes, with a significant number of individuals falling into the “Overweight” and “Obesity” categories.

3.8. HbA1c and Blood glucose Level

In reality, it is common for people with diabetes to have high levels of both blood glucose and HbA1c. Therefore, we aim to plot these two features together to confirm this trend. In Fig 10,we observe that the majority of values for both HbA1c and blood glucose level fall within around the middle range.

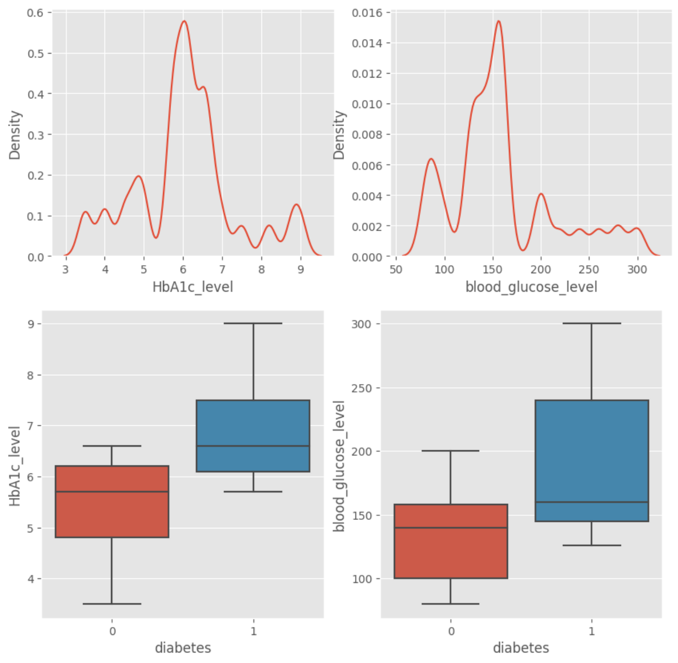


Fig. 10: HbA1c and blood glucose level

The box plots clearly separate diabetes cases and non-diabetes cases based on HbA1c and blood glucose levels. There is only a small overlap between the two boxes, with HbA1c showing a slightly smaller intersection. This indicates that both features are highly useful in classifying diabetes, and using either one alone would result in only a few misclassifications.

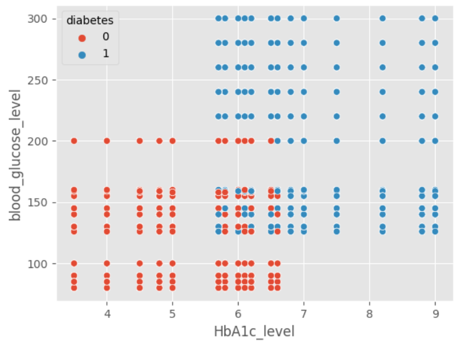


Fig. 11: HbA1c level combine with blood glucose level

However, when we combine these features as shown in Fig 11, we see that they successfully divide diabetes cases into two distinct areas. This significantly improves the model’s ability to make accurate predictions. However, there is a small overlapping region that presents a challenge. Since we are using all the features to train the model, this region has the potential to cause over-fitting if the model attempts to fit it too precisely.

4. METHODS

A classifier is a machine learning model that helps identify and differentiate various objects based on specific features. In this report, we will use four classifiers are Logistic Regression, Support Vector Machine, Decision Tree and Random Forest to make prediction for diabetes.

4.1. Logistic Regression

1) Definition

Logistic regression is a statistical modeling technique used to predict the probability of a **binary outcome** or to classify data into discrete categories. It is a **supervised learning** algorithm that is particularly well-suited for solving classification problems when the dependent variable or response variable is binary or dichotomous.

In logistic regression, the relationship between the independent variables (also known as predictors, features, or input variables) and the binary outcome variable is modeled using the **logistic function** or **sigmoid function**. The logistic function maps any real-valued input to a value between 0 and 1 representing the probability of the outcome occurring.

The logistic regression model estimates the coefficients or weights associated with each independent variable, indicating the strength and direction of their influence on the probability of the binary outcome. These coefficients are obtained using maximum likelihood estimation or other optimization techniques.

Once the model is trained, it can be used to predict the probability of the binary outcome for new observations or classify them into the respective categories based on a predetermined threshold. Logistic regression is widely used in various fields such as medicine, economics, social sciences and marketing, where binary classification tasks are prevalent, such as predicting disease presence, customer churn or loan default.

2) Sigmoid Function

The sigmoid function, also known as the logistic function, is a mathematical function that is used in logistic regression to map the linear combination of the features and their associated weights to a probability value between 0 and 1. The sigmoid function converts the log-odds (logit) into a probability. The sigmoid function is defined as follows:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Where $\sigma(z)$ is the sigmoid function and z is the input to the function. In logistic regression, z represents the linear combination of the feature values and their associated weights:

$$z = w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

Here w_0 is the intercept term $w_1, w_2, ..., w_n$ are the coefficients associated with the features $x_1, x_2, ..., x_n$. The sigmoid function takes the input z and computes the probability p that an instance belongs to the positive class. The output of

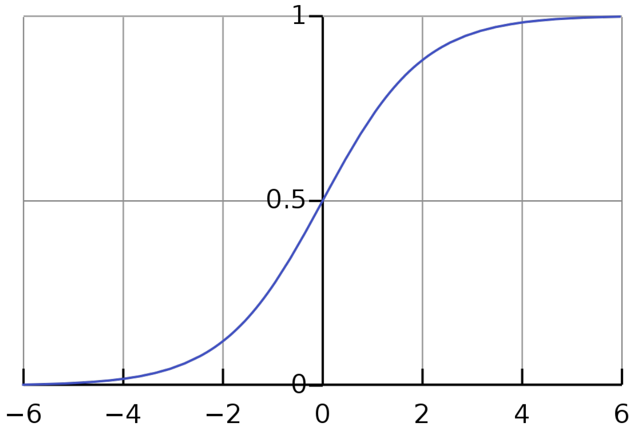


Fig. 12: Sigmoid Function

the sigmoid function is always between 0 and 1. This allowing it to represent a valid probability.

Base on the Sigmoid Function equation:

$$\sigma(x) = \frac{1}{1-e^{-x}} \tag{1}$$

We have:

$$lim_{x\to+\infty}\sigma(x) = lim_{x\to+\infty}\frac{1}{1-e^{-x}} = 1 \tag{2}$$

and

$$lim_{x\to-\infty}\sigma(x) = lim_{x\to-\infty}\frac{1}{1-e^{-x}} = 0 \tag{3}$$

From equation (2) and (3), we can obtain:

$$0 \leqslant \frac{1}{1-e^{-x}} \leqslant 1 \tag{4}$$

When z is positive or large, the sigmoid function approaches 1, indicating a high probability of the positive outcome. Conversely, when z is negative or small, the sigmoid function approaches 0, indicating a low probability of the positive outcome. When z is close to 0, the sigmoid function returns a probability of 0.5, indicating an equal likelihood of both outcomes.

The sigmoid function plays a crucial role in logistic regression as it transforms the linear combination of features and weights into a probability that can be used for classification. By applying a threshold (usually 0.5) the probability can be converted into class labels (0 or 1) allowing for binary classification based on the predicted probabilities.

3) Gradient Descent in Logistic Regression

Gradient Descent is an optimization algorithm commonly used to train logistic regression models. Logistic regression is a popular machine learning algorithm used for binary classification, which predicts the probability of a binary outcome based on the values of independent variables.

In logistic regression, the goal is to find the optimal values for the coefficients that minimize the error between the predicted probabilities and the actual binary outcomes. This optimization problem is solved using Gradient Descent.

The algorithm starts by initializing the coefficients to some initial values. Then, it iteratively updates the coefficients by taking steps proportional to the negative gradient of a cost function. The gradient represents the direction of steepest ascent, so taking the negative gradient allows the algorithm to move in the direction of steepest descent, gradually reducing the cost function.

The cost function in logistic regression is typically the log loss or cross-entropy loss function. It quantifies the error between the predicted probabilities and the actual binary outcomes. By minimizing this cost function, the logistic regression model can make more accurate predictions.

During each iteration of Gradient Descent, the coefficients are updated using the following equation:

$$\text{new\_coefficient} = \text{old\_coefficient} - \text{learning\_rate} \times \text{gradient}$$

Here, the learning rate determines the step size in each iteration and the gradient represents the derivative of the cost function with respect to the coefficients. The learning rate is a hyperparameter that needs to be carefully chosen to ensure convergence and avoid overshooting or slow convergence. The algorithm continues to update the coefficients iteratively until it reaches convergence or a maximum number of iterations. Convergence is typically determined when the change in the cost function or the coefficients falls below a certain threshold.

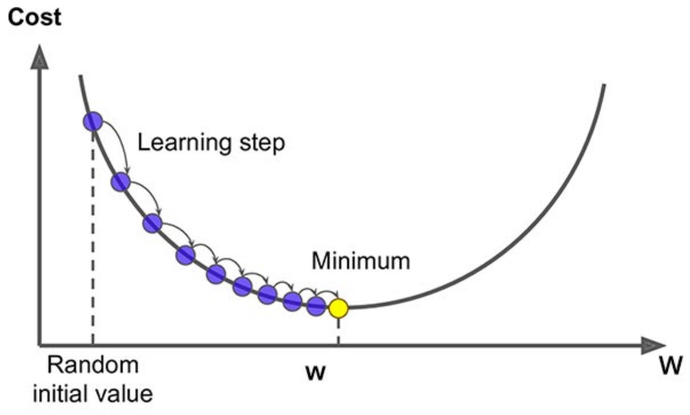


Fig. 13: Gradient descent

By using Gradient Descent, logistic regression can find the optimal values for the coefficients that minimize the cost function and improve the model's predictive performance. The algorithm is computationally efficient and can handle large datasets, making it suitable for various applications.

However, it's important to note that Gradient Descent is not without its limitations. It can converge to different solutions depending on the initial values of the coefficients and the choice of learning rate is crucial for achieving convergence. If the learning rate is too small, the algorithm will progress very slowly and take a long time to converge. In this case, the model training process will be prolonged and may consume significant computational resources. Conversely, if the learning rate is too large, the algorithm may fail to converge or converge slowly. When the learning rate is too large, the update steps will be too large as well, potentially skipping over the optimal point or oscillating around it without converging. This leads to the model being unable to find the optimal coefficients and not achieving the desired accuracy.

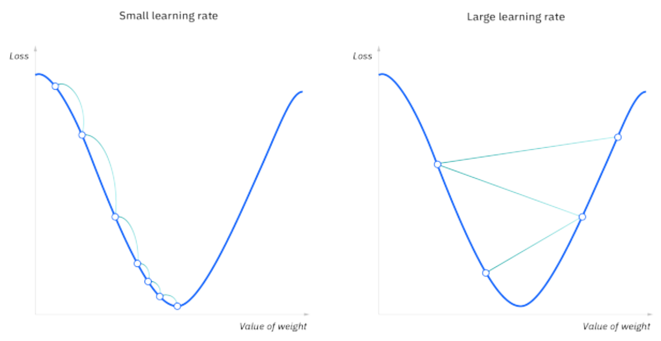


Fig. 14: Gradient descent

Gradient Descent may also get stuck in local optima. Especially in non-convex cost functions.

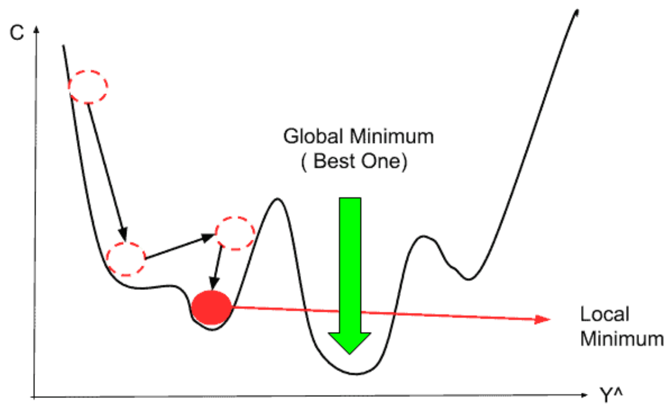


Fig. 15: Gradient descent

4) Model Training

To train a logistic regression model, we start with a dataset containing instances with known binary outcomes and their corresponding feature values. The dataset is divided into two parts: a training set and a validation set.

The model training process involves finding the optimal values for the coefficients or weights that minimize the error between the predicted probabilities and the actual binary outcomes in the training set. This is typically done using an optimization algorithm, such as gradient descent or Newton's method, to iteratively update the coefficients based on the training data.

During training, the model adjusts the coefficients to maximize the likelihood of the observed outcomes given the feature values. The process continues until convergence is achieved, or a maximum number of iterations is reached.

5) Logistic Regression Algorithm

Logistic regression is a statistical algorithm used for binary classification tasks. It models the relationship between a set of independent variables (features) and the probability of a binary outcome. The algorithm assumes a linear relationship between the features and the log-odds of the binary outcome. Here are the basic steps for this algorithm:

Step 1: Initialize the parameters

Initialize the parameters of the logistic regression model, including the weight vector (w) and the bias term (b) or intercept (w0).

Step 2: Build the sigmoid function

Define the sigmoid function to calculate the predicted probabilities. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

where z is the weighted sum of features:

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

Step 3: Define the loss function

Use the loss function to measure the error between the predictions and the actual values. In logistic regression, the commonly used loss function is the cross-entropy loss:

$$J(w,b) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$$

where m is the number of training examples, $y^{(i)}$ is the true label, and $\hat{y}^{(i)}$ is the prediction calculated using the sigmoid function.

Step 4: Optimize the parameters

Use gradient descent to optimize the parameters w and b. Gradient descent is an iterative method that updates the parameters in the direction of the negative gradient (partial derivatives) of the loss function. The parameter update equations are:

$$w := w - \alpha\frac{\partial J(w,b)}{\partial w}$$

$$b := b - \alpha\frac{\partial J(w,b)}{\partial b}$$

where α is the learning rate that determines the step size, and $\frac{\partial J(w,b)}{\partial w}$ and $\frac{\partial J(w,b)}{\partial b}$ are the gradients of the loss function with respect to w and b.

Step 5: Repeat steps 2-4

Iterate the process of building the sigmoid function, calculating the loss function, and optimizing the parameters for a certain number of iterations or until convergence criteria are met (e.g., reaching a certain number of iterations or the loss value doesn't change significantly).

Step 6: Evaluate the model

Evaluate the logistic regression model using metrics such as accuracy, recall, precision on the test set or using cross-validation techniques.

By following these steps, you can construct a logistic regression model using the sigmoid function and gradient descent.

6) Advantages and Disadvantages

Adavantages

- Simplicity: Logistic regression is a relatively simple and interpretable model compared to more complex algorithms.
- Efficiency: The training and prediction process of logistic regression is computationally efficient.
- Probability estimation: Logistic regression provides probability estimates allowing for a probabilistic interpretation of the predictions.
- Feature importance: The coefficients in logistic regression can indicate the importance of each feature in predicting the binary outcome.

Disadvantages

- Linearity assumption: Logistic regression assumes a linear relationship between the independent variables and the log-odds of the binary outcome.
- Lack of flexibility: Logistic regression may not capture complex nonlinear relationships between the features and the binary outcome.
- Sensitivity to outliers: Logistic regression can be sensitive to outliers which may affect the model's performance.
- Independence assumption: Logistic regression assumes that the independent variables are independent of each other.

Despite these limitations, logistic regression remains a popular and widely used algorithm for binary classification tasks due to its simplicity, interpretability and effectiveness in many real-world applications.

7) Hyperparameters

In logistic regression, hyperparameters are the configuration settings that are not learned from the data but are set by the user before training the model. These hyperparameters control the behavior and performance of the logistic regression algorithm. Here are some important hyperparameters in logistic regression:

(a) ***penalty: ["l1", "l2", "elasticnet", None], default = "l2"***
Logistic regression supports two common penalty types: L1 and L2 regularization. L1 regularization encourages sparsity by penalizing the absolute values of the coefficients, while L2 regularization penalizes the squared values of the coefficients. The choice of penalty type depends on the specific problem and the desire for feature selection or shrinkage of coefficients.

(b) ***C : float, default=1.0***
The regularization parameter controls the trade-off between model complexity and the fit to the training data. A smaller value of C increases the regularization strength, leading to a simpler model with potentially higher bias but lower variance. Conversely, a larger value of C decreases the regularization strength, allowing the model to fit the training data more closely but potentially increasing variance.

(c) ***solver : [“lbfgs”, “liblinear”, “newton-cg”, “newton-cholesky”, “sag”, “saga”], default = “lbfgs”***
The solver algorithm determines how the logistic regression model is optimized. Common solver algorithms include “liblinear”, “lbfgs”, “sag” and “newton-cg”. The choice of solver depends on the size of the dataset, the penalty type, and the desired convergence properties. For small datasets, “liblinear” is a good choice, whereas “sag” and “saga” are faster for large ones. For multiclass problems, only “newton-cg”, “sag”, “saga” and “lbfgs” handle multinomial loss. “liblinear” is limited to one-versus-rest schemes. “newton-cholesky” is a good choice for **n_samples** $>>$ **n_features**, especially with one-hot encoded categorical features with rare categories. Note that it is limited to binary classification and the one-versus-rest reduction for multi-class classification. Be aware that the memory usage of this solver has a quadratic dependency on n_features because it explicitly computes the Hessian matrix.

(d) ***max_iter : int, default = 100***
This hyperparameter specifies the maximum number of iterations that the solver algorithm will run before stopping. It ensures that the algorithm terminates even if convergence is not achieved within a certain number of iterations.

(e) ***class_weight : dict or “balanced”, default = None***
Logistic regression allows you to assign different weights to different classes in case of imbalanced dataset. The class_weight hyperparameter allows you to assign higher weights to minority classes to improve their representation in the model.

(f) ***multi_class: [“auto”, “ovr”, “multinomial”], default = “auto”***
Logistic regression can be extended to handle multi-class classification problems. The *multi_class* hyperparameter determines the strategy to use for multi-class classification. Common options include “ovr” (one-vs-rest) and “multinomial” (softmax regression).

4.2. Support Vector Machine

1) Definition

SVM stands for **Support Vector Machine**. It is a **supervised learning** algorithm used for classification and regression tasks. SVM is particularly effective in solving **binary classification** problems, where the goal is to divide a dataset into two classes. However, it can also be extended to handle multi-class classification.

The basic idea behind SVM is to find an optimal hyperplane that separates the classes in the feature space. The hyperplane is defined as the decision boundary that maximally separates the data points of different classes. SVM aims to find the hyperplane that has the maximum margin, which is the distance between the hyperplane and the nearest data points of each class.

In addition to linear separation, SVM can use kernel functions to transform the input space into a higher-dimensional feature space, where the data might be linearly separable. This transformation allows SVM to handle nonlinear classification problems effectively.

During training, SVM learns from a set of labeled examples, called the training set, to determine the optimal hyperplane. The SVM algorithm assigns each data point to a specific class based on which side of the decision boundary it falls on.

SVM has several advantages, including its ability to handle high-dimensional data, resistance to overfitting and effectiveness in handling complex datasets. It is widely used in various domains, such as image classification, text classification, bioinformatics, and finance.

2) Soft Margin

Allowing some margin violations or misclassifications in the training process. It is calculated through the optimization problem formulation of the SVM algorithm.

The goal for solf margin is to find an optimal hyperplane that separates the classes while allowing for a certain degree of misclassification. This is useful when the data is not perfectly separable or when there is some degree of noise or outliers in the dataset.

The objective function of the soft margin SVM is formulated as a trade-off between maximizing the margin and minimizing the misclassifications.

$$(w, b, \xi) = \underset{w,b,\xi}{agrmin} \frac{1}{2} * \|w\|^2 + C * \sum_{n=1}^{N} \xi_n$$

subject to:

$$y_n \left(w^T x_n + b\right) \geq 1 - \xi_n, \forall n = 1, 2, ..., N$$

$$\xi_n \geq 0, \forall n = 1, 2, ..., N$$

- $\|w\|$: is the norm of the weight vector w
- C: is the hyperparameter that controls the trade-off between margin maximization and misclassification penalty
- ξ_n: is the slack variable represents the amount by which a data point can violate the margin or be misclassified.
- y_n: is the class label of the it training sample
- x_n: is the feature vector of the it training sample
- b: is the bias term

3) Kernel Trick

Kernel functions play an important role in Support Vector Machines (SVM). They are used to transform the input data into a higher-dimensional feature space, allowing SVM to effectively handle nonlinear classification problems. Here are some commonly used kernel functions:

a) Linear Kernel

This is the most basic type of kernel, usually one-way. It is proven to be the best function when it has a lot of features. Linear kernel is mainly preferred for text classification problems. Since most of these classification problems can be linearly decomposed.

$$K(x, z) = x^T z$$

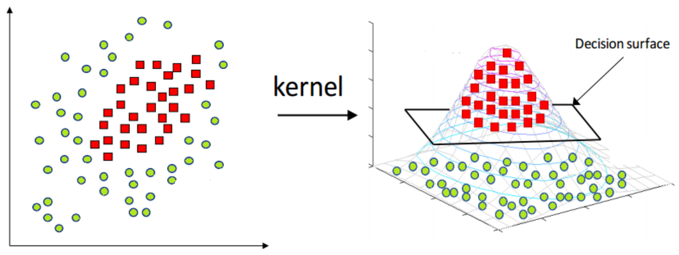


Fig. 16: Transform the input space into a higher-dimensional feature space

- $K(x, z)$: represents the linear kernel value between two feature vectors x and z
- x, z: feature vectors

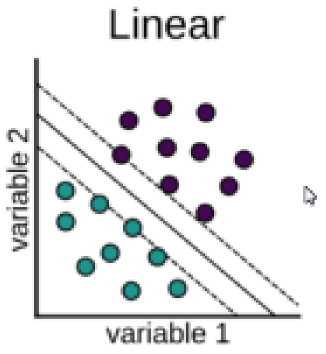


Fig. 17: Linear kernel

b) Polynomial Kernel

The polynomial kernel is a more general representation of the linear kernel. Represents the nonlinear relationship between the feature vectors by calculating the polynomial of the dot product and the parameters γ, r, d. The degree of the polynomial is determined by the parameter d. The polynomial kernel function is suitable for processing data with a nonlinear distribution.

$$K(x, z) = (\gamma x^T z + r)^d$$

- $K(x, z)$: represents the linear kernel value between two feature vectors x and z
- x, z: feature vectors
- γ: is the kernel coefficient
- r(coef0): independent term
- d: degree of the polynomial

c) Gaussian RBF (Radial Basis Function) Kernel

Being one of the preferred and most commonly used kernel functions in SVM. It is often chosen for non-linear data. It measures the similarity between two feature vectors based on the Euclidean distance and the parameter γ. It creates a non-linear feature space where nearby data points have large kernel values and distant data points have small kernel values. It is typically used when the characteristics of the data are unknown or when the data distribution is uneven.

$$K(x, z) = exp(-\gamma \|x - z\|^2)$$

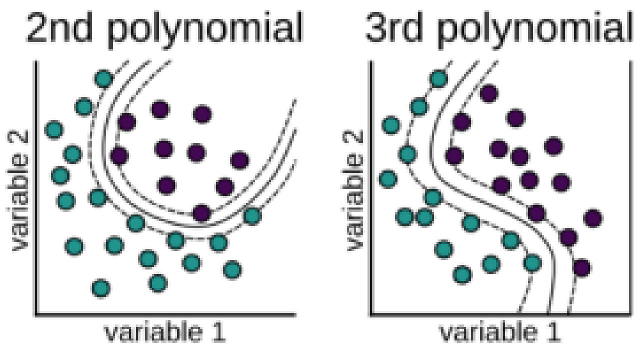


Fig. 18: Polynomial kernel

- $K(x, z)$: represents the linear kernel value between two feature vectors x and z
- x, z: feature vectors
- γ: scaling factor of the input data

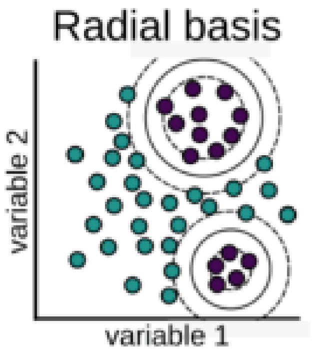


Fig. 19: Gaussian RBF (Radial Basis Function) kernel

d) Sigmoid Kernel

The Sigmoid kernel function represents a non-linear relationship between feature vectors using the hyperbolic tangent function (tanh) and parameters γ, r. The Sigmoid kernel function is often used for classifying non-linear data. However, it is less commonly used compared to other kernel functions because it can lead to unstable results in certain cases.

$$K(x, z) = tanh(\gamma x^T z + r)$$

- $K(x, z)$: represents the linear kernel value between two feature vectors x and z
- x, z: feature vectors
- γ: is the kernel coefficient
- r(coef0): independent term

4) Advantages and Disadvantages

Advantages

- SVM works better when the data is Linear
- It is more effective in high dimensions
- With the help of the kernel trick, we can solve any complex problem
- SVM is not sensitive to outliers
- Can help us with Image classification

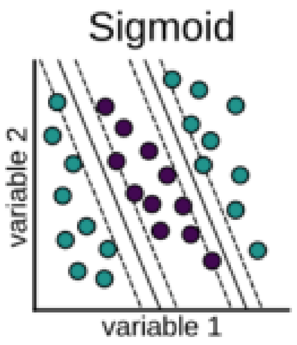


Fig. 20: Sigmoid kernel

Disadvantages

- Choosing a good kernel is not easy
- It doesn't show good results on a big dataset
- The SVM hyperparameters are Cost -C and gamma. It is not that easy to fine-tune these hyper-parameters. It is hard to visualize their impact

5) Hyperparameter

Hyperparameters are parameters that are set before the learning process begins and affect the behavior of the SVM algorithm. These hyperparameters help control the complexity of the SVM model and its generalization ability. Here are some commonly used hyperparameters in SVM:

(a) ***C : float, default=1.0***
Determines the trade-off between maximizing the margin and minimizing the training error. It controls the regularization strength of the SVM model. A smaller value of C leads to a larger margin but allows for more training errors, while a larger value of C results in a smaller margin but fewer training errors.

(b) ***kernel : ["linear", "poly", "rbf", "sigmoid", "precomputed"] or callable, default = "rbf"***
Determines the type of kernel function to be used for mapping the data into a higher-dimensional feature space. The choice of kernel function can significantly impact the performance of the SVM model.

(c) ***degree : int, default = 3***
Determines degree of the polynomial kernel function ("poly"). Must be non-negative. Ignored by all other kernels. used to map the input data into a higher-dimensional feature space. The degree parameter controls the complexity and flexibility of the decision boundaries that the SVM model can learn.

(d) ***gamma : "scale", "auto" or float, default = "scale"***
Used in kernel functions such as "RBF", "poly" and "sigmoid". controls the influence of each training example in the SVM model. It affects the shape of the decision boundary and the flexibility of the model. The choice of the "gamma" value is crucial as it can significantly impact the SVM's performance.

(e) ***coef0 : float, default=0.0***
Constant term that affects the degree of non-linearity in certain kernel functions, such as the "poly" and "sigmoid" kernels. It is used to adjust the influence of higher-order terms in the decision function.

(f) ***shrinking : bool, default=True***
Refers to a technique used to speed up the training process and improve the computational efficiency of the model. It controls whether or not the shrinking heuristic is applied during the training algorithm. The shrinking heuristic helps reduce the number of support vectors and speeds up the training process.

(g) ***probability : bool, default=False***
Refers to whether the SVM model should be trained to output class probabilities in addition to making class predictions. It allows the SVM algorithm to estimate the probabilities of each class assignment for a given data point.

(h) ***tol : float, default=1e-3***
Stands for tolerance and refers to the stopping criterion for the SVM training algorithm. It determines the level of tolerance or accuracy that is considered acceptable during the optimization process.

(i) ***cache_size : float, default=200***
Refers to the amount of memory allocated for caching the computed kernel matrix during the training process. The cache size affects the speed and memory usage of the SVM algorithm.

(j) ***class_weight : dict or "balanced", default=None***
Allows you to assign different weights to the classes in the SVM training process. It is used to address class imbalance or to emphasize the importance of certain classes in the model's training and decision-making process.

(k) ***verbose : bool, default=False***
Controls the amount of information displayed during the training process.

(l) ***max_iter : int, default=-1***
Stands for maximum number of iterations and specifies the maximum number of iterations or epochs allowed for the SVM training algorithm to converge.

(m) ***decision_function_shape : "ovo", "ovr", default="ovr"***
Determines the shape of the decision function used in SVM classification.
Set to **"ovr" (one-vs-rest)**, a binary classification problem is created for each class, where one class is treated as positive and the rest as negative. The final decision is based on the highest confidence score obtained among all the binary classifiers.
Set to **"ovo" (one-vs-one)**, a binary classification problem is created for each pair of classes. The final decision is made through a voting scheme where the class that receives the most votes from the binary classifiers is selected.

(n) ***break_ties : bool, default=False***
Used in SVM classification to determine the strategy when there is a tie in the decision function values for predicting class labels.

Set to True, it enables the decision function to break ties by choosing the class with the highest confidence score. This can be useful when dealing with imbalanced datasets or when there are cases of equal decision function values. Set to False, ties are not broken explicitly, and the class with the lower index is selected as the predicted label in case of a tie.

(o) ***random_state : int, RandomState instance or None, default=None***
Set the random seed for reproducibility of the results.
By setting a specific value for *"random_state"*, maybe ensure that the random number generation during the SVM training process is deterministic and will produce the same results each time you run the code with the same value of *"random_state"*.
This parameter is particularly useful when want to reproduce the same results for experimentation, debugging, or comparison purposes.

4.3. Decision Tree

1) Definition

Decision Tree is one of the most powerful tools of supervised learning algorithms used for both classification and regression tasks. It builds a flowchart-like tree structure where each internal node denotes a test on an attribute, each branch represents an outcome of the test and each leaf node (terminal node) holds a class label. It is constructed by recursively splitting the training data into subsets based on the values of the attributes until a stopping criterion is met such as the maximum depth of the tree or the minimum number of samples required to split a node. During training, the Decision Tree algorithm selects the best attribute to split the data based on a metric such as entropy or Gini impurity, which measures the level of impurity or randomness in the subsets.

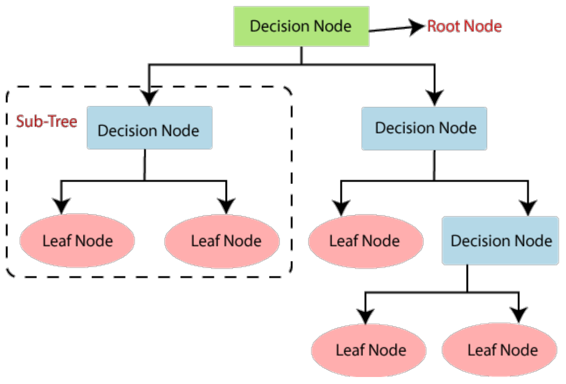


Fig. 21: Decision Tree Structure

The goal of using a Decision Tree is to create a training model that can use to predict the class or value of the target variable by **learning simple decision rules** inferred from prior data(training data). In Decision Trees, for predicting a class label for a record we start from the root of the tree. We compare the values of the root attribute with the record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

2) Classification and Regression Tree (CART)

To build the Decision Tree, CART algorithm is used. It works by selecting the best split at each node based on metrics like Gini impurity or information Gain. In order to create a decision tree. Here are the basic steps of the CART algorithm:

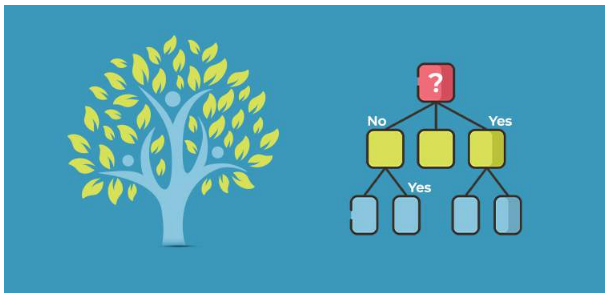


Fig. 22: CART Algorithm

Step 1: The root node of the tree is supposed to be the complete training dataset.

Step 2: Determine the impurity of the data based on each feature present in the dataset. Impurity can be measured using metrics like the Gini index or entropy.

Step 3: Then selects the feature that results in the highest information gain or impurity reduction when splitting the data.

Step 4: For each possible value of the selected feature, split the dataset into two subsets, one where the feature takes on that value, and another where it does not. The split should be designed to create subsets that are as pure as possible with respect to the target variable.

Step 5: Based on the target variable, determine the impurity of each resulting subset.

For each subset, repeat steps 2–5 iteratively until a stopping condition is met. For example, the stopping condition could be a maximum tree depth, a minimum number of samples required to make a split or a minimum impurity threshold. After that, assign the majority class label for classification tasks for each terminal node (leaf node) in the tree.

3) Attribute Selection Measures

Attribute selection measure (ASM) is a criterion used in decision tree algorithms to evaluate the usefulness of different attributes for splitting a dataset. The goal of ASM is to calculate values for every attribute. The values are sorted, and attributes are placed in the tree by following the order. The attribute with a high value(in case of information gain) is placed at the root.

While using Information Gain as a criterion, we assume attributes to be categorical and for the Gini index, attributes are assumed to be continuous.

a) Entropy

Entropy is the measure of the degree of randomness or uncertainty in the dataset which measures the impurity of the sample values. The higher the Entropy, the lower will be the purity and the higher will be the impurity. The formula for Entropy is shown below:

$$E(S) = -p_{(+)} log_2 p_{(+)} - p_{(-)} log_2 p_{(-)}$$

- S: The subset of the training example
- $p_{(+)}$: The probability of positive class
- $p_{(-)}$: The probability of negative class

As we know, the goal of machine learning is to decrease the uncertainty or impurity in the dataset. By using the entropy we are getting the impurity of a particular node, we don't know if the parent entropy or the entropy of a particular node has decreased or not.

For this, we bring a new metric called "Information gain" which tells us how much the parent entropy has decreased after splitting it with some feature.

b) Information Gain

Information gain is a statistical property that measures how well a given attribute separates the training examples according to their target classification. Constructing a decision tree is all about finding an attribute that returns the highest information gain and the smallest entropy. The higher the information gain, the more valuable the feature is in predicting the target variable.

It computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values:

$$\textit{Information Gain} = E(Y) - E(Y|X)$$

The attribute that maximizes information gain is chosen as the splitting criterion for building the decision tree. Hence, select the feature with the highest score as the root node and do the same for sub-nodes.

c) Gini index

It is a measure of purity or impurity while creating a decision tree. It is calculated by subtracting the sum of the squared probabilities of each class from one. CART uses the Gini index as an attribute selection measure to select the best attribute/feature to split. The attribute with a lower Gini index is used as the best attribute to split.

$$\textit{Gini index} = 1 - \sum_{j=1} p_j^2$$

d) Gain ratio

Information gain is biased towards choosing attributes with a large number of values as root nodes. It means it prefers the attribute with a large number of distinct values. Gain ratio overcomes the problem with information gain by taking into account the number of branches that would result before making the split. It corrects information gain by taking the intrinsic information of a split into account.

$$\textit{Gain ratio} = \frac{E(Y)-E(Y|X)}{E(X)}$$

4) Pruning Tree

Decision trees that are trained on any training data run the risk of overfitting the training data. Sometimes it looks like the tree memorized the training data set. If there is no limit set on a decision tree, it will give you 100% accuracy on the training data set because in the worse case it will end up making 1 leaf for each observation. Thus this affects the accuracy when predicting samples that are not part of the training set.

One of the techniques you can use to reduce overfitting in Decision Trees is Pruning. The aim of Decision Tree Pruning is to construct an algorithm that will perform worse on training data but will generalize better on test data. There are two types of pruning: Pre-pruning and Post-pruning.

The pre-pruning technique of Decision Trees is using hyperparameter tuning. We can set the maximum depth of our decision tree using the ***max_depth*** parameter or set the minimum number of samples for each spilt through ***min_sample_split***. And there are more hyperparameters for tuning will be discuss in the next section.

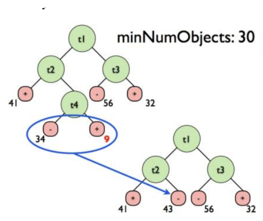


Fig. 23: Pre-Pruning

Post-pruning does the opposite of pre-pruning and allows the Decision Tree model to grow to its full depth. **Post-pruning** is the process splitting in fully grown trees until the stopping criteria are reached.

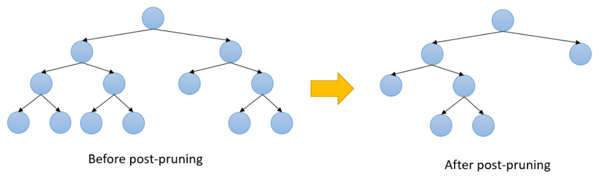


Fig. 24: Post-Pruning

The simplest technique is to prune out portions of the tree that result in the least information gain. This procedure does not require any additional data, and only bases the pruning on the information that is already computed when the tree is being built from training data.

5) Advantage and Disadvantage

Advantages

- Scalability
- Identify feature importance
- Able to handle various data type
- Visual representations is easily understand and consume
- Able to handle both continuous and categorical variables

Disadvantages

- Prone to overfitting
- High variance estimators
- Expensive computation to train
- Represent complex relationship between variables

6) Hyperparameters

In Decision Tree algorithm, hyperparameters are adjustable settings or configurations that influence the learning process and behavior of the decision tree model. Here are some commonly used hyperparameters for Decision Tree:

(a) ***criterion: ["gini", "entropy", "log_loss"], default = "gini"***
The function to measure the quality of a split at each node. The criterion affects how the decision tree selects the best feature to split on. Common options include "gini" for the Gini impurity and "entropy" for information gain. While using Information Gain as a criterion, we assume attributes to be categorical, and for the Gini index, attributes are assumed to be continuous.

(b) ***max_depth : int, default = None***
It defines the maximum depth or levels of the decision tree. The deeper the tree, the more splits it has and it captures more information about the data. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

(c) ***min_samples_split : int or float, default = 2***
Represents the minimum number of samples required to split an internal node. By default, the decision tree tries to split every node that has two or more rows of data inside it. In case of a value, number of samples at a node is below this threshold, the splitting process stops. Increasing this value can prevent overfitting.

(d) ***min_samples_leaf : int or float, default = 1***
The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. The more you increase the number, the more is the possibility of overfitting.

(e) ***max_features : int, float or ["auto", "sqrt", "log2"], default = None***
max_features represents the number of features to consider when looking for the best split. The higher the max feature value, the more accurate it is, but in return it takes more time.

(f) ***max_leaf_nodes : int, default = None***
This hyperparameter define the number of leaf nodes in a decision tree. It will allow the branches of a tree to have varying depths, another way to control the model's complexity. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

(g) ***min_impurity_decrease : float, default = 0.0***
This argument is used to supervise the threshold for splitting nodes. A split will only take place if it reduces the Gini Impurity, greater than or equal to the min_impurity_decrease value. Its default value is 0, and we can modify it to decrease over-fitting.

(h) ***ccp_alpha : non-negative float, default = 0.0***
Complexity parameter used for Minimal Cost-Complexity Pruning. Greater values of ccp_alpha increase the number of nodes pruned. By default, no pruning is performed. When ccp_alpha is set to zero, the tree overfits, leading to a 100% training accuracy. As alpha increases, more of the tree is pruned, thus creating a decision tree that generalizes better.

4.4. Random Forest

1) Definition

Random Forest is a machine learning algorithm that combines multiple outputs of large or small amount of decision trees which called **estimators** to obtain a final single result. Like a decision tree, it can be used for a variety of tasks including classification and regression.

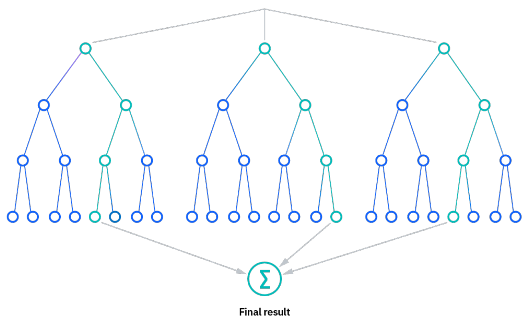


Fig. 25: Random Forest

Why don't we use the decision tree instead of using random forest? Because decision trees have a disadvantage that they can be prone to problems such as bias and overfitting. However, in the random forest algorithm, when multiple decision trees come together as a **ensemble** form , they provide more accurate predictions, especially when the individual trees are not strongly related to each other. The low correlation between decision tree is the key, while some trees can be wrong, many others tree will be right, allowing the group of trees to collectively move in the correct direction.

2) Ensemble methods

Ensemble methods is a machine learning technique that combines multiple base models to produce a single optimal predictive model. Example is Random forest is a combines of multiple decision tree. The most well-known ensemble methods are bagging, also known as bootstrap aggregation, and boosting.

Random Forest algorithm is an extension of bagging method. Then what is bagging method? **Bagging method** will choose random sample of data in training data and can be replacement, meaning that is can be chosen more than once. After several data are generalized, these models are trained independently and depend on what types of task, regression or classification, by averaging or considering the majority of those predictions, we obtain a more accurate estimate. This method is commonly used to deal with noisy dataset to reduce variance.

3) Random Forest Algorithm

Random Forest algorithm used both bagging and **feature randomness** to create an uncorrelated forest of decision trees. Feature randomness, also known as feature bagging or "the random subspace method". It generates a random subset of features, which ensures that each decision tree in the Random Forest algorithm is not strongly correlated. This is a key difference between Decision tree and Random forest. Unlike decision trees, which consider all possible feature splits, random forests only utilize a subset of those features.

How can we create a Random Forest? Let's assume that we have a N training examples, and for each samples we have M features. A Random Forest will consists of N_{tree} decision tree, or estimators.

Step 1: Bagging step. In this step, we will create a sample subset which is called **Bootstrapped dataset** have size n repeatedly select random n samples from N samples, where $n < N$ and can be choose replacement.

Step 2: Create a Decision tree and training that Decision tree by using the bootstrapped dataset, but only use a random subset of variables at each step, meaning that each step we will choose randomly only m features from M features where $m < M$ at each step in Decision tree algorithm.

Step 3: Repeat the first and second steps until amount of estimator created equal N_{tree}.

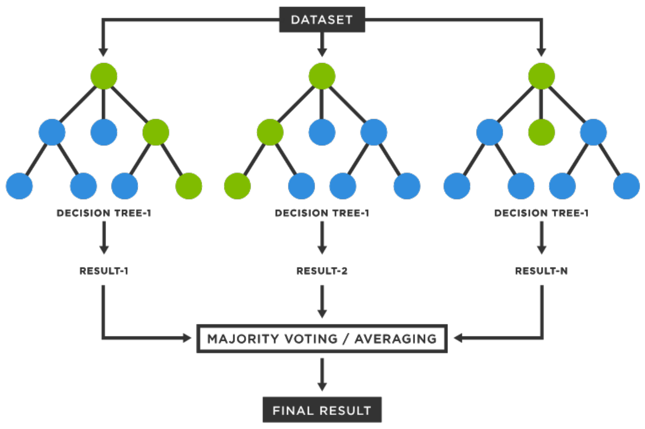


Fig. 26: Random Forest Algorithm

To make a prediction for a new input, we pass the relevant features of this input to each of the N_{tree} estimators. After that, we will have N_{tree} predictions, depend on what types of task we are solving, we will use the predictions from all estimators to get a final result. In case of classification, we will use majority voting to determine the predicted class.

4) Out-of-Bag Score

Out-of-Bag Score or OOB Score is a method for validating the Random Forest Model. In Random Forest Model, in first step when create the bootstrapped dataset, we randomly select 'n samples from N samples and can be replacement. After training that Decision Tree, there will be some samples that were not chosen, and these samples are referred to as Out-of-Bag samples for that specific decision tree. Each decision tree will have a different set of Out-of-Bag samples.

After creating the Random Forest, for each sample in the Out-of-Bag samples, the decision trees that did not have that sample during their initial creation will treat it as unseen data. These decision trees will then predict the output for that sample. The final prediction for the sample is obtained by majority voting among all that decision trees treat it as unseen. The OOB score is calculated by determining the number of correctly predicted samples from the out-of-bag (OOB) sample.

Where is OOB score useful? Only a subset of decision trees is used to determine the OOB score. This approach helps reduce the overall aggregation effect in bagging. If our dataset is not large enough and we want to use all of it for training purposes, the OOB score will provides a good trade-off.

5) Advantage and Disadvantage

Advantages

- Reduce risk of overfitting
- Provides flexibility
- Easy to determine feature importance
- Handles mixed data types

Disadvantages

- Time-consuming process
- Required more resources
- More complex
- Hyperparameter Sensitivity

6) Hyperparameters

In this report, we will use sklearn library in python forest it provides support for various hyperparameters that we can modify when constructing a random forest. Because random forest is build from many decision tree, so some hyperparameters of Random forest are similar to those of the decision tree.

(a) ***n_estimators: int, default = 100***
It represents the number of decision trees in the Random Forest. Increasing the number of trees generally improves performance, but it also increases training time and memory consumption.

(b) ***criterion: ["gini", "entropy", "log_loss"], default = "gini"***
The function used to measure the quality of a split when constructing decision trees within the ensemble. The criterion determines how the algorithm evaluates and selects the best feature to split a node based on the impurity or information gain.

(c) ***max_depth: int, default = None***
It sets the maximum depth of each decision tree in the Random Forest. Increasing max_depth can lead to overfitting, while reducing it may improve generalization. In default, trees are expanded until all leaves are pure or contain minimum samples.

(d) ***min_samples_split: int or float, default = 2***
It represents the minimum number of samples required to split an internal node. This may have the effect of smoothing the model, especially in regression because it controls the trade-off between too specific decisions (overfitting) and too general decisions (underfitting)

(e) ***min_samples_leaf: int or float, default = 1***
It sets the minimum number of samples required to be at a leaf node. It controls the minimum size of the final decision nodes. Smaller values can make the model more prone to overfitting.

- If *int*, then consider *min_samples_leaf* as the minimum number.
- If *float*, then *min_samples_leaf* is a fraction and *ceil(min_samples_leaf * n_samples)* are the minimum number of samples for each node.

(f) ***min_weight_fraction_leaf: float, default = 0.0***
It limits the size of leaf nodes based on the total weight (sum of sample weights) rather than the number of samples. It can be useful when working with weighted datasets or when trying to control the size of leaf nodes. By default, this hyperparameter is not used, and the maximum number of samples *(min_samples_leaf)* is used instead. Samples have equal weight when *sample_weight* is not provided

(g) ***max_features: ["sqrt", "log2", None], default = "sqrt"***
It determines the number of features to consider when looking for the best split at each node
- If *int* then consider *max_features* features at each split.
- If *float*, then *max_features* is a fraction and *max(1, int(max_features * n_features_in_))* features are considered at each split.
- If *"auto"*, then *max_features = sqrt(n_features)*.
- If *"sqrt"*, then *max_features = sqrt(n_features)*.
- If *"log2"*, then *max_features = log2(n_features)*.
- If *None*, then *max_features = n_features*.

(h) ***max_leaf_nodes: int, default = None***
It limits the maximum number of leaf nodes in each decision tree of the Random Forest. Setting this hyperparameter can help control the complexity of the individual trees and prevent overfitting. If this hyperparameter is specified, the Random Forest will grow trees with fewer leaf nodes by setting *max_leaf_nodes* as the stopping criterion for tree growth.

(i) ***min_impurity_decrease: float, default = 0.0***
A node will be split if this split induces a decrease of the impurity greater than or equal to this value. By setting a positive value for *min_impurity_decrease*, you can control the quality of splits in the Random Forest. The weighted impurity decrease equation is the following:

$$\frac{N_t}{N} * (imp - \frac{N_{t\_R}}{N_t} * r\_imp - \frac{N_{t\_R}}{N_t} * l\_imp)$$

- N : Total number of samples.
- N_t: Number samples at the current nodes.
- N_{t_L}: Number samples in the left child.
- N_{t_R}: Number samples in the right child.
- imp: impurity.
- l_imp: left impurity.
- r_imp: right impurity.

N, N_t, N_{t_R}, N_{t_L} all refer to the weighted sum, if *sample_weight* is passed.

(j) ***bootstrap: bool, default = True***
It specifies whether bootstrap samples should be used when building trees. If set to *True*, each tree is built on a random subset of the training data with replacement. If *False*, the whole dataset is used to build each tree.

(k) ***oob_score: bool, default = False***
Whether to use out-of-bag samples to estimate the generalization score. Only available if *bootstrap = True*.

(l) ***ccp_alpha: non-negative float, default = None***
Complexity parameter used for Minimal Cost-Complexity Pruning. The sub-tree with the largest cost complexity that is smaller than *ccp_alpha* will be chosen. By default, no pruning is performed. Pruning removes unnecessary branches of a decision tree, reducing its complexity and potentially improving generalization. Higher values of *ccp_alpha* increase the amount of pruning and lead to smaller, more generalized decision trees.

5. EXPERIMENT

5.1. Preprocessing

In order to gain a better understanding of the operation and interaction of four algorithms: Logistic Regression, Support Vector Machine, Decision Tree, Random Forest with the Diabetes dataset by training, tuning and evaluate model.

In the EXPLORATORY DATA ANALYSIS section, it is observed that some features in the dataset have a lower correlation with the target variable. However, in order to better understand how the algorithms will interact with these features, we decided to use all of them to training and evaluate.

First of all, we can see 2 features: "Gender", "Smoking history" which are a categorical features, so we will use Label Encoder from sklearn library to transform them to numerical features.

Moving forward, we currently have all the features represented as numerical values. However, they are not in the same scale. To address this, we will employ the MinMaxScaler to normalize the data, ensuring that all features are brought to a common scale without significant loss of information.

5.2. Metric

Confusion matrix: helps us to have an overview of the classification ability of the model in each class of diabetes and non-diabetic. From there, we can evaluate the model's performance and adjust the model's hyperparameter to improve the classification results.

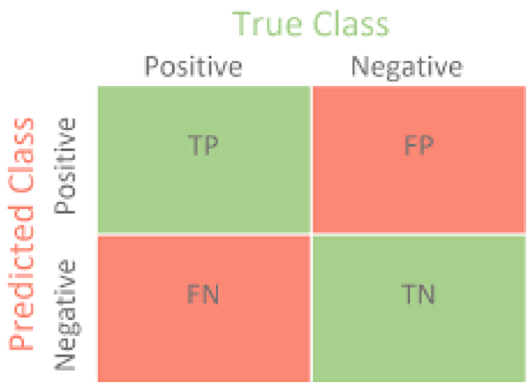


Fig. 27: Confusion Matrix

- True Positive (TP): It refers to the number of predictions where the classifier correctly predicts the positive class as positive.
- True Negative (TN): It refers to the number of predictions where the classifier correctly predicts the negative class as negative.
- False Positive (FP): It refers to the number of predictions where the classifier incorrectly predicts the negative class as positive.
- False Negative (FN): It refers to the number of predictions where the classifier incorrectly predicts the positive class as negative.

Accuracy: The percentage correctly classified in the total data.

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

Precision: Percentage classified as diabetic for which the correct result is diabetes. Determine the ability to accurately detect diabetes cases and avoid giving false results.

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

Recall: The percentage that is diabetes that is correctly classified. Determine the likelihood of missed and misclassified diabetes cases to avoid giving false results.

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

$\mathbf{F_1 - score}$: Compare performance between different models.

$$F_1 - score = \frac{2 * precision * recall}{precision + recall} = \frac{2TP}{2TP + FN + FP}$$

5.3. Protocol

In this project, to avoid overfitting when the model has trained "too well", we need to use **Train test split** which is supported by sklearn library. The train test split is a common technique used in machine learning to evaluate the performance and generalization ability of a model. Here are a few reasons why we need to use train test split:

- *Model Evaluation*: By splitting the available data into training and testing sets, we can evaluate how well our model performs on unseen data. The training set is used to train the model, and the testing set is used to evaluate its performance. This helps us estimate how well the model will perform on new, unseen data.
- *Assessing Generalization*: The main goal of a machine learning model is to generalize well to unseen data. The train-test split helps us simulate this scenario by evaluating the model's performance on data that it has not been exposed to during training. This gives us an indication of how well the model can generalize to new, unseen examples.
- *Preventing Overfitting*: Overfitting occurs when a model becomes too complex and starts to memorize the training data instead of learning the underlying patterns. By evaluating the model on a separate testing set, we can identify if the model is overfitting. If the model performs well on the training set but poorly on the testing set, it is a sign of overfitting, and adjustments may be needed to improve generalization.

Overall, the train-test split is a fundamental technique for model evaluation, assessing generalization and preventing overfitting. It helps us understand how well our model performs on unseen data and enables us to make informed decisions regarding model selection and optimization.

5.4. Hyperparameter tuning

1) Logistic Regression

Tuning hyperparameters is a critical step in maximizing the performance of logistic regression models, which are commonly used for classification tasks.

Several key hyperparameters can be fine-tuned to optimize the model's effectiveness, including the regularization parameter (C), choice of solver algorithm, and the type of penalty applied.

- **C**: [0.001, 0.01, 0.1, 1, 10, 100, 1000]
- **penalty**: ["l1", "l2"]
- **solver**: ["lbfgs", "liblinear", "saga"]

Rank	C	penalty	solver	Mean test score
1	1	l2	liblinear	0.884733
2	1	l2	lbfgs	0.884467
3	0.1	l1	liblinear	0.884400
3	0.1	l1	saga	0.884400
5	1	l2	saga	0.884400
6	1000	l2	liblinear	0.884267
6	1000	l1	saga	0.884267
6	1000	l1	liblinear	0.884267
6	100	l2	lbfgs	0.884267
6	100	l2	saga	0.884267

TABLE II: Tuning Logistic Regression hyperparameters

2) Support Vector Machine

Tuning the hyperparameters in the SVM is important to improve model performance and efficiency, help the model adapt to specific data characteristics, and achieve better prediction accuracy. Default hyperparameter values may not always be appropriate for a particular data set or problem.

Several key hyperparameters can be tweaked to optimize the model's efficiency, including the normalization parameter (C), the choice of the type of kernel to be used in the algorithm, the choice Kernel coefficient (gamma).

- **C**: 10 logarithmic values in interval from 10^2 to 10^4.
- **kernel**: ["rbf", "poly", "sigmoid"]
- **gamma**: 10 logarithmic values in interval from 10^{-1} to 10^{-10} + ["auto","scale"]

Rank	C	gamma	kernel	Mean test score
1	278.25594	scale	rbf	0.901467
2	100.0	scale	rbf	0.901400
3	774.263683	scale	rbf	0.901267
4	464.158883	scale	rbf	0.901133
5	1291.549665	scale	rbf	0.901067
6	166.810054	scale	rbf	0.901000
7	100.0	scale	poly	0.899867
8	3593.813664	scale	poly	0.899600
9	2154.43469	scale	poly	0.899600
10	10000.0	auto	rbf	0.899533

TABLE III: Tuning Support Vector Machine hyperparameters

3) Decision Tree

To avoid the overfiting in a decision tree and find the best tree for our model, we need to fine-tuning the hyperparameters. By optimizing it, the learning process can be controlled, the generalization error has been reduced and the better model will be created.

After taking survey and doing some experiments, we choose some important hyperparameters which impact our model in term of over-fitting and under-fitting for the fine-tuning process. The range of values that we consider for Decision Tree hyperparameters are setting up following:

- **criterion**: [“gini”, “entropy”]
- **max_depth**: [8, 10, 12, 15, 20]
- **min_sample_leaf**: [10, 12, 15, 20]
- **min_sample_split**: [5, 10, 15, 20, 25]

Rank	criterion	max_depth	min_sample_leaf	min_sample_split	Mean test score
1	entropy	11	10	15	0.90280
2	entropy	11	11	10	0.90260
2	entropy	11	10	5	0.90260
2	entropy	11	12	5	0.90260
5	entropy	11	10	25	0.90253
6	entropy	11	10	20	0.90253
6	entropy	11	11	5	0.90253
6	entropy	11	11	15	0.90253
6	entropy	11	10	10	0.90253
10	entropy	11	12	10	0.90247

TABLE IV: Tuning Decision Tree hyperparameters

4) Random Forest

Even though Random Forest models are less likely to overfit, we still aim to enhance performance and prevent overfitting occurrences. By fine-tuning the model’s parameters, we can improve its predictive abilities and minimize the risk of overfitting. After taking survey and doing some experiments, we choose some important hyperparameters which impart out model in term of over-fitting and under-fitting for the fine-tuning process. The range of values that we consider for Random Forest hyperparameters are setting up following:

- **n_estimators**: [100,200,300]
- **max_depth**: [5, 10, 15, 20]
- **criterion**: [“gini”, “entropy”]
- **min_sample_split**: [2, 5, 10, 20]
- **max_features**: [“log2”, “sqrt”]
- **ccp_alpha**: [0.0, 0.01, 0.015, 0.02]

Rank	criterion	max_depth	max_features	min_samples_split	n_estimator	Mean test score
1	entropy	15	sqrt	20	200	0.909933
2	entropy	15	log2	20	300	0.909733
2	gini	15	log2	20	200	0.909667
4	entropy	15	sqrt	20	300	0.909600
5	entropy	15	sqrt	10	200	0.909400
6	entropy	15	sqrt	10	300	0.909000
7	entropy	20	sqrt	20	200	0.908933
7	gini	15	sqrt	20	100	0.908933
9	entropy	20	sqrt	10	100	0.908867
9	gini	15	sqrt	20	300	0.908867

TABLE V: Tuning Random Forest hyperparameters

5.5. Result

In this experiment, we investigated the performance of four different classification methods based on some common metrics on the same Diabetes dataset to compare their performance.

	Accuracy	Recall	Precision	F1-score
Logistic Regression	0.8804	0.88	0.88	0.88
Tuned Logistic Regression	0.8847	0.88	0.88	0.88
Support Vector Machine	0.8894	0.88	0.88	0.88
Tuned Support Vector Machine	0.9015	0.89	0.89	0.89
Decision Tree	0.876	0.88	0.88	0.88
Tuned Decision Tree	0.9028	0.90	0.90	0.90
Random Forest	0.902	0.90	0.90	0.90
Tuned Random Forest	0.9099	0.91	0.91	0.91

TABLE VI: Evaluation of classification algorithms

Overall Logistic Regression, Support Vector Machine, Decision Tree and Random Forest bring high accuracy scores for Diabetes Classification. This result benefit from the good performance of preprocessing data step by dropping unnecessary features and scaling data. As can be seen, experimental results show that Tree algorithms are performed classification better than the others. Before tuning, Decision Tree’s accuracy is lower than Support Vector Machine’s accuracy but when we tuning hyperparameters of all models the accuracy of Decision Tree is better than Support Vector Machine. In conclusion, the results of all algorithms increase and the Random Forest still remains the best classification algorithm for our model. However, the Random Forest takes more time for training and tuning step than the other algorithms. This shows the efficiency in tuning hyper parameters for our classification model, since our model prioritized accuracy.

6. CONCLUSION AND FUTURE WORK

1) Conclusion

In conclusion, from analysis on comparison among classification algorithms (Decision Tree, Random Forest, Logistic Regression, Support Vector Machine) after tuning, it shows that Random Forest algorithm are the the best classification algorithm for our model. In this project, in order to predict patients at the high risk of developing the diabetes accurately, we indeed focus on the recall score. As a result, we choose the tuned Random Forest as the key algorithm to help people to detect diabetes base on their basic personal information.

2) Future work

For the diabetes classification problem, we have a number of development directions as follows:

Algorithms Combine: Experiment with combining multiple algorithms to take advantage of each one.

Incremental Learning: Investigate techniques for incremental learning, where the model can adapt and learn from new data over time. This is particularly useful in the healthcare domain, where new data are constantly being generated.

Real-time Monitoring: Develop systems that can continuously monitor diabetes-related parameters in real-time, such as blood glucose levels or patient symptoms. This can aid in early detection and proactive management of the disease.

Personalized Medicine: Investigate approaches to tailor diabetes classification models to individual patients based on their unique characteristics, such as genetics, lifestyle, or medical history. This can help in providing personalized treatment plans.

7. ACKNOWLEDGEMENT

We would like to express our gratitude to Ph.D. Tiep Nguyen Vinh of the University of Information Technology for instructing and guiding the project implementation team.

References

[1] Linear Models
https://scikit-learn.org/stable/modules/linear_model.html

[2] Logistic Regression - Detailed Overview
https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc

[3] Logistic Regression in Machine Learning
https://www.geeksforgeeks.org/understanding-logistic-regression/

[4] Logistic Regression
https://www.analyticsvidhya.com/blog/2021/10/everything-you-need-to-know-about-linear-regression/

[5] Support Vector Machines Models
https://scikit-learn.org/stable/modules/svm.html

[6] Support Vector Machines
https://medium.com/towards-data-science/support-vector-machines-svm-c9ef22815589

[7] Support Vector Machines - soft margin and kernel trick
https://towardsdatascience.com/support-vector-machines-soft-margin-formulation-and-kernel-trick-4c9729dc8efe

[8] Support Vector Machines in Machine Learning
https://machinelearningcoban.com/2017/04/09/smv/

[9] Support Vector Machines in Machine Learning soft margin
https://machinelearningcoban.com/2017/04/13/softmarginsmv/

[10] Support Vector Machines in Machine Learning kernel
https://machinelearningcoban.com/2017/04/22/kernelsmv/

[11] Decision Tree
https://scikit-learn.org/stable/modules/tree.html

[12] Decision Tree Algorithm – A Complete Guide
https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/

[13] Decision Tree in Machine Learning
https://machinelearningcoban.com/tabml_book/ch_model/decision_tree.html

[14] Decision Tree Algorithm, Explained
https://www.kdnuggets.com/2020/01/decision-tree-algorithm-explained.html

[15] Decision Tree Classification
https://www.datacamp.com/tutorial/decision-tree-classification-python

[16] A. Navada, A. N. Ansari, S. Patil and B. A. Sonkamble, ”Overview of use of decision tree algorithms in machine learning,” 2011 IEEE Control and System Graduate Research Colloquium, Shah Alam, Malaysia, 2011, pp. 37-42, doi: 10.1109/ICSGRC.2011.5991826.

[17] Ensemble methods
https://scikit-learn.org/stable/modules/ensemble.html

[18] Random Forest
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[19] Random Forest
https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

[20] Random Forest
https://towardsdatascience.com/understanding-random-forest

[21] Out-of-bag
https://towardsdatascience.com/what-is-out-of-bag-oob-score-in-random-forest