Nanotechnology Perceptions
ISSN 1660-6795
www.nano-ntp.com

AI Framework for Diabetes Detection via Health Records

Saurabh Kumar[1], Dr. Ritu Sindhu[2]

[1]*Ph.D scholar, Lingaya's Vidyapeeth, Sharma8saurabh@gmail.com*
[2]*Professor Lingaya's Vidyapeeth, Ritu.sindhu2628@gmail.com*

The prediction of diabetes is a crucial task in healthcare, with significant implications for early diagnosis and intervention. This study compares six commonly used classification algorithms—k-Nearest Neighbors (KNN), Naive Bayes, Decision Tree, Random Forest, Support Vector Machine (SVM), and Logistic Regression on a publicly available diabetes dataset. We evaluate the models based on accuracy, precision, recall, F1-score, and confusion matrix to determine the most effective algorithm for predicting diabetes outcome. Our findings indicate that Random Forest and Decision Tree outperform other classifiers in terms of accuracy and generalization.

Keywords: Machine learning; Classification; Diabetes.

1. Introduction

Diabetes is one of the most prevalent chronic diseases worldwide, affecting millions of individuals. Early detection and timely intervention are crucial for effective management, reducing complications and improving the quality of life for patients. Machine learning (ML) algorithms have shown promising potential in predicting the likelihood of diabetes onset based on various health-related features such as glucose levels, BMI, age, and family history.

This study aims to compare the performance of six classification models— k-Nearest Neighbors (KNN), Naive Bayes, Decision Tree, Random Forest, Support Vector Machine (SVM), and Logistic Regression on diabetes dataset. By comparing these models, we intend to identify the most suitable classifier for predicting diabetes outcomes in terms of accuracy, precision, recall, F1-score, and overall generalization performance.

2. Related Work

The application of machine learning in healthcare, specifically for the prediction of diabetes, has been widely explored. Several studies have focused on applying various classification algorithms to predict diabetes based on demographic, clinical, and laboratory data.

A study by Ali et al. (2020) demonstrated that the k-Nearest Neighbors (KNN) algorithm could effectively classify diabetes outcomes. KNN works by assigning a class label based on the majority class among the nearest neighbors of a data point. While KNN is simple and intuitive, its performance often depends on the choice of distance metric and the number of neighbors used.

Priya et al. (2020) applied Naive Bayes classification to predict diabetes from a dataset of medical features. The authors showed that Naive Bayes, being a probabilistic classifier, performs well when the features are conditionally independent. It is especially useful when dealing with noisy data but may underperform if the independence assumption is violated.

The decision tree algorithm is a popular model for classification due to its interpretability and simplicity. However, it can suffer from overfitting when the data is noisy. Permana et al. (2021) showed that decision trees can be a robust tool for diabetes classification, but combining multiple trees into an ensemble (i.e., Random Forest) leads to improved accuracy and generalization.

Support Vector Machines (SVM) have gained attention for their ability to handle high-dimensional data and their strong theoretical foundations. Studies like Kumari et al. (2013) have demonstrated that SVMs with non-linear kernels outperform traditional classifiers in binary classification tasks like diabetes prediction.

Logistic regression is one of the simplest classification models, commonly used for binary classification tasks. In Rajendra and Latifi (2021), logistic regression was applied to predict diabetes and found to perform reasonably well when combined with regularization to avoid overfitting.

Other works like Zheng et al. (2017) used Na¨ıve Bayes, Decision Tree and Random Forest to predict type 2 diabetes. Work of Pang et al. (2024) used BiLSTM-CRF, XGBoost, and Logistic Regression models to predict Diabetes. The work of Rashid et al. (2022) use 1-D convolutional RNN with LSTM layer with predict type 2 diabetes. Work of Nguyen et al. (2019) used deep learning to detect type 2 diabetes. Further work of Chaki et al. (2022) used KNN and SVM to classify patients as diabetic. Work of HL et al. (2023) used random forest to predict diabetes in canadian patients. The work of Patil et al. (2024) used SVM to classify patients diabetes. Work of Roy et al. (n.d.) used Structure Equation Modelling to monitor diabetes. Further work of Kenner et al. (2021) used machine learning to predict pancreatic cancer. The work of Menon et al. (2023) used advanced- spatial-vector-based Random Forest to classify diabetes health records. Work of Boudjemadi et al. (2021) used logistic regression to predict type 2 diabetes. The work of Khalifa and Albadawy (2024) use predictive modeling to classify patients as diabetic. Work of Ellahham (2020) highlights the prevalence of support vector machine, artificial neural network, Adaboost, logistic regression, decision tree, and random forest in predicting diabetes. THe work of Rigla et al. (2018) used decision trees, artificial neural networks and support vector machines predict type 2 diabetes.

While these algorithms have been widely applied in the healthcare domain, there is limited research directly comparing their performance on the same diabetes dataset. This paper aims to fill this gap by conducting a comprehensive evaluation of six classification algorithms and providing a comparative analysis of their strengths and weaknesses.

3. Methodology

3.1 Dataset Description

The dataset used in this study is the Diabetes Database Diabetes Dataset (2024), which contains information about 2768 patients. The dataset includes 8 features (e.g., pregnancies, glucose, blood pressure, skin thickness, insulin, BMI, age, and diabetes pedigree function, age) and a binary target variable, ‘Outcome‘, where ‘1‘ indicates the presence of diabetes and ‘0‘ indicates the absence.

3.2 Data Preprocessing and Analysis

- Handling Missing Values: Missing data in the dataset was handled by replacing missing values with the median of the respective feature.
- Feature Scaling: To improve the performance of models sensitive to the scale of data (e.g., KNN, SVM), the features were standardized, which centers the data around zero with a unit variance.‘
- Data Split: The dataset was randomly split into 80% for training and 20% for testing.

We first visualize the univariate distribution of numerical variable in Figure 1 Then we visualize the correlation of numerical variable in Figure 2

3.3 Models

We implemented and compared the following models:

- k-Nearest Neighbors (KNN): k-Nearest Neighbors (k-NN) is one of the simplest and most widely used machine learning algorithms for classification and regression tasks. It belongs to the family of instance- based learning algorithms, which means that the model doesn’t explicitly learn a discriminative function during the training phase. Instead, the model memorizes the training dataset and makes predictions based on the stored data during the testing or prediction phase. The core idea behind k-NN is to classify or predict the output for a new data point based on the majority label (in classification) or the average label (in regression) of its ’k’ nearest neighbors in the feature space. This makes k-NN particularly intuitive and easy to understand, but it also means that its performance heavily depends on the distance metric used and the value of ’k’.

The definition of k-NN begins with the concept of a feature space, where each data point is represented as a vector in an n-dimensional space. Suppose we have a training dataset D = {(xi, yi)}N , where xi ∈ Rn is the feature vector for the i-th instance, and yi is the corresponding label or target value. The goal is to predict the label or value y for a new query point xq based on its proximity to the training data points in the feature space.

The primary mathematical foundation behind k-NN is the distance metric used to compute the proximity between the query point and each of the training points. The most commonly used distance measure is the Euclidean distance, although other distance metrics, such as Manhattan distance or Minkowski distance, can also be employed depending on the problem and domain. The Euclidean distance between two points xi and xq in an n-dimensional space is given by:

$$d(x_i, x_q) = \sqrt{\sum_{j=1}^{n} (x_{ij} - x_{qj})^2}$$

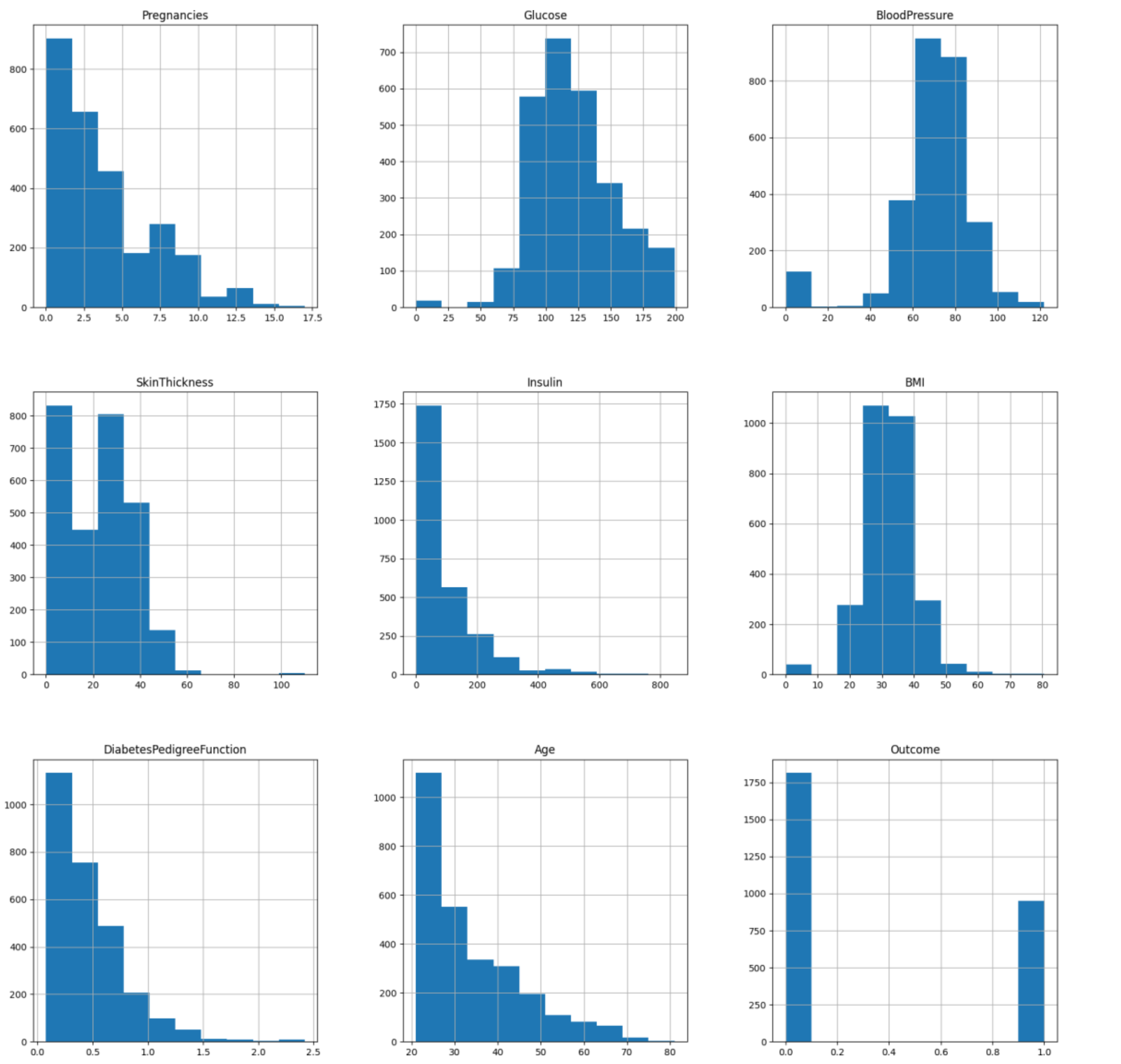

Figure 1: Univariate Distribution of Numerical variables.

where xij and xqj represent the j-th feature of the points xi and xq , respectively. This distance metric quantifies how far two points are from each other, with smaller values indicating closer proximity.

The training procedure in k-NN is essentially trivial in terms of model fitting. Unlike other machine learning algorithms, k-NN does not require optimization of model parameters or explicit learning from the training data. Instead, during training, the algorithm simply stores the entire dataset, which is why it is often referred to as a lazy learner. There is no need for any explicit training procedure in the traditional sense—data points are simply indexed for quick retrieval when making predictions. The only hyperparameter that needs to be determined is the value of k, which specifies how many nearest neighbors will be considered for making a prediction.

Once the training data has been stored, the prediction mechanism in k-NN involves two main steps: identi- fying the k nearest neighbors and then aggregating their labels or target values to make a final prediction. Let us focus on the classification task first. Given a query point xq , the algorithm computes the distances between xq and all training points xi. These distances are then sorted, and the k training points with the smallest distances to xq are selected. Once the nearest neighbors are identified, the next step is to make the prediction. In classification, the most common approach is to perform a majority vote among the labels of the k-nearest neighbors. The predicted label yˆq for the query point xq is given by:

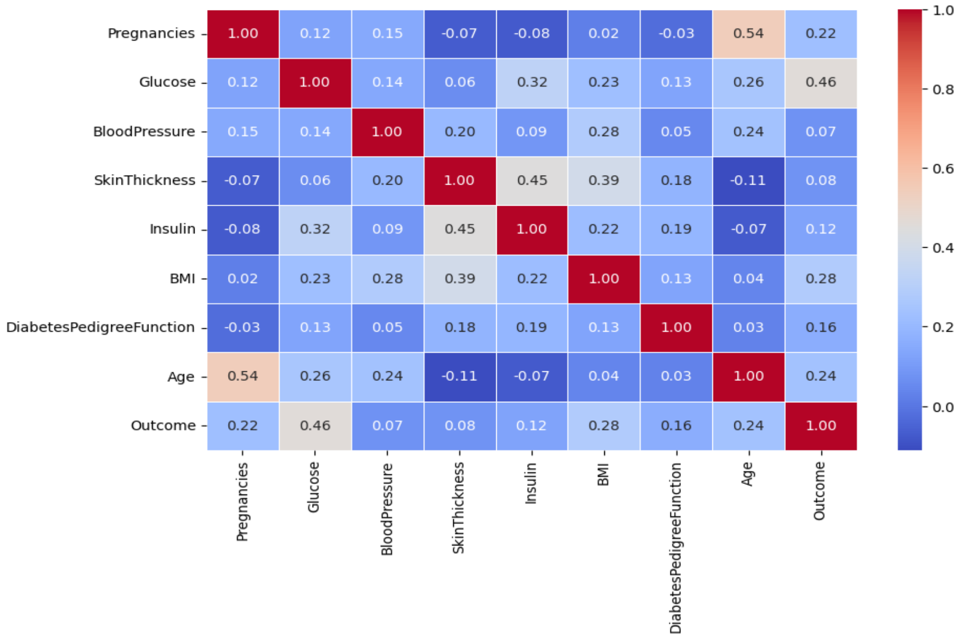

Figure 2: Correlation of Numerical variables.

$$\hat{y}_q = \text{mode}\,\{y_i : x_i \in \text{nearest neighbors of } x_q\}$$

where the mode function returns the most frequent label among the k-nearest neighbors. In the case of a tie, several strategies can be used, such as choosing the label with the smallest distance or randomly selecting one of the tied labels.

For regression tasks, the mechanism for predicting the output is slightly different. Instead of using a majority vote, the algorithm computes the mean or weighted average of the target values of the k-nearest neighbors. Let yi be the target value associated with the training point xi. The predicted value yˆq for the query point xq is then given by:

$$\hat{y_q} = \frac{1}{k} \sum_{i \in \text{nearest neighbors of } x_q} y_i$$

If weighted k-NN is used, where closer neighbors have more influence, the prediction is adjusted by weighting the contributions of each neighbor by the inverse of their distance. Specifically, the weighted prediction can be written as:

$$\hat{y_q} = \frac{\sum_{i \in \text{nearest neighbors of } x_q} \frac{y_i}{d(x_i, x_q)}}{\sum_{i \in \text{nearest neighbors of } x_q} \frac{1}{d(x_i, x_q)}}$$

where d(xi, xq) is the distance between the query point xq and the i-th neighbor. This weighted version of k-NN ensures that closer neighbors contribute more significantly to the prediction, which can often lead to better results, especially in cases where the data is noisy or non-linear.

In both classification and regression tasks, one important aspect of the k-NN algorithm is the choice of k, which can greatly affect model performance. A small value of k (e.g., k = 1) makes the model very sensitive to noise in the data, as the prediction is based only on the nearest neighbor, which might not be representative of the overall data distribution. On the other hand, a large value of k results in smoother decision boundaries, but it can also lead to over-smoothing, where the model fails to capture important patterns in the data. Therefore, selecting an optimal value of k is crucial, and it is often determined through techniques such as cross-validation or grid search.

Additionally, the choice of distance metric plays an important role in the performance of k-NN. While Euclidean distance is the default in many applications, it assumes that all features are on the same scale and equally important. In cases where features have different units or varying ranges, the distance metric may need to be normalized or standardized. For example, if xi and xq contain features with different magnitudes, such as age in years and income in thousands of dollars, it is often beneficial to scale the features before calculating the Euclidean distance. One common approach is to use z-score normalization, where each feature xj is standardized by subtracting its mean and dividing by its standard deviation:

$$x'_j = \frac{x_j - \mu_j}{\sigma j}$$

where μj is the mean and σj is the standard deviation of the j-th feature. After standardization, the Euclidean distance can be calculated as:

$$d(x_i, x_q) = \sqrt{\sum_{i=1}^{n} (x_{ij} - x_{qj})^2}$$

Alternatively, one might use other distance metrics such as Manhattan distance:

$$d(x_i, x_q) = \sum_{j=1}^{n} |x_{ij} - x_{qj}|$$

or Minkowski distance, which generalizes both Euclidean and Manhattan distances:

$$d(x_i, x_q) = \left(\sum_{j=1}^{n} |x_{ij} - x_{qj}|^p \right)^{1/p}$$

where p is a parameter that, when set to 2, results in the Euclidean distance, and when set to 1, results in the Manhattan distance.

One of the major advantages of k-NN is its non-parametric nature. It makes no assumption about the underlying distribution of the data, which makes it highly flexible and able to model complex, non-linear decision boundaries. However, this also means that the model can suffer from the “curse of dimensionality” in high-dimensional spaces. As the number of features increases, the distance between data points becomes more similar, and the algorithm’s ability to differentiate between neighbors diminishes. To mitigate this, di- mensionality reduction techniques such as Principal Component Analysis (PCA) or t-Distributed Stochastic Neighbor Embedding (t-SNE) are sometimes employed before applying k-NN to high-dimensional data.

Despite its simplicity, k-NN has several limitations. The most notable one is its computational cost, especially during the prediction phase. Since the algorithm has to compute the distance between the query point and all points in the training dataset, its time complexity for prediction is O(N), where N is the number of training samples. This can be prohibitively slow for large datasets, especially in high-dimensional spaces. To address this issue, various optimization techniques have been proposed, such as using data structures like k-d trees or ball trees, which allow for faster nearest neighbor searches by partitioning the feature space and reducing the number of comparisons.

Another limitation is that k-NN is sensitive to the quality of the data. If the dataset contains irrelevant features, noisy data, or outliers, the algorithm’s performance can degrade significantly. In such cases, preprocessing steps such as feature selection, outlier detection, and noise filtering can help improve the model’s robustness and accuracy.

- Naive Bayes: Naive Bayes is a family of probabilistic algorithms based on applying Bayes’ theorem with strong (naive) independence assumptions between the features. It is particularly popular in classification tasks, especially when the data can be modeled well with the assumption that each feature is conditionally independent given the class label. Despite its simplicity, Naive Bayes can perform surprisingly well, espe- cially in cases with high-dimensional data or when the feature independence assumption is approximately true.

The underlying mathematical model of Naive Bayes is grounded in Bayes’ theorem, which is a fundamental result in probability theory that describes how to update the probability of a hypothesis (in this case, the class label) based on observed evidence (features). Given a dataset with n features, denoted as x = (x1, x2, . . . , xn), the goal of Naive Bayes is to predict the class label y for a new observation based on these features. Bayes’ theorem provides a way to compute the posterior probability of the class y given the features x:

$$P(y \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid y)P(y)}{P(\mathbf{x})}$$

Here, P (y | x) is the posterior probability, P (x | y) is the likelihood, P (y) is the prior probability of the class y, and P (x) is the marginal likelihood of the features. The term P (x) is constant for all classes and does not affect the relative ranking of the classes, so it is typically ignored in the classification process. This simplification leads to the core of Naive Bayes, which involves maximizing the likelihood of the class given the data:

$$P(y \mid x) \propto P(x \mid y)P(y)$$

Now, the likelihood term P (x | y) is where the Naive Bayes assumption of feature independence comes into play. Naive Bayes assumes that the features x1, x2, . . . , xn are conditionally independent given the class label y. Therefore, the joint likelihood P (x | y) can be factored into the product of individual likelihoods for each feature:

$$P(\mathbf{x} \mid y) = \prod_{i=1}^{n} P(x_i \mid y)$$

Substituting this factorization back into the Bayes' theorem formula, we get:

$$P(y \mid \mathbf{x}) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

This equation expresses the posterior probability of the class y given the feature vector x. In practice, we compute the posterior probabilities for each possible class y and choose the class that maximizes this probability:

$$\hat{y} = \arg\max_{y} \left(P(y) \prod_{i=1}^{n} P(x_i \mid y) \right)$$

The next step is determining how to estimate the terms P (y) and P (xi | y). The prior probability P (y) represents the relative frequency of the class label y in the training data, and it can be estimated by simply counting the occurrences of each class in the dataset:

$$P(y) = \frac{\text{count}(y)}{N}$$

where count(y) is the number of instances in the training set with class label y, and N is the total number of instances in the training set.

The likelihood term P (xi | y) represents the conditional probability of feature xi given the class y. To esti- mate this probability, we typically rely on different methods depending on the type of data. For continuous features, we often assume that the feature values follow a normal (Gaussian) distribution, so the likelihood can be modeled as:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2} \right)$$

where µy and σ2 are the mean and variance of the feature xi for the class y, respectively. These parameters are typically estimated from the training data by calculating the sample mean and sample variance for each feature within each class.

For categorical features, the likelihood P (xi | y) is estimated by the relative frequency of the value xi occurring for class y. If the feature xi can take on k different values, then:

$$P(x_i = v \mid y) = \frac{\text{count}(x_i = v, y)}{\text{count}(y)}$$

where count(xi = v, y) is the number of instances where feature xi takes the value v and the class label is

y, and count(y) is the total number of instances in class y.

The process of training a Naive Bayes classifier involves estimating these parameters from the training data: the prior probabilities P (y) and the conditional probabilities P (xi | y) for each feature xi. The training procedure is straightforward, as it merely involves counting occurrences in the data. After training, the model is ready to predict the class label for new, unseen instances.

To predict the class label for a new observation x = (x1, x2, . . . , xn), we calculate the posterior probability for each possible class y using the formula:

$$P(y \mid \mathbf{x}) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

and then choose the class that maximizes this posterior probability. In practice, it is often more convenient to work with the logarithms of probabilities to avoid numerical underflow when multiplying many small probabilities. Taking the logarithm of both sides:

$$\log P(y \mid \mathbf{x}) = \log P(y) + \sum_{i=1}^{n} \log P(x_i \mid y)$$

The predicted class yˆ is then:

$$\hat{y} = \arg\max_{y} \left(\log P(y) + \sum_{i=1} \log P(x_i \mid y) \right)$$

One key feature of Naive Bayes is its simplicity and efficiency. Because the conditional independence assumption significantly reduces the complexity of the model, the training procedure is very fast, and it can handle large datasets with high-dimensional features. Moreover, the Naive Bayes model is computationally inexpensive during both training and prediction. This makes it particularly useful in applications such as text classification, spam filtering, and sentiment analysis, where the feature space is often large and sparse.

Despite its simplicity, Naive Bayes has some important limitations. The most significant is the strong inde- pendence assumption, which is rarely true in real-world datasets. If the features are highly correlated, the performance of Naive Bayes can suffer. However, in many practical situations, even when the independence assumption is violated, Naive Bayes can still provide competitive performance. Additionally, Naive Bayes does not handle feature interactions well, as it assumes that each feature contributes independently to the class label.

An extension of Naive Bayes, known as the Multinomial Naive Bayes, is commonly used in text classification tasks, where the features are typically word counts or term frequencies. In this variant, the likelihood term P (xi | y) is modeled as a multinomial distribution, which is appropriate for count data. The Multinomial Naive Bayes classifier has become particularly popular in natural language processing (NLP) tasks such as spam detection, sentiment analysis, and document classification.

Despite the simplifying assumptions and limitations, Naive Bayes remains a powerful tool for classification, particularly when computational resources are limited or when the problem involves high-dimensional data. It is often used as a baseline model in machine learning, with more complex models being compared to its performance. Furthermore, Naive Bayes' probabilistic nature provides a natural way to quantify uncertainty in predictions, which can be useful in decision-making contexts where probabilities rather than hard classifications are required.

• Decision Tree: A Decision Tree is a non-linear predictive model used for both classification and regression tasks. It builds a flowchart-like structure where each internal node represents a decision based on the value of a feature, each branch represents the outcome of that decision, and each leaf node represents a class label or a predicted value. The primary goal of a decision tree is to partition the data into subsets that are as pure as possible, based on a chosen criterion, which helps in making predictions or classifications. The process of constructing a Decision Tree involves two key phases: training the model, where the tree is built by recursively splitting the data, and using the model for prediction, where the tree is traversed to make decisions based on new input data.

Mathematically, the decision tree building process relies on concepts from information theory, such as entropy and Gini impurity, as criteria to select which feature and threshold to split on. For classification tasks, the entropy is defined as:

$$H(S) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

where H(S) is the entropy of a set S, and pi represents the proportion of elements in set S that belong to class i. The entropy measures the impurity of a set: the higher the entropy, the more mixed the set is in terms of class distribution. When building the tree, the algorithm looks for the feature that splits the data in such a way that the resulting subsets have lower entropy, indicating a purer classification.

The Gini impurity is another criterion used to measure the impurity of a dataset, and is defined as:

$$G(S) = 1 - \sum_{i=1}^{k} p_i^2$$

where G(S) is the Gini impurity of set S and pi is the proportion of elements in set S that belong to class i. Like entropy, a lower Gini score means a purer node. The Decision Tree algorithm evaluates the Gini index across all possible splits and chooses the one that results in the lowest Gini impurity for the subsets. During the tree-building process, the goal is to choose the best feature and threshold to split the data at each node. For each possible feature and its associated threshold, the algorithm calculates the reduction in impurity, which is a measure of how much the data is better partitioned. The ”best” split is the one that minimizes this impurity, which can be calculated for both entropy and Gini impurity using the following formulas:

$$\text{Information Gain} = H(S) - \sum_{i=1}^{m} \frac{|S_i|}{|S|} H(S_i)$$

where S is the dataset at a given node, Si is a subset of S formed by splitting on a particular feature, and |S| and |Si| are the sizes of the dataset and the subset, respectively. This formula quantifies how much the uncertainty (entropy) is reduced after the split. A higher information gain indicates a better feature split. Once the tree is trained, it can be used for prediction. In the case of classification, prediction is performed by passing a test instance down the tree. Starting at the root, the instance is evaluated against the decision rule at each node, and the appropriate branch is followed until a leaf node is reached. The prediction is determined by the majority class of the instances in that leaf. If yi is the class label associated with the i-th leaf, and ni is the number of instances assigned to this leaf, the predicted class for a new instance x is the class that has the most instances in the leaf:

$$y\hat{}(x) = \text{mode}(y_i \mid n_i)$$

In the case of regression, where the goal is to predict a continuous value, the prediction at a leaf node is typically the mean of the target variable in that leaf. If y1, y2, . . . , ym are the values of the target variable for the instances in a particular leaf, the predicted value yˆ(x) is the average:

$$y\hat{}(x) = \frac{1}{m} \sum_{i=1}^{m} y_i$$

The tree-building process typically uses recursive binary splitting, where each node is split into two child nodes based on a threshold for a particular feature. At each step, the decision tree algorithm evaluates all features and thresholds to determine the best split by calculating the impurity measures (either entropy or Gini index) and selecting the one that minimizes the overall impurity. The recursive process continues until a stopping criterion is met, such as a maximum tree depth, a minimum number of samples in a node, or a minimum impurity threshold. Another important aspect of training decision trees is the prevention of overfitting. Decision trees have a tendency to grow very deep, capturing even small variations in the data, which can result in high variance and poor generalization to unseen data. Techniques like pruning are employed to address this issue. Pruning involves removing branches from the tree after it has been fully grown to reduce complexity and prevent overfitting. One common approach to pruning is to use a post- pruning strategy, where the tree is grown fully and then pruned by removing nodes that do not contribute significantly to improving the model’s performance, often using a validation set to guide the process.

The Decision Tree algorithm, while simple and interpretable, can struggle with some issues. One issue is that it tends to be sensitive to small changes in the data, especially when the tree is deep, which leads to instability. This instability can be mitigated by using ensemble methods such as Random Forests or Gradient Boosting Machines, where multiple trees are trained on different subsets of the data and their predictions are combined to reduce variance and bias. In the case of Random Forests, each tree is trained on a random subset of the features and data points, and the final prediction is made by averaging the predictions of all the trees in the forest (for regression) or using a majority vote (for classification).

The training procedure for a decision tree typically involves the following steps:

1. Splitting the data: The data is split recursively based on the features and their respective thresholds.

2. Choosing the best feature: At each node, the algorithm evaluates all possible features and thresh- olds and selects the one that best separates the data.

3. Stopping criterion: The process stops when a stopping condition is met, such as when the tree reaches a maximum depth, when a node contains too few samples to justify a split, or when further splits do not reduce the impurity significantly.

The prediction mechanism for a decision tree is straightforward. Given a test sample, the algorithm begins at the root node and proceeds down the tree by evaluating the feature values at each node. At each decision node, the algorithm selects the appropriate branch depending on the feature value, and this process continues until a leaf node is reached. The prediction is then made based on the class or value associated with that leaf node, depending on whether the model is used for classification or regression.

In summary, the Decision Tree model is an interpretable, powerful tool for predictive modeling that is particularly useful when there are non-linear relationships in the data. It uses recursive partitioning to build a tree structure that maps input features to an output decision. The tree is trained by recursively splitting the data based on the feature that minimizes impurity, and it is used for prediction by traversing the tree to a leaf node. Though effective, decision trees are prone to overfitting, especially with deep trees, but techniques like pruning and ensemble methods such as Random Forests and Gradient Boosting can mitigate this problem, improving their performance and robustness. Despite these challenges, Decision Trees remain a fundamental building block in machine learning and data science, providing both accuracy and interpretability in many practical applications.

- Random Forest: Random Forest (RF) is a powerful and widely used ensemble learning technique primarily utilized for classification and regression tasks. It belongs to the family of decision tree-based algorithms, but what distinguishes it is its use of multiple decision trees to form a ”forest” of decision trees, which work together to produce a more robust and accurate prediction. In this model, the randomness comes from two sources: the random sampling of data points (bootstrapping) and the random selection of features at each split in a tree. By combining the results from many individual decision trees, Random Forest can significantly reduce overfitting and increase predictive accuracy compared to a single decision tree.

The mathematical foundation of a Random Forest begins with the construction of decision trees, which are non-linear models that recursively partition the feature space based on the values of input features. For a given dataset D = {(x1, y1), (x2, y2), . . . , (xn, yn)}, where xi represents the feature vector and yi represents the target variable (for classification, the target is discrete, while for regression, the target is continuous), a decision tree is constructed by recursively splitting the data based on feature values that minimize an impurity measure, such as Gini index for classification or mean squared error for regression.

For classification tasks, the Gini index is often used as the impurity measure at each split. The Gini index for a node t is defined as:

$$G(t) = 1 - \sum_{k=1}^{K} p_k^2$$

where pk is the proportion of samples in the node t that belong to class k, and K is the total number of classes. The decision tree algorithm aims to minimize the Gini index at each node. For regression tasks, the commonly used impurity measure is the mean squared error (MSE), which for a node t is defined as:

$$MSE(t) = \frac{1}{|t|} \sum_{i \in t} (y_i - \hat{y}_t)^2$$

where yˆt is the mean of the target values in node t, and |t| is the number of samples in node t. The goal is to split the dataset at each node such that the resulting sub-nodes have the smallest possible Gini index or MSE.

In Random Forest, the model is not defined by a single decision tree but by an ensemble of trees. The construction of each tree in the forest involves two sources of randomness. The first source is bootstrapping, where the training data is randomly sampled with replacement to form a training set for each tree. As a result, some data points may appear multiple times in the training set for a particular tree, while others may not appear at all. This technique helps to ensure that the trees are diverse, reducing the risk of overfitting that may arise from having a highly correlated set of trees.

The second source of randomness in Random Forest comes from the selection of features at each split in a decision tree. Instead of considering all features for each split, only a random subset of features is considered at each node. This ensures that individual trees are decorrelated, leading to a more diverse ensemble. Let m

denote the number of features, and at each split, a random subset of √m features is selected for evaluation.

This randomization introduces variability in the construction of the trees, further reducing overfitting and improving generalization.

Once all the trees in the forest are built, each tree is used to make a prediction. For classification tasks, the prediction of the Random Forest model is made by aggregating the predictions of each individual tree through a majority voting scheme. Specifically, for a given test sample x, the prediction from the Random Forest is the class that receives the most votes across all trees:

$$\hat{y}_{RF}(x) = \text{mode}\,(\hat{y}_1(x), \hat{y}_2(x), \ldots, \hat{y}_T(x))$$

where y^i(x) is the prediction of the i-th tree, and T is the total number of trees in the forest. This process ensures that the Random Forest model benefits from the collective knowledge of many individual trees, thereby improving predictive accuracy and robustness.

For regression tasks, the prediction is typically made by averaging the predictions from all the trees:

$$\hat{y}_{RF}(x) = \frac{1}{T} \sum_{i=1} \hat{y}_i(x)$$

where y^i(x) is the predicted value for x from the i-th tree, and T is the total number of trees. This averaging procedure ensures that the Random Forest is less sensitive to individual tree errors, leading to a more stable and accurate prediction.

Training a Random Forest involves several key steps. First, for each tree, a bootstrapped sample of the training data is selected. This sample is used to grow a decision tree by recursively splitting the data at each node based on the selected feature subset. The tree continues to grow until a stopping criterion is met, such as reaching a maximum depth or having a minimum number of samples at a node. Second, after all trees are trained, predictions are made by aggregating the individual tree predictions using majority voting or averaging, depending on the task at hand.

In terms of computational complexity, Random Forest is more resource-intensive compared to individual decision trees due to the need to train multiple trees. However, the parallel nature of the algorithm allows for efficient implementation on modern hardware. The training time scales linearly with the number of trees and the number of data points. More trees generally improve the performance of the model but come at the cost of increased computational burden.

Random Forest has several desirable properties. First, it is less prone to overfitting compared to a single decision tree, thanks to the averaging effect of the ensemble. Even if individual trees overfit the data, their errors tend to cancel out when combined in the Random Forest. This property makes it particularly suitable for high-dimensional and noisy datasets. Second, Random Forest is highly interpretable in terms of feature importance. Since each tree is built by considering different random subsets of features, the algorithm can provide insights into which features contribute most to the predictions. The importance of a feature can be assessed by measuring how much the prediction accuracy of the Random Forest decreases when that feature is excluded from the training process. This can be quantified using metrics such as the Gini importance or permutation importance, which rank features according to their contribution to the model's predictive power.

Another strength of Random Forest is its ability to handle missing data. During the construction of each tree, if a feature is missing for a particular sample, the algorithm can still use the available features to make a prediction. Moreover, Random Forest is robust to outliers, since the averaging mechanism and the majority voting process reduce the impact of extreme values on the final prediction.

Despite its strengths, Random Forest has some limitations. The model tends to be less interpretable compared to simpler models like decision trees or linear regression, especially when the number of trees is large. While the model itself is an ensemble of simple decision trees, the aggregation of so many trees can make it harder to extract insights about the decision-making process. Additionally, the model can be memory-intensive, particularly when working with large datasets or a large number of trees, as each tree requires storage for its structure and parameters.

Another limitation is the potential for computational inefficiency, especially in real-time prediction scenarios. While Random Forest can be parallelized, the need to evaluate many trees for each prediction can be slow, particularly for large datasets. This issue is particularly relevant in applications requiring low-latency predictions, such as real-time recommendation systems or online fraud detection.

In practice, Random Forest has been successfully applied in a wide range of domains. In bioinformatics, for instance, it has been used for gene expression analysis and protein function prediction. In finance, Random Forest models are employed for credit scoring and risk analysis, while in computer vision, they are used for image classification and object detection tasks. Its versatility, ease of use, and robust performance on a variety of tasks have made it one of the most popular machine learning algorithms.

In summary, Random Forest is a versatile and powerful ensemble learning technique that aggregates the predictions of multiple decision trees to improve predictive accuracy and reduce overfitting. By incorporat- ing randomness in both the data sampling and feature selection processes, Random Forest creates a diverse set of decision trees that work together to provide more reliable and generalizable predictions. Despite its relatively high computational cost, the model's ability to handle complex, high-dimensional datasets and its robustness to overfitting and outliers make it an invaluable tool in the machine learning practitioner's toolkit. Its wide range of applications and solid performance across a variety of domains further cement its position as one of the most popular and effective machine learning algorithms in use today.

• Support Vector Machine (SVM): Support Vector Machines (SVMs) are a class of supervised machine learning algorithms that are widely used for classification and regression tasks. The primary goal of an SVM is to find a hyperplane that best separates data points of different classes in a feature space. SVMs are particularly known for their effectiveness in high-dimensional spaces and are robust to overfitting, especially when the number of dimensions is larger than the number of samples. The fundamental idea behind SVMs is to define a decision boundary, or hyperplane, that maximizes the margin between the two classes of data points. This margin is defined as the distance between the hyperplane and the closest data points from either class, which are referred to as the support vectors. The larger this margin, the better the generalization ability of the classifier. Mathematically, the SVM tries to solve an optimization problem where the objective is to maximize the margin subject to certain constraints.

Let's assume that the data consists of n data points {(xi, yi)}n , where xi ∈ Rd are the feature vectors, and yi ∈ {−1, +1} is the class label. The goal of the SVM is to find a hyperplane characterized by the equation:

$$\mathbf{w}^T\mathbf{x} + b = 0$$

where w ∈ Rd is the weight vector that is orthogonal to the hyperplane, and b ∈ R is the bias term that determines the offset of the hyperplane. The decision rule for classifying a new data point x is given by:

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$$

A correct classification occurs if the sign of f (x) matches the true label y. The SVM aims to maximize the margin between the two classes, which is defined as the distance between the hyperplane and the closest data points. For a given data point xi, the margin is given by:

$$\text{Margin} = \frac{1}{\|\mathbf{w}\|}$$

The optimization problem now becomes finding w and b such that the margin is maximized while ensuring that all data points are correctly classified. The constraint for the classification is that each data point xi should lie on the correct side of the hyperplane. For yi = +1, the constraint is:

$$\mathbf{w}^T\mathbf{x}_i + b \geq 1$$

For yi = −1, the constraint is:

$$\mathbf{w}^T\mathbf{x}_i + b \leq -1$$

These constraints ensure that all points are classified correctly with a margin of at least 1. Therefore, the SVM optimization problem becomes a constrained optimization problem where the objective is to maximize the margin, subject to the constraints for all i:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

subject to the constraints:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \quad \text{for all} \quad i = 1, 2, \ldots, n$$

This is a convex optimization problem, and the solution to this problem provides the optimal hyperplane that maximizes the margin. To solve this optimization problem, one typically uses the method of Lagrange multipliers, which leads to the dual formulation of the problem. The dual form of the optimization problem allows for an efficient solution, particularly when the number of features is much larger than the number of data points.

In the dual formulation, we introduce Lagrange multipliers αi ≥ 0 for each constraint, leading to the following Lagrangian:

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

To find the saddle point of this Lagrangian, we take the partial derivatives of L(w, b, α) with respect to w and b and set them to zero:

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \quad \Rightarrow \quad \mathbf{w} = \sum_{i=1} \alpha_i \, y \, \mathbf{x}$$

$$\frac{\partial L}{\partial b} = 0 \qquad \Rightarrow \qquad \sum_{i=1}^{n} \alpha_i y_i = 0$$

Substituting these conditions into the Lagrangian, we obtain the dual optimization problem:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \; !$$

subject to the constraints:

$$\alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i y_i = 0$$

Solving this dual problem gives the optimal values of the Lagrange multipliers αi, which can then be used to compute the optimal weight vector w. The decision boundary is fully determined by the support vectors, which are the points where αi > 0. The bias term b can be computed by using the support vectors:

$$b = y_i - \mathbf{w}^T \mathbf{x}_i$$

For any support vector xi, this formula will give the same value of b. Once the optimal w and b are obtained, the classifier is ready to make predictions.

In the case of non-linearly separable data, SVM can be extended by using a kernel trick to map the data into a higher-dimensional feature space where a linear hyperplane can separate the classes. The kernel function K(xi, xj) computes the inner product between two data points in the higher-dimensional space, and the optimization problem is rewritten in terms of the kernel function. The kernel function is typically chosen to be a Radial Basis Function (RBF) kernel, polynomial kernel, or linear kernel, depending on the nature of the data. The SVM optimization problem in the kernelized form is:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \; !$$

subject to:

$$\alpha_i \geq 0, \quad \sum_{i=1} \alpha_i y_i = 0$$

This approach allows SVMs to handle complex, non-linear decision boundaries. The use of the kernel trick makes SVM a powerful tool for many practical machine learning tasks, especially when the data is not linearly separable in its original feature space.

Once the model is trained, the prediction mechanism involves computing the decision function for a new data point x. For the linear case, the decision function is:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right)$$

For the kernelized case, the decision function is:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b\right)$$

This decision function computes the sign of a weighted sum of the kernel values between the new point x and each of the support vectors, along with the bias term b. If the result is positive, the data point is classified into the positive class; otherwise, it is classified into the negative class.

Training an SVM typically involves solving a convex optimization problem, which can be computationally expensive, especially when the number of data points is large. However, various techniques such as Sequen- tial Minimal Optimization (SMO) have been developed to efficiently solve the SVM optimization problem. Moreover, for large-scale datasets, approximations like the stochastic gradient descent (SGD) algorithm can be used to train SVMs more efficiently.

In summary, the Support Vector Machine is a powerful machine learning model that works by finding a hyperplane that separates the classes of data with the maximum margin. The model is defined by a weight vector and a bias term, and it is trained by solving a constrained optimization problem. In the case of non-linearly separable data, the kernel trick is used to map the data into a higher-dimensional space, where a linear decision boundary can effectively separate the classes.

• Logistic Regression: Logistic Regression is a fundamental statistical and machine learning technique used primarily for binary classification tasks. It models the relationship between a set of independent variables (features) and a binary dependent variable (label) by estimating probabilities that a given input point belongs to a particular class. Despite its name, logistic regression is a classification algorithm rather than a regression algorithm because it predicts categorical outcomes. The core idea behind logistic regression is to model the probability of a binary outcome using a logistic function, also known as the sigmoid function. The model's formulation involves a weighted sum of the input features, passed through a logistic function to constrain the output to the range between 0 and 1. Mathematically, the logistic regression model can be expressed as:

$$P(y = 1 | \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}'\mathbf{x}}}$$

Here, P (y = 1 x) represents the probability that the output y is 1 (i.e., belonging to the positive class), given the input vector x. The vector w contains the model's weights, and the term wT x is the dot product between the weight vector and the input feature vector. The logistic function σ(z) = 1 , applied to this linear combination of inputs, squashes the result into the range [0, 1], making it interpretable as a probability.

The training procedure for logistic regression involves finding the optimal set of weights w that minimizes a cost function. The most commonly used cost function in logistic regression is the log-likelihood function, which measures how well the model predicts the observed data. For a dataset with N instances, the log-likelihood for logistic regression is defined as:

$$\ell(\mathbf{w}) = \sum_{i=1}^{N} [y_i \log P(y_i = 1|\mathbf{x}_i) + (1 - y_i) \log(1 - P(y_i = 1|\mathbf{x}_i))]$$

In this equation, yi denotes the true label for the i-th instance, and xi is the corresponding feature vector. The log-likelihood function essentially sums the logarithms of the predicted probabilities for each instance, penalizing the model more heavily for making incorrect predictions. The objective in logistic regression is to maximize the log-likelihood function, which is equivalent to minimizing the negative log-likelihood, also known as the cross-entropy loss function. In practice, the optimization process involves using methods like gradient descent to update the weights iteratively to minimize the cost.

The gradient of the log-likelihood with respect to the weights w is computed to guide the weight update during training. The gradient for the j-th weight is given by:

$$\frac{\partial \ell(\mathbf{w})}{\partial w_j} = \sum_{i=1}^{N} (y_i - P(y_i = 1|\mathbf{x}_i))\, x_{ij}$$

where xij is the j-th feature of the i-th input instance. The term (yi − P (yi = 1|xi)) represents the error between the true label and the predicted probability for the i-th instance. This error is multiplied by the corresponding feature value xij to form the gradient component for the weight wj . Once the gradients are computed, the weights are updated using an optimization algorithm like gradient descent. The weight update rule for gradient descent is:

$$w_j \leftarrow w_j + n \frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

where η is the learning rate, a hyperparameter that controls the step size of each update. This iterative process is repeated until the weights converge to values that minimize the negative log-likelihood or until a predefined number of iterations is reached.

In practice, gradient descent may be replaced by more sophisticated optimization methods, such as stochas- tic gradient descent (SGD), which updates the weights after evaluating each individual sample or a small mini-batch of samples. This can significantly speed up the training process, especially for large datasets. Another alternative is the use of second-order methods like Newton’s method, which leverages the Hessian matrix (the matrix of second-order partial derivatives of the cost function) to more efficiently find the optimal solution. However, these methods are computationally expensive and may not scale well for large datasets.

Once the weights have been learned during the training phase, the model can be used to make predictions on new, unseen data. The prediction process in logistic regression involves calculating the probability that an instance belongs to the positive class (i.e., y = 1). For a given test instance with feature vector x, the model computes the linear combination of the input features, wT x, and passes it through the logistic function to obtain the predicted probability:

$$P(\hat{y} = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}$$

If this predicted probability is greater than a certain threshold (typically 0.5), the model classifies the instance as belonging to the positive class (i.e., yˆ = 1), otherwise, it classifies it as belonging to the negative class (i.e., yˆ = 0):

$$\hat{y} = \begin{cases} 1 & \text{if } P(\hat{y} = 1|\mathbf{x}) \geq 0.5 \\ 0 & \text{if } P(\hat{y} = 1|\mathbf{x}) < 0.5 \end{cases}$$

This decision rule effectively translates the probability output from the logistic regression model into a binary class prediction. In cases where the threshold needs to be adjusted (for example, when dealing with imbalanced datasets), the threshold can be set to a value other than 0.5, depending on the desired trade-off between precision and recall.

The logistic regression model assumes that the relationship between the features and the log-odds of the outcome is linear. Specifically, it assumes that the log-odds of the probability P (y = 1|x) are a linear function of the input features, i.e.,

$$\log\left(\frac{P(y = 1|\mathbf{x})}{1 - P(y = 1|\mathbf{x})}\right) = \mathbf{w}^T\mathbf{x}$$

This equation is the logit function, and the model is named after it. The logit function maps the probability P (y = 1|x) into the entire real line, which makes the linear model suitable for predicting probabilities. The assumption of a linear relationship between the input features and the log-odds is a simplification, and while it often works well in practice, it may not hold in more complex scenarios. Extensions of logistic regression, such as polynomial logistic regression or the use of regularization techniques like L1 (Lasso) or L2 (Ridge) regularization, can help address these limitations and prevent overfitting by controlling the complexity of the model.

Regularization plays a crucial role in preventing overfitting, especially when the number of features is large relative to the number of data points. In the case of L2 regularization (Ridge), the cost function becomes:

$$\ell_{\text{reg}}(\mathbf{w}) = \ell(\mathbf{w}) - \lambda \sum_{j=1}^{d} w_j^2$$

where λ is the regularization parameter that controls the trade-off between fitting the data and penalizing large weights. The gradient of this regularized cost function with respect to the weights is:

$$\frac{\partial \ell_{\text{reg}}(\mathbf{w})}{\partial w_j} = \frac{\partial \ell(\mathbf{w})}{\partial w_j} - 2\lambda w_j$$

The regularization term −λ Σd w2 helps to shrink the weights, preventing the model from overfitting the training data by making it less sensitive to small fluctuations in the input features.

In summary, logistic regression is a powerful and interpretable model for binary classification tasks, based on the principle of modeling probabilities with the logistic function. Its simplicity, efficiency, and flexibility make it a popular choice in many practical machine learning applications. The training procedure involves maximizing the log-likelihood (or minimizing the cross-entropy loss) to learn the optimal set of weights. Optimization methods like gradient descent are used to iteratively update the weights, and regularization techniques like L2 regularization help prevent overfitting. Despite its simplicity, logistic regression forms the foundation for more complex models and is often a good starting point in a machine learning pipeline.

3.3.1 Evaluation Metrics

We evaluated the performance of each model using the following metrics:

- Accuracy: The percentage of correct predictions out of total predictions.
- Precision, Recall, F1-score: These metrics provide a more detailed evaluation, especially in imbalanced datasets.
- Confusion Matrix: To visualize the true positives, true negatives, false positives, and false negatives.

4. Experiments and Results

4.1 Experimental Setup

For each algorithm, we trained the model on the scaled training data and evaluated its performance on the scaled test data. Hyperparameters such as the number of neighbors for KNN is set to 5, the depth of the decision tree is arbitrary, regularization parameters for logistic regression were set to 1 and for SVM kernel was set to radial basis function.

4.2 Results

The performance of the models is summarized in the following table:

Table 1: Performance comparison of the classification models

Model	**Accuracy**	**Precision**	**Recall**	**F1-Score**
k-Nearest Neighbors	0.83	0.81	0.81	0.81
Naive Bayes	0.76	0.74	0.72	0.81
Decision Tree	0.96	0.96	0.95	0.96
Random Forest	0.98	0.99	0.98	0.98
Support Vector Machine	0.82	0.81	0.78	0.79
Logistic Regression	0.77	0.76	0.71	0.72

The Figure 6 and Figure 10 shows confusion matrix of the models:

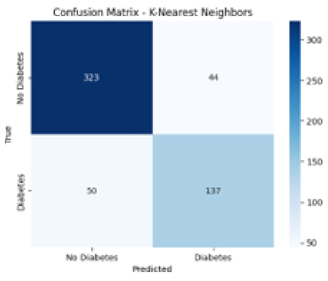

Figure 3: KNN

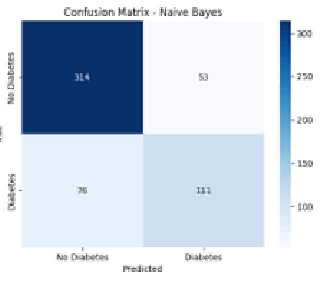

Figure 4: Naive Bayes

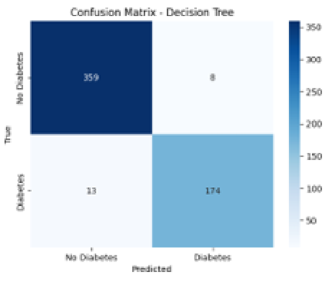

Figure 5: Decision Tree

Figure 6: Confusion Matrices of first three classifiers

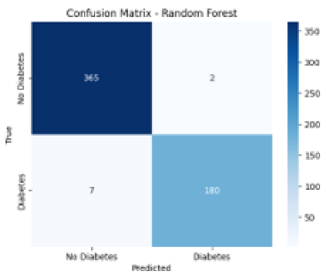

Figure 7: Random Forest

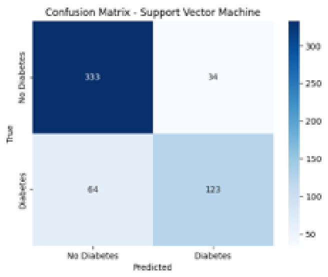

Figure 8: Support Vector Machine

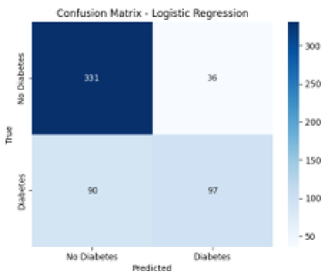

Figure 9: Logistic Regression

Figure 10: Confusion Matrices of last three classifiers

5. Conclusion

In this study, we compared six classification algorithms for predicting diabetes based on a set of medical features. Random Forest emerged as the best-performing model in terms of accuracy, precision, recall, and F1-score. It effectively handled the non-linearity in the data and provided robust predictions. Decision Tree also demonstrated strong performance, particularly in terms of precision, while KNN Regression and Support Vector Machine showed moderate results. Naive Bayes and Logistic Regression were the worst performers.

References

1. Ali, A., Alrubei, M., Hassan, L. M., Al-Ja'afari, M., & Abdulwahed, S. (2020). Diabetes classification based on knn. IIUM Engineering Journal , 21 (1), 175–181.
2. Boudjemadi, R., Jamila, M., Lunn, J., & Aljumaili, W. (2021). The implementation of ai in health and medicine: Electronic health records web based on integration of logistic regression model for diabetes type 2 prediction. In 2021 14th international conference on developments in esystems engineering (dese) (pp. 168–173).

3. Chaki, J., Ganesh, S. T., Cidham, S., & Theertan, S. A. (2022). Machine learning and artificial intelligence based diabetes mellitus detection and self-management: A systematic review. Journal of King Saud University-Computer and Information Sciences, 34 (6), 3204–3225.
4. Diabetes dataset. (2024). https://www.kaggle.com/datasets/loayosama/healthcare-dataset. (Accessed: 2024-12-03)
5. Ellahham, S. (2020). Artificial intelligence: the future for diabetes care. The American journal of medicine, 133 (8), 895–900.
6. HL, G., Ravi, V., Almeshari, M., Alzamil, Y., et al. (2023). Electronic health record (ehr) sys- tem development for study on ehr data-based early prediction of diabetes using machine learning algorithms. The Open Bioinformatics Journal , 16 (1).
7. Kenner, B. J., Abrams, N. D., Chari, S. T., Field, B. F., Goldberg, A. E., Hoos, W. A., . . . others (2021). Early detection of pancreatic cancer: applying artificial intelligence to electronic health records. Pancreas, 50 (7), 916–922.
8. Khalifa, M., & Albadawy, M. (2024). Artificial intelligence for diabetes: Enhancing prevention, diagnosis, and effective management. Computer Methods and Programs in Biomedicine Update, 100141.
9. Kumari, V. A., Chitra, R., et al. (2013). Classification of diabetes disease using support vector machine. International Journal of Engineering Research and Applications, 3 (2), 1797–1801.
10. Menon, S. P., Shukla, P. K., Sethi, P., Alasiry, A., Marzougui, M., Alouane, M. T.-H., & Khan, A. (2023). An intelligent diabetic patient tracking system based on machine learning for e-health applications. Sensors, 23 (6), 3004.
11. Nguyen, B. P., Pham, H. N., Tran, H., Nghiem, N., Nguyen, Q. H., Do, T. T., . . . Simpson, C. R. (2019). Predicting the onset of type 2 diabetes using wide and deep learning with electronic health records. Computer methods and programs in biomedicine, 182 , 105055.
12. Pang, H., Zhou, L., Dong, Y., Chen, P., Gu, D., Lyu, T., & Zhang, H. (2024). Electronic health records-based data-driven diabetes knowledge unveiling and risk prognosis. arXiv preprint arXiv:2412.03961 .
13. Patil, A. R., Mane, S. C., Patil, M. A., Gangurde, N. A., Rahate, P. G., & Dhanke, J. A. (2024). Artificial intelligence and machine learning techniques for diabetes healthcare: A review. Journal of Chemical Health Risks, 1058–1063.
14. Permana, B., Ahmad, R., Bahtiar, H., Sudianto, A., & Gunawan, I. (2021). Classification of diabetes disease using decision tree algorithm (c4. 5). In Journal of physics: Conference series (Vol. 1869, p. 012082).
15. Priya, K. L., Kypa, M. S. C. R., Reddy, M. M. S., & Reddy, G. R. M. (2020). A novel approach to predict diabetes by using naive bayes classifier. In 2020 4th international conference on trends in electronics and informatics (icoei)(48184) (pp. 603–607).
16. Rajendra, P., & Latifi, S. (2021). Prediction of diabetes using logistic regression and ensemble techniques. Computer Methods and Programs in Biomedicine Update, 1 , 100032.
17. Rashid, M. M., Askari, M. R., Chen, C., Liang, Y., Shu, K., & Cinar, A. (2022). Artificial intelligence algorithms for treatment of diabetes. Algorithms, 15 (9), 299.
18. Rigla, M., Garc´ıa-S´aez, G., Pons, B., & Hernando, M. E. (2018). Artificial intelligence methodologies and their application to diabetes. Journal of diabetes science and technology , 12 (2), 303–310.
19. Roy, M., Vasudeva, S., & Jamwal, M. (n.d.). A conceptual framework architecture for ai-based remote patient monitoring in diabetes (rpm-d): Gap analysis and feasibility assessment. system, 5 , 6.
20. Zheng, T., Xie, W., Xu, L., He, X., Zhang, Y., You, M., . . . Chen, Y. (2017). A machine learning-based framework to identify type 2 diabetes through electronic health records. International journal of medical informatics, 97 , 120–127.