<<<< SUPERVISED MACHINE LEARNING ALGORITHMS - By SURAJ>>>>

Table of contents

No. Concept

26. Multiclass Logistic Regression (Softmax Regression).
27. Restricted Boltzmann Machines.
28. Conditional Random Fields.
29. Multi-output regression algorithms (Multi output Decision Trees, SVR).
30. Stochastic Gradient Descent Classifier(SGD).
31. Radial Basis Function(RBF) Networks.
32. Adaboost.
33. Random Sample Consensus(RANSAC)

<<<<<< EXPLANATIONS >>>>>>>>>>>>

Concept 00: Linear Regression

Linear regression is a simple algorithm used for predicting a continuous output variable based on one or more input features. The goal of linear regression is to find the best-fitting line through the data points that minimizes the sum of squared errors.

Mathematically, linear regression can be represented as follows:

Given a set of input features represented by a matrix X with dimensions (n x m), where n is the number of data points and m is the number of features, and a corresponding vector of target values y with dimensions (n x 1), the objective is to find the best-fitting weight vector w of dimensions (m x 1) and a bias term b that minimizes the sum of squared errors (SSE):

minimize: $SSE(w, b) = 1/n * \sum(y - (Xw + b))^2$

where (Xw + b) represents the predicted target values.

To find the optimal values of w and b, we can use a technique called ordinary least squares (OLS) method. By taking the derivative of SSE with respect to w and b, and setting them to zero, we can solve for w and b as follows:

$$w = (X^T X)^{-1} X^T y$$
$$b = \text{mean}(y) - \text{mean}(X) * w$$

where X^T represents the transpose of X, and $^{-1}$ represents the matrix inverse.

Intuitively, linear regression can be understood as finding the best-fitting line that represents the relationship between the input features and the target variable. For example, in a simple case where we have only one input feature (univariate linear regression), we can visualize the data points on a scatter plot and the best-fitting line as a straight line that passes through the points in a way that minimizes the SSE.

Example: Suppose we have a dataset of housing prices with the living area as the input feature (in square feet) and the corresponding prices as the target variable (in thousands of dollars). We can use linear regression to find the best-fitting line that predicts the prices based on the living area. The weight w would represent the price per square foot, and the bias b would represent the base price.

Let's assume we have a dataset of 50 data points with the living area (in square feet) as the input feature denoted by X and the corresponding prices (in thousands of dollars) as the target

variable denoted by y. We can represent the input feature X as a column vector of dimensions (50 x 1) and the target variable y as a column vector of dimensions (50 x 1).

$X = [x1\ x2\ \dots\ x50]$

$y = [y1\ y2\ \dots\ y50]$

where xi represents the living area of the i-th data point and yi represents the price of the i-th data point.

Now, we can represent the linear regression problem mathematically as follows:

$\text{minimize: } SSE(w, b) = 1/50 * \sum(yi - (w * xi + b))^2$

where w is the weight (or slope) of the best-fitting line and b is the bias (or y-intercept) of the line.

To find the optimal values of w and b, we can use the OLS method. First, we need to represent the input feature X as a matrix of dimensions (50 x 2), where the first column represents the living area x and the second column represents a column of ones for the bias term b. We can then apply the formula for w and b as follows:

$X = [[x1, 1]$
$[x2, 1]$
$\dots$
$[x50, 1]]$

$w = (X^T X)^{-1} X^T y$
$b = \text{mean}(y) - \text{mean}(X) * w$

By substituting the values of X, y, and solving the above equations, we can obtain the optimal values of w and b that minimize the SSE. The resulting w would represent the price per square foot, and the b would represent the base price.

Once we have the optimal values of w and b, we can use them to make predictions on new data points. For example, if we have a new living area x_new, we can plug it into the equation of the best-fitting line (w * x_new + b) to obtain the predicted price for that living area.

Intuitively, linear regression finds the best-fitting line that captures the underlying linear relationship between the input features and the target variable. However, it is important to note that linear regression assumes a linear relationship between the input features and the target variable, and may not perform well if the relationship is non-linear. In such cases, other algorithms such as polynomial regression or decision trees may be more appropriate.

2. Logistic Regression

Logistic regression is a type of supervised machine learning algorithm used for binary classification tasks, where the target variable has only two possible classes. It is used to estimate the probability that an input data point belongs to a certain class. For example, it can be used to predict whether an email is spam or not spam, whether a customer will churn or not churn, or

whether a patient has a particular disease or not.

The main idea behind logistic regression is to model the probability of an event occurring using a logistic function (also known as sigmoid function), which maps the predicted values to a probability value between 0 and 1. The logistic function is given by:

sigmoid(z) = 1 / (1 + exp(-z))

where z is the weighted sum of the input features x and the corresponding weights w, plus a bias term b:

z = w * x + b

The logistic regression algorithm aims to learn the optimal values of w and b from the training data, so that the predicted probabilities are as close as possible to the actual class labels.

The objective function for logistic regression is the maximum likelihood estimation (MLE) or the cross-entropy loss function, which measures the difference between the predicted probabilities and the actual class labels. The cross-entropy loss for a binary classification problem is given by:

L(y, y_hat) = -[y * log(y_hat) + (1 - y) * log(1 - y_hat)]

where y is the actual class label (either 0 or 1), y_hat is the predicted probability from the sigmoid function, and the negative sign is added to convert the maximization problem of MLE into a minimization problem.

To find the optimal values of w and b that minimize the cross-

loss, we can use gradient descent optimization or other optimization techniques. Gradient descent involves iteratively updating the weights and biases using the gradients of the loss function with respect to w and b:

$w = w - learning\_rate * \partial L/\partial w$
$b = b - learning\_rate * \partial L/\partial b$

where learning_rate is a hyperparameter that controls the step size of the updates, and $\partial L/\partial w$ and $\partial L/\partial b$ are the partial derivatives of the loss function with respect to w and b, respectively.

Once the optimal values of w and b are obtained, we can use them to make predictions on new data points. For example, given a new input data point X_new, we can calculate the weighted sum z using the learned values of w and b, and then pass it through the sigmoid function to obtain the predicted probability y_hat. We can then threshold the predicted probability (e.g., if y_hat >= 0.5, predict class 1, otherwise predict class 0) to make the final binary classification decision.

Logistic regression is a simple and interpretable algorithm that can be used for binary classification tasks when the relationship between the input features and the target variable is assumed to be linear. However, like linear regression, it may not perform well when the relationship is non-linear, and other algorithms such as decision trees, support vector machines, or neural networks may be more appropriate in such cases.

Example - Suppose we have a dataset of emails labeled as spam (1) or not spam (0), and we want to build a binary classifier to

predict whether a given email is spam or not based on certain features such as the length of the email, the presence of certain keywords, and the number of exclamation marks. We can use logistic regression for this task.

We start by preparing our data, which involves splitting it into a training set and a test set. The training set is used to train our logistic regression model, and the test set is used to evaluate its performance.

Next, we define our logistic regression model, which consists of a sigmoid function to model the predicted probabilities, and a cross-entropy loss function to measure the difference between the predicted probabilities and the actual class labels.

We then use an optimization algorithm, such as gradient descent, to find the optimal values of the weights and biases that minimize the cross-entropy loss. This involves iteratively updating the weights and biases based on the gradients of the loss function with respect to the weights and biases.

Once the model is trained, we can use it to make predictions on new data points. For example, given a new email with features such as length, presence of keywords, and number of exclamation marks, we can calculate the weighted sum of the features using the learned weights and biases, and pass it through the sigmoid function to obtain the predicted probability that the email is spam. We can then threshold this probability to make the final binary classification decision.

After training and evaluating our logistic regression model, we can analyze its performance using various evaluation metrics such

as accuracy, precision, recall, and F1 score, and make any necessary adjustments to improve its performance.

This is a high-level example of how logistic regression can be used for binary classification tasks. It's important to note that the actual implementation and steps may vary depending on the programming language or framework used, and the specific requirements of the problem at hand.

3. Decision Trees

Decision Trees are a popular supervised machine learning algorithm used for both classification and regression tasks. They work by recursively splitting the input feature space into regions or segments that are homogenous with respect to the target variable.

The basic idea behind a Decision Tree is to divide the feature space into segments or regions that are as pure as possible in terms of class labels or target values. This is achieved by selecting the best feature and a corresponding threshold at each node of the tree, such that the resulting split maximizes the purity or homogeneity of the segments.

The purity or homogeneity of the segments is typically measured using metrics such as entropy or Gini impurity for classification tasks, and mean squared error (MSE) or mean absolute error (MAE) for regression tasks.

Let's take an example to illustrate Decision Trees for classification:

Suppose we have a dataset of patients with a binary classification task of predicting whether they have diabetes or not based on features such as age, BMI, blood pressure, and insulin levels. We can use a Decision Tree to build a classifier for this task.

We start by selecting the best feature and threshold to split the data at the root node of the tree. This is done by calculating the impurity of the segments created by different feature-threshold combinations, and selecting the combination that results in the lowest impurity.

We then recursively repeat this process for each child node until we reach the leaf nodes of the tree. The leaf nodes represent the final segments or regions in the feature space, and their class labels are determined based on the majority class or average target value of the instances in the segment.

Once the Decision Tree is built, we can use it to make predictions on new data points by traversing the tree from the root node to a leaf node, and assigning the corresponding class label of the leaf node to the data point.

Decision Trees are interpretable and can capture non-linear relationships in the data. However, they can also suffer from overfitting, as they can become too complex and overly specific to the training data. Techniques such as pruning, setting a maximum depth, or using ensemble methods like Random Forests can be used to mitigate overfitting and improve the generalization performance of Decision Trees.

Gini impurity and entropy are two common measures used to calculate the purity or homogeneity of segments in a Decision

Tree for classification tasks.

Gini Impurity:
Gini impurity measures the probability of misclassification of an element chosen uniformly at random from a segment. It is defined mathematically as:
Gini Impurity = 1 - Σ (pi^2)

where pi represents the probability of each class in the segment. A lower Gini impurity indicates a purer segment with fewer misclassifications.

Let's take an example to illustrate Gini impurity:

Suppose we have a segment of data with 10 instances, where 6 instances belong to class A and 4 instances belong to class B. The probabilities of class A and class B in this segment are 6/10 and 4/10, respectively.

Gini impurity = 1 - ((6/10)^2 + (4/10)^2)
= 1 - (36/100 + 16/100)
= 1 - 52/100
= 48/100
= 0.48

So, the Gini impurity of this segment is 0.48, indicating that it is not completely pure.

Entropy:
Entropy is another measure of the impurity or randomness of a segment in a Decision Tree. It is defined mathematically as:
Entropy = - Σ (pi * log2(pi))

where pi represents the probability of each class in the segment, and log2 is the base-2 logarithm. A lower entropy indicates a purer segment with less randomness.

Let's take an example to illustrate entropy:

Suppose we have a segment of data with 10 instances, where 7 instances belong to class A and 3 instances belong to class B. The probabilities of class A and class B in this segment are 7/10 and 3/10, respectively.

Entropy = - ((7/10) * log2(7/10) + (3/10) * log2(3/10))
≈ - (0.356 * log2(0.356) + 0.204 * log2(0.204))
≈ - (0.356 * (-1.862) + 0.204 * (-2.304))
≈ - (-0.663 + 0.471)
≈ -0.192

So, the entropy of this segment is approximately 0.192, indicating that it is not completely pure.

Both Gini impurity and entropy are used as splitting criteria in Decision Trees, where the feature and threshold combination that results in the lowest Gini impurity or entropy is selected as the optimal split at each node. These measures help in building a Decision Tree that can create pure and homogenous segments, leading to accurate predictions on new data points.

Information Gain:
Information gain is a measure used in Decision Trees to evaluate the effectiveness of a feature in splitting the data into pure segments. It is calculated as the difference between the impurity (Gini impurity or entropy) of the parent node and the

weighted average impurity of the child nodes after the split. Mathematically, information gain is given by:

Information Gain = Impurity(parent) - Σ ((ni/n) * Impurity(child_i))

where Impurity(parent) is the impurity of the parent node, ni is the number of instances in the i-th child node, n is the total number of instances in the parent node, and Impurity(child_i) is the impurity of the i-th child node.

A higher information gain indicates a better split, as it leads to greater purity in the child nodes and more accurate predictions.

Let's take an example to illustrate information gain:

Suppose we have a parent node with 100 instances, where 60 instances belong to class A and 40 instances belong to class B. The Gini impurity of the parent node is 0.48 (calculated previously).

Now, we split the data into two child nodes based on a feature, and after the split, we have 70 instances in the left child node and 30 instances in the right child node. In the left child node, 40 instances belong to class A and 30 instances belong to class B, while in the right child node, 20 instances belong to class A and 10 instances belong to class B.

The Gini impurity of the left child node is:
1 - ((40/70)^2 + (30/70)^2)
≈ 0.489

The Gini impurity of the right child node is:

1 - ((20/30)^2 + (10/30)^2)
≈ 0.444

Now, we can calculate the information gain as follows:
Information Gain = Impurity(parent) - ((70/100) * Impurity(left child) + (30/100) * Impurity(right child))
≈ 0.48 - ((70/100) * 0.489 + (30/100) * 0.444)
≈ 0.48 - (0.343 + 0.133)
≈ 0.004

So, the information gain for this split is approximately 0.004, indicating that it is not a very effective split as it does not lead to significant purity in the child nodes.

Information gain is used as a criterion to select the best feature and threshold for splitting the data at each node in a Decision Tree. The feature and threshold combination that results in the highest information gain is chosen as the optimal split, leading to a more accurate and optimal decision tree.

4. Random Forests

Random Forest is an ensemble learning technique that combines multiple Decision Trees to create a more accurate and robust model. It works by building a collection of decision trees and then making predictions by averaging or taking a majority vote of the predictions from the individual trees.
The steps involved in building a Random Forest are as follows:

- Randomly sample the training data with replacement (bootstrap) to create multiple subsets of data, called bootstrap samples.

- Build a decision tree on each bootstrap sample, but limit the number of features considered for splitting at each node to a random subset of features. This introduces randomness and diversity in the decision trees.
- Repeat step 2 a specified number of times to create a collection of decision trees, also known as an ensemble.
- To make predictions on new data, pass the data through each decision tree in the ensemble, and take the average or majority vote of the predictions from all the trees.

Random Forests have several advantages over single Decision Trees, including:

Improved accuracy: Random Forests reduce overfitting and improve generalization performance by averaging predictions from multiple trees.

Robustness: Random Forests are less sensitive to outliers and noisy data as they consider a random subset of features and data samples for building each tree.

Feature importance: Random Forests provide a measure of feature importance, which can be helpful in feature selection and understanding the importance of different features in making predictions.

Scalability: Random Forests can handle large datasets with a large number of features efficiently.
Random Forests are widely used in various machine learning tasks, such as classification, regression, and feature selection, and are known for their accuracy and robustness in handling complex datasets.

Example

Let's consider a binary classification problem where we have a dataset of 1000 samples with 10 features (X1, X2, ..., X10) and a binary target variable (Y) with classes 0 and 1.

- Randomly sample the training data with replacement (bootstrap):

Let's say we randomly sample 80% of the data (800 samples) with replacement to create a bootstrap sample for building each tree in the Random Forest.

- Build decision trees on bootstrap samples:

For each bootstrap sample, we build a decision tree with a maximum depth of 5 and consider only a random subset of 5 features (out of 10) for splitting at each node. This introduces randomness and diversity in the decision trees.

- Repeat step 2 a specified number of times:

Let's say we repeat step 2 for 100 trees, which means we create an ensemble of 100 decision trees.

- Make predictions:

To make predictions on new data, we pass the data through each decision tree in the ensemble. For each tree, the data follows the path of the decision tree based on the feature values, and reaches a leaf node where it is assigned a class label (0 or 1). We take the majority vote of the class labels from all the trees as the final prediction. For example, if 70 out of 100 trees predict class 1, we assign the majority class 1 to the input data point.

- Evaluate the model:

We can evaluate the performance of the Random Forest by using appropriate evaluation metrics such as accuracy, precision, recall, F1-score, etc., on a separate test dataset that was not used for training the model.

This is a high-level example of how Random Forests work in practice. The exact implementation details, such as the algorithm for building decision trees and determining feature importance, may vary depending on the specific implementation or library used, but the underlying principle of creating an ensemble of decision trees with randomness and diversity remains the same.

4. Support Vector Machines

Support Vector Machines (SVM) is a popular machine learning algorithm used for both classification and regression tasks. It is based on the idea of finding the optimal hyperplane that best separates the data into different classes or predicts the target values. SVM seeks to maximize the margin between the classes, which is the distance between the hyperplane and the closest data points from each class.

Mathematically, SVM aims to solve the following optimization problem:

For binary classification:

Minimize: 0.5 * ||w||^2 + C * sum(max(0, 1 - yi * (w.T * xi + b)))
Subject to: yi * (w.T * xi + b) >= 1 for all training examples (xi, yi)

where:

w is the weight vector of the hyperplane
b is the bias term
xi is the feature vector of the i-th training example
yi is the corresponding class label (-1 or 1)
C is the hyperparameter that controls the trade-off between maximizing the margin and minimizing the classification error. A smaller value of C allows for a wider margin but may tolerate some misclassifications, while a larger value of C leads to a narrower margin with fewer misclassifications.
For regression, the objective function is slightly different, but the basic idea of finding the optimal hyperplane still applies.

SVM uses a kernel function to transform the input data into a higher-dimensional feature space, which allows it to learn non-linear decision boundaries. Commonly used kernel functions are linear, polynomial, radial basis function (RBF), and sigmoid.

Once the optimization problem is solved, the trained SVM model can be used to predict the class labels for new data points or estimate target values for regression tasks.

Here's an example to illustrate the concept of SVM with a linear kernel for binary classification:

Suppose we have a dataset with two classes (class 0 and class 1) and two features (X1 and X2). We want to find a hyperplane that best separates the data into these two classes.

We start by randomly initializing the weight vector w and the bias term b. We then iterate over the training examples and update

w and b based on the optimization problem described above. This process continues until convergence is reached, i.e., when the weight vector w and bias term b no longer change significantly.

Once the SVM model is trained, we can use it to predict the class labels for new data points by computing w.T * x + b and checking the sign of the result. If the result is positive, the data point is predicted as class 1, and if it is negative, the data point is predicted as class 0.

SVM is a powerful algorithm that can handle complex datasets with non-linear decision boundaries. It has been widely used in various applications, such as image classification, text classification, and bioinformatics, among others.

Support Vector Machines (SVM) can use kernel functions to transform the input data into a higher-dimensional feature space, which allows for learning non-linear decision boundaries.

Kernel Functions:
Kernel functions are mathematical functions that map the original input data into a higher-dimensional space, where the data points are more separable. In SVM, the kernel function is denoted as K(x, x'), where x and x' are input data points. The kernel function computes the dot product of the transformed data points in the higher-dimensional space without explicitly calculating the transformation. Commonly used kernel functions include:

Linear Kernel: K(x, x') = x.T * x': This is a simple linear transformation that computes the dot product of the original feature vectors.

Polynomial Kernel: K(x, x') = (gamma * x.T * x' + r)^d: This is a polynomial transformation with parameters gamma, r, and d. It can capture non-linear patterns in the data by raising the dot product to a higher power d.

Radial Basis Function (RBF) Kernel: K(x, x') = exp(-gamma * ||x - x'||^2): This is a radial basis function transformation with parameter gamma. It is commonly used for capturing complex non-linear patterns in the data.

Gradient Descent:
Gradient descent is an optimization algorithm used to update the weight vector w and bias term b in SVM during training. It iteratively calculates the gradient of the objective function (e.g., the hinge loss function for SVM) with respect to w and b, and updates them in the opposite direction of the gradient to minimize the objective function.

Example:
Let's consider a binary classification problem where we have a dataset with two classes (class 0 and class 1) and two features (X1 and X2). We want to use SVM with a polynomial kernel to train a model for this problem.

1. We start by randomly initializing the weight vector w and the bias term b.
2 We calculate the hinge loss function for each training example, which measures the error between the predicted class and the true class.
3 We calculate the gradient of the hinge loss function with respect to w and b.
4 We update w and b using the gradient descent algorithm, which

involves multiplying the gradient with a learning rate and subtracting it from the current values of w and b.
5 We repeat steps 2-4 for a certain number of iterations or until convergence is reached, i.e., when the hinge loss function no longer decreases significantly.
6 Once the SVM model is trained, we can use it to predict the class labels for new data points by computing the kernel function $K(x, x')$, and checking the sign of the result after adding the bias term b.
7 We can also use the trained SVM model to estimate target values for regression tasks by modifying the objective function and loss function accordingly.

5. K -Nearest Neighbours

K-Nearest Neighbors (KNN) is a simple and popular algorithm used for both classification and regression tasks. KNN is a type of instance-based learning or lazy learning, where the algorithm stores all the training data points in memory and makes predictions based on the similarity of a new data point to the stored training data points. Let's dive into the concept of KNN and how it works.

Concept of KNN:
KNN works based on the principle of finding the K nearest neighbors of a new data point in the feature space and using the majority class (for classification) or averaging the target values (for regression) of those neighbors to make predictions.
Here are the main steps in the KNN algorithm:

Data Preparation: Prepare the data by splitting it into training and

testing sets, and normalize or scale the features if necessary.

Distance Metric: Choose a distance metric, such as Euclidean distance or Manhattan distance, to measure the similarity or dissimilarity between data points in the feature space.

K-Nearest Neighbors: Choose a value for K, the number of neighbors to consider. For a new data point, calculate the distances to all the training data points and select the K nearest neighbors based on the chosen distance metric.

Majority Voting or Averaging: For classification, calculate the majority class among the K nearest neighbors, and assign that class to the new data point. For regression, calculate the average of the target values among the K nearest neighbors, and use that as the predicted target value for the new data point.

Prediction and Evaluation: Repeat the process for all the data points in the test set, and evaluate the performance of the KNN model using appropriate evaluation metrics, such as accuracy, precision, recall, F1-score for classification, or mean squared error, mean absolute error for regression.

Mathematics:
Let's consider an example of KNN for classification, where we have a dataset with two features (X1 and X2) and two classes (class 0 and class 1).

Distance Metric: We choose the Euclidean distance as the distance metric, which is given by the formula:
distance = sqrt((X1_train - X1_test)^2 + (X2_train - X2_test)^2)

K-Nearest Neighbors: For a new data point with features (X1_test, X2_test), we calculate the distances to all the training data points (X1_train, X2_train), and select the K nearest neighbors based on the calculated distances.

Majority Voting: Among the K nearest neighbors, we count the number of occurrences of each class (class 0 or class 1), and assign the majority class as the predicted class for the new data point.

Prediction and Evaluation: We repeat the process for all the data points in the test set, and evaluate the performance of the KNN model using appropriate evaluation metrics, such as accuracy, precision, recall, F1-score, etc.

Let's take an example of KNN for classification to illustrate the concept further:

Consider a dataset with the following data points and their corresponding class labels:

Data Point X1 X2 Class
Data Point 1 2 3 Class 0
Data Point 2 4 2 Class 1
Data Point 3 3 5 Class 0
Data Point 4 5 3 Class 1
Data Point 5 6 4 Class 1
We will use this dataset for training and testing our KNN model.

Distance Metric: Let's choose the Euclidean distance as the distance metric, which is given by the formula:
distance = sqrt((X1_train - X1_test)^2 + (X2_train - X2_test)^2)

K-Nearest Neighbors: Suppose we choose k = 3, which means we will consider the 3 nearest neighbors for each test data point. For example, if we have a test data point with features X1_test = 4 and X2_test = 4, we calculate the distances to all the training data points and select the 3 nearest neighbors based on the calculated distances. Let's assume the nearest neighbors are Data Point 1, Data Point 3, and Data Point 4.

Majority Voting: Among the 3 nearest neighbors, we count the number of occurrences of each class (Class 0 or Class 1). If we have 2 neighbors with Class 0 and 1 neighbor with Class 1, we assign the majority class, which is Class 0, as the predicted class for the test data point.

Prediction and Evaluation: We repeat the process for all the data points in the test set, and evaluate the performance of the KNN model using appropriate evaluation metrics, such as accuracy, precision, recall, F1-score, etc.

This is a simplified example of how KNN works for classification. The actual implementation may involve additional steps, such as handling ties, choosing optimal values for K, handling missing values, and scaling the features.

6. Naive Bayes

Naive Bayes is a probabilistic classification algorithm that is based on the Bayes theorem. It assumes that all features are conditionally independent given the class label, which is a simplifying assumption referred to as "naive". Here's an overview of how Naive Bayes works with an example:

Consider a dataset of weather conditions (temperature, humidity, wind speed) and corresponding labels for whether people played tennis or not (class: Yes or No).

Data Point	Temperature	Humidity	Wind Speed	Play Tennis
Data 1	Hot	High	Weak	No
Data 2	Cool	Normal	Strong	Yes
Data 3	Mild	High	Strong	Yes
...	...	...	...	...

Training: We calculate the class probabilities, P(Play Tennis=Yes) and P(Play Tennis=No), by counting the occurrences of each class in the training data. We also calculate the conditional probabilities of each feature given the class label, P(Temperature=Hot|Play Tennis=Yes), P(Temperature=Cool|Play Tennis=No), P(Humidity=High|Play Tennis=Yes), P(Humidity=Normal|Play Tennis=No), and so on, by counting the occurrences of each feature value in the respective class.

Prediction: Given a new set of weather conditions (temperature, humidity, wind speed), we calculate the probability of the data point belonging to each class using Bayes theorem, which is given by the formula:
P(Play Tennis=Yes|Temperature, Humidity, Wind Speed) = P(Play Tennis=Yes) * P(Temperature|Play Tennis=Yes) * P(Humidity|Play Tennis=Yes) * P(Wind Speed|Play Tennis=Yes), and similarly for P(Play Tennis=No|Temperature, Humidity, Wind Speed).

Decision: We choose the class with the highest probability as the predicted class for the new data point.

Evaluation: We evaluate the performance of the Naive Bayes model using appropriate evaluation metrics and make necessary adjustments to the model if needed.

Note that Naive Bayes assumes that features are conditionally independent given the class label, which may not always be true in real-world scenarios. However, despite this simplifying assumption, Naive Bayes can perform surprisingly well in many practical applications, especially when the dataset is large and the features are not highly correlated.

7. Gradient Boosting Algorithms(XGBoost, LightGBM and CatBoost)

Gradient Boosting Algorithms, such as XGBoost, LightGBM, and CatBoost, are ensemble learning techniques that combine multiple weak models (typically decision trees) to create a strong predictive model. Let's discuss them in more detail:

XGBoost (eXtreme Gradient Boosting):

- XGBoost is an optimized gradient boosting framework that uses parallel computing techniques and hardware optimization to accelerate the training process and improve model performance.
- It uses a technique called "regularized boosting" that adds regularization terms to the loss function, such as L1 (Lasso) or L2 (Ridge) regularization, to prevent overfitting and improve model generalization.
- XGBoost uses a combination of pre-pruning and post-pruning techniques to control the depth and complexity of decision trees, which helps to prevent overfitting.
- The prediction for a new data point is calculated by summing the predictions from all the trees in the ensemble, weighted by

their respective learning rates.

LightGBM (Light Gradient Boosting Machine):

-LightGBM is a gradient boosting framework that uses a technique called "Gradient-based One-Side Sampling (GOSS)" to select a subset of data points for each tree in the ensemble, which reduces the computational cost and speeds up the training process.
-LightGBM uses a "leaf-wise" tree construction strategy, where it grows trees by choosing the leaf node that gives the maximum reduction in the loss function, resulting in a more balanced and accurate tree structure.
-It also supports L1 and L2 regularization, and uses a histogram-based technique for feature discretization to handle categorical features efficiently.
-LightGBM uses a "Gradient-based Early Stop (GBM)" technique that stops the training process early if the improvement in the loss function on the validation set is not significant, which helps to prevent overfitting.

CatBoost (Categorical Boosting):

- CatBoost is a gradient boosting framework that is specifically designed to handle categorical features efficiently without the need for extensive feature engineering or pre-processing.
- It uses a technique called "Ordered Boosting" that uses a permutation-based approach to handle categorical features by creating separate splits for each category.
- CatBoost also supports various techniques for handling missing values in categorical features, such as using the most common

category or creating a separate category for missing values.
- It uses a "Newton Method" for optimizing the loss function, which provides faster convergence and better accuracy compared to traditional gradient descent methods.
- CatBoost automatically handles the conversion of categorical features into numerical representations during the training process, which simplifies the model building process.

8. Neural Networks (MLP, CNNS,RNNS)

Neural networks are a type of machine learning model that are designed to mimic the functioning of the human brain. They consist of layers of interconnected nodes (neurons) that process input data and produce output predictions. Here's an overview of the internal mathematics involved in neural networks:

Activation Function: Each neuron in a neural network applies an activation function to its input, which introduces non-linearity into the model. Common activation functions include sigmoid, tanh, and ReLU (Rectified Linear Unit), among others. The activation function is typically applied element-wise to the output of each neuron.

Weighted Sum: Each neuron in a neural network computes a weighted sum of its inputs, where the weights represent the strength of the connections between neurons. Mathematically, the weighted sum is computed as the dot product of the input vector and the weight vector, plus an optional bias term. This can be expressed as:
weighted_sum = dot_product(input, weights) + bias

Feedforward: The weighted sum from each neuron is passed

through the activation function to produce the output of that neuron. This output is then used as the input for the next layer of neurons in a process known as feedforward propagation. This process is repeated for each layer in the neural network until the final output prediction is obtained.

Backpropagation: During training, the neural network is optimized to minimize the prediction error. The backpropagation algorithm is used to update the weights of the neurons in the network based on the gradient of the error with respect to the weights. This gradient is computed using the chain rule of calculus, and it allows the network to learn from the training data and adjust its weights to make better predictions.

Loss Function: The error or loss of the neural network is typically quantified using a loss function, such as mean squared error (MSE) or cross-entropy. The goal during training is to minimize the value of the loss function, which drives the model towards making better predictions.

Optimization Algorithm: During training, an optimization algorithm, such as gradient descent, is used to update the weights of the neurons in the network based on the computed gradients. These updates are performed iteratively in small steps to gradually minimize the loss function and improve the performance of the model.

Model Evaluation: Once the neural network is trained, it can be evaluated on a separate validation or test dataset to assess its performance. Common evaluation metrics for neural networks include accuracy, precision, recall, F1-score, and others, depending on the specific task.

9. Ridge Regression

Ridge Regression is a regularization technique used in linear regression to prevent overfitting by adding a penalty term to the objective function. The objective function of Ridge Regression is the same as that of ordinary linear regression, but with an additional term that is proportional to the square of the L2 norm of the regression coefficients:

Objective Function = RSS (Residual Sum of Squares) + α * summ(βi^2)

where:

RSS is the Residual Sum of Squares, which measures the sum of squared differences between the predicted and actual values of the target variable.
βi are the regression coefficients (also known as model parameters) for each feature in the dataset.
α is the regularization hyperparameter that controls the strength of the penalty term. A higher value of α results in stronger regularization, and vice versa.
The Ridge Regression algorithm minimizes this objective function to obtain the optimal values of the regression coefficients. The internal mathematics involved in Ridge Regression can be broken down into the following steps:

Data Preprocessing: Standardize or normalize the input features to ensure they are on the same scale.

Model Training: Fit the Ridge Regression model to the training data using the objective function mentioned above. The

optimization algorithm used (e.g., gradient descent) will iteratively update the regression coefficients to minimize the objective function.

Penalty Term: The penalty term (sum(βi^2)) is the sum of squared regression coefficients, which is multiplied by the regularization hyperparameter α. This term encourages smaller coefficient values, which helps in reducing the impact of irrelevant features and prevents overfitting.

Hyperparameter Tuning: Choose an appropriate value of α using techniques like cross-validation or grid search. A higher value of α will result in more regularization, while a lower value of α will result in less regularization.

Prediction: Once the model is trained, use it to make predictions on new data by applying the learned regression coefficients to the input features.

Evaluation: Evaluate the performance of the Ridge Regression model using appropriate evaluation metrics such as mean squared error (MSE), root mean squared error (RMSE), R-squared, etc.

10. Lasso Regression

Lasso Regression is a linear regression technique that adds a penalty term to the objective function to enforce feature selection by promoting sparsity in the model. It is used for feature selection and regularization to prevent overfitting in linear regression models.

The objective function of Lasso Regression is given by:

Objective function = RSS + α * sum|β_j|

where:
- RSS (Residual Sum of Squares) is the same as in ordinary linear regression, representing the sum of squared residuals between the predicted and actual values.
- β_j represents the regression coefficients (weights) for each feature j.
- α (alpha) is the regularization hyperparameter that controls the strength of the penalty term. A higher value of α results in a stronger penalty, leading to sparser models with fewer non-zero coefficients.

The key difference between Lasso Regression and ordinary linear regression is the penalty term, which is the absolute value of the regression coefficients ($|\beta_j|$), compared to the squared value (β_j^2) used in Ridge Regression. This makes Lasso Regression particularly effective for feature selection, as it tends to drive some of the coefficients exactly to zero, effectively excluding those features from the model.

To find the optimal values of the regression coefficients in Lasso Regression, we use a technique called coordinate descent or subgradient descent, which involves iteratively updating the coefficients based on the gradient of the objective function with respect to each coefficient.

The mathematics involved in updating the coefficients during each iteration of coordinate descent or subgradient descent is as follows:

$\beta_j = S(\text{sum}(x_i * (y_i - \text{sum}(x_j * \beta_j))) / N), \alpha)$

where:

- x_i is the value of the jth feature for the ith data point.
- y_i is the actual target value for the ith data point.
- β_j is the current value of the regression coefficient for the jth feature.
- N is the number of data points in the dataset.
- $S(z, \alpha)$ is the soft-thresholding function that applies the L1 penalty term with strength α to the value z. It is given by:

$S(z, \alpha) = \text{sign}(z) * \max(|z| - \alpha, 0)$

where sign(z) returns the sign of z (1 for positive values, -1 for negative values, and 0 for zero), and $\max(|z| - \alpha, 0)$ returns the maximum of $|z| - \alpha$ and 0.

II. Elastic Net

Elastic Net is a regularization technique used in machine learning for linear regression models to handle high-dimensional datasets with potential multicollinearity among features. It combines both L1 (Lasso) and L2 (Ridge) regularization methods.

The objective function of Elastic Net is given by:

Loss function = $(1/n) * \sum(y - \hat{y})^2 + \lambda 1 * \|\beta\|1 + \lambda 2 * \|\beta\|2^2$

where:

- n: Number of data points
- y: Observed target values

- $\hat{y}$: Predicted target values
- β: Regression coefficients
- $\|\beta\|1$: L1 norm of β (sum of absolute values of β)
- $\|\beta\|2^2$: L2 norm of β (sum of squared values of β)
- $\lambda 1$: Hyperparameter controlling the strength of L1 regularization (Lasso)
- $\lambda 2$: Hyperparameter controlling the strength of L2 regularization (Ridge)

The Elastic Net algorithm uses an iterative optimization process, such as coordinate descent or gradient descent, to find the optimal values of β that minimize the above objective function. During each iteration, the algorithm updates the regression coefficients β by taking into account both the L1 and L2 regularization terms, scaled by the respective hyperparameters $\lambda 1$ and $\lambda 2$.

The Elastic Net algorithm provides a trade-off between Lasso and Ridge regularization. The L1 regularization encourages sparsity in the model, resulting in some of the regression coefficients being exactly zero, effectively performing feature selection. The L2 regularization, on the other hand, helps in handling multicollinearity among features by shrinking the coefficients towards zero. The hyperparameters $\lambda 1$ and $\lambda 2$ control the amount of regularization applied, with higher values resulting in stronger regularization.

Elastic Net is commonly used in situations where there are many correlated features in the dataset and feature selection is desired, while also accounting for potential multicollinearity among features. It is a powerful technique for regularized linear regression and can be adjusted by tuning the hyperparameters $\lambda 1$

and $\lambda 2$ to achieve the desired level of regularization.