12. Principal Component Regression

PCR is a technique used in machine learning for regression problems that combines Principal Component Analysis (PCA) with linear regression. It is used to reduce the dimensionality of a dataset by creating new variables, called principal components (PCs), from the original features, and then using these PCs as predictors in a linear regression model.

The steps involved in PCR are as follows:

Perform PCA on the original feature matrix X to obtain the principal components. PCA is a technique that transforms the original features into a new set of orthogonal variables that capture the most important information in the data. The first principal component (PC1) explains the largest amount of variance in the data, and subsequent PCs explain decreasing amounts of variance.

Select a subset of the principal components to be used as predictors in the linear regression model. This can be done by choosing a fixed number of PCs or by setting a threshold on the amount of variance explained by the PCs.

Fit a linear regression model using the selected principal components as predictors and the target variable y as the response variable.

The mathematical formulation of PCR can be expressed as follows:

Perform PCA on the original feature matrix X:

$X = U * D * V^T$

where:

X: original feature matrix with dimensions (n x p), where n is the number of data points and p is the number of original features.
U: orthogonal matrix of left singular vectors with dimensions (n x p), representing the principal components.
D: Diagonal matrix of singular values with dimensions (p x p), representing the amount of variance explained by each principal component.
V^T: Transpose of the orthogonal matrix of right singular vectors with dimensions (p x p).
Select a subset of the principal components:
$PCs = U(:, 1:k)$

where:

k: Number of principal components to be selected, typically chosen based on a fixed number or a threshold on the amount of variance explained.
Fit a linear regression model:
$y = X * \beta + \varepsilon$

where:

y: Target variable with dimensions (n x 1).
X: Selected principal components with dimensions (n x k).
β: Regression coefficients with dimensions (k x 1).
ε: Error term with dimensions (n x 1).
An example of PCR can be illustrated as follows:

Suppose we have a dataset with 100 samples and 10 features. We perform PCA on this dataset and obtain 5 principal components that explain 90% of the total variance. We then use these 5 PCs as predictors in a linear regression model with the target variable y. The resulting PCR model combines the reduced-dimensional representation of the data using the 5 PCs with the linear regression coefficients β to make predictions on the target variable y.

13. Partial least squares(PLS)

(PLS) regression is a statistical method used for analyzing the relationships between a set of independent variables (X) and a set of dependent variables (Y), where there are potentially many correlated independent variables. PLS regression aims to create new linear combinations of the original independent variables (X) that capture the maximum covariance with the dependent variables (Y).

Here's the mathematical formulation of PLS regression:

PLS Regression Model:
Given n samples with p independent variables (X) and q dependent variables (Y).
X is an n x p matrix, where each row represents a sample and each column represents a feature.
Y is an n x q matrix, where each row represents a sample and each column represents a dependent variable.
The PLS regression model can be expressed as:

$X = T * P + E$
$Y = U * Q + F$

where:

X: The original independent variable matrix
Y: The original dependent variable matrix
T: Scores matrix for X, which captures the maximum covariance between X and Y
U: Scores matrix for Y, which captures the maximum covariance between X and Y
P: Loadings matrix for X, which represents the weights of the original independent variables in the new linear combinations
Q: Loadings matrix for Y, which represents the weights of the original dependent variables in the new linear combinations
E: Residuals matrix for X, representing the unexplained variance in X after accounting for T
F: Residuals matrix for Y, representing the unexplained variance in Y after accounting for U
PLS Algorithm:
The PLS algorithm iteratively calculates the scores matrix T and U, and the loadings matrix P and Q.
At each iteration, the algorithm calculates the weights for X (P) and Y (Q) by maximizing the covariance between the scores T and U.
The scores T and U are updated iteratively until convergence is achieved.
The final PLS model can be used for prediction or further analysis.

Example:

Let's consider an example where we have a dataset with 100 samples and 10 independent variables (X1, X2, ..., X10) and a single dependent variable (Y).

We want to build a PLS regression model to predict Y based on X1 to X10.
The PLS algorithm will calculate the scores matrix T and U, as well as the loadings matrix P and Q, which represent the linear combinations of the original variables that capture the maximum covariance with Y.
The resulting PLS model can be used for predicting Y for new samples based on their values of X1 to X10.

14. Gradient Descent Algorithms(Stochastic Gradient Descent, Mini-batch gradient descent)

Gradient Descent is an optimization algorithm used in machine learning to find the optimal values of parameters in a model by minimizing the loss or cost function. There are different variants of Gradient Descent, including Stochastic Gradient Descent (SGD) and Mini-batch Gradient Descent. Let's look at each of them in detail, along with the mathematics behind them:

Gradient Descent (GD):
Gradient Descent updates the model parameters by taking small steps in the direction of the negative gradient of the loss function with respect to the parameters. The update rule for GD is given by:
$\theta_j = \theta_j - \alpha * \partial J(\theta)/\partial \theta_j$

where:

θ_j: Parameter to be updated
α: Learning rate (hyperparameter) that controls the step size of the update

$J(\theta)$: Loss or cost function

$\partial J(\theta)/\partial \theta_j$: Partial derivative of the loss function with respect to the parameter θ_j, which gives the direction of steepest increase in the loss function

GD computes the gradient of the entire training dataset at each iteration, making it computationally expensive for large datasets.

Stochastic Gradient Descent (SGD):

SGD updates the model parameters by considering only one training data point at a time, rather than the entire dataset as in GD. The update rule for SGD is given by:

$$\theta_j = \theta_j - \alpha * \partial J_i(\theta)/\partial \theta_j$$

where:

$\partial J_i(\theta)/\partial \theta_j$: Partial derivative of the loss function for the i-th training data point with respect to the parameter θ_j

SGD is computationally more efficient compared to GD as it processes one data point at a time, but it introduces more noise in the parameter updates and can result in a more erratic convergence path.

Mini-batch Gradient Descent:

Mini-batch Gradient Descent is a compromise between GD and SGD, where a mini-batch of b training data points (where b is a hyperparameter) is used to compute the gradient and update the model parameters. The update rule for Mini-batch GD is similar to SGD:

$$\theta_j = \theta_j - \alpha * \partial J_b(\theta)/\partial \theta_j$$

where:

$\partial J_b(\theta)/\partial \theta_j$: Partial derivative of the loss function for the mini-batch of b training data points with respect to the parameter θ_j

Mini-batch GD strikes a balance between the computational efficiency of SGD and the smoother convergence of GD, making it a commonly used variant in practice.

Example:

Let's consider an example of linear regression with Gradient Descent. In linear regression, the objective is to minimize the mean squared error (MSE) loss function, given by:

$$J(\theta) = (1/2m)\ \text{sum}(y - \hat{y})^2$$

where m is the number of training data points, y is the observed target values, and $\hat{y}$ is the predicted target values.

The update rule for GD would be:

$$\theta_j = \theta_j - \alpha * (1/m)\ \text{sum}(y - \hat{y}) * x_j$$

where x_j is the j-th feature of the training data.

For SGD, the update rule would be:

$$\theta_j = \theta_j - \alpha * (y - \hat{y}) * x_j$$

where y and $\hat{y}$ are the target and predicted values for a single training data point.

Similarly, for Mini-batch GD, the update rule would be:

$$\theta_j = \theta_j - \alpha * (1/b)\ \text{sum}(y - \hat{y}) * x_j$$

where b is the mini-batch size, and the sum is computed over b

training data points.

In all three cases, the learning rate α determines the step size of the parameter updates. It is a hyperparameter that needs to be tuned to achieve optimal convergence.

It's important to note that these are iterative algorithms, where the model parameters are updated iteratively until a convergence criteria is met (e.g., a certain number of iterations or a small change in the loss function). The choice of hyperparameters, such as learning rate and mini-batch size, can have a significant impact on the convergence and performance of the gradient descent algorithms.

15. Bayesian methods(Bayesian Linear regression, Bayesian Networks).

Bayesian methods are a type of statistical inference that involves updating our beliefs or knowledge about a quantity of interest based on new data. Bayesian methods are widely used in various machine learning tasks, such as Bayesian linear regression and Bayesian networks.

1. Bayesian Linear Regression:
Bayesian linear regression is an extension of classical linear regression that incorporates Bayesian principles for estimating the regression parameters. In Bayesian linear regression, we assign a prior distribution to the regression parameters, which represents our beliefs about the parameters before observing any data. Then, we update the prior distribution with the observed data using Bayes' theorem to obtain the posterior distribution, which represents our updated beliefs about the

parameters after incorporating the data.

Mathematically, Bayesian linear regression can be represented as follows:

- Prior distribution: $P(\beta)$ represents our prior beliefs about the regression parameters β, where β is a vector of regression coefficients.
- Likelihood function: $P(y|X, \beta)$ represents the likelihood of the data y given the predictor variables X and the regression coefficients β, where y is the vector of observed target values and X is the design matrix of predictor variables.
- Posterior distribution: $P(\beta|y, X)$ represents the updated beliefs about the parameters after incorporating the data, which is given by Bayes' theorem:

$$P(\beta|y, X) = (P(y|X, \beta) * P(\beta)) / P(y|X)$$

- Prediction: Once we obtain the posterior distribution of the parameters, we can use it to make predictions on new data by averaging over the posterior distribution.

2. Bayesian Networks:

Bayesian networks, also known as Bayesian belief networks or graphical models, are graphical representations of probabilistic relationships among a set of variables. They are used for probabilistic inference and can model complex dependencies and uncertainties in data. Bayesian networks consist of nodes representing variables and edges representing probabilistic dependencies between the variables.

Mathematically, Bayesian networks can be represented as follows:

- Nodes: Represent variables or random variables in the problem domain.
- Edges: Represent probabilistic dependencies between variables, where an edge from node A to node B indicates that A is a parent of B in the probabilistic sense.
- Conditional probability distributions (CPDs): Represent the probabilities of a variable given its parents in the Bayesian network, which can be learned from data or specified by domain experts.
- Joint probability distribution: Represents the joint distribution of all variables in the Bayesian network, which can be computed using the CPDs and the graphical structure of the network.

Example:
Let's consider an example of Bayesian linear regression. Suppose we want to model the relationship between the house prices (target variable) and the square footage and number of bedrooms (predictor variables) of a set of houses based on a dataset. We can use Bayesian linear regression to estimate the regression coefficients β and make predictions on new houses.

Mathematically, we can represent Bayesian linear regression for this example as follows:

- Prior distribution: $P(\beta)$ represents our prior beliefs about the regression coefficients β, which can be a Gaussian distribution centered at zero with a certain variance.
- Likelihood function: $P(y|X, \beta)$ represents the likelihood of the house prices y given the square footage X and the regression

coefficients β, which can be a Gaussian distribution with a mean given by $X\beta$ and a certain variance.
- Posterior distribution: $P(\beta|y, X)$ represents the updated beliefs about the regression coefficients β after incorporating the observed house prices y and square footage X, which can be obtained using Bayes' theorem.
- Prediction: Once we obtain the posterior distribution of β, we can use it to make predictions on new houses by averaging over the posterior distribution.

16. Discriminant Analysis(LD, Quadratic Discriminant Analysis).

Discriminant Analysis is a statistical technique used for classification tasks where the goal is to assign objects or data points to predefined classes based on their features. Both LDA and QDA are supervised learning methods that assume that the data points in each class are normally distributed.

Linear Discriminant Analysis (LDA):
LDA is a technique that finds a linear combination of features that best discriminates between different classes. The goal of LDA is to find a projection of the feature space onto a lower-dimensional subspace that maximizes the separation between classes.

Mathematics:

Let X be the feature matrix with dimensions $(n \times d)$, where n is the number of data points and d is the number of features.
Let y be the vector of class labels with dimensions $(n \times 1)$.
The goal of LDA is to find a linear discriminant function $g(x) = w^T x + b$ that best separates the classes, where w is the weight

vector and b is the bias term.
The LDA algorithm involves calculating the class means, class covariance matrices, and overall covariance matrix of the features.
The weight vector w is then obtained as the solution of the generalized eigenvalue problem $Sw^{-1}T = \lambda Sb^{-1}T$, where Sw is the within-class scatter matrix and Sb is the between-class scatter matrix.
The projection of the features onto the linear discriminant function g(x) is then used for classification.

Quadratic Discriminant Analysis (QDA):
QDA is a technique that assumes that each class has its own covariance matrix, allowing for more flexibility in modeling the distribution of each class.
Mathematics:

The mathematics of QDA is similar to that of LDA, with the main difference being that the covariance matrix is calculated separately for each class, resulting in a different covariance matrix for each class.
The weight vector w and bias term b are obtained using the same generalized eigenvalue problem as in LDA, but with the class-specific covariance matrices.
Example:
Let's take an example of a dataset with two classes, "Iris setosa" and "Iris versicolor", and two features, sepal length and sepal width. We have a total of 100 data points, with 50 data points in each class. Our goal is to classify new data points into one of the two classes based on their sepal length and sepal width.

Mathematics:

For LDA, we would calculate the class means, class covariance matrices, and overall covariance matrix of the features.
We would then solve the generalized eigenvalue problem $Sw^T = \lambda Sb^T$ to obtain the weight vector w and bias term b.
For QDA, we would calculate the class-specific covariance matrices and solve the generalized eigenvalue problem with the class-specific covariance matrices to obtain the weight vector w and bias term b.
Once we have the weight vector and bias term, we can use them to project new data points onto the linear or quadratic discriminant functions, respectively, and make class predictions.

17. Perceptron

The Perceptron is a binary classification algorithm that is used to learn a linear decision boundary between two classes. It is a type of supervised learning algorithm that falls under the category of linear classifiers.

The mathematical representation of a Perceptron is as follows:

1. Input Features: Let's assume we have `n input features denoted by $X = [x1, x2, ..., xn]$, where xi is the value of the i-th feature.

2. Weights: Each input feature is associated with a weight denoted by $W = [w1, w2, ..., wn]$, where wi is the weight associated with the i-th feature.

3. Bias: The Perceptron also has a bias term denoted by b which is

added to the weighted sum of the input features.

4. Activation Function: The Perceptron uses an activation function, typically a step function or a sign function, to determine the output class based on the weighted sum of the input features and the bias.

The output of the Perceptron is calculated as follows:

output = f(w1 * x1 + w2 * x2 + ... + wn * xn + b)

where f() is the activation function.

The Perceptron algorithm works as follows:

1. Initialize the weights and bias to random values or zeros.
2. Loop through the training data, one data point at a time.
3. Calculate the weighted sum of the input features and the bias.
4. Apply the activation function to get the predicted output.
5. Update the weights and bias based on the prediction error and learning rate.
6. Repeat steps 2-5 until a stopping criterion is met (e.g., a maximum number of iterations or convergence criteria).

An example of Perceptron is the binary classification of whether an email is spam or not based on features such as the number of words, presence of certain keywords, etc. The Perceptron can learn a linear decision boundary to classify emails as spam or not based on the weighted sum of the input features and the bias.

18. Support Vector Regression

Support Vector Regression (SVR) is a type of regression algorithm that uses the principles of Support Vector Machines (SVM) for regression problems. SVR aims to find a regression function that best fits the data while minimizing the prediction error, subject to a user-defined tolerance level.

The mathematical formulation of SVR involves the following key components:

Objective function:
The objective of SVR is to find the optimal regression function that minimizes the prediction error, while keeping the error within a user-defined tolerance level ε. The objective function is given by:
Minimize: $(1/2) * \|w\|^2 + C * \sum(\xi + \xi^*)$

subject to the constraints:

$$y - \hat{y} \leq \varepsilon + \xi$$
$$\hat{y} - y \leq \varepsilon + \xi^*$$

where:

$\|w\|$: L2 norm of the weight vector w
C: Hyperparameter controlling the trade-off between model complexity and prediction error
ξ and ξ^*: Slack variables that represent the amount by which the predicted value deviates from the true value, allowing for some tolerance ε
Decision function:

The decision function of SVR is given by:

$\hat{y} = w^* x + b$

where:

$\hat{y}$: Predicted target value

w: Weight vector

x: Input features

b: Bias term

Loss function:

The loss function of SVR is defined as the hinge loss, which measures the prediction error and is given by:

$L(\xi, \xi^*) = \max(0, |y - \hat{y}| - \varepsilon) + \max(0, |\hat{y} - y| - \varepsilon)$

where:

y: Observed target value

$\hat{y}$: Predicted target value

ε: Tolerance level

Hyperparameters:

SVR has two main hyperparameters:

C: Controls the trade-off between model complexity and prediction error. A smaller value of C will result in a wider margin and allow more training points to violate the tolerance level, while a larger value of C will result in a narrower margin and penalize violations more heavily.

ε: Sets the tolerance level for the prediction error. A smaller value of ε will result in a tighter tolerance and require the prediction error to be within a smaller range, while a larger value of ε will allow for a wider tolerance.

Example:

Let's take an example to illustrate SVR. Suppose we have a

dataset of housing prices with features like square footage, number of bedrooms, and location, and we want to predict the prices of new houses. We can use SVR to train a regression model on the dataset and make predictions.

Given a training dataset with input features $X = [x1, x2, ..., xn]$ and corresponding target values $y = [y1, y2, ..., yn]$, the SVR algorithm aims to find the optimal weight vector w and bias term b that minimize the objective function, subject to the constraints and tolerance level ε. Once the model is trained, we can use it to predict the target values $\hat{y}$ for new input features.

19. Decision Tree Regression

Decision Tree Regression is a non-parametric supervised learning algorithm used for both classification and regression tasks. It creates a tree-like structure where each internal node represents a feature, and each leaf node represents a predicted value. The decision tree is constructed by recursively splitting the data based on the values of the features, aiming to minimize the prediction error.

The main steps in building a Decision Tree Regression model are as follows:

Splitting: The decision tree is built by recursively splitting the data at each node based on the values of the features. The splitting is done in a way that minimizes the prediction error, which is typically measured using mean squared error (MSE) or any other appropriate regression metric.

Leaf node prediction: Once the data is split until a certain depth or a stopping criterion is reached, the predicted value at each leaf node is computed. This can be done by taking the mean, median, or any other appropriate statistic of the target values of the samples in that leaf node.

Pruning: After the tree is fully grown, it may be too complex and prone to overfitting. Pruning techniques such as pre-pruning (e.g., setting a maximum depth) or post-pruning (e.g., cost-complexity pruning) can be applied to simplify the tree and improve its generalization performance.

The prediction made by a Decision Tree Regression model is simply the value at the leaf node where a new data point falls after traversing the tree based on the values of its features.

The mathematics behind Decision Tree Regression can be summarized as follows:

Splitting criterion: At each node, the decision tree algorithm selects the best feature and threshold for splitting the data to minimize the prediction error. This can be mathematically represented as finding the feature X and threshold T that minimize the splitting criterion, such as MSE:
MSE = (1/n) * summ(y - $\hat{y}$)^2

where:

n: Number of data points
y: Observed target values
$\hat{y}$: Predicted target values after splitting
Leaf node prediction: Once the data is split until a leaf node is

reached, the predicted value at that leaf node is simply the mean (or median, or any other appropriate statistic) of the target values of the samples in that leaf node.
Here's an example of Decision Tree Regression:

Suppose we have a dataset of housing prices with features such as square footage, number of bedrooms, and location, and we want to predict the prices of new houses. We can use Decision Tree Regression to build a model that predicts the prices based on these features.

Splitting: The algorithm selects the best feature and threshold (e.g., square footage > 1500 sq. ft.) to split the data into two groups: houses with square footage greater than 1500 sq. ft. and houses with square footage less than or equal to 1500 sq. ft.

Leaf node prediction: For the houses with square footage greater than 1500 sq. ft., the predicted price might be the mean price of all the houses in that group, and for the houses with square footage less than or equal to 1500 sq. ft., the predicted price might be the mean price of all the houses in that group.

Pruning: If the tree is overly complex, we can apply pruning techniques to simplify it, such as setting a maximum depth or using cost-complexity pruning.

20. KNN Regression

K-Nearest Neighbors (KNN) Regression is a supervised machine learning algorithm used for regression tasks. It works by finding the K nearest neighbors of a given data point in the training dataset and using their target values to predict the target

value of the query data point.

Here's a detailed explanation of KNN Regression along with its mathematics:

Training Phase:
The training phase of KNN Regression involves storing the training dataset, which consists of input features (x) and their corresponding target values (y).
No explicit model is built during the training phase, as KNN is a non-parametric algorithm and does not make assumptions about the underlying data distribution.
Prediction Phase:
The prediction phase of KNN Regression involves finding the K nearest neighbors of a query data point from the training dataset.
The distance metric, such as Euclidean distance or Manhattan distance, is used to measure the similarity or dissimilarity between the query data point and each training data point.
The K nearest neighbors are selected based on the smallest distance values.
The average or weighted average of the target values of these K neighbors is used as the predicted target value for the query data point.
Mathematics:

Let's consider a query data point with input features denoted as X_q, and its target value to be predicted denoted as y_q.

The KNN Regression algorithm finds the K nearest neighbors of X_q from the training dataset, denoted as $X_{n1}, X_{n2}, \ldots, X_{nK}$, with their corresponding target values denoted as $y_{n1}, y_{n2}, \ldots, y_{nK}$.

The distance metric D(Xq, Xni) measures the distance between Xq and Xni, where i = 1, 2, ..., K.

The predicted target value yq_pred for the query data point Xq is calculated as the average or weighted average of the target values of the K nearest neighbors:

yq_pred = (1/K) * summ(yni) or yq_pred = (1/K) * summ(wni * yni)

where n represents the index of the nearest neighbors (i.e., n = 1, 2, ..., K), and wni represents the weight assigned to each neighbor based on the distance metric.

Example:

- Let's consider a dataset with two input features, X1 and X2, and a target variable y.
- The training dataset consists of several data points with their corresponding input features and target values.
- To make a prediction for a query data point Xq, the KNN Regression algorithm finds the K nearest neighbors of Xq from the training dataset based on the distance metric (e.g., Euclidean distance).
- The predicted target value yq_pred for Xq is calculated as the average or weighted average of the target values of the K nearest neighbors.

21. Time Series Forecasting

Time series forecasting is a technique used in machine learning and statistics to predict future values of a time-dependent

dataset based on its historical data. It involves analyzing and modeling the patterns and trends in the data to make accurate predictions.

Time series data typically consists of a sequence of observations or measurements recorded over time. Let's consider a simple example of monthly temperature data for a city over the course of several years. The goal is to predict the temperature for the next month based on the historical temperature values.

Mathematically, a time series can be represented as a sequence of observations or measurements denoted by $y = \{y1, y2, y3, ..., yn\}$, where y represents the observed values, and n represents the total number of observations. The time index can be denoted by $t = \{1, 2, 3, ..., n\}$.

To make predictions in time series forecasting, there are several common methods used, including:

Autoregression (AR): Autoregression is a method where the value of the variable at a given time step is predicted based on its own previous values. The AR model is denoted as AR(p), where p is the order of autoregression. Mathematically, the AR(p) model can be represented as:

$$yt = \beta0 + \beta1 * yt-1 + \beta2 * yt-2 + ... + \beta p * yt-p + \varepsilon t$$

where yt is the predicted value at time t, $\beta0$ to βp are the coefficients, yt-1 to yt-p are the previous values of the variable, and εt is the error term at time t.

Moving Average (MA): Moving Average is a method where the value

of the variable at a given time step is predicted based on the average of its previous values. The MA model is denoted as MA(q), where q is the order of moving average. Mathematically, the MA(q) model can be represented as:

$$yt = \mu + \varepsilon t + \theta 1 * \varepsilon t-1 + \theta 2 * \varepsilon t-2 + \ldots + \theta_q * \varepsilon t-q$$

where yt is the predicted value at time t, μ is the mean of the time series, εt is the error term at time t, and $\theta 1$ to θ_q are the coefficients of the moving average terms.

Autoregressive Integrated Moving Average (ARIMA): ARIMA is a combined method that combines autoregression, moving average, and differencing to handle time series data with trends and seasonality. The ARIMA model is denoted as ARIMA(p, d, q), where p is the order of autoregression, d is the degree of differencing, and q is the order of moving average. Mathematically, the ARIMA(p, d, q) model can be represented as:

$$yt = \mu + \beta 1 * \Delta yt-1 + \beta 2 * \Delta yt-2 + \ldots + \beta p * \Delta yt-p + \varepsilon t + \theta 1 * \varepsilon t-1 + \theta 2 * \varepsilon t-2 + \ldots + \theta_q * \varepsilon t-q$$

where yt is the predicted value at time t, Δyt is the differenced value at time t, μ is the mean of the time series, εt is the error term at time t, and $\theta 1$ to θ_q are the coefficients of the moving average terms, and $\beta 1$ to βp are the coefficients of the autoregressive terms.

Seasonal Autoregressive Integrated Moving-Average (SARIMA): SARIMA is an extension of ARIMA that includes additional parameters to handle seasonal patterns in time series data. The SARIMA model is denoted as SARIMA(p, d, q)(P, D, Q)m, where p

is the order of autoregression, d is the degree of differencing, q is the order of moving average, P is the seasonal order of autoregression, D is the seasonal degree of differencing, Q is the seasonal order of moving average, and m is the number of time steps in each season. The mathematical representation of SARIMA is similar to ARIMA, but with additional seasonal terms.

Seasonal Decomposition of Time Series (STL): STL is a method that decomposes a time series into its seasonal, trend, and residual components, and then models each component separately. The STL model can be represented as:

$$yt = St + Tt + Rt + Et$$

where yt is the observed value at time t, St is the seasonal component, Tt is the trend component, Rt is the residual component, and Et is the error term at time t.

Example:

Let's consider a time series dataset of monthly average temperature for a city over the past 5 years, with 60 observations. We want to predict the temperature for the next month using an ARIMA model. We can use the historical temperature data to estimate the coefficients for autoregressive, differencing, and moving average terms, and then use these coefficients to make predictions for the next month's temperature.

Mathematically, the ARIMA model for this example can be represented as:

$yt = \mu + \beta 1 * \Delta yt-1 + \beta 2 * \Delta yt-2 + ... + \beta p * \Delta yt-p + \varepsilon t + \theta 1 * \varepsilon t-1 + \theta 2 * \varepsilon t-2 + ... + \theta q * \varepsilon t-q$

where yt is the predicted value at time t, Δyt is the differenced value at time t, μ is the mean of the time series, εt is the error term at time t, and $\theta 1$ to θq are the coefficients of the moving average terms, and $\beta 1$ to βp are the coefficients of the autoregressive terms. These coefficients can be estimated using techniques like maximum likelihood estimation or least squares estimation.

Once the coefficients are estimated, we can use them to make predictions for the next month's temperature by plugging in the values of the differenced variables, error terms, and coefficients into the ARIMA model equation.

Note: The specific steps and equations for time series forecasting may vary depending on the method or model used, and the data characteristics. It's important to carefully analyze and understand the data, choose an appropriate method, and validate the accuracy of the forecasts.

22. Ensemble methods(Bagging, Stacking, Voting)

Ensemble methods are techniques in machine learning that combine the predictions of multiple models to improve overall performance. Here's a detailed explanation of three popular ensemble methods: Bagging, Stacking, and Voting, along with their mathematics:

Bagging (Bootstrap Aggregating):
Bagging is an ensemble technique that involves training multiple

instances of the same model on different subsets of the training data, obtained through random sampling with replacement (also known as bootstrapping). The predictions of the individual models are then combined using a voting or averaging approach.
Mathematics:

Let's assume we have a training dataset of size N with features X and target Y.
Bagging involves creating B (e.g., 10 or 100) subsets of the training data, each with N samples randomly drawn with replacement.
Each subset is used to train an identical model, such as a decision tree, on the respective subset of data.
The final prediction is obtained by averaging (for regression) or voting (for classification) the predictions of all the trained models.
Example:
Suppose we have a dataset with 1000 samples and we want to use bagging to train a decision tree with 10 subsets. The algorithm would create 10 subsets of 1000 samples each by random sampling with replacement. Then, a decision tree would be trained on each of the 10 subsets. Finally, the predictions of these 10 decision trees would be combined through averaging (for regression) or voting (for classification) to obtain the final prediction.

Stacking (Stacked Generalization):
Stacking is an ensemble technique that involves training multiple models, where the predictions of one or more models are used as inputs to another model (meta-model) to make the final prediction. It uses a two-level approach where the first level models make predictions on the original training data, and the predictions are used as inputs for the second level meta-model.

Mathematics:

Let's assume we have a training dataset of size N with features X and target Y.
Stacking involves training multiple models, such as decision trees, support vector machines, or neural networks, on the original training data X to obtain their predictions.
The predictions of the first level models are then combined into a new feature matrix, which is used as input to train a meta-model, such as logistic regression or random forest, to make the final prediction.
The meta-model is trained on the predictions of the first level models, and the final prediction is obtained by applying the trained meta-model on the test data.
Example:
Suppose we have a dataset with 1000 samples and we want to use stacking to train a meta-model for classification. We can train multiple models, such as a decision tree, support vector machine, and neural network, on the original training data. Then, we can combine the predictions of these models into a new feature matrix. Finally, we can train a logistic regression model on this new feature matrix to obtain the final prediction.

Voting:
Voting is an ensemble technique that involves combining the predictions of multiple models by taking a majority vote (for classification) or averaging (for regression) to make the final prediction. It can be used with multiple models of the same type or different types.
Mathematics:

Let's assume we have a training dataset of size N with

features X and target Y.
Voting involves training multiple models, such as decision trees, support vector machines, or logistic regression, on the training data X to obtain their predictions.
The predictions of the individual models are then combined using a majority vote (for classification) or averaging (for regression) to obtain the final prediction.
Example:
Suppose we have a dataset with 1000 samples and we want to use voting to make a classification prediction. We can train three different models, such as a decision tree, support vector machine, and logistic regression, on the training data. Then, for a given test sample, each of these models would make its prediction. The final prediction would be determined by taking a majority vote among these individual predictions. For example, if two models predict class A and one model predicts class B, the final prediction would be class A, as it has the majority vote.

Ensemble methods, including Bagging, Stacking, and Voting, can significantly improve the predictive performance of machine learning models by leveraging the strengths of multiple models. They are widely used in various machine learning tasks, including classification, regression, and anomaly detection, among others.

Please note that the mathematics behind ensemble methods may vary depending on the specific implementation and type of models used.

23. Multi task learning

Multi-task learning is a machine learning paradigm where a model

is trained to perform multiple related tasks simultaneously, using shared information across tasks to improve the overall performance. Let's understand the mathematics behind it in a simplified manner:

In multi-task learning, we have T tasks, denoted by T1, T2, ..., TT, and a dataset with N samples denoted by (X1, Y1), (X2, Y2), ..., (XN, YN), where Xi represents the input features and Yi represents the corresponding target labels for the i-th sample.

The objective of multi-task learning is to learn a shared model with shared parameters θ, which can be used to make predictions for all tasks simultaneously. The objective function for multi-task learning can be formulated as:

$$\text{minimize}_{\theta} \{ (1/N) * \sum_{i=1 \text{ to } N} \{ \sum_{t=1 \text{ to } T} \{ \mathcal{L}t(Yit, f(Xi; \theta)) \} \} + \alpha * R(\theta) \}$$

where:

N: Number of samples in the dataset
T: Number of tasks
Xi: Input features for the i-th sample
Yit: True label for the t-th task of the i-th sample
$f(Xi; \theta)$: Prediction for the i-th sample using shared parameters θ
$\mathcal{L}t(Yit, f(Xi; \theta))$: Loss function for the t-th task, which measures the prediction error for the t-th task
α: Hyperparameter controlling the trade-off between task-specific learning and shared learning
$R(\theta)$: Regularization term applied to the shared parameters θ to prevent overfitting
The objective function consists of two terms: the task-specific

loss term, which measures the prediction error for each task, and the regularization term, which encourages the shared parameters θ to be regularized to prevent overfitting. The hyperparameter α controls the trade-off between task-specific learning and shared learning. A larger value of α emphasizes more on shared learning, while a smaller value of α emphasizes more on task-specific learning.

Multi-task learning can benefit from shared information across tasks, where the model can leverage common patterns or knowledge learned from one task to improve the performance of other related tasks, even when the tasks have different data distributions or sample sizes.

For example, in computer vision, a multi-task learning approach can simultaneously learn to perform tasks such as image classification, object detection, and image segmentation, using shared features learned from a large dataset of images. By jointly learning these tasks, the model can benefit from the shared knowledge and improve the overall performance compared to training separate models for each task individually.

24. Multinomial Naive Bayes

Multinomial Naive Bayes is a probabilistic classification algorithm that is used for classifying data points into multiple classes, typically in text classification tasks. It is based on the Naive Bayes algorithm, but it assumes that the features follow a multinomial distribution.

The mathematics behind Multinomial Naive Bayes can be

summarized as follows:

Data Representation:
Let's assume we have a dataset with N data points and K classes. Each data point is represented by a feature vector X of size D, where D is the number of features. The feature vector X contains counts or frequencies of occurrences of features in the data point. For example, in text classification, the feature vector X can represent the frequency of occurrence of each word in a document.

Parameter Estimation:
The first step in training a Multinomial Naive Bayes classifier is to estimate the parameters, i.e., the class priors P(Ck) and the conditional probabilities P(Xi|Ck) for each feature Xi and each class Ck. The class priors P(Ck) are estimated as the relative frequencies of each class in the training data. The conditional probabilities P(Xi|Ck) are estimated as the relative frequencies of occurrence of each feature Xi in the training data of each class Ck.

Classification:
Given a new data point with a feature vector X, the Multinomial Naive Bayes classifier computes the conditional probability P(Ck|X) for each class Ck using Bayes' theorem:

$$P(Ck|X) = P(X|Ck) * P(Ck) / P(X)$$

where:

P(Ck|X): Posterior probability of class Ck given the feature vector X

P(X|Ck): Likelihood of the feature vector X given the class Ck, estimated using the conditional probabilities P(Xi|Ck)
P(Ck): Prior probability of class Ck, estimated using the class priors P(Ck)
P(X): Evidence probability of the feature vector X, which is a constant for all classes and can be ignored for classification
The Multinomial Naive Bayes classifier then assigns the data point to the class with the highest posterior probability P(Ck|X).

Laplace Smoothing:
In practice, to handle cases where a feature does not occur in the training data of a particular class, Laplace smoothing is often applied to avoid zero probabilities. Laplace smoothing adds a small constant α to the counts or frequencies of occurrences of features during parameter estimation, effectively smoothing the probability estimates. The formula for estimating the conditional probabilities P(Xi|Ck) with Laplace smoothing is:

P(Xi|Ck) = (count of Xi in training data of Ck + α) / (total count of all features in training data of Ck + α * D)

where α is the smoothing parameter and D is the total number of features.

Example:
Let's consider an example of text classification, where we have a dataset of movie reviews labeled as positive, negative, or neutral. The features are the frequencies of occurrence of each word in the reviews, and the goal is to classify new movie reviews into one of the three classes.

In this case, the feature vector X for each review will represent

the frequencies of occurrence of each word in that review. The Multinomial Naive Bayes classifier will estimate the class priors `P(Ck)by calculating the relative frequencies of each class in the training data. It will also estimate the conditional probabilitiesP(Xi|Ck)by calculating the relative frequencies of occurrence of each word in the training data of each classCk`, using Laplace smoothing to avoid zero probabilities.

Once the parameters are estimated, the Multinomial Naive Bayes classifier can be used to classify new movie reviews. For a given review with a feature vector X representing the frequencies of occurrence of each word, the classifier will calculate the posterior probability P(Ck|X) for each class Ck using Bayes' theorem and assign the review to the class with the highest posterior probability.

Example:
Let's consider a movie review dataset with three classes - positive, negative, and neutral. Each movie review is represented by a feature vector X that contains the frequencies of occurrence of each word in the review. The goal is to classify new movie reviews into one of the three classes.

Parameter Estimation:
Class priors: The class priors P(Ck) are estimated as the relative frequencies of each class in the training data. For example, if we have 1000 training reviews, with 300 positive, 400 negative, and 300 neutral reviews, then the class priors would be:

P(positive) = 300/1000 = 0.3
P(negative) = 400/1000 = 0.4
P(neutral) = 300/1000 = 0.3

Conditional Probabilities: The conditional probabilities $P(x_i|C_k)$ are estimated as the relative frequencies of occurrence of each word in the training data of each class C_k, using Laplace smoothing. For example, let's say we have 1000 positive reviews, and the word "good" occurs 200 times in the positive reviews. If we use Laplace smoothing with a smoothing parameter $\alpha = 1$ and assuming a total of 5000 unique words in the training data, then the conditional probability P("good"|positive) would be:

$P(\text{"good"}|\text{positive}) = (200 + 1) / (1000 + 1 * 5000) = 201/6001 \approx 0.0335$

Similarly, the conditional probabilities for other words and classes can be estimated.

Classification:
Once the parameters are estimated, we can use the Multinomial Naive Bayes classifier to classify new movie reviews. For a given review with a feature vector X representing the frequencies of occurrence of each word, the classifier will calculate the posterior probability $P(C_k|X)$ for each class C_k using Bayes' theorem. For example, for a new review with the feature vector $X = [10, 5, 3, 0, 15, ...]$ representing the frequencies of occurrence of each word, the classifier will calculate the posterior probability for each class C_k and assign the review to the class with the highest posterior probability.