ASSIGNMENT - 15(KNN ~ K Nearest Neighbors)

Solution/Ans by - Pranav Rode(29)

1. What is the difference between Supervised And Unsupervised Machine Learning?

Supervised Learning	Unsupervised Learning
Supervised learning algorithms are trained using labeled data.	Unsupervised learning algorithms are trained using unlabeled data.
Supervised learning model takes direct feedback to check if it is predicting correct output or not.	Unsupervised learning model does not take any feedback.
Supervised learning model predicts the output.	Unsupervised learning model finds the hidden patterns in data.
In supervised learning, input data is provided to the model along with the output.	In unsupervised learning, only input data is provided to the model.
The goal of supervised learning is to train the model so that it can predict the output when it is given new data.	The goal of unsupervised learning is to find the hidden patterns and useful insights from the unknown dataset.
Supervised learning needs supervision to train the model.	Unsupervised learning does not need any supervision to train the model.
Supervised learning can be categorized in **Classification** and **Regression** problems.	Unsupervised Learning can be classified in **Clustering** and **Associations** problems.
Supervised learning can be used for those cases where we know the input as well as corresponding outputs.	Unsupervised learning can be used for those cases where we have only input data and no corresponding output data.
Supervised learning model produces an accurate result.	Unsupervised learning model may give less accurate result as compared to supervised learning.
Supervised learning is not close to true Artificial intelligence as in this, we first train the model for each data, and then only it can predict the correct output.	Unsupervised learning is more close to the true Artificial Intelligence as it learns similarly as a child learns daily routine things by his experiences.
It includes various algorithms such as Linear Regression, Logistic Regression, Support Vector Machine, Multi-class Classification, Decision tree, Bayesian Logic, etc.	It includes various algorithms such as Clustering, KNN, and Apriori algorithm.

2. What are the Parametric and Nonparametric Machine Learning Algorithms

Aspect	Parametric Algorithms	Nonparametric Algorithms
Underlying Assumption	Make assumptions about the form of the data distribution.	Make minimal assumptions about the form of the data distribution.
Number of Parameters	Fixed number of parameters regardless of dataset size.	Number of parameters can grow with the size of the dataset.
Model Structure	Predetermined structure based on assumptions.	Adaptive structure, adjusting to the complexity of the data.
Examples	Linear Regression, Logistic Regression.	K-Nearest Neighbors (KNN), Decision Trees.
Flexibility	Limited flexibility to capture complex patterns.	More flexibility, capable of capturing complex and varied patterns.
Speed on Large Datasets	Faster on large datasets due to a fixed number of parameters.	Potentially slower on large datasets due to a growing number of parameters.
Handling Complex Patterns	May struggle to capture highly complex patterns.	Well-suited for capturing complex and nonlinear relationships.
Suitability	Well-suited when underlying assumptions are met.	More versatile and suitable for various data distributions.
Examples of Algorithms	Linear Regression, Logistic Regression.	K-Nearest Neighbors (KNN), Decision Trees.
Risk of Overfitting	Lower risk of overfitting due to simplicity.	Higher risk of overfitting, especially with small datasets.

3. Explain the K Nearest Neighbor Classification in detail

Answer 1:

Overview: K Nearest Neighbor is a non-parametric and lazy learning algorithm used for both classification and regression. The focus here is on its application in classification, where the goal is to predict the class labels of data points based on their proximity to other labeled instances.

Working Principle:

1. **Data Points:** In a labeled dataset, each data point has features (attributes) and a corresponding class label.
2. **Distance Calculation:** The algorithm calculates the distance between the input data point and all other data points in the dataset. Commonly used distance metrics include Euclidean distance, Manhattan distance, or Minkowski distance.
3. **Choosing K:** K represents the number of nearest neighbors to consider. The algorithm looks for the K data points with the smallest distances to the input point.
4. **Voting Mechanism:** For classification, the algorithm counts the occurrences of each class among the K-nearest neighbors. The class with the majority votes becomes the predicted class for the input data point.
5. **Weighted Voting:** In some scenarios, the algorithm may use weighted voting, giving more influence to closer neighbors in the decision-making process.

Key Considerations:

- **Data Scaling:** It's crucial to scale features to ensure that no single feature dominates the distance calculations.
- **Curse of Dimensionality:** High-dimensional data can impact the algorithm's performance. Feature selection or dimensionality reduction techniques may be employed.
- **Optimal K Selection:** The choice of K is critical. A small K can be sensitive to noise, while a large K might oversimplify the model. Cross-validation or grid search can help find the optimal K.

Pros:

- **Simplicity:** KNN is intuitive and easy to understand, making it accessible for beginners.
- **Adaptability:** It can handle non-linear decision boundaries.

Cons:

- **Computational Cost:** The algorithm can be computationally expensive, especially for large datasets.
- **Sensitivity to Outliers:** Outliers can significantly influence the results.

Use Cases:

- **Small to Medium-sized Datasets:** KNN is effective when dealing with datasets of manageable size.
- **Variable Decision Boundaries:** It excels in scenarios where decision boundaries are irregular and hard to define analytically.

In Summary: K Nearest Neighbor Classification is a flexible and intuitive algorithm, suitable for scenarios where understanding the local relationships between data points is crucial. While it may have computational costs, its simplicity and adaptability make it a valuable tool in the data scientist's toolkit.

Answer 2:

K-Nearest Neighbor (KNN) classification is a simple yet powerful machine learning algorithm used for both classification and regression tasks. It is a non-parametric, lazy learning algorithm, meaning it doesn't make any assumptions about the underlying data distribution and doesn't learn any explicit model.
Instead, it relies on thestored data to make predictions.

Here's how KNN classification works:

1. Training Phase:

The training dataset consists of data points with known labels.
Each data point is represented by a set of features or attributes.

2. Prediction Phase:

To classify a new data point:

- Calculate the distance between the new data point and each data point in the training set.
- Commonly used distance metrics are Euclidean distance and Manhattan distance.
- Select the k nearest neighbors based on the calculated distances.
- Determine the most frequent class among the k nearest neighbors.
- Assign the new data point to the class with the highest frequency.

Choosing the value of k:

The value of k is a crucial parameter in KNN and significantly impacts its performance.
A small value of k can lead to overfitting, where the model memorizes the training data and fails to generalize to unseen data.
A large value of k can lead to underfitting, where the model fails to capture the underlying patterns in the data.
Choosing the optimal value of k often involves trial and error or cross-validation techniques.

Advantages of KNN:

- Simple to understand and implement.
- No explicit model training required, making it less prone to overfitting.
- Efficient for small datasets.
- Can handle multi-class classification problems.

Disadvantages of KNN:

- Computationally expensive for large datasets due to distance calculations.
- Sensitive to irrelevant features and outliers.
- Choice of k can significantly impact performance.

Variations of KNN:

- Weighted KNN: Assigns weights to neighbors based on their distance to the new data point.
- KNN with distance metrics other than Euclidean distance: Can be used for non-Euclidean data.
- Adaptive KNN: Automatically adjusts the value of k for each data point.

Applications of KNN:

- Image recognition
- Handwritten digit recognition
- Spam filtering
- Customer segmentation
- Medical diagnosis
- Additional Resources:

- KNN is a versatile algorithm with a wide range of applications.

- Understanding the strengths and weaknesses of KNN is crucial for using it effectively.
- Experimenting with different values of k and variations of KNN can help improve its performance.

4. Explain the K Nearest Neighbor Regression in detail

Overview: K Nearest Neighbor Regression is a supervised machine learning algorithm that predicts continuous values based on the average or weighted average of the K-nearest neighbors to a given data point.

Working Principle:

1. **Data Points:** Similar to KNN Classification, the algorithm operates on a dataset with labeled instances. Each instance has features (attributes) and a corresponding numeric value.

2. **Distance Calculation:** The algorithm calculates the distance between the input data point and all other data points in the dataset. Common distance metrics include Euclidean distance, Manhattan distance, or Minkowski distance.

3. **Choosing K:** K represents the number of nearest neighbors to consider. The algorithm identifies the K data points with the smallest distances to the input point.

4. **Averaging Mechanism:** For regression, the predicted value for the input data point is the average (or weighted average) of the target values of its K-nearest neighbors.

5. **Weighted Averaging:** Some implementations allow giving more weight to certain neighbors, emphasizing their influence on the final prediction.

Key Considerations:

- **Data Scaling:** As with KNN Classification, it's crucial to scale features to avoid biases based on feature scales.

- **Curse of Dimensionality:** High-dimensional data can affect the performance of KNN Regression. Feature selection or dimensionality reduction techniques can be applied.

- **Optimal K Selection:** The choice of K is essential. A smaller K can make predictions sensitive to noise, while a larger K may oversmooth the model. Cross-validation or grid search can assist in finding the optimal K.

Pros:

- **Simplicity:** KNN Regression is easy to understand and implement, making it accessible for various applications.
- **Non-Linearity:** It can capture non-linear relationships between features and target values.

Cons:

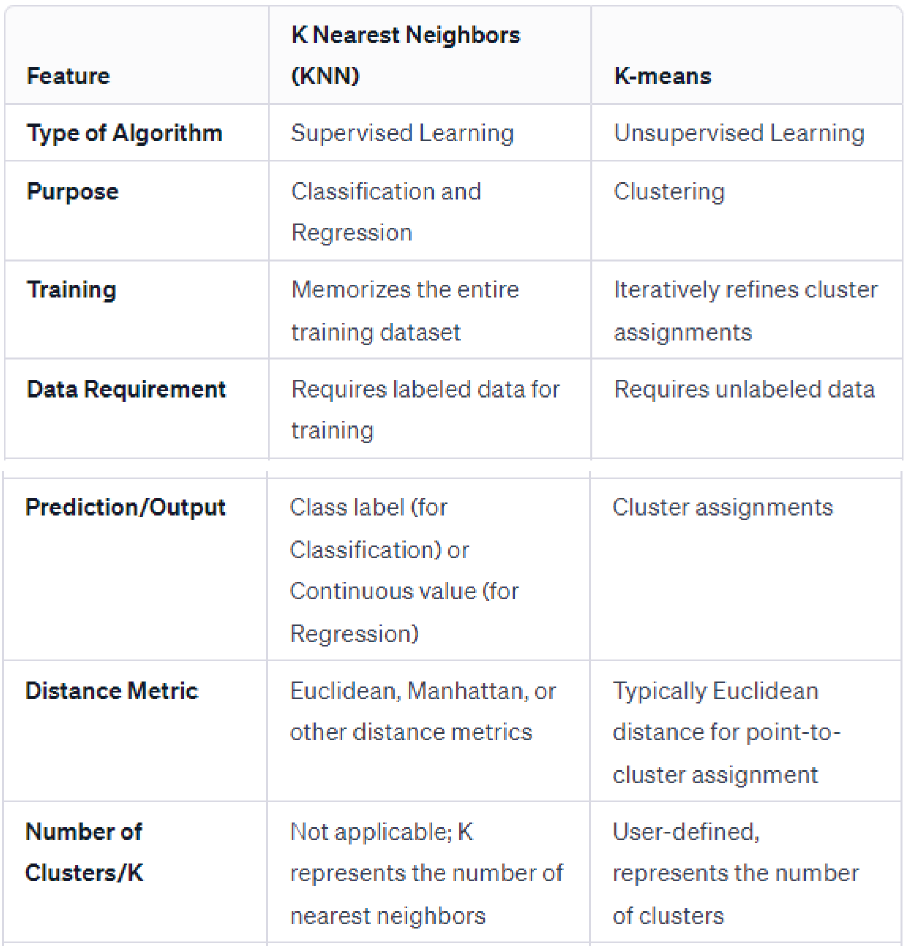
- **Computational Cost:** The algorithm can be computationally expensive, particularly for large datasets.
- **Sensitivity to Outliers:** Outliers can significantly impact the predictions.

Use Cases:

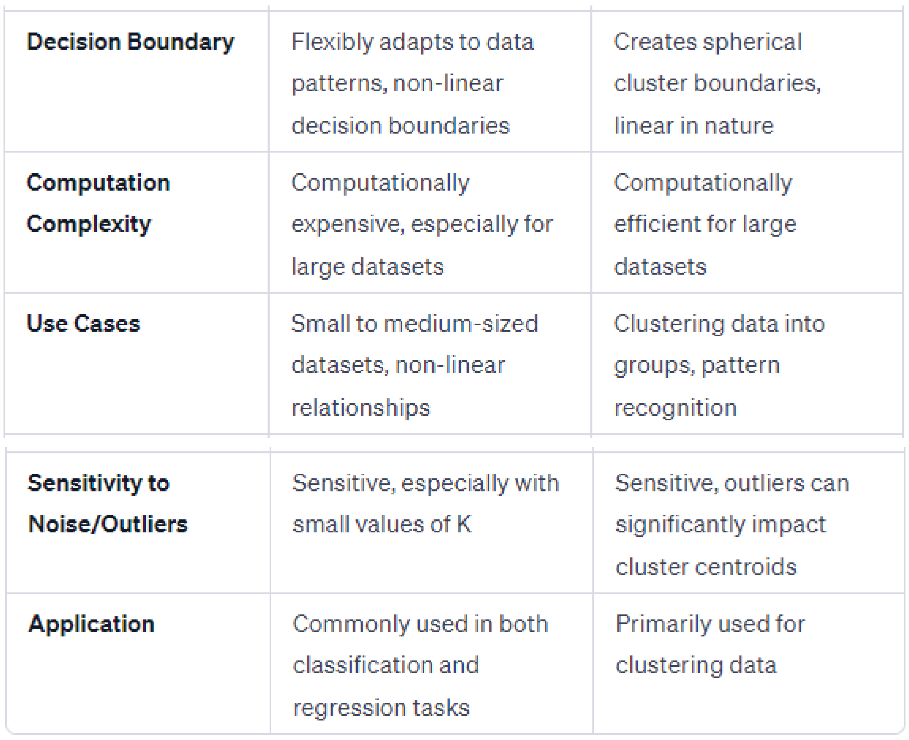
- **Non-linear Relationships:** KNN Regression is effective when the relationship between features and the target is complex and non-linear.
- **Small to Medium-sized Datasets:** It's suitable for datasets where the computational cost is manageable.

In Summary: K Nearest Neighbor Regression is a flexible algorithm for predicting continuous values. While it shares similarities with KNN Classification, its application in regression scenarios makes it valuable when dealing with non-linear relationships between variables.

5. What is the difference between KNN and K-means?



Feature	K Nearest Neighbors (KNN)	K-means
Type of Algorithm	Supervised Learning	Unsupervised Learning
Purpose	Classification and Regression	Clustering
Training	Memorizes the entire training dataset	Iteratively refines cluster assignments
Data Requirement	Requires labeled data for training	Requires unlabeled data
Prediction/Output	Class label (for Classification) or Continuous value (for Regression)	Cluster assignments
Distance Metric	Euclidean, Manhattan, or other distance metrics	Typically Euclidean distance for point-to-cluster assignment
Number of Clusters/K	Not applicable; K represents the number of nearest neighbors	User-defined, represents the number of clusters



Decision Boundary	Flexibly adapts to data patterns, non-linear decision boundaries	Creates spherical cluster boundaries, linear in nature
Computation Complexity	Computationally expensive, especially for large datasets	Computationally efficient for large datasets
Use Cases	Small to medium-sized datasets, non-linear relationships	Clustering data into groups, pattern recognition
Sensitivity to Noise/Outliers	Sensitive, especially with small values of K	Sensitive, outliers can significantly impact cluster centroids
Application	Commonly used in both classification and regression tasks	Primarily used for clustering data

Key differences between k-Nearest Neighbors (KNN) and k-Means clustering:

Feature	k-Nearest Neighbors (KNN)	k-Means Clustering
Type	Supervised Learning	Unsupervised Learning
Objective	Classification or Regression	Clustering
Input	Labeled training data (features and labels)	Unlabeled data points (features only)
Training	Memorizes training data	Clusters data points based on centroids
Prediction	Assigns a label or value to new data points	Assigns data points to clusters
Distance Measure	Typically uses Euclidean distance	Typically uses Euclidean distance or others
Parameters	Number of neighbors (k)	Number of clusters (k)
Algorithm Complexity	Higher complexity, especially with large datasets	Lower complexity, efficient for large datasets
Decision Boundary	Complex, can adapt to intricate patterns	Simple, linear boundaries between clusters
Scalability	Can be computationally expensive with large datasets	More scalable, efficient for large datasets
Use Cases	Classification and Regression tasks	Clustering, Anomaly Detection, Image Compression, etc.

It's important to note that KNN and k-Means serve different purposes and are used in distinct types of machine learning problems. KNN is mainly used for supervised learning tasks, while k-Means is a clustering algorithm used for unsupervised learning.

6. What is the “K” in KNN algorithm?

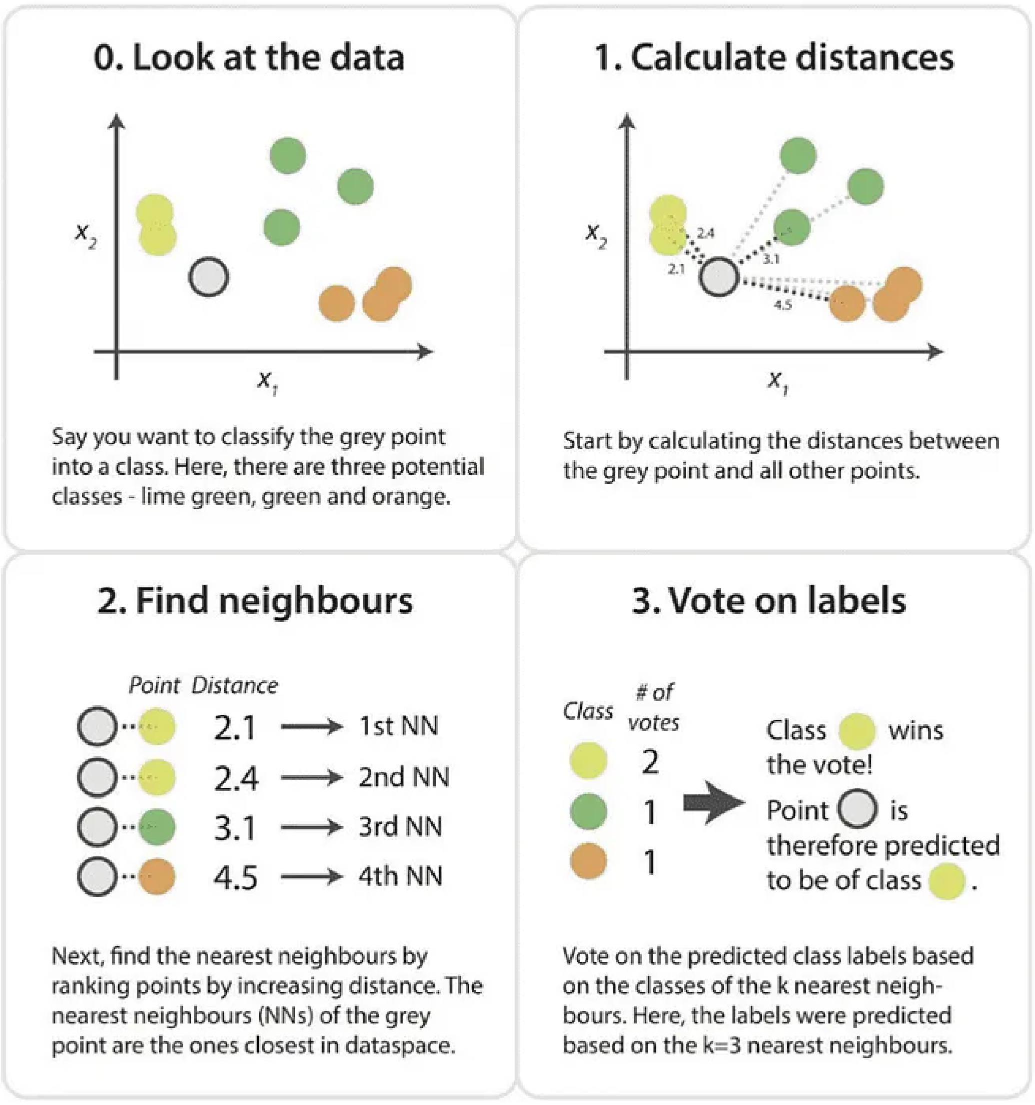
The "K" in K Nearest Neighbors (KNN) algorithm represents the number of nearest neighbors considered when making a prediction or classification for a new data point. In other words, K is a parameter that defines the size of the neighborhood used to determine the output.

Here's how it works:

1. **Distance Calculation:** KNN calculates the distance between the new data point and all other data points in the training dataset using a chosen distance metric (commonly Euclidean distance).
2. **Nearest Neighbors:** It identifies the K data points with the smallest distances to the new point.
3. **Voting (Classification) or Averaging (Regression):** For classification, the algorithm assigns the class label that is most common among the K-nearest neighbors.
   For regression, it predicts the average (or weighted average) of the target values of these neighbors.

The choice of K is crucial and can significantly impact the performance of the KNN algorithm.
A smaller K makes the algorithm more sensitive to noise, while a larger K may oversimplify the model.
Selecting the optimal K often involves experimentation, cross-validation, or grid search.

In summary, the "K" in KNN determines the size of the neighborhood used to make predictions, striking a balance between bias and variance in the model.


0. Look at the data
x_2
x_1
Say you want to classify the grey point into a class. Here, there are three potential classes - lime green, green and orange.
1. Calculate distances
2.4
2.1
3.1
4.5
Start by calculating the distances between the grey point and all other points.
2. Find neighbours
Point Distance
2.1 → 1st NN
2.4 → 2nd NN
3.1 → 3rd NN
4.5 → 4th NN
Next, find the nearest neighbours by ranking points by increasing distance. The nearest neighbours (NNs) of the grey point are the ones closest in dataspace.
3. Vote on labels
Class
# of votes
2
1
1
Class wins the vote!
Point is therefore predicted to be of class .
Vote on the predicted class labels based on the classes of the k nearest neighbours. Here, the labels were predicted based on the k=3 nearest neighbours.

7. How do we decide the value of "K" in KNN algorithm?

Choosing the optimal value of k in the KNN algorithm is crucial for its performance.
An inappropriate choice can lead to overfitting or underfitting, negatively impacting prediction accuracy.
Here are some methods for deciding the value of k:

1. Grid search:

- This method involves systematically testing a range of k values and evaluating the performance of the KNN model on each value. The k value with the best performance is chosen as the optimal value.
- Grid search can be computationally expensive, but it guarantees finding the best k within the specified range.

2. Elbow method:

- This method involves plotting the error rate (e.g., mean squared error) of the KNN model for different k values. The "elbow" of the plot, where the error rate starts to stabilize, indicates the optimal k value.
- The elbow method is a simple and efficient way to find a good k value, but it may not always be accurate.

3. Cross-validation:

- This method involves dividing the training data into smaller subsets called folds. The KNN model is trained on each fold excluding one, and its performance is evaluated on the excluded fold.

This process is repeated for each fold, and the average performance across all folds is used to compare different k values.
- Cross-validation is a more robust way to choose k than grid search or the elbow method because it estimates the generalization error of the model on unseen data.

4. Heuristic methods:

- Several heuristic methods exist for choosing k, such as the rule of thumb k = sqrt(n), where n is the number of data points in the training set.
- Heuristic methods are fast and easy to apply, but they may not be accurate for all datasets.

5. Adaptive k-nearest neighbors:

- This variation of KNN automatically adjusts the value of k for each data point based on the local data density.
- Adaptive KNN can be more accurate than traditional KNN, but it is computationally more expensive.

Additional factors to consider:

- **Dataset size:** Larger datasets may require larger values of k to capture the underlying patterns.
- **Data dimensionality:** Datasets with high dimensionality may benefit from smaller values of k to avoid the curse of dimensionality.
- **Noise level:** Noisy data may require larger values of k to be more robust to outliers.

Choosing the optimal value of k ultimately depends on the specific dataset and problem.
Experimenting with different methods and considering the factors mentioned above will help you find the best k value for your KNN model.

8. Why is the odd value of “K” preferable in KNN algorithm?

Using an odd value of "K" in the K Nearest Neighbors (KNN) algorithm is preferable, especially in binary classification problems, to avoid ties when voting for the majority class. Let's explore why this is the case:

Voting Mechanism in KNN: In KNN, when determining the class of a new data point, the algorithm looks at the class labels of its
K-nearest neighbors and assigns the class label that occurs most frequently. This process is known as voting.

Avoiding Ties: When the value of K is even, there is a possibility of a tie in the voting process. For example, if K = 4 and two neighbors belong to one class and the other two belong to another class, there is no clear majority.
In such cases, the algorithm may have difficulty making a definitive decision.

Odd K for Binary Classification: Using an odd value for K ensures that ties are less likely to occur. For binary classification (where there are only two classes), an odd K guarantees that there will be a clear majority when voting.
This can help in situations where having a clear
majority is desirable for making a confident classification.

Balancing Bias and Variance: Choosing an odd K is also seen as a way to strike a balance between bias and variance.
A smaller K (odd) tends to capture
more local information, potentially leading to a more flexible model. However, it can also make the model sensitive to noise. In contrast, a larger K (even) may oversimplify the model, potentially missing local patterns.

It's important to note that the preference for odd K is a guideline rather than a strict rule. The choice of K should ultimately depend on the characteristics of your data, the problem at hand, and the results of model evaluation through techniques like cross-validation.

In summary, while using an odd K is recommended for binary classification in KNN to avoid ties, the optimal choice of K should
be determined through experimentation and evaluation based on the specific characteristics of your dataset.

9. What distance metrics can be used in KNN?

1. **Euclidean Distance:**
   - **Formula:** $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

- **Importance:** Common and versatile, suitable for scenarios where the actual spatial distance matters. Applicable when features are continuous and have similar scales.
- **Use When:** Features have a clear notion of distance, and the dataset is not too high-dimensional.

2. **Manhattan Distance (L1 Norm):**

   - **Formula:** $|x_1 - x_2| + |y_1 - y_2|$
   - **Importance:** Useful when movement can only occur along grid lines (like city blocks). Less sensitive to outliers compared to Euclidean distance.
   - **Use When:** Movement along specific axes is more meaningful, and the dataset may have outliers.

3. **Minkowski Distance:**

   - **Formula:** $\left(|x_1 - x_2|^p + |y_1 - y_2|^p\right)^{\frac{1}{p}}$
   - **Importance:** Generalizes both Euclidean and Manhattan distances. Parameter (p) allows tuning between the two. Reduces to Euclidean distance for (p = 2).
   - **Use When:** You want to control the emphasis on different dimensions or when Euclidean and Manhattan distances are not sufficient.

4. **Chebyshev Distance (Infinity Norm):**

   - **Formula:** $\max(|x_1 - x_2|, |y_1 - y_2|)$
   - **Importance:** Measures the maximum absolute difference along any coordinate dimension. Robust to outliers.
   - **Use When:** Outliers are expected, and you want to focus on the maximum deviation.

5. **Hamming Distance:**

   - **Formula:** $\frac{\text{Number of differing symbols}}{\text{Total number of symbols}}$
   - **Importance:** Designed for categorical data, measures the proportion of differing symbols.
   - **Use When:** Dealing with categorical features, like text classification or DNA sequences.

6. **Cosine Similarity:**

   - **Formula:** $\frac{x_1 \cdot x_2 + y_1 \cdot y_2}{\sqrt{x_1^2 + y_1^2} \cdot \sqrt{x_2^2 + y_2^2}}$
   - **Importance:** Measures the cosine of the angle between vectors, useful for understanding similarity in direction.
   - **Use When:** Data is represented as vectors, commonly used in text mining or recommendation systems.

7. **Jaccard Similarity Coefficient:**

   - **Formula:** $\frac{\text{Intersection of sets}}{\text{Union of sets}}$
   - **Importance:** Measures the similarity between sets, commonly used for set-based data.
   - **Use When:** Dealing with sets of items, like document similarity or customer preferences.

In summary, the choice of distance metric in KNN depends on the nature of your data and the specific requirements of your problem. Experimentation and understanding the characteristics of each metric are key to selecting the most suitable one for your application.

10. What is the difference between Euclidean Distance and Manhattan

distance? What is the formula for Euclidean distance and Manhattan

distance?

Euclidean Distance vs. Manhattan Distance:

1. **Euclidean Distance:** **Formula:** $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

   - Measures the straight-line or "as-the-crow-flies" distance between two points in Euclidean space.
   - Emphasizes the magnitude of differences and is sensitive to large deviations along any dimension.
   - Suitable for scenarios where the actual spatial distance between points is important.

2. **Manhattan Distance (L1 Norm):** **Formula:** $|x_1 - x_2| + |y_1 - y_2|$

- Measures the sum of the absolute differences along each dimension, resembling the distance a taxi would travel in a city grid.
- Emphasizes movement along grid lines and is less sensitive to outliers compared to Euclidean distance.
- Suitable for scenarios where movement along specific axes is more meaningful.

Comparison:

- **Geometry:** Euclidean distance represents the shortest path between two points (diagonal line), while Manhattan distance represents the sum of horizontal and vertical distances (L-shape path).
- **Sensitivity:** Euclidean distance is sensitive to deviations in any direction, while Manhattan distance is less sensitive and considers only movement along the axes.
- **Use Cases:** Euclidean is suitable when the actual spatial distance is significant, while Manhattan is useful when movement along specific paths is more meaningful.

In summary, the choice between Euclidean and Manhattan distance depends on the characteristics of the data and the problem at hand. If the features have a clear notion of distance and the dataset is not too high-dimensional, Euclidean distance might be appropriate. If movement along specific axes is more meaningful or outliers are a concern, Manhattan distance could be a better choice.

11. Why do you need to scale your data for the k-NN algorithm?

Scaling your data is important for the k-NN (K Nearest Neighbors) algorithm for several reasons:

1. **Distance Computation:**
    - k-NN relies on the computation of distances between data points to determine the nearest neighbors.
    - If the features have different scales, those with larger magnitudes may dominate the distance calculations, leading to biased results.
2. **Equal Weighting of Features:**
    - In k-NN, each feature contributes equally to the distance measurement. If one feature has a larger scale than others, it can disproportionately influence the results.
    - Scaling ensures that all features contribute equally to the distance calculation, preventing the model from being biased towards features with larger scales.
3. **Numerical Stability:**
    - Scaling improves numerical stability during the distance computation process. Large differences in scale can result in numerical instability and precision issues.
4. **Consistent Model Performance:**
    - Scaling helps in achieving consistent and reliable model performance across different datasets or variations of the same dataset.
    - Without scaling, the model may perform well on one version of the data but poorly on another due to differences in feature scales.
5. **Model Convergence:**
    - Some distance-based algorithms, including k-NN, converge faster when the features are on similar scales. This can be important in cases where computational efficiency is a consideration.
6. **Model Interpretability:**
    - Scaling facilitates better model interpretability. When features are on the same scale, it becomes easier to compare the importance of different features and understand their contributions to the model.

Scaling Techniques:

- **Min-Max Scaling:** Scales features to a specific range (e.g., 0 to 1).
- **Standardization (Z-score normalization):** Scales features to have a mean of 0 and a standard deviation of 1.
- **Robust Scaling:** Scales features based on their interquartile range, making it robust to outliers.

In summary, scaling is crucial for k-NN to ensure that all features contribute equally to the distance calculations, prevent biases based on feature scales, and achieve consistent and reliable model performance. The choice of scaling method depends on the characteristics of your data and the requirements of your specific problem.

12. What are the Gradient Descent Based, Tree-Based, and distance-based algorithms?

1. **Gradient Descent-Based Algorithms:**
   - **Definition:** These algorithms aim to minimize a cost function by iteratively moving towards the direction of steepest decrease in the function.
   - **Key Characteristic:** They are iterative optimization algorithms that adjust model parameters to minimize a specified loss or error.
   - **Examples:**
     - **Linear Regression:** Minimizes the mean squared error by adjusting coefficients.
     - **Logistic Regression:** Optimizes the log-likelihood function to model binary or multi-class classification.
     - **Neural Networks:** Training involves gradient descent to minimize the error function.
2. **Tree-Based Algorithms:**
   - **Definition:** These algorithms construct decision trees during training and use them for predictions.
   - **Key Characteristic:** They make decisions based on recursive binary splits of the feature space.
   - **Examples:**
     - **Decision Trees:** Split the data based on features to create a tree structure for decision-making.
     - **Random Forest:** Ensemble of decision trees, each trained on a random subset of the data.
     - **Gradient Boosting:** Builds trees sequentially, each correcting errors of the previous one.
3. **Distance-Based Algorithms:**
   - **Definition:** These algorithms make predictions based on the distances between data points.
   - **Key Characteristic:** They rely on measuring the similarity or dissimilarity between instances in the feature space.
   - **Examples:**
     - **k-Nearest Neighbors (k-NN):** Classifies or regresses based on the majority or average of `the k-nearest neighbors.`
     - **Hierarchical Clustering:** Divides the data into clusters based on distances between data points.
     - **DBSCAN (Density-Based Spatial Clustering of Applications with Noise):** Clusters data points based on density and proximity.

Comparison:

- **Use Cases:**
  - **Gradient Descent:** Widely used in regression and classification tasks, especially for large-scale problems like deep learning.
  - **Tree-Based:** Effective for both classification and regression, handles complex relationships, and provides interpretability.
  - **Distance-Based:** Suitable for clustering and classification tasks where the proximity of data points is essential.
- **Interpretability:**
  - **Gradient Descent:** Interpretability can be challenging, especially in complex models like deep neural networks.
  - **Tree-Based:** Offers interpretability with the ability to visualize decision trees.
  - **Distance-Based:** Interpretability depends on the specific algorithm; k-NN can be interpretable, while clustering algorithms might not provide clear decision rules.
- **Scalability:**
  - **Gradient Descent:** Scales well for large datasets, especially with stochastic gradient descent variants.
  - **Tree-Based:** Can handle large datasets but may become computationally expensive for very large datasets.
  - **Distance-Based:** Can be computationally expensive for large datasets, as it involves pairwise distance calculations.

In summary, each category of algorithms has its strengths and weaknesses, and the choice depends on the specific characteristics of the data and the problem at hand. Gradient Descent-Based algorithms are powerful for optimization tasks, Tree-Based algorithms offer interpretability and handle complex relationships, and Distance-Based algorithms excel in tasks where the proximity of data points is crucial.

13. What is Normalization?

Normalization is a preprocessing technique used in data analysis and machine learning to scale and transform the features of a dataset into a standardized range. The goal of normalization is to ensure that all features contribute equally to the analysis and prevent features with larger magnitudes from dominating the learning process.

In the context of machine learning and statistics, normalization typically refers to scaling numerical features to a standard range, often between 0 and 1. This process is also known as feature scaling. The normalization formula for a feature (X) is given by:

$$X_{\text{normalized}} = \frac{X - \min(X)}{\max(X) - \min(X)}$$

Here:

- X is the original value of a feature.
- $\min(X)$ is the minimum value of the feature.
- $\max(X)$ is the maximum value of the feature.

This formula ensures that the normalized values fall within the range [0, 1].
Other normalization techniques may involve scaling to a different range or using statistical measures such as the mean and standard deviation.

Key Points about Normalization:

1. **Equalizes Feature Contributions:** Normalization ensures that all features contribute equally to the analysis by bringing them to a similar scale.
2. **Prevents Dominance of Large Magnitudes:** It prevents features with larger magnitudes from dominating the learning process, especially in algorithms sensitive to the scale of features (e.g., distance-based algorithms).
3. **Improves Convergence:** In optimization algorithms like gradient descent, normalization can help improve convergence and speed up the training process.
4. **Applicable to Various Scales:** Normalization is particularly useful when features have different ranges or units of measurement.
5. **Common Techniques:** Besides Min-Max normalization, other techniques include Z-score normalization (standardization), where features are scaled to have a mean of 0 and a standard deviation of 1.

Normalization is a standard practice in data preprocessing and is often applied before feeding data into machine learning models to ensure that the models perform effectively across different features and datasets.

14. What is Standardization?

Standardization, also known as Z-score normalization, is a preprocessing technique used in data analysis and machine learning to transform the features of a dataset by scaling them to have a mean of 0 and a standard deviation of 1. The goal of standardization is to ensure that features have comparable scales, making them suitable for algorithms that are sensitive to the scale of input features.

The standardization formula for a feature (X) is given by:

$$X_{\text{standardized}} = \frac{X-\mu}{\sigma}$$

Here:

- X is the original value of the feature.
- μ represents the mean (average) of the feature.
- σ represents the standard deviation of the feature.

The result of standardization is a set of values with a mean of 0 and a standard deviation of 1.
This process centers the data around 0 and scales it by the standard deviation, making it suitable for algorithms that assume a standard normal distribution of data.

Key Points about Standardization:

1. **Centering and Scaling:** Standardization involves centering the data by subtracting the mean and scaling it by dividing by the standard deviation.
2. **Comparable Scales:** Standardization makes different features comparable by ensuring they have similar scales.
3. **Mean of 0 and Standard Deviation of 1:** The standardized values have a mean of 0 and a standard deviation of 1.
4. **Applicable to Normally Distributed Data:** Standardization assumes that the features follow a normal distribution.
5. **Improves Algorithm Performance:** Standardization is particularly beneficial for algorithms that are sensitive to the scale of input features, such as gradient descent-based optimization algorithms.
6. **Commonly Used in Conjunction with Other Techniques:** Standardization is often used in conjunction with other preprocessing techniques like normalization to ensure that the data is appropriately prepared for machine learning models.

In summary, standardization is a common technique used to transform features into a standardized scale, ensuring that they have comparable magnitudes. This aids in the effective application of various machine learning algorithms, especially those that rely on the assumption of features having a standard normal distribution.

15. When to use Normalization and Standardization?

The choice between normalization and standardization depends on the characteristics of your data, the requirements of the machine learning algorithm you are using, and the nature of your problem.
Here are guidelines for when to use normalization and standardization:

Use Normalization:

1. **When Feature Ranges Differ Significantly:**
   - Normalize features when they have different scales, and you want to ensure that each feature contributes equally to the analysis.
2. **For Algorithms Sensitive to Feature Magnitudes:**
   - Use normalization with algorithms that are sensitive to the scale of features, such as k-Nearest Neighbors (KNN) and Support Vector Machines (SVM).
3. **When Maintaining Original Scale is Not Crucial:**
   - If maintaining the original scale of features is not essential for interpretability, normalization is a good choice.
4. **In Image Processing:**
   - Normalization is commonly applied in image processing to scale pixel values to a standard range, often [0, 1].
5. **When Features are on Different Ranges:**
   - If your dataset includes features with vastly different numerical ranges, normalization can bring them to a common scale.

Use Standardization:

1. **For Algorithms Assuming Normal Distribution:**
   - Standardize features when algorithms, like Principal Component Analysis (PCA), assume that features are normally distributed and centered around 0 with a standard deviation of 1.
2. **When Features Follow a Gaussian Distribution:**
   - Standardization is effective when features follow a Gaussian distribution or when an algorithm assumes a standardized distribution.
3. **For Optimization Algorithms Requiring Centered Data:**
   - Some optimization algorithms, like gradient descent, benefit from centered data. Standardization centers the data around 0, which can improve convergence.
4. **When Feature Importance Interpretability is Desired:**
   - If maintaining the interpretability of feature importance (measured in standard deviations from the mean) is crucial, standardization is preferred.
5. **In Multivariate Analysis:**
   - Standardization is often used in multivariate analysis when features are combined, and the assumption of a standard normal distribution is desired.

Considerations for Both:

1. **Experimental Evaluation:**
   - Experiment with both normalization and standardization and evaluate their impact on model performance through cross-validation.
2. **Domain Knowledge:**
   - Understand the nature of your data. If you have prior knowledge about the distribution of features, it can guide your decision.
3. **Data Distribution:**
   - Consider whether your data distribution aligns more with the assumptions of normalization or standardization.
4. **Algorithm Sensitivity:**
   - Some algorithms are more sensitive to the scale of features than others. Understand the requirements of the specific algorithm you are using.

In practice, it's common to try both preprocessing techniques and assess their impact on the performance of your machine learning models. The decision often involves experimentation and consideration of the specific characteristics of your data and modeling goals.

Normalization is generally less sensitive to outliers compared to standardization.
Let's understand why:

Normalization:

- Normalization scales the features to a specific range, commonly [0, 1], based on the minimum and maximum values of the data.
- Outliers that have extremely high or low values may affect the scaling but don't necessarily dominate the entire range.
- The impact of outliers is limited because the scaling is based on the range of the data.

Standardization:

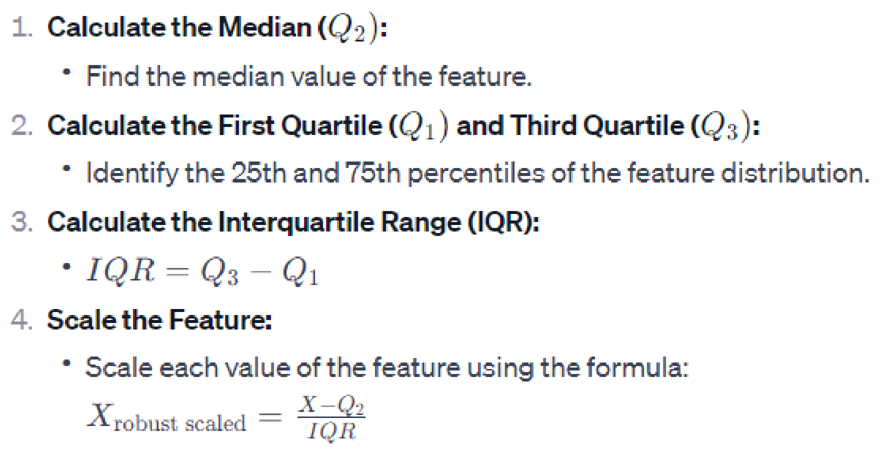
- Standardization involves subtracting the mean and dividing by the standard deviation.
- Outliers, especially those far from the mean, can significantly influence the standardization process. The mean and standard deviation are sensitive to extreme values.
- If a dataset has outliers, they can distort the calculation of the mean and standard deviation, affecting the standardization of the entire dataset.

In summary, standardization is more sensitive to outliers than normalization because it involves using the mean and standard deviation, which can be heavily influenced by extreme values. If your dataset contains outliers, and you want to minimize their impact on the scaling process, normalization may be a more suitable choice. However, in both cases, it's essential to assess the nature of your data and consider robust scaling techniques if outliers are a concern.

15.1 Explain robust scaling and when to use it ?

Robust scaling is a data preprocessing technique that aims to make features more resilient to the influence of outliers by using robust statistical measures. It is particularly useful when your dataset contains outliers that might significantly impact standard normalization or standardization methods.

The robust scaling process involves the use of the median and interquartile range (IQR) instead of the mean and standard deviation.
The key steps are as follows:



1. **Calculate the Median (Q_2):**
   - Find the median value of the feature.
2. **Calculate the First Quartile (Q_1) and Third Quartile (Q_3):**
   - Identify the 25th and 75th percentiles of the feature distribution.
3. **Calculate the Interquartile Range (IQR):**
   - $IQR = Q_3 - Q_1$
4. **Scale the Feature:**
   - Scale each value of the feature using the formula:
     $X_{\text{robust scaled}} = \frac{X - Q_2}{IQR}$

The resulting robustly scaled values center the data around the median and scale it based on the interquartile range. This makes the scaling process less sensitive to extreme values (outliers) compared to traditional mean and standard deviation-based scaling methods.

When to Use Robust Scaling:

1. **Presence of Outliers:**
   - Use robust scaling when your dataset contains outliers that may distort the mean and standard deviation.
2. **Sensitive Algorithms:**
   - Apply robust scaling when using algorithms sensitive to the scale of features, such as k-Nearest Neighbors (KNN) or clustering algorithms.
3. **Non-Normal Distributions:**
   - If the distribution of your features deviates significantly from a normal distribution, robust scaling can be a more suitable choice.
4. **Improved Resilience:**
   - When you want to improve the resilience of your features to the impact of extreme values, robust scaling can provide a more robust representation.
5. **Comparisons Across Features:**
   - If you need to compare features that have different scales and are influenced by outliers, robust scaling can offer a more consistent comparison.

In summary, robust scaling is a valuable preprocessing technique when dealing with datasets containing outliers. It provides a more robust way to scale features, making it less sensitive to extreme values and suitable for situations where traditional scaling methods might be influenced by outliers.

16. Why KNN Algorithm is called as Lazy Learner?

The k-Nearest Neighbors (KNN) algorithm is often referred to as a "lazy learner" because it doesn't learn a discriminative function from the training data during the training phase.

Instead, it defers the learning until the time of prediction or classification.
The primary characteristics that earn KNN its "lazy" label are:

1. **No Explicit Training Phase:**

    - KNN does not build an explicit model during the training phase. Unlike eager learners (or "eager learners"), such as decision trees or linear models, KNN does not analyze the entire training dataset to construct a generalized model.

2. **Instance-Based Learning:**

    - KNN is an instance-based learning algorithm. It memorizes the entire training dataset, storing the instances in memory rather than learning a compact representation of the data. The algorithm "learns" by remembering instances and their associated labels.

3. **Deferred Decision Boundary:**

    - KNN waits until a prediction is needed for a new, unseen instance. At that point, it looks at the k-nearest neighbors in the training dataset and assigns a label based on a majority vote or averaging (for classification or regression, respectively).

4. **Computation During Prediction:**

    - The computation in KNN happens primarily during the prediction phase, where distances between the query instance and all instances in the training set are calculated to identify the nearest neighbors.

5. **Sensitivity to New Data:**

    - KNN can adapt quickly to new data, as it doesn't require retraining the entire model. The decision boundaries can change dynamically with the addition of new instances or the removal of existing ones.

The term "lazy learner" is used in contrast to "eager learner," which includes algorithms that eagerly build a model during the training phase. While lazy learners like KNN can be computationally expensive during prediction, especially in high-dimensional spaces, they offer flexibility and adaptability to changes in the dataset without requiring a complete retraining process.

17. Why should we not use the KNN algorithm for large datasets?

While the k-Nearest Neighbors (KNN) algorithm is a simple and intuitive approach, it may not be suitable for large datasets due to several limitations:

1. **Computational Complexity:**

    - KNN involves calculating the distance between the query point and every point in the training dataset. In large datasets, this can become computationally expensive and result in slower prediction times.

2. **Memory Usage:**

    - KNN is memory-intensive as it requires storing the entire training dataset in memory. For large datasets, this can lead to high memory consumption and potentially cause performance issues.

3. **Search Time:**

    - Identifying the k-nearest neighbors involves searching through the entire dataset. As the dataset size increases, the search time grows linearly, making real-time or near-real-time predictions challenging.

4. **Curse of Dimensionality:**

    - In high-dimensional spaces, the "curse of dimensionality" becomes more pronounced. As the number of features increases, the distance between points tends to become more uniform, diminishing the effectiveness of distance-based similarity measures.

5. **Reduced Generalization:**

    - KNN may struggle to generalize well on large datasets, especially if the dataset is diverse and lacks clear patterns. The presence of noise or irrelevant features can lead to suboptimal predictions.

6. **Storage Requirements:**

    - Large datasets require substantial storage space, both in terms of the dataset itself and the additional data structures needed for efficient distance calculations, further contributing to resource-intensive processing.

7. **Parameter Tuning Challenges:**

- Choosing an appropriate value for (k) (the number of neighbors) becomes more challenging in large datasets. A small (k) may lead to noisy predictions, while a large (k) can result in over-smoothing and reduced sensitivity.

8. **Preprocessing Overhead:**

   - Preprocessing steps, such as feature scaling and dimensionality reduction, become crucial for large datasets. However, these steps can introduce additional computational overhead.

9. **Parallelization Difficulties:**

   - The inherent sequential nature of KNN makes parallelizing the algorithm challenging. Other machine learning algorithms, designed for parallel processing, may be more efficient for large-scale tasks.

In summary, while KNN is a straightforward and interpretable algorithm, its practicality diminishes for large datasets due to computational and memory constraints. For large-scale problems, alternative machine learning algorithms designed for efficiency and scalability, such as tree-based methods (Random Forest, Gradient Boosting) or linear models, may be more appropriate.

18. What are the advantages and disadvantages of the KNN algorithm?

The k-Nearest Neighbors (KNN) algorithm has its own set of advantages and disadvantages, making it suitable for certain types of problems while posing challenges in others.

Advantages of KNN:

1. **Simplicity and Intuitiveness:**

   - KNN is easy to understand and implement. It doesn't require assumptions about the underlying data distribution, making it a simple and intuitive algorithm.

2. **No Training Phase:**

   - KNN is a lazy learner, meaning it doesn't explicitly build a model during the training phase. This allows the algorithm to adapt quickly to new data without requiring retraining.

3. **Suitable for Small Datasets:**

   - KNN performs well on small to moderately sized datasets, where the computational and memory requirements are manageable.

4. **Adaptable to Different Data Types:**

   - KNN can be applied to both classification and regression problems, and it can handle datasets with mixed types of attributes (categorical and numerical).

5. **Effectiveness with Locally Homogeneous Data:**

   - KNN works well when the data is locally homogeneous, meaning instances with similar feature values tend to have the same class or target value.

6. **No Assumptions about Data Distribution:**

   - KNN makes no assumptions about the distribution of the data, making it versatile for various types of datasets.

Disadvantages of KNN:

1. **Computational Complexity:**

   - Calculating distances between the query point and all points in the training set can be computationally expensive, especially in high-dimensional spaces and with large datasets.

2. **Memory Intensive:**

   - KNN requires storing the entire training dataset in memory, leading to high memory consumption, particularly for large datasets.

3. **Sensitive to Noisy Data and Outliers:**

   - KNN is sensitive to noisy data and outliers, as they can significantly influence the identification of nearest neighbors and subsequent predictions.

4. **Curse of Dimensionality:**

- In high-dimensional spaces, the effectiveness of KNN can diminish due to the curse of dimensionality. The distance between points becomes less meaningful in high-dimensional feature spaces.

5. **Slow for Real-Time Predictions:**

   - As the size of the dataset increases, the time required to identify the k-nearest neighbors grows linearly, making real-time or near-real-time predictions challenging.

6. **Optimal (k) Selection:**

   - Choosing an appropriate value for (k) (the number of neighbors) can be challenging and may require experimentation. Small (k) values may lead to noisy predictions, while large (k) values can result in over-smoothing.

7. **Imbalanced Datasets:**

   - In datasets where classes are imbalanced, KNN can be biased toward the majority class. This can impact the prediction accuracy, especially for minority classes.

In conclusion, while KNN is a simple and versatile algorithm, its practicality depends on the specific characteristics of the dataset and the requirements of the problem at hand. It is most effective for small to moderately sized datasets with locally homogeneous patterns.

19. What is the difference between Model Parameters Vs HyperParameters?

Model parameters and hyperparameters are both essential components of a machine learning model, but they serve different purposes and are tuned or learned in different ways.

Model Parameters:

1. **Definition:**

   - Model parameters are the internal variables that the model learns from the training data.
   - They are the coefficients or weights that are adjusted during the training process to minimize the difference between the predicted output and the actual output.

2. **Learning:**

   - Model parameters are learned through the optimization process, typically using optimization algorithms like gradient descent.
   - The model learns the optimal values for these parameters during the training phase.

3. **Examples:**

   - In linear regression, the coefficients of the linear equation are model parameters.
   - In a neural network, the weights and biases are model parameters.

4. **Unique to the Model:**

   - Model parameters are specific to the chosen algorithm and architecture.

Hyperparameters:

1. **Definition:**

   - Hyperparameters are external configuration settings that are set prior to the training process.
   - They are not learned from the data but are set by the machine learning practitioner.

2. **Tuning:**

   - Hyperparameters are tuned or selected by the user based on experimentation, prior knowledge, or search algorithms.
   - The goal is to find the combination of hyperparameter values that leads to the best model performance.

3. **Examples:**

   - Learning rate in gradient descent, the number of hidden layers and nodes in a neural network, (k) in k-Nearest Neighbors, etc.

4. **Affect Model Behavior:**

- Hyperparameters influence the overall behavior and performance of the model.
- Changing hyperparameter values can impact the model's ability to generalize and its computational efficiency.

Key Differences:

1. **Learned vs. Set:**
   - Model parameters are learned from the training data, while hyperparameters are set by the practitioner.
2. **Optimized vs. Tuned:**
   - Model parameters are optimized during training to minimize the loss function, while hyperparameters are tuned to improve the overall model performance.
3. **Algorithm-Specific vs. Global:**
   - Model parameters are specific to the chosen machine learning algorithm and its architecture, while hyperparameters can be global settings that affect the entire modeling process.
4. **Dynamic vs. Fixed:**
   - Model parameters are dynamic and change during training, while hyperparameters are typically fixed before training begins.

In summary, model parameters are the internal variables that the model learns, while hyperparameters are external configuration settings that the practitioner must set. The distinction is crucial for understanding how models learn from data and how their behavior can be influenced.

20. What is Hyperparameter Tuning in Machine Learning?

Hyperparameter tuning, also known as hyperparameter optimization, is the process of finding the optimal set of hyperparameter values for a machine learning model to achieve better performance. Hyperparameters are external configuration settings that are not learned from the data but are set prior to the training process. Tuning involves selecting the best combination of hyperparameter values to improve the overall effectiveness of the model.

Key Steps in Hyperparameter Tuning:

1. **Define Hyperparameters:**
   - Identify the hyperparameters that influence the behavior of the machine learning model. Examples include learning rate, regularization strength, the number of layers or nodes in a neural network, and the (k) value in k-Nearest Neighbors.
2. **Define a Search Space:**
   - Specify the range or possible values for each hyperparameter. This defines the search space within which the tuning algorithm will explore to find the optimal values.
3. **Choose a Search Method:**
   - Select a hyperparameter optimization technique or search method. Common methods include grid search, random search, and more advanced techniques like Bayesian optimization and genetic algorithms.
4. **Evaluate Models:**
   - Train and evaluate multiple models with different sets of hyperparameter values using a performance metric (e.g., accuracy, F1 score, mean squared error) on a validation set. The metric serves as a guide for comparing the models.
5. **Update Hyperparameters:**
   - Based on the performance of each model, update the hyperparameter values for subsequent iterations. The goal is to iteratively refine the search space and converge towards the optimal set of hyperparameters.
6. **Cross-Validation:**
   - Employ cross-validation to ensure that the model's performance is robust across different subsets of the data. This helps prevent overfitting to a specific dataset or random split.
7. **Final Evaluation:**

- After finding the best hyperparameter values, evaluate the final model on a separate test set that was not used during the tuning process to obtain an unbiased estimate of its performance.

Common Hyperparameter Tuning Techniques:

1. **Grid Search:**
    - Exhaustively searches through a predefined set of hyperparameter values, evaluating each combination.
2. **Random Search:**
    - Randomly samples hyperparameter values from specified distributions, providing a more efficient search strategy compared to grid search.
3. **Bayesian Optimization:**
    - Utilizes a probabilistic model to model the objective function (performance metric) and iteratively selects hyperparameters to optimize the objective.
4. **Genetic Algorithms:**
    - Mimics the process of natural selection by evolving a population of potential solutions (hyperparameter sets) over multiple generations.

Benefits of Hyperparameter Tuning:

1. **Improved Model Performance:**
    - Finding optimal hyperparameter values can lead to a model that generalizes better to unseen data.
2. **Enhanced Robustness:**
    - Tuning helps ensure that the model is robust across different datasets and is not overfitting to specific characteristics of the training data.
3. **Efficient Resource Utilization:**
    - Optimal hyperparameters may allow the model to achieve better performance with fewer computational resources and training time.

Hyperparameter tuning is a critical step in the machine learning workflow, and the choice of hyperparameters can significantly impact the success of a model. It requires a careful balance between exploration and exploitation to find the right configuration within the specified search space.

Example:

Let's consider a practical example of hyperparameter tuning using a popular machine learning algorithm, such as a Support Vector Machine (SVM) for classification. In this example, we'll focus on tuning two hyperparameters: the choice of kernel and the regularization parameter (C).

```python
# Import necessary libraries
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load the Iris dataset
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    test_size=0.2, random_state=42)

# Define the SVM model
svm_model = SVC()

# Define the hyperparameter grid to search
param_grid = {'kernel': ['linear', 'rbf', 'poly'], 'C': [0.1, 1, 10, 100]}

# Use GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(svm_model, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Print the best hyperparameter values
print("Best Hyperparameters:", grid_search.best_params_)
```

```
# Train the model with the best hyperparameters
best_model = grid_search.best_estimator_
best_model.fit(X_train, y_train)

# Evaluate the model on the test set
accuracy = best_model.score(X_test, y_test)
print("Test Accuracy with Tuned Hyperparameters:", accuracy)
```

In this example:

1. We use the Iris dataset, splitting it into training and test sets.
2. We define an SVM model with the `SVC` class from scikit-learn.
3. We set up a grid of hyperparameters to search, including different kernel types ('linear', 'rbf', 'poly') and different values of the regularization parameter (C).
4. We use `GridSearchCV` to perform a cross-validated search over the hyperparameter grid, evaluating models based on accuracy.
5. The best hyperparameters are printed, and the model is trained with these optimal values.
6. The final model is evaluated on the test set to assess its performance.

This is a simplified example, but the process of hyperparameter tuning is similar across various machine learning algorithms. It involves defining a search space, selecting a search method, and using cross-validation to find the combination of hyperparameter values that leads to the best model performance.

21. How to tune hyperparameters in K Nearest Neighbors Classifier?

Tuning hyperparameters in a k-Nearest Neighbors (KNN) classifier involves selecting the optimal values for parameters that are not learned from the data but are set before the training process. In the case of KNN, the primary hyperparameter is (k) (the number of neighbors).
Here's a step-by-step guide using Python and scikit-learn:

In [13]:

```
# Import necessary libraries
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load the Iris dataset
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)

# Define the KNN classifier
knn_model = KNeighborsClassifier()

# Define the hyperparameter grid to search
param_grid = {'n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15], 'weights': ['uniform', 'distance']}

# Use GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(knn_model, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Print the best hyperparameter values
print("Best Hyperparameters:", grid_search.best_params_)

# Train the model with the best hyperparameters
best_model = grid_search.best_estimator_
best_model.fit(X_train, y_train)

# Evaluate the model on the test set
accuracy = best_model.score(X_test, y_test)
print("Test Accuracy with Tuned Hyperparameters:", accuracy)

# Evaluate the model on the train set
accuracy_ = best_model.score(X_train, y_train)
print("Train Accuracy with Tuned Hyperparameters:", accuracy_)
```

```
Best Hyperparameters: {'n_neighbors': 3, 'weights': 'uniform'}
Test Accuracy with Tuned Hyperparameters: 1.0
Train Accuracy with Tuned Hyperparameters: 0.95
```

In this example:

1. We use the Iris dataset, splitting it into training and test sets.
2. We define a KNN classifier with the `KNeighborsClassifier` class from scikit-learn.
3. We set up a grid of hyperparameters to search, including different values for (k) (number of neighbors) and different weight options ('uniform', 'distance').
4. We use `GridSearchCV` to perform a cross-validated search over the hyperparameter grid, evaluating models based on accuracy.
5. The best hyperparameters are printed, and the model is trained with these optimal values.
6. The final model is evaluated on the test set to assess its performance.

You can modify the `param_grid` dictionary to include other hyperparameters of interest.
The process is similar for tuning hyperparameters in other machine learning models, but the specific hyperparameters and their potential values may vary.

22. What are the approaches for Hyperparameter tuning?

Hyperparameter tuning involves finding the optimal set of hyperparameter values for a machine learning model to improve its performance. There are several approaches to hyperparameter tuning, each with its own advantages and disadvantages.
Here are some common approaches:

1. **Grid Search:**

   - **Description:** Grid search is an exhaustive search technique where predefined hyperparameter values are tested in a combinatorial manner.
   - **Pros:**
     - Straightforward and easy to implement.
     - Systematic exploration of hyperparameter space.
   - **Cons:**
     - Can be computationally expensive for large search spaces.

   ```
   from sklearn.model_selection import GridSearchCV
   ```
2. **Random Search:**

   - **Description:** Random search randomly samples hyperparameter combinations from predefined distributions.
   - **Pros:**
     - More computationally efficient than grid search.
     - May discover good hyperparameter values faster, especially in high-dimensional spaces.
   - **Cons:**
     - May not explore the entire space as thoroughly as grid search.

   ```
   from sklearn.model_selection import RandomizedSearchCV
   ```
3. **Bayesian Optimization:**

   - **Description:** Bayesian optimization models the objective function (model performance) as a probabilistic surrogate, allowing for more informed sampling of hyperparameter values.
   - **Pros:**
     - Efficient for expensive-to-evaluate objective functions.
     - Balances exploration and exploitation.
   - **Cons:**
     - More complex to implement compared to grid or random search.
     - Requires additional libraries (e.g., `scikit-optimize`, `BayesianOptimization`).

   ```
   from skopt import BayesSearchCV
   ```
4. **Genetic Algorithms:**

   - **Description:** Genetic algorithms use principles of natural selection to evolve a population of potential hyperparameter combinations over multiple generations.
   - **Pros:**
     - Effective for high-dimensional spaces.
     - Can handle complex search spaces.
   - **Cons:**
     - Computationally expensive.
     - Requires tuning of algorithm-specific parameters.

```
from evolutionary_search import EvolutionaryAlgorithmSearchCV
```

5. **Manual Search:**
   - **Description:** Manually trying different hyperparameter values based on domain knowledge or intuition.
   - **Pros:**
     - Allows for expert intuition and domain-specific knowledge.
     - Simplicity.
   - **Cons:**
     - Time-consuming.
     - May not explore the search space thoroughly.

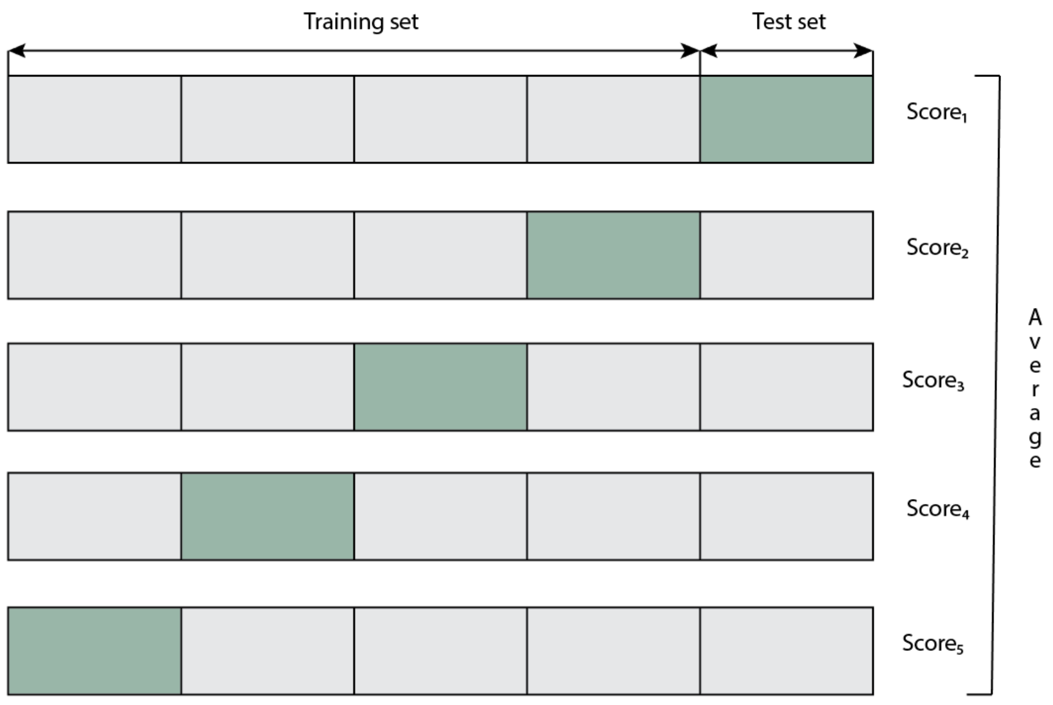
```
# Manually adjust hyperparameters in the code
```

6. **Automated Approaches (AutoML):**
   - **Description:** Automated Machine Learning (AutoML) platforms, such as Google AutoML, H2O.ai, or TPOT, automate the entire process of model selection and hyperparameter tuning.
   - **Pros:**
     - Fully automated.
     - Requires minimal user intervention.
   - **Cons:**
     - May lack transparency and control compared to manual approaches.
     - Limited customization.

```
# Depends on the AutoML library used
```

Each approach has its own trade-offs, and the choice depends on factors such as the size of the search space, computational resources, and the desired level of automation. It's common to start with a simpler approach like grid search and then explore more advanced methods as needed.

23. What is cross-validation?



Cross-validation is a statistical technique used in machine learning to assess how well a predictive model will generalize to an independent dataset. It involves partitioning the original dataset into multiple subsets, training the model on some of these subsets, and evaluating its performance on the remaining subsets. The primary goal of cross-validation is to provide a more accurate estimate of a model's performance by using multiple training and testing sets.

The most common form of cross-validation is k-fold cross-validation, where the dataset is divided into k equally-sized folds.
The following steps outline the k-fold cross-validation process:

1. **Data Splitting:**
    - The dataset is divided into k subsets, or "folds," of approximately equal size.
2. **Model Training and Evaluation:**
    - The model is trained k times, each time using k-1 folds as the training set and the remaining fold as the testing set. This results in k different models.
3. **Performance Metrics:**
    - The performance of the model is evaluated using a predefined metric (e.g., accuracy, precision, recall, F1 score) on each testing set.
4. **Average Performance:**
    - The performance metrics from each iteration are averaged to provide an overall assessment of the model's performance.

Advantages of Cross-Validation:

1. **Reduced Bias:**
    - Cross-validation provides a more robust estimate of a model's performance by reducing bias associated with a single train-test split.
2. **Effective Use of Data:**
    - It allows for the effective use of available data, as each data point is used for both training and testing across different folds.
3. **Model Generalization:**
    - Cross-validation helps assess how well a model generalizes to new, unseen data, which is crucial for evaluating model performance in real-world scenarios.

Types of Cross-Validation:

1. **k-Fold Cross-Validation:**
    - Divides the dataset into k subsets, training the model on k-1 folds and testing on the remaining fold.
2. **Stratified k-Fold Cross-Validation:**
    - Similar to k-fold, but ensures that each fold maintains the same distribution of the target variable as the original dataset. It is particularly useful for imbalanced datasets.
3. **Leave-One-Out Cross-Validation (LOOCV):**
    - Each data point serves as the testing set once, and the model is trained on all other data points. Suitable for small datasets.
4. **Shuffle-Split Cross-Validation:**
    - Randomly shuffles the data and splits it into training and testing sets for multiple iterations. Useful for large datasets.
5. **Time Series Cross-Validation:**
    - Specific to time series data, it involves using past data for training and future data for testing to simulate real-world scenarios.

Cross-validation is an essential tool for model evaluation, hyperparameter tuning, and ensuring that the model's performance estimates are reliable and not overly optimistic or pessimistic.

24. What are GridSearchCV and RandomizedSearchCV, differences between them?

`GridSearchCV` and `RandomizedSearchCV` are techniques used for hyperparameter tuning in machine learning models, and they have some key differences in terms of their search strategies and efficiency.

GridSearchCV:

1. **Search Strategy:**
    - **Exhaustive Search:** Grid search performs an exhaustive search over a specified hyperparameter grid, considering all possible combinations of hyperparameter values.
2. **Hyperparameter Space Exploration:**
    - **Defined Grid:** It explores a predefined grid of hyperparameter values. The user specifies the values to be tested for each hyperparameter.
3. **Computational Cost:**
    - **High:** Grid search can be computationally expensive, especially when the hyperparameter space is large, as it tests all possible combinations.
4. **Implementation:**
    - `GridSearchCV` **in scikit-learn:** The `GridSearchCV` function is part of the scikit-learn library and is commonly used for hyperparameter tuning.
5. **Example Code:**

```
from sklearn.model_selection import GridSearchCV
grid_search = GridSearchCV(estimator, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X, y)
```

RandomizedSearchCV:

1. **Search Strategy:**
    - **Randomized Search:** Randomized search samples a specified number of hyperparameter combinations randomly from the hyperparameter space.
2. **Hyperparameter Space Exploration:**
    - **Random Sampling:** It explores a randomly selected subset of the hyperparameter space. Users specify distributions for hyperparameter values, and the search samples from these distributions.
3. **Computational Cost:**
    - **Lower:** Randomized search is computationally less expensive compared to grid search, especially when the hyperparameter space is large, as it tests a random subset of combinations.
4. **Implementation:**
    - `RandomizedSearchCV` **in scikit-learn:** The `RandomizedSearchCV` function is also part of the scikit-learn library.
5. **Example Code:**

```
from sklearn.model_selection import RandomizedSearchCV
randomized_search = RandomizedSearchCV(estimator, param_distributions, n_iter=10, cv=5,
scoring='accuracy')
randomized_search.fit(X, y)
```

Key Differences:

- **Exploration Strategy:**
    - Grid search explores all possible combinations in a predefined grid.
    - Randomized search explores a random subset of combinations from distributions specified by the user.
- **Computational Cost:**
    - Grid search can be computationally expensive, especially for large hyperparameter spaces.
    - Randomized search is computationally less expensive due to its random sampling strategy.
- **Flexibility:**
    - Grid search is suitable for smaller hyperparameter spaces where an exhaustive search is feasible.
    - Randomized search is more suitable for larger hyperparameter spaces where an exhaustive search is impractical.
- **User Input:**
    - Grid search requires the user to specify exact values for each hyperparameter.
    - Randomized search allows the user to specify distributions for hyperparameter values.

In summary, while grid search systematically explores all possible combinations, randomized search efficiently samples a subset, making it more suitable for large hyperparameter spaces and scenarios where computational resources are limited. The choice between them depends on factors such as the size of the hyperparameter space and the available computing resources.

25. What are the Applications of KNN?

k-Nearest Neighbors (KNN) is a versatile and simple algorithm used in various applications across different domains. Here are some common applications of KNN:

1. **Classification:**
   - KNN is frequently used for classification tasks, where the goal is to assign a label to a data point based on the majority label of its k-nearest neighbors. It's applicable to both binary and multiclass classification problems.
2. **Regression:**
   - KNN can be used for regression tasks, where the goal is to predict a continuous value. The predicted value is often the average or weighted average of the target values of the k-nearest neighbors.
3. **Recommendation Systems:**
   - In collaborative filtering-based recommendation systems, KNN can be used to identify similar users or items. It recommends items that are liked or rated highly by users with similar preferences.
4. **Anomaly Detection:**
   - KNN can be applied to detect anomalies or outliers in a dataset by identifying data points that have significantly fewer neighbors than others. It's sensitive to local data patterns, making it suitable for anomaly detection.
5. **Image and Speech Recognition:**
   - KNN can be used in image recognition tasks by comparing the features of an image with those of its nearest neighbors. Similarly, in speech recognition, KNN can identify patterns in speech data to recognize spoken words.
6. **Text Mining:**
   - KNN is applied in text classification tasks, such as sentiment analysis or spam detection. It can be used to classify documents based on the similarity of their content.
7. **Bioinformatics:**
   - In bioinformatics, KNN is used for tasks such as protein-protein interaction prediction, gene expression analysis, and identifying disease biomarkers.
8. **Customer Segmentation:**
   - KNN can be employed in customer segmentation, where customers with similar purchasing behavior are grouped together. It's useful for targeted marketing and personalized recommendations.
9. **Credit Scoring:**
   - KNN can be used in credit scoring to assess the creditworthiness of individuals by comparing their financial behavior with that of similar individuals.
10. **Geographical Data Analysis:**
    - KNN is applied in geographical data analysis, such as predicting air quality, traffic congestion, or crime rates based on the characteristics of neighboring locations.
11. **Robotics:**
    - In robotics, KNN can be used for obstacle avoidance by identifying nearby obstacles and determining the robot's next move based on the surrounding environment.
12. **Healthcare:**
    - KNN is utilized in healthcare for tasks like disease prediction and patient diagnosis by comparing the health characteristics of patients with similar cases.

While KNN is powerful in certain scenarios, its effectiveness depends on the characteristics of the data, the choice of distance metric, and the value of k. Additionally, it may not scale well to high-dimensional or large datasets. Careful consideration and experimentation are necessary when applying KNN to specific applications.

26. How do you deal with outliers values in a dataset?

Handling outliers in a dataset is crucial for ensuring that statistical analyses and machine learning models are robust and not unduly influenced by extreme values.
Here are several methods commonly used to deal with outliers:

1. **Identification and Visualization:**
   - Start by identifying outliers using statistical methods (e.g., Z-scores, IQR) or visualization tools like box plots, scatter plots, or histograms. Understanding the nature of outliers is essential before deciding on a strategy.
2. **Data Truncation or Capping:**
   - Remove or cap extreme values by setting a threshold beyond which values are considered outliers. This approach is straightforward but may result in information loss.
3. **Data Transformation:**
   - Apply mathematical transformations to the data to reduce the impact of outliers. Common transformations include taking the logarithm, square root, or reciprocal of the values. These transformations can help normalize the distribution.
4. **Winsorizing:**
   - Similar to capping, winsorizing involves setting extreme values to a specified percentile (e.g., 95th or 99th percentile) instead of removing them. This retains information while mitigating the impact of outliers.
5. **Imputation:**
   - For missing values due to outliers, consider imputing values using statistical measures such as mean, median, or a more advanced imputation method. Imputation helps fill in missing values without removing entire data points.
6. **Robust Statistical Methods:**
   - Use robust statistical methods that are less sensitive to outliers. For example, replacing the mean with the median or using the interquartile range (IQR) instead of the standard deviation can provide more robust measures of central tendency and variability.
7. **Model-Based Detection and Removal:**
   - Train a model to identify outliers and remove them based on the model's predictions. This could involve using clustering algorithms or anomaly detection techniques.
8. **Binning or Discretization:**
   - Group data into bins or discrete intervals. This approach may make the analysis more robust to extreme values, especially in cases where the specific value of the variable is not critical.
9. **Robust Scaling:**
   - Use robust scaling techniques, such as the median and interquartile range (IQR) scaling, instead of standard scaling based on the mean and standard deviation. This can make models less sensitive to outliers during training.
10. **Domain-Specific Approaches:**
    - Depending on the domain knowledge and the nature of the data, specific approaches may be more appropriate. Consultation with domain experts can provide insights into whether outliers should be retained or removed.

It's important to note that the choice of method depends on the characteristics of the data and the goals of the analysis or modeling task. There is no one-size-fits-all solution, and the impact of outlier handling should be carefully evaluated to ensure that it aligns with the objectives of the analysis or model.

We can create user-defined functions for removing outliers using Z-scores and IQR methods. These functions take a dataset as input and return a new dataset with outliers removed based on the specified method.

1) Remove Outliers using Z-scores:

```
import numpy as np

def remove_outliers_zscore(data, z_threshold=3):
    """
    Remove outliers from a dataset using Z-scores.

    Parameters:
    - data: numpy array or pandas DataFrame, the input dataset
    - z_threshold: float, the Z-score threshold beyond which data points
    are considered outliers

    Returns:
    - data_no_outliers: numpy array or pandas DataFrame, the dataset
    with outliers removed
    """
    z_scores = np.abs((data - np.mean(data)) / np.std(data))
    mask_no_outliers = z_scores < z_threshold
    data_no_outliers = data[mask_no_outliers]
    return data_no_outliers
```

2) Remove Outliers using IQR (Interquartile Range):

```
import numpy as np

def remove_outliers_iqr(data, iqr_factor=1.5):
    """
    Remove outliers from a dataset using the Interquartile Range (IQR) method.

    Parameters:
    - data: numpy array or pandas DataFrame, the input dataset
    - iqr_factor: float, a factor multiplied by the IQR to determine the
    upper and lower bounds

    Returns:
    - data_no_outliers: numpy array or pandas DataFrame, the dataset with
    outliers removed
    """
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr = q3 - q1
    lower_bound = q1 - iqr_factor * iqr
    upper_bound = q3 + iqr_factor * iqr
    mask_no_outliers = (data >= lower_bound) & (data <= upper_bound)
    data_no_outliers = data[mask_no_outliers]
    return data_no_outliers
```

You can use these functions by passing your dataset (as a NumPy array or a Pandas DataFrame) to them. For example:

```
import numpy as np

# Example dataset
data = np.array([1, 2, 3, 4, 5, 1000])

# Remove outliers using Z-scores
data_no_outliers_zscore = remove_outliers_zscore(data)

# Remove outliers using IQR
data_no_outliers_iqr = remove_outliers_iqr(data)
```

Adjust the parameters such as `z_threshold` and `iqr_factor` according to your specific use case.

27. How do you deal with missing values in a dataset?

Handling missing values in a dataset is crucial to ensure the robustness of statistical analyses and machine learning models. The appropriate strategy for dealing with missing values depends on the nature of the data and the reasons for the missingness.
Here are several common methods:

1. **Remove Missing Values:**

   - If the proportion of missing values is small and missing values are randomly distributed, removing rows or columns with missing values might be a simple and effective approach. However, this should be done cautiously, as it may lead to information loss.

   ```
   # Remove rows with missing values
   df_cleaned = df.dropna()

   # Remove columns with missing values
   df_cleaned = df.dropna(axis=1)
   ```

2. **Imputation with Mean/Median/Mode:**

   - Fill missing values with the mean, median, or mode of the respective columns. This is suitable for numerical data and can be a straightforward approach.

   ```
   # Impute missing values with mean
   df_filled = df.fillna(df.mean())

   # Impute missing values with median
   df_filled = df.fillna(df.median())

   # Impute missing values with mode
   df_filled = df.fillna(df.mode().iloc[0])
   ```

3. **Imputation with Predictive Models:**

   - Use predictive models (e.g., regression models) to impute missing values based on the relationship with other variables. This method is more advanced but can provide accurate imputations.

   ```
   from sklearn.impute import KNNImputer

   # Use KNN imputer for imputation
   knn_imputer = KNNImputer(n_neighbors=5)
   df_imputed = pd.DataFrame(knn_imputer.fit_transform(df), columns=df.columns)
   ```

4. **Forward Fill or Backward Fill (Time Series):**

   - For time series data, you can use forward fill (fill missing values with the previous value) or backward fill (fill missing values with the next value).

   ```
   # Forward fill
   df_filled = df.ffill()

   # Backward fill
   df_filled = df.bfill()
   ```

5. **Interpolation:**

   - Interpolation methods estimate missing values based on the values of neighboring points. Methods like linear interpolation or spline interpolation can be used.

   ```
   # Linear interpolation
   df_interpolated = df.interpolate(method='linear')
   ```

6. **Flagging Missing Values:**

   - Instead of imputing, you can add an additional binary column to indicate whether a value was missing or not. This way, you retain information about the missingness.

   ```
   # Create a binary column indicating missing values
   df['feature_missing'] = df['feature'].isnull().astype(int)
   ```

7. **Multiple Imputation:**

   - For more sophisticated analyses, multiple imputation techniques can be employed. This involves creating multiple datasets with imputed values to account for uncertainty in the imputation process.

```
from sklearn.impute import IterativeImputer

# Use iterative imputer for multiple imputation
iterative_imputer = IterativeImputer()
df_imputed = pd.DataFrame(iterative_imputer.fit_transform(df), columns=df.columns)
```

The choice of method depends on the context of your data, the extent of missingness, and the assumptions you're willing to make. It's often beneficial to visualize the patterns of missing values before deciding on a strategy. Additionally, understanding the reasons behind missingness can inform the most appropriate handling method.

28. How do you use Knn for missing value imputation?

Using k-Nearest Neighbors (KNN) for missing value imputation involves estimating missing values based on the values of their k-nearest neighbors in the feature space.
The basic idea is to find the most similar data points to the one with missing values and use their values to impute the missing ones.
Here's a step-by-step guide on how to use KNN for missing value imputation:

In [21]:
```
### Step 1: Import Libraries


import numpy as np
import pandas as pd
from sklearn.impute import KNNImputer


### Step 2: Create or Load Data

# Example DataFrame with missing values
data = {
    'Feature1': [1, 2, np.nan, 4, 5],
    'Feature2': [5, 4, 3, np.nan, 1],
    'Feature3': [2, 4, 6, 8, 10]
}

df = pd.DataFrame(data)


### Step 3: Instantiate KNNImputer


# Instantiate KNNImputer with the desired number of neighbors
knn_imputer = KNNImputer(n_neighbors=3)


### Step 4: Fit and Transform

# Fit the KNNImputer on the data and transform it to impute missing values:


# Fit and transform the data
df_imputed = pd.DataFrame(knn_imputer.fit_transform(df), columns=df.columns)


### Step 5: Inspect Results


# Display the imputed DataFrame
print("Original DataFrame:\n")
display(df)
print("\nDataFrame after KNN imputation:\n")
display(df_imputed)
```

```
Original DataFrame:
```

	Feature1	Feature2	Feature3
0	1.0	5.0	2
1	2.0	4.0	4
2	NaN	3.0	6
3	4.0	NaN	8
4	5.0	1.0	10

```
DataFrame after KNN imputation:
```

	Feature1	Feature2	Feature3
0	1.000000	5.000000	2.0
1	2.000000	4.000000	4.0
2	2.333333	3.000000	6.0
3	4.000000	2.666667	8.0
4	5.000000	1.000000	10.0

In this example, the `KNNImputer` replaces missing values by considering the values of the k-nearest neighbors for each missing entry. Adjust the `n_neighbors` parameter based on your specific dataset characteristics.

Keep in mind the following considerations:

- The KNN imputation method is sensitive to the choice of the distance metric. The default is Euclidean distance, but you can specify other distance metrics based on your data characteristics.
- The method works well when the assumption of local similarity holds, and the missing values are not completely at random.
- Depending on the structure of your data, it may be useful to normalize or scale the features before applying KNN imputation.
- KNN imputation may be computationally expensive for large datasets, as it involves computing distances between data points.

As with any imputation method, it's essential to assess the appropriateness of KNN imputation for your specific use case and dataset. Evaluate the imputed data's performance in downstream analyses or modeling tasks to ensure the validity of the imputation.

29. What is NaN Euclidean Distance?

The concept of "NaN Euclidean Distance" refers to the distance metric used when calculating the Euclidean distance between two points in a dataset that may contain missing values (NaN, Not a Number). In the context of Euclidean distance computation, the presence of missing values requires a specialized approach to handle these situations.

The Euclidean distance between two points $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_n)$ in an n-dimensional space is given by the formula:

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

When dealing with missing values (NaN) in one or both of the points, a common approach is to modify the Euclidean distance calculation to handle these situations. Here are a few common strategies:

1. **Pairwise Deletion:**
   - If a dimension has a missing value in either (P) or (Q), exclude that dimension from the distance calculation. This approach essentially ignores missing values.
2. **Imputation:**

- Impute the missing values in (P) and (Q) before calculating the distance. This could involve replacing missing values with the mean, median, or another imputation strategy.

3. **Weighted Euclidean Distance:**

   - Assign weights to dimensions based on the availability of values. For example, if a dimension has a missing value in one point but not in the other, the contribution of that dimension to the distance calculation could be weighted differently.

4. **Special Handling for NaN:**

   - Introduce special handling for NaN values in the Euclidean distance calculation. For instance, consider setting the contribution of NaN values to zero or treating them as a distinct category.

The specific approach depends on the context of the data and the goals of the analysis. It's crucial to arefully consider the implications of each strategy, especially when missing values are not missing completely at random (MCAR), and the choice of handling strategy might introduce bias.

Some machine learning libraries, like scikit-learn in Python, provide functionality to handle missing values in distance calculations. For example, the `nan_euclidean_distances` function in scikit-learn can be used for calculating Euclidean distances with NaN values.

```python
from sklearn.metrics import pairwise_distances_nan_euclidean

# Example usage
distances = pairwise_distances_nan_euclidean(X)
```

Always check the documentation of the specific tools or libraries you are using for details on handling missing values in distance calculations.

30. What will be the distance between the below data points:

$P_1[3, \text{NaN}, \text{NaN}, 6]$ & $P_2[1, \text{NaN}, 4, 5]$

When calculating the Euclidean distance between two points with missing values (NaN), one common approach is to ignore the dimensions with missing values during the distance computation.
In this case, the Euclidean distance between $P_1[3, \text{NaN}, \text{NaN}, 6]$ & $P_2[1, \text{NaN}, 4, 5]$ would be computed as follows:

$$\text{Euclidean Distance} = \sqrt{(3-1)^2 + (6-5)^2}$$

Here, the second and third dimensions contain missing values (NaN) in one or both points, so those dimensions are excluded from the distance calculation.
The remaining dimensions are considered for the computation.

Using the above formula:

$$\text{Euclidean Distance} = \sqrt{(2)^2 + (1)^2}$$

$$\text{Euclidean Distance} = \sqrt{4 + 1}$$

$$\text{Euclidean Distance} = \sqrt{5}$$

So, the Euclidean distance between the points $P_1[3, \text{NaN}, \text{NaN}, 6]$ and $P_2[1, \text{NaN}, 4, 5]$ is $\sqrt{5}$