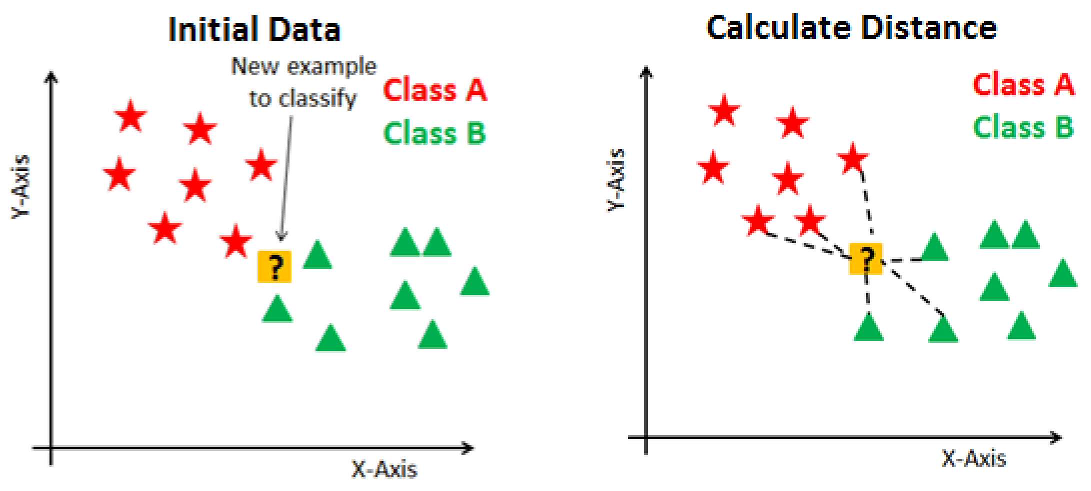
KNN

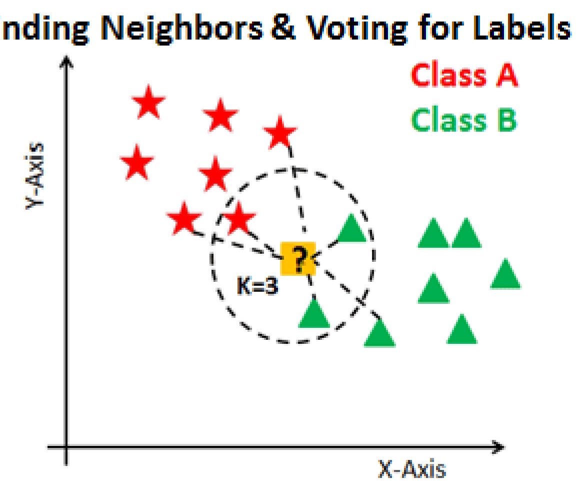
Agenda:

- Introduction to K Nearest Neighbors (KNN) Algorithm
  - KNN for Classification
  - KNN for Regression
- When Do We Use the KNN Algorithm?
- Basics of KNN
  - Distance Metrics
  - Manhattan Distance
  - Other Similarity Measures: Cosine Similarity
- Decision Making Process
- Hyperparameter K
  - How does the KNN algorithm work?
- How do we choose the factor k?
- Simulation
- Scikit-learn implementation of KNN
  - Confusion Matrix
  - Classification Report
  - Manhattan Distance
  - Parameter Tuning
- Multiclass Classification
  - Principal Component Analysis (PCA)
  - Preprocessing and training KNN classifier for multiclass classification
- Imputation
  - Imputation Techniques
    - Mean Imputation
    - Median Imputation
    - Zero Imputation
    - Placeholder Imputation
  - Advantages of Imputation Techniques
  - Limitations of Imputation Techniques
- KNN Imputation

Introduction to K Nearest Neighbors (KNN) Algorithm

- The K-Nearest Neighbors (KNN) algorithm is a popular machine learning technique used for classification and regression tasks. It relies on the idea that similar data points tend to have similar labels or values.
- During the training phase, the KNN algorithm stores the entire training dataset as a reference. When making predictions, it calculates the distance between the input data point and all the training examples, using a chosen distance metric such as Euclidean distance.
- Next, the algorithm identifies the K nearest neighbors to the input data point based on their distances. In the case of classification, the algorithm assigns the most common class label among the K neighbors as the predicted label for the input data point. For regression, it calculates the average or weighted average of the target values of the K neighbors to predict the value for the input data point.
- The KNN algorithm is straightforward and easy to understand, making it a popular choice in various domains. However, its performance can be affected by the choice of K and the distance metric, so careful parameter tuning is necessary for optimal results





KNN for Classification

- Prediction Process: For classification tasks, KNN predicts the class of a query point based on the majority class among its K nearest neighbors.
- Voting Mechanism: Each neighbor's class contributes to the final prediction, with the class receiving the most votes becoming the predicted class.
- Hyperparameter Selection: Experimentation with different values of K helps in determining the optimal performance of the classifier.
  -

KNN for Regression

- Prediction Process: In regression tasks, KNN predicts the value of a query point by averaging the values of its K nearest neighbors.
- Averaging Technique: The predicted value is calculated as the mean or weighted mean of the target values of the nearest neighbors.
- Use Cases: Regression with KNN is effective for predicting continuous variables based on neighboring data points' values.

When Do We Use the KNN Algorithm?

The K-Nearest Neighbors (KNN) algorithm is commonly used in supervised machine learning for both classification and regression tasks.

Here's when you might consider using KNN:

1. Classification Tasks: KNN is often used for classification problems where the output is a categorical variable. It works by assigning a data point to the class most common among its K nearest neighbors in the feature space.
2. Regression Tasks: KNN can also be used for regression problems where the output is a continuous variable. In regression, the predicted value for a data point is the average of the values of its K nearest neighbors.
3. Non-linear Data: KNN can effectively handle non-linear data because it doesn't make any assumptions about the underlying data distribution. It relies solely on the proximity of data points in the feature space.
4. Small to Medium-Sized Datasets: KNN can perform well with small to medium-sized datasets. However, its computational complexity grows linearly with the size of the dataset, so it may not be suitable for very large datasets.
5. Simple Implementation: KNN is easy to understand and implement, making it a good choice for quick prototyping or as a baseline algorithm for comparison with more complex models.
6. No Training Phase: KNN is instance-based and does not require a training phase. The model simply memorizes the training data, making it suitable for incremental learning scenarios.
7. Robust to Noisy Data: KNN can handle noisy data by considering multiple neighbors for classification or regression, which can help reduce the impact of outliers.

However, there are also some limitations to consider:

1. High Computational Cost during Inference: Predicting the label of a new data point requires computing distances to all training samples, which can be computationally expensive for large datasets.
2. Sensitive to Feature Scaling: KNN's distance metric is sensitive to the scale of features. Therefore, it's essential to normalize or standardize features before applying KNN to ensure all features contribute equally to the distance computation.
3. Memory Usage: As KNN memorizes the entire training dataset, it can be memory-intensive for large datasets.
4. Determining the Optimal Value of K: Choosing the appropriate value of K is crucial for KNN performance. A small K can make the model sensitive to noise, while a large K can lead to oversmoothing and loss of important patterns.

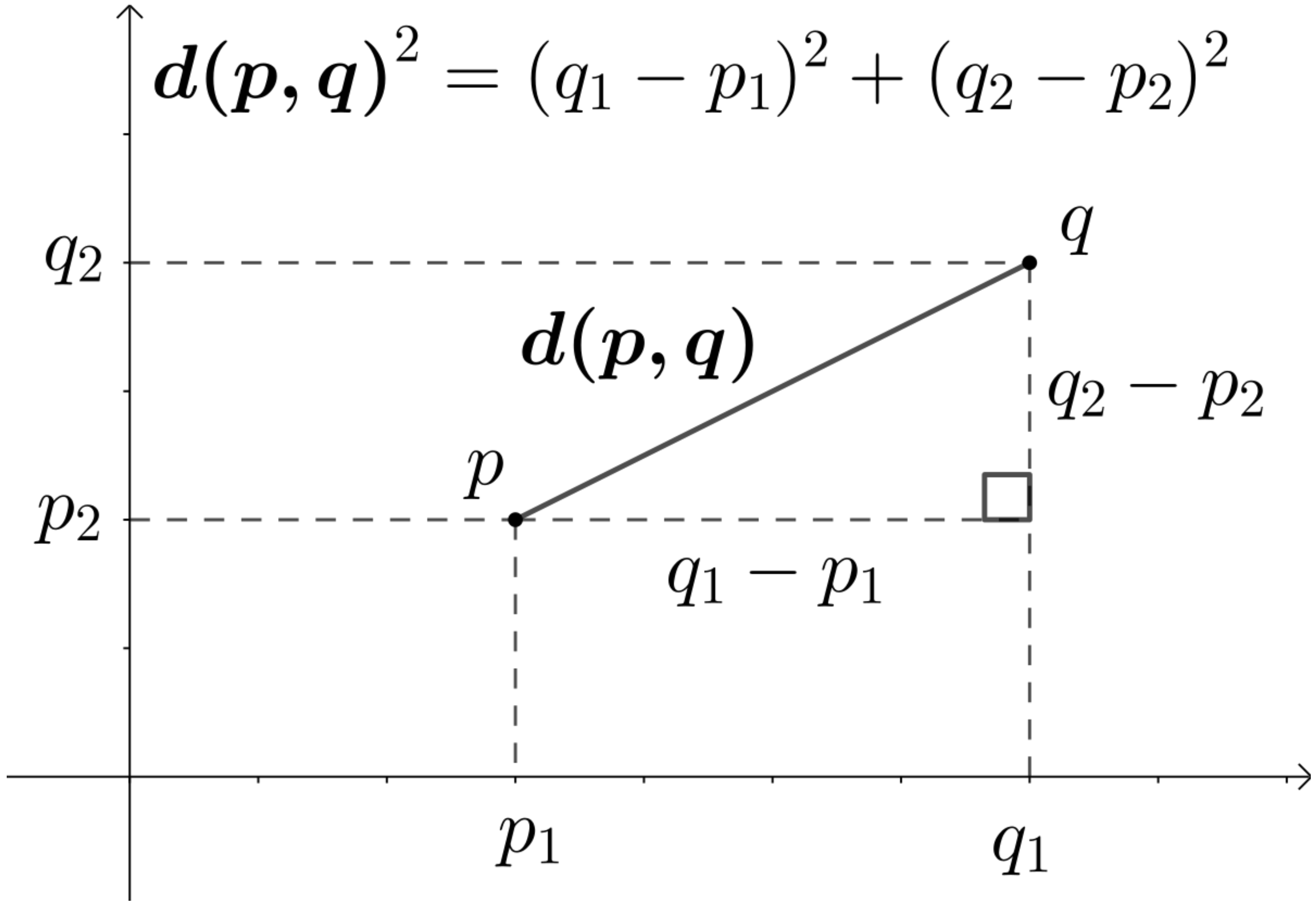
Basics of KNN

Distance Metrics

Distance metrics quantify the similarity or dissimilarity between data points in a feature space. In the context of K Nearest Neighbors (KNN), distance metrics are used to measure how 'close' or 'far' one data point is from another.

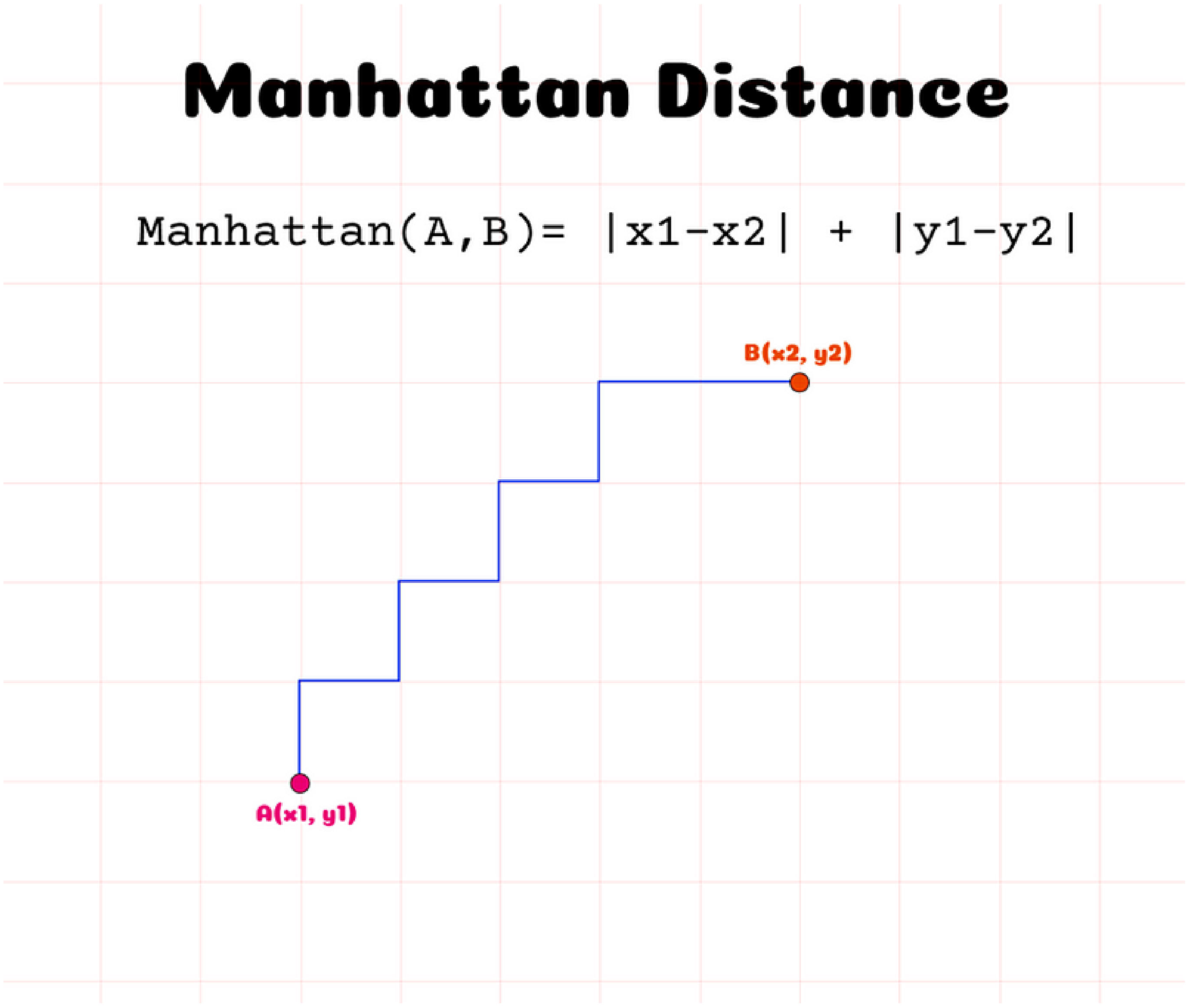
Common Metrics:

Mathematically, for two points p=(p_1, p_2,...,p_n) and q=(q_1, q_2,...,q_n) , the Euclidean distance is given by:



Manhattan Distance

Also known as city block distance or taxicab distance, it measures the sum of absolute differences between the coordinates of two points. It is particularly useful when dealing with high-dimensional data or when features have different scales. Mathematically, for two points A and B, the Manhattan distance is given by:



Other Similarity Measures: Cosine Similarity

While not a distance metric in the strict sense, cosine similarity measures the cosine of the angle between two vectors, representing the direction similarity between them. It is often used in text data or high-dimensional data where the magnitude of the vectors is not as important as their orientations.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Decision Making Process

KNN's decision-making process involves the following steps:

1. Finding Nearest Neighbors: Given a query point, the algorithm identifies the K nearest neighbors from the training dataset based on the chosen distance metric.
2. Majority Voting (Classification): For classification tasks, the algorithm counts the occurrences of each class among the K neighbors and assigns the query point to the class with the highest count (mode).
3. Averaging (Regression): For regression tasks, the algorithm calculates the average (or weighted average) of the target values of the K nearest neighbors and assigns this value to the query point.

Hyperparameter K

K is a hyperparameter of the KNN algorithm that determines the number of nearest neighbors to consider when making predictions.

The choice of K significantly impacts the performance and behavior of the KNN algorithm.

How Does the KNN Algorithm Work?

The K-Nearest Neighbors (KNN) algorithm is a simple, instance-based learning algorithm used for both classification and regression tasks.

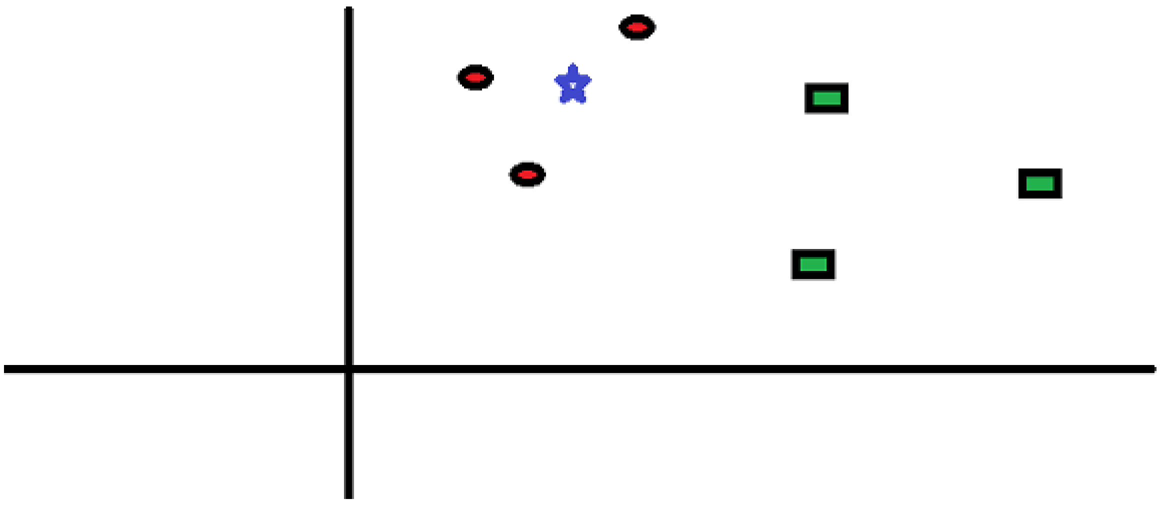
Here's how it works:

1. Training: KNN does not involve a training phase in the traditional sense. Instead, it memorizes the entire training dataset. This means that during the training phase, the algorithm simply stores the feature vectors and corresponding labels of all the training instances.
2. Prediction:
   - For Classification: When a new, unseen data point is presented for classification, KNN identifies the K closest training instances (neighbors) to this data point based on a distance metric (typically Euclidean distance).
   - For Regression: In regression tasks, instead of class labels, KNN considers the numerical values associated with the nearest neighbors.
3. Determining the Output:
   - For Classification: The algorithm assigns the class label that is most common among the K nearest neighbors. This is often done by simple majority voting, where the class with the highest frequency among the neighbors is assigned to the new data point.
   - For Regression: The predicted value for the new data point is typically the average (mean) of the values of its K nearest neighbors.

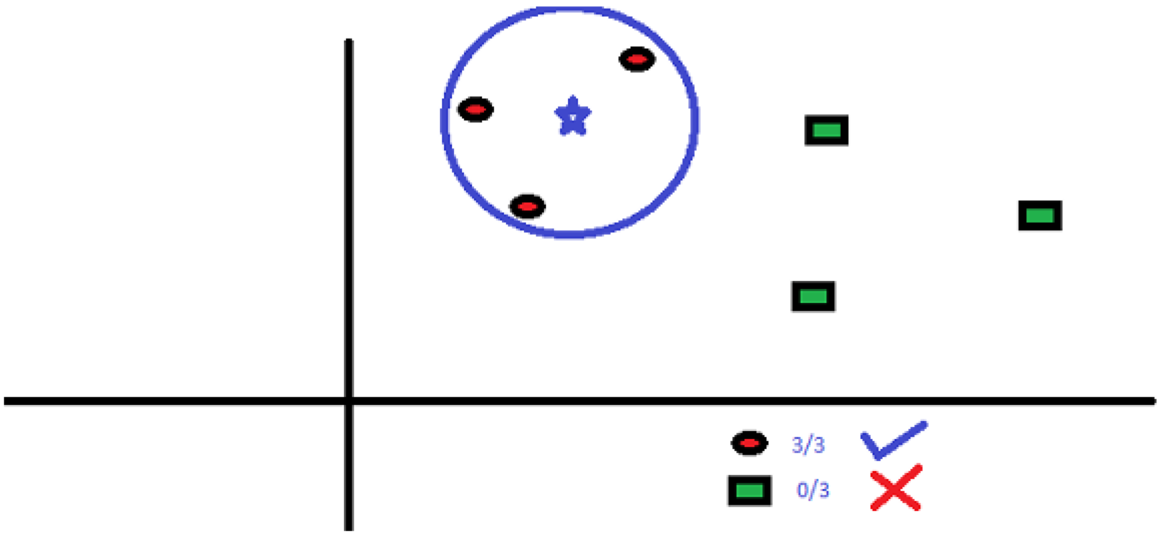
4. Choosing the Value of K: The choice of K, the number of neighbors to consider, is crucial in KNN. A smaller value of K may lead to overfitting, making the model sensitive to noise, while a larger value of K may lead to oversmoothing, potentially ignoring local patterns. The optimal value of K is often determined through cross-validation.
5. Distance Metric: The distance metric used to measure the proximity between data points is typically Euclidean distance. However, other distance metrics such as Manhattan distance or cosine similarity can also be used, depending on the nature of the data and the problem at hand.
6. Handling Ties: In classification tasks, ties can occur when two or more classes have the same number of votes among the K nearest neighbors. In such cases, various strategies can be employed, such as selecting the class with the smallest distance or randomly choosing among the tied classes.

Example:

Let’s take a simple case to understand this algorithm. Following is a spread of red circles (RC) and green squares (GS):



You intend to find out the class of the blue star (BS). BS can either be RC or GS and nothing else. The “K” in KNN algorithm is the nearest neighbor we wish to take the vote from. Let’s say K = 3. Hence, we will now make a circle with BS as the center just as big as to enclose only three data points on the plane. Refer to the following diagram for more details:



The three closest points to BS are all RC. Hence, with a good confidence level, we can say that the BS should belong to the class RC. Here, the choice became obvious as all three votes from the closest neighbor went to RC. The choice of the parameter K is very crucial in this algorithm.

How Do We Choose the Factor K?

Choosing the value of K, the number of nearest neighbors to consider is a crucial aspect of the K-Nearest Neighbors (KNN) algorithm. The choice of K can significantly impact the performance and behavior of the model.

Here are some common methods for selecting the value of K:

1. Grid Search with Cross-Validation: One of the most common techniques is to perform a grid search over a range of possible values of K, typically from 1 to some maximum value. For each value of K, cross-validation is used to evaluate the model's performance on a validation set. The value of K that yields the best performance metric (such as accuracy for classification or mean squared error for regression) on the validation set is selected.
2. Odd Values for Binary Classification: When dealing with binary classification problems, it's often recommended to choose an odd value for K to avoid ties when determining the majority class. This helps in breaking ties more effectively.
3. Square Root Rule: The square root of the total number of samples in the training dataset is sometimes used as a heuristic to determine the value of K. This rule suggests that K should be approximately the square root of the number of data points. For example, if you have 100 data points, you might start with K = 10.
4. Domain Knowledge and Experimentation: In some cases, domain knowledge about the problem or the dataset can guide the selection of K. For instance, if you know that the decision boundary between classes is smooth, a larger value of K might be appropriate. Experimenting with different values of K and evaluating the model's performance can also provide insights into the suitable range of values.

5. Model Complexity and Overfitting: Consider the trade-off between model complexity and overfitting. A smaller value of K may lead to a more complex decision boundary, potentially capturing noise in the data (overfitting), while a larger value of K may lead to oversmoothing and loss of important patterns.
6. Use Case and Performance Goals: The choice of K may also depend on the specific requirements and goals of your application. For example, if computational efficiency is critical, you might choose a smaller value of K even if it slightly reduces performance.

Simulation

```
# Importing necessary libraries
import pandas as pd  # For data manipulation and analysis
import numpy as np   # For numerical operations

import matplotlib.pyplot as plt  # For data visualization
import seaborn as sns            # For enhanced data visualization

# - Importing pandas as pd and numpy as np for data manipulation and numerical op
# - Importing matplotlib.pyplot as plt for basic plotting functionalities.
# - Importing seaborn as sns for enhanced data visualization, typically used to c
# - The use of aliases (pd, np, plt, sns) makes it easier to reference these libr
# - These libraries are commonly used in data analysis and visualization tasks in
```

```
# Importing the necessary function from the google.colab library
from google.colab import drive

# Mounting Google Drive to the Colab environment
drive.mount('/content/drive')

# - This code imports the `drive` function from the google.colab library.
# - The `drive.mount()` function is then called with the argument '/content/drive
# - This allows access to files stored in Google Drive within the Colab notebook
```

```
Mounted at /content/drive
```

Data Exploration

Download Dataset From Here

```
# Reading a CSV file using pandas and storing the data in a DataFrame named 'data
data = pd.read_csv('/content/drive/MyDrive/KNN_DataSet.csv')

# - This code uses the `pd.read_csv()` function from the pandas library to read a
# - The file path '/content/drive/MyDrive/KNN_DataSet.csv' specifies the location
# - The data from the CSV file is loaded into a pandas DataFrame named 'data'.
# - This assumes that the CSV file named 'KNN_DataSet.csv' is located in the spec
```

```
data.head()
```

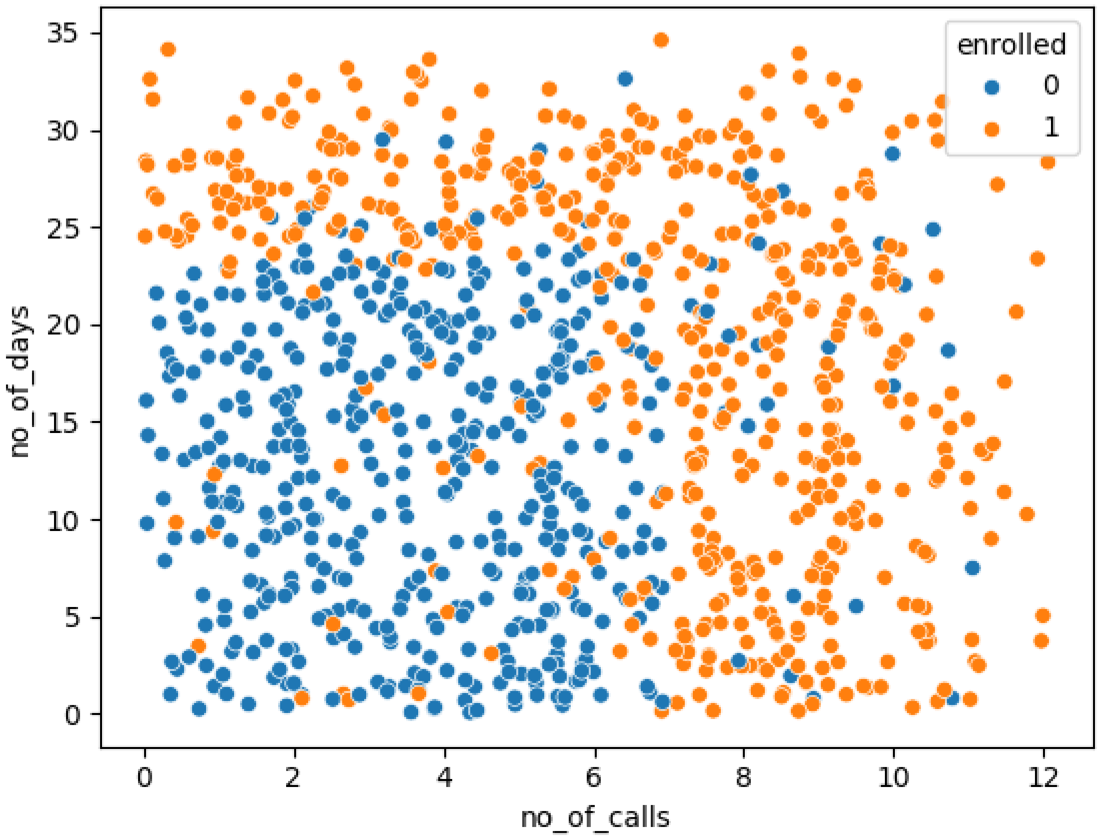
	user_id	gender	no_of_days	no_of_calls	enrolled	age
0	82238717	Male	24.502541	1.936580	1	69
1	75264736	Male	12.734882	1.448820	0	18
2	65851203	Female	15.594056	2.782494	0	39
3	50811194	Male	3.270349	4.791917	0	22
4	79334402	Male	15.335051	5.200419	0	45

Data Description:

- The dataset contains four columns, with "enrolled" being the target variable, representing a binary classification problem with values 0 or 1.
- Two variables of interest are "no_of_calls" and "no_of_days," with "gender" being ignored for now.

```
# Creating a scatter plot using seaborn library to visualize the relationship bet
# 'no_of_calls' and 'no_of_days' with respect to the target variable 'enrolled'

# sns.scatterplot() is used to create a scatter plot
# - 'data' parameter specifies the DataFrame containing the data to be plotted
# - 'x' parameter specifies the variable to be plotted on the x-axis, which is "n
# - 'y' parameter specifies the variable to be plotted on the y-axis, which is "n
# - 'hue' parameter specifies the variable used for coloring the points based on

sns.scatterplot(data=data, x="no_of_calls", y="no_of_days", hue="enrolled")

# - This code creates a scatter plot to visualize the relationship between the nu
#   the number of days ('no_of_days'), and whether a person is enrolled or not ('
# - Each point in the scatter plot represents an observation in the dataset.
# - The 'enrolled' variable is used to color the points, allowing for visual diff
# - This visualization can help in understanding any potential patterns or relati
```

```
<Axes: xlabel='no_of_calls', ylabel='no_of_days'>
```



Is there any linear boundary that can seprate these two classes?

```
# Importing the train_test_split function from the sklearn.model_selection module
from sklearn.model_selection import train_test_split


# Splitting the features ('no_of_calls' and 'no_of_days') and target ('enrolled')
# into training and testing sets with a test size of 33% and a random state of 42

X_train, X_test, y_train, y_test = train_test_split(data[['no_of_calls', 'no_of_d
                                                     data['enrolled'].values,
                                                     test_size=0.33, random_state=
```

Importance of Normalizing Data

- Precondition: Normalization is crucial, especially when using distance-based metrics in algorithms like KNN.
- Purpose: Normalizing data brings all variables to a common scale, ensuring that the contribution of each variable to the distance metric is fair and not skewed by its scale.
- Example: In KNN, where distance calculations are central to the algorithm, failure to normalize data can lead to inaccuracies in predictions.

```
# Calculating the mean (mu) and standard deviation (sigma) of the features in the
mu = X_train.mean(axis=0)
sigma = X_train.std(axis=0)


# Normalizing the features in the training and testing sets by subtracting the me

X_train = (X_train - mu)/sigma
X_test = (X_test - mu)/sigma
```

Calculating Distance

```
def dist(x1, x2):
    """Calculate the Euclidean distance between two vectors.

    Args:
        x1 (numpy.ndarray): First vector.
        x2 (numpy.ndarray): Second vector.

    Returns:
        float: Euclidean distance between the two input vectors.
    """
    return np.sqrt(np.sum((x1 - x2) ** 2))
```

Predicting the class label

```python
import numpy as np

def knn(X, Y, queryPoint, k=5):
    """Predict the class label for queryPoint using k-nearest neighbors algorithm

    Args:
        X (numpy.ndarray): Feature vectors of the training data.
        Y (numpy.ndarray): Labels of the training data.
        queryPoint (numpy.ndarray): Feature vector of the query point.
        k (int, optional): Number of nearest neighbors to consider. Defaults to 5

    Returns:
        int: Predicted class label for the query point.
    """
    distances = []
    m = X.shape[0]

    # Calculate distances from query point to each training point
    for i in range(m):
        d = dist(queryPoint, X[i])
        distances.append((d, Y[i]))  # Storing both distance and label

    # Sort the distances based on distance
    distances = sorted(distances)

    # Nearest/First k points
    distances = distances[:k]

    # Extract labels of the k nearest points
    k_nearest_labels = [label for _, label in distances]

    # Count the occurrences of each label
    unique_labels, label_counts = np.unique(k_nearest_labels, return_counts=True)

    # Find the label with the maximum count
    pred = unique_labels[np.argmax(label_counts)]

    return int(pred)
```

```python
X_test[100]
```

```
array([ 1.88929713, -1.63899491])
```

```python
# Predict the class label for the 100th test data point using the KNN algorithm
prediction = knn(X_train, y_train, X_test[100])
```

Let's see it is correct or not.

```python
y_test[100]
```

```
1
```

Scikit-learn Implementation of KNN

```
# Importing necessary modules from scikit-learn
from sklearn.neighbors import KNeighborsClassifier  # Importing K Neighbors Class
from sklearn.metrics import confusion_matrix, classification_report  # Importing
```

```
# Creating a K Neighbors Classifier object with default parameters
knn = KNeighborsClassifier()
# This initializes a KNeighborsClassifier object with default parameters:
# - n_neighbors: Number of neighbors to consider, default is 5
# - weights: Weight function used in prediction, default is 'uniform'
# - algorithm: Algorithm used to compute the nearest neighbors, default is 'auto'
# - leaf_size: Leaf size passed to BallTree or KDTree, default is 30
# - p: Power parameter for the Minkowski metric, default is 2
# - metric: Distance metric to use, default is 'minkowski'

# Fitting the K Neighbors Classifier model to the training data
knn.fit(X_train, y_train)
# Parameters:
# - X_train: Training input data
# - y_train: Target training labels
```

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

```
# Generating predictions using the trained K Neighbors Classifier model
y_pred = knn.predict(X_test)
# This line predicts the labels for the test input data X_test using the trained
# Parameters:
# - X_test: Test input data for which predictions are generated
# Returns:
# - y_pred: Predicted labels for the test input data
```

Confusion Matrix

```
# Generating the confusion matrix using the actual labels (y_test) and the predic
confusion_matrix(y_test, y_pred)
# This line calculates the confusion matrix which is a table used to evaluate the
# It compares the actual labels (y_test) with the predicted labels (y_pred) and c
# true negatives, and false negatives.
# Parameters:
# - y_test: Actual labels of the test data
# - y_pred: Predicted labels of the test data
# Returns:
# - Confusion matrix: A matrix showing the counts of true positives, false positi
```

```
array([[126,  29],
       [ 16, 159]])
```

Classification Report

```
# Printing the classification report based on the actual labels (y_test) and the
print(classification_report(y_test, y_pred))
# This line prints a classification report which includes precision, recall, F1-s
# as well as the average metrics across all classes.
# Parameters:
# - y_test: Actual labels of the test data
# - y_pred: Predicted labels of the test data
# Output:
# - Classification report: A text summary of the precision, recall, F1-score, and
#   the average metrics across all classes.
```

```
              precision    recall  f1-score   support

           0       0.89      0.81      0.85       155
           1       0.85      0.91      0.88       175

    accuracy                           0.86       330
   macro avg       0.87      0.86      0.86       330
weighted avg       0.87      0.86      0.86       330
```

Manhattan Distance

```
# Creating a KNeighborsClassifier object with Manhattan distance metric (p=1)
knn = KNeighborsClassifier(p=1)
# The 'p' parameter specifies the power parameter for the Minkowski metric. When
# This means that the algorithm will compute the distance between points using th
# (i.e., the Manhattan distance).
# This classifier will be trained using the training data (X_train and y_train).
knn.fit(X_train, y_train)
```

```
▾ KNeighborsClassifier
KNeighborsClassifier(p=1)
```

```
y_pred = knn.predict(X_test)
```

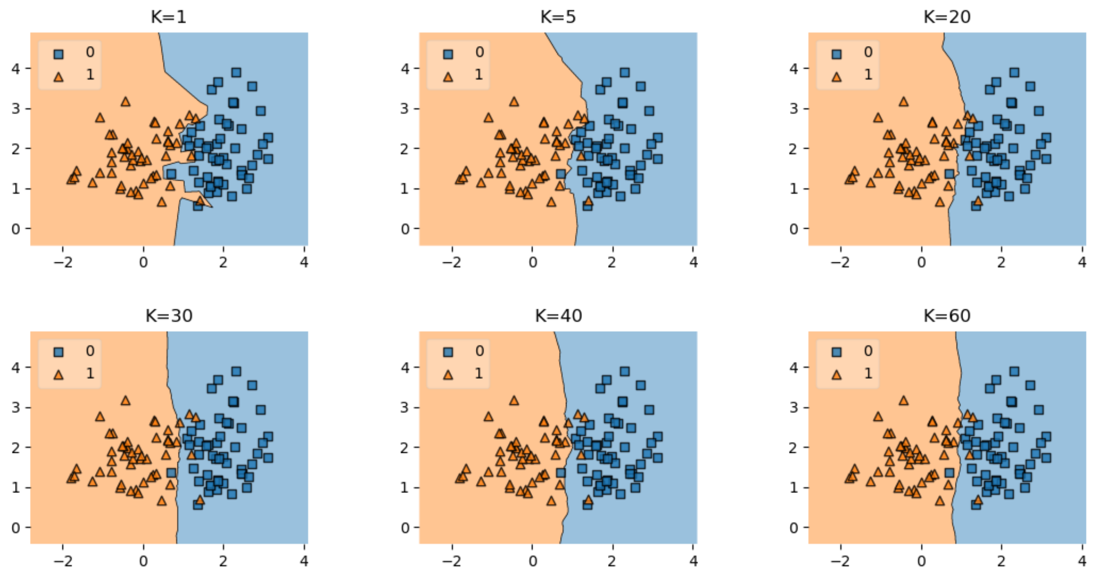
```
print("Classification Report with Manhatton Distance")
print(classification_report(y_test, y_pred))
```

```
Classification Report with Manhatton Distance
              precision    recall  f1-score   support

           0       0.88      0.84      0.86       155
           1       0.86      0.90      0.88       175

    accuracy                           0.87       330
   macro avg       0.87      0.87      0.87       330
```

```
    weighted avg       0.87      0.87      0.87       330
```

Parameter Tuning

Parameter tuning is a critical step in machine learning model development, where we adjust the hyperparameters of the model to optimize its performance. One such hyperparameter in the K-nearest neighbors (KNN) algorithm is the value of k, which represents the number of nearest neighbors to consider when making a prediction for a new data point. The impact of changing the k value on model performance can be explored through a process called hyperparameter tuning.



To systematically evaluate the effect of differentk values on model performance, we can use a loop to iterate over a range of values for k.

Here's how this process can be implemented:

1. Define a Range of k Values: Determine a range of values for k that you want to explore. This range can vary based on the specific dataset and problem you are working on. For example, we choose to explore values from 1 to 20.
2. Iterate Over Each k Value: Use a loop to iterate over each k value in the defined range. For each iteration:
    - Create a KNN classifier instance with the current value of k.
    - Fit the classifier to the training data.
    - Evaluate the classifier's performance on both the training and test datasets, typically using a performance metric such as accuracy.

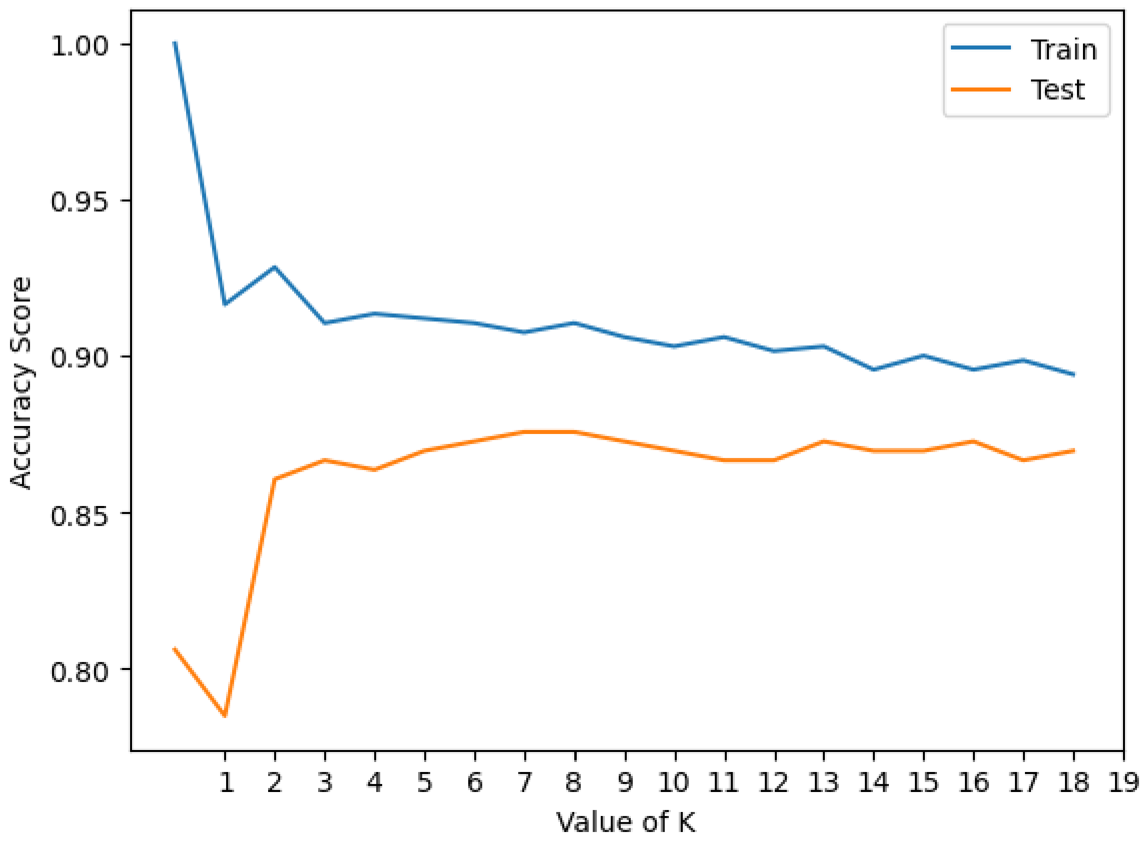
3. Store Performance Metrics: Keep track of the performance metrics (e.g., accuracy) obtained for each k value on both the training and test datasets. This allows you to compare the performance of the model across different k values.
4. Visualize the Results: Plot the performance metrics (e.g., accuracy) against the corresponding k values to visualize how the model's performance changes with different values of k. This visualization can help identify the optimal k value that yields the best performance on the test dataset without overfitting.

```
# Creating a KNeighborsClassifier object with k=1 (using only the nearest neighbo
knn = KNeighborsClassifier(n_neighbors=1)
# This classifier will be trained using the training data (X_train and y_train).
knn.fit(X_train, y_train)
# Using the trained classifier to predict the labels for the test data (X_test).
y_pred = knn.predict(X_test)

# Printing the classification report to evaluate the performance of the model
print("Classification Report with Manhattan Distance")
print(classification_report(y_test, y_pred))
```

```
Classification Report with Manhattan Distance
              precision    recall  f1-score   support

           0       0.80      0.79      0.79       155
           1       0.81      0.82      0.82       175

    accuracy                           0.81       330
   macro avg       0.81      0.80      0.81       330
weighted avg       0.81      0.81      0.81       330
```

```
# Initializing empty lists to store training and test accuracies for different va
train_accuracy = []
test_accuracy = []

# Looping through values of k from 1 to 19
for i in range(1, 20):
    # Creating a KNeighborsClassifier model with k=i
    kn = KNeighborsClassifier(n_neighbors=i)
    # Training the model on the training data (X_train and y_train)
    kn.fit(X_train, y_train)

    # Calculating the training accuracy and test accuracy for the current value o
    tr = kn.score(X_train, y_train)
    te = kn.score(X_test, y_test)

    # Appending the training accuracy and test accuracy to their respective lists
    train_accuracy.append(tr)
    test_accuracy.append(te)
```

```
# Importing necessary libraries for plotting
import matplotlib.pyplot as plt
import numpy as np

# Plotting the training and test accuracies against the values of k
plt.plot(train_accuracy, label="Train")  # Plotting training accuracy
plt.plot(test_accuracy, label="Test")    # Plotting test accuracy
plt.legend()                             # Adding legend to the plot
plt.xlabel("Value of K")                 # Adding label to x-axis
plt.ylabel("Accuracy Score")             # Adding label to y-axis
plt.xticks(np.arange(1, 20))             # Setting ticks on x-axis from 1 to 19
plt.show()                               # Displaying the plot
```



Effect of K on Decision Boundaries

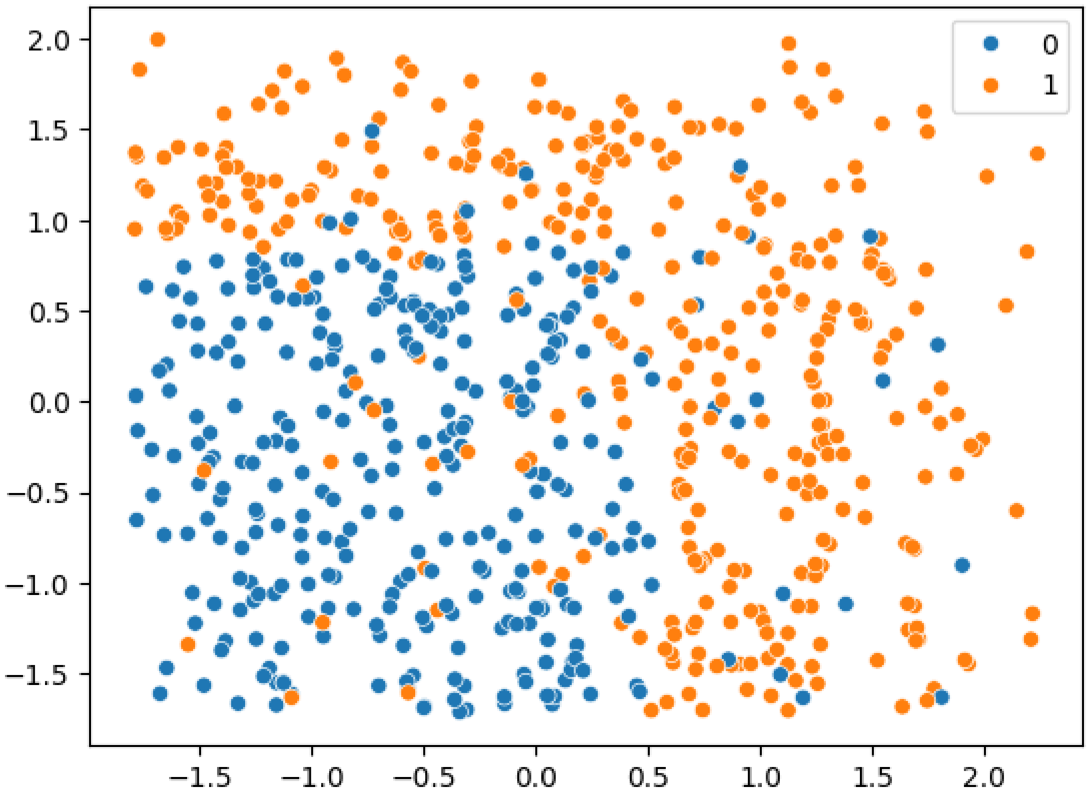
Visual Representation of Decision Boundaries for Different Values of K:

Decision boundaries in KNN represent the regions in the feature space where different classes are distinguished. Visualizing these boundaries helps understand how the algorithm classifies data points.

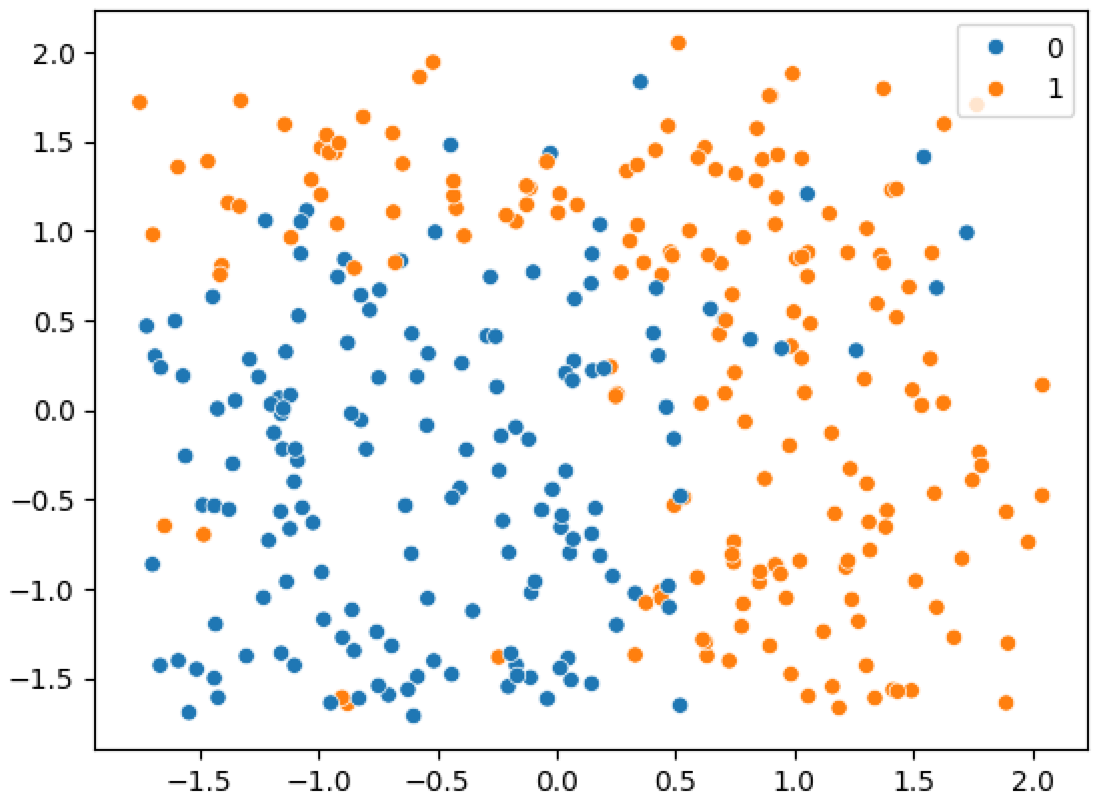
- Small K (e.g., K = 1): When K is small, such as K = 1, decision boundaries are highly flexible and adaptive to the training data. Each data point essentially acts as a decision boundary, leading to jagged and intricate boundaries. These boundaries are sensitive to even minor fluctuations or noise in the data. While this can result in accurate predictions for the training data, it often leads to poor generalization to unseen data, indicating overfitting.

- Increasing K: As K increases, decision boundaries become less sensitive to individual data points. The algorithm considers a larger number of nearest neighbors to make predictions, leading to smoother and more generalized boundaries. By incorporating more neighbors into the decision-making process, the model becomes less influenced by outliers or noisy data points. This reduction in sensitivity to individual data points typically improves the algorithm's ability to generalize to unseen data.
- Extremely High K (e.g., very large K): However, excessively high K values can lead to overly simplified decision boundaries. In extreme cases, when K is very large, decision boundaries may resemble linear boundaries or exhibit little variation across the feature space. This simplification can result in underfitting, where the model fails to capture the underlying complexity of the data. Instead, it assigns the majority class to large regions of the feature space, potentially leading to decreased predictive performance.

```
# Create a scatter plot of the training data
sns.scatterplot(x=X_train[:,0], y=X_train[:,1], hue=y_train)

# Show the plot
plt.show()
```



```
sns.scatterplot(x=X_test[:,0], y=X_test[:,1], hue=y_test)
plt.show()
```



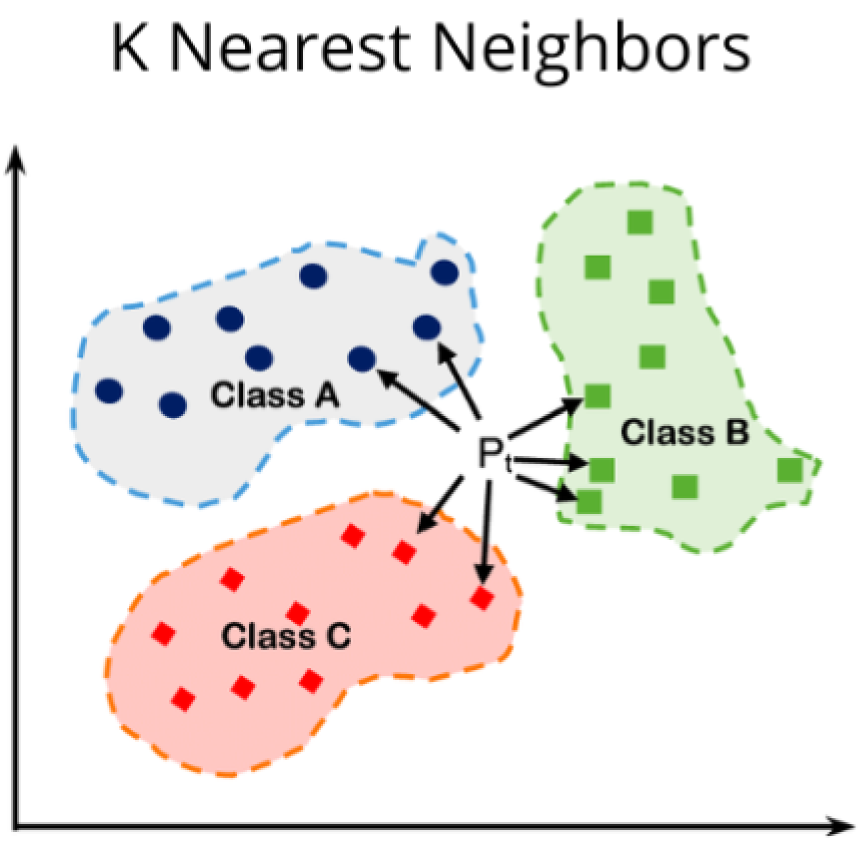
Multiclass Classification

Introduction to Multiclass Classification

Multiclass classification is a fundamental task in machine learning, where the objective is to predict the target variable from a set of three or more distinct classes. In contrast to binary classification, which deals with scenarios where the target variable has only two possible outcomes, multiclass classification tasks involve predicting among multiple categories or classes.

- Predicting Multiple Classes: In multiclass classification, the model's goal is to assign each instance to one of several possible classes. For example, in a handwritten digit recognition task, the model might need to classify an image of a digit as 0, 1, 2, ..., 9.
- Complexity Over Binary Classification: Multiclass classification tasks are inherently more complex than binary classification because there are more potential outcomes to consider. Algorithms designed for binary classification may not be directly applicable or optimal for multiclass problems.
- Requirement for Specialized Algorithms: Multiclass classification tasks necessitate algorithms capable of handling multiple classes simultaneously. While some binary classification algorithms can be extended or modified to accommodate multiple classes, others are specifically designed for multiclass scenarios.

- Decision Boundaries: In multiclass classification, the decision boundaries separating different classes can be more intricate compared to binary classification. The model needs to learn to distinguish between multiple classes while considering the potential overlap between them.
- Evaluation Metrics: Evaluation metrics used in multiclass classification differ from those used in binary classification. Common metrics include accuracy, precision, recall, F1-score, and confusion matrix analysis.
- Application Diversity: Multiclass classification finds applications in various domains, including image recognition, natural language processing, medical diagnosis, and sentiment analysis, among others. Many real-world problems inherently involve multiple classes, making multiclass classification a crucial component of machine learning applications.



Exploring Data

[Download Dataset From Here]

The dataset used as an example is the handwritten digits dataset, which is a classic benchmark dataset commonly employed in machine learning and pattern recognition tasks.

Here's a detailed introduction to the dataset:

1. Nature of the Dataset:

- The dataset consists of images of handwritten digits.
- Each image is represented as a 28x28 matrix of pixel values.
- The pixels of each image are flattened or collapsed into a single vector, resulting in 784 features (28 * 28) for each image.
- Each feature represents the intensity of a pixel, ranging from 0 to 255, where 0 indicates black and 255 indicates white.

2. Target Variable (Label):
    - The target variable or label associated with each image indicates the digit (0-9) represented by the handwritten image.
    - It's a multi-class classification problem with 10 classes, where each class corresponds to one of the digits from 0 to 9.
    - The goal is to build a machine learning model that can accurately predict the correct digit based on the input image.
3. Dataset Size and Composition:
    - The dataset typically contains a substantial number of images, with each digit represented by a sufficient number of samples.
    - The distribution of samples across different digits may vary, but the dataset is usually balanced to some extent to ensure fair representation of each class.
    - The dataset may be divided into training, validation, and test sets to facilitate model training, evaluation, and testing.
4. Application and Importance:
    - Handwritten digit recognition is a fundamental problem in pattern recognition and optical character recognition (OCR).
    - The dataset serves as a standard benchmark for evaluating the performance of various machine learning algorithms, including classifiers like K-Nearest Neighbors (KNN), support vector machines (SVM), neural networks, and more.
    - Applications of handwritten digit recognition include postal automation (such as ZIP code recognition), bank check processing, and digitizing historical documents.

The handwritten digits dataset provides a rich and diverse set of images for developing and evaluating machine learning algorithms, making it a widely used resource in the field of pattern recognition and machine learning.

```
# Loading the MNIST training dataset from a CSV file located at '/content/drive/M
mnist = pd.read_csv('/content/drive/MyDrive/mnist_train.csv')

# Displaying the shape of the MNIST dataset to understand its dimensions
mnist.shape
```

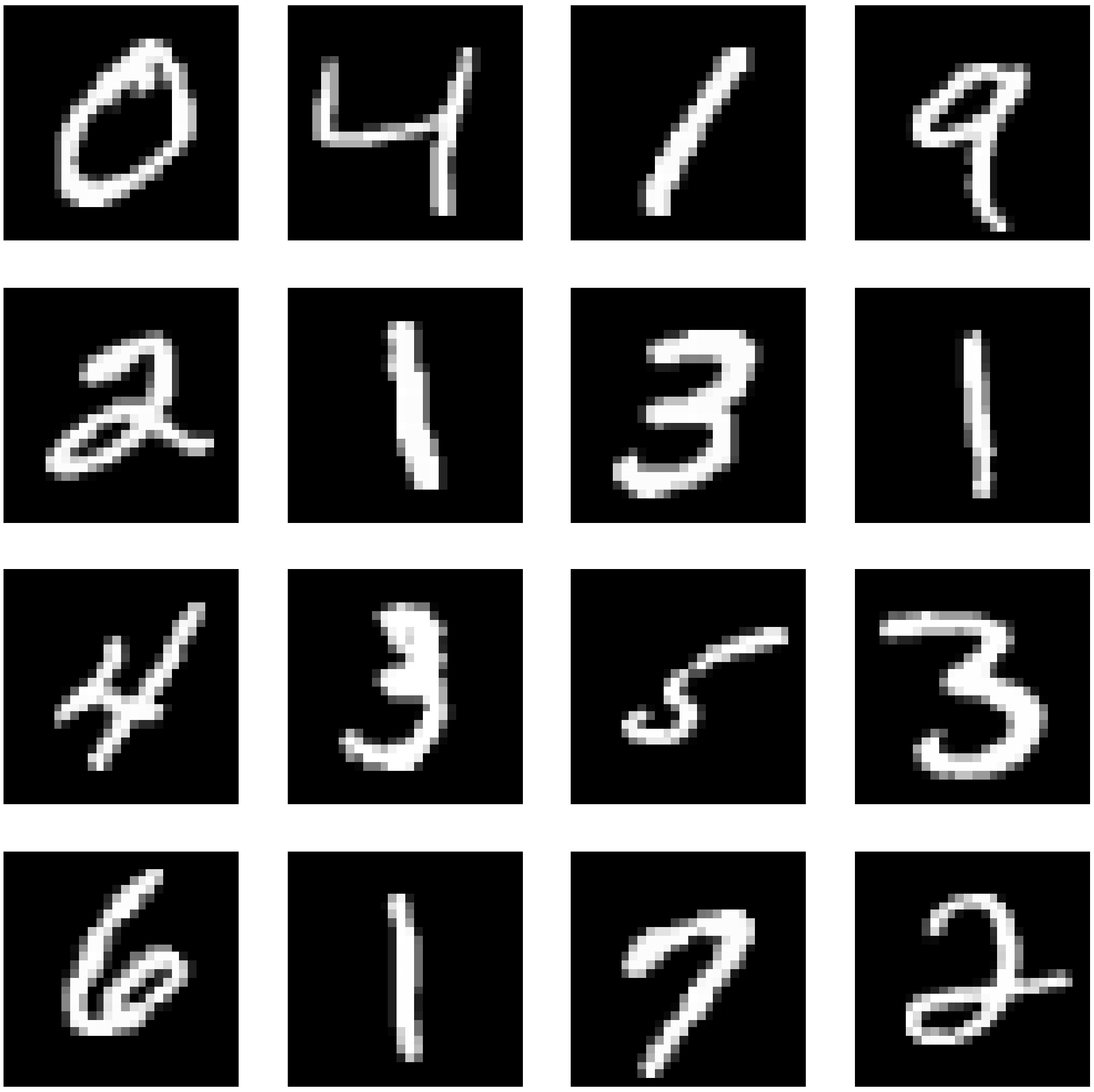
```
(60000, 785)
```

```
# Displaying the first few rows of the MNIST dataset to examine its structure and
mnist.head()
```

	label	1x1	1x2	1x3	1x4	1x5	1x6	1x7	1x8	1x9	...	28x19	28x20	28x21
0	5	0	0	0	0	0	0	0	0	0	...	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2	4	0	0	0	0	0	0	0	0	0	...	0	0	0
3	1	0	0	0	0	0	0	0	0	0	...	0	0	0
4	9	0	0	0	0	0	0	0	0	0	...	0	0	0

5 rows × 785 columns

```
# Separating the labels (target variable) from the MNIST dataset and storing them
mnist_labels = mnist['label']

# Extracting the pixel values (features) from the MNIST dataset and storing them
# Dropping the 'label' column to isolate the pixel values
mnist_data = mnist.drop(columns=['label'])


def plot_img(img):
    """
    Function to plot an image represented as a 1D array into a 28x28 grid.

    Parameters:
    - img: numpy array, 1D array representing the image pixels.

    Returns:
    - None
    """

    # Reshaping the 1D array into a 28x28 matrix to reconstruct the image
    # and plotting it using imshow() function from matplotlib.pyplot
    # cmap='gray' is used to display the image in grayscale
    plt.imshow(img.reshape(28, 28), cmap='gray')

    # Disabling the axis to remove the coordinate axes from the plot
    plt.axis(False)


plt.figure(figsize=(8, 8))  # Creating a new figure with a specified size of 8x8

# Looping through the range from 1 to 17 (exclusive) to create subplots
for i in range(1, 17):
    plt.subplot(4, 4, i)  # Creating a subplot grid with 4 rows and 4 columns, an

    # Calling the plot_img() function to plot the image corresponding to the i-th
    # mnist_data.iloc[i].values retrieves the pixel values of the i-th image as a
    plot_img(mnist_data.iloc[i].values)

# Displaying the plot with the subplots
```



Data Preprocessing

Principal Component Analysis (PCA)

- PCA is a technique used for dimensionality reduction, which helps in reducing the number of features while preserving most of the variability in the data.
- In this context, PCA is applied to the dataset to reduce the dimensionality of the feature space.
- The number of principal components chosen determines the dimensionality of the reduced feature space. Here, it's reduced to 2 principal components for visualization purposes.

```
# Importing the PCA class from the sklearn.decomposition module
from sklearn.decomposition import PCA


# Instantiating a PCA object with the number of components set to 4
pca = PCA(n_components=4)
```

- n_components specifies the number of principal components to retain after the dimensionality reduction process.
- In this case, PCA will retain 4 principal components from the original dataset.
- Setting n_components to a lower value reduces the dimensionality of the dataset, which can be beneficial for visualization, computational efficiency, and sometimes for improving model performance by reducing noise and focusing on the most important features.

```
# Applying PCA transformation to the mnist_data
X = pca.fit_transform(mnist_data)
```

- pca.fit_transform(mnist_data) fits the PCA model to the input data (mnist_data) and then transforms it into the new reduced-dimensional space.
- The transformed data X now contains the original data projected onto the principal components, where each row represents a sample and each column represents a principal component.
- The number of columns in X corresponds to the number of principal components specified earlier when creating the PCA object. In this case, it's 4.

```
# Printing the first 5 rows of the transformed data after PCA
print(X[:5])
```

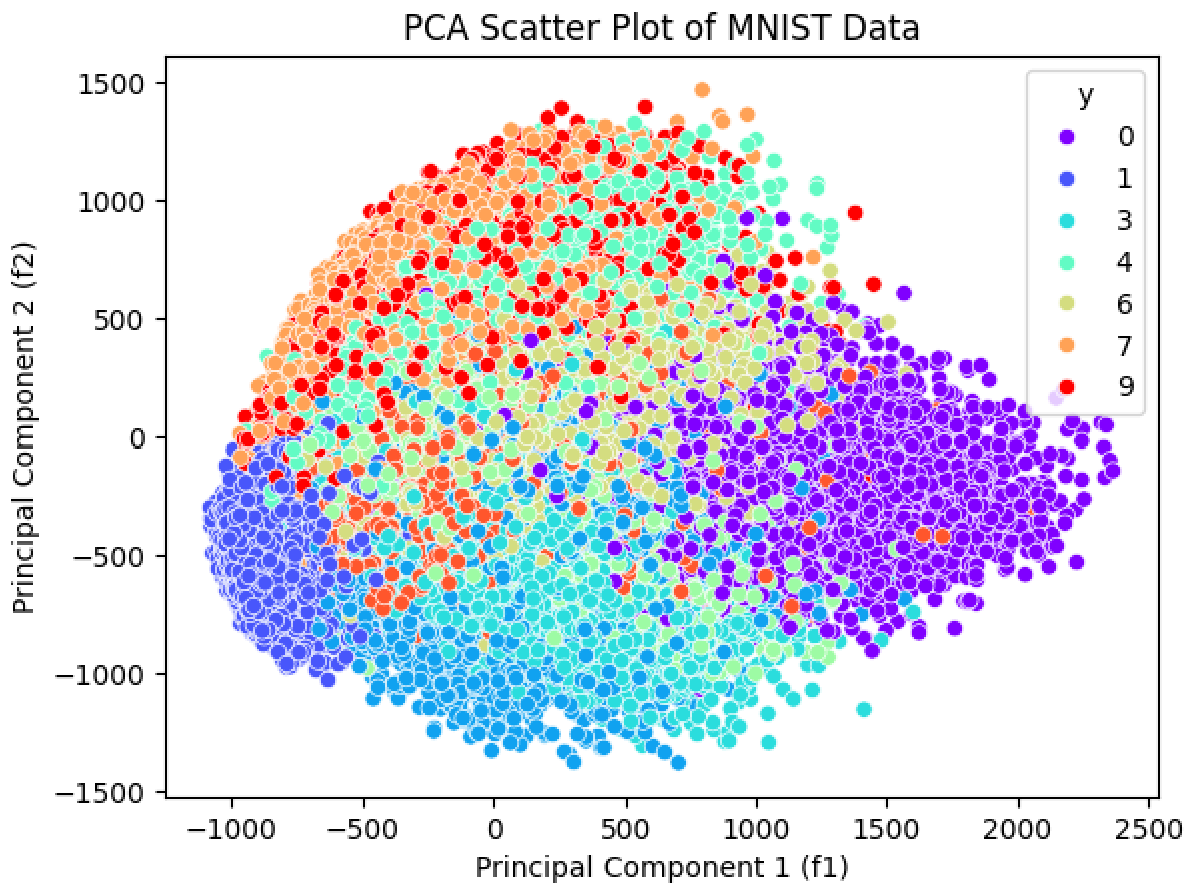
```
[[ 123.93294472 -312.67107248  -24.50114629 -555.84568135]
 [1011.71839335 -294.85680811  596.34107466 -460.787909  ]
 [ -51.84980262  392.17125759 -188.50756092  521.05665919]
 [-799.12717766 -607.19739848  273.66053659  105.93014851]
 [-382.75485484  730.54347329   16.33891697 -241.77006305]]
```

- Creating a DataFrame df with columns labeled as 'f1', 'f2', 'f3', and 'f4' to represent the principal components obtained from PCA.
- Adding a new column 'y' to the DataFrame, which contains the original labels (digit classes) from the mnist_labels variable.
- Displaying the first few rows of the DataFrame to verify the structure and content.

```
# Creating a DataFrame 'df' to store the transformed data along with the original
df = pd.DataFrame(X, columns=['f1', 'f2', 'f3', 'f4'])  # Creating DataFrame with
df['y'] = mnist_labels  # Adding the original labels as a column named 'y'

# Displaying the first few rows of the DataFrame
df.head()
```

	f1	f2	f3	f4	y
0	123.932945	-312.671072	-24.501146	-555.845681	5
1	1011.718393	-294.856808	596.341075	-460.787909	0
2	-51.849803	392.171258	-188.507561	521.056659	4
3	-799.127178	-607.197398	273.660537	105.930149	1
4	-382.754855	730.543473	16.338917	-241.770063	9

```
# Creating a scatter plot to visualize the data points in the reduced feature spa
sns.scatterplot(data=df, x='f1', y='f2', hue='y', palette='rainbow')

# Adding labels and title to the plot
plt.xlabel('Principal Component 1 (f1)')
plt.ylabel('Principal Component 2 (f2)')
plt.title('PCA Scatter Plot of MNIST Data')

# Displaying the plot
plt.show()
```



Generated a scatter plot using Seaborn to visualize the data points in the reduced feature space after PCA

Preprocessing and training K-Nearest Neighbors (KNN) classifier

Splitting the dataset into training and testing sets, followed by standardizing the features to ensure that each feature contributes equally to the distance computation in KNN.

```
# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df[['f1', 'f2', 'f3', 'f4']].
                                                    test_size=0.33, random_state=


# Standardizing the features
mu = X_train.mean(axis=0)
sigma = X_train.std(axis=0)

X_train = (X_train - mu)/sigma
X_test = (X_test - mu)/sigma


# Initializing the KNN classifier with 10 neighbors
knn2 = KNeighborsClassifier(n_neighbors=10)


# Fitting the classifier to the training data
knn2.fit(X_train, y_train)
```

```
▾        KNeighborsClassifier
KNeighborsClassifier(n_neighbors=10)
```

```
KNeighborsClassifier()
```

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

```
# Predicting the labels for the test data
y_pred = knn2.predict(X_test)
```

Confusion Matrix

```
# Calculating and printing the confusion matrix
confusion_matrix(y_test, y_pred)
```

```
array([[1665,    0,   55,   11,   13,   41,   82,    0,   75,    4],
       [   0, 2117,   16,    9,    2,   11,   20,    4,   34,    5],
       [  94,   17, 1267,   71,   33,   28,  377,    3,   47,    4],
       [  45,   19,   82, 1344,   14,  210,   24,   17,  260,   20],
       [   2,   15,   39,    3, 1193,   37,   82,  249,    5,  293],
       [ 129,   19,   34,  241,   43,  792,   61,   62,  384,   34],
       [  70,   23,  349,    8,   55,   44, 1391,    0,   40,    0],
       [   6,   63,   11,    6,  294,   44,   18, 1261,   32,  386],
       [ 160,   40,   40,  265,   32,  367,   52,   38,  873,   27],
       [  12,   11,    0,   27,  480,   55,   20,  579,   18,  746]])
```

The confusion matrix provides a detailed breakdown of the model's predictions versus the actual labels across all classes. Each row corresponds to the actual class, while each column represents the predicted class.

- Diagonal Elements (True Positives): The diagonal elements represent the number of instances where the predicted label matches the actual label. Higher values along the diagonal indicate accurate predictions.
- Off-diagonal Elements (False Positives and False Negatives): Off-diagonal elements represent misclassifications. For example, the element at row 1, column 3 indicates the number of instances where the actual label was class 1 but was predicted as class 3, representing a false positive. Similarly, the element at row 3, column 1 indicates the number of instances where the actual label was class 3 but was predicted as class 1, representing a false negative.
- Class Imbalance: The distribution of values across the matrix can reveal any class imbalances. Classes with disproportionately higher or lower counts may indicate potential biases in the dataset.

Classification Report

```
# Printing the classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.76      0.86      0.81      1946
           1       0.91      0.95      0.93      2218
           2       0.67      0.65      0.66      1941
           3       0.68      0.66      0.67      2035
           4       0.55      0.62      0.59      1918
           5       0.49      0.44      0.46      1799
           6       0.65      0.70      0.68      1980
           7       0.57      0.59      0.58      2121
           8       0.49      0.46      0.48      1894
           9       0.49      0.38      0.43      1948

    accuracy                           0.64     19800
   macro avg       0.63      0.63      0.63     19800
weighted avg       0.63      0.64      0.63     19800
```

The classification report provides a summary of various evaluation metrics for each class and overall performance.

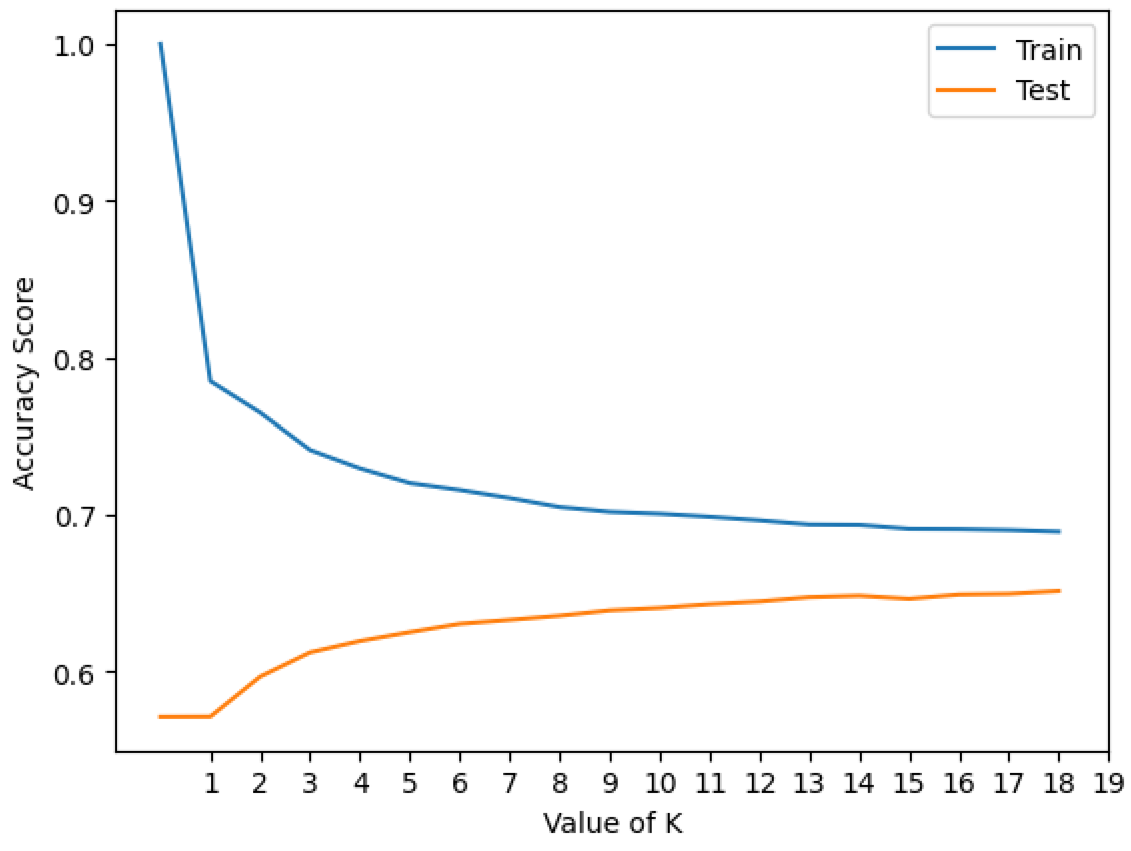
- Precision: Precision measures the accuracy of positive predictions. It is the ratio of true positives to the sum of true positives and false positives. Higher precision indicates fewer false positives.
- Recall (Sensitivity): Recall measures the ability of the classifier to correctly identify positive instances. It is the ratio of true positives to the sum of true positives and false negatives. Higher recall indicates fewer false negatives.
- F1-Score: The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall, with higher values indicating better model performance.
- Support: Support is the number of actual occurrences of each class in the test dataset. It provides insights into the distribution of classes and helps interpret the significance of precision, recall, and F1-score.

Overall Assessment:

- The classification report shows varying performance metrics across different classes. For instance, classes 1 and 7 have high precision and recall, indicating accurate predictions for these classes.
- Class 4 has relatively lower precision and recall compared to other classes, suggesting that the model struggles more with this class.
- The overall accuracy of the model is around 64%, indicating that the model performs reasonably well in predicting the classes but may benefit from further optimization.
- The macro-average and weighted-average F1-scores are both around 0.63, suggesting a balanced performance across classes, with slightly better performance for the macro-average.

Evaluating different values of K for KNN classifier

```
train_accuracy = []  # List to store training accuracy scores for different value
test_accuracy = []   # List to store testing accuracy scores for different values

# Loop through different values of K from 1 to 19
for i in range(1, 20):
    # Initialize KNN classifier with current value of K
    kn = KNeighborsClassifier(n_neighbors=i)

    # Fit the classifier to the training data
    kn.fit(X_train, y_train)

    # Calculate training and testing accuracy
    tr = kn.score(X_train, y_train)  # Training accuracy
    te = kn.score(X_test, y_test)     # Testing accuracy

    # Append training and testing accuracy scores to respective lists
    train_accuracy.append(tr)
    test_accuracy.append(te)



# Plotting the accuracy scores for different values of K
plt.plot(train_accuracy, label="Train")  # Plotting training accuracy scores
plt.plot(test_accuracy, label="Test")    # Plotting testing accuracy scores
plt.legend()                             # Displaying legend
plt.xlabel("Value of K")                  # Labeling x-axis
plt.ylabel("Accuracy Score")             # Labeling y-axis
plt.xticks(np.arange(1, 20))             # Setting x-axis ticks
plt.show()                               # Displaying the plot
```



- Selection of K
  - You can visually inspect the plot to determine the best value of K. Typically, a K value that yields the highest testing accuracy without significant overfitting is chosen. This is often done by observing where the testing accuracy curve peaks.
- Model Evaluation
  - The plot helps in evaluating the performance of the KNN classifier for different values of K. It provides insights into how the model generalizes to unseen data and whether adjustments need to be made to improve its performance.
- Generalization
  - A model that generalizes well to unseen data is desirable. Hence, the goal is to select a value of K that maximizes testing accuracy while ensuring that the model does not overfit the training data.

You can also perform cross-validation or use other evaluation metrics to further analyze the model's performance and ensure its robustness. Additionally, examining the variance and bias of the model can provide deeper insights into its behavior.

Imputation

- Imputation is a data preprocessing technique used to handle missing values in a dataset.
- It involves replacing missing values with estimated or calculated values to ensure completeness and consistency in the dataset.

Imputation Techniques

Mean Imputation

- Definition: Replace missing values with the mean (average) value of the variable.
- Procedure:
  - Calculate the mean of the variable.
  - Replace missing values with the calculated mean.
- Use Case:
  - Suitable for variables with a normal distribution where missing values are assumed to be average values.

Median Imputation

↳ 1 cell hidden

Zero Imputation

↳ 1 cell hidden

Placeholder Imputation

↳ 1 cell hidden

Advantages of Imputation Techniques

↳ 1 cell hidden

Limitations of Imputation Techniques

↳ 1 cell hidden

KNN Imputation

- KNN (K-Nearest Neighbors) imputation is a method used to handle missing values by estimating them based on the values of similar observations in the dataset.
- It involves identifying the nearest neighbors for observations with missing values and imputing the missing values based on the values of those neighbors.

Procedure

↳ 1 cell hidden

Advantages of KNN Imputation

↳ 1 cell hidden

Simulation

Download Dataset From Here

```
# Importing the KNNImputer class from the sklearn.impute module
from sklearn.impute import KNNImputer
```

```
# Reading the CSV file into a DataFrame
# The file path '/content/drive/MyDrive/loan-train.csv' specifies the location of
# The pd.read_csv() function is used to read the CSV file and create a DataFrame
# The DataFrame 'df' will contain the data from the CSV file
df = pd.read_csv('/content/drive/MyDrive/loan-train.csv')
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            614 non-null    object 
 1   Gender             601 non-null    object 
 2   Married            611 non-null    object 
 3   Dependents         599 non-null    object 
 4   Education          614 non-null    object 
 5   Self_Employed      582 non-null    object 
 6   ApplicantIncome    614 non-null    int64  
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object 
 12  Loan_Status        614 non-null    object 
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

- Loan_ID: Unique identifier for each loan application.
- Gender: Gender of the applicant (male/female). Contains 601 non-null values, indicating there are 13 missing values.
- Married: Marital status of the applicant (yes/no). Contains 611 non-null values, indicating there are 3 missing values.
- Dependents: Number of dependents on the applicant. Contains 599 non-null values, indicating there are 15 missing values.
- Education: Education level of the applicant (e.g., graduate, not graduate).
- Self_Employed: Indicates whether the applicant is self-employed (yes/no). Contains 582 non-null values, indicating there are 32 missing values.
- ApplicantIncome: Income of the applicant (in dollars). Contains 614 non-null values.
- CoapplicantIncome: Income of the co-applicant (if any) (in dollars). Contains 614 non-null