NumPy is primarily focused on numerical computing and provides support for large, multi-dimensional arrays and matrices.

Pandas is designed for data analysis and manipulation.

**Supervised Learning: Supervised learning is the most common type of machine learning, where the model is trained on labelled data. Labelled data means that each training example is paired with an output label (or target value), and the goal is to learn a mapping from input to output.**

**Classification: The output is a category (e.g., spam vs. non-spam emails, image recognition).**

**Regression: The output is a continuous value (e.g., predicting house prices based on features like size and location).**

**Examples:**

* **Image Classification: Given a set of images labelled as "cat" or "dog," the model learns to classify new images into one of these two categories.**
* **Spam Detection: Given emails labelled as "spam" or "not spam," the model learns to classify new emails into these categories.**

**Algorithm Examples:**

* **Linear Regression Support Vector Machines (SVM)**
* **Logistic Regression Decision Trees**

**Unsupervised Learning: Unsupervised learning, on the other hand, works with unlabelled data, where the model must find patterns, structures, or relationships in the data without explicit instructions on what the output should be. The model tries to learn the underlying structure of the data.**

**Clustering: Group similar data points together (e.g., customer segmentation).**

**Dimensionality Reduction: Reduce the number of features while retaining the most important information (e.g., PCA, t-SNE).**

**Examples:**

* **Customer Segmentation: Given a set of customer data (age, income, purchase history), the model groups customers into different clusters based on similarities.**
* **Market Basket Analysis: Finding patterns in product purchases, such as which items are often bought together.**

**Algorithm Examples:**

* **K-means Clustering**
* **Hierarchical Clustering**

**Classification**: Classification is a type of supervised learning where the goal is to predict a discrete label or category for each input. The output variable in classification is categorical, meaning it can take on one of a finite set of possible values (e.g., class labels).

**Spam Detection**: Given an email, classify it as "spam" or "not spam."

**Algorithms Used:**

* **Logistic Regression** **Decision Trees** **Support Vector Machines (SVM)**

**Regression**:is a **supervised learning** task where the goal is to predict a **continuous** numerical value based on input data.

**Example:**

* **House Price Prediction**: Given features like the size of a house, the number of bedrooms, and the location, predict the price of the house.
* **Stock Price Forecasting**: Predict the future price of a stock based on historical data.
* **Temperature Prediction**: Predict the temperature for the next day based on weather data.

**Algorithms Used:**

* **Linear Regression** **Support Vector Regression (SVR)**

**Clustering** is an **unsupervised learning** task where the goal is to group similar data points together into **clusters** based on their features. In clustering, the model doesn't have labelled data — it needs to find patterns or structures in the data on its own.

**Example:**

* **Customer Segmentation**: Grouping customers based on purchasing behavior to identify distinct market segments.
* **Document Clustering**: Grouping news articles or research papers by topic (e.g., sports, politics, health).

**Algorithms Used:**

* **k-Means Clustering** **Hierarchical Clustering**

**Confusion Matrix:**

A **Confusion Matrix** is a fundamental tool used for evaluating the performance of a classification model. It is particularly helpful for understanding how well your model is performing by comparing the predicted labels with the true labels (actual labels).

A **heatmap** is a data visualization technique that uses color to represent the intensity of data points in a dataset

The primary goal of using train\_test\_split() is to split the dataset into **training** and **testing** sets to evaluate machine learning models.

**Matplotlib** is like a "basic drawing tool" for plots. It's highly customizable but requires you to manually control all the details.

**Seaborn** is a "higher-level tool" built on top of Matplotlib. It is designed to make it easier to create **beautiful** and **statistical** plots with less code and nicer default settings.

import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5] y = [1, 4, 9, 16, 25]

# Create a line plot

plt.plot(x, y)

# Labeling the axes

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

# Title

plt.title('Simple Line Plot')

# Show the plot

plt.show()

sns.barplot(data=heart\_data,x='Age',y='AHD')

sns.countplot

sns.histplot

sns.scatterplot(data=heart\_data,x='RestBP',y='Chol',hue='AHD')

sns.boxplot

# Customize the plot using Matplotlib

plt.xlabel('Prediction', fontsize=13)

plt.ylabel('Actual', fontsize=13)

plt.title('Confusion Matrix', fontsize=17, pad=20)

# Show the plot

plt.show()

| **Feature** | **Matplotlib** | **Seaborn** |
| --- | --- | --- |
| **Type of Library** | Low-level, highly customizable visualizations | High-level, statistical plotting library built on top of Matplotlib |
| **Ease of Use** | Requires more code to create complex plots | Simplifies the creation of complex plots |
| **Customization** | Provides complete control over plot details | Limited control (but more aesthetic out of the box) |
| **Built-in Plots** | Basic plots like line, bar, scatter, histograms | Complex statistical plots like boxplots, violin plots, pair plots |

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import classification\_report

from sklearn.metrics import accuracy\_score

df.head()

df.tail()

df.shape

df.dtypes

df.columns

df.isnull().sum()

df.info()

mean\_ca=df['Ca']. mean()

df['Ca'].fillna(mean\_ca,inplace=True)

df[col]=df[col].fillna(df[col].mode()[0])

**inplace**: bool, default False Whether to modify the original DataFrame or return a new one.

(df==0).sum()

data=df[['Age','Sex','ChestPain','RestBP','Chol']]

train,test **=** train\_test\_split(data,test\_size**=**0.25,random\_state**=**40)

**test\_size**: The proportion of the dataset to be included in the test split.

**train\_size**: (optional) Proportion of data to use for the training set. If not specified, it is set to the complement of test\_size.

**random\_state**: (optional) Controls the shuffling of the data before splitting. Using a fixed random\_state ensures that the split is reproducible. If None, the data is shuffled randomly each time.

# Creating a confusion matrix

cm = confusion\_matrix(y\_true, y\_pred)

# Plotting the heatmap of the confusion matrix

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Negative', 'Positive'], yticklabels=['Negative', 'Positive'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix Heatmap')

plt.show()

* **Feature(s)**: These are the independent variables that are used as input to make predictions. Features are often the columns in your dataset that contain the information needed to predict the target. In this case, 'Glucose' is the feature you're selecting.
* **Target**: This is the dependent variable you are trying to predict. The target is usually a column in the dataset that you want the model to estimate or classify. In your case, 'Outcome' is the target variable, which could be a binary (0 or 1) value indicating some outcome (e.g., whether a patient has diabetes).

**Linear regression** is a statistical method used to model the relationship between two variables by fitting a linear equation (a straight line) to the observed data. It is widely used in data analysis, predictive modeling, and machine learning to understand how one variable (the dependent or response variable) changes as a function of another (the independent or predictor variable).

**Key Concepts:**

1. **Dependent Variable (Y):** The variable you are trying to predict or explain. For example, in a business context, this might be sales revenue.
2. **Independent Variable (X):** The variable that you believe has an effect on the dependent variable. For example, this could be advertising expenditure.
3. **Linear Relationship:** Linear regression assumes that the relationship between the dependent and independent variables can be approximated by a straight line. This relationship is often represented by the equation:

Y=β0+β1X+ϵY

**Types of Linear Regression:**

* **Simple Linear Regression:** Involves one independent variable (X) and one dependent variable (Y).
* **Multiple Linear Regression:** Involves two or more independent variables to predict the dependent variable. The equation becomes:

Y=β0+β1X1+β2X2+⋯+βn

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error,r2\_score

from sklearn.model\_selection import train\_test\_split

# Selecting 'Glucose' as the feature and 'Outcome' as the target

x = df[['Glucose']]   # Feature: this is what you're using to make predictions

y = df['Outcome']     # Target: this is what you're trying to predict

x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y,test\_size=0.2,random\_state=42);

model=LinearRegression()

model.fit(x,y)

# Coefficients

print(f"Coefficient (Slope): {model.coef\_[0]}")

print(f"Intercept: {model.intercept\_}")

# Residual sum of squares (RSS)

print(f"Residual Sum of Squares (RSS): {((y\_test - y\_pred) \*\* 2).sum()}")

# Coefficient of determination (R^2)

print(f"Coefficient of Determination (R^2): {r2\_score(y\_test, y\_pred)}")

import matplotlib.pyplot as plt

# Plot the actual data points

plt.scatter(x\_train, y\_train, color='blue', label='Actual Data')

# Plot the regression line (using the model's predictions)

plt.plot(x\_test, y\_pred, color='red', label='Regression Line')

plt.title("Newspaper advertising spending ")

plt.xlabel("Newspaper")

plt.ylabel("sales")

plt.show()

**Naive Bayes Classifiers**

every pair of features being classified is independent of each other. To start

Naive Bayes classifiers are a collection of classification algorithms based on [Bayes’ Theorem](https://www.geeksforgeeks.org/bayes-theorem/). It is not a single algorithm but a family of algorithms where all of them share a common principle. It is its speed, fast and making prediction is easy

Eg : spam filtering, sentiment detection, rating classification

It is because it assumes that one feature in the model is independent of existence of another feature. In other words, each feature contributes to the predictions with no relation between each other.

# Import necessary libraries

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

# Step 1: Load the dataset

df = pd.read\_csv('NaiveBayes.csv')

# Display the first few rows of the dataset

print(df.head())

# Step 2: Data Preprocessing

# Check for missing values

print(df.isnull().sum())

# Features (Age, Salary) and target variable (Purchased)

X = df[['Age', 'Salary']] # Features

y = df['Purchased'] # Target variable

# Step 3: Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Step 4: Create and train the Naïve Bayes classifier (Gaussian Naïve Bayes for continuous data)

nb\_classifier = GaussianNB()

# Train the classifier

nb\_classifier.fit(X\_train, y\_train)

# Step 5: Make predictions on the test set

y\_pred = nb\_classifier.predict(X\_test)

# Step 6: Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

class\_report = classification\_report(y\_test, y\_pred)

# Display evaluation results

print(f"Accuracy: {accuracy \* 100:.2f}%")

print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(class\_report)

**A Decision Tree is a popular and intuitive machine learning algorithm used for both classification and regression tasks. It works by splitting data into subsets based on feature values to make predictions.**

**Encoding Categorical Data**

* **Label Encoding**: is used to convert categorical labels into numerical values, and it's best suited for **ordinal** data where there's a meaningful order between categories.we convert them into numerical values (e.g., Male = 1, Female = 0).

**Key Steps to Apply a Decision Tree on the Dataset:**

1. **Load the dataset**
2. **Handle missing values** in the data
3. **Encode categorical variables** (e.g., Gender, Married, Education, etc.)
4. **Split the dataset** into features (X) and target (y)
5. **Split the data** into training and testing sets
6. **Train a Decision Tree model**
7. **Evaluate the model** using accuracy and other metrics
8. **Predict loan eligibility** on new/unseen data
9. from sklearn.preprocessing import LabelEncoder
10. label\_encoder=LabelEncoder()
11. df['Gender']=label\_encoder.fit\_transform(df['Gender'])
12. df['Married']=label\_encoder.fit\_transform(df['Married'])
13. df['Self\_Employed']=label\_encoder.fit\_transform(df['Self\_Employed'])
14. df['Education']=label\_encoder.fit\_transform(df['Education'])
15. df['Property\_Area']=label\_encoder.fit\_transform(df['Property\_Area'])
16. df['Loan\_Status']=label\_encoder.fit\_transform(df['Loan\_Status'])
17. df['Loan\_ID']=label\_encoder.fit\_transform(df['Loan\_ID'])
18. df['Dependents']=label\_encoder.fit\_transform(df['Dependents'])
19. df.head()

from sklearn.metrics import classification\_report,confusion\_matrix,accuracy\_score

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split

x=df.drop(columns=['Loan\_ID','Loan\_Status'])

y=df['Loan\_Status']

X\_train,X\_test,Y\_train,Y\_test=train\_test\_split(x,y,test\_size=0.2,random\_state=42)

model=DecisionTreeClassifier(random\_state=42)

model.fit(X\_train,Y\_train)

y\_pred=model.predict(X\_test)

print("Confusion Matrix: ", confusion\_matrix(Y\_test, y\_pred))

print ("Accuracy : ", accuracy\_score(Y\_test,y\_pred)\*100)

print("Report : ", classification\_report(Y\_test, y\_pred))

print("f1\_score: ", f1\_score(Y\_test, y\_pred))

print ("recall\_score : ", recall\_score(Y\_test,y\_pred))

**What is K-Means Clustering?**

**K-Means** is a type of **unsupervised machine learning algorithm** used for clustering data into groups based on similarity. The "K" in K-Means refers to the number of clusters you want to create. The algorithm divides the data into **K** clusters, where each data point belongs to the cluster whose center (called the centroid) is nearest.

**Key Concepts of K-Means:**

1. **Clustering**:
   * Clustering is the task of grouping a set of data points in such a way that data points in the same group (cluster) are more similar to each other than to those in other groups.
   * **Unsupervised learning**: The algorithm works on unlabeled data (data without a target variable).
2. **Centroids**:
   * Each cluster has a **centroid**, which is the mean of all the points in that cluster. This centroid represents the center of the cluster.
3. **Distance Metric**:
   * K-Means uses a distance metric (usually **Euclidean distance**) to determine how far apart data points are from each other and from the centroids.
4. **Iterations**:
   * K-Means works iteratively to refine the centroids and the clusters. It repeats the process until the centroids no longer change significantly or a specified number of iterations is reached.

**K-Means Algorithm: Step-by-Step Process**

Let's break down the steps of the K-Means algorithm:

**Step 1: Choose the number of clusters (K)**

You need to decide the value of **K** (the number of clusters you want). This can be done by analyzing the data or using methods like the **Elbow Method** to determine an optimal K.

**Step 2: Initialize centroids**

The algorithm randomly selects **K** points from the dataset to serve as the initial centroids (cluster centers).

**Step 3: Assign each data point to the nearest centroid**

Each data point is assigned to the cluster with the nearest centroid. The "nearest" is determined using a distance metric (commonly Euclidean distance).

**Step 4: Recalculate centroids**

Once all data points are assigned to clusters, the centroids are recalculated by taking the **mean** of all the data points in each cluster.

**Step 5: Repeat steps 3 and 4**

* Steps 3 and 4 are repeated until:
  + The centroids **no longer change** (i.e., convergence is reached), or
  + A predefined **number of iterations** is completed.

**Step 6: Final clusters**

Once convergence is achieved, the algorithm stops, and the final cluster assignments are returned, along with the corresponding centroids.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

data['total\_graduates\_scaled'] = scaler.fit\_transform(data[['total\_graduates']])

sse=[]

k\_range = range(1,11)

for k in k\_range:

    kmeans = KMeans(n\_clusters=k,random\_state = 42)

    kmeans.fit(data[['total\_graduates\_scaled']])

    sse.append(kmeans.inertia\_)

plt.figure(figsize=(10,5))

plt.plot(k\_range,sse,marker='o')

plt.title("Elbow method for optimal k")

plt.xlabel("Number of clusters (k)")

plt.ylabel("Sum of squared error (SSE)")

plt.show()

kmeans = KMeans(n\_clusters= 3 ,random\_state=42)

data['cluster'] = kmeans.fit\_predict(data[['total\_graduates\_scaled']])

plt.figure(figsize=(10, 5))

plt.scatter(data['total\_graduates'], data['cluster'], c=data['cluster'], cmap='viridis', label='Data points')

plt.title("K-means Clustering on total graduates")

plt.xlabel("Total Graduates")

plt.ylabel("Cluster")

# Ensure centers include both dimensions and apply inverse transformation if necessary.

(kmeans.cluster\_centers\_)

plt.legend()

plt.show()

**Hierarchical Clustering** is a type of unsupervised machine learning algorithm used to group similar data points into clusters, but unlike K-means clustering (which requires the number of clusters to be specified beforehand), hierarchical clustering builds a hierarchy of clusters and doesn’t need the number of clusters to be predefined.

**Key Concepts of Hierarchical Clustering**

1. **Agglomerative vs. Divisive Clustering:**
   * **Agglomerative (Bottom-up approach)**: This is the most common type of hierarchical clustering. In this method, each data point starts as its own individual cluster, and the algorithm progressively merges the closest pairs of clusters.
   * **Divisive (Top-down approach)**: The algorithm starts with all data points in one large cluster and recursively splits the data into smaller clusters.

In practice, **agglomerative clustering** is typically used because it is simpler and more efficient for most problems.

1. **Distance Metrics**:
   * Hierarchical clustering relies on the idea of **distance** between clusters (or data points) to decide which clusters to merge or split. Common distance metrics include:
     + **Euclidean distance**: The straight-line distance between two points.
     + **Manhattan distance**: The sum of the absolute differences of the coordinates.
     + **Cosine similarity**: Measures the cosine of the angle between two vectors (often used in text clustering).
2. **Linkage Criteria**:
   * The **linkage criterion** determines how the distance between two clusters is calculated. Common linkage methods include:
     + **Single linkage (nearest point)**: Distance between the closest points in the two clusters.
     + **Complete linkage (farthest point)**: Distance between the farthest points in the two clusters.
     + **Average linkage**: The average distance between all pairs of points in the two clusters.
     + **Ward's linkage**: Minimizes the variance of the merged clusters.

**How Hierarchical Clustering Works (Agglomerative Approach):**

1. **Initialization**:
   * Each data point is considered its own cluster. So, if you have n data points, you start with n clusters.
2. **Calculate Distance**:
   * Calculate the distance (similarity) between all pairs of clusters.
3. **Merge the Closest Clusters**:
   * Find the two clusters that are closest to each other (based on a distance metric like Euclidean distance) and merge them into a single cluster.
4. **Repeat**:
   * Recalculate the distances between the new cluster and the existing clusters, then merge the closest clusters.
   * This process is repeated until only one cluster remains, or a stopping criterion is met.
5. **Dendrogram**:
   * The result of hierarchical clustering is typically represented as a **dendrogram**—a tree-like diagram that shows how clusters are merged or split at each step. The dendrogram can be "cut" at a specific height to form a specific number of clusters.
6. xx=dff[['CRuns']]
7. xx\_scale=scale.fit\_transform(xx)
8. z=linkage(xx\_scale, method='ward', metric='euclidean')
9. dendrogram(z)
10. plt.show()

from scipy.cluster.hierarchy import dendrogram, linkage

from sklearn.cluster import AgglomerativeClustering

from sklearn.preprocessing import StandardScaler

data = df[['effective\_literacy\_rate\_total']]

# Step 3: Data Transformation (Scaling the data if necessary)

# Although 'effective\_literacy\_rate\_total' is already in a reasonable range, we can standardize it for better results

scaler = StandardScaler()

data\_scaled = scaler.fit\_transform(data)

# Step 4: Apply Hierarchical Clustering

# Use 'ward' linkage method which minimizes the variance within clusters

linked = linkage(data\_scaled, method='ward')

# Step 5: Plot Dendrogram

plt.figure(figsize=(10, 7))

dendrogram(linked)

plt.title('Dendrogram for Hierarchical Clustering')

plt.xlabel('Cities')

plt.ylabel('Euclidean Distance')

plt.show()

# Step 6: Apply Agglomerative Clustering to obtain the cluster labels

# We can decide the number of clusters from the dendrogram, let's assume 4 clusters based on the plot

agg\_clust = AgglomerativeClustering(n\_clusters=4, affinity='euclidean', linkage='ward')

df['Cluster'] = agg\_clust.fit\_predict(data\_scaled)

# Display the first few rows to see the clusters

print(df[['name\_of\_city', 'effective\_literacy\_rate\_total', 'Cluster']].head())

# Step 7: Visualize the Clusters

plt.figure(figsize=(10, 7))

plt.scatter(df['effective\_literacy\_rate\_total'], np.zeros\_like(df['effective\_literacy\_rate\_total']),

c=df['Cluster'], cmap='viridis')

plt.title('Hierarchical Clustering Result')

plt.xlabel('Effective Literacy Rate (Total)')

plt.ylabel('Cluster')

plt.show()

**Apriori algorithm**

**Step-by-Step Code Implementation:**

Below is a Python code implementation using **Pandas**, **mlxtend** (for the Apriori algorithm), and **Matplotlib** for visualization.

**Step 1: Import Required Libraries**

python

Copy code

import pandas as pd

import numpy as np

from mlxtend.frequent\_patterns import apriori, association\_rules

from mlxtend.preprocessing import TransactionEncoder

import matplotlib.pyplot as plt

import seaborn as sns

**Step 2: Data Preprocessing**

* We will load the dataset, group by Member\_number and Date, and aggregate the items purchased by each customer on each date.

python

Copy code

# Load the dataset

df = pd.read\_csv('Order1.csv')

# Check for missing values or duplicates and handle them if necessary

df = df.dropna()

# Group the items by Member\_number and Date, and aggregate the list of items purchased by each customer

transactions = df.groupby(['Member\_number', 'Date'])['itemDescription'].apply(list).reset\_index()

# Print out the first few rows to inspect the transaction format

print(transactions.head())

**Step 3: Convert Data into Transaction Format for Apriori**

* The Apriori algorithm needs data in a specific format. We will use the **TransactionEncoder** to convert the list of transactions into a one-hot encoded format (each item as a column).

python

Copy code

# TransactionEncoder is used to convert the list of transactions into one-hot encoded data

te = TransactionEncoder()

te\_ary = te.fit(transactions['itemDescription']).transform(transactions['itemDescription'])

# Convert to DataFrame for easy manipulation

df\_transformed = pd.DataFrame(te\_ary, columns=te.columns\_)

print(df\_transformed.head())

**Step 4: Apply Apriori Algorithm**

Now that we have the one-hot encoded transactions, we can apply the **Apriori** algorithm to find frequent itemsets. We will specify a minimum support threshold to identify frequent itemsets.

python

Copy code

# Apply Apriori Algorithm to find frequent itemsets

frequent\_itemsets = apriori(df\_transformed, min\_support=0.01, use\_colnames=True)

# Display the frequent itemsets

print(frequent\_itemsets.head())

**Step 5: Generate Association Rules**

Next, we will generate association rules from the frequent itemsets. We will use **confidence** as the metric and filter the rules based on a minimum confidence threshold.

python

Copy code

# Generate association rules from frequent itemsets

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.5)

# Display the rules

print(rules.head())

**Step 6: Visualize the Results**

You can visualize the association rules using a scatter plot or a bar chart to show relationships between items, confidence, and lift.

**Plotting the Support vs Confidence:**

python

Copy code

# Plotting support vs confidence for the rules

plt.figure(figsize=(10, 6))

sns.scatterplot(x='support', y='confidence', data=rules, hue='lift', size='lift', sizes=(50, 200))

plt.title("Support vs Confidence for Association Rules")

plt.xlabel("Support")

plt.ylabel("Confidence")

plt.legend()

plt.show()

**Visualize the Top Rules:**

You can also visualize the top rules sorted by the lift value.

python

Copy code

# Sort rules by lift

sorted\_rules = rules.sort\_values(by='lift', ascending=False)

# Show top 10 rules

print(sorted\_rules.head(10))

**Visualize the Frequency of Itemsets:**

To understand which items are frequently purchased together, you can plot the frequent itemsets.

python

Copy code

# Plotting top frequent itemsets based on support

frequent\_itemsets['length'] = frequent\_itemsets['itemsets'].apply(lambda x: len(x))

plt.figure(figsize=(10,6))

sns.barplot(x="support", y="itemsets", data=frequent\_itemsets[frequent\_itemsets['length'] == 2], palette="Blues\_d")

plt.title("Top 2-item Frequent Itemsets")

plt.xlabel("Support")

plt.ylabel("Itemsets")

plt.show()

**Explanation of the Code:**

1. **Data Preprocessing**:
   * The dataset is grouped by Member\_number (customer) and Date (purchase date) so that each entry represents a unique transaction (list of items purchased).
2. **Transaction Encoding**:
   * The **TransactionEncoder** from mlxtend is used to convert the list of items purchased into a format suitable for the Apriori algorithm. It uses a one-hot encoding approach, where each item corresponds to a column, and if a customer bought that item, the column will have a value of 1 (otherwise 0).
3. **Apriori Algorithm**:
   * The **Apriori** function is applied to find frequent itemsets with a specified **minimum support**. This will identify the items that occur frequently together.
4. **Association Rules**:
   * The **association\_rules** function is used to generate rules based on the frequent itemsets, using **confidence** as the metric. The rules with a confidence value greater than 0.5 are selected, but you can adjust this threshold.
5. **Visualization**:
   * A scatter plot is used to visualize the relationship between support and confidence of the association rules, with lift as a third dimension (size of the points).
   * Additionally, the top frequent itemsets are visualized to show which combinations of items appear frequently in the dataset.

**Step 1.1:** **Data Preprocessing** — Convert the dataset into a list of transactions. Each transaction is a set of items purchased together. This step often involves:

 **Step 2.1:** **Generate 1-itemsets**: Start by considering each item individually as a potential itemset.**Count the frequency** (support) of each item in the dataset.

 **Step 2.3:** Prune any itemsets that do not meet the minimum **support threshold** (i.e., support must be greater than or equal to the minimum support value).

**3. Generate Association Rules:**

* **Step 3.1:** For each frequent itemset, generate possible **association rules**.
  + An association rule is of the form A→BA \rightarrow BA→B, where AAA and BBB are itemsets.
* **Step 3.2:** For each rule, calculate the **confidence**. If the confidence of the rule is greater than or equal to the minimum confidence threshold, retain the rule.
* **Step 3.3:** Optionally, calculate the **lift** of the rule to assess how strongly the items are associated. If lift > 1, the items are positively correlated.

**4. Filter and Select the Best Rules:**

* **Step 4.1:** Filter rules based on metrics like confidence and lift.

**What is SVM (Support Vector Machine)?**

**Support Vector Machine (SVM)** is a supervised machine learning algorithm primarily used for classification tasks but can also be used for regression. SVM works by finding a hyperplane that best separates the data points into different classes in the feature space. It is particularly powerful in high-dimensional spaces and for datasets that have clear margins of separation.

**Core Concepts of SVM**

1. **Hyperplane**: A hyperplane is a decision boundary that separates different classes. In two-dimensional space, a hyperplane is simply a line. In three-dimensional space, it’s a plane. In higher dimensions, it's a "hyperplane." SVM’s goal is to find the hyperplane that maximizes the margin between different classes.
2. **Margin**: The margin is the distance between the hyperplane and the nearest data points from any class. The larger the margin, the better the generalization capability of the classifier. SVM aims to maximize this margin.
3. **Support Vectors**: These are the data points that are closest to the hyperplane. These points are critical because they influence the position and orientation of the hyperplane. The algorithm is focused on these support vectors when determining the optimal decision boundary.
4. **Kernel Trick**: Sometimes data is not linearly separable in its original space. In these cases, SVM uses the **kernel trick** to map the data into a higher-dimensional space where a linear hyperplane can separate the classes. Common kernel functions include:
   * **Linear Kernel**: No mapping, used when data is already linearly separable.
   * **Polynomial Kernel**: Maps the data into a higher-dimensional space based on polynomial functions.
   * **Radial Basis Function (RBF) Kernel**: A Gaussian kernel that maps data into a higher-dimensional space.
   * **Sigmoid Kernel**: Similar to a neural network activation function.
5. **Types of SVM**:
   * **SVM for Classification**: This is the most common use of SVM. The algorithm creates a hyperplane that separates different classes.
   * **SVM for Regression (SVR)**: This is a version of SVM used for continuous data prediction. It tries to find a function that approximates the underlying relationship in the data while staying as flat as possible.