```
# Apriori Algrithm

from itertools import combinations

# Sample dataset
transactions = [
    ['milk', 'bread', 'butter'],
    ['bread', 'butter'],
    ['milk', 'bread'],
    ['milk', 'butter'],
    ['bread']
]

# Set minimum support and confidence
min_support = 0.4  # At least in 40% of transactions
min_confidence = 0.6  # Confidence threshold

# Step 1: Create a list of all items
items = set(item for transaction in transactions for item in transaction)

# Step 2: Generate all item combinations and count support
def get_frequent_itemsets(transactions, items, k):
    candidate_counts = {}
    for transaction in transactions:
        for combo in combinations(sorted(set(transaction)), k):
            combo = tuple(combo)
            candidate_counts[combo] = candidate_counts.get(combo, 0) + 1
    # Filter by min_support
    num_transactions = len(transactions)
    frequent = {item: count for item, count in candidate_counts.items()
                if count / num_transactions >= min_support}
```

```
    return frequent

# Step 3: Generate frequent itemsets
frequent_itemsets = {}
k = 1
while True:
    frequent_k = get_frequent_itemsets(transactions, items, k)
    if not frequent_k:
        break
    frequent_itemsets.update(frequent_k)
    k += 1

# Step 4: Generate association rules
print("Frequent Itemsets:")
for itemset, count in frequent_itemsets.items():
    print(f"{itemset}: support = {count/len(transactions):.2f}")

print("\nAssociation Rules:")
for itemset in frequent_itemsets:
    if len(itemset) < 2:
        continue
    for i in range(1, len(itemset)):
        for A in combinations(itemset, i):
            A = set(A)
            B = set(itemset) - A
            A = tuple(sorted(A))
            B = tuple(sorted(B))
            support_AB = frequent_itemsets[itemset] / len(transactions)
            support_A = frequent_itemsets.get(A, 0) / len(transactions)
            if support_A == 0:
                continue
```

```
confidence = support_AB / support_A
if confidence >= min_confidence:
    print(f"{A} => {B} (conf = {confidence:.2f}, supp = {support_AB:.2f})")
```

1 . Build a spam filter using Python and the Naive Bayes algorithm.

```
# Import necessary libraries
import pandas as pd                    # For data handling
from sklearn.model_selection import train_test_split  # To split dataset
from sklearn.feature_extraction.text import CountVectorizer  # To convert text to numbers
from sklearn.naive_bayes import MultinomialNB  # Naive Bayes model
from sklearn.metrics import accuracy_score, classification_report  # For evaluation

# Step 1: Load dataset
# Dataset format: First column is label (spam/ham), second is message
url = "https://raw.githubusercontent.com/justmarkham/pycon-2016-tutorial/master/data/sms.tsv"
df = pd.read_csv(url, sep='\t', header=None, names=['label', 'message'])

# Step 2: Convert labels to binary (spam=1, ham=0)
df['label'] = df['label'].map({'ham': 0, 'spam': 1})

# Step 3: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
   df['message'],     # Input features (text)
   df['label'],       # Output labels
   test_size=0.2,     # 80% training, 20% testing
   random_state=42    # For reproducibility
)

# Step 4: Convert text into numerical feature vectors using Bag of Words
vectorizer = CountVectorizer()
X_train_vec = vectorizer.fit_transform(X_train)  # Learn vocab and transform train data
X_test_vec = vectorizer.transform(X_test)        # Transform test data with same vocab
```

```python
# Step 5: Train the Naive Bayes classifier
model = MultinomialNB()
model.fit(X_train_vec, y_train)  # Train the model with vectorized text

# Step 6: Make predictions on the test set
y_pred = model.predict(X_test_vec)

# Step 7: Evaluate model performance
print("Accuracy:", accuracy_score(y_test, y_pred))  # % of correct predictions
print("\nClassification Report:")
print(classification_report(y_test, y_pred))  # Precision, Recall, F1-score
```

2.Classify DDoS attacks with Artificial Intelligence.

```
# Import libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
file_path = 'c396cf2f-f304-41c0-b82a-b5a7d7965afa.csv'
df = pd.read_csv(file_path)

# Drop non-numeric and identifier columns
df = df.drop(columns=['Flow ID', 'Src IP', 'Dst IP', 'Timestamp'], errors='ignore')

# Drop missing values
df = df.dropna()

# Convert all labels to binary: Normal vs Attack
df['Label'] = df['Label'].apply(lambda x: 'Normal' if str(x).lower() == 'normal' else 'Attack')

# Encode target label: Normal=0, Attack=1
label_encoder = LabelEncoder()
df['Label'] = label_encoder.fit_transform(df['Label'])

# Separate features and labels
```

```
X = df.drop(columns=['Label'])
y = df['Label']

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split dataset (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.3, random_state=42, stratify=y
)

# Train Random Forest model
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)

# Predict on test set
y_pred = clf.predict(X_test)

# Print classification report
print("=== Classification Report ===")
print(classification_report(y_test, y_pred, target_names=['Normal', 'Attack']))

# Plot confusion matrix
cm = confusion_matrix(y_test, y_pred)
labels = ['Normal', 'Attack']

plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
```

```
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.tight_layout()
plt.show()

# Plot top 10 feature importances
importances = clf.feature_importances_
indices = np.argsort(importances)[-10:][::-1]
feature_names = X.columns

plt.figure(figsize=(8, 5))
sns.barplot(x=importances[indices], y=[feature_names[i] for i in indices])
plt.title('Top 10 Important Features')
plt.xlabel('Feature Importance Score')
plt.tight_layout()
plt.show()
```

3. Split sample data into training and test sets. (Use suitable data set).

```
# Step 1: Import required libraries
from sklearn.datasets import load_iris         # Load sample dataset
from sklearn.model_selection import train_test_split  # For splitting data
import pandas as pd

# Step 2: Load the Iris dataset
iris = load_iris()

# Convert to a pandas DataFrame for better understanding
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target

# Optional: View first few rows of the dataset
print("Sample Data:")
print(df.head())

# Step 3: Define features (X) and target (y)
X = df.drop(columns=['target'])  # Features
y = df['target']              # Labels

# Step 4: Split data into training and test sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# Step 5: Print the results
print("\nTraining Set:")
```

```
print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")

print("\nTest Set:")
print(f"X_test shape: {X_test.shape}")
print(f"y_test shape: {y_test.shape}")
```

4: Perform feature engineering operations on raw data. (Use suitable data set).

```
# Step 1: Import libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Step 2: Load Titanic dataset from seaborn
import seaborn as sns
df = sns.load_dataset('titanic')

# Step 3: View raw data
print("Raw data sample:")
print(df.head())

# Step 4: Drop irrelevant columns
df.drop(columns=['deck', 'embark_town', 'alive', 'who', 'adult_male', 'class'], inplace=True)

# Step 5: Handle missing values
df['age'].fillna(df['age'].median(), inplace=True)
df['embarked'].fillna(df['embarked'].mode()[0], inplace=True)

# Step 6: Encode categorical features
label_encoders = {}
categorical_cols = ['sex', 'embarked', 'embarked', 'alone']
for col in categorical_cols:
   le = LabelEncoder()
   df[col] = le.fit_transform(df[col])
```

```
    label_encoders[col] = le  # Save encoders in case we need to decode later

# Step 7: Create new features
df['family_size'] = df['sibsp'] + df['parch'] + 1  # 1 for self
df['is_child'] = df['age'].apply(lambda x: 1 if x < 16 else 0)  # Binary child indicator

# Step 8: Final dataset after feature engineering
print("\nAfter Feature Engineering:")
print(df.head())

# Step 9: Define X and y
X = df.drop(columns=['survived'])  # Features
y = df['survived']                 # Target

# Step 10: Split into train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

print("\nTrain/Test shapes:")
print(f"X_train: {X_train.shape}, y_train: {y_train.shape}")
print(f"X_test: {X_test.shape}, y_test: {y_test.shape}")
```