**NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY**



**Data Mining**

**COCSC16**

**Submitted By:**

**Laksh Sachdeva**

**2022UCS1572**

**CSE-1**

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| **S.No.** | **EXPERIMENT** |
| 1 | Exploratory Data Analysis using numpy and pandas in python |
| 2 | Exploratory Data Analysis using Weka |
| 3 | Data Preprocessing and Data Warehouse Operations |
| 4 | Implement Linear Regression from scratch |
| 5 | Naïve Bayes classifier and Bayesian network |
| 6 | Implementing decision trees in python |
| 7 | Forming strong association rules in python. |

**EXPERIMENT 1: Exploratory Data Analysis using NumPy and Pandas in Python**

**Dataset used – data.csv -> Duration , Pulse, Maxpulse , Calories**

Step 1: Load a CSV file into a Pandas DataFrame. Display the first five rows, check for missing values, and generate summary statistics for the numerical columns.

import pandas as pd

import matplotlib.pyplot as plt

# check for missing values, and generate summary statistics for the numerical columns.

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

print("First five rows of the DataFrame:->")

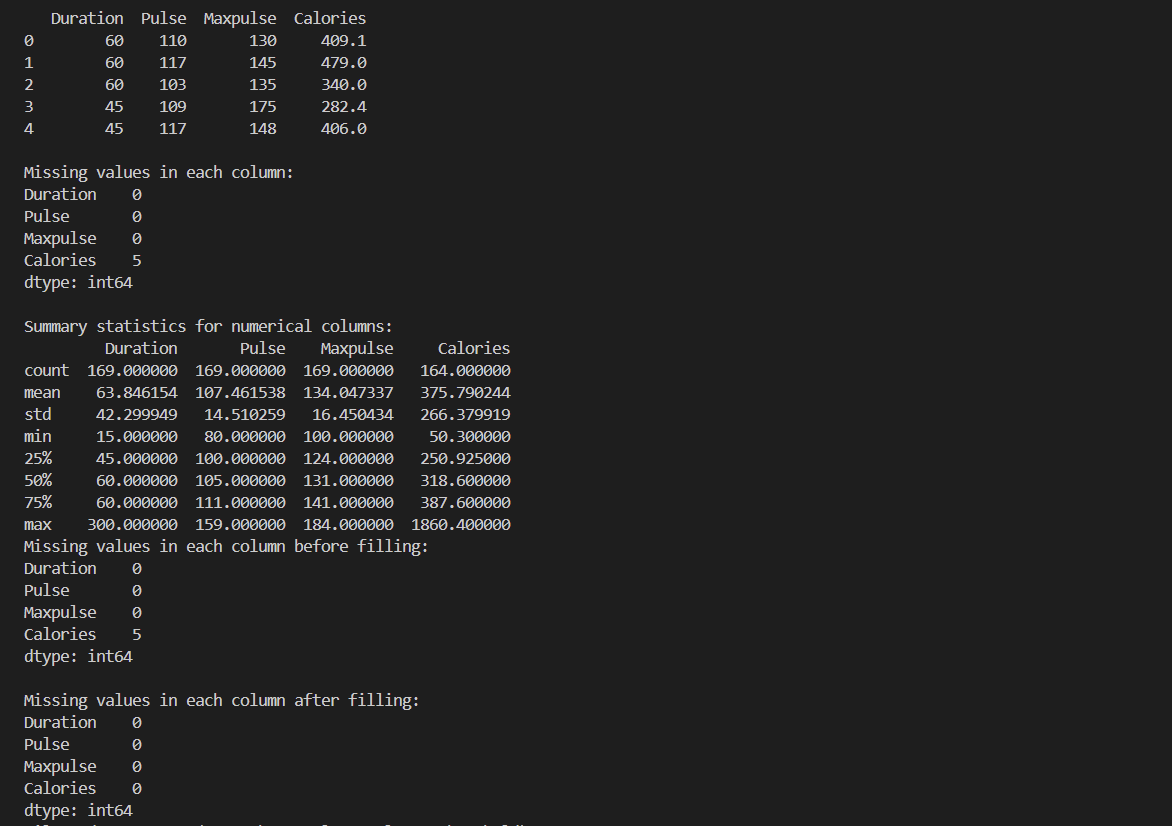
print(df.head())

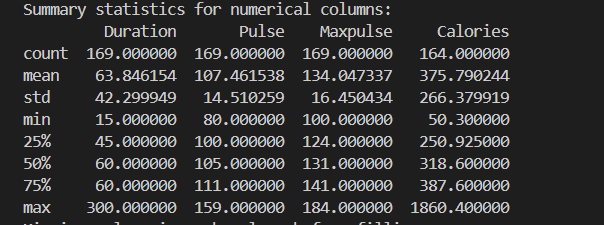
print("\nMissing values in each column:")

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

print("\nSummary statistics for numerical columns:")

print(df.describe())

****

****

Step 2: Identify missing values in the dataset and fill them with the median value of their respective columns.

missing\_values = df.isnull().sum()

print("Missing values in each column before filling:")

print(missing\_values)

# Compute the median for each column with missing values

medians = df.median()

# Fill missing values with the median of their respective columns

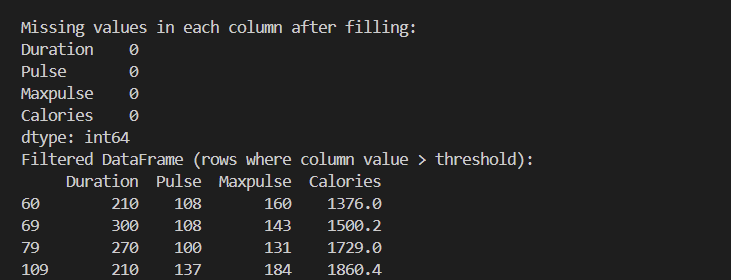
df\_filled = df.fillna(medians)

# Verify that there are no missing values left

missing\_values\_after = df\_filled.isnull().sum()

print("\nMissing values in each column after filling:")

print(missing\_values\_after)



Step 3: Filter the dataset to include only rows where a specific numerical column's value is greater than a given threshold. Display the first five rows of the filtered DataFrame.

# Specify the column and the threshold value

column\_name = 'Duration'  # Replace with your actual column name

threshold = 200  # Replace with your actual threshold value

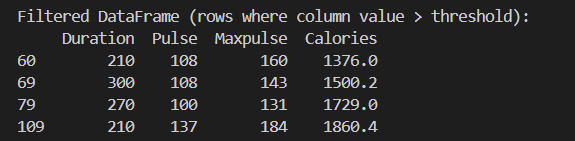
# Filter the DataFrame based on the condition

filtered\_df = df[df[column\_name] > threshold]

# Display the first five rows of the filtered DataFrame

print("Filtered DataFrame (rows where column value > threshold):")

print(filtered\_df.head())



Step 4: Group the dataset by a categorical column and calculate the mean and standard deviation of numerical columns for each group. Display the results.

# Group the dataset by a categorical column and calculate the mean and standard deviation of numerical columns for each group. Display the results.

# Specify the categorical column to group by

categorical\_column = 'Duration'  # Replace with your actual categorical column name

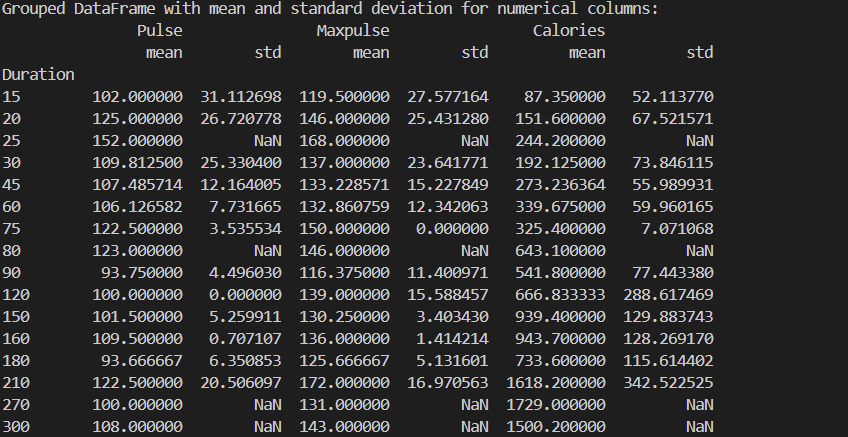
# Group by the categorical column and calculate mean and standard deviation

grouped\_df = df.groupby(categorical\_column).agg(['mean', 'std'])

# Display the results

print("Grouped DataFrame with mean and standard deviation for numerical columns:")

print(grouped\_df)

****

Step 5:  Create a new column in the DataFrame based on a calculation involving existing columns. Display the first five rows of the updated DataFrame.# Let's split the DataFrame into two based on a criterion

df1 = df[df['Duration'] <= 60]

df2 = df[df['Duration'] > 60]

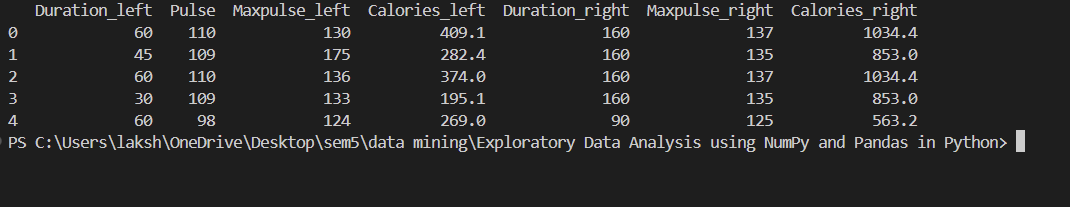
# For demonstration, let's merge df1 and df2 on a common column, here 'Pulse'

# Note: This is just an example; 'Pulse' might not be a good merge key for real use cases

merged\_df = pd.merge(df1, df2, on='Pulse', suffixes=('\_left', '\_right'))

# Display the first five rows of the merged DataFrame

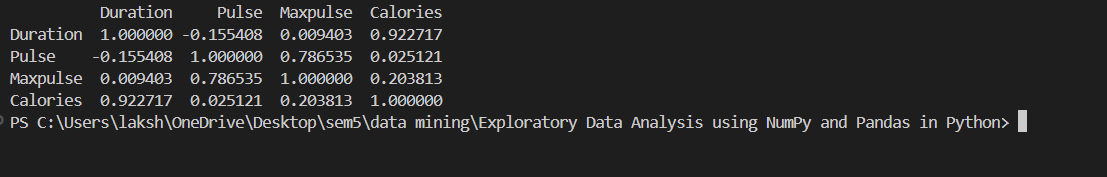
print(merged\_df.head())



Step 6: Calculate and display the correlation matrix for the numerical columns in the dataset.

correlation\_matrix = df.corr()

print(correlation\_matrix)



Step 7: Generate a scatter plot to visualize the relationship between two numerical columns. Include appropriate labels and titles.

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

plt.scatter(df['Duration'], df['Calories'], color='blue', alpha=0.7)

# Adding labels and title

plt.xlabel('Duration (minutes)')

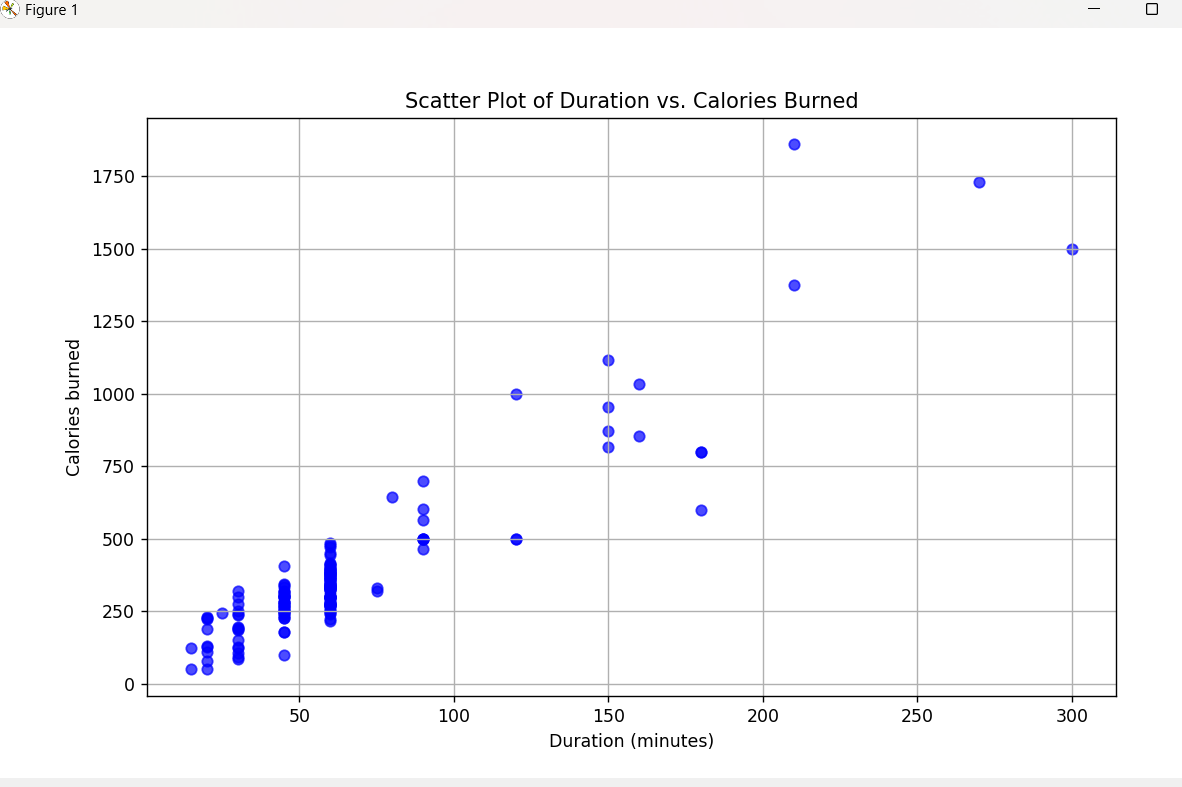
plt.ylabel('Calories burned')

plt.title('Scatter Plot of Duration vs. Calories Burned')

# Show the plot

plt.grid(True)

plt.show()

** EXPERIMENT 2: Exploratory Data Analysis using Weka**

Step 1: Download Weka

Step 2: Load and explore a loan data set in Weka.

import pandas as pd

# Load dataset

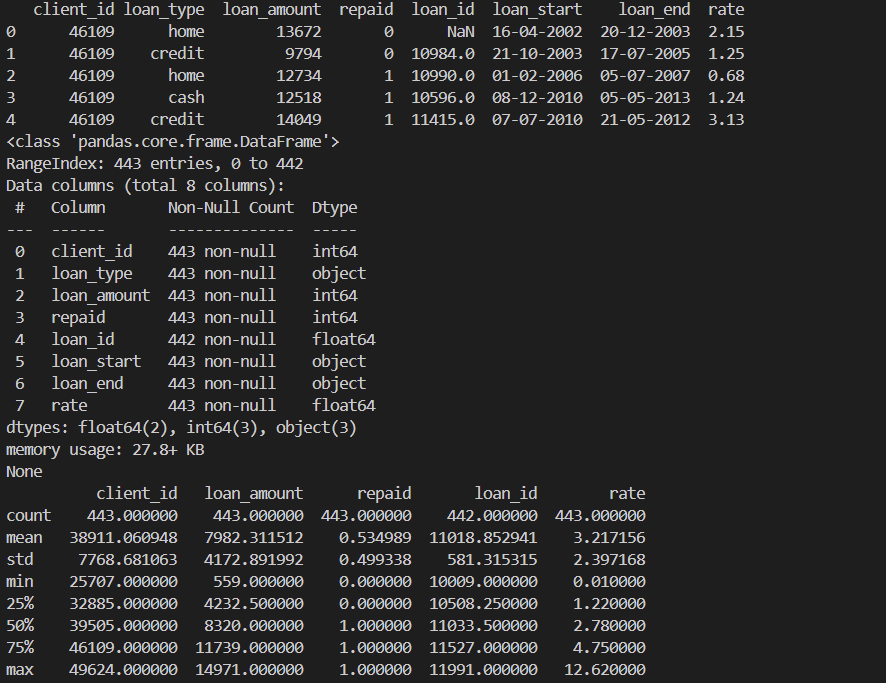
Python ->

data = pd.read\_csv("loan\_data.csv") # Use your dataset path here

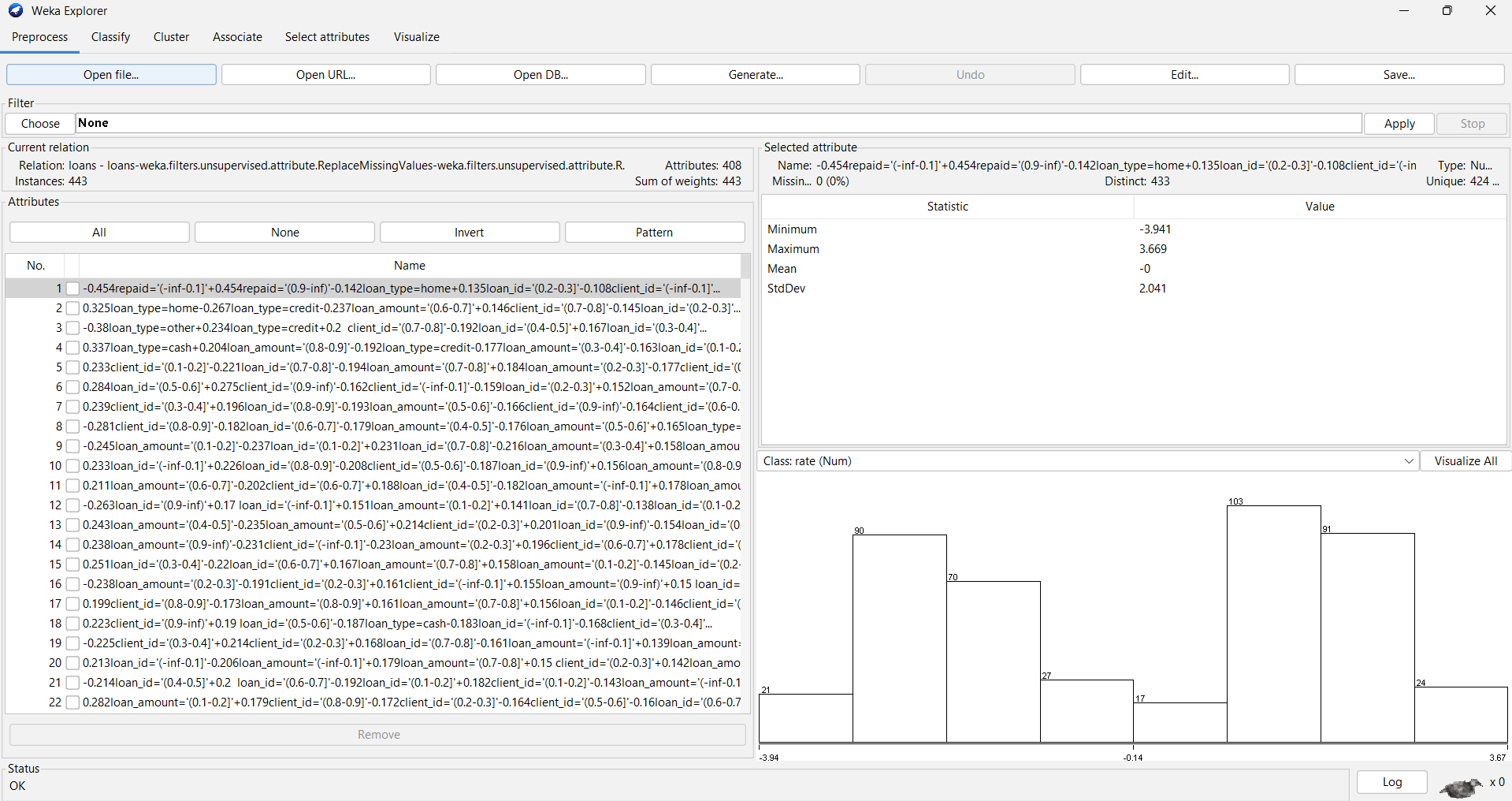
print(data.head()) # Preview the first few rows

print(data.info()) # Overview of dataset (data types, missing values)

print(data.describe()) # Statistical summary of numerical attributes



Weka->



Step 3: Handle missing data in Weka using imputation techniques

Python ->

from sklearn.impute import SimpleImputer

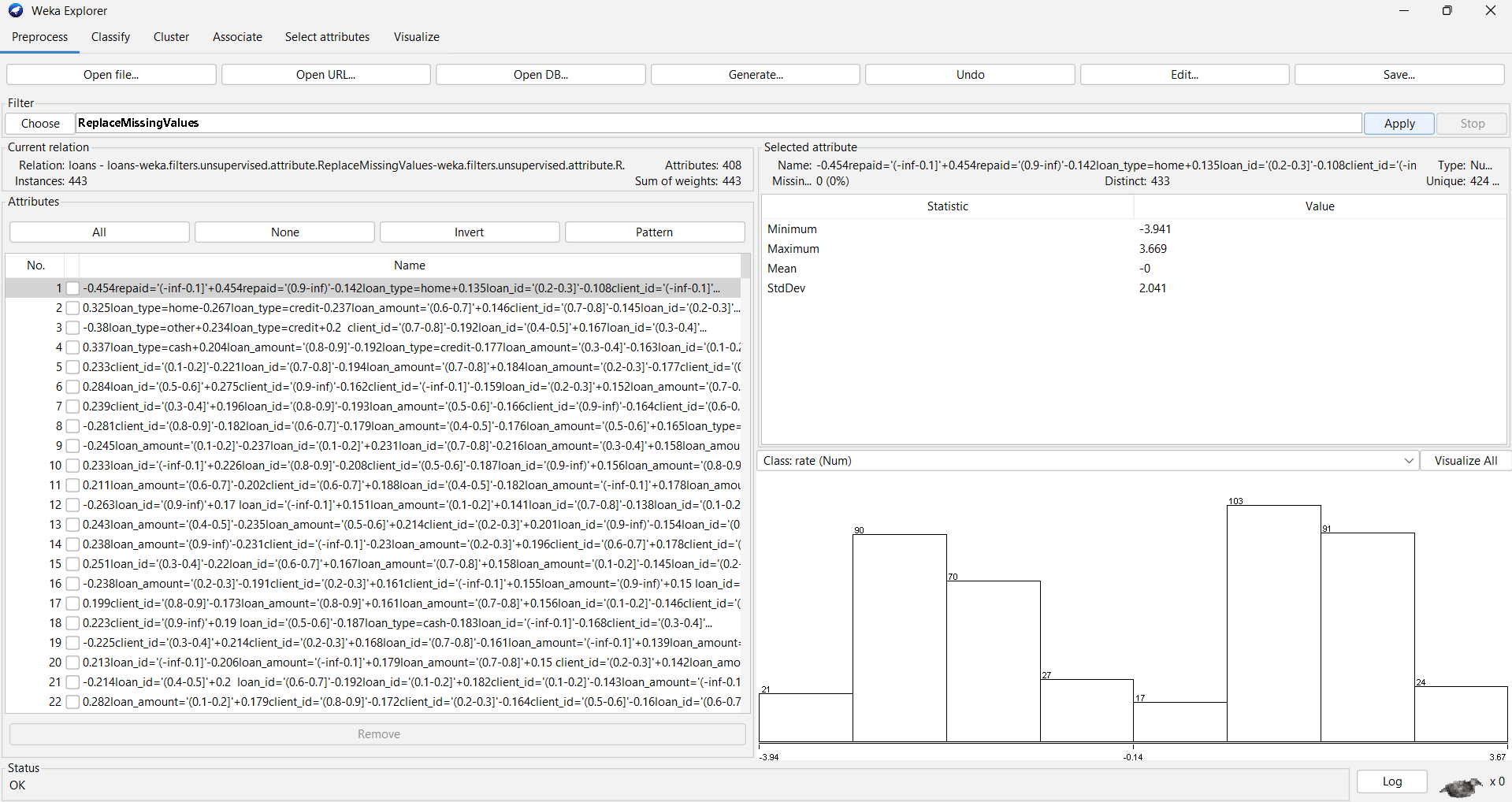
# Impute missing values with median for numerical features

imputer = SimpleImputer(strategy="median")

data\_imputed = data.copy()

data\_imputed[data.select\_dtypes(include=['float64', 'int64']).columns] = imputer.fit\_transform(data.select\_dtypes(include=['float64', 'int64']))

Weka ->



Step 4: Identify and remove outliers

import numpy as np

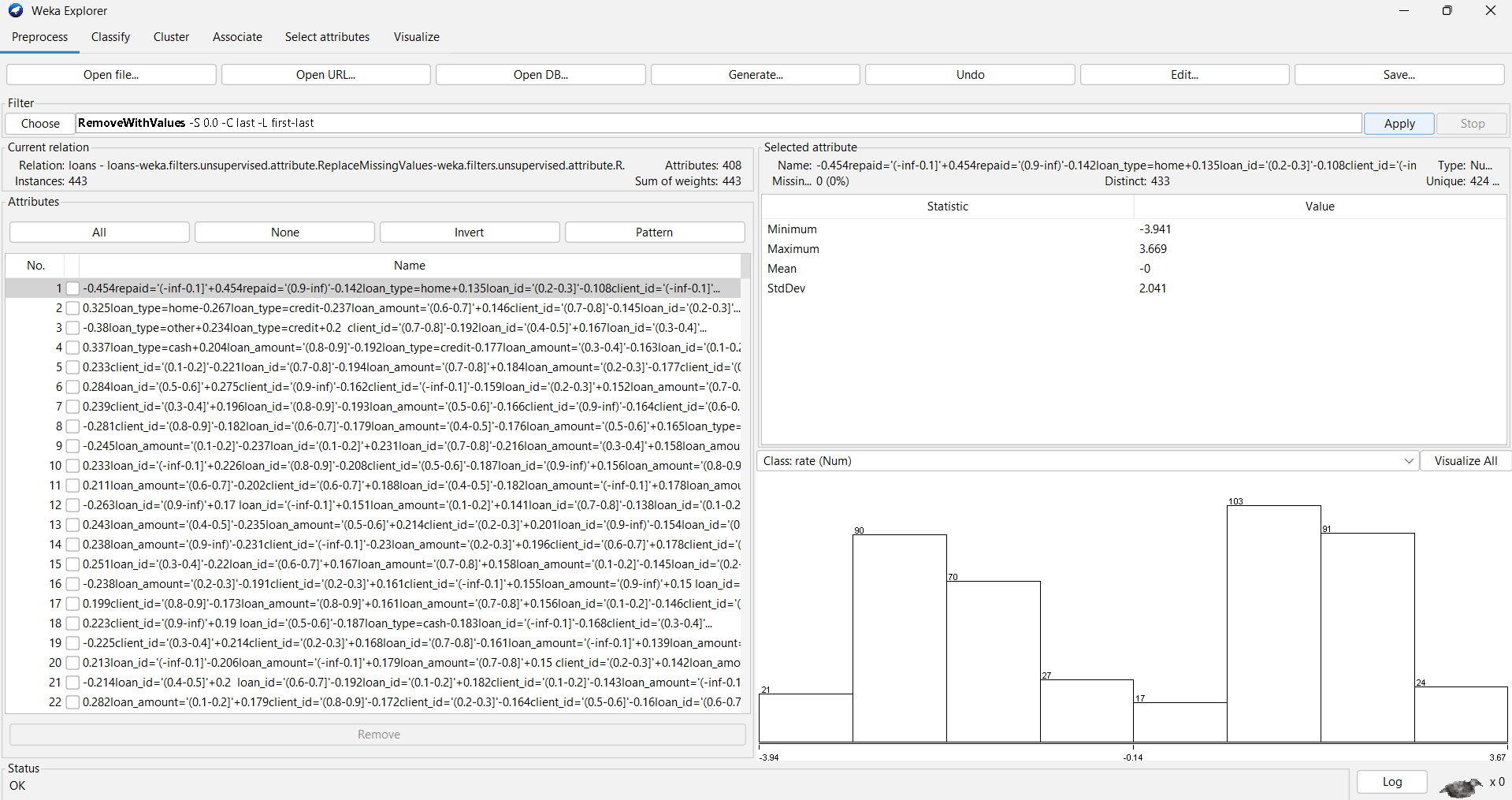
# Using IQR to filter out outliers

Q1 = data\_imputed.quantile(0.25)

Q3 = data\_imputed.quantile(0.75)

IQR = Q3 - Q1

data\_no\_outliers = data\_imputed[~((data\_imputed < (Q1 - 1.5 \* IQR)) | (data\_imputed > (Q3 + 1.5 \* IQR))).any(axis=1)]

print("Data after outlier removal:", data\_no\_outliers.shape) 

Step 5: Normalise or standardise the data

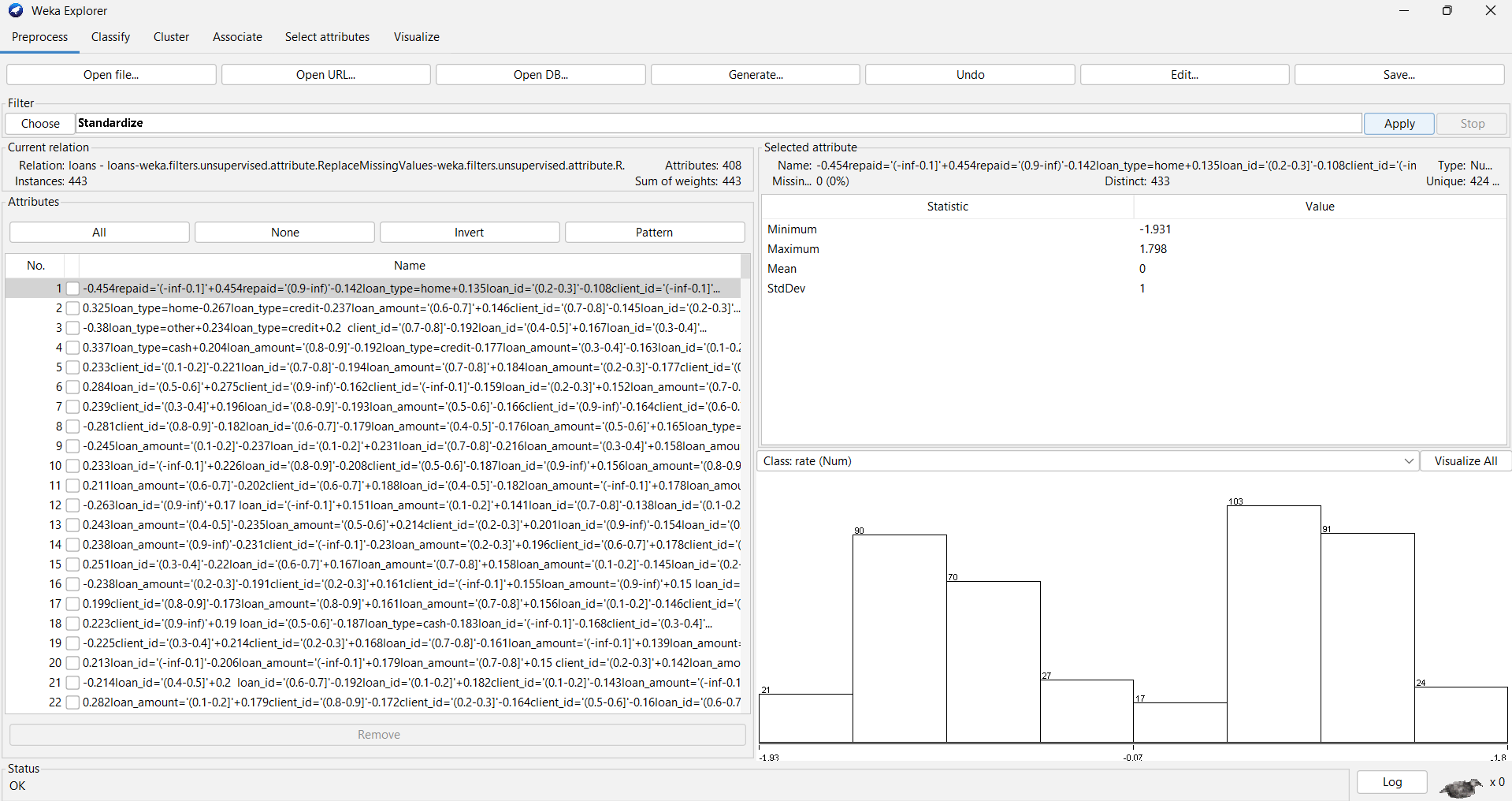
Python->

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

data[['loan\_amount', 'rate']] = scaler.fit\_transform(data[['loan\_amount', 'rate']])

Weka->



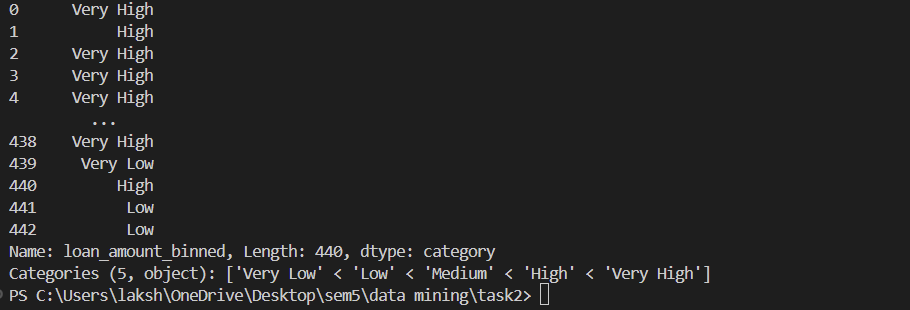
Step 6: Perform data discretization using Weka’s binning methods.

Python->

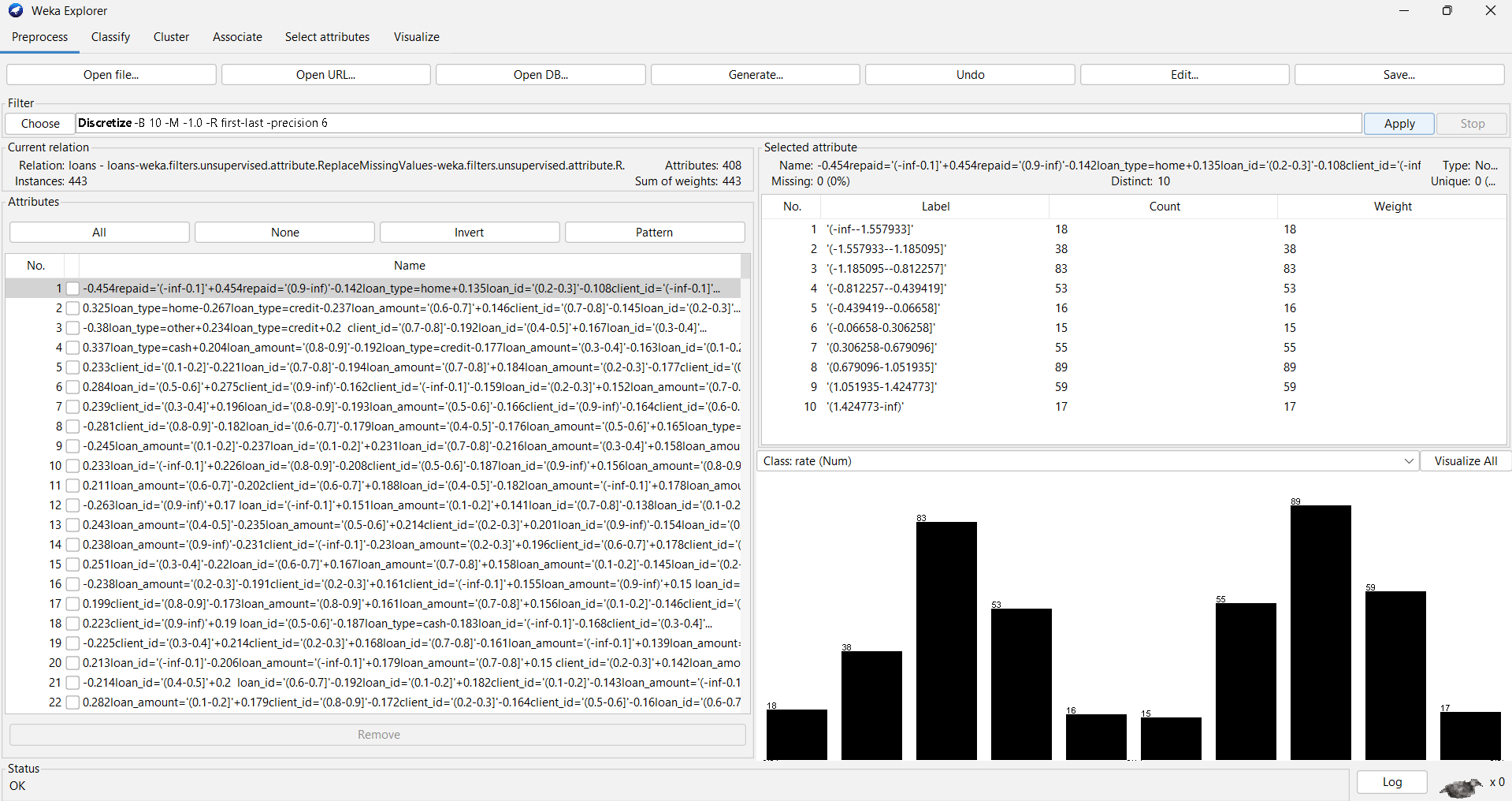
data = pd.get\_dummies(data, columns=['loan\_type'], drop\_first=True)

data['loan\_amount\_binned'] = pd.cut(data['loan\_amount'], bins=5, labels=['Very Low', 'Low', 'Medium', 'High', 'Very High'])

print(data['loan\_amount\_binned'])

Output -> 

Weka ->



Step 7: Generate concept hierarchies for categorical data in Weka and apply PCA.

Python ->

def loan\_type\_hierarchy(loan\_type):

if loan\_type in ["home\_loan", "auto\_loan"]:

return "secured\_loan"

elif loan\_type in ["personal\_loan", "business\_loan"]:

return "unsecured\_loan"

else:

return "other"

# Apply the hierarchy function to create a new column

data['loan\_type\_hierarchy'] = data['loan\_type'].apply(loan\_type\_hierarchy)

def loan\_amount\_hierarchy(amount):

if amount < 50000:

return "low"

elif 50000 <= amount < 150000:

return "medium"

else:

return "high"

# Apply the function

data['loan\_amount\_hierarchy'] = data['loan\_amount'].apply(loan\_amount\_hierarchy)

#pca

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

# Select only numerical columns to perform PCA

numerical\_cols = ['loan\_amount', 'rate']  # Add other numerical columns if available

data\_numerical = data[numerical\_cols]

# Step 1: Standardize the numerical data

scaler = StandardScaler()

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

# Step 2: Apply PCA to reduce to 2 principal components

pca = PCA(n\_components=2)

data\_pca = pca.fit\_transform(data\_scaled)

# Step 3: Add the PCA results back to the original dataset

data['PCA1'] = data\_pca[:, 0]

data['PCA2'] = data\_pca[:, 1]

# Print explained variance for each component

print("Explained variance by each component:", pca.explained\_variance\_ratio\_)

# Step 4: Visualize the PCA components

plt.figure(figsize=(8, 6))

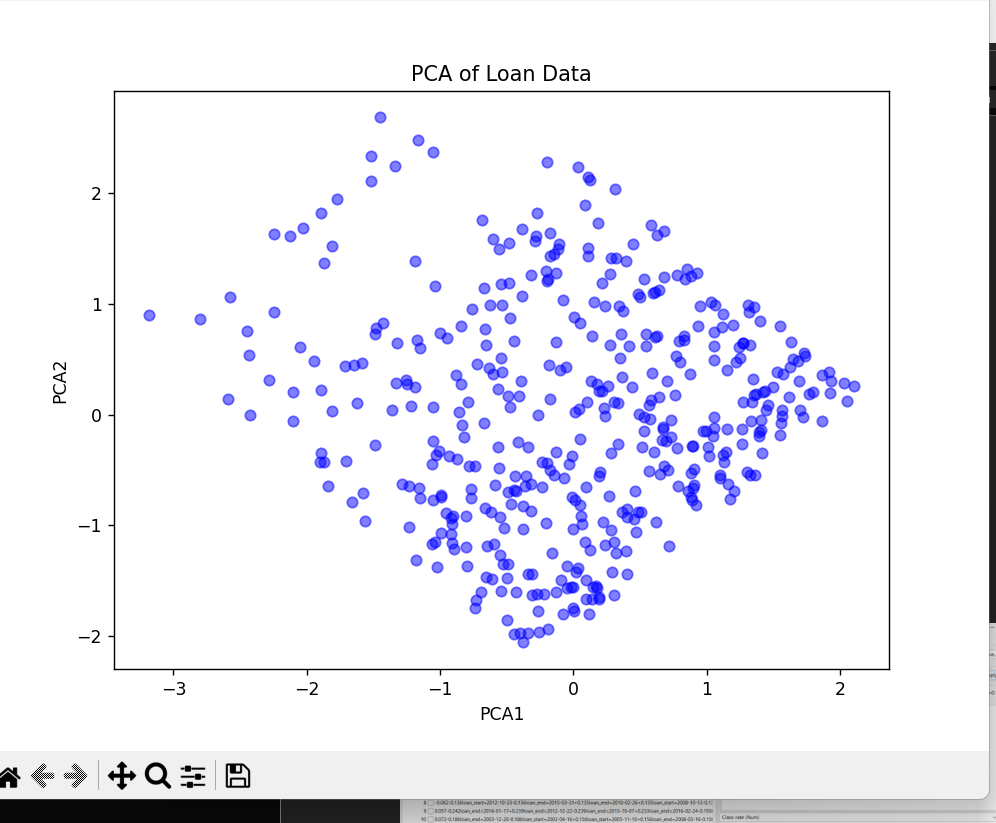
plt.scatter(data['PCA1'], data['PCA2'], c='blue', alpha=0.5)

plt.xlabel('PCA1')

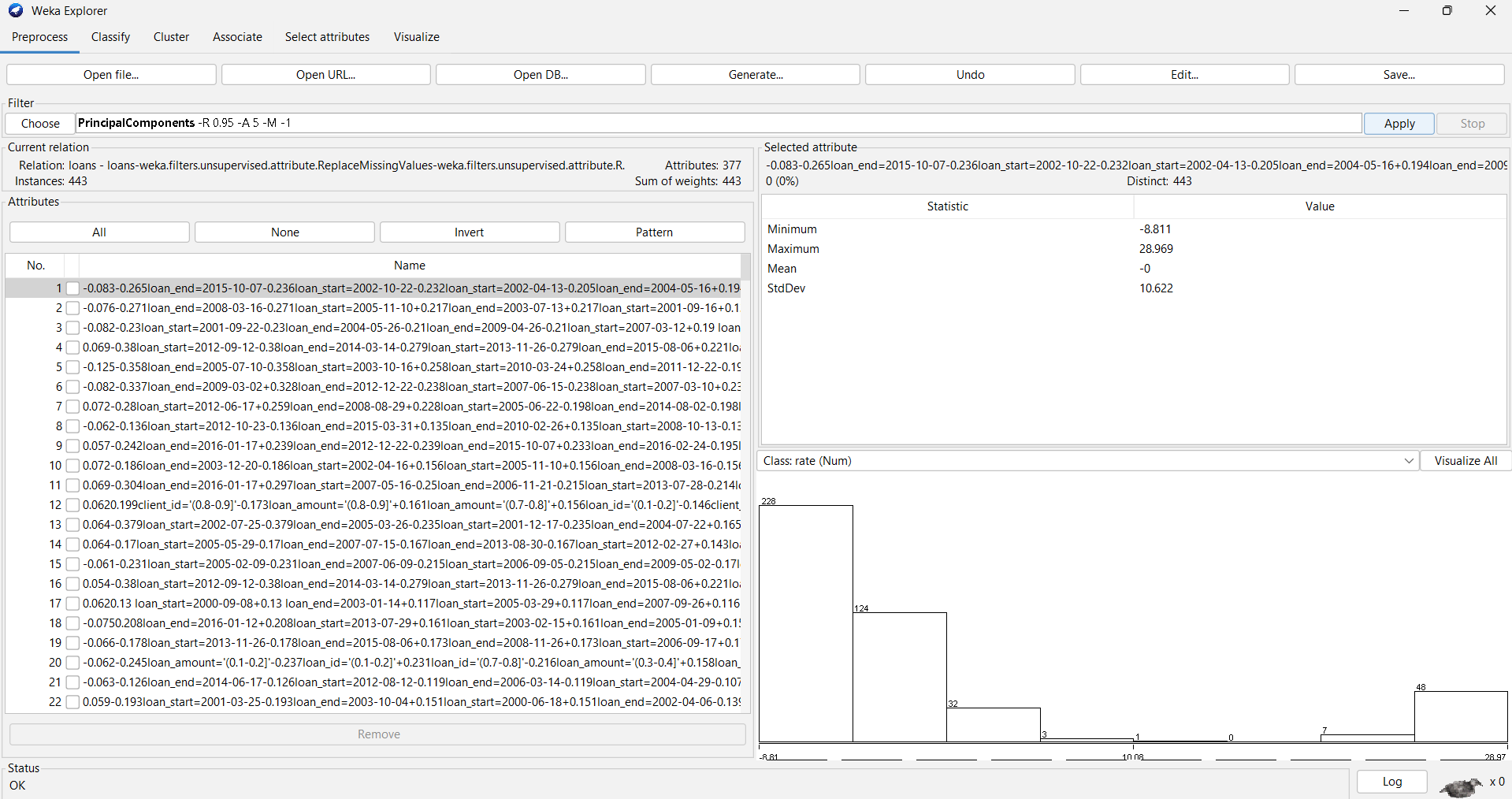
plt.ylabel('PCA2')

plt.title('PCA of Loan Data')

plt.show()

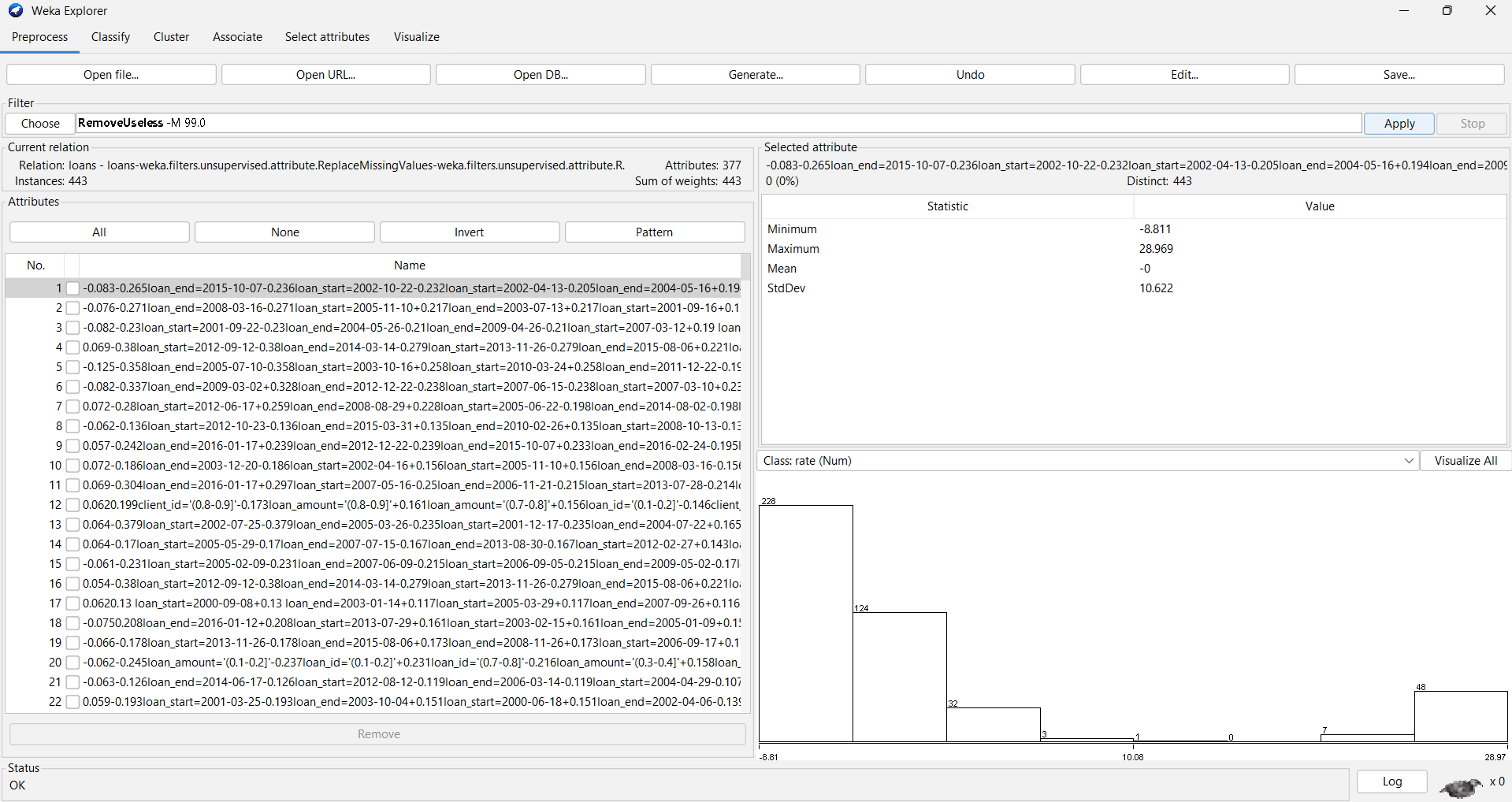


Weka ->



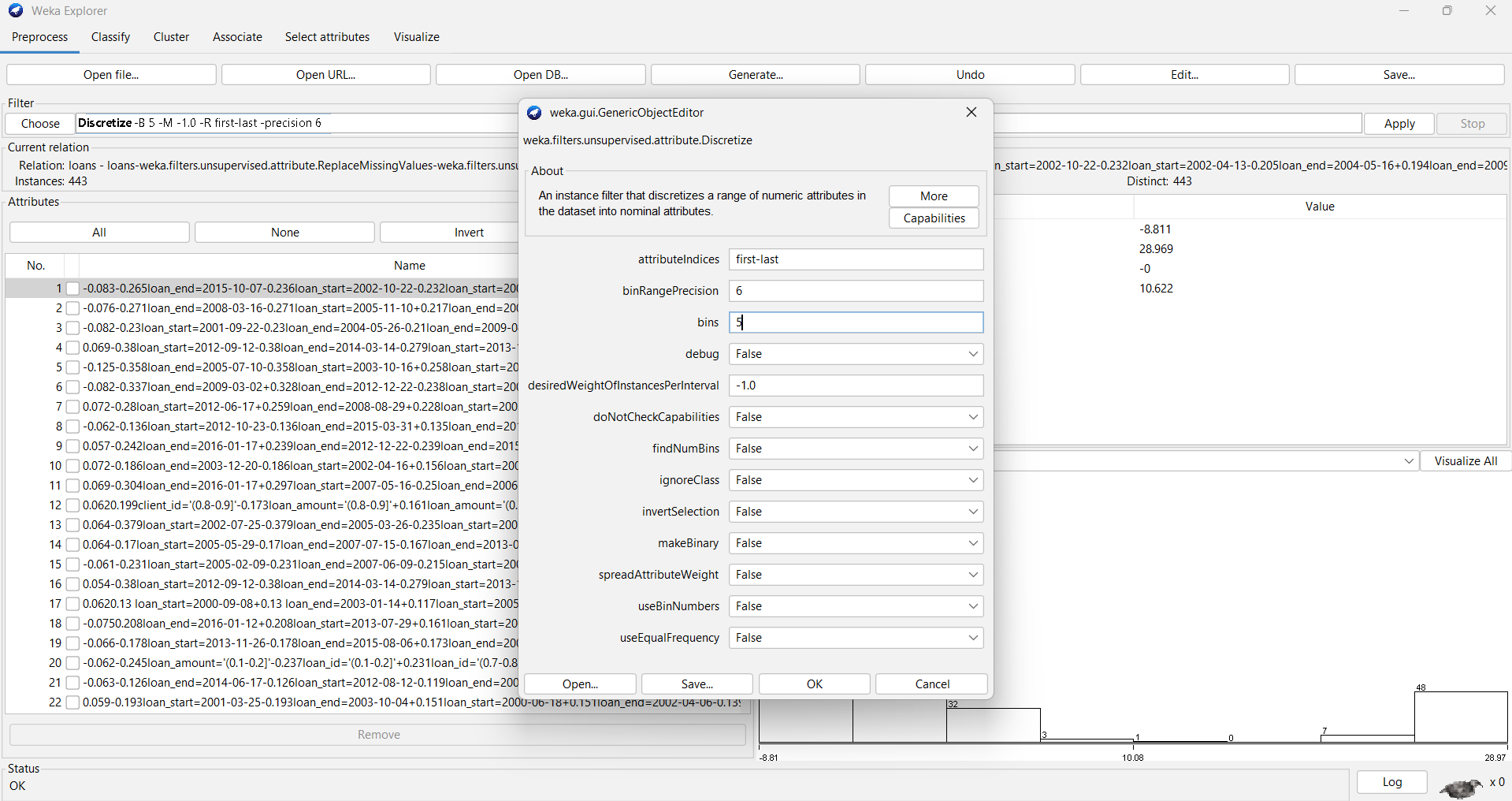
Step 8: Apply filters to preprocess and refine the data.

Weka ->



Step 9: Discretize continuous attributes in weka using built-in methods.

Weka ->



A screenshot of a computer

Description automatically generated

**EXPERIMENT 3 : Data Preprocessing and Data Warehouse Operations**

import pandas as pd

try:

    product\_df = pd.read\_csv('Online\_retail.csv', encoding='ISO-8859-1')

    store\_df = pd.read\_csv('train.csv', encoding='ISO-8859-1')

    salesperson\_df = pd.read\_csv('Salesperson.csv', encoding='ISO-8859-1')

    sales\_fact\_df = pd.read\_csv('Global\_Superstore2.csv', encoding='ISO-8859-1')

except UnicodeDecodeError as e:

    print(f"Error reading file: {e}")

# Display the first few rows of each dataset to verify the load

print("Product Dimension:")

print(product\_df.head())

print("\nStore Dimension:")

print(store\_df.head())

print("\nSalesperson Dimension:")

print(salesperson\_df.head())

print("\nSales Fact:")

print(sales\_fact\_df.head())

product\_df = product\_df[['StockCode', 'Description', 'UnitPrice']]

store\_df = store\_df[['Order ID', 'Order Date', 'Product ID', 'Sales', 'Region']]

salesperson\_df = salesperson\_df[['EmployeeNumber', 'EmployeeCount', 'Department', 'JobRole']]

sales\_fact\_df = sales\_fact\_df[['Product Name', 'Sales', 'Quantity', 'Discount', 'Profit']]

product\_df.dropna(inplace=True)

store\_df.dropna(inplace=True)

salesperson\_df.dropna(inplace=True)

sales\_fact\_df.dropna(inplace=True)

product\_df['UnitPrice'] = product\_df['UnitPrice'].astype(float)

sales\_fact\_df['Sales'] = sales\_fact\_df['Sales'].astype(float)

sales\_fact\_df['Quantity'] = sales\_fact\_df['Quantity'].astype(int)

sales\_fact\_df['Discount'] = sales\_fact\_df['Discount'].astype(float)

sales\_fact\_df['Profit'] = sales\_fact\_df['Profit'].astype(float)

from sqlalchemy import create\_engine, Table, Column, Integer, String, Float, Date

from sqlalchemy.ext.declarative import declarative\_base

# Create a SQLite engine and base class

engine = create\_engine('sqlite:///data\_warehouse.db')

Base = declarative\_base()

# Define dimension tables

class Product(Base):

    \_\_tablename\_\_ = 'Product'

    ProductID = Column(Integer, primary\_key=True)

    Description = Column(String)

    UnitPrice = Column(Float)

    StockCode = Column(String)

class Store(Base):

    \_\_tablename\_\_ = 'Store'

    StoreID = Column(Integer, primary\_key=True)

    StoreName = Column(String)

    City = Column(String)

    State = Column(String)

    Region = Column(String)

class Salesperson(Base):

    \_\_tablename\_\_ = 'Salesperson'

    EmployeeID = Column(Integer, primary\_key=True)

    Name = Column(String)

    Region = Column(String)

    JobRole = Column(String)

# Define the fact table

class SalesFact(Base):

    \_\_tablename\_\_ = 'SalesFact'

    OrderID = Column(Integer, primary\_key=True)

    OrderDate = Column(Date)

    ProductID = Column(Integer)

    StoreID = Column(Integer)

    EmployeeID = Column(Integer)

    Sales = Column(Float)

    Quantity = Column(Integer)

    Discount = Column(Float)

    Profit = Column(Float)

# Create tables

Base.metadata.create\_all(engine)

from sqlalchemy import inspect

# Inspect the tables

inspector = inspect(engine)

# List tables

print("Tables in the database:")

for table\_name in inspector.get\_table\_names():

    print(table\_name)

# Inspect columns in SalesFact table

columns = inspector.get\_columns('SalesFact')

print("\nColumns in SalesFact:")

for column in columns:

    print(f"Column: {column['name']}, Type: {column['type']}")

# Example query: Total sales per day

query = """

    SELECT OrderDate, SUM(Sales) as TotalSales

    FROM SalesFact

    GROUP BY OrderDate

    ORDER BY OrderDate

"""

sales\_trends = pd.read\_sql(query, engine)

print("Sales Trends:")

print(sales\_trends)

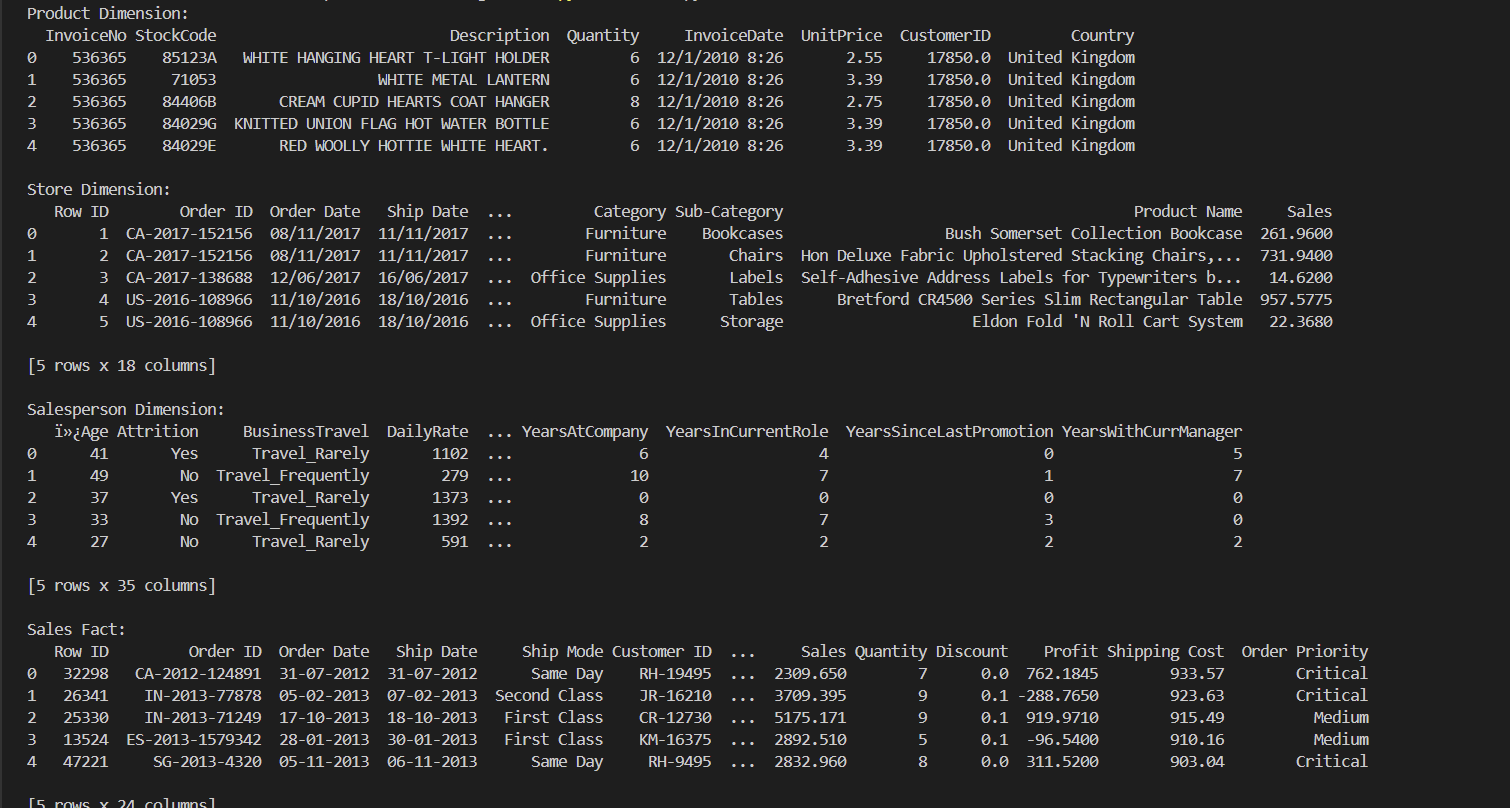
# Check indexes in the database

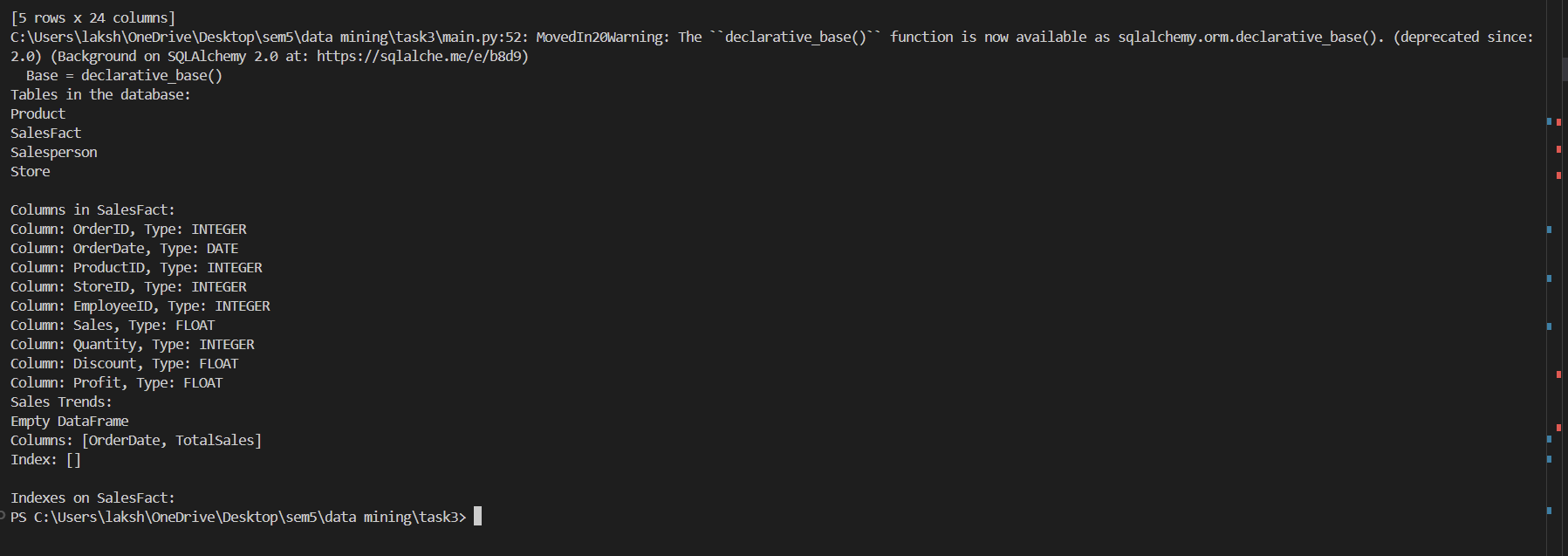
indexes = inspector.get\_indexes('SalesFact')

print("\nIndexes on SalesFact:")

for index in indexes:

    print(f"Index Name: {index['name']}, Columns: {index['column\_names']}")





**EXPERIMENT 4: Implement Linear Regression from scratch**

import numpy as np

class MyLR:

    def \_\_init\_\_(self):

        self.m = None

        self.b = None

    def fit(self,X\_train,y\_train):

        num = 0

        den = 0

        for i in range(X\_train.shape[0]):

            num += ((X\_train[i] - X\_train.mean())\*(y\_train[i] - y\_train.mean()))

            den += ((X\_train[i] - X\_train.mean())\*(X\_train[i] - X\_train.mean()))

        self.m = num/den

        self.b = y\_train.mean() - (self.m \* X\_train.mean())

        print(self.m)

        print(self.b)

    def predict(self,X\_test):

        print(X\_test)

        return self.m \* X\_test + self.b

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

# extracting X and y

X = df.iloc[:,0].values

y = df.iloc[:,1].values

#training the model

#splitting data into train and test

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

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=0.2,random\_state=2)

#lr will have fit and transform

lr = MyLR()

#fit the model with training data

lr.fit(X\_train,y\_train)

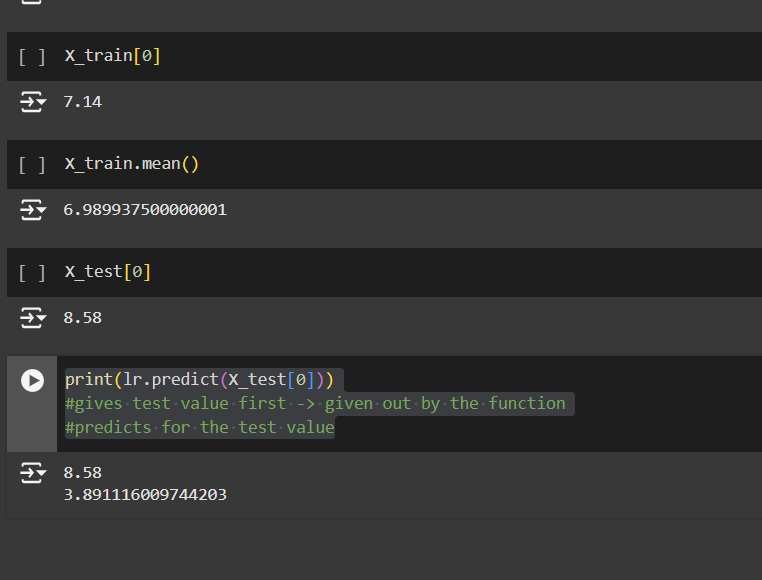
#gives slope(m) and b(constant)

print(lr.predict(X\_test[0]))

#gives test value first -> given out by the function

#predicts for the test value

Output:



**EXPERIMENT 5: Naïve Bayes classifier and Bayesian network**

1. Download the Titanic dataset and import it into Python.
2. Preprocess the data by handling missing values, converting categorical attributes (e.g., "Sex", "Embarked") into numerical form.
3. Perform feature selection in both Weka and Python (consider features like "Pclass", "Sex", "Age", "Fare", etc.).
4. In Python, implement Naive Bayes using GaussianNB from the sklearn library.
5. Perform 10-fold cross-validation in both Weka and Python, and record the accuracy of the model.
6. Generate confusion matrices and classification reports in Python. Compare the results and provide insights based on the differences in model performance across the two platforms

import pandas as pd

from sklearn.naive\_bayes import GaussianNB

from sklearn.feature\_selection import SelectKBest, chi2

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

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

from sklearn.preprocessing import StandardScaler

# Load and prepare the dataset

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

print("Initial DataFrame:")

print(df.head())

# Fill missing values

df['Age'].fillna(df['Age'].median(), inplace=True)

df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)

df['Fare'].fillna(df['Fare'].median(), inplace=True)

# Convert categorical columns to numerical values

df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})

df['Embarked'] = df['Embarked'].map({'C': 0, 'Q': 1, 'S': 2})

# Feature selection

X = df[['Pclass', 'Sex', 'Age', 'Fare', 'Embarked']]

y = df['Survived'] # Assuming this column exists

# Split dataset into training and testing sets

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

# Train Naive Bayes model

model = GaussianNB()

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

# Make predictions and calculate accuracy

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

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

print(f"Accuracy of Naive Bayes model: {accuracy:.2f}")

# Feature selection using SelectKBest with Chi-squared test

selector = SelectKBest(score\_func=chi2, k='all')

selector.fit(X\_train, y\_train)

# Get scores and p-values

scores = selector.scores\_

p\_values = selector.pvalues\_

# Create DataFrame to display feature scores and p-values

feature\_scores = pd.DataFrame({'Feature': X.columns, 'Score': scores, 'p-value': p\_values})

print("\nFeature scores and p-values:")

print(feature\_scores)

# Select features based on p-value < 0.05

selected\_features = feature\_scores[feature\_scores['p-value'] < 0.05]['Feature']

print("\nSelected Features based on p-value < 0.05:")

print(selected\_features.tolist())

# Cross-validation

scaler = StandardScaler()

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

cv\_scores = cross\_val\_score(model, X\_scaled, y, cv=10, scoring='accuracy')

print("Cross-Validation Accuracy for each fold:")

print(cv\_scores)

print(f"\nMean Accuracy: {cv\_scores.mean():.2f}")

# Confusion Matrix and Classification Report

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

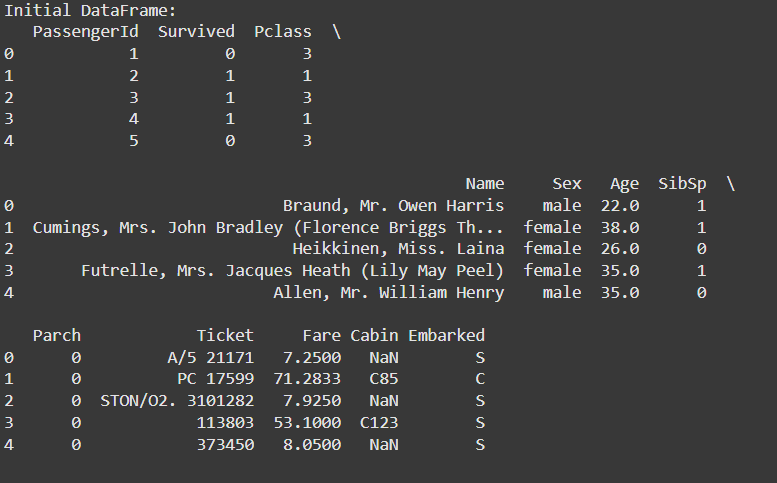
class\_report = classification\_report(y\_test, y\_pred)

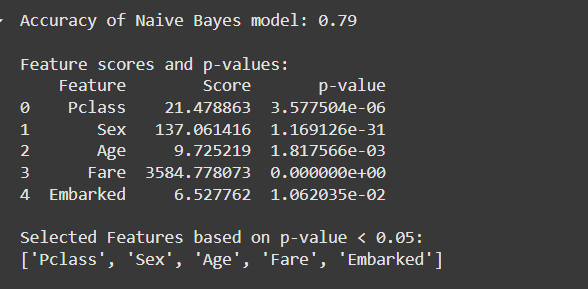
print("\nConfusion Matrix:")

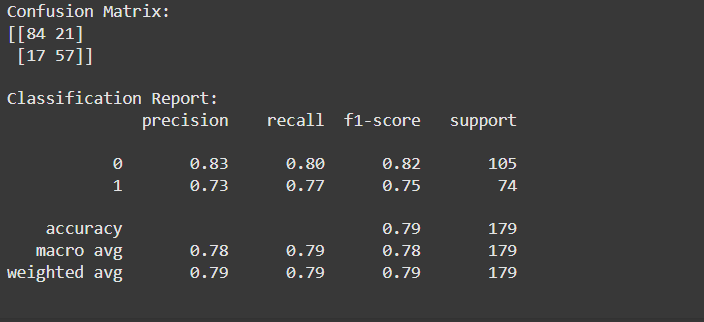
print(conf\_matrix)

print("\nClassification Report:")

print(class\_report)







1. Use bayesian net classifier on the preprocessed dataset to construct a bayesian network using pgmpy library

import pandas as pd

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

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

import seaborn as sns

import matplotlib.pyplot as plt

from pgmpy.models import BayesianNetwork

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.inference import VariableElimination

# Load data

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

# Preprocess missing values

df['Age'].fillna(df['Age'].median(), inplace=True)

df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)

df['Fare'].fillna(df['Fare'].median(), inplace=True)

# Convert categorical columns to numerical values

df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})

df['Embarked'] = df['Embarked'].map({'C': 0, 'Q': 1, 'S': 2})

# Define the features and target

X = df[['Pclass', 'Sex', 'Age', 'Fare', 'SibSp', 'Parch']]

y = df['Survived']

# Split data into training and testing sets

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

# Combine X\_train and y\_train into a single DataFrame for fitting

train\_data = X\_train.copy()

train\_data['Survived'] = y\_train

# Construct the Bayesian Network with relevant nodes and edges

model = BayesianNetwork([

('Pclass', 'Survived'),

('Sex', 'Survived'),

('Age', 'Survived'),

('Fare', 'Survived'),

('SibSp', 'Survived'),

('Parch', 'Survived')

])

# Fit the model with Maximum Likelihood Estimation

model.fit(train\_data, estimator=MaximumLikelihoodEstimator)

# Create an inference object for querying the model

inference = VariableElimination(model)

# Initialize predictions list

y\_pred\_bn = []

# Function to check if nodes exist in the model and filter evidence

def check\_nodes\_in\_model(evidence, model\_nodes):

"""

Filters evidence to include only nodes that exist in the model.

"""

filtered\_evidence = {key: value for key, value in evidence.items() if key in model\_nodes}

missing\_nodes = [key for key in evidence if key not in model\_nodes]

if missing\_nodes:

print(f"Warning: The following nodes are missing and will be ignored: {missing\_nodes}")

return filtered\_evidence

# Track the nodes in the model

model\_nodes = [node for node in model.nodes()]

# Iterate through the test instances and make predictions

for \_, row in X\_test.iterrows():

# Prepare evidence for the test instance

evidence = row[['Pclass', 'Sex', 'Age', 'Fare', 'SibSp', 'Parch']].to\_dict()

# Filter the evidence to include only valid nodes and remove NaN values

filtered\_evidence = check\_nodes\_in\_model(evidence, model\_nodes)

filtered\_evidence = {k: v for k, v in filtered\_evidence.items() if pd.notna(v)}

# Discretize continuous variables like 'Age' and 'Fare'

for var in ['Age', 'Fare']:

if var in filtered\_evidence:

discretized\_value = pd.cut([filtered\_evidence[var]], bins=5, labels=False)

if discretized\_value.size > 0 and pd.notna(discretized\_value[0]):

filtered\_evidence[var] = int(discretized\_value[0]) # Cast to int

else:

del filtered\_evidence[var]

print(f"Warning: Discretization for {var} resulted in an invalid label. Removing from evidence.")

# Query the model for prediction

try:

result = inference.query(variables=['Survived'], evidence=filtered\_evidence)

predicted\_class = 1 if result.values[1] > result.values[0] else 0

except KeyError as e:

predicted\_class = 0 # Default to 'Not Survived' if error occurs

# Append the prediction to y\_pred\_bn

y\_pred\_bn.append(predicted\_class)

# Check for consistency in length before evaluating

if len(y\_pred\_bn) == len(y\_test):

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

print(f"Bayesian Network Accuracy on Test Data: {accuracy}")

# Generate confusion matrix

cm = confusion\_matrix(y\_test, y\_pred\_bn)

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",

xticklabels=['Not Survived', 'Survived'],

yticklabels=['Not Survived', 'Survived'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix (Bayesian Network)')

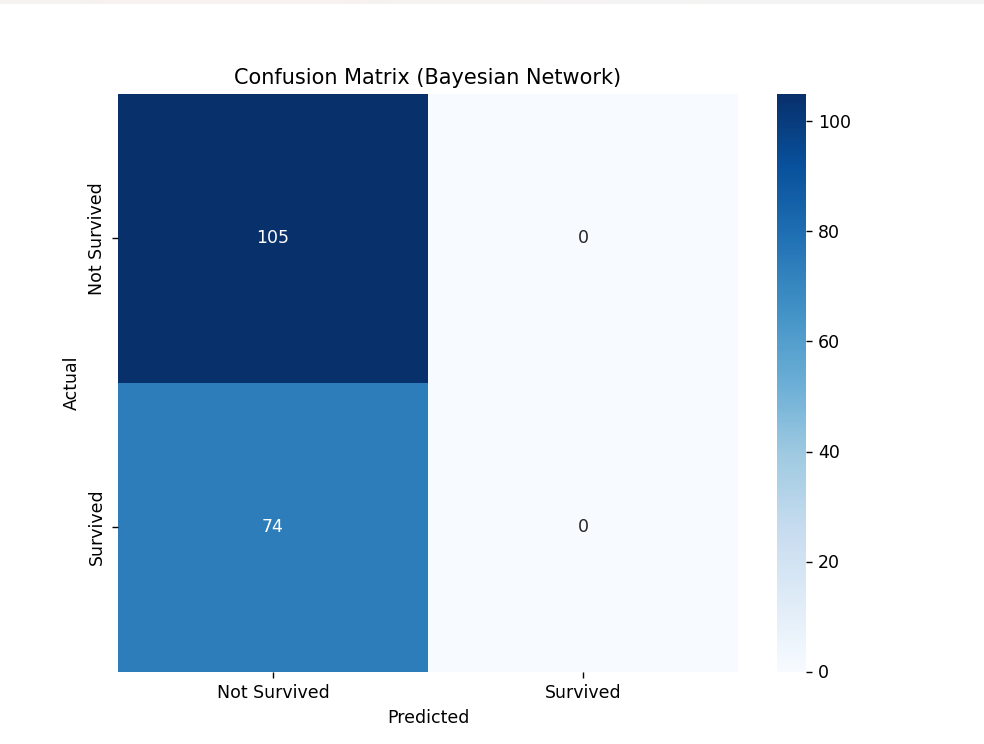
plt.show()

else:

print("Warning: Length mismatch between y\_test and y\_pred\_bn.")

print(f"y\_test length: {len(y\_test)}, y\_pred\_bn length: {len(y\_pred\_bn)}")

!Bayesian Network Accuracy on Test Data: 0.5865921787709497



**EXPERIMENT 6 : Decision Tree Implementation in Python**

1. Data Loading and Preprocessing:  
   Load a dataset (e.g., the UCI Iris dataset or any classification dataset of your choice) using pandas.  
   Perform basic preprocessing steps, such as handling missing values and normalizing the features if needed.  
   Split the dataset into features and labels, and then perform an 80-20 train-test split using train\_test\_split.

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

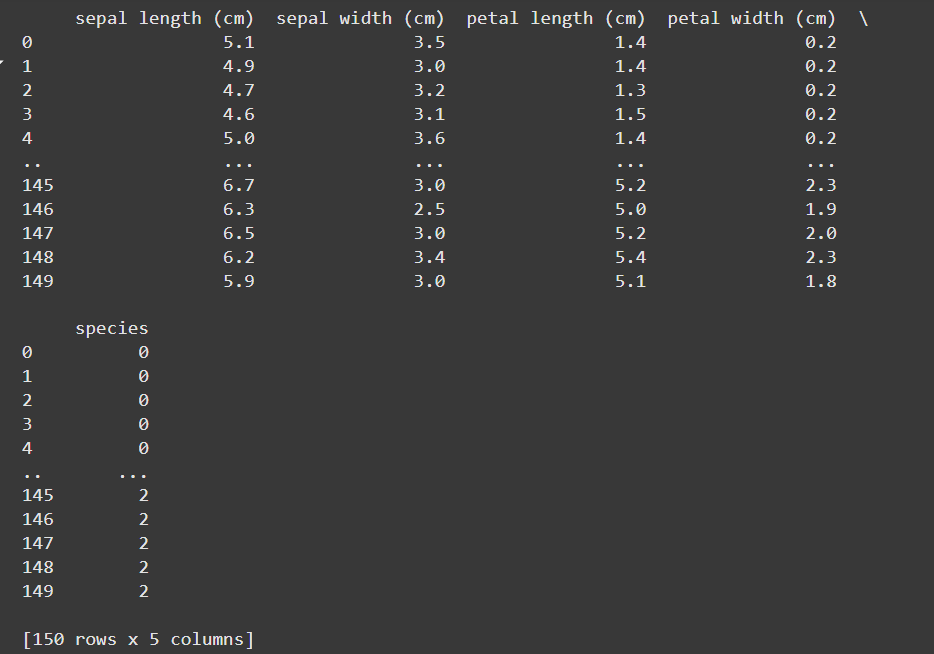
from sklearn.datasets import load\_iris

iris = load\_iris()

data = pd.DataFrame(data=iris.data, columns=iris.feature\_names)

data['species'] = iris.target

print(data)



scaler = StandardScaler()

features = scaler.fit\_transform(data.iloc[:, :-1])

X = features

y = data['species']

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

1. Decision Tree Classifier with Cross-Validation:  
   Implement a Decision Tree classifier using sklearn.tree.DecisionTreeClassifier.  
   Apply 5-fold cross-validation using cross\_val\_score to evaluate the model’s accuracy across different data splits.  
   Record the average accuracy and standard deviation from the cross-validation.

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import cross\_val\_score

import numpy as np

clf = DecisionTreeClassifier(random\_state=42)

cv\_scores = cross\_val\_score(clf, X\_train, y\_train, cv=5, scoring='accuracy')

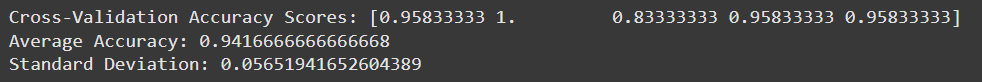
average\_accuracy = np.mean(cv\_scores)

std\_deviation = np.std(cv\_scores)

print("Cross-Validation Accuracy Scores:", cv\_scores)

print("Average Accuracy:", average\_accuracy)

print("Standard Deviation:", std\_deviation)



1. Holdout Set Evaluation:  
   Use the holdout test set from the initial train-test split to evaluate the model’s performance after training on the full training set.  
   Compute key performance metrics: accuracy, precision, recall, F1-score, and confusion matrix using classification\_report and confusion\_matrix.

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

clf.fit(X\_train, y\_train)

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

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

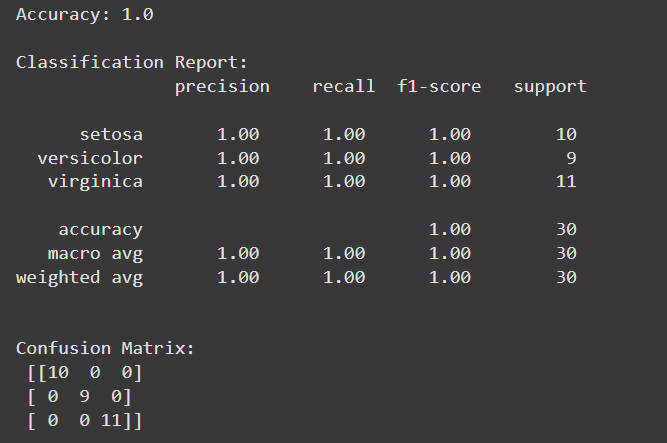
report = classification\_report(y\_test, y\_pred, target\_names=iris.target\_names)

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

print("Accuracy:", accuracy)

print("\nClassification Report:\n", report)

print("\nConfusion Matrix:\n", conf\_matrix)



1. Overfitting Prevention Techniques:  
   Implement strategies to avoid overfitting:  
   Prune the decision tree by setting parameters such as max\_depth, min\_samples\_split, and min\_samples\_leaf in the DecisionTreeClassifier.  
   Experiment with different values for these parameters and observe their effect on model performance.  
   Use GridSearchCV to optimize these hyperparameters and find the best combination.

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import GridSearchCV

param\_grid = {

    'max\_depth': [3, 5, 7, 10, None],

    'min\_samples\_split': [2, 5, 10],

    'min\_samples\_leaf': [1, 2, 4]

}

clf = DecisionTreeClassifier(random\_state=42)

grid\_search = GridSearchCV(estimator=clf, param\_grid=param\_grid, cv=5, scoring='accuracy', n\_jobs=-1)

grid\_search.fit(X\_train, y\_train)

best\_params = grid\_search.best\_params\_

best\_score = grid\_search.best\_score\_

best\_clf = grid\_search.best\_estimator\_

best\_clf.fit(X\_train, y\_train)

y\_pred = best\_clf.predict(X\_test)

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

report = classification\_report(y\_test, y\_pred, target\_names=iris.target\_names)

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

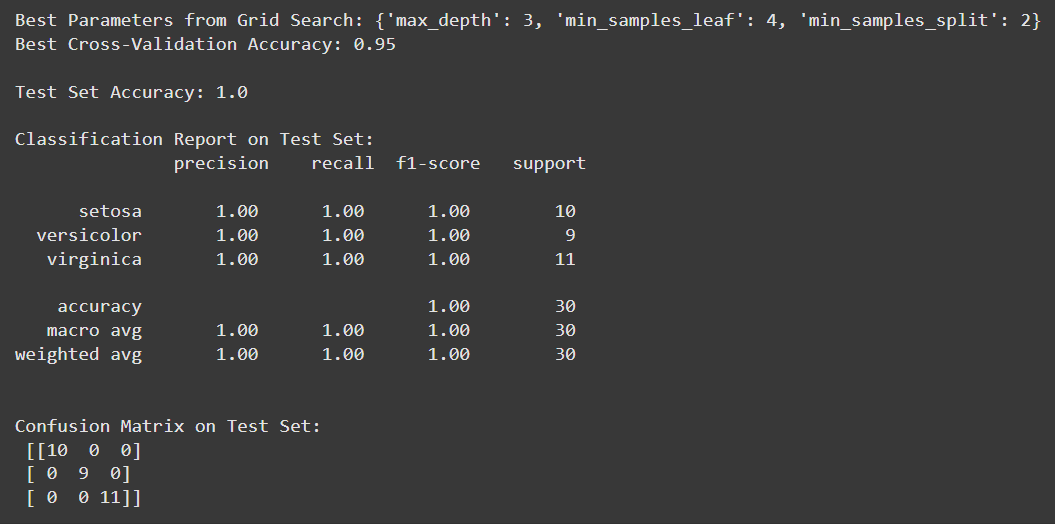
print("Best Parameters from Grid Search:", best\_params)

print("Best Cross-Validation Accuracy:", best\_score)

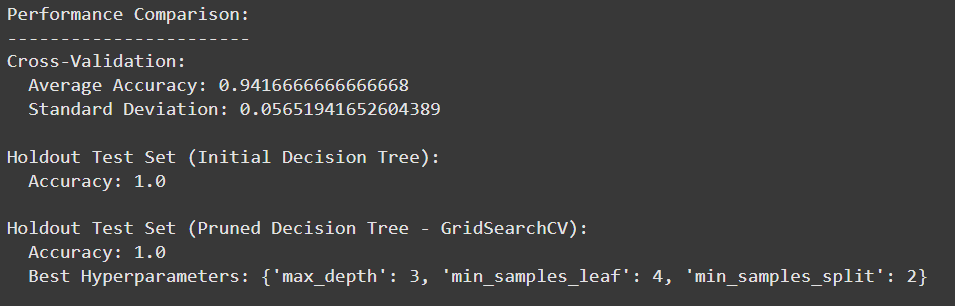
print("\nTest Set Accuracy:", accuracy)

print("\nClassification Report on Test Set:\n", report)

print("\nConfusion Matrix on Test Set:\n", conf\_matrix)



1. Performance Comparison and Conclusion:  
   Compare the performance metrics from the cross-validation, holdout test set, and the pruned tree. Discuss how pruning and cross-validation help reduce overfitting and improve the model’s generalizability.



**EXPERIMENT 7 : Forming strong association rules in python.**

from itertools import chain, combinations

from collections import defaultdict

import matplotlib.pyplot as plt

# Sample dataset

transactions = [

    {"Milk", "Bread", "Butter"},

    {"Bread", "Butter"},

    {"Milk", "Bread", "Sugar"},

    {"Milk", "Sugar"},

    {"Milk", "Bread", "Butter", "Sugar"},

    {"Bread", "Sugar"},

    {"Milk", "Bread"}

]

# Function to calculate all itemsets and their support

def calculate\_support(transactions):

    itemsets\_support = defaultdict(int)

    total\_transactions = len(transactions)

    # Generate all possible itemsets for each transaction and count their occurrences

    for transaction in transactions:

        for itemset in chain.from\_iterable(combinations(transaction, r) for r in range(1, len(transaction) + 1)):

            itemsets\_support[frozenset(itemset)] += 1

    # Calculate support as fraction of transactions

    for itemset in itemsets\_support:

        itemsets\_support[itemset] /= total\_transactions

    return itemsets\_support

# Task 1: Calculate support for all itemsets

itemsets\_support = calculate\_support(transactions)

# Task 2: Frequent Itemset Generation based on a minimum support threshold

min\_support\_threshold = 0.3  # Example threshold

frequent\_itemsets = {itemset: support for itemset, support in itemsets\_support.items() if support >= min\_support\_threshold}

# Task 3: Calculate confidence for association rules

def calculate\_confidence(frequent\_itemsets):

    rules\_confidence = {}

    for itemset in frequent\_itemsets:

        if len(itemset) > 1:

            for antecedent in chain.from\_iterable(combinations(itemset, r) for r in range(1, len(itemset))):

                antecedent = frozenset(antecedent)

                consequent = itemset - antecedent

                if antecedent in frequent\_itemsets:

                    confidence = frequent\_itemsets[itemset] / frequent\_itemsets[antecedent]

                    rules\_confidence[(antecedent, consequent)] = confidence

    return rules\_confidence

rules\_confidence = calculate\_confidence(frequent\_itemsets)

# Task 4: Calculate lift for association rules

def calculate\_lift(rules\_confidence, frequent\_itemsets):

    rules\_lift = {}

    for (antecedent, consequent), confidence in rules\_confidence.items():

        if consequent in frequent\_itemsets:

            lift = confidence / frequent\_itemsets[consequent]

            rules\_lift[(antecedent, consequent)] = lift

    return rules\_lift

rules\_lift = calculate\_lift(rules\_confidence, frequent\_itemsets)

# Task 5: Visualize Frequent Itemsets and their Support Values

itemset\_labels = [' '.join(itemset) for itemset in frequent\_itemsets.keys()]

support\_values = list(frequent\_itemsets.values())

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

plt.bar(itemset\_labels, support\_values, color='skyblue')

plt.xlabel("Frequent Itemsets")

plt.ylabel("Support")

plt.title("Frequent Itemsets and Their Support Values")

plt.xticks(rotation=45, ha='right')

plt.tight\_layout()

plt.show()

# Output results

itemsets\_support, frequent\_itemsets, rules\_confidence, rules\_lift

