Week 7 Programs

31. Write a program to classify the Iris dataset using a Decision Tree classifier.

```
# Import necessary libraries

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score

# Load the Iris dataset

iris = load_iris()

X = iris.data  # Features

y = iris.target  # Target labels (species)

# Split the data into training and testing 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)

# Initialize the Decision Tree classifier

clf = DecisionTreeClassifier(random_state=42)

# Train the classifier

clf.fit(X_train, y_train)

# Make predictions on the test set

y_pred = clf.predict(X_test)

# Evaluate the classifier's accuracy

accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy * 100:.2f}%')
```

32. Write a program to create and visualize a Decision Tree for the Iris dataset.

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn import tree
import matplotlib.pyplot as plt
# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target labels (species)
# Split the data into training and testing 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)
# Initialize the Decision Tree classifier
clf = DecisionTreeClassifier(random_state=42)
# Train the classifier
clf.fit(X_train, y_train)
# Make predictions on the test set
y_pred = clf.predict(X_test)
# Evaluate the classifier's accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
# Plot the Decision Tree (optional)
plt.figure(figsize=(12, 8))
tree.plot_tree(clf, filled=True)
plt.show()
```

33. Write a program to train a Decision Tree classifier on a simple dataset and make predictions.

```
# Import necessary libraries
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# Create a simple synthetic dataset
X, y = make_classification(n_samples=100, n_features=4, n_informative=2, n_classes=2, random_state=42)
# Split the dataset into training and testing 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)
# Initialize the Decision Tree classifier
clf = DecisionTreeClassifier(random_state=42)
# Train the classifier on the training data
clf.fit(X_train, y_train)
# Make predictions on the test data
y_pred = clf.predict(X_test)
# Evaluate the model by calculating accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
# Print the predictions
print(f'Predictions: {y_pred}')
```

34. Write a program to handle missing values in a dataset and train a Decision Tree classifier.

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score
# Create a synthetic dataset with missing values
X, y = make_classification(n_samples=100, n_features=4, n_informative=2, n_classes=2,
random_state=42)
# Introduce missing values randomly in the feature matrix X
rng = np.random.RandomState(42)
missing_mask = rng.rand(*X.shape) < 0.1  # 10% missing values
X[missing_mask] = np.nan  # Set those positions to NaN
# Convert X to a pandas DataFrame to handle missing values easily
X_df = pd.DataFrame(X)
# Handle missing values using SimpleImputer (imputation with the mean)
imputer = SimpleImputer(strategy='mean')  # You can also use 'median' or 'most_frequent'
X_imputed = imputer.fit_transform(X_df)
# Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X_imputed, y, test_size=0.2,
random_state=42)
# Initialize the Decision Tree classifier
clf = DecisionTreeClassifier(random_state=42)
# Train the classifier on the training data
clf.fit(X_train, y_train)
```

 # Make predictions on the test data

y_pred = clf.predict(X_test)

 # Evaluate the model by calculating accuracy

accuracy = accuracy_score(y_test, y_pred)

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

 # Print the predictions

print(f'Predictions: {y_pred}')

35. Write a program to train a Decision Tree classifier on the Breast Cancer dataset and print the accuracy score.

 # Import necessary libraries

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score

 # Load the Breast Cancer dataset

data = load_breast_cancer()

X = data.data # Features

y = data.target # Labels (Malignant: 0, Benign: 1)

 # Split the data into training and testing 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)

 # Initialize the Decision Tree classifier

clf = DecisionTreeClassifier(random_state=42)

 # Train the classifier on the training data

clf.fit(X_train, y_train)

 # Make predictions on the test data

y_pred = clf.predict(X_test)

 # Evaluate the model by calculating accuracy

accuracy = accuracy_score(y_test, y_pred)

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

Week 8 Programs

36. Write down the Procedure for Implementation of KNN using sklearn.

Step-by-step procedure to implement the K-Nearest Neighbors (KNN) algorithm using sklearn:

1. Import Libraries

First, import the necessary libraries. You'll need sklearn for KNN and some other libraries for data handling and evaluation.

```
# Import necessary libraries

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score
```

2. Load the Dataset

Load your dataset. You can use pandas to load a CSV, or if you're using any sklearn dataset, you can load it directly.

```
# Example: Load a sample dataset

# Replace this with your dataset

data = pd.read_csv('your_dataset.csv')

# Or you can use an sklearn dataset

# from sklearn.datasets import load_iris

# data = load_iris()
```

3. Preprocess the Data

Ensure your dataset is clean and formatted properly. This includes handling missing values, converting categorical variables (if needed), and separating the features (X) and the target (y).

```
# Split dataset into features (X) and target (y)

X = data.drop('target_column', axis=1)  # Replace with your target column name

y = data['target_column']  # Replace with your target column name
```

4. **Split the Dataset**

Divide the dataset into training and testing sets, typically using an 80-20 or 70-30 split.

```
# Split the data into training and testing 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)
```

5. **Normalize the Data**

KNN is sensitive to the scale of the features, so it's a good practice to standardize the data.

```
# Standardize the feature values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

6. **Create and Train the KNN Model**

Instantiate the KNN classifier and fit it to the training data. You can choose the value of k (the number of neighbors) according to your needs.

```
# Create the KNN model
k = 5  # You can choose any other value for k
knn = KNeighborsClassifier(n_neighbors=k)
# Train the model
knn.fit(X_train, y_train)
```

7. **Make Predictions**

Use the trained model to make predictions on the test set.

```
# Make predictions on the test set
y_pred = knn.predict(X_test)
```

8. **Evaluate the Model**

Evaluate the model's performance using accuracy or other metrics.

```
# Evaluate the model (Accuracy Score)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

37. Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions.

 # Importing necessary libraries

import numpy as np

import pandas as pd

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score

 # Load the Iris dataset

iris = load_iris()

X = iris.data # Features

y = iris.target # Target variable (species)

 # Split the data into training and testing 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)

 # Standardize the features (important for KNN)

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)

 # Initialize the KNN classifier with k=5

knn = KNeighborsClassifier(n_neighbors=5)

 # Train the model on the training data

knn.fit(X_train, y_train)

 # Make predictions on the test data

y_pred = knn.predict(X_test)

 # Print the correct and wrong predictions

```
print("Correct Predictions:")
for i in range(len(y_test)):
    if y_pred[i] == y_test[i]:
        print(f"Sample {i + 1}: Predicted = {iris.target_names[y_pred[i]]}, Actual = {iris.target_names[y_test[i]]}")


print("\nWrong Predictions:")
for i in range(len(y_test)):
    if y_pred[i] != y_test[i]:
        print(f"Sample {i + 1}: Predicted = {iris.target_names[y_pred[i]]}, Actual = {iris.target_names[y_test[i]]}")


# Calculate and print the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy: {accuracy * 100:.2f}%")
```

38. Write a python program to calculate Gini Impurity for the attributes of data set.

```
import pandas as pd
from collections import Counter
# Function to calculate Gini Impurity for a given dataset
def gini_impurity(class_values):
    # Count the frequency of each class in the dataset
    class_counts = Counter(class_values)
    total_instances = len(class_values)
    # Calculate the probability of each class
    probabilities = [count / total_instances for count in class_counts.values()]
    # Calculate the Gini Impurity
    gini = 1 - sum(p**2 for p in probabilities)
    return gini
# Load a dataset (For example, using Iris dataset from sklearn)
```

```
from sklearn.datasets import load_iris
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target  # Add target column
# Calculate Gini Impurity for each attribute (feature)
for column in df.columns[:-1]:  # Excluding target column
    feature_values = df[column]
    unique_values = feature_values.unique()
    # Group by the unique values of the feature and calculate Gini for each group
    print(f"\nGini Impurity for feature '{column}':")
    # Iterate through the unique values of the feature and calculate Gini for subsets
    for value in unique_values:
        subset = df[df[column] == value]['target']
        gini = gini_impurity(subset)
        print(f"  Gini Impurity for {column} = {value}: {gini:.4f}")
```

39. Write a python program to calculate Gini gain values to select the splitting position.

```
import pandas as pd
from sklearn.datasets import load_iris
from collections import Counter
# Function to calculate Gini Impurity for a given dataset
def gini_impurity(class_values):
    class_counts = Counter(class_values)
    total_instances = len(class_values)
    probabilities = [count / total_instances for count in class_counts.values()]
    gini = 1 - sum(p**2 for p in probabilities)
    return gini
# Function to calculate Gini Gain for a given feature
```

```python
def gini_gain(df, feature, target):
    # Calculate Gini Impurity for the parent node (whole dataset)
    parent_gini = gini_impurity(df[target])
    # Get unique values of the feature for potential splits
    feature_values = df[feature].unique()
    weighted_gini_sum = 0
    total_instances = len(df)
    # Iterate over the unique values of the feature to calculate the Gini of each subset
    for value in feature_values:
        subset = df[df[feature] == value]
        subset_gini = gini_impurity(subset[target])
        weight = len(subset) / total_instances
        weighted_gini_sum += weight * subset_gini
    # Gini Gain is the reduction in impurity after the split
    gini_gain_value = parent_gini - weighted_gini_sum
    return gini_gain_value
# Load Iris dataset from sklearn
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target  # Add target column
# Calculate Gini Gain for each feature in the dataset
for column in df.columns[:-1]:  # Exclude target column
    gini_gain_value = gini_gain(df, column, 'target')
    print(f"Gini Gain for feature '{column}': {gini_gain_value:.4f}")
```