Individual Assignment 1 Task 2

Name: Rohit Panda

UOW ID: 8943060

Before Running the code, ensure you have the kaggle.json file to upload into the Colab.

Download the kaggle.json from this link: [Download Now]

Dataset Loading & Setup

We load the Drug Classification dataset from Kaggle using the Kaggle API. The dataset is processed using pandas for further analysis and preparation. The dataset contains categorical and continuous attributes to predict drug type.

```
# ========== Task 2: Setup and Dataset Download ==========

# Step 0: Upload kaggle.json
from google.colab import files
uploaded = files.upload()  # Upload kaggle.json manually here
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving kaggle.json to kaggle.json

```
# Step 1: Move kaggle.json to the correct location
import os
if not os.path.exists(os.path.expanduser("~/.kaggle")):
    os.makedirs(os.path.expanduser("~/.kaggle"))
os.rename("kaggle.json", os.path.expanduser("~/.kaggle/kaggle.json"))
os.chmod(os.path.expanduser("~/.kaggle/kaggle.json"), 0o600)


# Step 2: Install Kaggle API (if not already installed)
!pip install -q kaggle
```

```
# Step 3: Download new dataset (replace this with your specific dataset name)
dataset_name = "prathamtripathi/drug-classification"
!kaggle datasets download -d {dataset_name}
```

```
Dataset URL: https://www.kaggle.com/datasets/prathamtripathi/drug-classification
License(s): CC0-1.0
Downloading drug-classification.zip to /content
  0% 0.00/1.68k [00:00<?, ?B/s]
100% 1.68k/1.68k [00:00<00:00, 6.61MB/s]
```

```
# Step 4: Unzip dataset
import zipfile
zip_file = dataset_name.split("/")[-1] + ".zip"  # credit-score-classification.zip
extract_path = "task2_data"

if os.path.exists(zip_file):
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        zip_ref.extractall(extract_path)
    print(f"✅ Dataset extracted to {extract_path}/")
else:
    print(f"❌ ERROR: Zip file {zip_file} not found.")
```

```
✅ Dataset extracted to task2_data/
```

```
# Step 5: Load CSV (adjust the filename if different)
import pandas as pd
csv_path = f"{extract_path}/drug200.csv"  # Replace with test.csv or other if needed

if os.path.exists(csv_path):
    df = pd.read_csv(csv_path)
    print("✅ DataFrame loaded successfully!")
    print(f"Shape: {df.shape}")
    print("First 5 rows:")
    print(df.head())
else:
    print(f"❌ ERROR: CSV file not found at {csv_path}")
```

```
✅ DataFrame loaded successfully!
Shape: (200, 6)
First 5 rows:
   Age Sex      BP Cholesterol  Na_to_K   Drug
0   23   F    HIGH        HIGH   25.355  DrugY
1   47   M     LOW        HIGH   13.093  drugC
2   47   M     LOW        HIGH   10.114  drugC
3   28   F  NORMAL        HIGH    7.798  drugX
4   61   F     LOW        HIGH   18.043  DrugY
```

```
import pandas as pd
import numpy as np
import random
import math
```

Preprocessing and Binning

All missing values are handled using mean (for numeric) and mode (for categorical) imputation. Then, all continuous attributes are transformed into categorical bins (`Low`, `Medium`, `High`) using quantile-based binning (`pd.qcut()`).

```
print("=== Task 2: Drug Classification with Decision Tree ===")

# Load the dataset
print("\n1. Loading Dataset")
# Note: Replace 'drug200.csv' with your actual file path
df = pd.read_csv(f"{extract_path}/drug200.csv")

print(f"Dataset shape: {df.shape}")
print(f"Columns: {list(df.columns)}")
print("\nFirst 5 rows:")
print(df.head())

# Check for missing values
print(f"\nMissing values per column:")
print(df.isnull().sum())

# Data preprocessing
print("\n2. Data Preprocessing")

# Check data types and unique values
print("Data types and unique values:")
for col in df.columns:
    print(f"{col}: {df[col].dtype}, unique values: {df[col].nunique()}")
    if df[col].dtype == 'object':
        print(f"  Values: {df[col].unique()}")

# Handle missing values if any
def preprocess_data(data):
    """Preprocess the data by handling missing values"""
    processed_data = data.copy()

    for col in processed_data.columns:
        if processed_data[col].isnull().sum() > 0:
            if processed_data[col].dtype == 'object':
                # Fill categorical with mode
                mode_val = processed_data[col].mode()[0]
                processed_data[col] = processed_data[col].fillna(mode_val)
```

```
                print(f"Filled {col} missing values with mode: {mode_val}")
            else:
                # Fill numerical with mean
                mean_val = processed_data[col].mean()
                processed_data[col] = processed_data[col].fillna(mean_val)
                print(f"Filled {col} missing values with mean: {mean_val:.2f}")

    return processed_data

df = preprocess_data(df)
```

```
=== Task 2: Drug Classification with Decision Tree ===

1. Loading Dataset
Dataset shape: (200, 6)
Columns: ['Age', 'Sex', 'BP', 'Cholesterol', 'Na_to_K', 'Drug']

First 5 rows:
   Age Sex      BP Cholesterol  Na_to_K   Drug
0   23   F    HIGH        HIGH   25.355  DrugY
1   47   M     LOW        HIGH   13.093  drugC
2   47   M     LOW        HIGH   10.114  drugC
3   28   F  NORMAL        HIGH    7.798  drugX
4   61   F     LOW        HIGH   18.043  DrugY

Missing values per column:
Age            0
Sex            0
BP             0
Cholesterol    0
Na_to_K        0
Drug           0
dtype: int64

2. Data Preprocessing
Data types and unique values:
Age: int64, unique values: 57
Sex: object, unique values: 2
  Values: ['F' 'M']
BP: object, unique values: 3
  Values: ['HIGH' 'LOW' 'NORMAL']
Cholesterol: object, unique values: 2
  Values: ['HIGH' 'NORMAL']
Na_to_K: float64, unique values: 198
Drug: object, unique values: 5
  Values: ['DrugY' 'drugC' 'drugX' 'drugA' 'drugB']
```

```
# (1) Use binning to transform continuous attributes into discrete values
print("\n3. Binning Continuous Attributes")

continuous_cols = df.select_dtypes(include=[np.number]).columns
continuous_cols = [col for col in continuous_cols if col != 'Drug']  # Exclude target if

print(f"Continuous columns found: {list(continuous_cols)}")
```

```
for col in continuous_cols:
    # Create 3 bins for each continuous attribute
    df[f'{col}_binned'] = pd.qcut(df[col], q=3, labels=['Low', 'Medium', 'High'], duplic
    print(f"Binned {col} into 3 categories")
    print(f"  Bin distribution: {df[f'{col}_binned'].value_counts().to_dict()}")

# Encode categorical variables
print("\n4. Encoding Categorical Variables")

categorical_cols = df.select_dtypes(include=['object']).columns
categorical_cols = [col for col in categorical_cols if col != 'Drug']  # Exclude target

for col in categorical_cols:
    df[col + '_encoded'] = pd.Categorical(df[col]).codes
    print(f"Encoded {col}: {dict(enumerate(df[col].unique()))}")

# Prepare final dataset for decision tree
# Select encoded/binned features and target
feature_cols = []
for col in df.columns:
    if col.endswith('_binned') or col.endswith('_encoded'):
        feature_cols.append(col)

# If no binned columns, use original categorical columns encoded
if not feature_cols:
    # Encode all categorical columns
    for col in df.select_dtypes(include=['object']).columns:
        if col != 'Drug':
            df[col + '_encoded'] = pd.Categorical(df[col]).codes
            feature_cols.append(col + '_encoded')

    # Include original continuous columns
    for col in continuous_cols:
        feature_cols.append(col)

# Add target column
target_col = 'Drug'
if df[target_col].dtype == 'object':
    df[target_col + '_encoded'] = pd.Categorical(df[target_col]).codes
    target_col = target_col + '_encoded'

print(f"Selected features: {feature_cols}")
print(f"Target column: {target_col}")

# Create final dataset
final_df = df[feature_cols + [target_col]].copy()
print(f"\nFinal dataset shape: {final_df.shape}")
print("Final dataset head:")
print(final_df.head())
```

```
# (2) Split data into 80% training and 20% test
print("\n5. Splitting Data (80% Train, 20% Test)")

def train_test_split(data, test_size=0.2, random_state=42):
    """Split data into training and testing sets"""
    random.seed(random_state)
    n_test = int(len(data) * test_size)

    # Randomly select test indices
    test_indices = random.sample(range(len(data)), n_test)
    train_indices = [i for i in range(len(data)) if i not in test_indices]

    train_data = data.iloc[train_indices].reset_index(drop=True)
    test_data = data.iloc[test_indices].reset_index(drop=True)

    return train_data, test_data

train_data, test_data = train_test_split(final_df)

print(f"Training set size: {len(train_data)}")
print(f"Test set size: {len(test_data)}")
```

```
3. Binning Continuous Attributes
Continuous columns found: ['Age', 'Na_to_K']
Binned Age into 3 categories
  Bin distribution: {'Low': 70, 'High': 67, 'Medium': 63}
Binned Na_to_K into 3 categories
  Bin distribution: {'Low': 67, 'High': 67, 'Medium': 66}

4. Encoding Categorical Variables
Encoded Sex: {0: 'F', 1: 'M'}
Encoded BP: {0: 'HIGH', 1: 'LOW', 2: 'NORMAL'}
Encoded Cholesterol: {0: 'HIGH', 1: 'NORMAL'}
Selected features: ['Age_binned', 'Na_to_K_binned', 'Sex_encoded', 'BP_encoded', 'Ch
Target column: Drug_encoded

Final dataset shape: (200, 6)
Final dataset head:
  Age_binned Na_to_K_binned  Sex_encoded  BP_encoded  Cholesterol_encoded  \
0        Low           High            0           0                    0
1     Medium         Medium            1           1                    0
2     Medium            Low            1           1                    0
3        Low            Low            0           2                    0
4       High           High            0           1                    0

   Drug_encoded
0             0
1             3
2             3
3             4
4             0

5. Splitting Data (80% Train, 20% Test)
```

```
Training set size: 160
Test set size: 40
```

Decision Tree Classifier

We implement a decision tree classifier from scratch using information gain and entropy. The tree is built recursively with optional pre-pruning via `max_depth` and `min_samples_split` to avoid overfitting.

```
# (3) Decision Tree Implementation
print("\n6. Decision Tree Implementation")

class DecisionTreeClassifier:
    def __init__(self, max_depth=None, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.tree = None
        self.feature_names = None
        self.target_name = None

    def calculate_entropy(self, y):
        """Calculate entropy of a target variable"""
        if len(y) == 0:
            return 0

        _, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        entropy = -np.sum(probabilities * np.log2(probabilities + 1e-10))
        return entropy

    def calculate_information_gain(self, X, y, feature_idx, threshold=None):
        """Calculate information gain for a feature"""
        parent_entropy = self.calculate_entropy(y)

        # For categorical features, split by unique values
        unique_values = np.unique(X[:, feature_idx])
        if len(unique_values) <= 1:
            return 0, None # Return 0 gain and None threshold

        best_gain = 0
        best_threshold = None

        for value in unique_values:
            left_mask = X[:, feature_idx] == value
            right_mask = ~left_mask

            if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
                continue
```

```
            left_entropy = self.calculate_entropy(y[left_mask])
            right_entropy = self.calculate_entropy(y[right_mask])

            weighted_entropy = (np.sum(left_mask) / len(y)) * left_entropy + \
                             (np.sum(right_mask) / len(y)) * right_entropy

            gain = parent_entropy - weighted_entropy

            if gain > best_gain:
                best_gain = gain
                best_threshold = value

        return best_gain, best_threshold


    def find_best_split(self, X, y):
        """Find the best feature and threshold to split on"""
        best_gain = 0
        best_feature = None
        best_threshold = None

        for feature_idx in range(X.shape[1]):
            gain, threshold = self.calculate_information_gain(X, y, feature_idx)

            if gain > best_gain:
                best_gain = gain
                best_feature = feature_idx
                best_threshold = threshold

        return best_feature, best_threshold, best_gain

    def build_tree(self, X, y, depth=0):
        """Recursively build the decision tree"""
        # Base cases
        if len(np.unique(y)) == 1:
            return {'class': y[0], 'samples': len(y)}

        if (self.max_depth is not None and depth >= self.max_depth) or \
           len(y) < self.min_samples_split:
            # Return most common class
            unique_classes, counts = np.unique(y, return_counts=True)
            majority_class = unique_classes[np.argmax(counts)]
            return {'class': majority_class, 'samples': len(y)}

        # Find best split
        best_feature, best_threshold, best_gain = self.find_best_split(X, y)

        if best_feature is None or best_gain == 0:
            # No good split found
            unique_classes, counts = np.unique(y, return_counts=True)
```

```
            majority_class = unique_classes[np.argmax(counts)]
            return {'class': majority_class, 'samples': len(y)}

        # Split data
        left_mask = X[:, best_feature] == best_threshold
        right_mask = ~left_mask

        # Recursively build subtrees
        left_tree = self.build_tree(X[left_mask], y[left_mask], depth + 1)
        right_tree = self.build_tree(X[right_mask], y[right_mask], depth + 1)

        return {
            'feature': best_feature,
            'threshold': best_threshold,
            'left': left_tree,
            'right': right_tree,
            'samples': len(y)
        }

    def fit(self, X, y):
        """Train the decision tree"""
        if isinstance(X, pd.DataFrame):
            self.feature_names = X.columns.tolist()
            X = X.values

        if isinstance(y, pd.Series):
            self.target_name = y.name
            y = y.values

        self.tree = self.build_tree(X, y)
        return self

    def predict_single(self, x, tree):
        """Predict a single sample"""
        if 'class' in tree:
            return tree['class']

        if x[tree['feature']] == tree['threshold']:
            return self.predict_single(x, tree['left'])
        else:
            return self.predict_single(x, tree['right'])

    def predict(self, X):
        """Predict multiple samples"""
        if isinstance(X, pd.DataFrame):
            X = X.values

        predictions = []
        for x in X:
            predictions.append(self.predict_single(x, self.tree))
```

```
        return np.array(predictions)

    def print_tree(self, tree=None, depth=0):
        """Print the decision tree structure"""
        if tree is None:
            tree = self.tree

        indent = "  " * depth

        if 'class' in tree:
            print(f"{indent}Predict: {tree['class']} (samples: {tree['samples']})")
        else:
            feature_name = self.feature_names[tree['feature']] if self.feature_names els
            print(f"{indent}If {feature_name} == {tree['threshold']}:")
            self.print_tree(tree['left'], depth + 1)
            print(f"{indent}Else:")
            self.print_tree(tree['right'], depth + 1)

# Train the decision tree
print("Training Decision Tree...")

# Prepare training data
X_train = train_data[feature_cols]
y_train = train_data[target_col]

# Initialize and train the decision tree
dt = DecisionTreeClassifier(max_depth=5, min_samples_split=5)
dt.fit(X_train, y_train)

print("Decision Tree trained successfully!")

# Print tree structure
print("\nDecision Tree Structure:")
dt.print_tree()
```

```
6. Decision Tree Implementation
Training Decision Tree...
Decision Tree trained successfully!

Decision Tree Structure:
If BP_encoded == 0:
  If Na_to_K_binned == High:
    Predict: 0 (samples: 23)
  Else:
    If Age_binned == High:
      If Na_to_K_binned == Low:
        Predict: 2 (samples: 6)
      Else:
        If Sex_encoded == 0:
          Predict: 2 (samples: 3)
        Else:
          Predict: 2 (samples: 3)
```

```
        Else:
          If Na_to_K_binned == Low:
            If Age_binned == Low:
              Predict: 1 (samples: 6)
            Else:
              Predict: 1 (samples: 7)
          Else:
            If Cholesterol_encoded == 0:
              Predict: 1 (samples: 6)
            Else:
              Predict: 0 (samples: 9)
    Else:
      If Na_to_K_binned == High:
        Predict: 0 (samples: 27)
      Else:
        If Cholesterol_encoded == 0:
          If BP_encoded == 1:
            If Na_to_K_binned == Low:
              Predict: 3 (samples: 9)
            Else:
              Predict: 3 (samples: 7)
          Else:
            If Na_to_K_binned == Low:
              Predict: 4 (samples: 8)
            Else:
              Predict: 4 (samples: 11)
        Else:
          If Age_binned == Medium:
            If Na_to_K_binned == Low:
              Predict: 4 (samples: 8)
            Else:
              Predict: 4 (samples: 8)
          Else:
            Predict: 4 (samples: 19)
```

Prediction and Accuracy Evaluation

We test the decision tree on the test set and evaluate its performance using accuracy. Additionally, we print predicted vs actual results and class distribution to analyze misclassifications.

```
# (4) Test the classifier
print("\n7. Testing the Classifier")

# Prepare test data
X_test = test_data[feature_cols]
y_test = test_data[target_col]

# Make predictions
predictions = dt.predict(X_test)

# Calculate accuracy
```

```python
accuracy = np.mean(predictions == y_test.values)
print(f"Test Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")

# Display classification results
print("\nClassification Results (first 10 samples):")
print("Predicted | Actual")
print("-" * 18)
for i in range(min(10, len(predictions))):
    print(f"{predictions[i]:>9} | {y_test.iloc[i]:>6}")

# Show confusion matrix-like statistics
unique_classes = np.unique(np.concatenate([predictions, y_test.values]))
print(f"\nClass distribution in test set:")
for cls in unique_classes:
    actual_count = np.sum(y_test.values == cls)
    predicted_count = np.sum(predictions == cls)
    print(f"Class {cls}: Actual={actual_count}, Predicted={predicted_count}")

print(f"\nTotal test samples: {len(y_test)}")
print(f"Correctly classified: {np.sum(predictions == y_test.values)}")
print(f"Incorrectly classified: {np.sum(predictions != y_test.values)}")

print("\n=== Task 2 Completed Successfully ===")
```

```
7. Testing the Classifier
Test Accuracy: 0.8500 (85.00%)

Classification Results (first 10 samples):
Predicted | Actual
------------------
        0 |      0
        0 |      0
        4 |      0
        0 |      0
        2 |      2
        0 |      0
        0 |      0
        4 |      4
        0 |      0
        0 |      0

Class distribution in test set:
Class 0: Actual=23, Predicted=17
Class 1: Actual=2, Predicted=2
Class 2: Actual=4, Predicted=5
Class 3: Actual=3, Predicted=5
Class 4: Actual=8, Predicted=11

Total test samples: 40
Correctly classified: 34
Incorrectly classified: 6

=== Task 2 Completed Successfully ===
```

END OF ASSIGNMENT 1 TASK 2