YUNUS EMRE ALTUĞ 2019400057

Information About Dataset

This dataset contains a cleaned version of this dataset from UCI machine learning repository on credit card approvals. Missing values have been filled and feature names and categorical names have been inferred, resulting in more context and it being easier to use.

- *Gender:*
- *0=Female, 1=Male*

- *Age:*
- *Age in years*

- *Debt:*
- *Outstanding debt*

- *Married:*
- *Married, 0=Single/Divorced/etc, 1=Married*

- *BankCustomer:*
- *Bank customer, 0=does not have a bank account, 1=has a bank account*

- *Industry:*
- *Industry - job sector of current or most recent job*

- *Ethnicity:*
- *Ethnicity*

- *YearsEmployed:*
- *Years employed*

- *PriorDefault*
- *Prior default, 0=no prior defaults, 1=prior default*

- *Employed*
- *Employed, 0=not employed, 1=employed*

- *CreditScore*
- *Credit score (this feature has been scaled)*

- *DriversLicense*
- *Drivers license, 0=no license, 1=has license*

- *Citizen*
- *Citizenship, either ByBirth, ByOtherMeans or Temporary*

- *ZipCode*
- *ZipCode (5 digit number)*

- *Income*
- *Income (this feature has been scaled)*

- *Approved*
- *Approved, 0=not approved, 1=approved*

The aim of decision tree is finding the credit card approvals. Dataset includes fields above. The last field is the field which is aimed to be classified. You can visit from there:

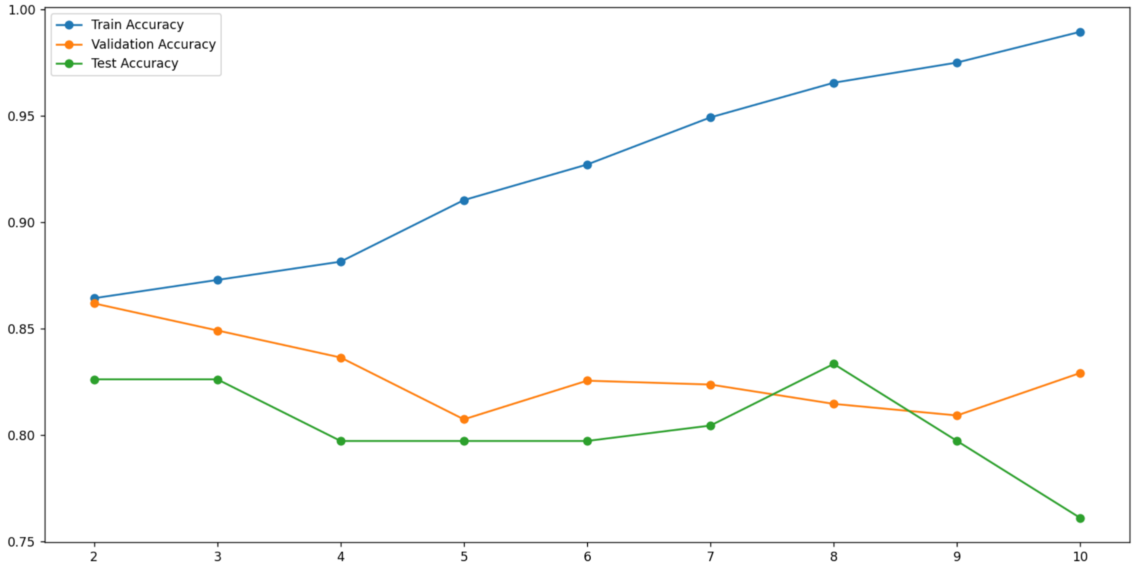
https://www.kaggle.com/datasets/samuelcortinhas/credit-card-approval-clean-data?select=clean_dataset.csv

5-Cross Validation

The dataset is divided into temp data and train data with 20% after shuffling. Then, the temp data is shuffled and divided into five parts. Each of part is validation data and the remaining is training in 5-cross validation. For each fold, the tree is structured, and validation score is calculated. With these five validation score, the average is calculated. Thus, for each max depth value, I find best tree.

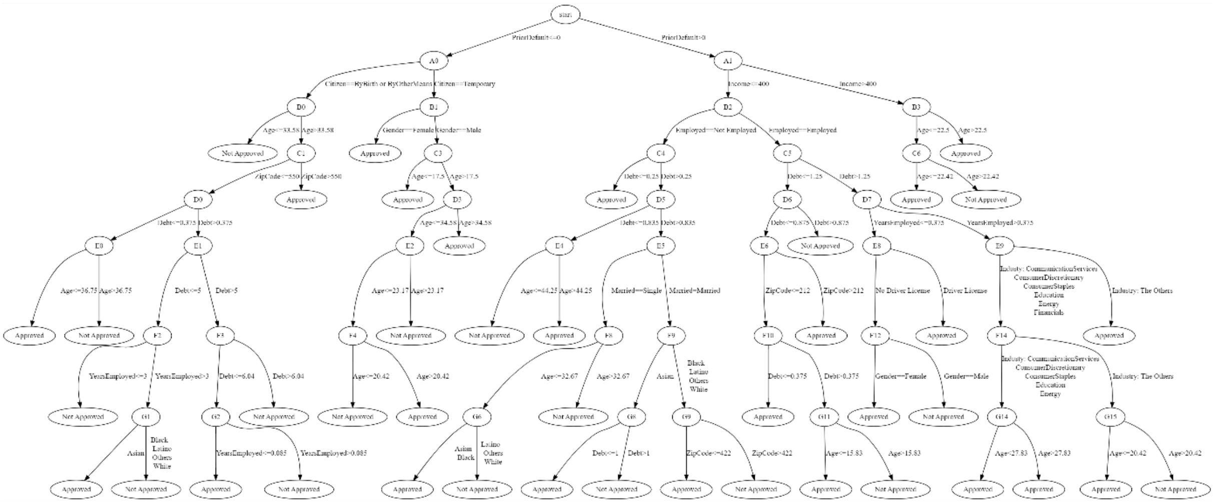
At the end I take these trees and predict for each max depth size in test data.

Error Plots



The training accuracy is raised as expected. However, test and validation scores are not raised.

Decision Tree



Source Code

Main Code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv('clean_dataset.csv')


# Assume the last column is the target variable and the rest are features
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

#Data is divided
```

```
k = 5
np.random.seed(42)

temp_x, temp_y, test_X, test_y = train_test_split(X, y, test_size=0.2)

depths = [2,3,4,5,6,7,8,9,10]
train_accuracies = []
val_accuracies = []
test_accuracies = []
best_tree = None
best_val=0
best_test_val=0
best_test_tree=None
```

```
# Modify max_depth to see how it affects the accuracy
for depth in depths:
    avg_train_accuracy=0
    avg_val_accuracy=0
    best_tree=None
    best_val=0
    # Perform k-fold cross-validation
    for train_X, train_y, val_x, val_y in k_fold_split(temp_x, temp_y, k):
        # Build the tree using the training data
        tree = build_tree(train_X, train_y, max_depth=depth)
        # Make predictions on training and validation data
        predictions_val = [predict(tree, x) for x in val_x]
        predictions_train = [predict(tree, x) for x in train_X]
        # Calculate training and validation accuracy
        avg_train_accuracy+=accuracy(train_y, predictions_train)
        val_accuracy=accuracy(val_y, predictions_val)
        avg_val_accuracy+=val_accuracy

        # Save the best tree
        if val_accuracy>best_val:
            best_val=val_accuracy
            best_tree=tree
    # Save the accuracies for visualization
    avg_train_accuracy/=k
    avg_val_accuracy/=k
    val_accuracies.append(avg_val_accuracy)
    train_accuracies.append(avg_train_accuracy)

    # Make predictions on test data
    predictions_test = [predict(best_tree, x) for x in test_X]
    test_accuracy = accuracy(test_y, predictions_test)
    test_accuracies.append(test_accuracy)

    # Save the best test accuracy
    if test_accuracy>best_test_val:
        best_test_val=test_accuracy
        best_test_tree=best_tree
```

```python
# Visualize the accuracies
plt.plot(depths, train_accuracies, marker='o', label="Train Accuracy")
plt.plot(depths, val_accuracies, marker='o' , label="Validation Accuracy")
plt.plot(depths, test_accuracies, marker='o', label="Test Accuracy")
plt.legend()
plt.show()

print("Best Test Accuracy: ",best_test_val)

# This part shows nodes. It is not necessary to run this part.
"""
stack=[]
best_test_tree.depth=0
stack.append(best_test_tree)
columns=list(data.columns)
while len(stack)>0:
    current_node=stack[0]
    if current_node.value is None:
        if current_node.left is not None:
            current_node.left.depth=current_node.depth+1
            stack.append(current_node.left)
        if current_node.right is not None:
            current_node.right.depth=current_node.depth+1
            stack.append(current_node.right)
    try:
        print(columns[current_node.feature_index],current_node.threshold, current_node.depth, current_node.value)
    except:
        print(None,current_node.threshold, current_node.depth, current_node.value)
    stack.pop(0)


"""
```

```python
# Define the decision tree node
class Node:
    def __init__(self, feature_index=None, threshold=None, value=None, left=None, right=None, depth=-1):
        self.feature_index = feature_index
        self.threshold = threshold
        self.value = value
        self.left = left
        self.right = right
        self.depth=depth
```

```python
# Recursively build the decision tree
def build_tree(X, y, depth=0, max_depth=None):
    # Check for stopping criteria, if none, build the tree recursively. Max_depth is the maximum depth of the tree.
    # If max_depth is None, the tree will expand until all leaves are pure.
    # np.unique(y) returns the unique values in y, len(np.unique(y)) returns the number of unique values in y.
    # If len(np.unique(y)) is 1, all samples in the node belong to the same class, so the node is pure and we can stop.
    if depth == max_depth or len(np.unique(y)) == 1:
        # Leaf node
        unique_classes, counts = np.unique(y, return_counts=True)
        value = unique_classes[np.argmax(counts)]
        # Return a node with the most common class value in the leaf node
        return Node(value=value)
    # we are deriving new nodes from the root node, we need to find the best split for the root node.
    result = find_best_split(X, y)
    # result is not a tuple of (feature_index, threshold), we haven't found a suitable split, so we stop.
    if result is None:
        feature_index, threshold = None, None
        # No suitable split found
        unique_classes, counts = np.unique(y, return_counts=True)
        value = unique_classes[np.argmax(counts)]
        return Node(value=value)
    else:
        # We have found a suitable split, so we can build the tree recursively.
        feature_index, threshold = result

    # Split dataset into left and right
    left_X, left_y, right_X, right_y = split_dataset(X, y, feature_index, threshold)

    # Recursively build the left and right subtree
    left_node = build_tree(left_X, left_y, depth + 1, max_depth)
    right_node = build_tree(right_X, right_y, depth + 1, max_depth)

    return Node(feature_index=feature_index, threshold=threshold, left=left_node, right=right_node)
```

```python
# Find the best split for a node
def find_best_split(X, y):
    best_information_gain = 0
    best_split = None
    # Iterate over all features and possible thresholds to find the best split
    for feature_index in range(X.shape[1]):
        thresholds = np.unique(X[:, feature_index])
        for threshold in thresholds:
            # Find information gain
            information_gain = calculate_information_gain(X, y, feature_index, threshold)
            if information_gain > best_information_gain:
                # Update best split
                best_information_gain = information_gain
                best_split = (feature_index, threshold)

    return best_split
```

```python
# Calculate information gain
def calculate_information_gain(X, y, feature_index, threshold):
    # Calculate parent entropy. This is the entropy before splitting, which is the entropy of the entire dataset.
    total_entropy = calculate_entropy(y)
    left_X, left_y, right_X, right_y = split_dataset(X, y, feature_index, threshold)
    left_entropy = calculate_entropy(left_y)
    right_entropy = calculate_entropy(right_y)
    weighted_entropy = (len(left_y) / len(y)) * left_entropy + (len(right_y) / len(y)) * right_entropy
    information_gain = total_entropy - weighted_entropy
    return information_gain
```

```python
# Calculate entropy
def calculate_entropy(y):
    # Calculate the entropy of a label sequence
    unique_classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    entropy = -np.sum(probabilities * np.log2(probabilities))
    return entropy

# Split dataset based on a feature and threshold
def split_dataset(X, y, feature_index, threshold):
    left_mask = X[:, feature_index] <= threshold
    right_mask = ~left_mask
    return X[left_mask], y[left_mask], X[right_mask], y[right_mask]
```

```python
# Split dataset into k folds for cross-validation. This function returns the training and validation data for each fold.
def k_fold_split(X, y, k):
    fold_size = len(X) // k
    indices = np.arange(len(X))
    np.random.shuffle(indices)

    for i in range(k):
        test_indices = indices[i * fold_size: (i + 1) * fold_size]
        train_indices = np.concatenate([indices[:i * fold_size], indices[(i + 1) * fold_size:]])
        #the function returns the training and testing data for each fold, in this implement I give k as 5
        yield X[train_indices], y[train_indices], X[test_indices], y[test_indices]


# Split dataset into training and testing sets before k-fold cross-validation
def train_test_split(X, y, test_size=0.2):
    fold_size = int(len(X) * test_size)
    indices = np.arange(len(X))
    np.random.shuffle(indices)

    test_indices = indices[:fold_size]
    temp_indices = indices[fold_size:]

    return X[temp_indices], y[temp_indices], X[test_indices], y[test_indices]
```

```python
# Predict using the decision tree
def predict(tree, x):
    if tree.value is not None:
        return tree.value

    if x[tree.feature_index] <= tree.threshold:
        return predict(tree.left, x)
    else:
        return predict(tree.right, x)

# Evaluate the accuracy of predictions
def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)
```