Applying the Decision Tree Learning Algorithm with Entropy to an *Anime* Dataset

Cmpe 480 - Introduction to Artificial Intelligence

Fall 2023

Asude Ebrar Kızıloğlu

2019400009

The Dataset

The Anime list dataset from Kaggle is utilized in this project. This dataset offers detailed information about anime of various genres that are available on myanimelist website. This dataset consists of 3168 entries and 14 data columns. I only utilized some of the available columns. The columns that are used in this decision tree learning algorithm are as follows:

- `source`: Determines the source of the Anime. 15 different sources are available: *['Manga', '4-koma manga', 'Light novel', 'Game', 'Original', 'Web manga', 'Card game', 'Novel', 'Other', 'Radio', 'Visual novel', 'Book', 'Mixed media', 'Picture book', 'Music']*
- `demographic`: Determines the intended age group for the anime. Five values are available: *['Shounen', 'Seinen', 'Shoujo', 'Josei', 'Kids']*
- `status`: either *Finished* or *Airing*.
- `eps_avg_duration_in_min`: Determines the average duration of each episode in minutes. The data takes values between 0 and 30 under this data column.
- `rating`: The average ratings of Anime that is ranging between 0 and 10. This is the result (Y) values column.
- I modified the `eps_avg_duration_in_min` and `rating` columns. The episode durations were averaged as follows:

```
if  eps_avg_duration_in_min  < 10:
    eps_duration = 5
elif  eps_avg_duration_in_min  < 20:
    eps_duration = 15
elif  eps_avg_duration_in_min  < 30:
    eps_duration = 25
elif  eps_avg_duration_in_min  == 30:
    eps_duration = 30
```

Then the ratings are split into two categories such that an anime is considered 'GOOD' if the rating > 7/10, and 'BAD' otherwise. This binary classification allowed the decision tree to output more pure leaf nodes at the end. Hence, the dataset that is utilized in the decision tree algorithm is in the following format:

	source	demographic	status	eps_duration	star_ratings
223	Manga	Shounen	Finished	25	GOOD
1288	Manga	Shoujo	Finished	25	GOOD
899	Manga	Seinen	Finished	25	GOOD
7	Manga	Shounen	Finished	25	GOOD
1525	Original	Shoujo	Finished	25	BAD
1494	4-koma manga	Shoujo	Finished	5	BAD
1664	Original	Kids	Finished	5	BAD
1107	Manga	Seinen	Finished	25	GOOD

After clearing the data entries with null data, 1604 data entries were left. This data is then split into training and testing portions. The test dataset was 10% of the whole set.

Applying the Decision Tree Learning Algorithm

The algorithm utilizes some objects:

- `NodeQuestion`: This object holds a question per decision node. The question has the following structure: **Is** *'question_name'* [**==** | **>=**] *'value'***?**
- `DecisionTreeNode`: This object holds the nodes of the tree. Each node can be either the decision or the leaf node. The decision nodes store a question and two children nodes. The leaf nodes store the prediction values.
- `DecisionTreeSolver`: This object consists of almost all utility functions necessary to build the tree, to choose the best split at each branch of the tree, and to compute the information gain through entropy.

I control the growth of the decision tree with two parameters:

- `min_samples_split`: determines the number of samples a leaf should include at maximum. Set to 10 for this implementation.
- `max_depth`: determines the maximum depth of the tree. Set to 5 for this implementation.

Considering these two constraints, if the tree includes some impure leaf nodes upon being built, the leaf adapts the more common rating value ['GOOD' or 'BAD'] as its final prediction value. This decision causes a huge drop in the accuracy of the model. Yet, on the other hand, each leaf node at the end should have one certain value, so such a decision should be made.

Utilizing 5-fold Cross-Validation

For the training dataset (which has 90% * 1604 = 1443 entries), I split it into 5 parts and took the validation dataset as a different data set at each run. This prevented the algorithm from depending only on a single train-test split. 5-fold cross-validation provides a more reliable estimate of the model's performance by averaging over 5 folds, instead of relying on one train-test split. Thus, the model's performance becomes less dependent on the specific instances selected for training and testing.

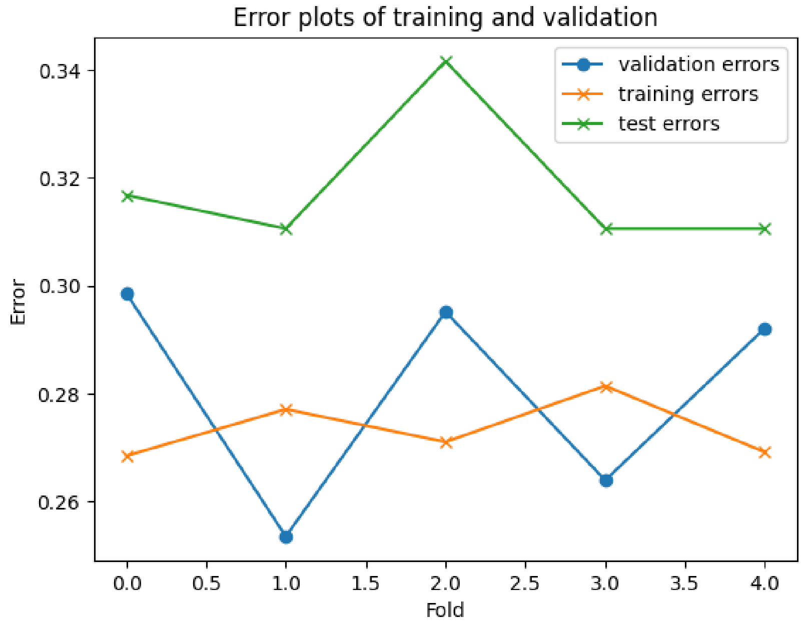
It is also beneficial to gain more insight into the variability of the model's performance. The model might perform with very high accuracy for a certain test-train split but might perform very poorly for a different dataset split. By utilizing 5-fold cross-validation, I observe that the model is in fact performing stably with different training-validation dataset combinations. Here, I provide the model accuracies for the training and validation datasets for each of 5 folds:

```
Fold 1: Validation Accuracy: 0.701 and Training Accuracy: 0.732
Fold 2: Validation Accuracy: 0.747 and Training Accuracy: 0.723
Fold 3: Validation Accuracy: 0.705 and Training Accuracy: 0.729
Fold 4: Validation Accuracy: 0.736 and Training Accuracy: 0.719
Fold 5: Validation Accuracy: 0.708 and Training Accuracy: 0.731
Average Validation Accuracy Across Folds: 0.719
Average Training Accuracy Across Folds: 0.727
Best Validation Accuracy Across Folds: 0.7465
The best fold number is Fold 1
```

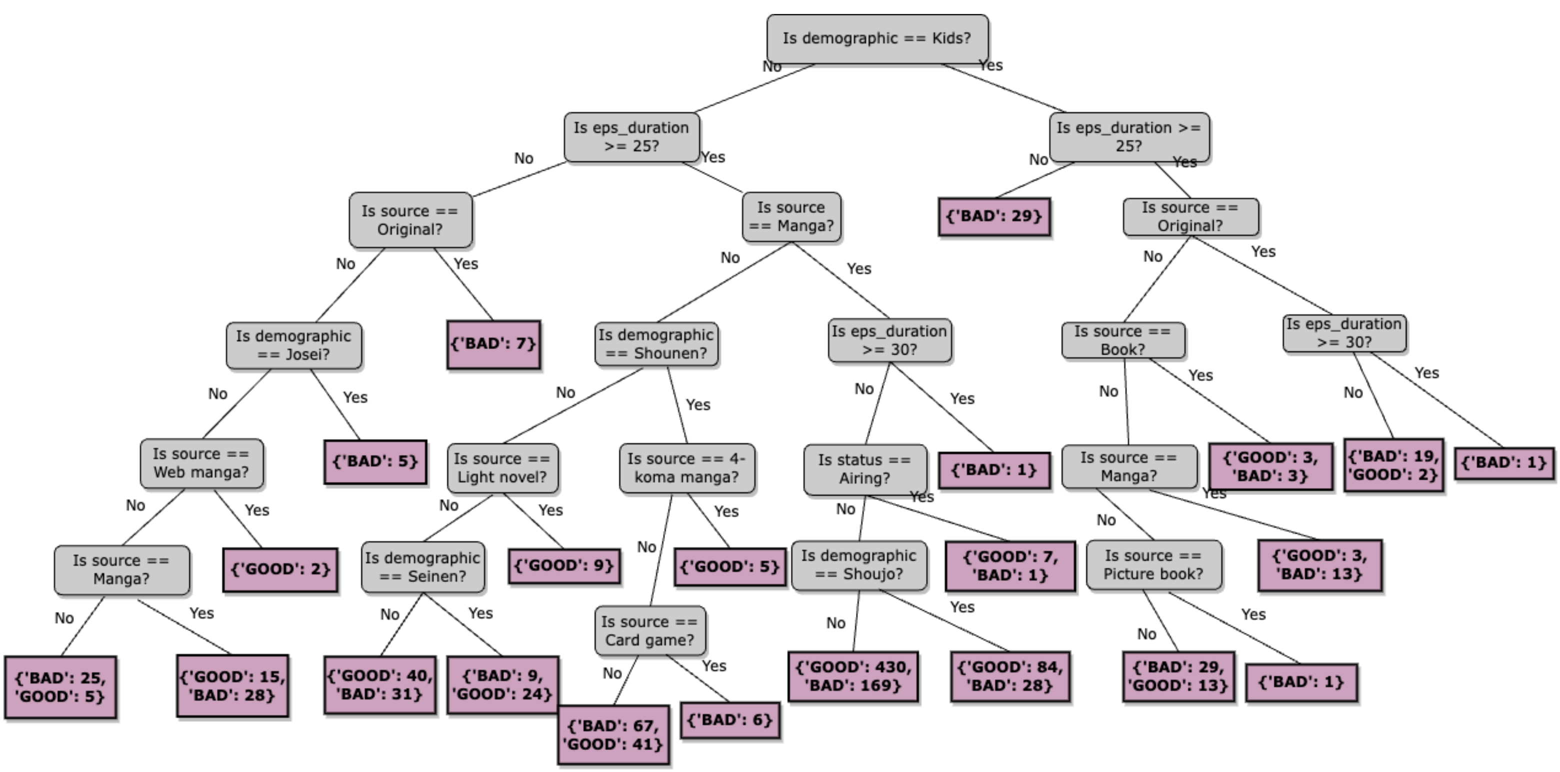
```
Test accuracy for the best fold is 0.689
```

Error Plots for Training, Validation, and Test datasets

The following plot displays the error values for the training, validation, and test datasets for each of the 5 folds. The test sets have considerably higher error rates than others as expected. The errors of validation and training sets are similar even though the validation is not particularly used in the training.



Final Decision Tree



Source Code [with comments]

```
# -*- coding: utf-8 -*-
"""Cmpe-480-Decision-Tree-Learning.ipynb
Automatically generated by Colaboratory.
Original file is located at
   https://colab.research.google.com/drive/10WqvmP85BS67UTadHEBuTtmtFi4Z8FCV
"""

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Data Preparation
data = pd.read_csv("anime-list 4-sept-2022 update.csv")
data = data.dropna()
# Get rid of the unnecessary columns and missing data entries
col_names = ['rating', 'source', 'demographic', 'status', 'eps_avg_duration_in_min']
df = data[col_names]

# Fix the 'eps_avg_duration_in_min' values to the average durations. The ranges are as
follows:
# [0, 10)   -> 5
# [10, 20)  -> 15
# [20, 30)  -> 25
# 30 -> 30
eps_durations = np.array([ (10 * int(1 + (val//10))) - 5 if val < 30.0 else 30 for val
in df['eps_avg_duration_in_min'].values])
eps_durations.min()
df.loc[:, 'eps_duration'] = eps_durations
df = df.drop(columns='eps_avg_duration_in_min')
# Fix the 'rating' values such that an anime is considered GOOD if the rating is over
7/10 and BAD otherwise.
# [0, 7)  -> 'BAD'
# [7, 10) -> 'GOOD'
mapping_dict = {1: 'BAD', 2: 'GOOD'}
star_ratings = np.array([int(1 + (val) // 7) for val in df['rating'].values])
star_ratings = np.vectorize(mapping_dict.get)(star_ratings).astype(object)
df.loc[:, 'star_ratings'] = star_ratings
df = df.drop(columns='rating')
df.info()
```

```python
# Following part consists of some utility functions and necessary objects to build the
Tree:
def class_counts(dataset):
   # Counts the number of each type of label in the dataset.
   counts = {}  # label -> count.
   for data in dataset:
       # the label is adjusted to be the last column
       label = data[-1]
       if label not in counts:
           counts[label] = 0
       counts[label] += 1
   return counts

question_header = ['source', 'demographic', 'status', 'eps_duration']
class NodeQuestion:
   # A Question is used to split a dataset.
   # Records a 'column index' and a'column value'.
   # Each Decision Node includes a Question.
   def __init__(self, index, value):
       self.index = index
       self.value = value

   def match(self, example):
       # Compares the given feature value to the feature value in this question.
       feature = example[self.index]
       if self.__is_numeric(feature):
           return feature >= self.value
       else:
           return feature == self.value

   def __repr__(self):
       # A helper method to print the question in a readable format.
       condition = "=="
       if self.__is_numeric(self.value):
           condition = ">="
       return "Is %s %s %s?" % (
           question_header[self.index], condition, str(self.value))

   def __is_numeric(self, value):
     return isinstance(value, int) or isinstance(value, float)
```

```
class DecisionTreeNode:
   # Nodes can be either a Decision or a Leaf Node.
   # The field 'is_leaf' determines the type of the Node.
   # A Decision Node includes a question, and two child nodes.
   # A Leaf node classifies data. It contains a dictionary of label -> count in the
'predictions' field.
   def __init__(self, question=None, true_child=None, false_child=None, is_leaf=False,
rows=None):
       self.question = question
       self.true_child = true_child
       self.false_child = false_child
       self.is_leaf = is_leaf
       if is_leaf:
         self.predictions = class_counts(rows) if is_leaf else None
         self.majority_rating = max(self.predictions, key=lambda k:
self.predictions[k]) if is_leaf else None
         self.count = sum(value for value in self.predictions.values())
       else:
         self.predictions, self.majority_rating, self.count = None, None, None
   # def print_leaf_node(self, indent):
   #   print (indent + "Predict", self.predictions)

# Decision Tree Class:
class DecisionTreeSolver():
   def __init__(self, min_samples_split=2, max_depth=5):
       self.root = None
       self.min_samples_split = min_samples_split
       self.max_depth = max_depth

   def fit(self, X, Y):
     dataset = np.concatenate((X, Y), axis=1)
     self.root = self.build_tree(dataset)
     return self.root

   def build_tree(self, dataset, curr_depth=0):
       # Builds the tree.
       X = dataset[:,:-1]
       Y = dataset[:,-1]
       num_samples, num_features = np.shape(X)
       if num_samples >= self.min_samples_split and curr_depth <= self.max_depth:
         # Try partitioing the dataset on each of the unique data group,
         # calculate the information gain,
```

```
        # and return the question that produces the highest gain.
        gain, question = self.find_best_split(dataset)
        # Base case: no further info gain: Return a leaf
        if gain == 0:
            return DecisionTreeNode(is_leaf=True, rows=dataset)

        true_rows, false_rows = self.partition(dataset, question)
        # Build the true branch.
        true_child = self.build_tree(np.array(true_rows), curr_depth+1)
        # Build the false branch.
        false_child = self.build_tree(np.array(false_rows), curr_depth+1)

        # Return a Decision node.
        return DecisionTreeNode(question, true_child, false_child)
      return DecisionTreeNode(is_leaf=True, rows=dataset)

  def partition(self, dataset, question):
    # For each row in the dataset, check if it matches the given question. If
    # so, add it to 'true dataset', otherwise, add it to 'false dataset'.
    true_dataset, false_dataset = [], []
    for data in dataset:
        if question.match(data):
            true_dataset.append(data)
        else:
            false_dataset.append(data)
    return true_dataset, false_dataset

  def find_best_split(self, dataset):
    # Find the best question to ask by iterating over every feature
    # and calculating the information gain.
    best_gain = 0
    best_question = None
    parent_entropy = self.entropy(dataset)
    n_features = len(dataset[0]) - 1

    for col in range(n_features):  # for each feature
        values = set([row[col] for row in dataset])
        for val in values:  # for each value
            question = NodeQuestion(col, val)
            # try splitting:
            true_dataset, false_dataset = self.partition(dataset, question)
            # Skip this split if it doesn't divide the dataset
```

```
             if len(true_dataset) == 0 or len(false_dataset) == 0:
                 continue

             # Calculate the information gain from this split
             gain = self.info_gain(true_dataset, false_dataset, parent_entropy)

             # Use '>=':
             if gain >= best_gain:
                 best_gain, best_question = gain, question

     return best_gain, best_question

   def entropy(self, dataset):
     # Calculates entropy for a given group of data
     label_counts = class_counts(dataset)
     entropy = 0
     total_count = float(len(dataset))
     for label in label_counts:
         prob = label_counts[label] / total_count
         entropy -= prob * np.log2(prob)
     return entropy

   def info_gain(self, left_dataset, right_dataset, parent_entropy):
       # Information Gain.
       # Entropy of parent minus the weighted entropy of two child nodes
       p = float(len(left_dataset)) / (len(left_dataset) + len(right_dataset))
       return parent_entropy - p * self.entropy(left_dataset) - (1 - p) *
self.entropy(right_dataset)

   def print_tree(self, node, indent="", depth=0):
     # Prints the decision tree model in a readable format

     # Base case: Leaf node
     if node.is_leaf:
       print (indent + "Predict", node.predictions)
       return

     # Print the question of the decision node
     print (indent + f'Depth {depth}: ' + str(node.question))

     # Print recursively the true branch
     print (indent + '--> True:')
```

```
        self.print_tree(node.true_child, indent + "  ", depth + 1)

        # Print recursively the false branch
        print (indent + '--> False:')
        self.print_tree(node.false_child, indent + "  ", depth + 1)

    def predict(self, X):
        # Predict a new dataset
        preditions = [self.make_prediction(x, self.root) for x in X]
        return preditions

    def make_prediction(self, x, node):
        # Predict a single data point
        if node.predictions != None:  # leaf node
          majority_rating = max(node.predictions, key=lambda k: node.predictions[k])
          return majority_rating
        if node.question.match(x):
          return self.make_prediction(x, node.true_child)
        else:
          return self.make_prediction(x, node.false_child)

def train_decision_tree(X_train, Y_train, min_samples_split=2, max_depth=5):
  solver = DecisionTreeSolver(min_samples_split, max_depth)
  my_tree = solver.fit(X_train, Y_train)
  return solver

def predict_decision_tree(trained_model, X_test):
  return trained_model.predict(X_test)

def calculate_accuracy(val_predictions, Y_val):
  true = 0
  false = 0
  for index, value in enumerate(val_predictions):
    if value == Y_val[index]:
      true += 1
    else:
      false += 1
  if true + false == 0:
    return -1
  accuracy = (true) / (true + false)
  return accuracy
```

```
# Split the data into X and Y values, and train and test parts:
Y = df.iloc[:, -1].values.reshape(-1,1)
X = df.drop(columns=df.columns[-1]).values
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1,
random_state=41)
X = X_train
Y = Y_train
print(X.shape + Y.shape + X_test.shape + Y_test.shape)

# Perform 5-fold cross-validation
num_samples = len(X)
fold_size = num_samples // 5
print(f'num of samples is {num_samples} anf fold size is {fold_size}')
val_accuracies = []
training_accuracies = []
test_accuracies = []
best_val_accuracy = 0
best_tree = None
best_fold = -1
for fold in range(5):
   start_idx = fold * fold_size
   end_idx = (fold + 1) * fold_size if fold < 4 else num_samples
   # Extract the training and validation sets for this fold
   X_train = np.concatenate([X[:start_idx], X[end_idx:]], axis=0)
   Y_train = np.concatenate([Y[:start_idx], Y[end_idx:]], axis=0)
   X_val = X[start_idx:end_idx]
   Y_val = Y[start_idx:end_idx]

   # Train your decision tree on the training set
   trained_model = train_decision_tree(X_train, Y_train, min_samples_split=10,
max_depth=5)
   # Evaluate the performance on the training set
   training_predictions = predict_decision_tree(trained_model, X_train)
   training_accuracy = calculate_accuracy(training_predictions, Y_train)
   training_accuracies.append(training_accuracy)
   # Evaluate the performance on the validation set
   val_predictions = predict_decision_tree(trained_model, X_val)
   val_accuracy = calculate_accuracy(val_predictions, Y_val)
   val_accuracies.append(val_accuracy)

   # Evaluate the performance on the test set
   test_predictions = predict_decision_tree(trained_model, X_test)
```

```
    test_accuracy = calculate_accuracy(test_predictions, Y_test)
    test_accuracies.append(test_accuracy)
    if val_accuracy > best_val_accuracy:
      best_val_accuracy = val_accuracy
      best_tree = trained_model
      best_fold = fold
    print(f"Fold {fold + 1}: Validation Accuracy: {val_accuracy:.3f} and Training
Accuracy: {training_accuracy:.3f}")
average_val_accuracy = np.mean(val_accuracies)
average_training_accuracy = np.mean(training_accuracies)
print(f"Average Validation Accuracy Across Folds: {average_val_accuracy:.3f}")
print(f"Average Training Accuracy Across Folds: {average_training_accuracy:.3f}")
print(f"Best Validation Accuracy Across Folds: {best_val_accuracy:.4f}")
print(f'The best fold number is Fold {best_fold}')
# Evaluate the performance on the test set
test_predictions = predict_decision_tree(best_tree, X_test)
test_accuracy = calculate_accuracy(test_predictions, Y_test)
print(f'Test accuracy for the best fold is {test_accuracy:.3f}')
print(f'Test accuracies are {test_accuracies}')

print('Best Tree is')
best_tree.print_tree(best_tree.root, "", 0)

# Error plots for training, validation and test datasets.
import matplotlib.pyplot as plt
val_errors = [1.0 - acc for acc in val_accuracies]
training_errors = [1.0 - acc for acc in training_accuracies]
test_errors = [1.0 - acc for acc in test_accuracies]
plt.plot(val_errors, label='validation errors', marker='o')
plt.plot(training_errors, label='training errors', marker='x')
plt.plot(test_errors, label='test errors', marker='x')
plt.xlabel('Fold')
plt.ylabel('Error')
plt.title('Error plots of training and validation')
plt.legend()
plt.show()
```