David Rosenfeld
UCF ID: 1550340

Assignment 1

1) Decision Tree Basics
   a) Given A = 5 and D = 3, we calculate permutations P(5, 3), meaning that 3 are chosen from a set of 5 where order matters:

$$P(5,3) = \frac{5!}{(5-3)!} = \frac{120}{2} = 60$$

   For the general case A >> D, the formula is:

$$P(A,D) = \frac{A!}{(A-D)!}$$

   b) Here is the output for the tree of depth 1:

```
{'Finished HMK':
  {
    1: {'Entropy': 0.863120568566631, 'Positives': 5, 'Negatives': 2},
    0: {'Entropy': 0.9852281360342516, 'Positives': 3, 'Negatives': 4}
  }
}
```

   Here is the information gain for all attributes at depth 1:

```
{
  'Early': 0.02024420715375619,
  'Finished HMK': 0.06105378373381032,
  'Senior': 0.011265848648557286,
  'Likes Coffee': 0.03914867190307081,
  'Liked The Last Jedi': 0.0013397424044413464
}
```

   Here is the output for the tree of depth 2:

```
{'Finished HMK':
  {
  1: {
    'Entropy': 0.863120568566631,
    'Positives': 5,
    'Negatives': 2,
    'Next Split': 'Early'},
  0: {
    'Entropy': 0.9852281360342516,
    'Positives': 3,
    'Negatives': 4,
    'Next Split': 'Likes Coffee'}
  }
}
```

   (See next page for information gain)

Information gain at depth 2 for Finished HMK = 1:

```
{
  'Early': 0.41379956460568024,
  'Senior': 0.2916919971380598,
  'Likes Coffee': 0.18385092540042136,
  'Liked The Last Jedi': 0.2916919971380598
}
```

Information gain at depth 2 for Finished HMK = 0:

```
{
  'Early': 0.02024420715375619,
  'Senior': 0.12808527889139454,
  'Likes Coffee': 0.46956521111470706,
  'Liked The Last Jedi': 0.02024420715375619
}
```

The code for this is in the appendix at the end of the file.

c) Here is the output when the tree is extended to a depth of 3:

```
{'Finished HMK':
  {
  1: {
    'Entropy': 0.863120568566631,
    'Positives': 5,
    'Negatives': 2,
    'Next Split': 'Early',
    'Depth 3 Splits': {
      'Early': {
        1: None,
        0: 'Senior'}
     }
  },
  0: {
    'Entropy': 0.9852281360342516,
    'Positives': 3,
    'Negatives': 4,
    'Next Split': 'Likes Coffee',
    'Depth 3 Splits': {
      'Likes Coffee': {
         0: 'Senior',
         1: None}
     }
     }
  }
}
```

Here is the information gain at depth 3:

Information gain at depth 3 for Finished HMK = 1, Early = 0:

```
{
  'Senior': 0.29650626049338447,
  'Likes Coffee': -0.014771863965748366,
  'Liked The Last Jedi': 0.29650626049338447
}
```

Information gain at depth 3 for Finished HMK = 0, Likes Coffee = 0:

```
{
  'Early': 0.43425063560155797,
  'Senior': 0.5852281360342516,
  'Liked The Last Jedi': 0.43425063560155797
}
```

The code for this is in the appendix at the end of the file.

Overall, I would choose the depth=2 tree. A depth of 1 likely does not capture enough of the complexity in the dataset, given the number of features, and a depth of 3 may cause overfitting.

d) A decision tree is realizable if there is some tree that perfectly classifies the given data set.
A tree of "no fixed depth" means that there is no set limit to the depth of the tree, so the tree can grow to arbitrary depth.
A dataset where two instances (table rows) have the same attribute (input) values but a different label (output) value will not be realizable, because no decision tree can be constructed that will label these two instances correctly. This can be proved by contradiction:
Assume two instances, x_1 and x_2, that have the same attributes, but x_1 has label y, and x_2 has label z, where $y \neq z$.
Because x_1 and x_2 have the same attributes, their feature vectors are identical. A deterministic decision tree must assign the same label to identical feature vectors. Since we know that $y \neq z$, we know that the tree does not classify one of x_1 or x_2 correctly. And since a realizable decision tree must classify all instances correctly, no such tree can exist.

2) Application of Decision Tree on Real-World Data Set

a) Here is the training accuracy for cut-off depths between 2 and 10:

```
'Depth': 2, 'Training Accuracy': 0.9444575312119404
'Depth': 3, 'Training Accuracy': 0.9448083679575788}
'Depth': 4, 'Training Accuracy': 0.945615292472547
'Depth': 5, 'Training Accuracy': 0.9492238989991129
'Depth': 6, 'Training Accuracy': 0.9497551660710796
'Depth': 7, 'Training Accuracy': 0.9520305929642197
'Depth': 8, 'Training Accuracy': 0.9526671110598778}
'Depth': 9, 'Training Accuracy': 0.9536695017617016
'Depth': 10, 'Training Accuracy': 0.9550177172556548}]
```

Based on these results, a depth of k = 10 gives the best training accuracy.

b) Here is the test accuracy for a depth of 10:

```
Test accuracy for depth=10: 0.9506525530763217
```

This equates top 95.07% accuracy in classifying the test data.

c) Based on the results from the test data, I do not see any overfitting issues. The accuracy of 95.07% with the test data was very close to the 95.50% accuracy with the training data.

The code for this is in the appendix at the end of this file.

3) Independent Events and Bayes Theorem

a) Bayes theorem states:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Event *B* can occur in two ways: with event *A* occurring, and without event *A* occurring. This means that *P(B)* is equivalent to the following:

$$P(B) = P(B|A)P(A) + P(B|\neg A)P(\neg A)$$

This gives the expression shown in the denominator of the question. So substituting this expanded expression for *P(B)* in the definition of Bayes theorem gives the proof.

b)

i) Is *X* independent of *Y*? Why or why not?

X and *Y* are independent if *P(X, Y) = P(X)P(Y)*

First, compute *P(X)*:
P(X = 0) = P(0,0,0) + P(0,1,0) + P(0,0,1) + P(0,1,1)
P(X = 0) = 0.1 + 0.2 + 0.1 + 0.175 = 0.575

P(X = 1) = P(1,0,0) + P(1,1,0) + P(1,0,1) + P(1,1,1)
P(X = 1) = 0.05 + 0.1 + 0.1 + 0.175 = 0.425

Next, compute *P(Y)*:
P(Y = 0) = P(0,0,0) + P(1,0,0) + P(0,0,1) + P(1,0,1)
P(Y=0) = 0.1 + 0.05 + 0.1 + 0.1 = 0.35

P(Y = 1) = P(0,1,0) + P(1,1,0) + P(0,1,1) + P(1,1,1)
P(Y = 1) = 0.2 + 0.1 + 0.175 + 0.175 = 0.65

Next, compute *P(X,Y)*:
P(X = 0, Y = 0) = P(0,0,0) + P(0,0,1) = 0.1 + 0.1 = 0.2
P(X = 0, Y = 1) = P(0,1,0) + P(0,1,1) = 0.2 + 0.175 = 0.375
P(X = 1, Y = 0) = P(1,0,0) + P(1,0,1) = 0.05 + 0.1 = 0.15
P(X = 1, Y = 1) = P(1,1,0) + P(1,1,1) = 0.1 + 0.175 = 0.275

(continued on next page)

Next, compare P(X)P(Y) with P(X,Y):

X	***Y***	***P(X, Y)***	***P(X)P(Y)***
0	0	0.2	0.20125
0	1	0.375	0.37375
1	0	0.15	0.14875
1	1	0.275	0.27625

Since *P(X,Y) ≠ P(X)P(Y)*, *X* and *Y* are not independent.

ii) Is *X* conditionally independent of *Y*, given Z? Why or why not?
X and *Y* are conditionally independent, given Z, if:

$$P(X, Y|Z) = P(X|Z)P(Y|Z)$$

This must be true for all values of X, Y, and Z.

Step 1: Calculate *P(Z)*:

P(Z = 0) = P(0,0,0) + P(0,1,0) + P(1,0,0) + P(1,1,0)
= 0.1 + 0.2 + 0.05 + 0.1 = 0.45
P(Z=1) = P(0,0,1) + P(0,1,1) + P(1,0,1) + P(1,1,1)
= 0.1 + 0.175 + 0.1 + 0.175 = 0.55

Step 2: Calculate *P(X|Z)*:

For X = 0:
P(X=0 | Z=0) = (P(0,0,0) + P(0,1,0)) / P(Z=0) = (0.1 + 0.2) / 0.45 = 0.6667
P(X=0 | Z=1) = (P(0,0,1) + P(0,1,1)) / P(Z=1) = (0.1 + 0.175) / 0.55 = 0.5000
For X = 1:
P(X=1 | Z=0) = (P(1,0,0) + P(1,1,0)) / P(Z=0) = (0.05 + 0.1) / 0.45 = 0.3333
P(X=1 | Z=1) = (P(1,0,1) + P(1,1,1)) / P(Z=1) = (0.1 + 0.175) / 0.55 = 0.5000

Step 3: Calculate *P(Y|Z)*

For Y = 0:
P(Y=0 | Z=0) = (P(0,0,0) + P(1,0,0)) / P(Z=0) = (0.1 + 0.05) / 0.45 = 0.3333
P(Y=0 | Z=1) = (P(0,0,1) + P(1,0,1)) / P(Z=1) = (0.1 + 0.1) / 0.55 = 0.3636
For Y = 1:
P(Y=1 | Z=0) = (P(0,1,0) + P(1,1,0)) / P(Z=0) = (0.2 + 0.1) / 0.45 = 0.6667
P(Y=1 | Z=1) = (P(0,1,1) + P(1,1,1)) / P(Z=1) = (0.175 + 0.175) / 0.55 = 0.6364

Step 4: Calculate *P(X,Y | Z)*:

Using the formula:
P(X, Y | Z) = P(X, Y, Z) / P(Z)

For Z = 0:
P(0,0 | 0) = P(0,0,0) / P(Z=0) = 0.1 / 0.45 = 0.2222
P(0,1 | 0) = P(0,1,0) / P(Z=0) = 0.2 / 0.45 = 0.4444
P(1,0 | 0) = P(1,0,0) / P(Z=0) = 0.05 / 0.45 = 0.1111
P(1,1 | 0) = P(1,1,0) / P(Z=0) = 0.1 / 0.45 = 0.2222
For Z = 1:
P(0,0 | 1) = P(0,0,1) / P(Z=1) = 0.1 / 0.55 = 0.1818
P(0,1 | 1) = P(0,1,1) / P(Z=1) = 0.175 / 0.55 = 0.3182
P(1,0 | 1) = P(1,0,1) / P(Z=1) = 0.1 / 0.55 = 0.1818
P(1,1 | 1) = P(1,1,1) / P(Z=1) = 0.175 / 0.55 = 0.3182

Step 5: Compute P(X | Z)P(Y | Z):
P(0 | 0) P(0 | 0) = (0.6667)(0.3333) = 0.2222
P(0 | 0) P(1 | 0) = (0.6667)(0.6667) = 0.4444
P(1 | 0) P(0 | 0) = (0.3333)(0.3333) = 0.1111
P(1 | 0) P(1 | 0) = (0.3333)(0.6667) = 0.2222
P(0 | 1) P(0 | 1) = (0.5000)(0.3636) = 0.1818
P(0 | 1) P(1 | 1) = (0.5000)(0.6364) = 0.3182
P(1 | 1) P(0 | 1) = (0.5000)(0.3636) = 0.1818
P(1 | 1) P(1 | 1) = (0.5000)(0.6364) = 0.3182

Given that the values obtained in Step 4 and Step 5 all match, we have proven that

$$P(X,Y|Z) = P(X|Z)P(Y|Z)$$

Therefore, *X* and *Y* are conditionally independent, given *Z*.

iii) Calculate *P(X ≠ Y | Z = 0)*

$$P(X \neq Y|Z=0) = \frac{P(X \neq Y, Z=0)}{P(Z=0)}$$

$$P(X \neq Y, Z=0) = P(0,1,0) + P(1,0,0) = 0.2 + 0.05 = 0.25$$

$$P(Z=0) = P(0,0,0) + P(1,0,0) + P(0,1,0) + P(1,1,0)$$
$$P(Z=0) = 0.1 + 0.05 + 0.2 + 0.1 = 0.45$$

$$P(X \neq Y|Z) = \frac{0.25}{0.45}$$

$$P(X \neq Y|Z) = 0.556$$

David Rosenfeld
UCF ID: 1550340

CAP 5610 – Machine Learning
Spring 2025

Implementing Naïve Bayes

The following results were obtained from the Naïve Bayes classifier:

```
Training Accuracy: 0.9693
Testing Accuracy: 0.9823
Training Time: 0.1093 seconds
```

The code for this implementation is in the appendix.

Appendices

The following pages contain the code for sections 1, 2, and 4.

question1.py

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

# Define a Pandas dataframe with the contents of the table from the problem statement.
data = pd.DataFrame([
    [1, 1, 0, 0, 1, 1],
    [1, 1, 1, 0, 1, 1],
    [0, 0, 1, 0, 0, 0],
    [0, 1, 1, 0, 1, 0],
    [0, 1, 1, 0, 0, 1],
    [0, 0, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 0],
    [0, 1, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1],
    [1, 0, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 1],
    [0, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 1, 0],
    [1, 0, 0, 1, 0, 1],
], columns=['Early', 'Finished HMK', 'Senior', 'Likes Coffee', 'Liked The Last Jedi', 'A'])

# Define an entropy function.  The input is an attribute (one column from the table).
def entropy(attr):
    total = len(attr)
    # If there are no values, the entropy is 0
    if total == 0:
        return 0
    # Attributes are binary (0 and 1), so proportions are calculated by summing and dividing.
    # Proportion of attr = 1
    p1 = sum(attr) / total
    # Proportion of attr = 0
    p0 = 1 - p1
    # Calcuate entropy
    if p1 == 0 or p0 == 0:
        # Entropy is 0 if either proportion is 0.
         e = 0
    else:
        # Otherwise, use the formula.
        e = - (p1 * math.log2(p1) + p0 * math.log2(p0))
    return e

# Compute the entropy of the dataset.
# This value is global so it can be reference in the information_gain
# function defined below.
H_S = entropy(data['A'])

```

```python
# Define a function to calculate information gain for an attribute.
def information_gain(data, attribute, target):
    # Get the unique values for the attribute (in this case, 0 and 1)
    values = data[attribute].unique()
    # Initialize the weighted entropy to zero.
    weighted_entropy = 0
    # Loop through each unique value.
    for v in values:
        # Extract the subset that matches this value.
        subset = data[data[attribute] == v][target]
        # Calculate the weighted entropy of the subset, and add it to
        # the overall weighted entropy.
        weighted_entropy += (len(subset) / len(data)) * entropy(subset)
    return H_S - weighted_entropy

# Compute the information gain for each attribute.
attributes = ['Early', 'Finished HMK', 'Senior', 'Likes Coffee', 'Liked The Last Jedi']
i_g_values = {attr: information_gain(data, attr, 'A') for attr in attributes}

print(f"Information gain for all attributes at depth 1:\n{i_g_values}")
# Select the best attribute (max information gain) to split at depth 1.
best_attr = max(i_g_values, key=i_g_values.get)

# Split data based on that best attribute
# The splits are in a hash where the key is the 0/1 value of the split attribute,
# and the value is the subset of the original table for that attribute value.
depth_1_split = {v: data[data[best_attr] == v] for v in data[best_attr].unique()}
print("-----------------------------------------")
print(f"depth_1_split: {depth_1_split}")

# Select the best attributes to split at depth 2.
# These splits will also go in a hash as before.
depth_2_splits = {}
# Loop through the keys/values of the depth 1 split.
for v, subset in depth_1_split.items():
    # If there is only one target value in the split, then no further
    # splits are needed or possible.
    if len(subset['A'].unique()) == 1:
        depth_2_splits[v] = None
    else:
        # Generate a list of the remaining attributes.
        remaining_attrs = [attr for attr in attributes if attr != best_attr]
        # For each of the remaining attributes, calculate its information gain.
        # Store those in a hash where the key is the attribute.
        ig_sub = {attr: information_gain(subset, attr, 'A') for attr in remaining_attrs}
        print(f"Information gain at depth 2 for {best_attr} = {v}:\n{ig_sub}")
        # Pick the highest information gain.
        best_sub_attr = max(ig_sub, key=ig_sub.get)
        # Store that attribute.
```

```python
        depth_2_splits[v] = best_sub_attr

print("----------------------------------------")
print(f"depth_2_splits: {depth_2_splits}")
print("----------------------------------------")

# Construct decision trees for depth 1 and depth 2
decision_tree_depth_1 = {best_attr: {v: {'Entropy': entropy(subset['A']), 'Positives': sum(subset['A']), 'Negatives': len(subset) - sum(subset['A'])} for v, subset in depth_1_split.items()}}
decision_tree_depth_2 = {best_attr: {v: {'Entropy': entropy(subset['A']), 'Positives': sum(subset['A']), 'Negatives': len(subset) - sum(subset['A']), 'Next Split': depth_2_splits[v]} for v, subset in depth_1_split.items()}}

# Extend to depth 3 for part C of this section.

# The depth_2_split hash that I created above does not have the table values.  It only has the attributes.
# So to compute the depth 3 split, I have to start again with the depth 1 split and work down to it.
# This is inefficient and could be improved.

depth_3_splits = {}
for v, subset in depth_1_split.items():
    if len(subset['A'].unique()) == 1:
        depth_3_splits[v] = None
    else:
        remaining_attrs = [attr for attr in attributes if attr != best_attr]
        ig_sub = {attr: information_gain(subset, attr, 'A') for attr in remaining_attrs}
        best_sub_attr = max(ig_sub, key=ig_sub.get)

        split_2 = {vv: subset[subset[best_sub_attr] == vv] for vv in subset[best_sub_attr].unique()}

        depth_3_splits[v] = {best_sub_attr: {}}

        for vv, subset_2 in split_2.items():
            if len(subset_2['A'].unique()) == 1:
                depth_3_splits[v][best_sub_attr][vv] = None
            else:
                remaining_attrs_2 = [attr for attr in remaining_attrs if attr != best_sub_attr]
                ig_sub_2 = {attr: information_gain(subset_2, attr, 'A') for attr in remaining_attrs_2}
                print(f"Information gain at depth 3 for {best_attr} = {v}, {best_sub_attr} = {vv}:\n{ig_sub_2}")
                best_sub_attr_2 = max(ig_sub_2, key=ig_sub_2.get)
                depth_3_splits[v][best_sub_attr][vv] = best_sub_attr_2

# Construct decision tree for depth 3
```

```
decision_tree_depth_3 = {best_attr: {}}
for v, subset in depth_1_split.items():
    decision_tree_depth_3[best_attr][v] = {
        'Entropy': entropy(subset['A']),
        'Positives': sum(subset['A']),
        'Negatives': len(subset) - sum(subset['A']),
        'Next Split': depth_2_splits[v],
        'Depth 3 Splits': depth_3_splits[v] if depth_3_splits[v] is not None else "Leaf Node"
    }

print("----------------------------------------")
print(f"depth_3_splits: {depth_3_splits}")
print("----------------------------------------")

# Display the decision tree results.
print("Decision tree, depth = 1:")
print(decision_tree_depth_1)
print("Decision tree, depth = 2")
print(decision_tree_depth_2)
print("Decision tree, depth = 3:")
print(decision_tree_depth_3)

```

question2.py

```python
import numpy as np
import pandas as pd
import math
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score

# The CSV data has no headers.  These headers will be added after import.
column_headers = [
    "age",
    "class_of_worker",
    "detailed_industry_recode",
    "detailed_occupation_recode",
    "education",
    "wage_per_hour",
    "enroll_in_edu_inst_last_wk",
    "marital_stat",
    "major_industry_code",
    "major_occupation_code",
    "race",
    "hispanic_origin",
    "sex",
    "member_of_labor_union",
    "reason_for_unemployment",
    "full_or_part_time_employment_stat",
    "capital_gains",
    "capital_losses",
    "dividends_from_stocks",
    "tax_filer_stat",
    "region_of_previous_residence",
    "state_of_previous_residence",
    "detailed_household_and_family_stat",
    "detailed_household_summary_in_household",
    "unknown_value",
    "migration_code_change_in_msa",
    "migration_code_change_in_reg",
    "migration_code_move_within_reg",
    "live_in_this_house_1_year_ago",
    "migration_prev_res_in_sunbelt",
    "num_persons_worked_for_employer",
    "family_members_under_18",
    "country_of_birth_father",
    "country_of_birth_mother",
    "country_of_birth_self",
    "citizenship",
    "own_business_or_self_employed",
    "fill_inc_questionnaire_for_veterans_admin",
```

```python
    "veterans_benefits",
    "weeks_worked",
    "year",
    "income_50k_plus"
]

# The names of the files to import.
datafiles = ["data/census-income.data.csv", "data/census-income.test.csv"]

# Initialize an array to hold dataframes for the data and test sets.
dataframes = []

# Load the CSV files.
for f in datafiles:
    # Read the file into a dataframe.  As noted, there are no headers.
    df = pd.read_csv(f, header=None)
    # Add the column headers.
    df.columns = column_headers

    # Data cleaning operations:

    # Handle missing values using forward fill and backward fill.
    df = df.ffill().bfill()

    # Upon analyzying the data, the two values in the "income_50k_plus" column, after
    # import, look like this: [' - 50000.', ' 50000+.'].  I am replacing these values using
    # one-hot encoding so that less than 50k income is a 0, and 50k+ is a 1.

    # Strip whitespace
    df["income_50k_plus"] = df["income_50k_plus"].str.strip()
    # Map the existing values to 0 or 1.
    df["income_50k_plus"] = df["income_50k_plus"].map({"- 50000.": 0, "50000+.": 1})

    # Upon analyzing the data, the the values in the "race" column contain leading
    # and/or trailing spaces.  These will be removed.

    # Strip whitespace
    df["race"] = df["race"].str.strip()

    # Add this dataframe to the array
    dataframes.append(df)

# Assign the data frames as training or test data.
training_data = dataframes[0]
test_data = dataframes[1]

# Define features and target variable
target_col = "income_50k_plus"
feature_cols = [col for col in training_data.columns if col != target_col]
```

```python

# Identify categorical and numerical features
categorical_cols = training_data.select_dtypes(include=["object"]).columns.tolist()
numerical_cols = training_data.select_dtypes(include=["int64", "float64"]).columns.tolist()

# Create a one-hot encoder for categorical features.
# Note: The income_50k_plus column was encoded earlier, becaue that code was carried
# over from Assignment 0.
encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)

# Encode the categorical columns from the test data.
encoded_train = encoder.fit_transform(training_data[categorical_cols])

# Encode the categorical columns from the test data.
encoded_test = encoder.transform(test_data[categorical_cols])

# Convert the encoded columns to dataframes.
encoded_train_df = pd.DataFrame(encoded_train, columns=encoder.get_feature_names_out(categorical_cols))
encoded_test_df = pd.DataFrame(encoded_test, columns=encoder.get_feature_names_out(categorical_cols))

# Assemble the prepared training and test data from the numerical features and the one-hot encoded
# categorical features.  Remove the target column from numerical features before during this process.
X_train = pd.concat([training_data[numerical_cols].drop(columns=[target_col]).reset_index(drop=True), encoded_train_df], axis=1)
X_test = pd.concat([test_data[numerical_cols].drop(columns=[target_col]).reset_index(drop=True), encoded_test_df], axis=1)

# Error checking: The code will exit if the target column is in the training or test data.
assert target_col not in X_train.columns, "Target column should not be in X_train!"
assert target_col not in X_test.columns, "Target column should not be in X_test!"

# Get the target column from the original training and test datasets.
y_train = training_data[target_col]
y_test = test_data[target_col]

# Part A: Train decision trees for depths of 2 - 10 and store accuracies.
depth_accuracies = []
for depth in range(2, 11):
    # The random_state value is arbitrary, as long as the same value is used
    # each time to allow reproducibility.
    model = DecisionTreeClassifier(max_depth=depth, random_state=42)
    model.fit(X_train, y_train)
    train_acc = accuracy_score(y_train, model.predict(X_train))
    depth_accuracies.append({"Depth": depth, "Training Accuracy": train_acc})
```

```python

print(depth_accuracies)
# Find the maximum accuracy and its associated depth.
max_accuracy = max([entry["Training Accuracy"] for entry in depth_accuracies])
# The optimal depth is the depth associated with max_accuracy.
optimal_depth = next(item["Depth"] for item in depth_accuracies if item["Training Accuracy"]
== max_accuracy)
print(f"max_accuracy = {max_accuracy}")
print(f"optimal_depth = {optimal_depth}")

# Part B: use the optimal depth found above to classify the test data.
# Re-train the model using the optimal depth found above.
model = DecisionTreeClassifier(max_depth=optimal_depth, random_state=42)
model.fit(X_train, y_train)
# Get the accuracy score for the training data (should be the same as before).
train_acc = accuracy_score(y_train, model.predict(X_train))
# Now get the accuracy score for the test data.
test_acc = accuracy_score(y_test, model.predict(X_test))
print(f"Training accuracy for depth=10: {train_acc}")
print(f"Test accuracy for depth=10: {test_acc}")


```

question4.py

```python
import numpy as np
import pandas as pd
import time
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# Load training data.  The shape is (4527, 5180).
X_train = pd.read_csv("nbdata/train.csv", header=None)
# Load training target data.  The shape is (4527).
y_train = pd.read_csv("nbdata/train_labels.txt", header=None).values.ravel()

# Load test data.  The shape is (1806, 5180).
X_test = pd.read_csv("nbdata/test.csv", header=None)  # Shape: (1806, 5180)
# Load test target data.  The shape is (1806).
y_test = pd.read_csv("nbdata/test_labels.txt", header=None).values.ravel()  # Shape: (1806,)

# Instantiate a Multinomial Naive Bayes Classifier
nbc = MultinomialNB()

# Record the starting time.
start_time = time.time()
# Train the classifer.
nbc.fit(X_train, y_train)
# When complete, calculate the training time.
training_time = time.time() - start_time

# Get the predictions for the training and test data.
y_train_pred = nbc.predict(X_train)
y_test_pred = nbc.predict(X_test)

# Calculate accuracies.
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print results
print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Testing Accuracy: {test_accuracy:.4f}")
print(f"Training Time: {training_time:.4f} seconds")


```