Untitled

September 29, 2023

1 CS 57300: Assignment 2

Name: Ishaan Roychowdhury
Date: September 21, 2023

1.1 1) Preprocessing (4 pts)

[1]:
```
import pandas as pd
part1_output_statements = []
df = pd.read_csv('dating-full.csv')
```

1) The format of values in some columns of the dataset is not unified. Strip the surrounding quotes in the values for columns race, race o and field (e.g., 'Asian/Pacific Islander/Asian-American' → Asian/Pacific Islander/Asian-American), count how many cells are changed after this pre-processing step, and output this number.

[2]:
```
number_of_changed_cells = 0
columns_to_modify = ['race', 'race_o', 'field']
for column_name in columns_to_modify:
    original_values = df[column_name].tolist()
    df[column_name] = df[column_name].str.strip('\'"')
    for original_value, modified_value in zip(original_values, df[column_name]):
        if original_value != modified_value:
            number_of_changed_cells += 1
print(f"Quotes removed from {number_of_changed_cells} cells.")
part1_output_statements.append(f"Quotes removed from {number_of_changed_cells}␣
↪cells.")
```

```
Quotes removed from 8316 cells.
```

---

2) Convert all the values in the column field to lowercase if they are not already in lowercases (e.g., Law → law). Count the number of cells that are changed after this pre-processing step, and output this number.

[3]:
```
changed_cells_case = 0
for col in ['field']:
    original_values = df[col].tolist()
```

```
    df[col] = df[col].str.lower()
    for original, modified in zip(original_values, df[col]):
        if original != modified:
            changed_cells_case += 1

print(f"Standardized {changed_cells_case} cells to lower case.")
part1_output_statements.append(f"Standardized {changed_cells_case} cells to␣
 ↪lower case.")
```

```
Standardized 5707 cells to lower case.
```

---

3) Use label encoding to convert the categorical values in columns gender, race, race o and field to numeric values start from 0. The process of label encoding works by mapping each categorical value of an attribute to an integer number between 0 and nvalues − 1 where nvalues is the number of distinct values for that attribute. Sort the values of each categorical attribute lexicographically/alphabetically before you start the encoding process. You are then asked to output the mapped numeric values for 'male' in the gender column, for 'European/Caucasian-American' in the race column, for 'Latino/Hispanic American' in the race o column, and for 'law' in the field column.

[4]:
```
def make_map(column_data):
    unique_values_list = sorted(column_data.unique())
    value_to_integer_map = {value: index for index, value in␣
 ↪enumerate(unique_values_list)}
    return value_to_integer_map

def encodeForQuestion3(column_data, value_to_integer_map):
    return column_data.map(value_to_integer_map)

gender_map = make_map(df['gender'])
df['gender'] = encodeForQuestion3(df['gender'], gender_map)

race_map = make_map(df['race'])
df['race'] = encodeForQuestion3(df['race'], race_map)

race_o_map = make_map(df['race_o'])
df['race_o'] = encodeForQuestion3(df['race_o'], race_o_map)

field_map = make_map(df['field'])
df['field'] = encodeForQuestion3(df['field'], field_map)

print(f"Value assigned for male in column gender: {gender_map['male']}.")
print(f"Value assigned for European/Caucasian-American in column race:␣
 ↪{race_map['European/Caucasian-American']}.")
print(f"Value assigned for Latino/Hispanic American in column race_o:␣
 ↪{race_o_map['Latino/Hispanic American']}.")
```

```
print(f"Value assigned for law in column field: {field_map['law']}.")
part1_output_statements.append(f"Value assigned for male in column gender:␣
 ↪{gender_map['male']}.")
part1_output_statements.append(f"Value assigned for European/Caucasian-American␣
 ↪in column race: {race_map['European/Caucasian-American']}.")
part1_output_statements.append(f"Value assigned for Latino/Hispanic American in␣
 ↪column race_o: {race_o_map['Latino/Hispanic American']}.")
part1_output_statements.append(f"Value assigned for law in column field:␣
 ↪{field_map['law']}.")
```

```
Value assigned for male in column gender: 1.
Value assigned for European/Caucasian-American in column race: 2.
Value assigned for Latino/Hispanic American in column race_o: 3.
Value assigned for law in column field: 121.
```

---

4) Normalization: As the speed dating experiments are conducted in several different batches, the instructions participants received across different batches vary slightly. For example, in some batches of experiments participants are asked to allocate a total of 100 points among the six attributes (i.e., attractiveness, sincerity, intelligence, fun, ambition, shared interests) to indicate how much they value each of these attributes in their romantic partner—that is, the values in preference scores of participant columns of a row should sum up to 100 (similarly, values in preference scores of partner columns of a row should also sum up to 100)—while in some other batches of experiments, participants are not explicitly instructed to do so

[5]:
```
def normalize_columns(df, columns):
    totals = df[columns].sum(axis=1)
    for col in columns:
        df[col] /= totals
    return df

participant_cols = [
    'attractive_important', 'sincere_important', 'intelligence_important',
    'funny_important', 'ambition_important', 'shared_interests_important'
]

partner_cols = [
    'pref_o_attractive', 'pref_o_sincere', 'pref_o_intelligence',
    'pref_o_funny', 'pref_o_ambitious', 'pref_o_shared_interests'
]

df = normalize_columns(df, participant_cols)
df = normalize_columns(df, partner_cols)

for col in participant_cols + partner_cols:
    print(f"Mean of {col}: {df[col].mean():.2f}.")
    part1_output_statements.append(f"Mean of {col}: {df[col].mean():.2f}.")
```

```
Mean of attractive_important: 0.22.
Mean of sincere_important: 0.17.
Mean of intelligence_important: 0.20.
Mean of funny_important: 0.17.
Mean of ambition_important: 0.11.
Mean of shared_interests_important: 0.12.
Mean of pref_o_attractive: 0.22.
Mean of pref_o_sincere: 0.17.
Mean of pref_o_intelligence: 0.20.
Mean of pref_o_funny: 0.17.
Mean of pref_o_ambitious: 0.11.
Mean of pref_o_shared_interests: 0.12.
```

[6]:
```
n = 0
for statement in part1_output_statements:
    print(statement)
    n += 1
```

```
Quotes removed from 8316 cells.
Standardized 5707 cells to lower case.
Value assigned for male in column gender: 1.
Value assigned for European/Caucasian-American in column race: 2.
Value assigned for Latino/Hispanic American in column race_o: 3.
Value assigned for law in column field: 121.
Mean of attractive_important: 0.22.
Mean of sincere_important: 0.17.
Mean of intelligence_important: 0.20.
Mean of funny_important: 0.17.
Mean of ambition_important: 0.11.
Mean of shared_interests_important: 0.12.
Mean of pref_o_attractive: 0.22.
Mean of pref_o_sincere: 0.17.
Mean of pref_o_intelligence: 0.20.
Mean of pref_o_funny: 0.17.
Mean of pref_o_ambitious: 0.11.
Mean of pref_o_shared_interests: 0.12.
```

[7]:
```
print("Checking Lines of Output: ", n)
```

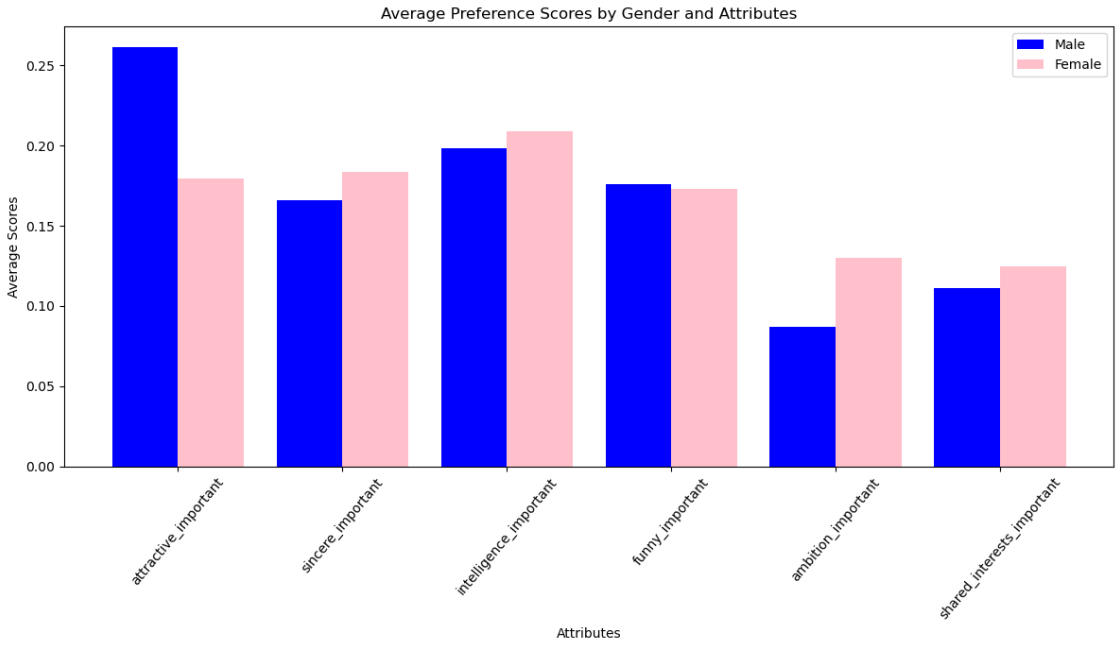
```
Checking Lines of Output:  18
```

1.2 2) Visualizing interesting trends in data (6 pts)

1) First, let's explore how males and females differ in terms of what are the attributes they value the most in their romantic partners. Please perform the following task on dating.csv, include your visualization code in this section and answer the following questions.

(a) Divide the dataset into two sub-datasets by the gender of participant
(b) Within each sub-dataset, compute the mean values for each column in the set preference scores of participant
(c) Use a single barplot to contrast how females and males value the six attributes in their romantic partners differently. Please use color of the bars to indicate gender. What do you observe from this visualization? What characteristics do males favor in their romantic partners? How does this differ from what females prefer?

[8]:
```
import matplotlib.pyplot as plt
male_participants = df[df['gender'] == 1]
female_participants = df[df['gender'] == 0]
male_preference_means = male_participants[participant_cols].mean()
female_preference_means = female_participants[participant_cols].mean()
attribute_labels = participant_cols
num_attributes = len(attribute_labels)
bar_width = 0.40
fig, ax = plt.subplots(figsize=(12,7))
male_bars = ax.bar(
    [i - bar_width/2 for i in range(num_attributes)],
    male_preference_means,
    bar_width,
    label='Male',
    color='blue'
)
female_bars = ax.bar(
    [i + bar_width/2 for i in range(num_attributes)],
    female_preference_means,
    bar_width,
    label='Female',
    color='pink'
)
ax.set_xlabel('Attributes')
ax.set_ylabel('Average Scores')
ax.set_title('Average Preference Scores by Gender and Attributes')
ax.set_xticks(range(num_attributes))
ax.set_xticklabels(attribute_labels, rotation=50)
ax.legend()
plt.tight_layout()
plt.show()
```



Q) What do you observe from this visualization? What characteristics do males favor in their romantic partners? How does this differ from what females prefer?

In this we can see that both males(blue) and females(pink) have comparable value for sincerity, intelligence, funny and shared interests. We can see that the biggest difference is in "Attrativeness" for males. Males prefer their romantic partners to be favored in "attractiveness".

This differs from females where the highest value is given to intelligence as a value for females in their romantic partners. The biggest difference between females and males is for "ambition". This means that ambition is something that is the biggest difference point which females prioritize more.

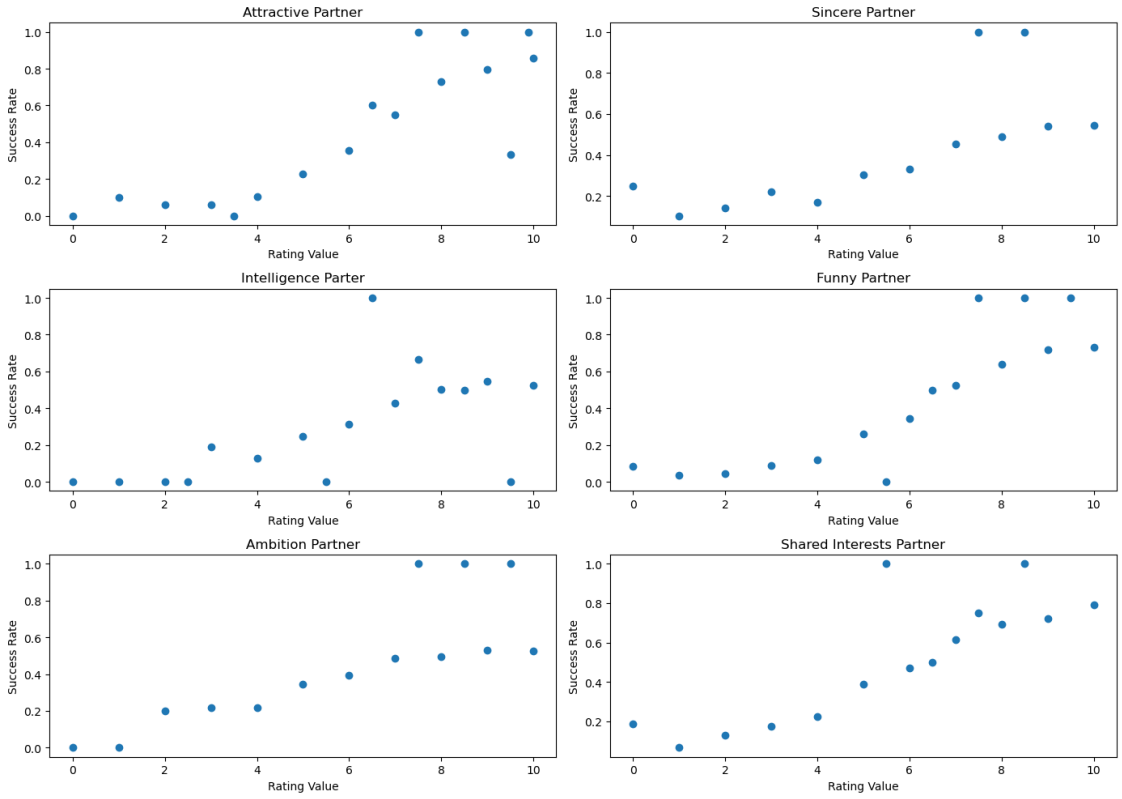
---

2) Next, let's explore how a participant's rating to their partner on each of the six attributes relate to how likely he/she will decide to give the partner a second date. Please perform the following task on dating.csv, include your visualization code in this section and answer the following question.

(a) Given an attribute in the set rating of partner from participant (e.g., attractive partner), determine the number of distinct values for this attribute.
(b) Given a particular value for the chosen attribute (e.g., a value of 10 for attribute 'attractive partner'), compute the fraction of participants who decide to give the partner a second date among all participants whose rating of the partner on the chosen attribute (e.g., attractive partner) is the given value (e.g., 10). We refer to this probability as the success rate for the group of partners whose rating on the chosen attribute is the specified value.
(c) Repeat the above process for all distinct values on each of the six attributes in the set rating of partner from participant.

(d) For each of the six attributes in the set rating of partner from participant, draw a scatter plot using the information computed above. Specifically, for the scatter plot of a particular attribute (e.g., attractive partner), use x-axis to represent different values on that attribute and y-axis to represent the success rate. We expect 6 scatter plots in total. What do you observe from these scatter plots?

[9]:
```
partner_ratings = [
    'attractive_partner', 'sincere_partner', 'intelligence_parter',
    'funny_partner', 'ambition_partner', 'shared_interests_partner'
]

def unique_val(column):
    return df[column].unique()

def success_rate_fetch(column, distinct_values):
    success_rates = []
    for value in distinct_values:
        subset = df[df[column] == value]
        rate = subset['decision'].sum() / len(subset)
        success_rates.append(rate)
    return success_rates

def plot_attribute_scatter(ax, column):
    distinct_values = unique_val(column)
    success_rates = success_rate_fetch(column, distinct_values)
    ax.scatter(distinct_values, success_rates)
    ax.set_title(column.replace('_', ' ').title())
    ax.set_xlabel('Rating Value')
    ax.set_ylabel('Success Rate')

fig, axs = plt.subplots(3, 2, figsize=(14, 10))
for idx, column in enumerate(partner_ratings):
    row_idx = idx // 2
    col_idx = idx % 2
    current_ax = axs[row_idx, col_idx]
    plot_attribute_scatter(current_ax, column)

plt.tight_layout()
plt.show()
```



We observe that attractive has got a high success rate if the rating is high. We can back this up since males ranked attractive as their top parameter. We also see that sincerity has lower success rates compared to others.

1.3 3) Convert continuous attributes to categorical attributes (3 pts)

Write a Python script to discretize all columns in continuous valued columns by splitting them into 5 bins of equal-width in the range of values for that column (check field-meaning.pdf for the range of each column; for those columns that you've finished pre-processing in Question 1(iv), the range should be considered as [0, 1]). If you encounter any values that lie outside the specified range of a certain column, please treat that value as the max value specified for that column. The script reads dating.csv as input and produces dating-binned.csv as output. As an output of your scripts, please print the number of items in each of the 5 bins. Bins should be sorted from small value ranges to large value ranges for each column in continuous valued columns. The sample outputs are as follows. We expect 47 lines of output, and the order of the attributes in the output should be the same as the order they occur in the dataset:

[10]:
```
import pandas as pd

columns_ranges = {
    'age': [18, 58],
```

```
    'age_o': [18, 58],
    'importance_same_race': [0, 10],
    'importance_same_religion': [0, 10],
    'pref_o_attractive': [0, 1],
    'pref_o_sincere': [0, 1],
    'pref_o_intelligence': [0, 1],
    'pref_o_funny': [0, 1],
    'pref_o_ambitious': [0, 1],
    'pref_o_shared_interests': [0, 1],
    'attractive_important': [0, 1],
    'sincere_important': [0, 1],
    'intelligence_important': [0, 1],
    'funny_important': [0, 1],
    'ambition_important': [0, 1],
    'shared_interests_important': [0, 1],
    'attractive': [0, 10],
    'sincere': [0, 10],
    'intelligence': [0, 10],
    'funny': [0, 10],
    'ambition': [0, 10],
    'attractive_partner': [0, 10],
    'sincere_partner': [0, 10],
    'intelligence_parter': [0, 10],
    'funny_partner': [0, 10],
    'ambition_partner': [0, 10],
    'shared_interests_partner': [0, 10],
    'sports': [0, 10],
    'tvsports': [0, 10],
    'exercise': [0, 10],
    'dining': [0, 10],
    'museums': [0, 10],
    'art': [0, 10],
    'hiking': [0, 10],
    'gaming': [0, 10],
    'clubbing': [0, 10],
    'reading': [0, 10],
    'tv': [0, 10],
    'theater': [0, 10],
    'movies': [0, 10],
    'concerts': [0, 10],
    'music': [0, 10],
    'shopping': [0, 10],
    'yoga': [0, 10],
    'interests_correlate': [-1, 1],
    'expected_happy_with_sd_people': [0, 10],
    'like': [0, 10],
}
```

```
df_copy = df.copy()
count_lines = 0
for column_name, (minimum_value, maximum_value) in columns_ranges.items():

    clipped_series = df_copy[column_name].clip(lower=minimum_value,␣
 ↪upper=maximum_value)
    binned_series, bin_edges = pd.cut(clipped_series, bins=5, labels=False,␣
 ↪retbins=True, include_lowest=True)
    df_copy[column_name] = binned_series
    sorted_bin_counts = df_copy[column_name].value_counts().sort_index()
    print(f"{column_name}: {sorted_bin_counts.tolist()}")
    count_lines += 1

df_copy.to_csv('dating-binned.csv', index=False)
```

```
age: [3032, 3390, 297, 20, 5]
age_o: [2990, 3362, 371, 16, 5]
importance_same_race: [2980, 1213, 977, 1013, 561]
importance_same_religion: [3203, 1188, 1110, 742, 501]
pref_o_attractive: [4333, 1987, 344, 51, 29]
pref_o_sincere: [1416, 4378, 865, 79, 6]
pref_o_intelligence: [666, 3935, 1873, 189, 81]
pref_o_funny: [1255, 4361, 1048, 55, 25]
pref_o_ambitious: [1963, 2352, 2365, 42, 22]
pref_o_shared_interests: [1506, 2068, 1981, 1042, 147]
attractive_important: [4323, 2017, 328, 57, 19]
sincere_important: [546, 2954, 2782, 377, 85]
intelligence_important: [630, 3976, 1861, 210, 67]
funny_important: [1282, 4306, 1070, 58, 28]
ambition_important: [1913, 2373, 2388, 49, 21]
shared_interests_important: [1464, 2197, 1950, 1007, 126]
attractive: [131, 726, 899, 4122, 866]
sincere: [57, 228, 352, 2715, 3392]
intelligence: [127, 409, 732, 3190, 2286]
funny: [19, 74, 1000, 2338, 3313]
ambition: [225, 697, 559, 2876, 2387]
attractive_partner: [284, 948, 2418, 2390, 704]
sincere_partner: [94, 353, 1627, 3282, 1388]
intelligence_parter: [36, 193, 1509, 3509, 1497]
funny_partner: [279, 733, 2296, 2600, 836]
ambition_partner: [119, 473, 2258, 2804, 1090]
shared_interests_partner: [701, 1269, 2536, 1774, 464]
sports: [650, 961, 1369, 2077, 1687]
tvsports: [2151, 1292, 1233, 1383, 685]
exercise: [619, 952, 1775, 2115, 1283]
dining: [39, 172, 1118, 2797, 2618]
```

```
museums: [117, 732, 1417, 2737, 1741]
art: [224, 946, 1557, 2500, 1517]
hiking: [963, 1386, 1575, 1855, 965]
gaming: [2565, 1522, 1435, 979, 243]
clubbing: [912, 1068, 1668, 2193, 903]
reading: [131, 398, 1071, 2317, 2827]
tv: [1188, 1216, 1999, 1642, 699]
theater: [288, 811, 1585, 2300, 1760]
movies: [144, 462, 530, 2783, 2825]
concerts: [222, 777, 1752, 2282, 1711]
music: [62, 196, 1106, 2583, 2797]
shopping: [1093, 1098, 1709, 1643, 1201]
yoga: [2285, 1392, 1369, 1056, 642]
interests_correlate: [75, 985, 2312, 2597, 775]
expected_happy_with_sd_people: [321, 1262, 3292, 1596, 273]
like: [273, 865, 2539, 2560, 507]
```

[11]:
```
print("Checking lines of output:" , count_lines)
```

```
Checking lines of output: 47
```

1.4 4) Training-Test Split (2 pts)

Use the sample function from pandas with the parameters initialized as random state = 47, frac = 0.2 to take a random 20% sample from the entire dataset. This sample will serve as your test dataset, and the rest will be your training dataset. (Note: The use of the random state will ensure all students have the same training and test datasets; incorrect or no initialization of this parameter will lead to non-reproducible results). Create a new script that takes dating-binned.csv as input and outputs trainingSet.csv and testSet.csv.

[12]:
```
file = 'dating-binned.csv'
df_binned = pd.read_csv(file)
test_df = df_binned.sample(frac=0.2, random_state=47)
train_df = df_binned.drop(test_df.index)
train_df.to_csv('trainingSet.csv', index=False)
test_df.to_csv('testSet.csv', index=False)
```

[13]:
```
training_df_loaded = pd.read_csv('trainingSet.csv')
test_df_loaded = pd.read_csv('testSet.csv')
print(f"Number of Columns in Training DataFrame: {len(training_df_loaded.
  ↪columns)}")
print(f"Number of Rows in Training DataFrame: {training_df_loaded.shape[0]}")
print("\n----------------------------------------\n")
print(f"Number of Columns in Test DataFrame: {len(test_df_loaded.columns)}")
print(f"Number of Rows in Test DataFrame: {test_df_loaded.shape[0]}")
```

```
Number of Columns in Training DataFrame: 53
Number of Rows in Training DataFrame: 5395

----------------------------------------

Number of Columns in Test DataFrame: 53
Number of Rows in Test DataFrame: 1349
```

1.5 5) Implement a Naive Bayes Classifier (15 pts)

Write a function named nbc(t frac) to train your NBC which takes a parameter t frac that represents the fraction of the training data to sample from the original training set. Use the sample function from pandas with the parameters initialized as random state = 47, frac = t frac to generate random samples of training data of different sizes. 1. Use all the attributes and all training examples in trainingSet.csv to train the NBC by calling your nbc(t frac) function with t frac = 1. After get the learned model, apply it on all examples in the training dataset (i.e., trainingSet.csv) and test dataset (i.e., testSet.csv) and compute the accuracy respectively. Please include both your code and your outputs.

[14]:
```
def nbc(train_df, test_df, t_frac):
    sampled_train_df = train_df.sample(frac=t_frac, random_state=47)
    class_probabilities = {}
    total_samples = len(sampled_train_df)
    for target_class in sampled_train_df['decision'].unique():
        class_df = sampled_train_df[sampled_train_df['decision'] ==␣
 ↪target_class]
        class_probabilities[target_class] = len(class_df) / total_samples
    attribute_probabilities = {}
    for target_class in class_probabilities.keys():
        attribute_probabilities[target_class] = {}
        class_df = sampled_train_df[sampled_train_df['decision'] ==␣
 ↪target_class]
        for attribute in class_df.columns:
            if attribute != 'decision':
                attribute_value_counts = class_df[attribute].value_counts()
                num_categories = len(train_df[attribute].unique())
                attribute_probabilities[target_class][attribute] =␣
 ↪(attribute_value_counts + 1) / (len(class_df) + num_categories)

    training_correct = 0
    testing_correct = 0
    training_total = len(sampled_train_df)
    testing_total = len(test_df)
    for _, row in sampled_train_df.iterrows():
        training_class_likelihoods = {}
        for target_class in class_probabilities.keys():
```

```
            likelihood = class_probabilities[target_class]
            for attribute in row.index:
                if attribute != 'decision':
                    value = row[attribute]
                    if value in␣
 ↪attribute_probabilities[target_class][attribute]:
                        likelihood *=␣
 ↪attribute_probabilities[target_class][attribute][value]
            training_class_likelihoods[target_class] = likelihood
        training_predicted_class = max(training_class_likelihoods,␣
 ↪key=training_class_likelihoods.get)
        if training_predicted_class == row['decision']:
            training_correct += 1
    for _, row in test_df.iterrows():
        testing_class_likelihoods = {}
        for target_class in class_probabilities.keys():
            likelihood = class_probabilities[target_class]
            for attribute in row.index:
                if attribute != 'decision':
                    value = row[attribute]
                    if value in␣
 ↪attribute_probabilities[target_class][attribute]:
                        likelihood *=␣
 ↪attribute_probabilities[target_class][attribute][value]
            testing_class_likelihoods[target_class] = likelihood
        testing_predicted_class = max(testing_class_likelihoods,␣
 ↪key=testing_class_likelihoods.get)
        if testing_predicted_class == row['decision']:
            testing_correct += 1
    training_accuracy = training_correct / training_total
    testing_accuracy = testing_correct / testing_total
    print(f"Training Accuracy: {training_accuracy:.2f}")
    print(f"Testing Accuracy: {testing_accuracy:.2f}")
    return training_accuracy, testing_accuracy

train_df = pd.read_csv('trainingSet.csv')
test_df = pd.read_csv('testSet.csv')
nbc(train_df, test_df, t_frac=1)
```

```
Training Accuracy: 0.76
Testing Accuracy: 0.73
```

[14]: `(0.7592215013901761, 0.7346182357301705)`

---

2. Examine the effects of varying the number of bins for continuous attributes during the discretization step. Please include both your code and your outputs.

(i) Given the number of bins b B = {2, 5, 10, 50, 100, 200}, perform discretization for all columns in set continuous valued columns by splitting the values in each column into b bins of equal width within its range. For this task, you can re-use your discretization code to perform the binning procedure, now taking the number of bins as a parameter and using dating.csv as input as earlier.)

(ii) Repeat the train-test split as described in Question 4 for the obtained dataset after discretizing each continuous attribute into b bins.

(iii) For each value of b, train the NBC on the corresponding new training dataset by calling your nbc(t frac) function with t frac = 1, and apply the learned model on the corresponding new test dataset.

(iv) Draw a plot to show how the value of b affects the learned NBC model's performance on the training dataset and the test dataset, with x-axis representing the value of b and y-axis representing the model accuracy. Comment on what you observe in the plot.

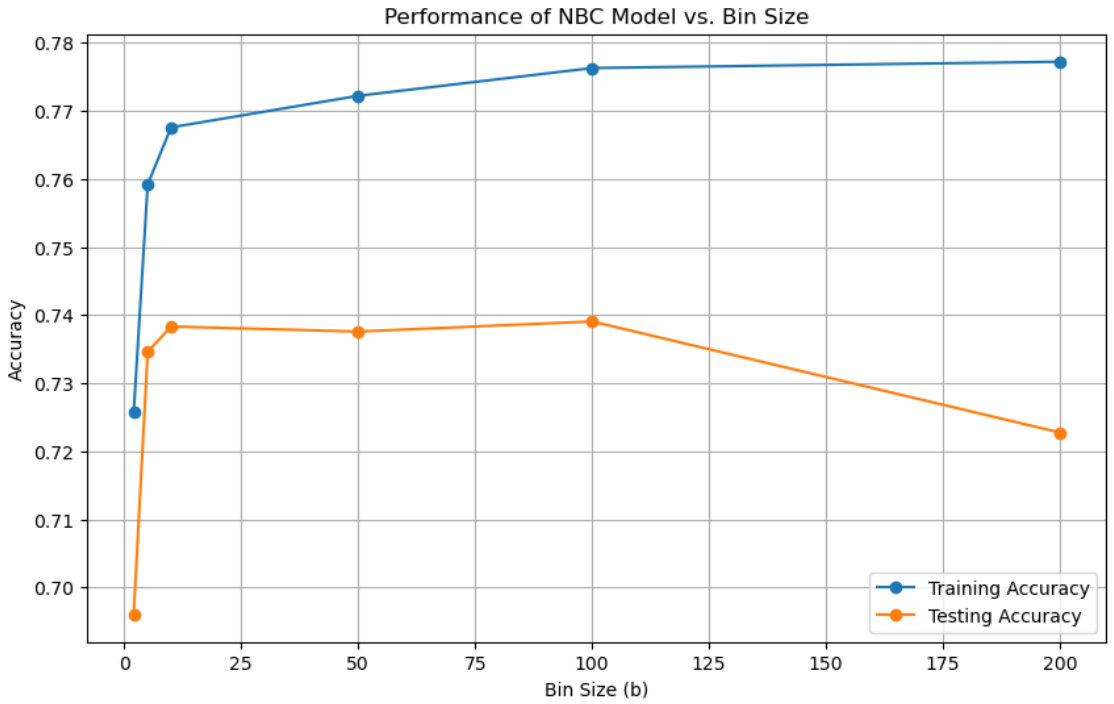
[15]:
```
def discretize_df(df, bins):
    df_copy = df.copy()
    for column_name, (minimum_value, maximum_value) in columns_ranges.items():
        clipped_series = df_copy[column_name].clip(lower=minimum_value,␣
 ↪upper=maximum_value)
        binned_series, bin_edges = pd.cut(clipped_series, bins=bins,␣
 ↪labels=False, retbins=True, include_lowest=True)
        df_copy[column_name] = binned_series
    return df_copy
```

[16]:
```
bin_sizes = [2, 5, 10, 50, 100, 200]
training_accuracies = []
testing_accuracies = []

for bin_size in bin_sizes:
    discretized_df = discretize_df(df, bin_size)
    test_df = discretized_df.sample(frac=0.2, random_state=47)
    train_df = discretized_df.drop(test_df.index)
    train_acc, test_acc = nbc(train_df, test_df, 1)
    print(f"Bin size: {bin_size}")
    print("-------")

    training_accuracies.append(train_acc)
    testing_accuracies.append(test_acc)
```

```
Training Accuracy: 0.73
Testing Accuracy: 0.70
Bin size: 2
-------
Training Accuracy: 0.76
Testing Accuracy: 0.73
Bin size: 5
-------
```

```
Training Accuracy: 0.77
Testing Accuracy: 0.74
Bin size: 10
-------
Training Accuracy: 0.77
Testing Accuracy: 0.74
Bin size: 50
-------
Training Accuracy: 0.78
Testing Accuracy: 0.74
Bin size: 100
-------
Training Accuracy: 0.78
Testing Accuracy: 0.72
Bin size: 200
-------
```

[17]:
```
plt.figure(figsize=(10, 6))
plt.plot(bin_sizes, training_accuracies, marker='o', label='Training Accuracy')
plt.plot(bin_sizes, testing_accuracies, marker='o', label='Testing Accuracy')
plt.xlabel('Bin Size (b)')
plt.ylabel('Accuracy')
plt.title('Performance of NBC Model vs. Bin Size')
plt.legend()
plt.grid(True)
```



In this we can see that as bin size increase in test data, our accuracy eventually overfits and starts to decrease. Our training data on the other hand reaches a very high number of accuracy as bins increase. When we use a larger number of bins, we are increasing the complexity of our model. Each bin represents a specific range of values for a feature. With more bins, the model can become more sensitive to variations in the data, including noise leading to overfitting

---

3. Plot the learning curve. Please include your code in this section.

(i) For each f in F = {0.01, 0.1, 0.2, 0.5, 0.6, 0.75, 0.9, 1}, randomly sample a fraction of the training data in trainingSet.csv with our fixed seed (i.e., random state=47).

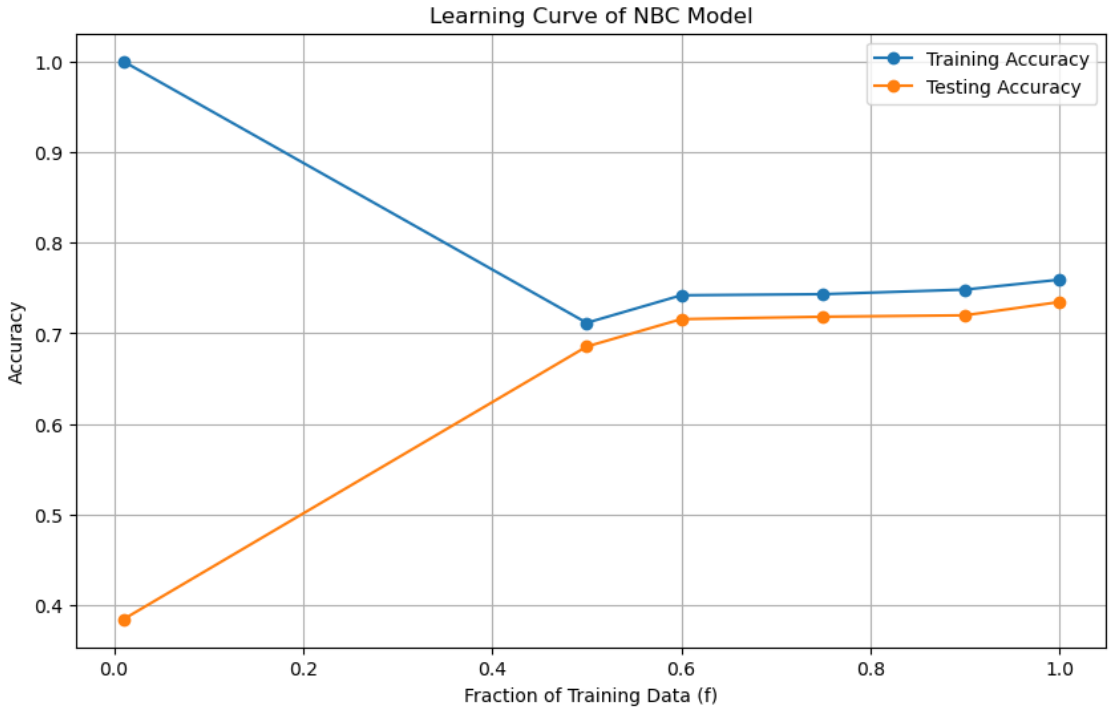
(ii) Train a NBC model on the selected f fraction of the training dataset (You can call your nbc(t frac) function with t frac = f). Evaluate the performance of the learned model on all examples in the selected samples of training data as well as all examples in the test dataset (i.e., testSet.csv), and compute the accuracy respectively. Do so for all f F.

(iii) Draw one plot of learning curves where the x-axis representing the values of f and the y-axis representing the corresponding model's accuracy on training/test dataset. Comment on what you observe in this plot.

[18]:
```
import pandas as pd
import matplotlib.pyplot as plt

F = [0.01, 0.1, 0.2, 0.5, 0.6, 0.75, 0.9, 1]
train_df = pd.read_csv('trainingSet.csv')
test_df = pd.read_csv('testSet.csv')

training_accuracies = []
testing_accuracies = []

for f in F:
    sampled_train_df = train_df.sample(frac=f, random_state=47)
    sampled_test_df = test_df.sample(frac=f, random_state=47)
    train_acc, test_acc = nbc(sampled_train_df, sampled_test_df, t_frac=f)
    training_accuracies.append(train_acc)
    testing_accuracies.append(test_acc)

plt.figure(figsize=(10, 6))
plt.plot(F, training_accuracies, marker='o', label='Training Accuracy')
plt.plot(F, testing_accuracies, marker='o', label='Testing Accuracy')
plt.xlabel('Fraction of Training Data (f)')
plt.ylabel('Accuracy')
plt.title('Learning Curve of NBC Model')
plt.legend()
plt.grid(True)
plt.show()
```

```
Training Accuracy: 1.00
```

```
Testing Accuracy: 0.38
Training Accuracy: 0.71
Testing Accuracy: 0.69
Training Accuracy: 0.74
Testing Accuracy: 0.72
Training Accuracy: 0.74
Testing Accuracy: 0.72
Training Accuracy: 0.75
Testing Accuracy: 0.72
Training Accuracy: 0.76
Testing Accuracy: 0.73
```



Initially, when the fraction f is very small, the model is likely overfitting to the training data, leading to high training accuracy but low testing accuracy. As the fraction f increases, meaning as the model is trained on more and more data, it starts to generalize better to unseen data, so the testing accuracy improves. Concurrently, as the model is exposed to more variability and noise in the training data, the training accuracy might decrease slightly but will be more representative of the model's true performance. Eventually, with enough data, the model will reach a point where it generalizes well to unseen data, and both training and testing accuracies will converge to a similar value, representing the model's true ability to generalize to unseen data.

[ ]:

[ ]: