Lab7

September 3, 2024

1 Q1

[1]:
```
def bayesTheorem(pA, pB, pBA):
    return (pB * pBA) / pA

# Problem a
P_H = 0.60  # Probability of being a hosteler
P_D = 0.40  # Probability of being a day scholar
P_A_given_H = 0.30  # Probability of scoring A grade given hosteler
P_A_given_D = 0.20  # Probability of scoring A grade given day scholar

# Calculate P(A)
P_A = (P_A_given_H * P_H) + (P_A_given_D * P_D)

# Calculate P(H|A)
P_H_given_A = bayesTheorem(P_A, P_H, P_A_given_H)
print(f"Probability that a student is a hosteler given they scored A grade:␣
 ↪{P_H_given_A:.4f}")

# Problem b
P_Disease = 0.01  # Prevalence of the disease
P_Test_Positive_given_Disease = 0.99  # Sensitivity
P_False_Positive = 0.02  # False positive rate
P_Test_Positive_given_NoDisease = P_False_Positive  # False positive rate
P_No_Disease = 1 - P_Disease  # Probability of not having the disease

# Calculate P(T)
P_Test_Positive = (P_Test_Positive_given_Disease * P_Disease) +␣
 ↪(P_Test_Positive_given_NoDisease * P_No_Disease)

# Calculate P(Disease|T)
P_Disease_given_Test_Positive = bayesTheorem(P_Test_Positive, P_Disease,␣
 ↪P_Test_Positive_given_Disease)
print(f"Probability of having the disease given a positive test result:␣
 ↪{P_Disease_given_Test_Positive:.4f}")
```

```
Probability that a student is a hosteler given they scored A grade: 0.6923
Probability of having the disease given a positive test result: 0.3333
```

2 Q2

[2]:
```
import pandas as pd
import numpy as np

class NaiveBayesClassifier:
    def __init__(self):
        self.prior_probs = {}
        self.likelihoods = {}
        self.classes = []
        self.features = []

    def fit(self, X, y):
        self.classes = np.unique(y)
        self.features = X.columns
        self.prior_probs = {cls: np.mean(y == cls) for cls in self.classes}
        self.likelihoods = {cls: {} for cls in self.classes}

        for cls in self.classes:
            cls_data = X[y == cls]
            for feature in self.features:
                feature_values = cls_data[feature].value_counts(normalize=True)
                self.likelihoods[cls][feature] = feature_values.to_dict()

    def predict(self, X):
        predictions = []
        for _, row in X.iterrows():
            class_probs = {}
            for cls in self.classes:
                prior = self.prior_probs[cls]
                likelihood = 1
                for feature in self.features:
                    feature_value = row[feature]
                    if feature in self.likelihoods[cls] and feature_value in␣
 ↪self.likelihoods[cls][feature]:
                        likelihood *= self.
 ↪likelihoods[cls][feature][feature_value]
                    else:
                        likelihood *= 1e-6  # Smoothing for unseen feature␣
 ↪values
                class_probs[cls] = prior * likelihood
            predictions.append(max(class_probs, key=class_probs.get))
        return predictions

# Load the dataset
df = pd.read_csv('buyers_data.csv')
```

```
# Preprocess data
df['buys_computer'] = df['buys_computer'].map({'yes': 'yes', 'no': 'no'})
X = df.drop('buys_computer', axis=1)
y = df['buys_computer']

# Encode categorical features
X_encoded = pd.get_dummies(X)

# Train the Naive Bayes classifier
nb_classifier = NaiveBayesClassifier()
nb_classifier.fit(X_encoded, y)

# Make predictions on the training data
predictions = nb_classifier.predict(X_encoded)
accuracy = np.mean(predictions == y)

print(f"Training Accuracy: {accuracy:.2f}")

# Example prediction for a new data point
new_data = pd.DataFrame({
    'age': ['<=30'],
    'income': ['high'],
    'student': ['no'],
    'credit_rating': ['fair']
})

new_data_encoded = pd.get_dummies(new_data)
new_data_encoded = new_data_encoded.reindex(columns=X_encoded.columns,␣
 ↪fill_value=0)
prediction = nb_classifier.predict(new_data_encoded)
print(f"Prediction for new data point: {prediction[0]}")
```

```
Training Accuracy: 0.86
Prediction for new data point: no
```

3 Q3

[3]:
```
import pandas as pd
from collections import Counter
import numpy as np
import re

class NaiveBayesTextClassifier:
    def __init__(self):
        self.class_probs = {}
        self.word_probs = {}
        self.vocab = set()
```

```
        self.classes = []

    def preprocess(self, text):
        # Convert to lowercase and remove non-alphanumeric characters
        text = text.lower()
        text = re.sub(r'[^a-z0-9\s]', '', text)
        return text

    def fit(self, X, y):
        # Get the classes and their prior probabilities
        self.classes = np.unique(y)
        class_counts = y.value_counts()
        total_count = len(y)
        self.class_probs = {cls: count / total_count for cls, count in␣
↪class_counts.items()}

        # Initialize word counts
        word_counts = {cls: Counter() for cls in self.classes}
        class_word_counts = {cls: 0 for cls in self.classes}

        # Count words in each class
        for text, cls in zip(X, y):
            words = self.preprocess(text).split()
            word_counts[cls].update(words)
            class_word_counts[cls] += len(words)
            self.vocab.update(words)

        # Calculate word probabilities with Laplace smoothing
        self.word_probs = {cls: {} for cls in self.classes}
        vocab_size = len(self.vocab)
        for cls in self.classes:
            total_words = class_word_counts[cls] + vocab_size
            for word in self.vocab:
                self.word_probs[cls][word] = (word_counts[cls][word] + 1) /␣
↪total_words

    def predict(self, X):
        predictions = []
        for text in X:
            words = self.preprocess(text).split()
            class_scores = {}
            for cls in self.classes:
                log_prob = np.log(self.class_probs[cls])
                for word in words:
                    if word in self.word_probs[cls]:
                        log_prob += np.log(self.word_probs[cls][word])
                    else:
```

```
                        # Use a small probability for unknown words
                        log_prob += np.log(1 / (class_word_counts[cls] +␣
↪len(self.vocab)))
                class_scores[cls] = log_prob
            predictions.append(max(class_scores, key=class_scores.get))
        return predictions

    def evaluate(self, X, y_true):
        predictions = self.predict(X)
        accuracy = np.mean(predictions == y_true)
        precision, recall = {}, {}

        for cls in self.classes:
            true_positive = np.sum((predictions == cls) & (y_true == cls))
            false_positive = np.sum((predictions == cls) & (y_true != cls))
            false_negative = np.sum((predictions != cls) & (y_true == cls))

            precision[cls] = true_positive / (true_positive + false_positive)␣
↪if (true_positive + false_positive) > 0 else 0
            recall[cls] = true_positive / (true_positive + false_negative) if␣
↪(true_positive + false_negative) > 0 else 0

        return accuracy, precision, recall

# Load the dataset
df = pd.read_csv('text_data.csv')

# Prepare data
X = df['Text']
y = df['Tag']

# Initialize and train the Naive Bayes classifier
nb_classifier = NaiveBayesTextClassifier()
nb_classifier.fit(X, y)

# Evaluate the model
accuracy, precision, recall = nb_classifier.evaluate(X, y)

print(f"Accuracy: {accuracy:.2f}")
print("Precision:")
for cls, prec in precision.items():
    print(f"  {cls}: {prec:.2f}")
print("Recall:")
for cls, rec in recall.items():
    print(f"  {cls}: {rec:.2f}")

# Predict the tag for a new sentence
```

```
new_sentence = ["A very close game"]
prediction = nb_classifier.predict(new_sentence)
print(f"Prediction for '{new_sentence[0]}': {prediction[0]}")
```

```
Accuracy: 1.00
Precision:
  Not sports: 0.00
  Sports: 0.00
Recall:
  Not sports: 0.00
  Sports: 0.00
Prediction for 'A very close game': Sports
```

[4]:
```
import pandas as pd
from collections import defaultdict
import math

# Sample dataset
data = {
    "Text": [
        "A great game",
        "The election was over",
        "Very clean match",
        "A clean but forgettable game",
        "It was a close election"
    ],
    "Tag": [
        "Sports",
        "Not sports",
        "Sports",
        "Sports",
        "Not sports"
    ]
}

# Convert to DataFrame
df = pd.DataFrame(data)

# Preprocess text
def preprocess_text(text):
    return text.lower().split()

df['Processed Text'] = df['Text'].apply(preprocess_text)

# Initialize frequency tables
word_counts = defaultdict(lambda: defaultdict(int))
class_counts = defaultdict(int)
```

```
total_docs = 0

# Populate frequency tables
for _, row in df.iterrows():
    text = row['Processed Text']
    tag = row['Tag']

    class_counts[tag] += 1
    total_docs += 1

    for word in text:
        word_counts[tag][word] += 1

# Calculate class probabilities
class_probs = {cls: count / total_docs for cls, count in class_counts.items()}

# Calculate word probabilities
def calculate_word_probs(word_counts, class_counts, total_docs):
    word_probs = defaultdict(lambda: defaultdict(float))
    vocab = set(word for word_counts_class in word_counts.values() for word in␣
 ↪word_counts_class)

    for cls, counts in word_counts.items():
        total_words_in_class = sum(counts.values())
        vocab_size = len(vocab)
        for word in vocab:
            word_probs[cls][word] = (counts.get(word, 0) + 1) /␣
 ↪(total_words_in_class + vocab_size)

    return word_probs

word_probs = calculate_word_probs(word_counts, class_counts, total_docs)

# Classify new text
def classify_text(text, class_probs, word_probs):
    words = preprocess_text(text)
    class_scores = {cls: math.log(prob) for cls, prob in class_probs.items()}

    for cls in class_probs:
        for word in words:
            if word in word_probs[cls]:
                class_scores[cls] += math.log(word_probs[cls][word])

    return max(class_scores, key=class_scores.get)

# Test the classifier with a new sentence
new_sentence = "A very close game"
```

```
tag = classify_text(new_sentence, class_probs, word_probs)
print(f'The sentence "{new_sentence}" is classified as: {tag}')
```

```
The sentence "A very close game" is classified as: Sports
```

[5]:
```
import pandas as pd
from collections import defaultdict

# Sample dataset
data = {
    "Text": [
        "A great game",
        "The election was over",
        "Very clean match",
        "A clean but forgettable game",
        "It was a close election"
    ],
    "Tag": [
        "Sports",
        "Not sports",
        "Sports",
        "Sports",
        "Not sports"
    ]
}

# Convert to DataFrame
df = pd.DataFrame(data)

# Preprocess text
def preprocess_text(text):
    return text.lower().split()

df['Processed Text'] = df['Text'].apply(preprocess_text)

# Initialize frequency tables
word_counts = defaultdict(lambda: defaultdict(int))
class_counts = defaultdict(int)

# Populate frequency tables
for _, row in df.iterrows():
    text = row['Processed Text']
    tag = row['Tag']

    class_counts[tag] += 1

    for word in text:
```

```
        word_counts[tag][word] += 1

# Convert frequency tables to DataFrames for display
word_counts_df = pd.DataFrame(word_counts).fillna(0).astype(int)
class_counts_df = pd.DataFrame(list(class_counts.items()), columns=['Class',␣
 ↪'Count'])

# Display the frequency occurrence tables
print("Word Occurrence Frequency Table by Class:")
print(word_counts_df)
print("\nClass Frequency Table:")
print(class_counts_df)
```

```
Word Occurrence Frequency Table by Class:
             Sports  Not sports
a                 2           1
great             1           0
game              2           0
very              1           0
clean             2           0
match             1           0
but               1           0
forgettable       1           0
the               0           1
election          0           2
was               0           2
over              0           1
it                0           1
close             0           1

Class Frequency Table:
        Class  Count
0      Sports      3
1  Not sports      2
```

[ ]: