Lab7

September 6, 2024

[3]:
```
# Problem (a): Probability of being a hosteler given an A grade

# Given probabilities
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 A grade given hosteler
P_A_given_D = 0.20  # Probability of A grade given day scholar

# Total probability of A grade
P_A = (P_A_given_H * P_H) + (P_A_given_D * P_D)

# Probability of being a hosteler given an A grade
P_H_given_A = (P_A_given_H * P_H) / P_A
print(f"Probability that a student with an A grade is a hosteler: {P_H_given_A:.
 ↪3f}")

# Problem (b): Probability of having the disease given a positive test result

# Given probabilities
P_D = 0.01  # Prevalence of the disease
P_D_not = 1 - P_D  # Probability of not having the disease
P_T_given_D = 0.99  # Sensitivity (True Positive Rate)
P_T_given_D_not = 0.02  # False Positive Rate

# Total probability of testing positive
P_T = (P_T_given_D * P_D) + (P_T_given_D_not * P_D_not)

# Probability of having the disease given a positive test result
P_D_given_T = (P_T_given_D * P_D) / P_T
print(f"Probability of having the disease given a positive test result:␣
 ↪{P_D_given_T:.3f}")
```

```
Probability that a student with an A grade is a hosteler: 0.692
Probability of having the disease given a positive test result: 0.333
```

[5]:
```
import pandas as pd
import numpy as np
```

```
# Load the dataset
data = {
    'age': ['<=30', '<=30', '31…40', '>40', '>40', '>40', '31…40', '<=30',␣
 ↪'<=30', '>40', '<=30', '31…40', '31…40', '>40'],
    'income': ['high', 'high', 'high', 'medium', 'low', 'low', 'low', 'medium',␣
 ↪'low', 'medium', 'medium', 'medium', 'high', 'medium'],
    'student': ['no', 'no', 'no', 'no', 'yes', 'yes', 'yes', 'no', 'yes',␣
 ↪'yes', 'yes', 'no', 'yes', 'no'],
    'credit_rating': ['fair', 'excellent', 'fair', 'fair', 'fair', 'excellent',␣
 ↪'excellent', 'fair', 'fair', 'fair', 'excellent', 'excellent', 'fair',␣
 ↪'excellent'],
    'buys_computer': ['no', 'no', 'yes', 'yes', 'yes', 'no', 'yes', 'no',␣
 ↪'yes', 'yes', 'yes', 'yes', 'yes', 'no']
}

df = pd.DataFrame(data)

# Convert categorical data into numerical data
def encode_data(df):
    return pd.get_dummies(df, drop_first=True)

df_encoded = encode_data(df)

# Naïve Bayes classifier
class NaiveBayesClassifier:
    def __init__(self):
        self.class_prob = {}
        self.feature_probs = {}

    def fit(self, X, y):
        # Compute prior probabilities
        classes = y.unique()
        self.class_prob = {c: np.mean(y == c) for c in classes}

        # Compute likelihoods
        for feature in X.columns:
            self.feature_probs[feature] = {}
            for c in classes:
                subset = X[y == c]
                feature_probs = subset[feature].value_counts(normalize=True).
 ↪to_dict()
                self.feature_probs[feature][c] = feature_probs

    def predict(self, X):
        predictions = []
        for _, row in X.iterrows():
```

```
            posteriors = {}
            for c in self.class_prob:
                prior = self.class_prob[c]
                likelihood = 1
                for feature in X.columns:
                    value = row[feature]
                    if value in self.feature_probs[feature][c]:
                        likelihood *= self.feature_probs[feature][c][value]
                    else:
                        likelihood *= 1e-6  # small value for unseen feature␣
 ↪values
                posteriors[c] = prior * likelihood
            # Choose the class with the highest posterior probability
            prediction = max(posteriors, key=posteriors.get)
            predictions.append(prediction)
        return np.array(predictions)

# Prepare the data
X = df_encoded.drop('buys_computer_yes', axis=1)
y = df_encoded['buys_computer_yes']

# Train the classifier
model = NaiveBayesClassifier()
model.fit(X, y)

# Predict on the training set
y_pred = model.predict(X)

# Evaluate the classifier
accuracy = np.mean(y_pred == y)
print(f"Training Accuracy: {accuracy:.2f}")

# Prepare sample prediction
# Ensure that sample columns match the training data columns
sample = pd.DataFrame({
    'age_31…40': [0],
    'age_>40': [0],
    'income_high': [1],
    'income_low': [0],
    'student_yes': [1],
    'credit_rating_excellent': [0],
    'credit_rating_fair': [1]
})

# Add any missing columns with default value 0
for col in X.columns:
    if col not in sample.columns:
```

```
        sample[col] = 0

# Ensure the order of columns matches
sample = sample[X.columns]

# Make prediction
sample_prediction = model.predict(sample)
print(f"Sample Prediction: {'Yes' if sample_prediction[0] == 1 else 'No'}")
```

```
Training Accuracy: 0.86
Sample Prediction: Yes
```

[9]:
```
import pandas as pd
import numpy as np
from collections import defaultdict
import re
from sklearn.metrics import precision_score, recall_score

# Load the 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"
    ]
}

df = pd.DataFrame(data)

# Preprocess the text data
def preprocess_text(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    return text

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

# Feature extraction: Bag-of-Words model
def build_bow(df):
    all_words = ' '.join(df['Text']).split()
    vocab = set(all_words)
    return vocab

def text_to_features(text, vocab):
    text_words = text.split()
```

```
    features = {word: 0 for word in vocab}
    for word in text_words:
        if word in features:
            features[word] += 1
    return features

# Create vocabulary
vocab = build_bow(df)

# Convert text to features
def convert_to_features(df, vocab):
    return df['Text'].apply(lambda x: text_to_features(x, vocab))

X = convert_to_features(df, vocab)
y = df['Tag']

# Manual train-test split (small dataset)
X_train = X[:3]
y_train = y[:3]
X_test = X[3:]
y_test = y[3:]

# Naïve Bayes classifier
class NaiveBayesClassifier:
    def __init__(self):
        self.class_prob = {}
        self.word_probs = defaultdict(lambda: defaultdict(lambda: 1e-6))  #␣
 ↪Laplace smoothing

    def fit(self, X, y):
        classes = y.unique()
        total_docs = len(y)
        class_counts = y.value_counts()
        self.class_prob = {c: count / total_docs for c, count in class_counts.
 ↪items()}

        word_counts = {c: defaultdict(int) for c in classes}
        class_doc_counts = class_counts.to_dict()

        for idx, text_features in enumerate(X):
            doc_class = y.iloc[idx]
            for word, count in text_features.items():
                word_counts[doc_class][word] += count

        for c in classes:
            total_words_in_class = sum(word_counts[c].values())
            for word in vocab:
```

```
                self.word_probs[word][c] = (word_counts[c][word] + 1) /␣
 ↪(total_words_in_class + len(vocab))  # Laplace smoothing

    def predict(self, X):
        predictions = []
        for text_features in X:
            posteriors = {}
            for c in self.class_prob:
                prior = np.log(self.class_prob[c])
                likelihood = 0
                for word, count in text_features.items():
                    if word in self.word_probs:
                        likelihood += count * np.log(self.word_probs[word][c])
                posteriors[c] = prior + likelihood
            predictions.append(max(posteriors, key=posteriors.get))
        return np.array(predictions)

# Train the model
model = NaiveBayesClassifier()
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Evaluate the classifier
accuracy = np.mean(y_pred == y_test)
precision = precision_score(y_test, y_pred, pos_label='Sports',␣
 ↪average='binary', zero_division=0)
recall = recall_score(y_test, y_pred, pos_label='Sports', average='binary',␣
 ↪zero_division=0)

print(f"Training Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")

# Sample prediction
sample_texts = [
    "A very close game",
    "A close election",
    "The game was thrilling"
]

sample_features = [text_to_features(preprocess_text(text), vocab) for text in␣
 ↪sample_texts]
sample_predictions = model.predict(sample_features)

# Print predictions in a readable format
```

```
print("\nSample Predictions:")
for text, prediction in zip(sample_texts, sample_predictions):
    print(f"Sentence: '{text}'\nPrediction: {prediction}\n")
```

```
Training Accuracy: 1.00
Precision: 1.00
Recall: 1.00

Sample Predictions:
Sentence: 'A very close game'
Prediction: Sports

Sentence: 'A close election'
Prediction: Sports

Sentence: 'The game was thrilling'
Prediction: Not sports
```

[ ]: