DATA 221 Homework 2

Chris Low

Problem 1

1(a)

Calculate

$$P(\text{WB score} = \text{Improved}, \text{PCC symptoms} = \text{Less} \mid \text{Vaccinated})$$

```
import pandas as pd

train = pd.read_csv("data/PCC_study_train.csv")

vacc = train[train["vax_status"] == "Vaccinated"]

p_vacc = (
(vacc["WBscore"] == "Improved") &
(vacc["PCCsymp"] == "Less")
).mean()

print(p_vacc)
```

```
0.75
```

1(b)

Calculate

$$P(\text{WB score} = \text{Improved}, \text{PCC symptoms} = \text{Less} \mid \text{Unvaccinated})$$

```
unvacc = train[train["vax_status"] == "Unvaccinated"]

p_unvacc = (
(unvacc["WBscore"] == "Improved") &
(unvacc["PCCsymp"] == "Less")
).mean()

print(p_unvacc)
```

```
0.07692307692307693
```

1(c)

```
import pandas as pd

def make_table(df, label):
    wb = (
        df["WBscore"]
        .value_counts(normalize=True)
        .rename("Probability")
        .reset_index()
        .rename(columns={"index": "WBscore"})
    )

    pcc = (
        df["PCCsymp"]
        .value_counts(normalize=True)
        .rename("Probability")
        .reset_index()
        .rename(columns={"index": "PCCsymp"})
    )

    print(f"\n{label} - Well-being score")
    display(wb)

    print(f"\n{label} - PCC symptoms")
    display(pcc)
```

```
make_table(vacc, "Vaccinated")
make_table(unvacc, "Unvaccinated")
```

```
Vaccinated - Well-being score
```

	WBscore	Probability
0	Improved	0.777778
1	Unchanged	0.138889
2	Worsened	0.083333

```
Vaccinated - PCC symptoms
```

	PCCsymp	Probability
0	Less	0.861111
1	Same	0.083333
2	More	0.055556

```
Unvaccinated - Well-being score
```

	WBscore	Probability
0	Unchanged	0.512821
1	Worsened	0.307692
2	Improved	0.179487

```
Unvaccinated - PCC symptoms
```

	PCCsymp	Probability
0	Same	0.512821
1	More	0.282051

	PCCsymp	Probability
2	Less	0.205128

1(d)

Assuming conditional independence, write a function that computes

$$P(\text{outcome} \mid \text{class})$$

```
def naive_bayes_likelihood(
    wb_score,
    pcc_symp,
    wb_probs,
    pcc_probs,
    class_prob
):
    """
    wb_score: The observed WBscore (e.g., "Improved")
    pcc_symp: The observed PCCsymp (e.g., "Less")
    wb_probs: Dictionary of conditional probs for WBscore given the class
    pcc_probs: Dictionary of conditional probs for PCCsymp given the class
    class_prob: The prior probability of the class (e.g., P(Vaccinated))
    """
    return (
        wb_probs[wb_score] *
        pcc_probs[pcc_symp] *
        class_prob
    )
```

1(e)

```
p_vaccinated = (train["vax_status"] == "Vaccinated").mean()
p_unvaccinated = (train["vax_status"] == "Unvaccinated").mean()

vacc_wb_probs = vacc["WBscore"].value_counts(normalize=True)
vacc_pcc_probs = vacc["PCCsymp"].value_counts(normalize=True)

unvacc_wb_probs = unvacc["WBscore"].value_counts(normalize=True)
unvacc_pcc_probs = unvacc["PCCsymp"].value_counts(normalize=True)
```

```
# likelihoods for WB = Worsened, PCC = Same
lik_vacc = naive_bayes_likelihood(
    "Worsened",
    "Same",
    vacc_wb_probs,
    vacc_pcc_probs,
    p_vaccinated
)

lik_unvacc = naive_bayes_likelihood(
    "Worsened",
    "Same",
    unvacc_wb_probs,
    unvacc_pcc_probs,
    p_unvaccinated
)

print("Vaccinated likelihood:", lik_vacc)
print("Unvaccinated likelihood:", lik_unvacc)
```

```
Vaccinated likelihood: 0.003333333333333333
Unvaccinated likelihood: 0.08205128205128205
```

1(f)

```
test = pd.read_csv("data/PCC_study_test.csv")

def predict_vax_status(row):
    lv = naive_bayes_likelihood(
        row["WBscore"],
        row["PCCsymp"],
        vacc_wb_probs,
        vacc_pcc_probs,
        p_vaccinated
    )
    lu = naive_bayes_likelihood(
        row["WBscore"],
        row["PCCsymp"],
        unvacc_wb_probs,
```

```
        unvacc_pcc_probs,
        p_unvaccinated
    )
    return "Vaccinated" if lv > lu else "Unvaccinated"

test["predicted_vax_status"] = test.apply(predict_vax_status, axis=1)

test.head()
```

	Unnamed: 0	vax_status2	WBscore	PCCsymp	predicted_vax_status
0	1	Vaccinated	Improved	Less	Vaccinated
1	2	Vaccinated	Improved	Less	Vaccinated
2	3	Vaccinated	Improved	Less	Vaccinated
3	4	Vaccinated	Worsened	Less	Vaccinated
4	5	Vaccinated	Unchanged	Less	Vaccinated

1(g)

```
from sklearn.naive_bayes import CategoricalNB
from sklearn.preprocessing import LabelEncoder

label_wb = LabelEncoder()
label_pcc = LabelEncoder()
label_y = LabelEncoder()

X_train = pd.DataFrame({
    "WBscore": label_wb.fit_transform(train["WBscore"]),
    "PCCsymp": label_pcc.fit_transform(train["PCCsymp"])
})

y_train = label_y.fit_transform(train["vax_status"])

X_test = pd.DataFrame({
    "WBscore": label_wb.transform(test["WBscore"]),
    "PCCsymp": label_pcc.transform(test["PCCsymp"])
})

model = CategoricalNB()
model.fit(X_train, y_train)
```

```
sklearn_preds = label_y.inverse_transform(model.predict(X_test))

# KEY: Compare with manual predictions
comparison = test["predicted_vax_status"].values == sklearn_preds

print("Are all predictions identical?", comparison.all())
```

```
Are all predictions identical? True
```

We see that running this code confirms a 100% match between the manual calculation and the Scikit-Learn implementation.

Problem 2

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from scipy.stats import norm
```

2(a)

```
# Load Data
iris = pd.read_csv('data/iris.csv')
df = iris.copy()

# Split (80% Train, 20% Test)
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)

print(f"Training set size: {len(train_df)}")
print(f"Test set size: {len(test_df)}")
```

```
Training set size: 120
Test set size: 30
```

```
print(train_df.columns.tolist())
```

```
['rownames', 'Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width', 'Species']
```

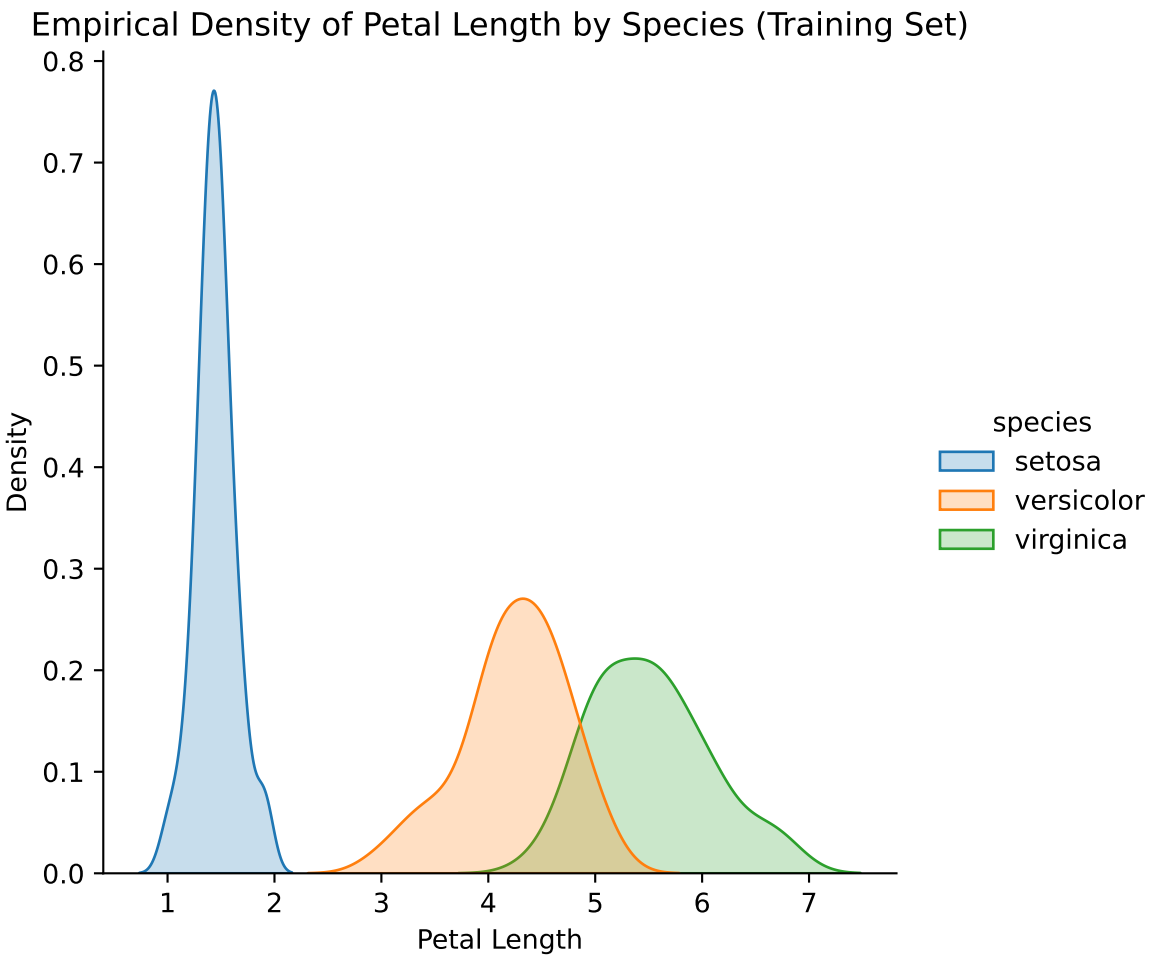
```
train_df.columns = (
    train_df.columns
    .str.strip()
    .str.lower()
    .str.replace(" ", ".", regex=False)
)

test_df.columns = (
    test_df.columns
    .str.strip()
    .str.lower()
    .str.replace(" ", ".", regex=False)
)
```

2(b)

```
sns.displot(
    data=train_df,
    x="petal.length",
    hue="species",
    kind="kde",
    fill=True
)

plt.title("Empirical Density of Petal Length by Species (Training Set)")
plt.xlabel("Petal Length")
plt.ylabel("Density")
plt.show()
```



We plot kernel density estimates using seaborn's KDE, which provides a smooth approximation of the empirical distribution of petal length for each species.

2(c)

```
petal_col = "petal.length"

summary = (
    train_df
    .groupby("species")[petal_col]
    .agg(["count", "mean", "std"])
)
```

```
print(summary)
```

```
            count      mean       std
species
setosa         40  1.450000  0.183973
versicolor     41  4.241463  0.481132
virginica      39  5.520513  0.541528
```

2(d)

```
x = 5.12
petal_col = "petal.length"
likelihoods = {}

for sp in train_df["species"].unique():
    vals = train_df.loc[
        train_df["species"] == sp, petal_col
    ]

    mu = vals.mean()
    sigma = vals.std()

    likelihoods[sp] = norm.pdf(x, loc=mu, scale=sigma)

likelihoods
```

```
{'setosa': np.float64(8.385828085894835e-87),
 'versicolor': np.float64(0.1565430450019126),
 'virginica': np.float64(0.5604136072758437)}
```

Note that each species appears roughly equally often, so $P(y)$ is approximately the same and hence the prior can be ignored in Naives Bayes classification.

```
print("I would classify the species as " + max(likelihoods, key=likelihoods.get) + ".")
```

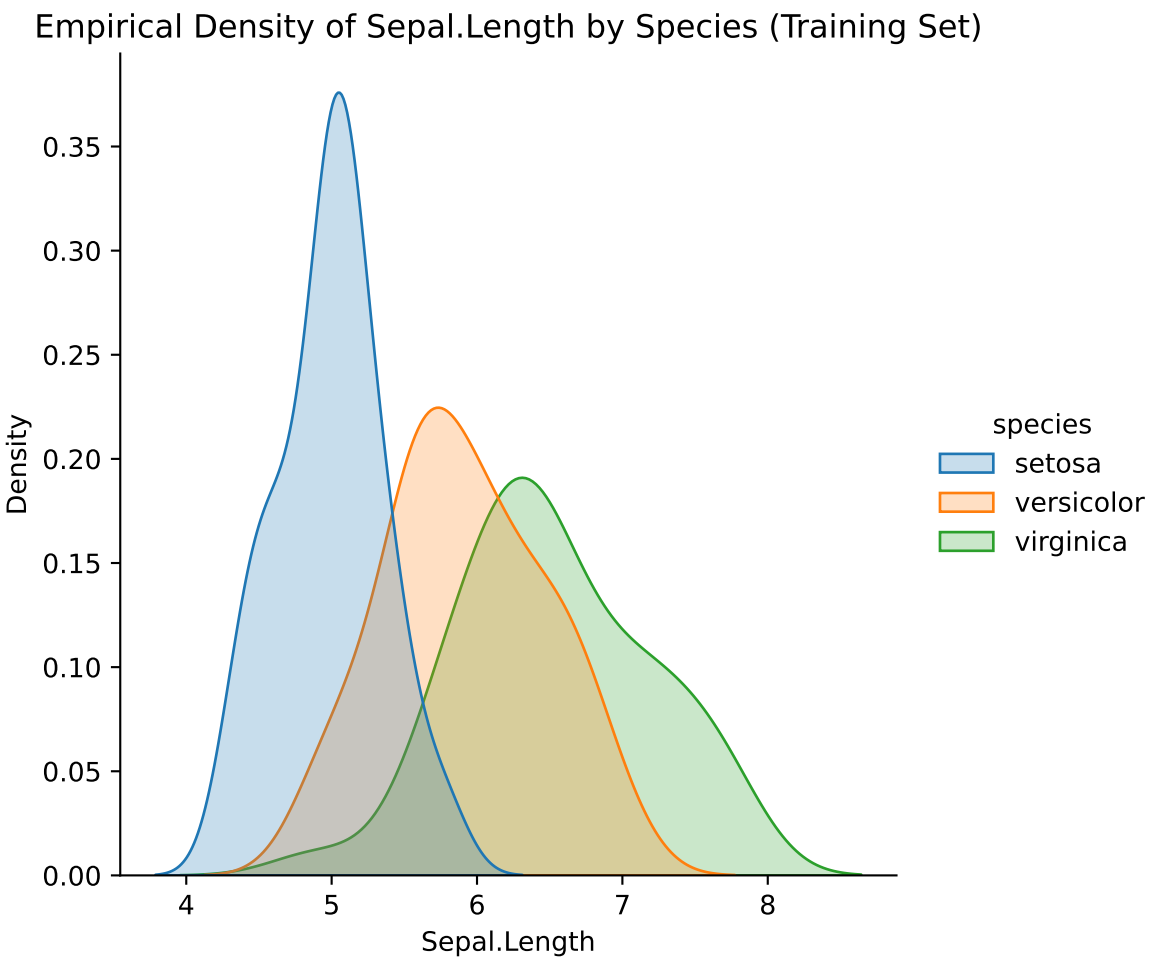
```
I would classify the species as virginica.
```

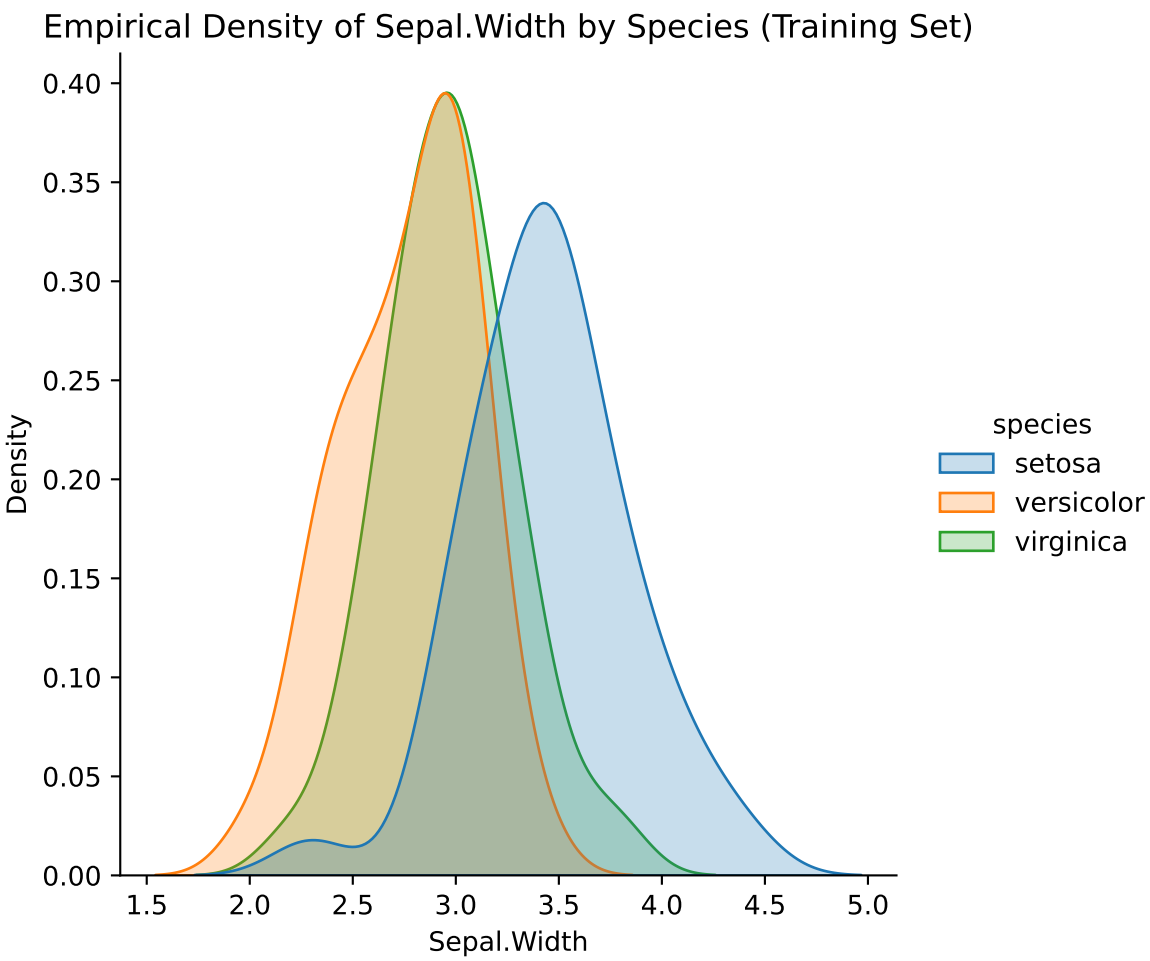
2(e)

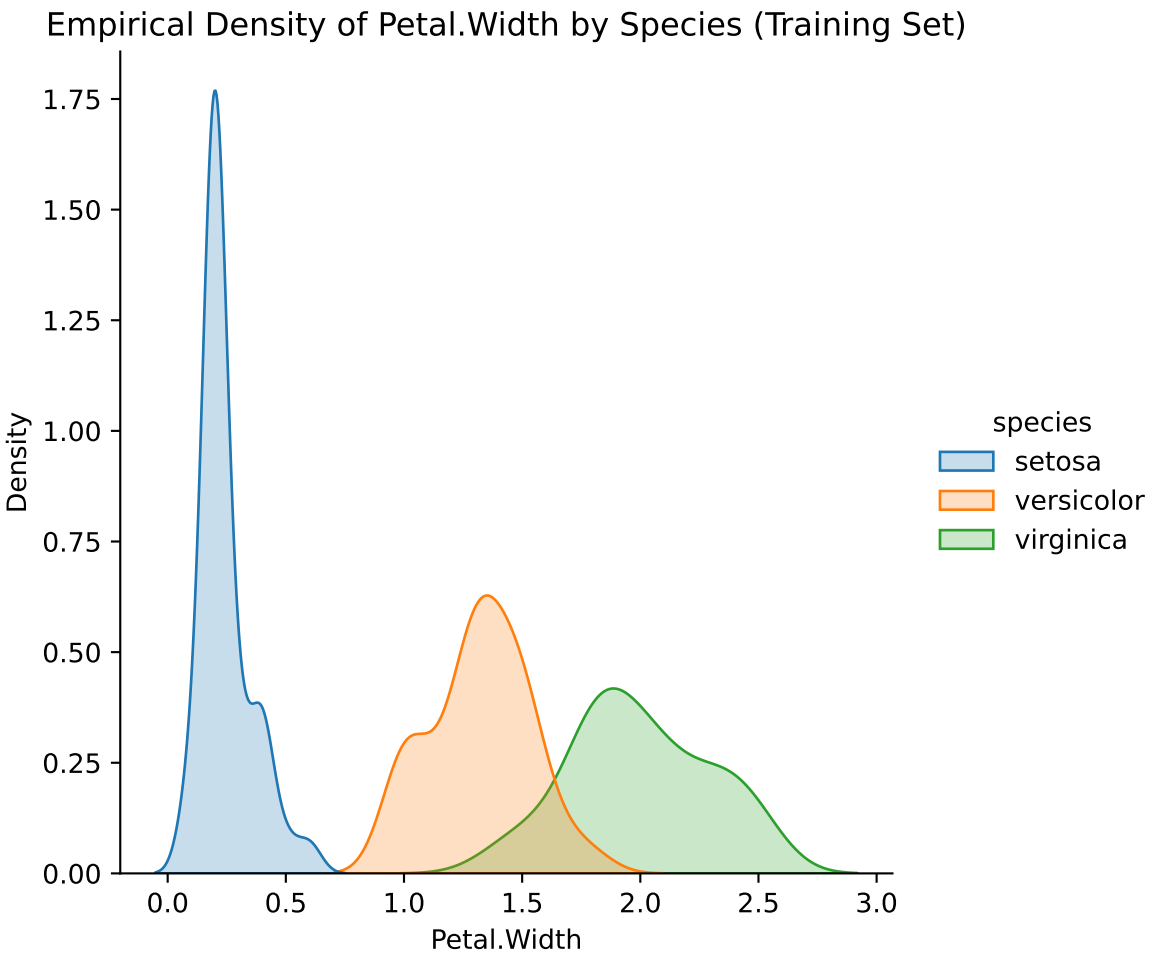
```
features = [
    "sepal.length",
    "sepal.width",
    "petal.width"
]

for feature in features:
    sns.displot(
        data=train_df,
        x=feature,
        hue="species",
        kind="kde",
        fill=True
    )

    plt.title(
        f"Empirical Density of {feature.replace('_', ' ').title()} "
        "by Species (Training Set)"
    )
    plt.xlabel(feature.replace('_', ' ').title())
    plt.ylabel("Density")
    plt.show()
```


Empirical Density of Sepal.Length by Species (Training Set)
Density
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
4
5
6
7
8
Sepal.Length
species
setosa
versicolor
virginica


Empirical Density of Sepal.Width by Species (Training Set)
Density
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Sepal.Width
species
setosa
versicolor
virginica



2(f)

```
x_new = {
    "sepal.length": 5.42,
    "sepal.width": 3.81,
    "petal.length": 4.23,
    "petal.width": 2.15
}

features = list(x_new.keys())
scores = {}
```

```
for sp in train_df["species"].unique():
    subset = train_df[train_df["species"] == sp]

    log_likelihood = 0

    for feature in features:
        mu = subset[feature].mean()
        sigma = subset[feature].std()

        log_likelihood += np.log(
            norm.pdf(x_new[feature], loc=mu, scale=sigma)
        )

    # equal class priors like (d), so no need to add log(P(sp))
    scores[sp] = log_likelihood

scores
```

```
{'setosa': np.float64(-274.28402662307417),
 'versicolor': np.float64(-13.411970487031656),
 'virginica': np.float64(-8.175213213592475)}
```

```
print("The species is " + max(scores, key=scores.get) + ".")
```

```
The species is virginica.
```

Problem 3

3(a)

```
features = ["sepal.length", "sepal.width", "petal.length", "petal.width"]

predictions = []

for _, row in test_df.iterrows():
    scores = {}

    for sp in train_df["species"].unique():
        subset = train_df[train_df["species"] == sp]
        log_likelihood = 0

        for feature in features:
            # Data from train
            mu = subset[feature].mean()
            sigma = subset[feature].std()

            # Prediction on test
            log_likelihood += np.log(
                norm.pdf(row[feature], loc=mu, scale=sigma)
            )

        scores[sp] = log_likelihood

    predictions.append(max(scores, key=scores.get))

test_df = test_df.copy()
test_df["predicted_species"] = predictions
test_df[["species", "predicted_species"]].head()
```

	species	predicted_species
73	versicolor	versicolor
18	setosa	setosa
118	virginica	virginica
78	versicolor	versicolor
76	versicolor	versicolor

3(b)

```
confusion = pd.crosstab(
    test_df["species"],
    test_df["predicted_species"],
    rownames=["Actual"],
    colnames=["Predicted"]
)

confusion
```

Predicted Actual	setosa	versicolor	virginica
setosa	10	0	0
versicolor	0	9	0
virginica	0	0	11

3(c)

```
accuracy = (
    test_df["species"] == test_df["predicted_species"]
).mean()

print(f"Accuracy: {accuracy:.3f}")
```

```
Accuracy: 1.000
```

3(d)

```
errors = test_df[test_df["species"] != test_df["predicted_species"]]

errors.value_counts(["species", "predicted_species"])
```

```
Series([], Name: count, dtype: int64)
```

The classifier achieved 100% accuracy on the test set, so no misclassifications were made. As a result, there is no "most common" error in this evaluation. (It is possible that petal measurements, which we saw in the empirical density plots from Problem 2, show strong separation between the species in the feature space.)

Problem 4

4(a)

```
spam_df = pd.read_csv("data/spam.csv", encoding="latin-1")

print(spam_df.columns)

# Remove the unnecessary columns
spam_df = spam_df.iloc[:, :2]
spam_df.columns = ["label", "message"]

print(spam_df.shape)
```

```
Index(['v1', 'v2', 'Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'], dtype='object')
(5572, 2)
```

4(b)

```
spam_messages = spam_df[spam_df["label"] == "spam"]
ham_messages = spam_df[spam_df["label"] == "ham"]

print("Spam messages:", len(spam_messages))
print("Ham messages:", len(ham_messages))
```

```
Spam messages: 747
Ham messages: 4825
```

4(c)

```
spam_string = spam_messages["message"].str.cat(sep=" ")
ham_string = ham_messages["message"].str.cat(sep=" ")

spam_tokens = spam_string.split()
ham_tokens = ham_string.split()

print("Number of spam tokens:", len(spam_tokens))
print("Number of ham tokens:", len(ham_tokens))
```

```
Number of spam tokens: 17817
Number of ham tokens: 68518
```

4(d)

```
import nltk
from nltk import FreqDist

spam_freq = FreqDist(spam_tokens)
ham_freq = FreqDist(ham_tokens)
```

4(e)

```
print("Unique tokens in spam dictionary:", len(spam_freq))
print("Unique tokens in ham dictionary:", len(ham_freq))
```

```
Unique tokens in spam dictionary: 4312
Unique tokens in ham dictionary: 12479
```

Note: There might be small differences in the number of unique tokens due to case sensitivity and punctuation being treated as distinct tokens. I would assume that since the same preprocessing is used consistently throughout the analysis, these differences do not affect the validity of the results.

```
word = "Sorry"

p_sorry_ham = ham_freq[word] / sum(ham_freq.values())
p_sorry_spam = spam_freq[word] / sum(spam_freq.values())

print(f"P('Sorry' | Ham) = {p_sorry_ham:.5f}")
print(f"P('Sorry' | Spam) = {p_sorry_spam:.5f}")
```

```
P('Sorry' | Ham) = 0.00074
P('Sorry' | Spam) = 0.00006
```

4(f)

```
def message_probability(message, freq_dist):
    tokens = message.split()
    total_tokens = sum(freq_dist.values())

    prob = 1
    for token in tokens:
        prob *= freq_dist[token] / total_tokens

    return prob
```

4(g)

```
# P (sentnece | ham) and P (sentence | spam)
msg = "Sorry my roommates took forever, it ok if I come by now?"

msg_clean = msg.replace(",", "").replace("?", "").lower()
p_msg_ham = message_probability(msg_clean, ham_freq)
p_msg_spam = message_probability(msg_clean, spam_freq)

print("P(message | ham) =", p_msg_ham)
print("P(message | spam) =", p_msg_spam)
```

```
P(message | ham) = 3.832146372029724e-36
P(message | spam) = 0.0
```

4(h)

The ham probability is tiny because we multiply many small word probabilities. The spam probability is 0 because at least one word in the message had count 0 in the spam dictionary, so the whole product becomes 0.

To address this, I assign a **fixed small penalty probability** to words with zero frequency. I intentionally avoid using a penalty that depends on the total number of tokens in each class, since the ham corpus is substantially larger than the spam corpus. A size-dependent penalty would therefore penalize unseen words in ham more severely than in spam, introducing an unintended bias. Using a constant, very small penalty instead ensures that unseen words are handled consistently across classes and that classification is driven by the relative evidence from observed words rather than differences in corpus size.

```
total_ham = len(ham_tokens)
total_spam = len(spam_tokens)

def log_prob_word(word, freq_dist, total_tokens):
    """
    Returns log(P(w|Class)).
    Uses a fixed small penalty (1e-10) if word is not found to avoid
    biasing the model based on the total number of tokens in the class.
    """
    penalty = 1e-10
    count = freq_dist[word]

    if count == 0:
        return np.log(penalty)

    return np.log(count / total_tokens)


def message_log_odds(message, ham_freq, spam_freq, total_ham, total_spam):
    """
    Returns log odds of message under ham vs spam, i.e.:
    log P(message | ham) - log P(message | spam)
    """
    words = message.split()

    log_ham = 0
    log_spam = 0

    for w in words:
        log_ham += log_prob_word(w, ham_freq, total_ham)
        log_spam += log_prob_word(w, spam_freq, total_spam)

    return log_ham - log_spam
```

4(i)

Using the log-likelihoods from part (h), we score each message by computing the log odds ratio of the message being ham versus spam. (i.e. return log_ham - log_spam)

Positive scores mean that messages that are more likely to be ham, while negative scores indicate messages that are more likely to be spam.

```
def score_message(message):
    """
    Returns the log-odds score for a message being ham vs spam.
    Positive => ham-like, Negative => spam-like.
    """
    return message_log_odds(
        message,
        ham_freq,
        spam_freq,
        total_ham,
        total_spam
    )
```

4(j)

```
msg = "Sorry my roommates took forever, it ok if I come by now?"
msg_clean = msg.replace(",", "").replace("?", "")

score_j = score_message(msg_clean)
print("Score of the message:", score_j)
```

```
Score of the message: 56.93076605866902
```

Because the score is positive, we would predict this as Ham.

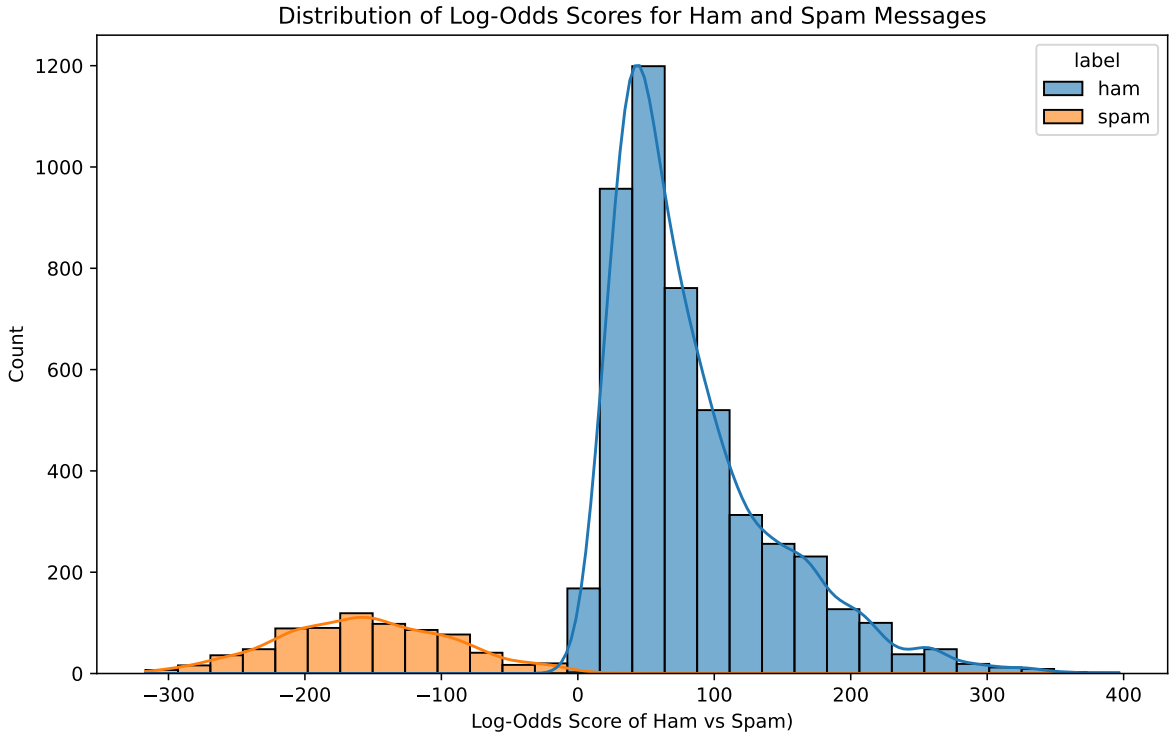
4(k)

```
spam_df["token_len"] = spam_df["message"].apply(lambda x: len(x.split()))
short_msgs = spam_df[spam_df["token_len"] <= 50].copy()

short_msgs["score"] = short_msgs["message"].apply(score_message)

plt.figure(figsize=(10, 6))
sns.histplot(
    data=short_msgs,
    x="score",
    hue="label",
```

```
    bins=30,
    alpha=0.6,
    kde=True
)
plt.xlabel("Log-Odds Score of Ham vs Spam)")
plt.ylabel("Count")
plt.title("Distribution of Log-Odds Scores for Ham and Spam Messages")
plt.show()
```



Description: Ham messages tend to have larger or more positive log-odds scores, while spam messages tend to have smaller or negative scores. We do see that there is some overlap between the two distributions. However, spam messages are generally shifted toward lower values, so this shows that our scoring function is effective at separating ham from spam.

Note: In this analysis, we compare messages using the log-likelihood ratio $\log P(\text{message} \mid \text{ham}) - \log P(\text{message} \mid \text{spam})$ and do not explicitly include class priors. Since the prior probabilities are constant across messages, omitting them does not affect the relative ordering of the scores.

```
print(short_msgs.columns)
```

```
Index(['label', 'message', 'token_len', 'score'], dtype='object')
```

Problem 5

5(a)

```
# Predict spam if score < 1 else ham
short_msgs["pred_label"] = np.where(
    short_msgs["score"] < 1, "spam", "ham"
)

cm = pd.crosstab(
    short_msgs["label"],
    short_msgs["pred_label"],
    rownames=["Actual"],
    colnames=["Predicted"]
)

cm
```

Predicted Actual	ham	spam
ham	4755	8
spam	2	745

Computing metrics:

```
TP = cm.loc["spam", "spam"]
FN = cm.loc["spam", "ham"]
FP = cm.loc["ham",  "spam"]
TN = cm.loc["ham",  "ham"]

accuracy = (TP + TN) / (TP + TN + FP + FN)

# Sensitivity = TPR = recall for spam
```

```
sensitivity = TP / (TP + FN) if (TP + FN) > 0 else np.nan

# Specificity = TNR for ham
specificity = TN / (TN + FP) if (TN + FP) > 0 else np.nan

print("Confusion matrix counts:")
print(f"TP={TP}, FN={FN}, FP={FP}, TN={TN}\n")

print(f"Accuracy:     {accuracy:.4f}")
print(f"Sensitivity:  {sensitivity:.4f}")
print(f"Specificity:  {specificity:.4f}")
```

```
Confusion matrix counts:
TP=745, FN=2, FP=8, TN=4755

Accuracy:     0.9982
Sensitivity:  0.9973
Specificity:  0.9983
```

5(b)

My versions above do not allow a flexibility in penalty. Below, I updated them to reflect that:

```
def log_prob_word(word, freq_dist, total_tokens, penalty=1e-10):
    """
    Returns log(P(w|Class)).
    Uses a fixed small penalty (1e-10).
    """
    count = freq_dist[word]

    if count == 0:
        return np.log(penalty)

    return np.log(count / total_tokens)


def message_log_odds(message, ham_freq, spam_freq, total_ham, total_spam, penalty_mult=1.0):
    """
    Returns log odds of message under ham vs spam.
    """
```

```
    words = message.split()

    log_ham = 0
    log_spam = 0

    for w in words:
        log_ham += log_prob_word(w, ham_freq, total_ham, penalty_mult)
        log_spam += log_prob_word(w, spam_freq, total_spam, penalty_mult)

    return log_ham - log_spam

def score_message(message, penalty_mult=1.0):
    """
    Log-odds score for ham vs spam.
    """
    return message_log_odds(
        message,
        ham_freq,
        spam_freq,
        total_ham,
        total_spam,
        penalty_mult
    )
```

Now modifying to different errors:

```
from sklearn.metrics import confusion_matrix, accuracy_score


spam_df["token_len"] = spam_df["message"].apply(lambda x: len(x.split()))
short_msgs = spam_df[spam_df["token_len"] <= 50].copy()


penalties = [1e-5, 1e-10, 1e-20]

print(f"{'Penalty':<12} | {'Accuracy':<10} | {'Sensitivity':<12} | {'Specificity':<11}")
print("-" * 60)

for penalty in penalties:
    # Recompute scores
    scores = short_msgs["message"].apply(lambda x: score_message(x, penalty_mult=penalty))
```

```
    # Predict spam if score < 1
    preds = scores.apply(lambda x: 1 if x < 1 else 0)  # 1=spam, 0=ham

    # Confusion matrix (spam is positive)
    y_true = short_msgs["label"].map({"spam": 1, "ham": 0})

    cm = confusion_matrix(y_true, preds, labels=[1, 0])
    tp, fn = cm[0]
    fp, tn = cm[1]

    accuracy = accuracy_score(y_true, preds)
    sensitivity = tp / (tp + fn) if (tp + fn) > 0 else np.nan
    specificity = tn / (tn + fp) if (tn + fp) > 0 else np.nan

    print(f"{penalty:<12} | {accuracy:<10.4f} | {sensitivity:<12.4f} | {specificity:<11.4f}"
```

```
Penalty      | Accuracy   | Sensitivity  | Specificity
------------------------------------------------------------
1e-05        | 0.9840     | 0.9906       | 0.9830
1e-10        | 0.9982     | 0.9973       | 0.9983
1e-20        | 0.9985     | 1.0000       | 0.9983
```

Interpretation: As the penalty for unseen words becomes smaller (e.g. towards power of -20), spam messages containing rare or unseen tokens are penalized more heavily. This makes them easier to identify.

This **increases sensitivity** (how many real spam messages we correctly catch), eventually reaching 1.0, while **specificity** (how many real ham messages we correctly keep as ham) remains largely **unchanged** because ham messages tend to consist of more common words. Overall **accuracy increases only slightly** since the classifier already performs well on the majority of messages.