NLP-disaster-NB

October 19, 2025

1 NLP Disaster Tweets Continued

1.1 Bernoulli Naive Bayes

1.1.1 Import preprocessed data

I imported preprocessed data using the same preprocessing steps from HW2. A summary of the preprocessing steps is shown below.

1. Lowercase all letters
2. Only keep alphanumerics, # and @
3. Keep the http or https part of links, get rid of the rest of the link
4. Keep the @ part of @usernames, get rid of the actual username
5. Keep the # of #topics, keep both topic and # but separate them
6. Lemmatize all words
7. Remove common stop words (and, or, the ..)

[2]:
```
import pandas as pd

train = pd.read_csv("/home/jiwonjjeong/cornell-tech/CT-applied-machine-learning/
 ↪hw-2/data-from-NLP-disaster-tweets/preprocessed/train.csv")
test = pd.read_csv("/home/jiwonjjeong/cornell-tech/CT-applied-machine-learning/
 ↪hw-2/data-from-NLP-disaster-tweets/preprocessed/test.csv")
val = pd.read_csv("/home/jiwonjjeong/cornell-tech/CT-applied-machine-learning/
 ↪hw-2/data-from-NLP-disaster-tweets/preprocessed/validate.csv")
print(train.shape)
train[10:20]
```

```
(5329, 5)
```

[2]:
```
       id      keyword                    location  \
10  10691        wreck  ?currently writing a book?
11   7148     mudslide    Malibu/SantaFe/Winning!
12   2017   casualties             Canadian bread
13    262    ambulance                  Amsterdam
14   4468  electrocuted                       NaN
15   4176        drown                        NaN
16  10243      volcano        West Coast, Cali USA
17    681       attack                     Mumbai
18   1399  body%20bag                   New York
```

```
19    296   annihilated                Higher Places

                                                    text  target
10  friggin wreck destiel suck read vine descripti…       0
11  sterlingscott red carpet fundrais oso mudslid …       1
12  @ honest peopl want go rampag let use hand fee…       0
13  http twelv fear kill pakistani air ambul helic…       1
14  got electrocut last night work first time life…       0
15  older nativ australian believ ocean creat urin…       0
16            architect behind kany west volcano http       0
17  india shud give ani evid pakthey share terrori…       1
18  auth loui vuitton brown saumur 35 cross bodi s…       0
19  episod trunk annihil freiza cleanest shit ever…       0
```

1.1.2 Create feature vectors using Naive-Bayes assumption

I will assume that each word can be considered independently, allowing me to represent the inclusion of each word as its own individual feature. Below, I am creating the word dictionary to use as features based on the training set words.

[3]:
```
# Get set of unique words in the training set
unique_words = set(' '.join(train['text']).split())
len(unique_words)
words = list(unique_words)
```

[4]:
```
# Generate 10339 dimensional vector to represent each sample and embedd using␣
 ↪naive bayes assumption
train_dict = {}
val_dict = {}
test_dict = {}
for word in words:
    train_dict[f'has_{word}'] = train['text'].str.contains(word, na=False).
 ↪astype(int)
    val_dict[f'has_{word}'] = val['text'].str.contains(word, na=False).
 ↪astype(int)
    test_dict[f'has_{word}'] = test['text'].str.contains(word, na=False).
 ↪astype(int)
```

[5]:
```
# Concatenate the dictionary into dataframe
train_no_target = pd.DataFrame(train_dict)
val_no_target = pd.DataFrame(val_dict)
test = pd.DataFrame(test_dict)
train = pd.concat([train_no_target,train["target"]], axis=1)
val = pd.concat([val_no_target,train["target"]], axis=1)
train.head()
```

```
[5]:    has_wasnt  has_cree  has_rescueadoptionloc  has_tcop5zicjudxo  has_presley  \
     0          0         0                      0                  0            0
     1          0         0                      0                  0            0
     2          0         0                      0                  0            0
     3          0         0                      0                  0            0
     4          0         0                      0                  0            0

        has_sel  has_damnwa  has_hitter  has_humanitarian  has_antonio  …  \
     0        0           0           0                 0            0  …
     1        0           0           0                 0            0  …
     2        0           0           0                 0            0  …
     3        0           0           0                 0            0  …
     4        0           0           0                 0            0  …

        has_stuffin  has_declin  has_vermilion  has_pjnet  has_convert  has_cfc  \
     0            0           0              0          0            0        0
     1            0           0              0          0            0        0
     2            0           0              0          0            0        0
     3            0           0              0          0            0        0
     4            0           0              0          0            0        0

        has_homeland  has_nh  has_bruv  target
     0             0       0         0       0
     1             0       0         0       1
     2             0       0         0       1
     3             0       0         0       1
     4             0       0         0       0

     [5 rows x 10340 columns]
```

1.1.3 Compute model parameters

I used the closed-form solution for the parameters for a Bernoulli Naive Bayes model. This results in an analytical solution for the parameters psis which helps calculate P(x|y) and parameters phis which helps calculate P(y).

I also used Laplace smoothing to make sure that words outside of the training/model vocabulary will give a P(x|y) probability of 0.5 (no bias towards either 0 or 1).

```
[6]: # Compute model parameters
     # Code adapted from lecture notes
     import numpy as np
     alpha = 1
     n = train.shape[0]
     d = train.shape[1] - 1
     K = 2

     psis = np.zeros([K,d])
```

```
phis = np.zeros([K])

y_train = train["target"]
X_train = train.drop(columns=["target"])

for k in range(K):
    X_k = X_train[y_train == k]
    psis[k] = (X_k.sum(axis=0) + alpha)/ (2*alpha + X_k.shape[0])
    phis[k] = X_k.shape[0]/ float(n)

print(phis)
```

```
[0.56746106 0.43253894]
```

1.1.4 Compute predictions from model

Using the modeled distribution of inputs for a given class P(x|y=k) and the modeled distribution of the class P(y=k), I can compute the predicted P(y=k|x) for each class k by multiplying these two values. To avoid numerical instability, I added the logs of these probabilities.

Then, I determined that the predicted class for a given input should be the class that has the highest probability for that input.

[7]:
```
# Compute predicitons from model
# Code adapted from lecture notes
# And code adapted from ChatGPT (2025)
# Prompt: "how can i make sure i dont crash due to memory in this funciton..."
def nb_predictions(x, psis, phis):
    x = x.to_numpy()

    n, d = x.shape
    K = psis.shape[0]

    logpy = np.log(phis).reshape([K])
    logpyx = np.zeros((K, n))

    psis = psis.clip(1e-14, 1-1e-14)
    log_psis = np.log(psis)
    log_one_minus_psis = np.log(1-psis)

    for i in range(n):
        xi = x[i]  # shape (d,)
        logpxy = xi * log_psis + (1-xi) * log_one_minus_psis  # shape (K,d)
        logpyx[:, i] = logpxy.sum(axis=1) + logpy

    return logpyx.argmax(axis=0), logpyx

idx, logpyx = nb_predictions(X_train, psis, phis)
```

```
print(idx[:10])
```

```
[1 0 1 0 0 1 1 0 0 1]
```

1.2 Compare models

I applied the model to calculate the predicted class of the validation/development set. Then I used the F1 score to evaluate the classification model's precision and recall.

[8]:
```
# Apply to validation set
X_val = val.drop(columns=["target"])
idx_val, logpyx = nb_predictions(X_val, psis, phis)
print(idx_val[:10])
```

```
[0 0 0 0 0 0 0 0 0 1]
```

[9]:
```
# Evaluate on validation set
from sklearn.metrics import f1_score
f1 = f1_score(idx_val, val["target"])
print("F1 score:", f1)
```

```
F1 score: 0.2289348171701113
```

[14]:
```
# Compare models to HW2
```

Recall, the best model between the N-gram and bag of words approach had an F1 score of 0.743139407244786. The best model of overall was with the M=3 feature set using L2 regularization with C=1 with the bag of words model.

When the model was rebuilt using the entire training data and submitted to Kaggle, the F1 score rose to 0.7537796976241901.

By comparison, the Naive-Bayes approach had an F1 score of 0.2289348171701113. Thus, it seems that the Naive-Bayes approach is significantly worse at generalizing and predicting if tweets are disasters on new data.

I found this low score so surprising that I decided to check if my model was overfitting.

[11]:
```
# See F1 score for testing data
from sklearn.metrics import f1_score
f1 = f1_score(idx, train["target"])
print("Testing F1 score:", f1)
```

```
Testing F1 score: 0.8132992327365729
```

Because the testing F1 score is very high while validation F1 score is low, it is observed that my NB model is overfitting on the testing data. I think one reason is for this is that I included all words in the vocabulary as features, even if it only appears once in the training data. This is likely created too many features compared to samples and induced overfitting on these poorly represented single-occurence words.

A better alternative is to only take the top 1000 highest frequency words, which is better done through the sklearn CountVectorizer library.

1.2.1 Comparison of generative and discriminative models

The previous bag of words and N-gram models are discriminative models, where they explicitly determine P(y|x) by modeling the data generative function between the input x and label y. This Bernoulli Naive-Bayes model is a generative model, where it models the distribution of inputs for each label. It is still able to perform discriminative-like predictions using the Bayes Rule (this is different from the NB assumption).

One positive of generative models is that it can perform discriminative-model-like predictions while also performing its task to generate new inputs with high probability of class k. Discriminative models do not have the capability to generate new inputs.

An additional positive is that generative models are more descriptive of the dataset. Discriminative models only classify classes of datasets by determining decision boundaries between classes. On the other hand, generative models show the distributions of the data. This means it can capture clustering of data within classes, showing their means and variances. This information about the distribution can help with more tasks like data imputation, handling missing data, and anomaly detection. Discriminative models do not have this additional capability.

On a performance level, discriminative models do better when there is more data close to the true decision boundary. So the difference in predictive power of a discriminative vs generative supervised learning model will depend on the dataset and underlying decision boundaries. For example, with less data, generative models will perform better.

On a negative for generative models, they require stronger assumptions to realistically learn distributions. Thus, there is a higher risk of inaccuracies coming for inappropriate, strong assumptions. The comparison of assumptions will be discussed in the next section.

Additionally, discriminative models are easier to implement and understand on an interpretative level for the task of classification prediction. We can simply look at the model parameter magnitudes to determine the most important and least important features with a discrinative model. By comparison, a generative model requires more steps (Bayes Rule) to perform predictive classification tasks. These complexities become harder to track and compare as we grow from a binary classification to a multiclassification task. By comparison, comparing magnitudes of parameters in a discrimnative classification is more straight-forward.

1.2.2 Asssumptions of generative and discriminative models

The Naive Bayes assumption is that the features for a given label can be considered independent from eachother. In this case, the NB assumption manifests as the presence of some specific word for non-disaster tweets like "happy" is independent from the word "help".

There is also an assumption of independence in disciminative models. Here, independence means that two different features each contribute an independent effect on the decision to classify a tweet as disaster or not. While this sounds very similar, in the N-gram and bag of words model, we were able to construct our features the a unique collection of words. For example, a vocabulary of 3 words would cause features to have the form of (0,0,0), (0,0,1), (0,1,0), (0,1,1),... (1,1,1). This means that there is no assumption of individual words having independent effect on the classification from eachother.

In terms of efficiency, the NB assumption is definitely important and very efficient for memory. In the example above, we saw that a vocabulary of 3 requires 9 different features for each combination of the word being present or absent. By using the NB assumption, we can represent the same vocabulary of 3 words with only 3 features. With each feature being whether each word is present or not, independently. In a general form, the NB assumption allows reduction of parameters from 2^(d-1) to ~Kd, where d is the vocabulary size and K is the number of classes. Using the NB assumption allows the inclusion of more vocabulary while maintaining memory constraints.

In terms of validity, there is some appropriateness. For most unrelated words like "girl" and "computer", assuming independence of these word occurences is valid. However, we can also construct cases where words together create different meanings. For example, "not" and "happy" is very different from the independent occurence of "not" and "happy". Some words even change semantic meaning in a dependent way to another word. For example "bank" means different things when paired with "river" versus "Chase" (thus showing dependence between words). In general, independence is a poor assumption in natural language text since many words have context around other words.

GDA

October 19, 2025

1 Gaussian Discriminant Analysis

1.1 Importing and Splitting

I split the dataset into a 30-70 test-train split.

[1]:
```
from sklearn import datasets
iris = datasets.load_iris(as_frame=True)
```

[2]:
```
import pandas as pd
data = iris["frame"]
data.head()
```

[2]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0       0
1       0
2       0
3       0
4       0
```

[4]:
```
# Randomly split 30-70 test-train
import numpy as np

indices = np.arange(len(data))
np.random.shuffle(indices)

split = int(0.7 * len(data))
train_idx, test_idx = indices[:split], indices[split:]

train, test = data.iloc[train_idx], data.iloc[test_idx]
```

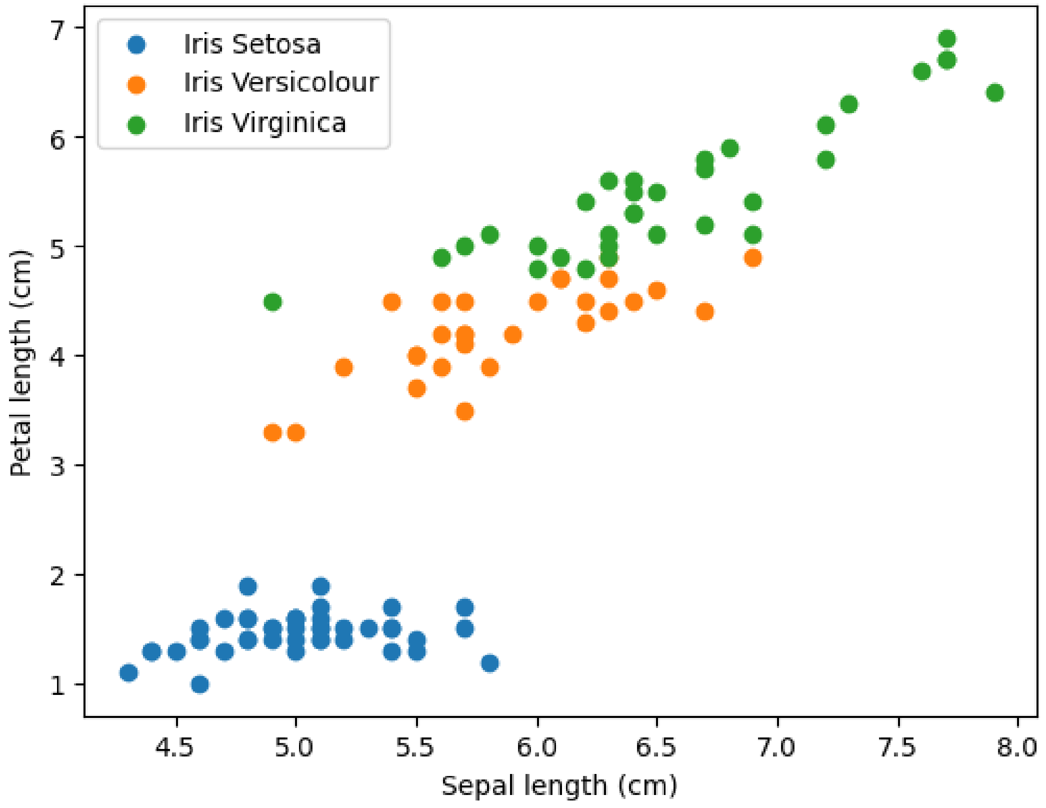
1.2 Visualization

I then plotted the different classes across two feature dimensions, the sepal length and petal length.

[5]:
```
class0 = train[train["target"] == 0]
class1 = train[train["target"] == 1]
class2 = train[train["target"] == 2]
x_class0 = class0["sepal length (cm)"]
y_class0 = class0["petal length (cm)"]
x_class1 = class1["sepal length (cm)"]
y_class1 = class1["petal length (cm)"]
x_class2 = class2["sepal length (cm)"]
y_class2 = class2["petal length (cm)"]
```

[57]:
```
import matplotlib.pyplot as plt

plt.scatter(x_class0, y_class0, label="Iris Setosa")
plt.scatter(x_class1, y_class1, label="Iris Versicolour")
plt.scatter(x_class2, y_class2, label="Iris Virginica")
plt.xlabel("Sepal length (cm)")
plt.ylabel("Petal length (cm)")
plt.legend()
plt.show()
```



1.2.1 Pattern discussion

As seen, there is some visual clustering of the three different classes from eachother. We can see the best separation between the Iris Setosa and the other two flower types. This is seen by a lower mean sepal and petal length in Iris Setosa.

Interestingly, across all flower types, there is larger variance in the sepal length than the petal length (the clusters are longer horizontally).

Lastly, there seems to be some positive covariance in the sepal length and petal length, but only for the Iris Versicolour and Iris Virginica flower types. We see this through the diagonally slanted data distribution. Interestingly, the Iris Setosa shows no covariance with these dimensions.

1.3 Recovering and printing prior probabilities

I calculated the proportion of observations for each class over all the observations in the training set. Interestingly it deviates significantly from the original 33% in the presplit dataset with both the training and testing data.

[35]:
```
class0_size = class0.shape[0]
class1_size = class1.shape[0]
class2_size = class2.shape[0]
total = class0_size + class1_size + class2_size
phis = [class0_size/total, class1_size/total, class2_size/total]
print(f"The probability of class 0: {phis[0]}")
print(f"The probability of class 1: {phis[1]}")
print(f"The probability of class 2: {phis[2]}")
phis = np.array(phis)
```

```
The probability of class 0: 0.4095238095238095
The probability of class 1: 0.2761904761904762
The probability of class 2: 0.3142857142857143
```

1.4 Computing multivariate Gaussian means and covariance matrices

I computed the empirical mean and covariance matrix for each class k, assuming that each class distribution was one multivariate Gaussian.

[63]:
```
import numpy as np
# Code adapted from HW3 code companion
X = train[["sepal length (cm)", "sepal width (cm)", "petal length␣
 ↪(cm)",        "petal width (cm)"]]
d = 4 # number of features in our toy dataset
K = 3 # number of clases
n = X.shape[0] # size of the dataset

# these are the shapes of the parameters
mus = np.zeros([K,d])
```

```
Sigmas = np.zeros([K,d,d])

# we now compute the parameters
for k in range(K):
    X_k = X[train["target"] == k]
    mus[k] = np.mean(X_k, axis=0)
    Sigmas[k] = np.cov(X_k.T)

# print out the means
print("The columns are sepal length, sepal width, petal length, petal width")
print(f"I. Setosa means: {mus[0]}")
print(f"I. Versicolour means: {mus[1]}")
print(f"I. Virginica means: {mus[2]}")
print("I. Setosa covariance matrix:")
print(Sigmas[0])
print("I. Versicolour covariance matrix:")
print(Sigmas[1])
print("I. Virginica covariance matrix:")
print(Sigmas[2])
```

```
The columns are sepal length, sepal width, petal length, petal width
I. Setosa means: [5.00697674 3.45813953 1.46511628 0.25348837]
I. Versicolour means: [5.86206897 2.74482759 4.24137931 1.32413793]
I. Virginica means: [6.58484848 2.95757576 5.51212121 2.        ]
I. Setosa covariance matrix:
[[0.13114064 0.10696567 0.01548726 0.01199889]
 [0.10696567 0.15392027 0.01064784 0.00919712]
 [0.01548726 0.01064784 0.03184939 0.0069103 ]
 [0.01199889 0.00919712 0.0069103  0.01254707]]
I. Versicolour covariance matrix:
[[0.23100985 0.06390394 0.15948276 0.05059113]
 [0.06390394 0.08827586 0.06736453 0.02923645]
 [0.15948276 0.06736453 0.18536946 0.06289409]
 [0.05059113 0.02923645 0.06289409 0.03189655]]
I. Virginica covariance matrix:
[[0.46695076 0.12558712 0.38862689 0.055625  ]
 [0.12558712 0.13189394 0.0933428  0.050625  ]
 [0.38862689 0.0933428  0.40297348 0.0565625 ]
 [0.055625   0.050625   0.0565625  0.075     ]]
```

1.5 Implement and run GDA to classify flowers

1.5.1 Prediction

I used the parameters from the previous section to predict the class of the test set that was reserved in the first section.

```
[37]: test.head()
      X = test.drop(["target"], axis=1)
```

```
[39]: # Return P(x|y)P(y) for given x
      # Code adapted from HW3 code companion
      def gda_predictions(x, mus, Sigmas, phis):
          """This returns class assignments and p(y|x) under the GDA model.

          We compute argmax_y p(y|x) as argmax_y p(x|y)p(y)
          """
          # adjust shapes
          n, d = x.shape
          x = np.reshape(x, (1, n, d, 1))
          mus = np.reshape(mus, (K, 1, d, 1))
          Sigmas = np.reshape(Sigmas, (K, 1, d, d))

          # compute probabilities
          py = np.tile(phis.reshape((K,1)), (1,n)).reshape([K,n,1,1])
          pxy = (
              np.sqrt(np.abs((2*np.pi)**d*np.linalg.det(Sigmas))).reshape((K,1,1,1))
              * -.5*np.exp(
                  np.matmul(np.matmul((x-mus).transpose([0,1,3,2]), np.linalg.
       ↪inv(Sigmas)), x-mus)
              )
          )
          pyx = pxy * py
          return pyx.argmax(axis=0).flatten(), pyx.reshape([K,n])

      idx, pyx = gda_predictions(X, mus, Sigmas, phis)
      print(idx)
```

```
[2 2 2 2 0 2 2 1 1 1 0 0 1 2 1 2 2 2 1 2 1 2 2 1 2 1 2 0 1 2 1 2 0 1 2 0 1
 1 1 2 1 1 1 1 0]

/tmp/ipykernel_2540/3115108218.py:18: RuntimeWarning: overflow encountered in
exp
  * -.5*np.exp(
```

1.5.2 Evaluation

I used the F1 score to evaluate this prediction of the test set. The F1 score is a good choice for a classification task, because it appropriately considers both the precision and recall. The precision is a metric of how correct the positive predictions are for a class k (indicate how many negatives may have been inappropriately predicted positive. The recall is a metric of how complete the positive predicitons are for a class k (indicate how many positive may have been left out).

To combine the scores across multiple classes, I used the 'macro' average that considers the three flower classes equally. I thought this was appropriate, because the original dataset has an equal split of the three flower types. In addition, I have no a priori motivation to focus my model on distinguishing a specific flower type.

```
[43]: from sklearn.metrics import f1_score

      y_true = test["target"]
      print(f1_score(y_true, idx, average ='macro'))
```

```
0.9648148148148148
```

As seen, the model gives an F1 score of 0.9648 on the testing set which indicates that it is very good at predicting the flower type.

1.5.3 K-means clustering

Intuitively, I thought that K-means clustering would create different clusters than the GDA algorithm. Here, I perform a K nearest neighhors algorithm with n_neighbors = 2.

```
[54]: from sklearn.neighbors import KNeighborsClassifier
      neigh = KNeighborsClassifier(n_neighbors=2)
      neigh.fit(train.drop(["target"],axis=1), train["target"])
```

```
[54]: KNeighborsClassifier(n_neighbors=2)
```

```
[55]: KNN_preds = neigh.predict(X)
```

```
[56]: print(f1_score(y_true, KNN_preds, average ='macro'))
```

```
0.9648148148148148
```

As seen, the F1 score is exactly the same! This made me rethink my initial intuition and realize that the K-means clustering (is equivalent to the GDA algorithm performed here. This is because the GDA algorithm assumed that each flower type was represented by a single multivariate Gaussian distribution, not by their own mixture of Gaussians. Thus, for each flower type, the model complexity can only capture the dimension-weighted distance from the Gaussian mean. This is similar to the K-means algorithm that relies on distances to determine clustering.

However, an important note is that this only applies to a K-means clustering with three centroid (K=3) to match the number of Gaussian distributions in the GDA algorithm. A K-means clustering with two centroid (K=2) will not match the GDA algorithm. This is because two centroids can not match the model complexity of three Gaussians.

For K=2 K-means clustering, a common result would be that the Iris Virginica and Iris Versicolour become clustered together without distinction.

```
[ ]:
```

①

a) The Naive Bayes (NB) assumption is that we can independently consider $x_1, x_2 \ldots$ features as being yes/no. given class k

In our example, this means we are assuming that being/not being a biker is independent to being/not being a skier given being a masters' student or a PHD student.

We say that

$$p(x|y) = p(x_1|y) \cdot p(x_2|y)$$

Where x_1 = binary indicator of biker

x_2 = binary indicator of skier

b) $P(\overset{y=}{\text{Masters}} \mid x_1 = 0, x_2 = 0)$

$$= P(y=\text{Masters}) \cdot P(x \mid y=\text{Masters})$$

$$= \frac{20}{50} \cdot P(x_1=0 \mid y=\text{Masters}) \cdot P(x_2=0 \mid y=\text{Masters})$$

$$= \frac{20}{50} \cdot \frac{20-5}{20} \cdot \frac{20-5}{20}$$

$$= 0.225 = \boxed{22.5\%}$$

c) No, we should <u>not</u> consider $P(x_1 | y=PhD)$ and $P(x_2 | y=PhD)$ as independent because we know that they are dependent.

We know that all PhD students who ski will bike so

$$P(x_1=1 \mid x_2=1, y=PhD) = 1$$

but

$$P(x_1=1 \mid y=PhD) = \frac{20}{30} \neq 1$$

So we show that for PhD students, being a biker is dependent w/ being a skier.

However our answer to b should not change. The key NB assumption in part b was

$$P(x \mid y=Masters) = P(x_1=0 \mid y=Masters) \cdot P(x_2=0 \mid y=Masters)$$

This conditional independence still holds true, since we are considering Masters students.

②

a)

$$\frac{1}{n}\sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)})$$

$$= \frac{1}{n}\sum_{i=1}^{n} \log\left[P_\theta(x^{(i)}|y^{(i)}) \cdot P_\theta(y^{(i)})\right]$$

$$= \underbrace{\frac{1}{n}\sum_{i=1}^{n} \log P_\theta(x^{(i)}|y^{(i)})}_{\substack{\vdots \\ \text{see problem b} \\ \text{(will not involve } \phi)}} + \underbrace{\frac{1}{n}\sum_{i=1}^{n} \log P_\theta(y^{(i)})}_{= \frac{1}{n}\sum_{i=1}^{n} \log P_\theta(y^{(i)}; \phi)}$$

$$\Downarrow$$

$$\max_\phi \frac{1}{n}\sum_{i=1}^{n} \log P_\theta(y^{(i)}; \phi)$$

$$\max_\phi \frac{1}{n}\sum_{k=1}^{K}\sum_{i=1}^{n_k} \log P_\theta(y=k; \phi_k)$$

$$\max_\phi \sum_{k=1}^{K}\sum_{i=1}^{n_k} \frac{1}{n} \log \phi_k$$

Each ϕ_k only in one class

$$\max_\phi \sum_{k=1}^{K} \frac{n_k}{n} \log \phi_k$$

independently optimize each ϕ_k $\rightarrow$

$$\frac{\partial \frac{n_k}{n} \log \phi_k}{\partial \phi_k} = \frac{n_k}{n} \cdot \frac{1}{\phi_k^*} = 0$$

$$\boxed{\frac{n_k}{n} = \phi_k^*}$$

$$\frac{1}{n}\sum_{i=1}^{n} \log P_\theta(x^{(i)}, y^{(i)})$$

$$= \frac{1}{n}\sum_{i=1}^{n} \log\left[P_\theta(x^{(i)}|y^{(i)}) \cdot P_\theta(y^{(i)})\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n} \log P_\theta(x^{(i)}|y^{(i)}) + \underbrace{\frac{1}{n}\sum_{i=1}^{n} \log P_\theta(y^{(i)})}_{\text{see problem a (will not involve } \psi)}$$

group by words

$$= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d} \log P_\theta(x_j^{(i)}|y^{(i)})$$

group by labels

$$= \frac{1}{n}\sum_{k=1}^{K}\sum_{j=1}^{d}\sum_{i:y^{(i)}=k} \log P(x_j^{(i)}|y^{(i)}; \psi_{jk})$$

$$= \sum_{k=1}^{K}\sum_{j=1}^{d}\sum_{i:y^{(i)}=k} \frac{1}{n_k} \log P(x_j^{(i)}|y^{(i)}=k; \psi_{jk})$$

NB

$$= \sum_{k=1}^{K}\sum_{j=1}^{d}\sum_{i:y^{(i)}=k}\sum_{j:x_j^{(i)}=\ell} \frac{1}{n_k} \log P(x_j^{(i)}=\ell|y^{(i)}=k; \psi_{jk\ell})$$

each ψ_{jk} only appears in

$$\sum_{j:x_j^{(i)}=\ell}\sum_{i:y^{(i)}=k} \frac{1}{n_k}\log P(x_j^{(i)}|y^{(i)}; \psi_{jk\ell})$$

$$= n_{jk\ell} \cdot \frac{1}{n_k} \log(\psi_{jk\ell})$$

$$\frac{\partial\, n_{jk\ell}\frac{1}{n_{k\ell}}\log(\psi_{jk\ell})}{\partial \psi_{jk\ell}} = \frac{n_{jk\ell}}{n_{k\ell}}\frac{1}{\psi_{jk\ell}^*} = 0$$

$$\boxed{\psi_{jk\ell}^* = \frac{n_{jk\ell}}{n_{k\ell}}}$$

③

$$d_e^{(w)}(x_i, x_{i'}) = \frac{\sum_{i=1}^{p} w_\ell (x_{i\ell} - x_{i'\ell})^2}{\boxed{\sum_{\ell=1}^{p} w_\ell}}$$

treat this as just a number

$$= \sum_{i=1}^{p} \frac{1}{\sum_{\ell=1}^{p} w_\ell} \cdot w_\ell (x_{i\ell} - x_{i'\ell})^2$$

$$= \sum_{i=1}^{p} \left[\sqrt{\frac{w_\ell}{\sum_{\ell=1}^{p} w_\ell}} \left(x_{i\ell} - x_{i'\ell} \right) \right]^2$$

$$= \sum_{i=1}^{p} \left[x_{i\ell} \left(\frac{w_\ell}{\sum_{\ell=1}^{p} w_\ell} \right)^{1/2} - x_{i'\ell} \left(\frac{w_\ell}{\sum_{i=1}^{p} w_\ell} \right)^{1/2} \right]^2$$

$$= \sum_{i=1}^{p} (z_{i\ell} - z_{i'\ell})^2$$

$$\hookrightarrow = \boxed{d_e(z_i, z_{i'})}$$