M. Yusa Erguven - 19260

Machine Learning HW1

Question 1

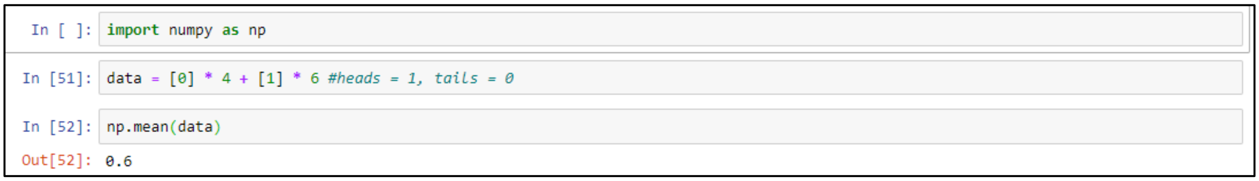
1-a:

N = 10

6 of the coin tosses = head

4 of the coin tosses = tails

So our observed data is as follows:



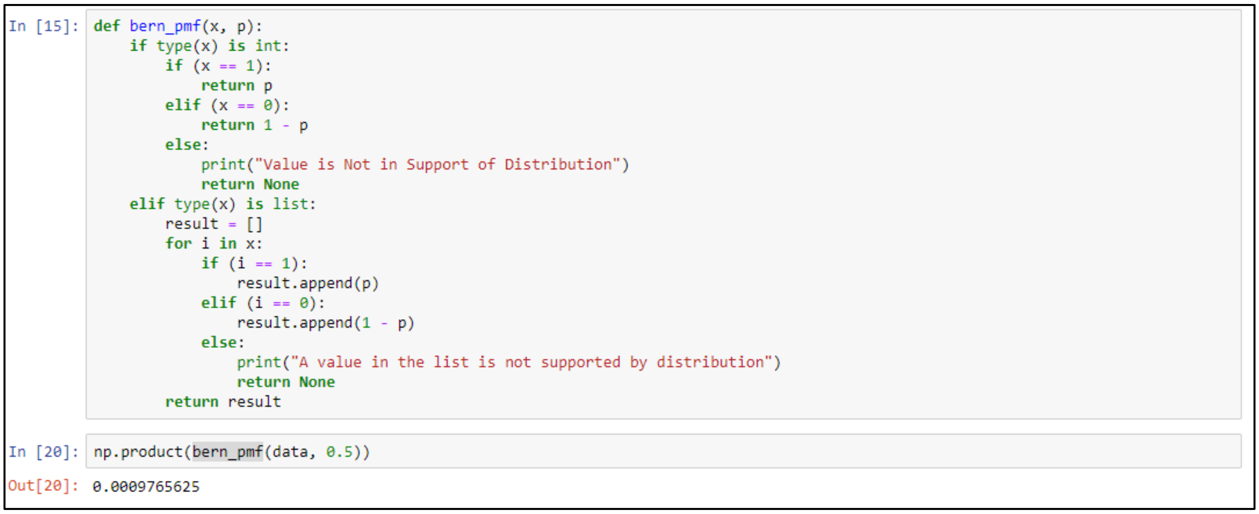
```
In [ ]:   import numpy as np

In [51]:  data = [0] * 4 + [1] * 6 #heads = 1, tails = 0

In [52]:  np.mean(data)
Out[52]:  0.6
```

After creating the data, we need to decide on the distribution how our data is generated. Since it is a coin toss with 2 outcomes, we can use Bernoulli Distribution. It takes one parameter let's say *p* and the parameter is the probability of getting 1, we can say in our case probability of head in the coin toss. Then distribution returns a value of 1 with the given probability and 0 with the probability (1 – *p*). We can consider our data is the output of Bernoulli distribution and use the Probability Mass Function of it.

I wrote a small PMF function which takes either integer or a list and a parameter *p*(probability of heads), then returns the probability of it to happen.



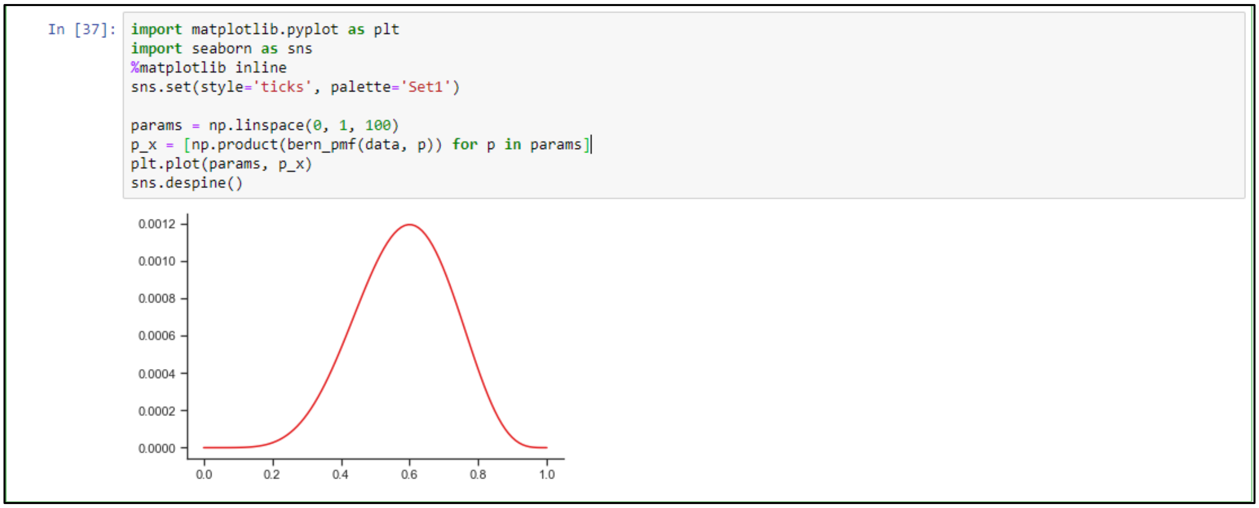
```
In [15]:  def bern_pmf(x, p):
              if type(x) is int:
                  if (x == 1):
                      return p
                  elif (x == 0):
                      return 1 - p
                  else:
                      print("Value is Not in Support of Distribution")
                      return None
              elif type(x) is list:
                  result = []
                  for i in x:
                      if (i == 1):
                          result.append(p)
                      elif (i == 0):
                          result.append(1 - p)
                      else:
                          print("A value in the list is not supported by distribution")
                          return None
                  return result

In [20]:  np.product(bern_pmf(data, 0.5))
Out[20]:  0.0009765625
```

Since every toss of coin is independent, probability of our data to happen is the product of every single data's probability to be seen as individual.

$$\boxed{p(x_1, \ldots, x_n|\beta) = p(x_1|\beta) * \ldots * p(x_n|\beta)}$$

So that we get the product of our data with the numpy's product function. Now we just assumed the probability of our coin toss is 0.5 and calculated the likelihood of it to happen. What if *p* is 0.01 or 0.82. Let's try all the interval between 0 and 1 with increasing 0.01 at each step.



In [37]:

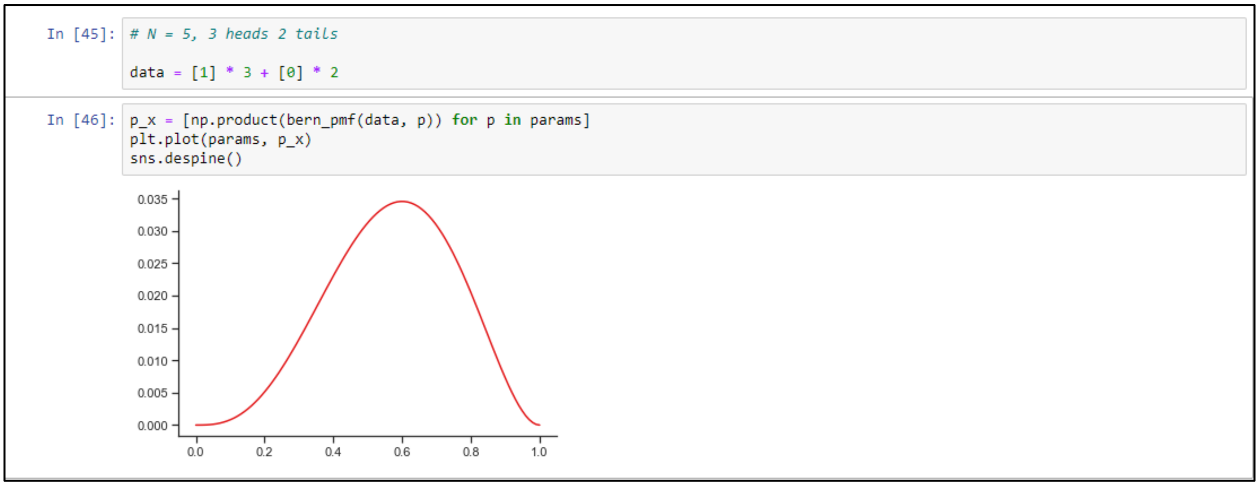
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style='ticks', palette='Set1')

params = np.linspace(0, 1, 100)
p_x = [np.product(bern_pmf(data, p)) for p in params]
plt.plot(params, p_x)
sns.despine()
```

Thus we created the graph which has every *p* value in {0, 0.01, 0.02 … 1.0} in the x axis and the likelihood of it in the y axis.

1-b:

For the other graphs with the other data, I just recreated data and with changing effected parts, redrew the graphs as follows.



In [45]:

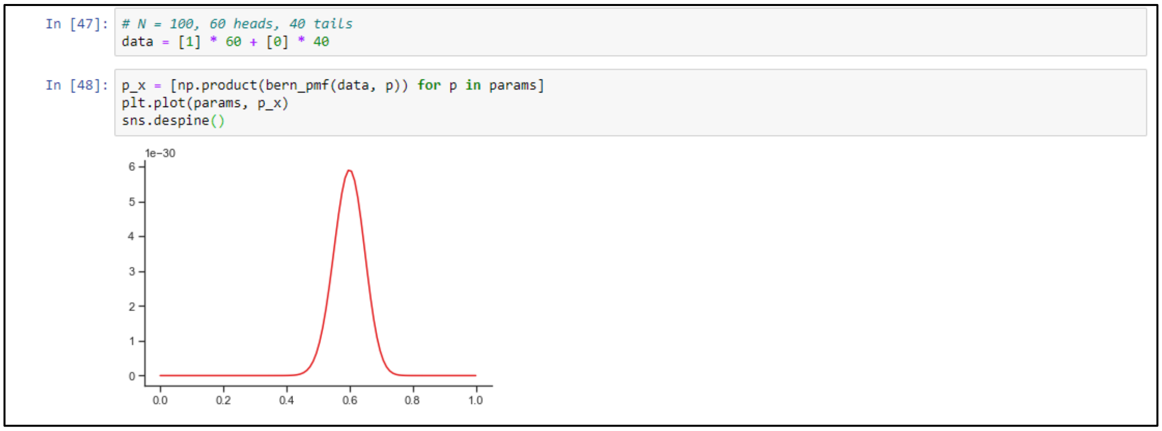
```
# N = 5, 3 heads 2 tails

data = [1] * 3 + [0] * 2
```

In [46]:

```
p_x = [np.product(bern_pmf(data, p)) for p in params]
plt.plot(params, p_x)
sns.despine()
```

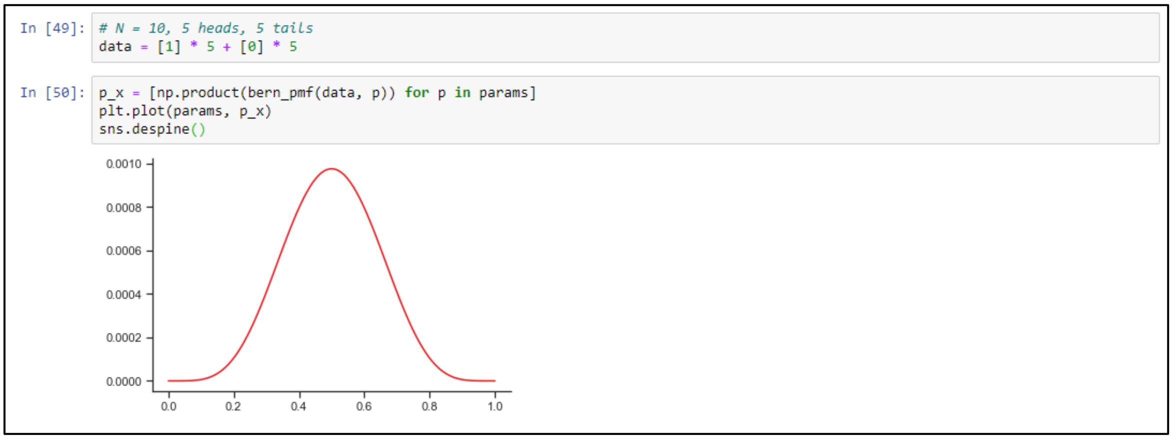
Figure 1: N = 5, heads = 3, tails = 2



```
In [47]: # N = 100, 60 heads, 40 tails
         data = [1] * 60 + [0] * 40

In [48]: p_x = [np.product(bern_pmf(data, p)) for p in params]
         plt.plot(params, p_x)
         sns.despine()
```

Figure 2: N = 100, heads = 60, tails = 40



```
In [49]: # N = 10, 5 heads, 5 tails
         data = [1] * 5 + [0] * 5

In [50]: p_x = [np.product(bern_pmf(data, p)) for p in params]
         plt.plot(params, p_x)
         sns.despine()
```

Figure 3: N = 10, heads = 5, tails = 5

1-c:

As far as I observed, when the dataset got bigger, the standard deviation got smaller. Considering the following three sample data set we observed:

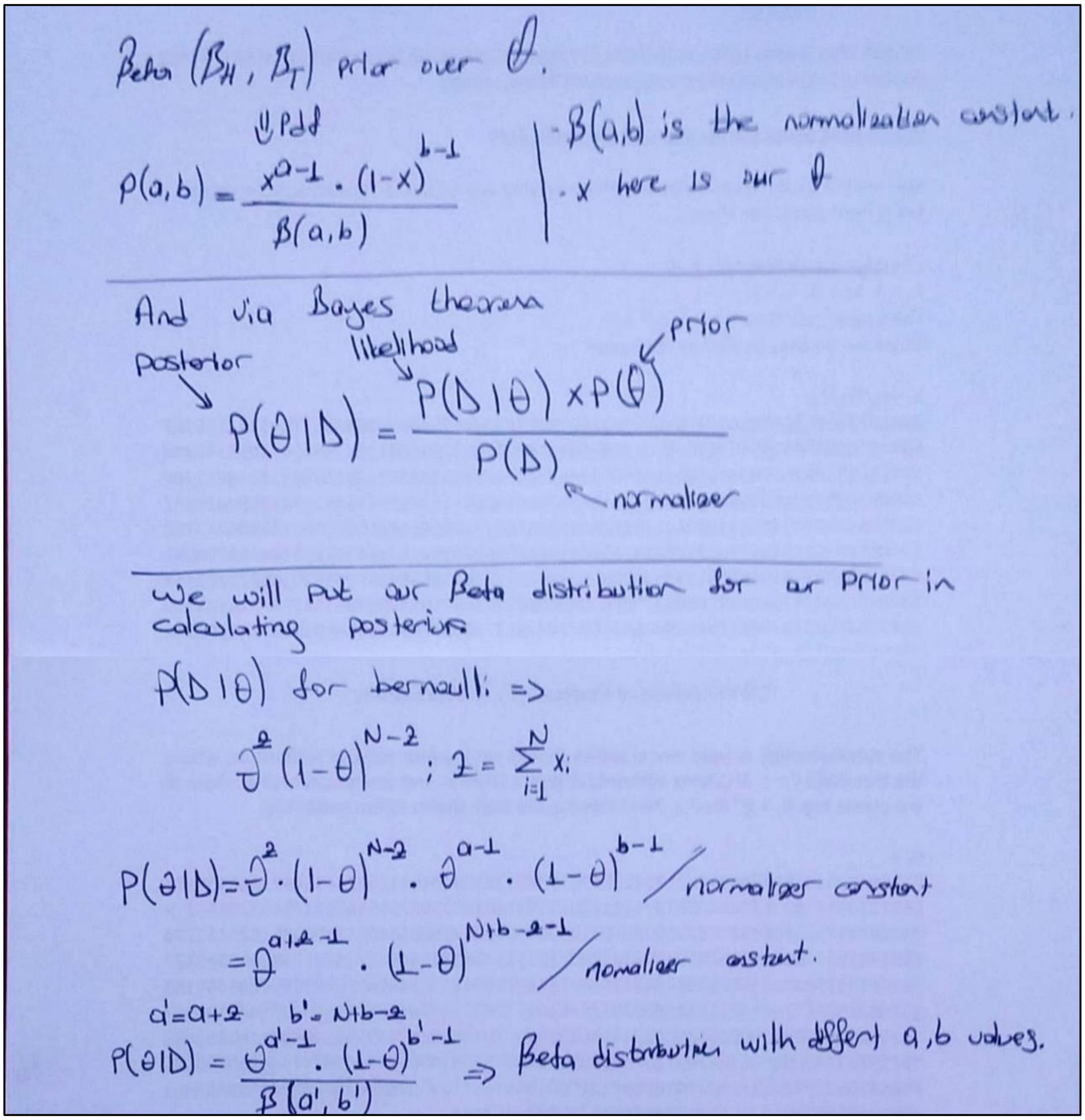
- N = 10, Heads = 6, Tails = 4
- N = 5, Heads = 3, Tails = 2
- N = 100, Heads = 60, Tails = 40

The maximum likelihood estimate (peak of the graph) of these datasets are the same which is 0.6 also it is the mean of all three datasets. When the dataset got bigger the standard deviation decreased. So that our graph looks thinner when the N (size of dataset) increases.

At the last dataset we again have 10 entry with 5 heads and 5 tails. At that point our MLE is moved to 0.5 which is the mean of our dataset. Since MLE is calculated as follows, which is the result after we took derivative of our function and assign it to 0 to find the peak point of graph.

$$\boxed{\widehat{\theta}_{MLE} = \frac{\alpha_H}{\alpha_H + \alpha_T}}$$

1-d:



Beta (β_H, β_T) prior over θ.

$$p(a,b) = \frac{\overset{\Downarrow \text{Pdf}}{x^{a-1} \cdot (1-x)^{b-1}}}{\beta(a,b)}$$

- $\beta(a,b)$ is the normalization constant.
- x here is our θ

And via Bayes theorem

$$\underset{\text{posterior}}{P(\theta \mid D)} = \frac{\underset{\text{likelihood}}{P(D \mid \theta)} \times \underset{\text{prior}}{P(\theta)}}{\underset{\text{normalizer}}{P(D)}}$$

We will put our Beta distribution for our Prior in calculating posteriors

$P(D \mid \theta)$ for bernoulli $\Rightarrow$

$$\theta^{z}(1-\theta)^{N-z} \; ; \; z = \sum_{i=1}^{N} x_i$$

$$P(\theta \mid D) = \theta^{z} \cdot (1-\theta)^{N-z} \cdot \theta^{a-1} \cdot (1-\theta)^{b-1} \,/\, \text{normalizer constant}$$

$$= \theta^{a+z-1} \cdot (1-\theta)^{N+b-z-1} \,/\, \text{nomalizer constant.}$$

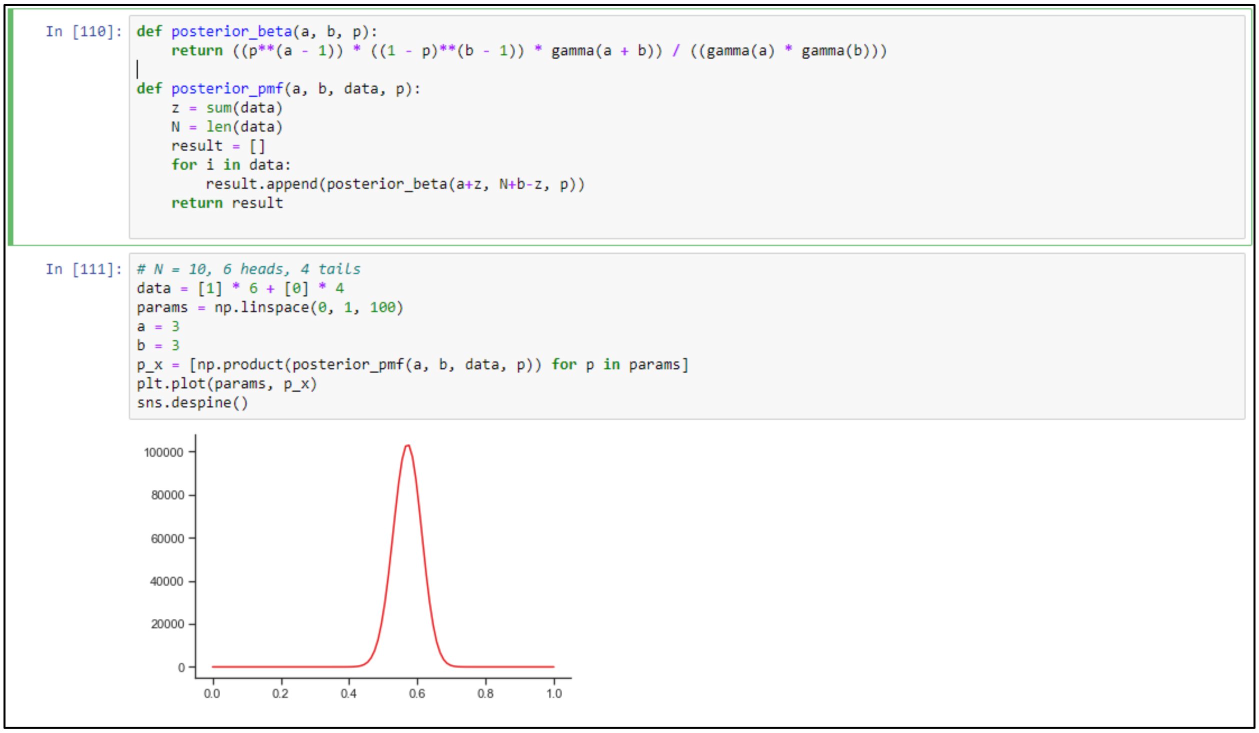
$a' = a+z \qquad b' = N+b-z$

$$P(\theta \mid D) = \frac{\theta^{a'-1} \cdot (1-\theta)^{b'-1}}{\beta(a', b')} \Rightarrow \text{Beta distribution with different } a,b \text{ values.}$$

Since Posterior itself is another Beta distribution it behaves so. It takes two parameters. If the parameters are the same, the distribution is normal distribution. When the values of them increase, the deviation decreases. If a is small while b is big, the mean of the distribution approaches to 0 and graph becomes more positively skewed. If b is small and a is big, mean gets closer to 1 and graph becomes more negatively skewed.

1-e:

For this part, I wrote the posterior distribution function and created the graph that can be seen in the following image.



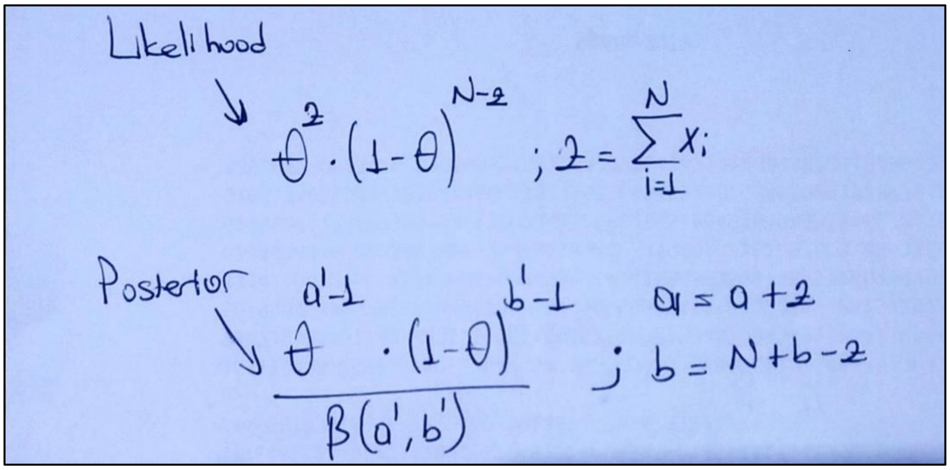
```
In [110]: def posterior_beta(a, b, p):
              return ((p**(a - 1)) * ((1 - p)**(b - 1)) * gamma(a + b)) / ((gamma(a) * gamma(b)))

          def posterior_pmf(a, b, data, p):
              z = sum(data)
              N = len(data)
              result = []
              for i in data:
                  result.append(posterior_beta(a+z, N+b-z, p))
              return result
```

```
In [111]: # N = 10, 6 heads, 4 tails
          data = [1] * 6 + [0] * 4
          params = np.linspace(0, 1, 100)
          a = 3
          b = 3
          p_x = [np.product(posterior_pmf(a, b, data, p)) for p in params]
          plt.plot(params, p_x)
          sns.despine()
```

As can be seen in the graph and compared to the first group while calculating MLE with the same data without prior, it is thinner. Which means deviation is less. It is because Bayesian estimation helps us when the data is sparse with falling back to prior and avoid issues caused by MLE. In our case, our data is very small, using the prior with given a = 3, b = 3 parameters moved the peak point of the graph to a little left, closer to 0.5.

1-f:



Likelihood

$$\theta^{z} \cdot (1-\theta)^{N-z} \; ; \; z = \sum_{i=1}^{N} x_i$$

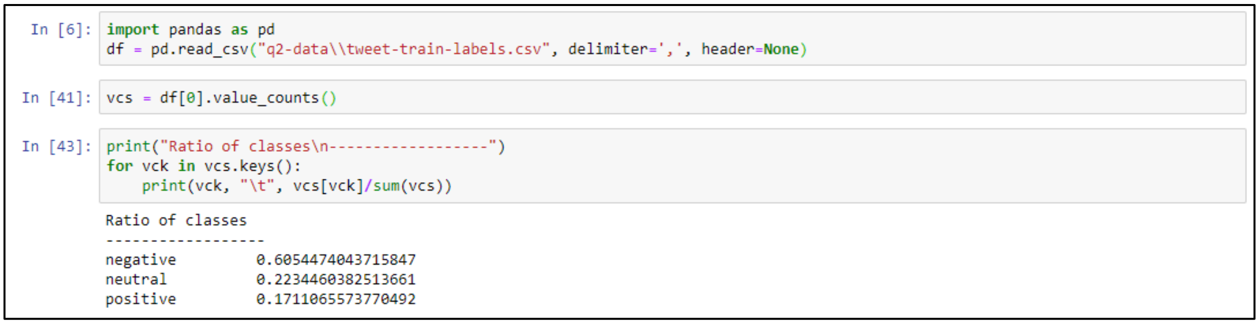
Posterior

$$\frac{\theta^{a'-1} \cdot (1-\theta)^{b'-1}}{\beta(a', b')} \; ; \; a' = a+z \quad b = N+b-z$$

Since both likelihood and posterior has N as exponent with the base in the range [0, 1], when N goes infinity, MLE and MAP estimates converges to 0. Thus, they converge if N goes infinity. When the data is abundant, the likelihood dominates the prior and the prior’s effect on the posterior distribution will be neglectable. So that MLE and MAP converge when N increase.

Question 2

2.1:



```
In [6]:  import pandas as pd
         df = pd.read_csv("q2-data\\tweet-train-labels.csv", delimiter=',', header=None)

In [41]: vcs = df[0].value_counts()

In [43]: print("Ratio of classes\n-----------------")
         for vck in vcs.keys():
             print(vck, "\t", vcs[vck]/sum(vcs))

         Ratio of classes
         -----------------
         negative        0.6054474043715847
         neutral         0.2234460382513661
         positive        0.1711065573770492
```

The dataset is not balanced since it is skewed towards negative class. The percentage of tweets is as follows:

```
Ratio of classes
------------------
negative         0.6054474043715847
neutral          0.2234460382513661
positive         0.1711065573770492
```

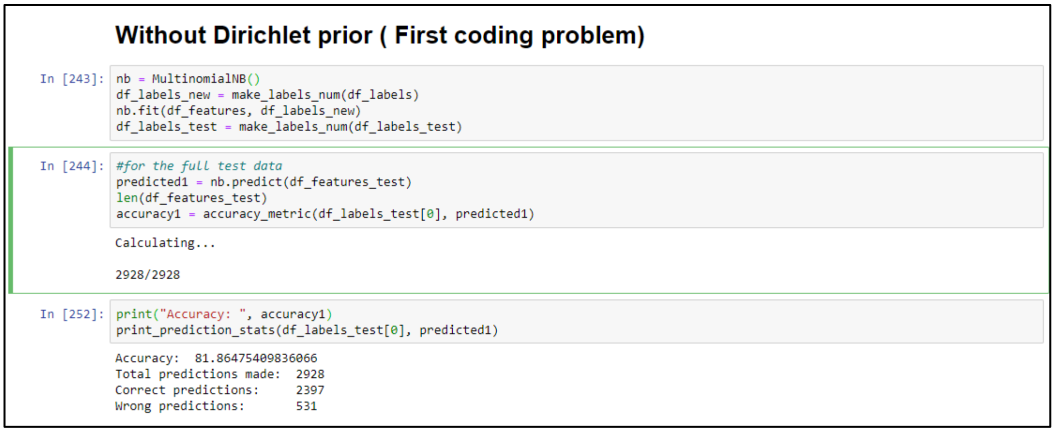
2.2:

Our Y can have 3 values: negative, neutral, positive and features (X_i) can take 2 values: 0 or 1. So that without Naïve Bayes we need to estimate 2^{5722} x 3 parameters. With the help of Naïve Bayes, 3 x 5722 = 17166 parameters.

2.3 – 2.4:

Based on the description provided in the homework, I worked with Python and wrote a Multinomial Naïve Bayes class. The code is in the file named "**HW1 - Question 2.ipynb**". With the data provided as train, I trained my model and tested with the provided test data. The code I wrote assumes the test, train data files are in the **same directory** with the codes. In the same code file you can find both Multinomial Naïve Bayesian function and the one with the Dirichlet prior. I gave the alpha to the class as parameter which also has a field named alpha at default 0. Used the same model with creating the object with alpha = 1.

My accuracy and statistics are given in the image below:



Without Dirichlet prior (First coding problem)

```
In [243]: nb = MultinomialNB()
          df_labels_new = make_labels_num(df_labels)
          nb.fit(df_features, df_labels_new)
          df_labels_test = make_labels_num(df_labels_test)

In [244]: #for the full test data
          predicted1 = nb.predict(df_features_test)
          len(df_features_test)
          accuracy1 = accuracy_metric(df_labels_test[0], predicted1)

          Calculating...

          2928/2928

In [252]: print("Accuracy: ", accuracy1)
          print_prediction_stats(df_labels_test[0], predicted1)

          Accuracy:  81.86475409836066
          Total predictions made:  2928
          Correct predictions:     2397
          Wrong predictions:       531
```

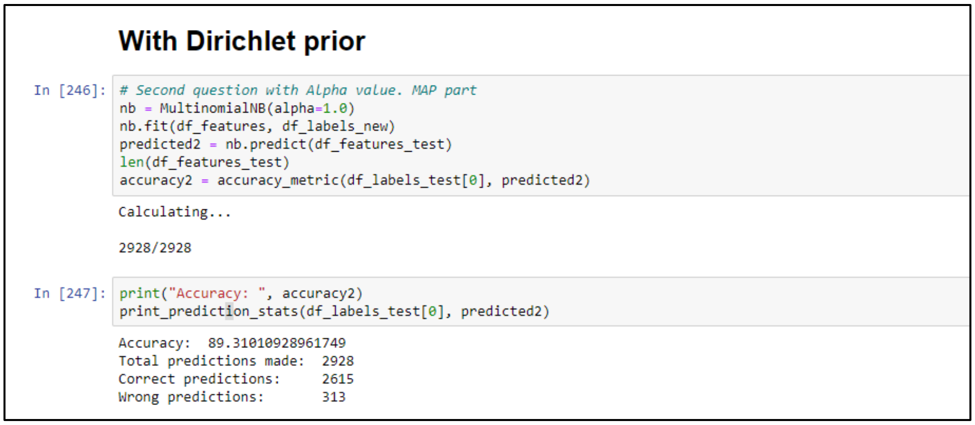
```
Accuracy:  81.86475409836066
Total predictions made: 2928
Correct predictions:    2397
Wrong predictions:      531.
```

Why is using the MLE estimate is a bad idea in this situation?

Because it is very likely to see some words that are not seen in the training data set. If I hadn't use log likelihood, I would get a lot of 0 results due to new words that doesn't exist in training set and they will be predicted as "neutral" since we favor "neutral". So that we can use alpha hyperparameter for smoothing.

$$\hat{P}(w_i|c) = \frac{\text{count}(w,c)+1}{\sum_{w' \in V} \text{count(w',c)} + |V|}$$

Which is the 4th part of the 2nd question. We use exactly same thing in that part and our prediction accuracy went up. Statistics and accuracy are given below.



With Dirichlet prior

```
In [246]: # Second question with Alpha value. MAP part
          nb = MultinomialNB(alpha=1.0)
          nb.fit(df_features, df_labels_new)
          predicted2 = nb.predict(df_features_test)
          len(df_features_test)
          accuracy2 = accuracy_metric(df_labels_test[0], predicted2)

          Calculating...

          2928/2928

In [247]: print("Accuracy: ", accuracy2)
          print_prediction_stats(df_labels_test[0], predicted2)

          Accuracy:  89.31010928961749
          Total predictions made:  2928
          Correct predictions:     2615
          Wrong predictions:       313
```

```
Accuracy:  89.31010928961749
Total predictions made: 2928
Correct predictions:    2615
Wrong predictions:      313
```

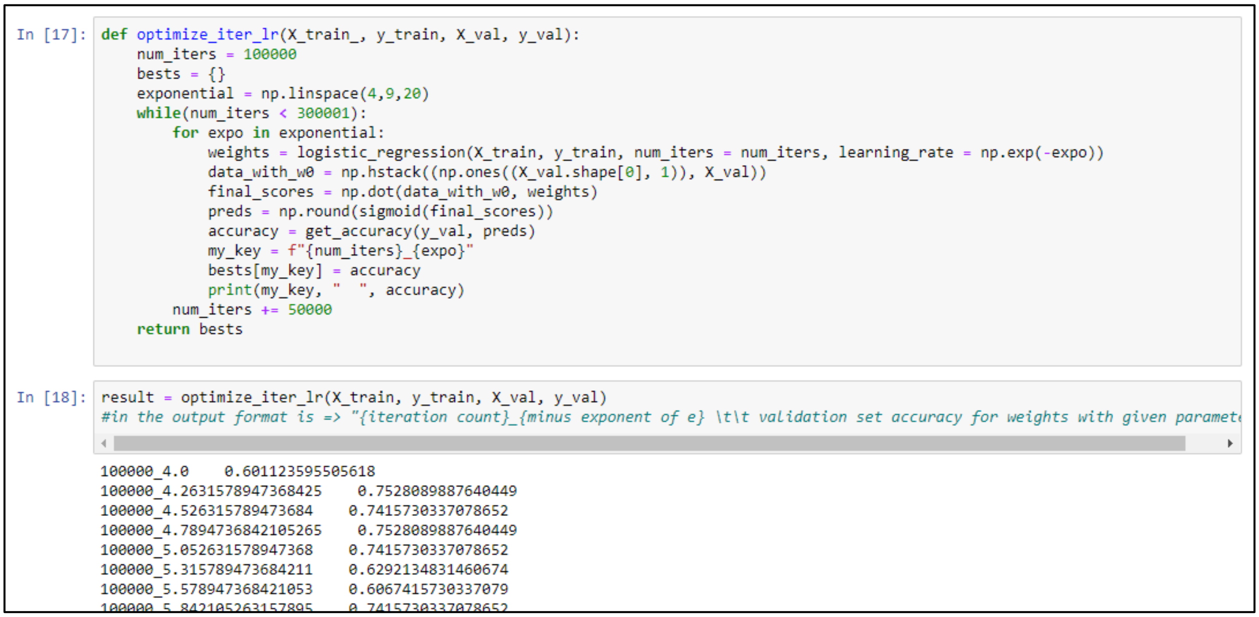
It is obvious our prediction got better with the help of smoothing. It is just an imaginary number of times the word has been seen which helps us with creating non-zero results for words that are not seen before. Alpha value doesn’t have to be 1. We can tune it with creating another dataset with splitting our training data set: validation set or development set.

Question 3

I implemented my gradient function and Logistic Regression functions, you can find it in the “HW1-Question3-solution.ipynb” file.

I splatted my data set into three random sets, 60% as train, 20% validation and 20% as test. Then tuned my ***learning rate*** and ***iteration count***. Used the validation set’s accuracy in this process.

I implemented the following optimizer function.



```
In [17]: def optimize_iter_lr(X_train_, y_train, X_val, y_val):
             num_iters = 100000
             bests = {}
             exponential = np.linspace(4,9,20)
             while(num_iters < 300001):
                 for expo in exponential:
                     weights = logistic_regression(X_train, y_train, num_iters = num_iters, learning_rate = np.exp(-expo))
                     data_with_w0 = np.hstack((np.ones((X_val.shape[0], 1)), X_val))
                     final_scores = np.dot(data_with_w0, weights)
                     preds = np.round(sigmoid(final_scores))
                     accuracy = get_accuracy(y_val, preds)
                     my_key = f"{num_iters}_{expo}"
                     bests[my_key] = accuracy
                     print(my_key, "  ", accuracy)
                 num_iters += 50000
             return bests
```

```
In [18]: result = optimize_iter_lr(X_train, y_train, X_val, y_val)
         #in the output format is => "{iteration count}_{minus exponent of e} \t\t validation set accuracy for weights with given paramete

         100000_4.0    0.601123595505618
         100000_4.2631578947368425    0.7528089887640449
         100000_4.526315789473684    0.7415730337078652
         100000_4.7894736842105265    0.7528089887640449
         100000_5.052631578947368    0.7415730337078652
         100000_5.315789473684211    0.6292134831460674
         100000_5.578947368421053    0.6067415730337079
         100000_5.842105263157895    0.7415730337078652
```

I tried optimizing with iterations starting from 50.000 and increasing with 10.000 at every iteration up to 300.000 (including 300.000) and picked the exponent of e starting from -4 to -9 with increasing 0.1 at each step. It took my 6 hours and it was still not finished.

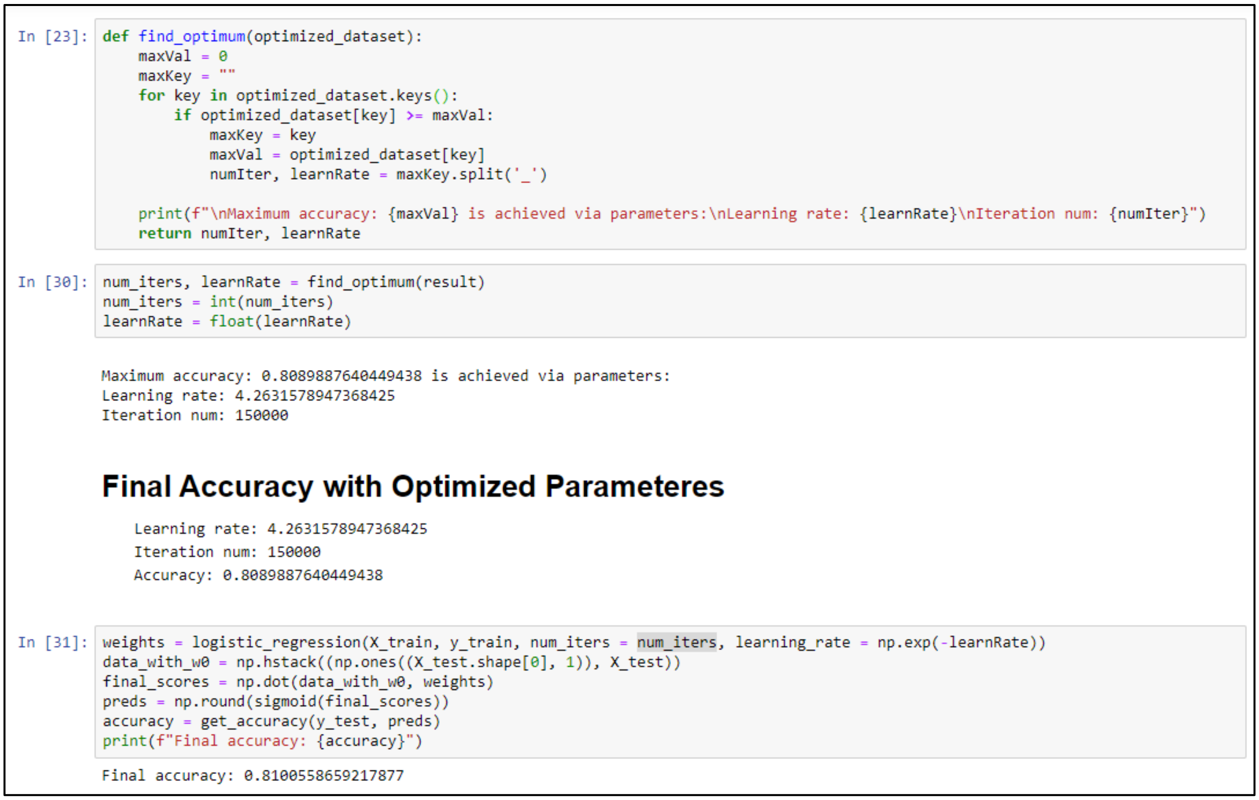
Then I narrowed my range:

- Iterations starts from 100.000 up to 300.000 with increasing 50.000 at each iteration.
- Exponent of e in learning rate starts from 4 and goes up to 9 with increasing ~ 0.25 at each iteration.

Then I ended up with the following optimum parameters with the following accuracy:

```
Maximum accuracy: 0.8089887640449438 is achieved via parameters:
Learning rate: 4.2631578947368425
Iteration num: 150000.
```

The accuracy above is the based-on validation set. My final accuracy calculated with the ***test data*** with using the given parameters above is as follows:



```
In [23]: def find_optimum(optimized_dataset):
             maxVal = 0
             maxKey = ""
             for key in optimized_dataset.keys():
                 if optimized_dataset[key] >= maxVal:
                     maxKey = key
                     maxVal = optimized_dataset[key]
                     numIter, learnRate = maxKey.split('_')

             print(f"\nMaximum accuracy: {maxVal} is achieved via parameters:\nLearning rate: {learnRate}\nIteration num: {numIter}")
             return numIter, learnRate
```

```
In [30]: num_iters, learnRate = find_optimum(result)
         num_iters = int(num_iters)
         learnRate = float(learnRate)
```

```
Maximum accuracy: 0.8089887640449438 is achieved via parameters:
Learning rate: 4.2631578947368425
Iteration num: 150000
```

Final Accuracy with Optimized Parameteres

```
Learning rate: 4.2631578947368425
Iteration num: 150000
Accuracy: 0.8089887640449438
```

```
In [31]: weights = logistic_regression(X_train, y_train, num_iters = num_iters, learning_rate = np.exp(-learnRate))
         data_with_w0 = np.hstack((np.ones((X_test.shape[0], 1)), X_test))
         final_scores = np.dot(data_with_w0, weights)
         preds = np.round(sigmoid(final_scores))
         accuracy = get_accuracy(y_test, preds)
         print(f"Final accuracy: {accuracy}")

         Final accuracy: 0.8100558659217877
```

```
Final accuracy: 0.8100558659217877
Learning rate: 4.2631578947368425
Iteration num: 150000
```