HOMEWORK 4

Dario Placencio - 907 284 6018

Instructions: Use this latex file as a template to develop your homework. Submit your homework on time as a single pdf file to Canvas. Late submissions may not be accepted. Please wrap your code and upload to a public GitHub repo, then attach the link below the instructions so that we can access it. You can choose any programming language (i.e. python, R, or MATLAB). Please check Piazza for updates about the homework.

1 Best Prediction Under 0-1 Loss (10 pts)

Suppose the world generates a single observation $x \sim \text{multinomial}(\theta)$, where the parameter vector $\theta = (\theta_1, \ldots, \theta_k)$ with $\theta_i \geq 0$ and $\sum_{i=1}^{k} \theta_i = 1$. Note $x \in \{1, \ldots, k\}$. You know θ and want to predict x. Call your prediction $\hat{x}$. What is your expected 0-1 loss:

$$\mathbb{E}[\mathbb{1}_{\hat{x} \neq x}]$$

Using the following two prediction strategies respectively? Prove your answer.

Strategy 1: $\hat{x} \in \arg\max_x \theta_x$, the outcome with the highest probability.

Give outcome as i^*, such that: $i^* = \arg\max_i \theta_i$.
Now, there are two possible scenarios:

1. $x = i^*$ which occurs with probability θ_{i^*}. In this case, the 0-1 loss is 0.
2. $x \neq i^*$ which occurs with probability $1 - \theta_{i^*}$. In this case, the 0-1 loss is 1.

Therefore, the expected 0-1 loss for Strategy 1 is:

$\mathbb{E}[\nVdash_{\hat{x} \neq x}] = 1 - \theta_{i^*}$

Strategy 2: You mimic the world by generating a prediction $\hat{x} \sim \text{multinomial}(\theta)$. (Hint: your randomness and the world's randomness are independent)

The prediction $\hat{x}$ is also drawn from the multinomial distribution with parameter vector θ.

Since the randomness and the world's randomness are independent, the probability that $\hat{x} = x$ is the sum of the probabilities that both we and the world draw the same outcome:

$\mathbb{P}(\hat{x} = x) = \sum_{i=1}^{k} \theta_i^2$

The probability that $\hat{x} \neq x$ is:

$1 - \sum_{i=1}^{k} \theta_i^2$

So, the expected 0-1 loss for Strategy 2 is:

$\mathbb{E}[\nVdash_{\hat{x} \neq x}] = 1 - \sum_{i=1}^{k} \theta_i^2$

2 Best Prediction Under Different Misclassification Losses (6 pts)

Like in the previous question, the world generates a single observation $x \sim \text{multinomial}(\theta)$. Let $c_{ij} \geq 0$ denote the loss you incur, if $x = i$ but you predict $\hat{x} = j$, for $i, j \in \{1, \dots, k\}$. $c_{ii} = 0$ for all i. This is a way to generalize different costs on false positives vs false negatives from binary classification to multi-class classification. You want to minimize your expected loss:

$$\mathbb{E}[c_{x\hat{x}}]$$

Derive your optimal prediction $\hat{x}$.

Given the loss matrix c_{ij}, the goal is to minimize the expected loss:

$\mathbb{E}[c_{x\hat{x}}]$

To find the optimal prediction $\hat{x}$, let's consider the loss incurred by predicting each possible j and then choose the j that minimizes this expected loss.

For a fixed prediction $\hat{x} = j$, the expected loss is:

$\mathbb{E}[c_{xj}] = \sum_{i=1}^{k} \theta_i c_{ij}$

This is because the probability that the true observation is i is θ_i and the loss incurred in that case is c_{ij}.

To find the prediction j that minimizes this expected loss, let's compute $\mathbb{E}[c_{xj}]$ for each j and choose the j that gives the smallest value.

The optimal prediction $\hat{x}$ is then:

$\hat{x} = \arg\min_j \sum_{i=1}^{k} \theta_i c_{ij}$

Thus, to minimize the expected loss, we should predict j that minimizes the weighted sum of the loss, where the weights are given by the probabilities θ_i.

3 Language Identification with Naive Bayes (8 pts each)

Implement a character-based Naive Bayes classifier that classifies a document as English, Spanish, or Japanese - all written with the 26 lower case characters and space.
The dataset is languageID.tgz, unpack it. This dataset consists of 60 documents in English, Spanish and Japanese. The correct class label is the first character of the filename: $y \in \{e, j, s\}$. (Note: here each file is a document in corresponding language, and it is regarded as one data.)
We will be using a character-based multinomial Naïve Bayes model. You need to view each document as a bag of characters, including space. We have made sure that there are only 27 different types of printable characters (a to z, and space) – there may be additional control characters such as new-line, please ignore those. Your vocabulary will be these 27 character types. (Note: not word types!)

1. Use files 0.txt to 9.txt in each language as the training data. Estimate the prior probabilities $\hat{p}(y = e)$, $\hat{p}(y = j)$, $\hat{p}(y = s)$ using additive smoothing with parameter $\frac{1}{2}$. Give the formula for additive smoothing with parameter $\frac{1}{2}$ in this case. Print and include in final report the prior probabilities. (Hint: Store all probabilities here and below in $\log()$ internally to avoid underflow. This also means you need to do arithmetic in log-space. But answer questions with probability, not log probability.)

2. Using the same training data, estimate the class conditional probability (multinomial parameter) for English

$$\theta_{i,e} := \hat{p}(c_i \mid y = e)$$

where c_i is the i-th character. That is, $c_1 = a, \dots, c_{26} = z, c_{27} = space$. Again use additive smoothing with parameter $\frac{1}{2}$. Give the formula for additive smoothing with parameter $\frac{1}{2}$ in this case. Print θ_e and include in final report which is a vector with 27 elements.

3. Print θ_j, θ_s and include in final report the class conditional probabilities for Japanese and Spanish.

4. Treat e10.txt as a test document x. Represent x as a bag-of-words count vector (Hint: the vocabulary has size 27). Print the bag-of-words vector x and include in final report.

5. Compute $\hat{p}(x \mid y)$ for $y = e, j, s$ under the multinomial model assumption, respectively. Use the formula

$$\hat{p}(x \mid y) = \prod_{i=1}^{d} \theta_{i,y}^{x_i}$$

where $x = (x_1, \ldots, x_d)$. Show the three values: $\hat{p}(x \mid y = e), \hat{p}(x \mid y = j), \hat{p}(x \mid y = s)$. Hint: you may notice that we omitted the multinomial coefficient. This is ok for classification because it is a constant w.r.t. y.

6. Use Bayes rule and your estimated prior and likelihood, compute the posterior $\hat{p}(y \mid x)$. Show the three values: $\hat{p}(y = e \mid x), \hat{p}(y = j \mid x), \hat{p}(y = s \mid x)$. Show the predicted class label of x.

7. Evaluate the performance of your classifier on the test set (files 10.txt to 19.txt in three languages). Present the performance using a confusion matrix. A confusion matrix summarizes the types of errors your classifier makes, as shown in the table below. The columns are the true language a document is in, and the rows are the classified outcome of that document. The cells are the number of test documents in that situation. For example, the cell with row = English and column = Spanish contains the number of test documents that are really Spanish, but misclassified as English by your classifier.

8. If you take a test document and arbitrarily shuffle the order of its characters so that the words (and spaces) are scrambled beyond human recognition. How does this shuffling affect your Naive Bayes classifier's prediction on this document? Explain the key mathematical step in the Naive Bayes model that justifies your answer.

   Shuffling the characters in a document does not affect the Naive Bayes classifier's prediction when using a character-based model. This is because:

   1. Independence Assumption: The Naive Bayes model assumes that characters are conditionally independent given the language. This means the occurrence of one character does not depend on the occurrence of another, given the language.

   2. Representation: Documents are represented as bag-of-characters, which only considers character counts, not their order. Shuffling doesn't change these counts.

   Mathematically, the likelihood term is: $p(x \mid y) = \prod_{i=1}^{d} \theta_{i,y}^{x_i}$

   Where $\theta_{i,y}$ is the probability of character i given language y and x_i is the count of character i. Since only the counts x_i matter and they remain unchanged with shuffling, the prediction remains the same.

4 Simple Feed-Forward Network (20pts)

In this exercise, you will derive, implement back-propagation for a simple neural network and compare your output with some standard library's output. Consider the following 3-layer neural network.

$$\hat{y} = f(x) = g(W_2\sigma(W_1 x))$$

Suppose $x \in \mathbb{R}^d$, $W_1 \in \mathbb{R}^{d_1 \times d}$, and $W_2 \in \mathbb{R}^{k \times d_1}$ i.e. $f : \mathbb{R}^d \to \mathbb{R}^k$, Let $\sigma(z) = [\sigma(z_1), ..., \sigma(z_n)]$ for any $z \in \mathbb{R}^n$ where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid (logistic) activation function and $g(z_i) = \frac{exp(z_i)}{\sum_{i=1}^{k} exp(z_i)}$ is the softmax function. Suppose the true pair is (x, y) where $y \in \{0,1\}^k$ with exactly one of the entries equal to 1, and you are working with the cross-entropy loss function given below,

$$L(x, y) = -\sum_{i=1}^{k} y \log(\hat{y})$$

1. Derive backpropagation updates for the above neural network. (5 pts)

   Given:

$$\hat{y} = f(x) = g(W_2\sigma(W_1x))$$

Where: $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid activation function. $g(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$ is the softmax function. Loss function $L(x, y) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$.

Forward Pass:

1. Compute the input to the hidden layer:

$$a_1 = W_1x$$

2. Compute the output of the hidden layer:

$$h = \sigma(a_1)$$

3. Compute the input to the output layer:

$$a_2 = W_2h$$

4. Compute the output of the network:

$$\hat{y} = g(a_2)$$

Backward Pass:

1. Gradient of the loss with respect to the output:

$$\frac{\partial L}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

2. Gradient of the loss with respect to a_2 (input of softmax):

$$\frac{\partial L}{\partial a_2} = \hat{y} - y$$

3. Gradient of the loss with respect to W_2:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial a_2} h^T$$

4. Gradient of the loss with respect to h:

$$\frac{\partial L}{\partial h} = W_2^T \frac{\partial L}{\partial a_2}$$

5. Gradient of the loss with respect to a_1 (input of sigmoid):

$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial h} \cdot h \cdot (1 - h)$$

6. Gradient of the loss with respect to W_1:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial a_1} x^T$$

2. Implement it in NumPy or PyTorch using basic linear algebra operations. (e.g. You are not allowed to use auto-grad, built-in optimizer, model, etc. in this step. You can use library functions for data loading, processing, etc.). Evaluate your implementation on MNIST dataset, report test errors and learning curve. (10 pts)

3. Implement the same network in PyTorch (or any other framework). You can use all the features of the framework e.g. auto-grad etc. Evaluate it on MNIST dataset, report test errors, and learning curve. (2 pts)

4. Try different weight initialization a) all weights initialized to 0, and b) initialize the weights randomly between -1 and 1. Report test error and learning curves for both. (You can use either of the implementations) (3 pts)

You should play with different hyperparameters like learning rate, batch size, etc. for your own learning. You only need to report results for any particular setting of hyperparameters. You should mention the values of those along with the results. Use $d_1 = 300$, $d_2 = 200$. For optimization use SGD (Stochastic gradient descent) without momentum, with some batch size say 32, 64, etc. MNIST can be obtained from here (https://pytorch.org/vision/stable/datasets.html)

Homework 4 - Dario Placencio

October 25, 2023

1 Homework 4 - Dario Placencio

[ ]:
```
# import the necessary packages
import numpy as np
import pandas as pd
import os
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from IPython.display import Image
import plotly.io as pio
```

1.0.1 3 - Language Identification with Naive Bayes

1. Use files 0.txt to 9.txt in each language as the training data. Estimate the prior probabilities $\hat{p}(y = e)$, $\hat{p}(y = j)$, $\hat{p}(y = s)$ using additive smoothing with parameter $\frac{1}{2}$. Give the formula for additive smoothing with parameter $\frac{1}{2}$ in this case. Print and include in final report the prior probabilities. (Hint: Store all probabilities here and below in log() internally to avoid underflow. This also means you need to do arithmetic in log-space. But answer questions with probability, not log probability.)

[ ]:
```
# Adjust the extract_path to point directly to the languageID folder
extract_path = 'languageID'
```

[ ]:
```
# Read the contents of the training files
def read_files(lang, start, end):
    """Read files for a given language between start and end indices."""
    contents = ""
    for i in range(start, end + 1):
        with open(os.path.join(extract_path, f"{lang}{i}.txt"), 'r',␣
 ↪errors='ignore') as f:
            contents += f.read()
    return contents
```

```
# Extract training data for each language
english_train = read_files('e', 0, 9)
spanish_train = read_files('s', 0, 9)
japanese_train = read_files('j', 0, 9)

len(english_train), len(spanish_train), len(japanese_train)
```

[ ]: (15339, 16500, 15496)

[ ]:
```
# Given data and smoothing parameters
N_y = 10  # Number of English training documents
N = 30   # Total number of training documents
alpha = 0.5  # Smoothing parameter
V = 3  # Number of classes (languages)

# Compute smoothed prior probability for English
p_e_smoothed = (N_y + alpha) / (N + alpha * V)
p_e_smoothed
```

[ ]: 0.3333333333333333

[ ]:
```
# Store as log probabilities
p_e_smoothed = np.log(p_e_smoothed)
```

[ ]:
```
# Compute smoothed prior probability for Spanish
p_s_smoothed = (N_y + alpha) / (N + alpha * V)
p_s_smoothed
```

[ ]: 0.3333333333333333

[ ]:
```
# Store as log probabilities
p_s_smoothed = np.log(p_s_smoothed)
```

[ ]:
```
# Compute smoothed prior probability for Japanese
p_j_smoothed = (N_y + alpha) / (N + alpha * V)
p_j_smoothed
```

[ ]: 0.3333333333333333

[ ]:
```
# Store as log probabilities
p_j_smoothed = np.log(p_j_smoothed)
```

2. Using the same training data, estimate the class conditional probability (multinomial parameter) for English $\theta_{i,e} := \hat{p}(c_i \mid y = e)$ where c_i is the i-th character. That is, $c_1 = a, \ldots, c_{26} = z, c_{27} = space$.

Again use additive smoothing with parameter $\frac{1}{2}$. Give the formula for additive smoothing with parameter $\frac{1}{2}$ in this case.

Print θ_e and include in final report which is a vector with 27 elements.

[ ]:
```
from collections import defaultdict

# Function to count the frequency of each character in a given text
def char_frequencies(text):
    freq = defaultdict(int)
    for char in text:
        # Only consider lowercase alphabets and space
        if char in 'abcdefghijklmnopqrstuvwxyz ':
            freq[char] += 1
    return freq

# Assuming english_train contains the combined content of English training files
english_freq = char_frequencies(english_train)

# Compute the total number of characters in the English training data
total_english_chars = sum(english_freq.values())

# Vocabulary
vocabulary = 'abcdefghijklmnopqrstuvwxyz '

# Smoothing parameter
alpha = 0.5

# Compute the class conditional probabilities for English
theta_e = {}
for char in vocabulary:
    theta_e[char] = (english_freq[char] + alpha) / (total_english_chars + alpha␣
 ↪* len(vocabulary))

print(theta_e)
```

```
{'a': 0.0601685114819098, 'b': 0.011134974392863043, 'c': 0.021509995043779945,
'd': 0.021972575582355856, 'e': 0.1053692383941847, 'f': 0.018932760614571286,
'g': 0.017478936064761277, 'h': 0.047216256401784236, 'i': 0.055410540227986124,
'j': 0.001420783082768875, 'k': 0.0037336857756484387, 'l':
0.028977366595076822, 'm': 0.020518751032545846, 'n': 0.057921691723112505, 'o':
0.06446390219725756, 'p': 0.01675202378985627, 'q': 0.0005617049396993227, 'r':
0.053824549810011564, 's': 0.06618205848339666, 't': 0.08012555757475633, 'u':
0.026664463902197257, 'v': 0.009284652238559392, 'w': 0.015496448042293078, 'x':
0.001156451346439782, 'y': 0.013844374690236246, 'z': 0.0006277878737815959, '
': 0.1792499586981662}
```

[ ]:
```
# Store as log probabilities
for char in vocabulary:
    theta_e[char] = np.log(theta_e[char])
```

3. Print θ_j, θ_s and include in final report the class conditional probabilities for Japanese and

Spanish.

```
[ ]: # Compute character frequencies for Japanese and Spanish
japanese_freq = char_frequencies(japanese_train)
spanish_freq = char_frequencies(spanish_train)

# Compute the total number of characters in the Japanese and Spanish training␣
↪data
total_japanese_chars = sum(japanese_freq.values())
total_spanish_chars = sum(spanish_freq.values())

# Compute the class conditional probabilities for Japanese
theta_j = {}
for char in vocabulary:
    theta_j[char] = (japanese_freq[char] + alpha) / (total_japanese_chars +␣
↪alpha * len(vocabulary))

# Compute the class conditional probabilities for Spanish
theta_s = {}
for char in vocabulary:
    theta_s[char] = (spanish_freq[char] + alpha) / (total_spanish_chars + alpha␣
↪* len(vocabulary))

print("Theta_j:", theta_j)
print("Theta_s:", theta_s)
```

```
Theta_j: {'a': 0.1317656102589189, 'b': 0.010866906600510151, 'c':
0.005485866033054963, 'd': 0.01722631818022992, 'e': 0.06020475907613823, 'f':
0.003878542227191726, 'g': 0.014011670568503443, 'h': 0.03176211607673224, 'i':
0.09703343932352633, 'j': 0.0023411020650616725, 'k': 0.05740941332681086, 'l':
0.001432614696530277, 'm': 0.03979873510604843, 'n': 0.05671057688947902, 'o':
0.09116321324993885, 'p': 0.0008735455466648031, 'q': 0.00010482546559977637,
'r': 0.04280373178657535, 's': 0.0421747789929767, 't': 0.056990111464411755,
'u': 0.07061742199238269, 'v': 0.0002445927530661449, 'w': 0.01974212935462455,
'x': 3.4941821866592126e-05, 'y': 0.01415143785596981, 'z': 0.00772214263251686,
' ': 0.12344945665466997}
Theta_s: {'a': 0.10456045141993771, 'b': 0.008232863618143134, 'c':
0.03752582405722919, 'd': 0.039745922111559924, 'e': 0.1138108599796491, 'f':
0.00860287996053159, 'g': 0.0071844839813758445, 'h': 0.0045327001942585795,
'i': 0.049859702136844375, 'j': 0.006629459467793161, 'k':
0.0002775122567913416, 'l': 0.052943171656748174, 'm': 0.02580863988159477, 'n':
0.054176559464709693, 'o': 0.07249236841293824, 'p': 0.02426690512164287, 'q':
0.007677839104560451, 'r': 0.05929511886774999, 's': 0.06577040485954797, 't':
0.03561407295488884, 'u': 0.03370232185254849, 'v': 0.00588942678301625, 'w':
9.250408559711388e-05, 'x': 0.0024976103111220747, 'y': 0.007862847275754679,
'z': 0.0026826184823163022, ' ': 0.16826493170115014}
```

```
[ ]: # Store as log probabilities
for char in vocabulary:
    theta_j[char] = np.log(theta_j[char])
    theta_s[char] = np.log(theta_s[char])
```

4. Treat e10.txt as a test document x. Represent x as a bag-of-words count vector (Hint: the vocabulary has size 27).

Print the bag-of-words vector x and include in final report.

```
[ ]: # Read the content of e10.txt
with open('languageID/e10.txt', 'r', errors='ignore') as f:
    test_content = f.read()

# Compute character frequencies for the test content
test_freq = char_frequencies(test_content)

# Create the bag-of-words vector
bow_vector = [test_freq[char] for char in vocabulary]

print("Bag-of-Words Vector:", bow_vector)
```

```
Bag-of-Words Vector: [164, 32, 53, 57, 311, 55, 51, 140, 140, 3, 6, 85, 64, 139,
182, 53, 3, 141, 186, 225, 65, 31, 47, 4, 38, 2, 498]
```

5. Compute $\hat{p}(x \mid y)$ for $y = e, j, s$ under the multinomial model assumption, respectively. Use the formula $\hat{p}(x \mid y) = \prod_{i=1}^{d} \theta_{i,y}^{x_i}$ where $x = (x_1, \ldots, x_d)$.

Show the three values: $\hat{p}(x \mid y = e), \hat{p}(x \mid y = j), \hat{p}(x \mid y = s)$.

Hint: you may notice that we omitted the multinomial coefficient. This is ok for classification because it is a constant w.r.t. y.

```
[ ]: import math

# Function to compute the log-probability log(p(x|y)) for a given language and␣
 ↪its log(theta) values
def compute_log_probability(bow_vector, log_theta_values):
    log_prob = 0
    for xi, char in zip(bow_vector, vocabulary):
        log_prob += xi * log_theta_values[char]
    return log_prob

# Compute log(p(x|y)) for each language
log_p_x_given_e = compute_log_probability(bow_vector, theta_e)
log_p_x_given_j = compute_log_probability(bow_vector, theta_j)
log_p_x_given_s = compute_log_probability(bow_vector, theta_s)

print("log(p(x|y=e)):", log_p_x_given_e)
print("log(p(x|y=j)):", log_p_x_given_j)
```

```
print("log(p(x|y=s)):", log_p_x_given_s)
```

```
log(p(x|y=e)): -7841.865447060635
log(p(x|y=j)): -8771.433079075032
log(p(x|y=s)): -8467.282044010557
```

[ ]:
```
# Convert the log probabilities back to regular probabilities
p_x_given_e = math.exp(log_p_x_given_e)
p_x_given_j = math.exp(log_p_x_given_j)
p_x_given_s = math.exp(log_p_x_given_s)

print("p(x|y=e):", p_x_given_e)
print("p(x|y=j):", p_x_given_j)
print("p(x|y=s):", p_x_given_s)
```

```
p(x|y=e): 0.0
p(x|y=j): 0.0
p(x|y=s): 0.0
```

[ ]:
```
# Print p_x_given_e, p_x_given_j, and p_x_given_s with 10 decimal places
print("p(x|y=e): {:.10f}".format(p_x_given_e))
print("p(x|y=j): {:.10f}".format(p_x_given_j))
print("p(x|y=s): {:.10f}".format(p_x_given_s))
```

```
p(x|y=e): 0.0000000000
p(x|y=j): 0.0000000000
p(x|y=s): 0.0000000000
```

6. Use Bayes rule and your estimated prior and likelihood, compute the posterior $\hat{p}(y \mid x)$.

Show the three values: $\hat{p}(y = e \mid x), \hat{p}(y = j \mid x), \hat{p}(y = s \mid x)$.

Show the predicted class label of x.

[ ]:
```
# Compute log posterior for each language using the log priors
log_posterior_e = log_p_x_given_e + p_e_smoothed
log_posterior_j = log_p_x_given_j + p_j_smoothed
log_posterior_s = log_p_x_given_s + p_s_smoothed

# Normalize log posteriors by subtracting the max value among them
max_log_posterior = max(log_posterior_e, log_posterior_j, log_posterior_s)

# Convert normalized log posteriors back to regular probabilities
posterior_e = math.exp(log_posterior_e - max_log_posterior)
posterior_j = math.exp(log_posterior_j - max_log_posterior)
posterior_s = math.exp(log_posterior_s - max_log_posterior)

# Normalize the posteriors to sum to 1
normalizing_factor = posterior_e + posterior_j + posterior_s
normalized_posterior_e = posterior_e / normalizing_factor
```

```
normalized_posterior_j = posterior_j / normalizing_factor
normalized_posterior_s = posterior_s / normalizing_factor

# Determine the predicted class label
predicted_class_label = max([('e', normalized_posterior_e), ('j',␣
↪normalized_posterior_j), ('s', normalized_posterior_s)], key=lambda x:␣
↪x[1])[0]

print("p(y=e|x):", normalized_posterior_e)
print("p(y=j|x):", normalized_posterior_j)
print("p(y=s|x):", normalized_posterior_s)
print("Predicted class label:", predicted_class_label)
```

```
p(y=e|x): 1.0
p(y=j|x): 0.0
p(y=s|x): 2.4267389118368303e-272
Predicted class label: e
```

7. Evaluate the performance of your classifier on the test set (files 10.txt to 19.txt in three languages).

Present the performance using a confusion matrix. A confusion matrix summarizes the types of errors your classifier makes, as shown in the table below. The columns are the true language a document is in, and the rows are the classified outcome of that document. The cells are the number of test documents in that situation. For example, the cell with row = English and column = Spanish contains the number of test documents that are really Spanish, but misclassified as English by your classifier.

[ ]:
```
def predict_language(document):
    bow_vector = [document.count(char) for char in vocabulary]
    log_p_x_given_e = compute_log_probability(bow_vector, theta_e)
    log_p_x_given_j = compute_log_probability(bow_vector, theta_j)
    log_p_x_given_s = compute_log_probability(bow_vector, theta_s)
    log_posterior_e = log_p_x_given_e + p_e_smoothed
    log_posterior_j = log_p_x_given_j + p_j_smoothed
    log_posterior_s = log_p_x_given_s + p_s_smoothed
    predicted_class_label = max([('e', log_posterior_e), ('j',␣
↪log_posterior_j), ('s', log_posterior_s)], key=lambda x: x[1])[0]
    return predicted_class_label

# Confusion matrix evaluation
languages = ['e', 'j', 's']
confusion_matrix = {lang1: {lang2: 0 for lang2 in languages} for lang1 in␣
↪languages}

base_path = "languageID/"
for lang in languages:
    for i in range(10, 20):
```

```
        filename = os.path.join(base_path, f"{lang}{i}.txt")
        with open(filename, 'r', encoding='utf-8', errors='ignore') as file:
            document = file.read().lower()
        predicted_label = predict_language(document)
        confusion_matrix[predicted_label][lang] += 1  # Rows: Predicted,␣
 ↪Columns: True

print(confusion_matrix)
```

```
{'e': {'e': 10, 'j': 0, 's': 0}, 'j': {'e': 0, 'j': 10, 's': 0}, 's': {'e': 0,
'j': 0, 's': 10}}
```

1.0.2 4 - Simple Feed-Forward Network

2. Implement it in NumPy or PyTorch using basic linear algebra operations. (e.g. You are not allowed to use auto-grad, built-in optimizer, model, etc. in this step. You can use library functions for data loading, processing, etc.). Evaluate your implementation on MNIST dataset, report test errors and learning curve. (10 pts)

[ ]:
```
# Load the MNIST dataset

# Transform the data to torch tensors and normalize it
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

# Load the training and test datasets
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,␣
 ↪transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,␣
 ↪transform=transform, download=True)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=100,␣
 ↪shuffle=False)
```

[ ]:
```
# Define the Network Architecture

input_size = 28*28  # MNIST images are 28x28
hidden_size1 = 300  # First hidden layer
hidden_size2 = 200  # Second hidden layer
output_size = 10   # Number of output classes (digits 0-9)

# Initialize weights and biases with Xavier initialization
W1 = torch.randn(input_size, hidden_size1) * torch.sqrt(torch.tensor(2. /␣
 ↪(input_size + hidden_size1)))
W1.requires_grad_()
b1 = torch.randn(hidden_size1, requires_grad=True)
```

```
W2 = torch.randn(hidden_size1, hidden_size2) * torch.sqrt(torch.tensor(2. /␣
↪(hidden_size1 + hidden_size2)))
W2.requires_grad_()
b2 = torch.randn(hidden_size2, requires_grad=True)

W3 = torch.randn(hidden_size2, output_size) * torch.sqrt(torch.tensor(2. /␣
↪(hidden_size2 + output_size)))
W3.requires_grad_()
b3 = torch.randn(output_size, requires_grad=True)
```

[ ]:
```
# Implement Forward and Backward Passes

def sigmoid(x):
    return 1 / (1 + torch.exp(-x))

def softmax(x):
    return torch.exp(x) / torch.sum(torch.exp(x), dim=1).view(-1, 1)

def forward(x):
    z1 = torch.mm(x, W1) + b1
    a1 = sigmoid(z1)
    z2 = torch.mm(a1, W2) + b2
    a2 = sigmoid(z2)
    z3 = torch.mm(a2, W3) + b3
    y_hat = softmax(z3)
    return y_hat, a1
```

[ ]:
```
# Train the Network

learning_rate = 1e-5
num_epochs = 10
batch_size = 64
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
↪batch_size=batch_size, shuffle=True)
loss_list = []

for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.view(-1, 28*28)

        # Forward pass
        y_hat, a1 = forward(images)

        # Convert labels to one-hot encoding
        labels_onehot = torch.zeros(labels.size(0), 10)
        labels_onehot.scatter_(1, labels.view(-1, 1), 1)
```

```
        # Compute the loss
        epsilon = 1e-7
        loss = -torch.sum(labels_onehot * torch.log(y_hat + epsilon))

        # Backward pass
        # Compute gradients
        loss.backward()

        # Update weights and biases
        with torch.no_grad():
            W1 -= learning_rate * W1.grad
            b1 -= learning_rate * b1.grad
            W2 -= learning_rate * W2.grad
            b2 -= learning_rate * b2.grad
            W3 -= learning_rate * W3.grad
            b3 -= learning_rate * b3.grad

            # Set gradients to zero for the next iteration
            W1.grad.zero_()
            b1.grad.zero_()
            W2.grad.zero_()
            b2.grad.zero_()
            W3.grad.zero_()
            b3.grad.zero_()

        if (i+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(train_loader)}], Loss: {loss.item()}')
            loss_list.append(loss.item())
```

```
Epoch [1/10], Step [100/938], Loss: 150.44065856933594
Epoch [1/10], Step [200/938], Loss: 158.19175720214844
Epoch [1/10], Step [300/938], Loss: 152.0260009765625
Epoch [1/10], Step [400/938], Loss: 149.39987182617188
Epoch [1/10], Step [500/938], Loss: 150.20960998535156
Epoch [1/10], Step [600/938], Loss: 150.26626586914062
Epoch [1/10], Step [700/938], Loss: 147.4139862060547
Epoch [1/10], Step [800/938], Loss: 144.841796875
Epoch [1/10], Step [900/938], Loss: 145.0714569091797
Epoch [2/10], Step [100/938], Loss: 144.92933654785156
Epoch [2/10], Step [200/938], Loss: 145.10665893554688
Epoch [2/10], Step [300/938], Loss: 145.93634033203125
Epoch [2/10], Step [400/938], Loss: 144.9344482421875
Epoch [2/10], Step [500/938], Loss: 145.16329956054688
Epoch [2/10], Step [600/938], Loss: 145.87161254882812
Epoch [2/10], Step [700/938], Loss: 144.6089324951172
Epoch [2/10], Step [800/938], Loss: 145.76742553710938
```

```
Epoch [2/10], Step [900/938], Loss: 145.1107177734375
Epoch [3/10], Step [100/938], Loss: 144.79183959960938
Epoch [3/10], Step [200/938], Loss: 144.9639892578125
Epoch [3/10], Step [300/938], Loss: 144.24851989746094
Epoch [3/10], Step [400/938], Loss: 144.83367919921875
Epoch [3/10], Step [500/938], Loss: 144.0528106689453
Epoch [3/10], Step [600/938], Loss: 143.1071472167969
Epoch [3/10], Step [700/938], Loss: 143.80262756347656
Epoch [3/10], Step [800/938], Loss: 144.7071533203125
Epoch [3/10], Step [900/938], Loss: 144.11297607421875
Epoch [4/10], Step [100/938], Loss: 144.13900756835938
Epoch [4/10], Step [200/938], Loss: 144.18218994140625
Epoch [4/10], Step [300/938], Loss: 143.67921447753906
Epoch [4/10], Step [400/938], Loss: 144.0732879638672
Epoch [4/10], Step [500/938], Loss: 143.0061492919922
Epoch [4/10], Step [600/938], Loss: 143.6227569580078
Epoch [4/10], Step [700/938], Loss: 142.7310333251953
Epoch [4/10], Step [800/938], Loss: 142.706787109375
Epoch [4/10], Step [900/938], Loss: 143.005859375
Epoch [5/10], Step [100/938], Loss: 142.14376831054688
Epoch [5/10], Step [200/938], Loss: 142.5039825439453
Epoch [5/10], Step [300/938], Loss: 143.33448791503906
Epoch [5/10], Step [400/938], Loss: 142.2433319091797
Epoch [5/10], Step [500/938], Loss: 141.51812744140625
Epoch [5/10], Step [600/938], Loss: 141.9885711669922
Epoch [5/10], Step [700/938], Loss: 143.1004638671875
Epoch [5/10], Step [800/938], Loss: 141.04580688476562
Epoch [5/10], Step [900/938], Loss: 140.5126190185547
Epoch [6/10], Step [100/938], Loss: 141.87103271484375
Epoch [6/10], Step [200/938], Loss: 140.72235107421875
Epoch [6/10], Step [300/938], Loss: 140.95265197753906
Epoch [6/10], Step [400/938], Loss: 141.5476837158203
Epoch [6/10], Step [500/938], Loss: 140.26698303222656
Epoch [6/10], Step [600/938], Loss: 140.1280517578125
Epoch [6/10], Step [700/938], Loss: 140.3040313720703
Epoch [6/10], Step [800/938], Loss: 141.64251708984375
Epoch [6/10], Step [900/938], Loss: 140.82797241210938
Epoch [7/10], Step [100/938], Loss: 139.7085723876953
Epoch [7/10], Step [200/938], Loss: 140.82366943359375
Epoch [7/10], Step [300/938], Loss: 138.39443969726562
Epoch [7/10], Step [400/938], Loss: 141.16297912597656
Epoch [7/10], Step [500/938], Loss: 140.76480102539062
Epoch [7/10], Step [600/938], Loss: 140.22271728515625
Epoch [7/10], Step [700/938], Loss: 139.89588928222656
Epoch [7/10], Step [800/938], Loss: 139.31581115722656
Epoch [7/10], Step [900/938], Loss: 139.6656951904297
Epoch [8/10], Step [100/938], Loss: 139.0992889404297
Epoch [8/10], Step [200/938], Loss: 139.98033142089844
```

```
Epoch [8/10], Step [300/938], Loss: 138.3582763671875
Epoch [8/10], Step [400/938], Loss: 138.02357482910156
Epoch [8/10], Step [500/938], Loss: 140.56471252441406
Epoch [8/10], Step [600/938], Loss: 136.2198486328125
Epoch [8/10], Step [700/938], Loss: 137.43252563476562
Epoch [8/10], Step [800/938], Loss: 135.9687957763672
Epoch [8/10], Step [900/938], Loss: 138.53297424316406
Epoch [9/10], Step [100/938], Loss: 137.4494171142578
Epoch [9/10], Step [200/938], Loss: 138.25155639648438
Epoch [9/10], Step [300/938], Loss: 137.78530883789062
Epoch [9/10], Step [400/938], Loss: 138.82151794433594
Epoch [9/10], Step [500/938], Loss: 138.03427124023438
Epoch [9/10], Step [600/938], Loss: 139.8271942138672
Epoch [9/10], Step [700/938], Loss: 138.09976196289062
Epoch [9/10], Step [800/938], Loss: 136.7034912109375
Epoch [9/10], Step [900/938], Loss: 138.18231201171875
Epoch [10/10], Step [100/938], Loss: 139.20860290527344
Epoch [10/10], Step [200/938], Loss: 135.65049743652344
Epoch [10/10], Step [300/938], Loss: 138.6070556640625
Epoch [10/10], Step [400/938], Loss: 134.47235107421875
Epoch [10/10], Step [500/938], Loss: 132.58444213867188
Epoch [10/10], Step [600/938], Loss: 135.80419921875
Epoch [10/10], Step [700/938], Loss: 135.9111328125
Epoch [10/10], Step [800/938], Loss: 135.47509765625
Epoch [10/10], Step [900/938], Loss: 134.71739196777344
```

[ ]:
```python
# Test the Network and Report Test Errors

correct = 0
total = 0

with torch.no_grad():
    for images, labels in test_loader:
        images = images.view(-1, 28*28)
        y_hat, _ = forward(images)
        _, predicted = torch.max(y_hat.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()

print(f'Accuracy of the network on the 10000 test images: {100 * correct /␣
 ↪total} %')
```

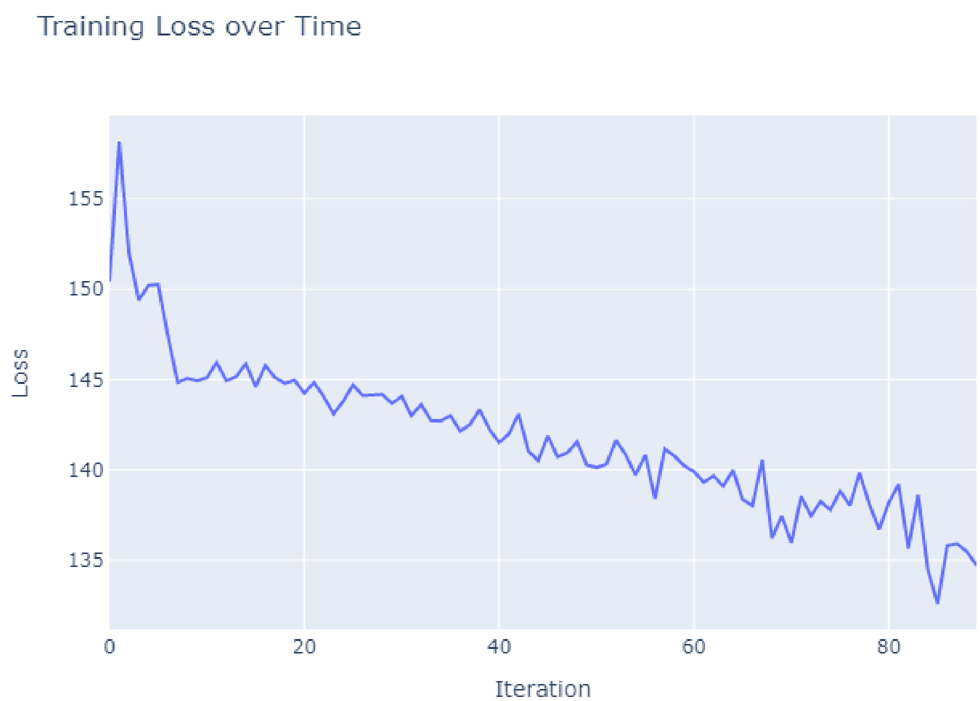
```
Accuracy of the network on the 10000 test images: 54.40999984741211 %
```

[ ]:
```python
import plotly.graph_objects as go

# Create the figure
fig = go.Figure()
```

```python
# Add the loss curve data
fig.add_trace(go.Scatter(y=loss_list, mode='lines', name='Loss'))

# Set the layout for the figure
fig.update_layout(
    title='Training Loss over Time',
    xaxis_title='Iteration',
    yaxis_title='Loss',
    xaxis=dict(showgrid=True, showline=True, showticklabels=True),
    yaxis=dict(showgrid=True, showline=True, showticklabels=True)
)

# Display the figure
fig.show()
```

[ ]:
```python
img_bytes = pio.to_image(fig, format="png")
Image(img_bytes)
```

[ ]:



3. Implement the same network in PyTorch (or any other framework). You can use all the features of the framework e.g. auto-grad etc. Evaluate it on MNIST dataset, report test errors, and learning curve. (2 pts)

```
[ ]: # Load the training and test datasets
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
 ↪5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,␣
 ↪transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,␣
 ↪transform=transform)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=64,␣
 ↪shuffle=False)
```

```
[ ]: # Define the Network Architecture with Sigmoid for hidden layers
class NeuralNetworkSigmoid(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(NeuralNetworkSigmoid, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_size)
        self.sigmoid = nn.Sigmoid()

        # Xavier (Glorot) Initialization for Sigmoid
        nn.init.xavier_uniform_(self.fc1.weight)
        nn.init.xavier_uniform_(self.fc2.weight)

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        x = self.fc3(x)  # removed softmax here
        return x

# Initialize the network with the specified dimensions
model_sigmoid = NeuralNetworkSigmoid(28*28, 300, 200, 10)
```

```
[ ]: # Set Loss Function and Optimizer
learning_rate = 0.01
optimizer = optim.SGD(model_sigmoid.parameters(), lr=learning_rate)  # SGD␣
 ↪without momentum
```

```
[ ]: # Train the Network
num_epochs = 10
loss_list_sigmoid = []
criterion = nn.CrossEntropyLoss()

for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.view(-1, 28*28)
```

```python
        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model_sigmoid(images)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(train_loader)}], Loss: {loss.item()}')
            loss_list_sigmoid.append(loss.item())
```

```
Epoch [1/10], Step [100/938], Loss: 2.292510747909546
Epoch [1/10], Step [200/938], Loss: 2.2745885848999023
Epoch [1/10], Step [300/938], Loss: 2.236137866973877
Epoch [1/10], Step [400/938], Loss: 2.2385945320129395
Epoch [1/10], Step [500/938], Loss: 2.246877431869507
Epoch [1/10], Step [600/938], Loss: 2.2270617485046387
Epoch [1/10], Step [700/938], Loss: 2.2525980472564697
Epoch [1/10], Step [800/938], Loss: 2.221071243286133
Epoch [1/10], Step [900/938], Loss: 2.1858716011047363
Epoch [2/10], Step [100/938], Loss: 2.1568338871002197
Epoch [2/10], Step [200/938], Loss: 2.119809150695801
Epoch [2/10], Step [300/938], Loss: 2.0830423831939697
Epoch [2/10], Step [400/938], Loss: 2.0672149658203125
Epoch [2/10], Step [500/938], Loss: 1.9432536363601685
Epoch [2/10], Step [600/938], Loss: 1.9893263578414917
Epoch [2/10], Step [700/938], Loss: 1.715041160583496
Epoch [2/10], Step [800/938], Loss: 1.8554497957229614
Epoch [2/10], Step [900/938], Loss: 1.6881569623947144
Epoch [3/10], Step [100/938], Loss: 1.6012632846832275
Epoch [3/10], Step [200/938], Loss: 1.6248172521591187
Epoch [3/10], Step [300/938], Loss: 1.4931164979934692
Epoch [3/10], Step [400/938], Loss: 1.423913836479187
Epoch [3/10], Step [500/938], Loss: 1.5174356698989868
Epoch [3/10], Step [600/938], Loss: 1.3259869813919067
Epoch [3/10], Step [700/938], Loss: 1.2630555629730225
Epoch [3/10], Step [800/938], Loss: 1.1542565822601318
Epoch [3/10], Step [900/938], Loss: 1.1549248695373535
Epoch [4/10], Step [100/938], Loss: 1.037700891494751
Epoch [4/10], Step [200/938], Loss: 1.0694125890731812
```

```
Epoch [4/10], Step [300/938], Loss: 0.9972710013389587
Epoch [4/10], Step [400/938], Loss: 1.0252350568771362
Epoch [4/10], Step [500/938], Loss: 1.0819385051727295
Epoch [4/10], Step [600/938], Loss: 1.0091618299484253
Epoch [4/10], Step [700/938], Loss: 0.8767440319061279
Epoch [4/10], Step [800/938], Loss: 0.7152828574180603
Epoch [4/10], Step [900/938], Loss: 0.8610665798187256
Epoch [5/10], Step [100/938], Loss: 0.9560669660568237
Epoch [5/10], Step [200/938], Loss: 0.7160088419914246
Epoch [5/10], Step [300/938], Loss: 0.8273577690124512
Epoch [5/10], Step [400/938], Loss: 0.7674003839492798
Epoch [5/10], Step [500/938], Loss: 0.82974773645401
Epoch [5/10], Step [600/938], Loss: 0.7529183030128479
Epoch [5/10], Step [700/938], Loss: 0.6722601652145386
Epoch [5/10], Step [800/938], Loss: 0.7306991815567017
Epoch [5/10], Step [900/938], Loss: 0.7301172614097595
Epoch [6/10], Step [100/938], Loss: 0.7639034390449524
Epoch [6/10], Step [200/938], Loss: 0.7602452039718628
Epoch [6/10], Step [300/938], Loss: 0.5960515141487122
Epoch [6/10], Step [400/938], Loss: 0.9589837789535522
Epoch [6/10], Step [500/938], Loss: 0.6737836003303528
Epoch [6/10], Step [600/938], Loss: 0.6552358865737915
Epoch [6/10], Step [700/938], Loss: 0.4662915766239166
Epoch [6/10], Step [800/938], Loss: 0.6583096981048584
Epoch [6/10], Step [900/938], Loss: 0.523502767086029
Epoch [7/10], Step [100/938], Loss: 0.7014241218566895
Epoch [7/10], Step [200/938], Loss: 0.6115772128105164
Epoch [7/10], Step [300/938], Loss: 0.722679853439331
Epoch [7/10], Step [400/938], Loss: 0.5301729440689087
Epoch [7/10], Step [500/938], Loss: 0.5377673506736755
Epoch [7/10], Step [600/938], Loss: 0.8452486991882324
Epoch [7/10], Step [700/938], Loss: 0.512640118598938
Epoch [7/10], Step [800/938], Loss: 0.41439181566238403
Epoch [7/10], Step [900/938], Loss: 0.4587152898311615
Epoch [8/10], Step [100/938], Loss: 0.5379469394683838
Epoch [8/10], Step [200/938], Loss: 0.4712846875190735
Epoch [8/10], Step [300/938], Loss: 0.4268187880516052
Epoch [8/10], Step [400/938], Loss: 0.4376557171344757
Epoch [8/10], Step [500/938], Loss: 0.505620002746582
Epoch [8/10], Step [600/938], Loss: 0.43054917454719543
Epoch [8/10], Step [700/938], Loss: 0.6603127121925354
Epoch [8/10], Step [800/938], Loss: 0.4829855263233185
Epoch [8/10], Step [900/938], Loss: 0.43703171610832214
Epoch [9/10], Step [100/938], Loss: 0.4643995463848114
Epoch [9/10], Step [200/938], Loss: 0.7232795357704163
Epoch [9/10], Step [300/938], Loss: 0.39175742864608765
Epoch [9/10], Step [400/938], Loss: 0.6205026507377625
Epoch [9/10], Step [500/938], Loss: 0.5583638548851013
```

```
Epoch [9/10], Step [600/938], Loss: 0.3877916932106018
Epoch [9/10], Step [700/938], Loss: 0.5199547410011292
Epoch [9/10], Step [800/938], Loss: 0.38301512598991394
Epoch [9/10], Step [900/938], Loss: 0.4598042368888855
Epoch [10/10], Step [100/938], Loss: 0.5218845009803772
Epoch [10/10], Step [200/938], Loss: 0.39239010214805603
Epoch [10/10], Step [300/938], Loss: 0.5380221605300903
Epoch [10/10], Step [400/938], Loss: 0.48484063148498535
Epoch [10/10], Step [500/938], Loss: 0.47320041060447693
Epoch [10/10], Step [600/938], Loss: 0.474805623292923
Epoch [10/10], Step [700/938], Loss: 0.6889896392822266
Epoch [10/10], Step [800/938], Loss: 0.4263448417186737
Epoch [10/10], Step [900/938], Loss: 0.3722385764122009
```

[ ]:
```
# Test the Network
correct_sigmoid = 0
total_sigmoid = 0

with torch.no_grad():
    for images, labels in test_loader:
        images = images.view(-1, 28*28)
        outputs = model_sigmoid(images)
        _, predicted = torch.max(outputs.data, 1)
        total_sigmoid += labels.size(0)
        correct_sigmoid += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 *␣
 ↪correct_sigmoid / total_sigmoid} %')
print(f'Number of parameters in the network: {sum(p.numel() for p in␣
 ↪model_sigmoid.parameters())}')
print(f'Hyperparameters: {num_epochs}, {batch_size}, {learning_rate}')
```

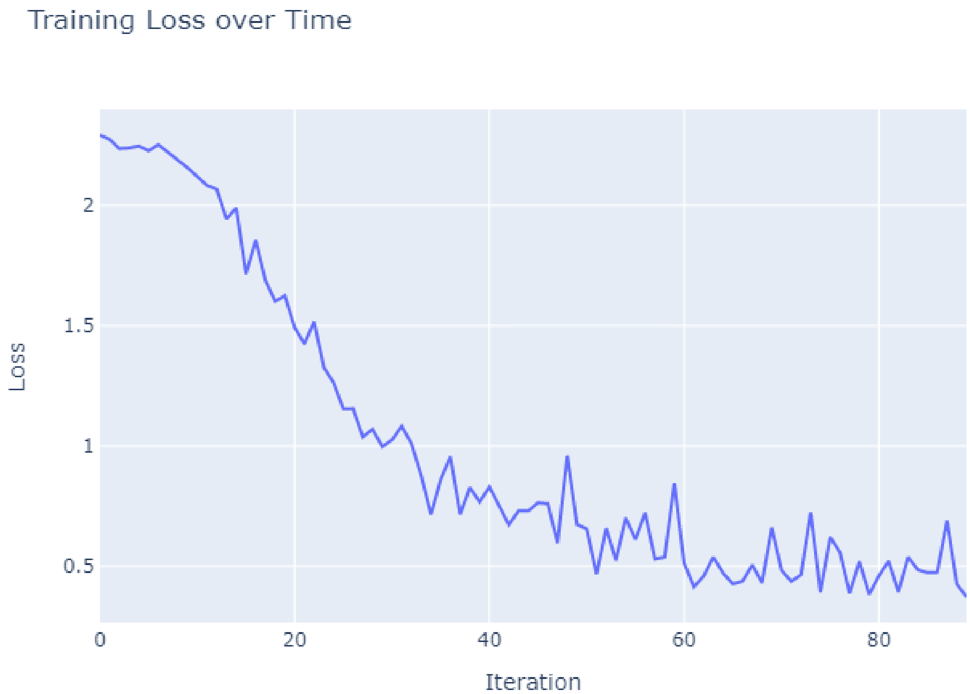
```
Accuracy of the network on the 10000 test images: 88.89 %
Number of parameters in the network: 297710
Hyperparameters: 10, 64, 0.01
```

[ ]:
```
# Create the figure
fig = go.Figure()

# Add the loss curve data
fig.add_trace(go.Scatter(y=loss_list_sigmoid, mode='lines', name='Loss'))

# Set the layout for the figure
fig.update_layout(
    title='Training Loss over Time',
    xaxis_title='Iteration',
    yaxis_title='Loss',
    xaxis=dict(showgrid=True, showline=True, showticklabels=True),
```

```
    yaxis=dict(showgrid=True, showline=True, showticklabels=True)
)

# Display the figure
fig.show()
```

[ ]:
```
img_bytes = pio.to_image(fig, format="png")
Image(img_bytes)
```

[ ]:



4. Try different weight initialization a) all weights initialized to 0, and b) initialize the weights randomly between -1 and 1. Report test error and learning curves for both. (You can use either of the implementations) (3 pts)

[ ]:
```
# Load the training and test datasets
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,␣
↪transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,␣
↪transform=transform)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
↪batch_size=64, shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=64,␣
 ↪shuffle=False)
```

[ ]:
```
# Define the Network Architecture with Sigmoid for hidden layers
class NeuralNetworkSigmoid(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(NeuralNetworkSigmoid, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x
```

[ ]:
```
# Define custom weight initializations
def zero_weights_initialization(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            nn.init.constant_(m.weight, 0)
            nn.init.constant_(m.bias, 0)

def random_weights_initialization(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            nn.init.uniform_(m.weight, -1, 1)
            nn.init.uniform_(m.bias, -1, 1)

# Initialize models and apply custom weight initializations
model_zero = NeuralNetworkSigmoid(28*28, 300, 200, 10)
model_random = NeuralNetworkSigmoid(28*28, 300, 200, 10)

zero_weights_initialization(model_zero)
random_weights_initialization(model_random)
```

[ ]:
```
# Training function
def train_model(model, num_epochs=10):
    learning_rate = 0.01
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
    criterion = nn.CrossEntropyLoss()
    loss_list = []

    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_loader):
```

```
            images = images.view(-1, 28*28)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            if (i+1) % 100 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
 ↪{len(train_loader)}], Loss: {loss.item()}')
                loss_list.append(loss.item())

    return loss_list

# Testing function
def test_model(model):
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images = images.view(-1, 28*28)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

[ ]:
```
# Train and test functions remain the same ...

loss_list_zero = train_model(model_zero)
loss_list_random = train_model(model_random)

accuracy_zero = test_model(model_zero)
accuracy_random = test_model(model_random)

print(f'Accuracy with Zero Initialization: {accuracy_zero} %')
print(f'Number of parameters in the network: {sum(p.numel() for p in model_zero.
 ↪parameters())}')
print(f'Hyperparameters: {num_epochs}, {batch_size}, {learning_rate}')
```

```
Epoch [1/10], Step [100/938], Loss: 2.2959036827087402
Epoch [1/10], Step [200/938], Loss: 2.3098185062408447
Epoch [1/10], Step [300/938], Loss: 2.2983412742614746
Epoch [1/10], Step [400/938], Loss: 2.300004243850708
Epoch [1/10], Step [500/938], Loss: 2.324035882949829
Epoch [1/10], Step [600/938], Loss: 2.2947351932525635
Epoch [1/10], Step [700/938], Loss: 2.300985813140869
```

```
Epoch [1/10], Step [800/938], Loss: 2.2945473194122314
Epoch [1/10], Step [900/938], Loss: 2.2986879348754883
Epoch [2/10], Step [100/938], Loss: 2.2957682609558105
Epoch [2/10], Step [200/938], Loss: 2.3193600177764893
Epoch [2/10], Step [300/938], Loss: 2.3031725883483887
Epoch [2/10], Step [400/938], Loss: 2.308579444885254
Epoch [2/10], Step [500/938], Loss: 2.313770055770874
Epoch [2/10], Step [600/938], Loss: 2.291682481765747
Epoch [2/10], Step [700/938], Loss: 2.3063442707061768
Epoch [2/10], Step [800/938], Loss: 2.308854103088379
Epoch [2/10], Step [900/938], Loss: 2.3122706413269043
Epoch [3/10], Step [100/938], Loss: 2.3091959953308105
Epoch [3/10], Step [200/938], Loss: 2.3086392879486084
Epoch [3/10], Step [300/938], Loss: 2.2995505332946777
Epoch [3/10], Step [400/938], Loss: 2.315145969390869
Epoch [3/10], Step [500/938], Loss: 2.2901668548583984
Epoch [3/10], Step [600/938], Loss: 2.297959327697754
Epoch [3/10], Step [700/938], Loss: 2.3273708820343018
Epoch [3/10], Step [800/938], Loss: 2.282715082168579
Epoch [3/10], Step [900/938], Loss: 2.2930617332458496
Epoch [4/10], Step [100/938], Loss: 2.3076090812683105
Epoch [4/10], Step [200/938], Loss: 2.3027360439300537
Epoch [4/10], Step [300/938], Loss: 2.294618844985962
Epoch [4/10], Step [400/938], Loss: 2.286463499069214
Epoch [4/10], Step [500/938], Loss: 2.3102128505706787
Epoch [4/10], Step [600/938], Loss: 2.3031625747680664
Epoch [4/10], Step [700/938], Loss: 2.2965002059936523
Epoch [4/10], Step [800/938], Loss: 2.3000760078430176
Epoch [4/10], Step [900/938], Loss: 2.2989449501037598
Epoch [5/10], Step [100/938], Loss: 2.287170171737671
Epoch [5/10], Step [200/938], Loss: 2.313420057296753
Epoch [5/10], Step [300/938], Loss: 2.2981622219085693
Epoch [5/10], Step [400/938], Loss: 2.293731451034546
Epoch [5/10], Step [500/938], Loss: 2.299309730529785
Epoch [5/10], Step [600/938], Loss: 2.298801898956299
Epoch [5/10], Step [700/938], Loss: 2.2951977252960205
Epoch [5/10], Step [800/938], Loss: 2.304935932159424
Epoch [5/10], Step [900/938], Loss: 2.288813591003418
Epoch [6/10], Step [100/938], Loss: 2.3064780235290527
Epoch [6/10], Step [200/938], Loss: 2.309649705886841
Epoch [6/10], Step [300/938], Loss: 2.2970387935638428
Epoch [6/10], Step [400/938], Loss: 2.306081771850586
Epoch [6/10], Step [500/938], Loss: 2.301093101501465
Epoch [6/10], Step [600/938], Loss: 2.316398859024048
Epoch [6/10], Step [700/938], Loss: 2.2961008548736572
Epoch [6/10], Step [800/938], Loss: 2.3125345706939697
Epoch [6/10], Step [900/938], Loss: 2.3104517459869385
Epoch [7/10], Step [100/938], Loss: 2.284198522567749
```

```
Epoch [7/10], Step [200/938], Loss: 2.28190279006958
Epoch [7/10], Step [300/938], Loss: 2.2970802783966064
Epoch [7/10], Step [400/938], Loss: 2.292177438735962
Epoch [7/10], Step [500/938], Loss: 2.3092918395996094
Epoch [7/10], Step [600/938], Loss: 2.3049864768981934
Epoch [7/10], Step [700/938], Loss: 2.289271593093872
Epoch [7/10], Step [800/938], Loss: 2.294886350631714
Epoch [7/10], Step [900/938], Loss: 2.3200173377990723
Epoch [8/10], Step [100/938], Loss: 2.28857421875
Epoch [8/10], Step [200/938], Loss: 2.285102605819702
Epoch [8/10], Step [300/938], Loss: 2.3046154975891113
Epoch [8/10], Step [400/938], Loss: 2.3181662559509277
Epoch [8/10], Step [500/938], Loss: 2.2945666313171387
Epoch [8/10], Step [600/938], Loss: 2.302605152130127
Epoch [8/10], Step [700/938], Loss: 2.29838228225708
Epoch [8/10], Step [800/938], Loss: 2.3037962913513184
Epoch [8/10], Step [900/938], Loss: 2.3151023387908936
Epoch [9/10], Step [100/938], Loss: 2.300825357437134
Epoch [9/10], Step [200/938], Loss: 2.297396421432495
Epoch [9/10], Step [300/938], Loss: 2.304267168045044
Epoch [9/10], Step [400/938], Loss: 2.3073456287384033
Epoch [9/10], Step [500/938], Loss: 2.298769474029541
Epoch [9/10], Step [600/938], Loss: 2.3157100677490234
Epoch [9/10], Step [700/938], Loss: 2.2858476638793945
Epoch [9/10], Step [800/938], Loss: 2.289113759994507
Epoch [9/10], Step [900/938], Loss: 2.312711477279663
Epoch [10/10], Step [100/938], Loss: 2.3149399757385254
Epoch [10/10], Step [200/938], Loss: 2.302381992340088
Epoch [10/10], Step [300/938], Loss: 2.312058448791504
Epoch [10/10], Step [400/938], Loss: 2.3146228790283203
Epoch [10/10], Step [500/938], Loss: 2.3108367919921875
Epoch [10/10], Step [600/938], Loss: 2.2990293502807617
Epoch [10/10], Step [700/938], Loss: 2.3037703037261963
Epoch [10/10], Step [800/938], Loss: 2.301426649093628
Epoch [10/10], Step [900/938], Loss: 2.3111307621002197
Epoch [1/10], Step [100/938], Loss: 2.953606128692627
Epoch [1/10], Step [200/938], Loss: 1.9727871417999268
Epoch [1/10], Step [300/938], Loss: 1.8311190605163574
Epoch [1/10], Step [400/938], Loss: 1.6887493133544922
Epoch [1/10], Step [500/938], Loss: 1.4824713468551636
Epoch [1/10], Step [600/938], Loss: 1.508542776107788
Epoch [1/10], Step [700/938], Loss: 1.0630067586898804
Epoch [1/10], Step [800/938], Loss: 1.2491583824157715
Epoch [1/10], Step [900/938], Loss: 0.9693819880485535
Epoch [2/10], Step [100/938], Loss: 0.8525758981704712
Epoch [2/10], Step [200/938], Loss: 1.0639961957931519
Epoch [2/10], Step [300/938], Loss: 1.3694777488708496
Epoch [2/10], Step [400/938], Loss: 0.7700785994529724
```

```
Epoch [2/10], Step [500/938], Loss: 0.856069028377533
Epoch [2/10], Step [600/938], Loss: 1.098111629486084
Epoch [2/10], Step [700/938], Loss: 0.8513131141662598
Epoch [2/10], Step [800/938], Loss: 1.1163603067398071
Epoch [2/10], Step [900/938], Loss: 1.30469810962677
Epoch [3/10], Step [100/938], Loss: 0.8735064268112183
Epoch [3/10], Step [200/938], Loss: 1.0822932720184326
Epoch [3/10], Step [300/938], Loss: 0.6591507792472839
Epoch [3/10], Step [400/938], Loss: 0.9457104802131653
Epoch [3/10], Step [500/938], Loss: 0.8144713044166565
Epoch [3/10], Step [600/938], Loss: 1.1147946119308472
Epoch [3/10], Step [700/938], Loss: 0.9592292308807373
Epoch [3/10], Step [800/938], Loss: 0.5626049637794495
Epoch [3/10], Step [900/938], Loss: 0.8171972632408142
Epoch [4/10], Step [100/938], Loss: 0.6341459155082703
Epoch [4/10], Step [200/938], Loss: 0.6588419079780579
Epoch [4/10], Step [300/938], Loss: 0.6808494925498962
Epoch [4/10], Step [400/938], Loss: 0.5139976739883423
Epoch [4/10], Step [500/938], Loss: 0.7045794725418091
Epoch [4/10], Step [600/938], Loss: 0.6749289035797119
Epoch [4/10], Step [700/938], Loss: 0.800560712814331
Epoch [4/10], Step [800/938], Loss: 0.7610225677490234
Epoch [4/10], Step [900/938], Loss: 0.5226768851280212
Epoch [5/10], Step [100/938], Loss: 0.48000991344451904
Epoch [5/10], Step [200/938], Loss: 0.7091707587242126
Epoch [5/10], Step [300/938], Loss: 0.6106627583503723
Epoch [5/10], Step [400/938], Loss: 0.5412269234657288
Epoch [5/10], Step [500/938], Loss: 0.900904655456543
Epoch [5/10], Step [600/938], Loss: 0.6600114107131958
Epoch [5/10], Step [700/938], Loss: 0.5399045348167419
Epoch [5/10], Step [800/938], Loss: 0.7446325421333313
Epoch [5/10], Step [900/938], Loss: 0.4665955603122711
Epoch [6/10], Step [100/938], Loss: 0.5178843140602112
Epoch [6/10], Step [200/938], Loss: 0.30246806144714355
Epoch [6/10], Step [300/938], Loss: 0.47359079122543335
Epoch [6/10], Step [400/938], Loss: 0.5458940267562866
Epoch [6/10], Step [500/938], Loss: 0.37737202644348145
Epoch [6/10], Step [600/938], Loss: 0.49188363552093506
Epoch [6/10], Step [700/938], Loss: 0.5421923995018005
Epoch [6/10], Step [800/938], Loss: 0.6272466778755188
Epoch [6/10], Step [900/938], Loss: 0.5193513035774231
Epoch [7/10], Step [100/938], Loss: 0.26116812229156494
Epoch [7/10], Step [200/938], Loss: 0.590369701385498
Epoch [7/10], Step [300/938], Loss: 0.4917355179786682
Epoch [7/10], Step [400/938], Loss: 0.5760741233825684
Epoch [7/10], Step [500/938], Loss: 0.6997534036636353
Epoch [7/10], Step [600/938], Loss: 0.5752143859863281
Epoch [7/10], Step [700/938], Loss: 0.4811353385448456
```

```
Epoch [7/10], Step [800/938], Loss: 0.5643269419670105
Epoch [7/10], Step [900/938], Loss: 0.573438286781311
Epoch [8/10], Step [100/938], Loss: 0.7224077582359314
Epoch [8/10], Step [200/938], Loss: 0.64201819896698
Epoch [8/10], Step [300/938], Loss: 0.3862292766571045
Epoch [8/10], Step [400/938], Loss: 0.4425695836544037
Epoch [8/10], Step [500/938], Loss: 0.7631415128707886
Epoch [8/10], Step [600/938], Loss: 0.33784785866737366
Epoch [8/10], Step [700/938], Loss: 0.4337313175201416
Epoch [8/10], Step [800/938], Loss: 0.556617796421051
Epoch [8/10], Step [900/938], Loss: 0.5486451387405396
Epoch [9/10], Step [100/938], Loss: 0.48117464780807495
Epoch [9/10], Step [200/938], Loss: 0.5255522727966309
Epoch [9/10], Step [300/938], Loss: 0.5729469656944275
Epoch [9/10], Step [400/938], Loss: 0.6391357779502869
Epoch [9/10], Step [500/938], Loss: 0.5159413814544678
Epoch [9/10], Step [600/938], Loss: 0.44273942708969116
Epoch [9/10], Step [700/938], Loss: 0.26776325702667236
Epoch [9/10], Step [800/938], Loss: 0.6065534949302673
Epoch [9/10], Step [900/938], Loss: 0.407464861869812
Epoch [10/10], Step [100/938], Loss: 0.34952011704444885
Epoch [10/10], Step [200/938], Loss: 0.3152521252632141
Epoch [10/10], Step [300/938], Loss: 0.4001403748989105
Epoch [10/10], Step [400/938], Loss: 0.4168870151042938
Epoch [10/10], Step [500/938], Loss: 0.4659384787082672
Epoch [10/10], Step [600/938], Loss: 0.28651031851768494
Epoch [10/10], Step [700/938], Loss: 0.4751476049423218
Epoch [10/10], Step [800/938], Loss: 0.5462862253189087
Epoch [10/10], Step [900/938], Loss: 0.4771941006183624
Accuracy with Zero Initialization: 10.28 %
Number of parameters in the network: 297710
Hyperparameters: 10, 64, 0.01
```

[ ]:
```
print(f'Accuracy with Random Initialization: {accuracy_random} %')
print(f'Number of parameters in the network: {sum(p.numel() for p in␣
↪model_random.parameters())}')
print(f'Hyperparameters: {num_epochs}, {batch_size}, {learning_rate}')
```

```
Accuracy with Random Initialization: 85.8 %
Number of parameters in the network: 297710
Hyperparameters: 10, 64, 0.01
```

[ ]:
```
# Create the figure
fig = go.Figure()

# Add the loss curve data for both initializations
fig.add_trace(go.Scatter(y=loss_list_zero, mode='lines', name='Zero␣
↪Initialization'))
```

```
fig.add_trace(go.Scatter(y=loss_list_random, mode='lines', name='Random␣
    ↪Initialization'))

# Set the layout for the figure
fig.update_layout(
    title='Training Loss over Time',
    xaxis_title='Iteration',
    yaxis_title='Loss',
    xaxis=dict(showgrid=True, showline=True, showticklabels=True),
    yaxis=dict(showgrid=True, showline=True, showticklabels=True)
)

# Display the figure
fig.show()
```

[ ]:
```
img_bytes = pio.to_image(fig, format="png")
Image(img_bytes)
```

[ ]: