hw4

November 13, 2024

1 Overview: HW3 - Question 4

In this coding question, you'll implement a classifier with logistic regression

$$F(w) = \frac{1}{N}\sum_{i=1}^{N}\log(1 + e^{-\langle w, x_i \rangle y_i}).$$

For this problem, I would suggest using functions to prepare the dataset, run gradient descent, and return classification error. By doing this, you only have to write the code one time and just use the functions to return results for part (4c).

2 Loading MNIST Data

In this section, you will learn to load MNIST data. If you do not have tensorflow available on your jupyter notebook, uncomment the next cell, run it, restart the kernel, and comment the next cell once more.

[1]:
```
#!pip3 install sklearn
!pip3 install scikit-learn
```

```
Requirement already satisfied: scikit-learn in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(1.5.2)
Requirement already satisfied: numpy>=1.19.5 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (1.14.1)
Requirement already satisfied: joblib>=1.2.0 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from scikit-learn) (3.5.0)
```

[2]:
```
# import statements
import pandas as pd
import numpy as np
```

```python
from matplotlib import pyplot as plt
from sklearn.datasets import fetch_openml
```

[3]:
```python
!pip install --upgrade certifi
!brew install openssl
!brew link openssl --force
import certifi
import ssl
import os

os.environ['SSL_CERT_FILE'] = certifi.where()
import requests

url = 'https://example.com'
response = requests.get(url, verify=False)
!/Applications/Python\ 3.12.4/Install\ Certificates.command
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

```
Requirement already satisfied: certifi in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(2024.8.30)
==> Downloading https://formulae.brew.sh/api/formula.jws.json
######################################################################### 100.0%
==> Downloading https://formulae.brew.sh/api/cask.jws.json
######################################################################### 100.0%
Warning: openssl@3 3.4.0 is already installed and up-to-date.
To reinstall 3.4.0, run:
  brew reinstall openssl@3
Warning: Already linked: /opt/homebrew/Cellar/openssl@3/3.4.0
To relink, run:
  brew unlink openssl@3 && brew link openssl@3
zsh:1: no such file or directory: /Applications/Python 3.12.4/Install
Certificates.command

/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/urllib3/connectionpool.py:1099: InsecureRequestWarning: Unverified
HTTPS request is being made to host 'example.com'. Adding certificate
verification is strongly advised. See:
https://urllib3.readthedocs.io/en/latest/advanced-usage.html#tls-warnings
  warnings.warn(
```

[7]:
```python
# this cell will take a minute to run depending on your internet connection
X, y = fetch_openml('mnist_784', version=1, return_X_y=True) # getting data␣
 ↪from online

print('X shape:', X.shape, 'y shape:', y.shape)
```

```
X shape: (70000, 784) y shape: (70000,)
```

[8]:
```
# this cell processes some of the data

# if this returns an error of the form "KeyError: 0", then try running the␣
↪following first:
X = X.values # this converts X from a pandas dataframe to a numpy array

digits = {j:[] for j in range(10)}
for j in range(len(y)): # takes data assigns it into a dictionary
    digits[int(y[j])].append(X[j].reshape(28,28))
digits = {j:np.stack(digits[j]) for j in range(10)} # stack everything to be␣
↪one numpy array
for j in range(10):
    print('Shape of data with label', j, ':', digits[j].shape )
```

```
Shape of data with label 0 : (6903, 28, 28)
Shape of data with label 1 : (7877, 28, 28)
Shape of data with label 2 : (6990, 28, 28)
Shape of data with label 3 : (7141, 28, 28)
Shape of data with label 4 : (6824, 28, 28)
Shape of data with label 5 : (6313, 28, 28)
Shape of data with label 6 : (6876, 28, 28)
Shape of data with label 7 : (7293, 28, 28)
Shape of data with label 8 : (6825, 28, 28)
Shape of data with label 9 : (6958, 28, 28)
```

[9]:
```
# this cell would stack 100 examples from each class together
# this cell also ensures that each pixel is a flot between 0 and 1 instead of␣
↪an int between 0 and 255
data = []
for i in range(10):
    flattened_images = digits[i][:100].reshape(100,-1)
    data.append(flattened_images)

data = np.vstack(data)
data = data.astype('float32') / 255.0
```

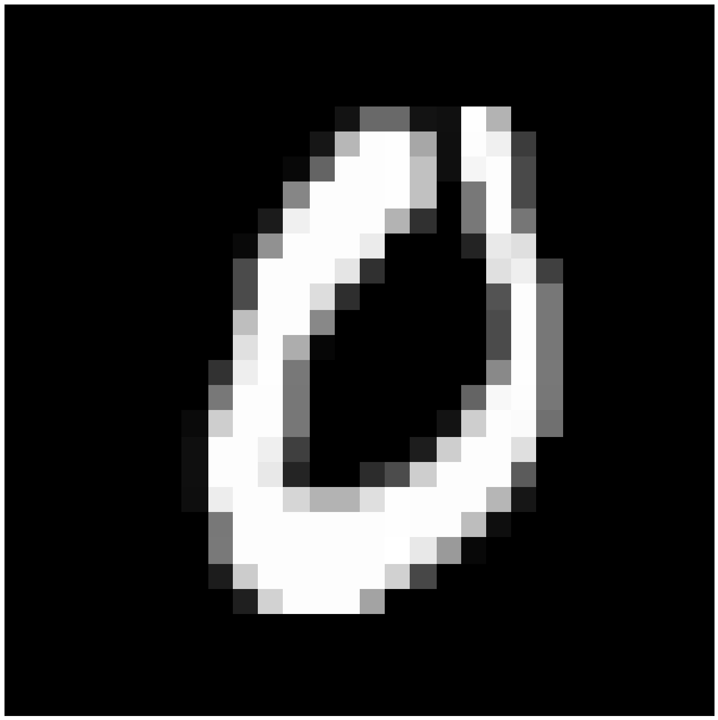
3 (4a) Plotting

Display one randomly selected image from your training data for each digit class. Provide the index number for each image.

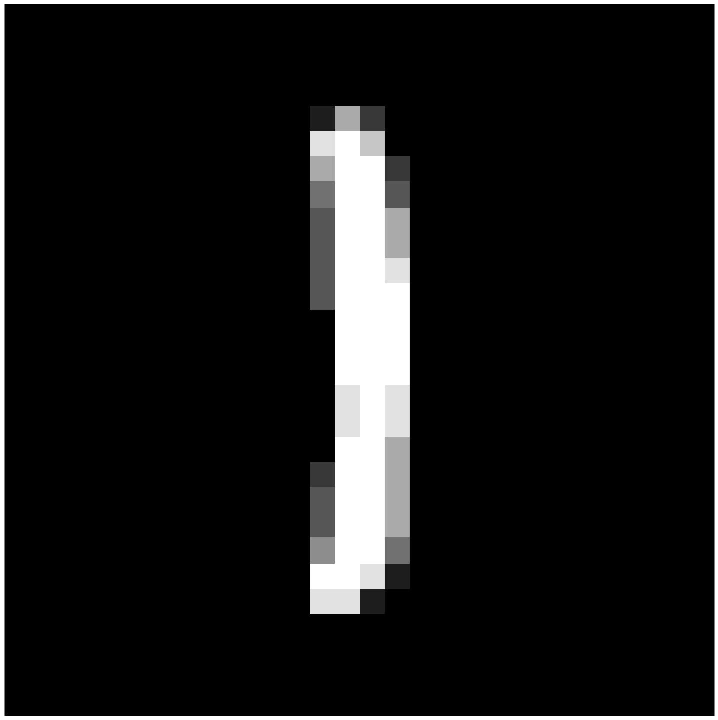
[10]:
```
import random
import matplotlib.pyplot as plt
#image_index = random.randint(0,500)
for i in range(10):
    image_index = random.randint(0, len(digits[i]))
    image = digits[i][image_index]
```

```python
    plt.imshow(image, cmap='gray')
    plt.title(f"digit: {i}, index: {image_index}")
    plt.axis('off')
    plt.show()
```

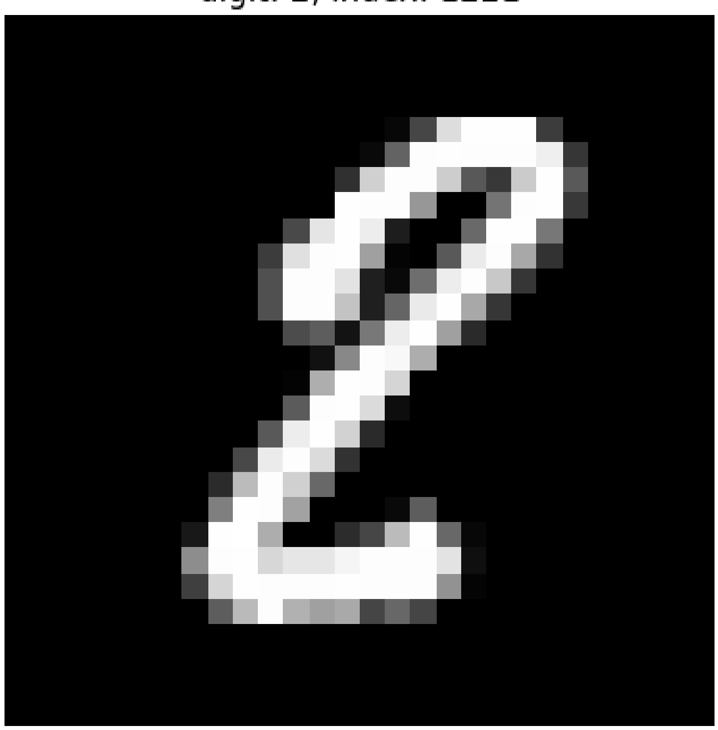
digit: 0, index: 5556



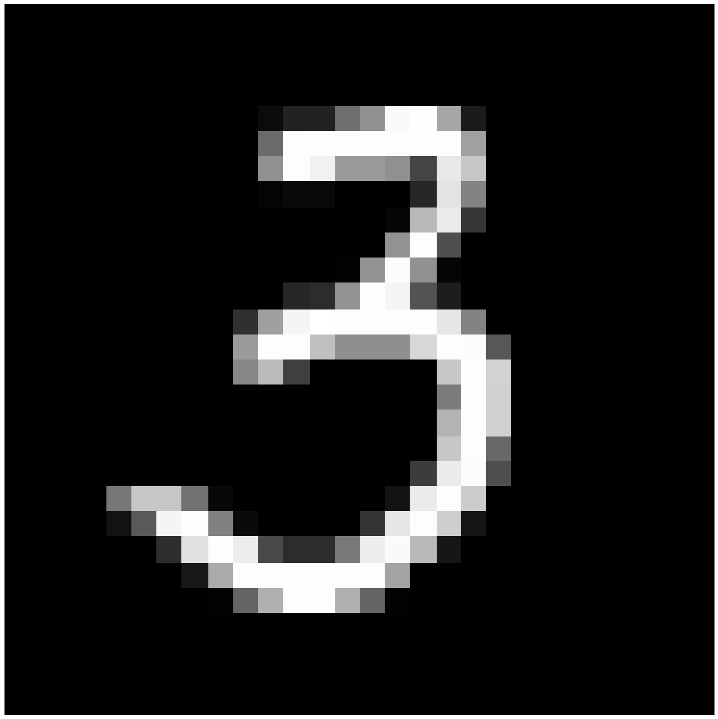
digit: 1, index: 4474



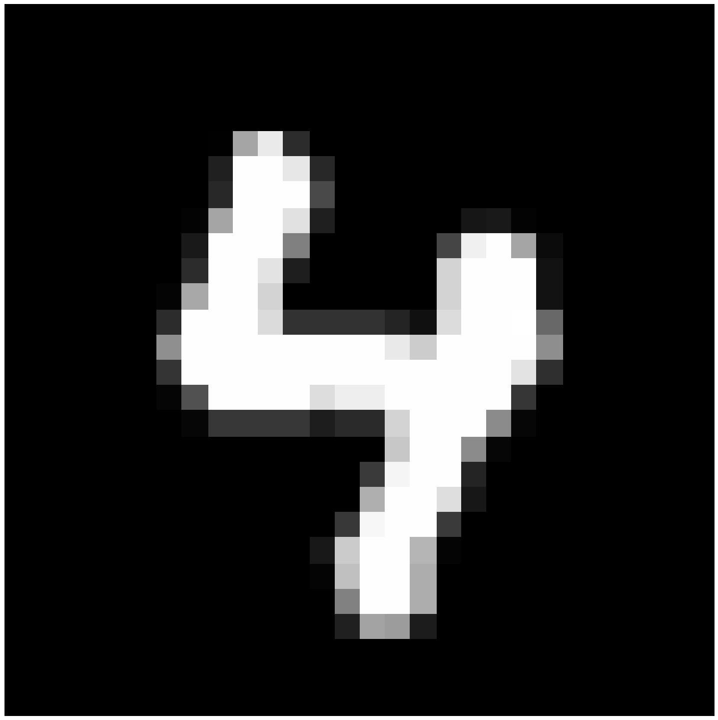
digit: 2, index: 1221



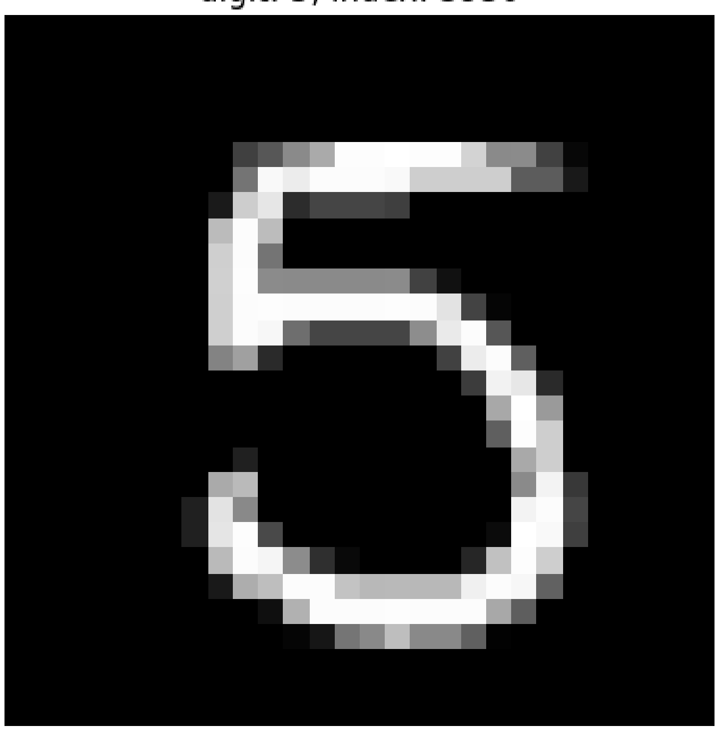
digit: 3, index: 4154



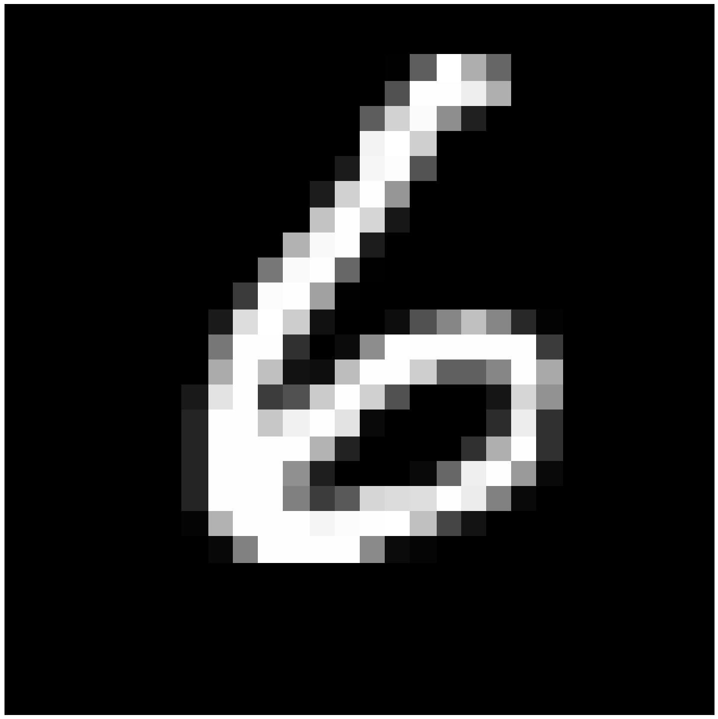
digit: 4, index: 5353



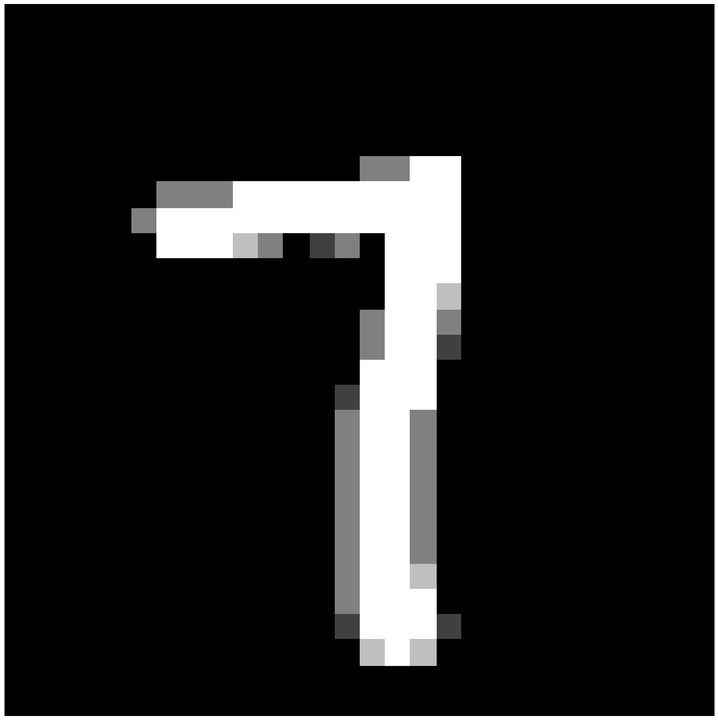
digit: 5, index: 3930



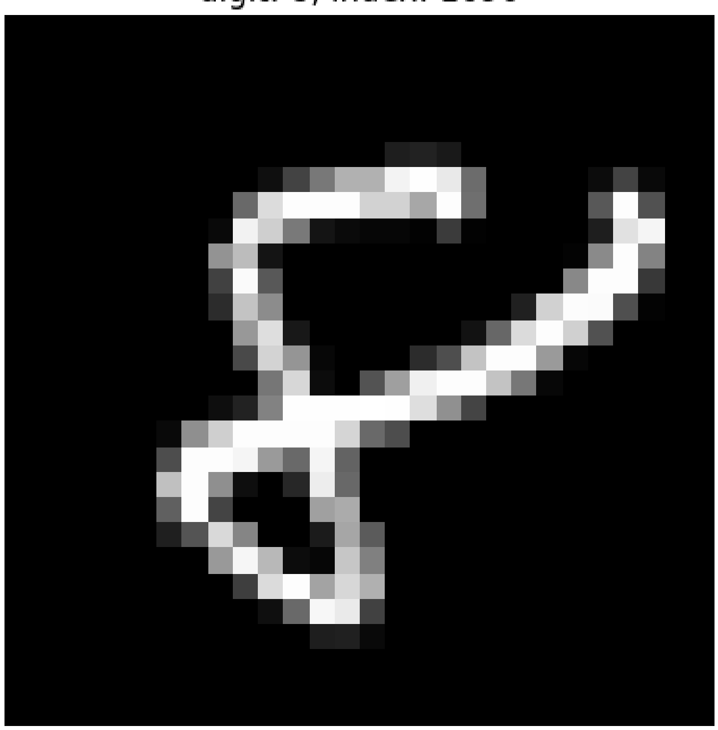
digit: 6, index: 5085



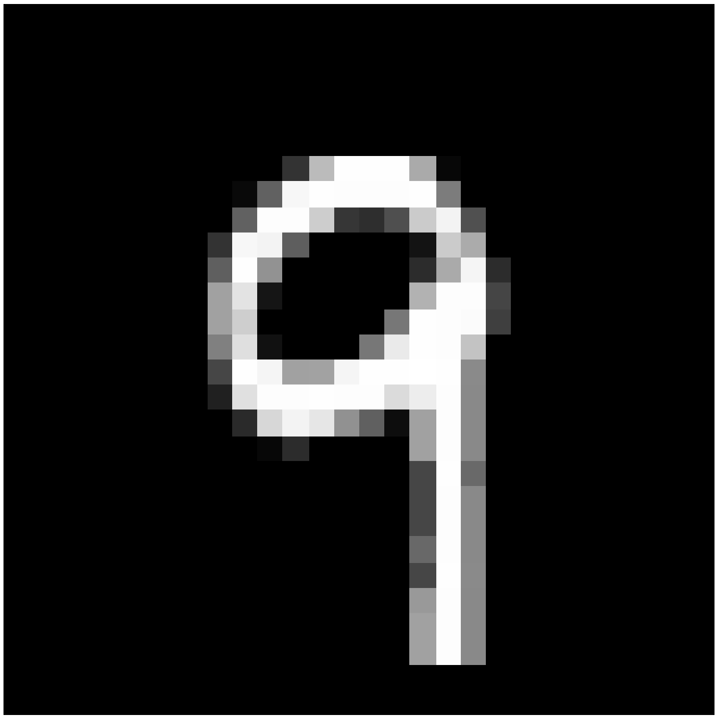
digit: 7, index: 1177



digit: 8, index: 1696



digit: 9, index: 3603



4 (4b) Label data

Select the first 500 examples of 0's and 1's for this example, those will form the training data $(x_i, y_i) \in \mathbb{R}^{784} \times \{-1, 1\}, i = 1, ..., 1000$. Assign label $y_i = 1$ for 1s and $y_i = -1$ for 0s. Also, renormalize your x_i so that the pixel values are floats between 0 and 1, instead of ints from 0 to 255. You can do this by augmenting the code given above for stacking data from different classes.

[11]:
```
# create dataset here (essentially just create a numpy array of 1's and -1's
↪for the labels)

#zeros = digits[0][:500]
#ones = digits[1][:500]
nbr_examples = 500
numbers = [4,9]
map = {((-1)** (i+1)):[digit.flatten() for digit in digits[i][:nbr_examples]]
↪for i in numbers}
for key in [-1, 1]:
    x = map[key]
    for i in range(len(x)):
```

```
        x_i = (x[i] - x[i].min()) / (x[i].max() - x[i].min())
        map[key][i] = x_i
```

5 4 (a)

[36]:

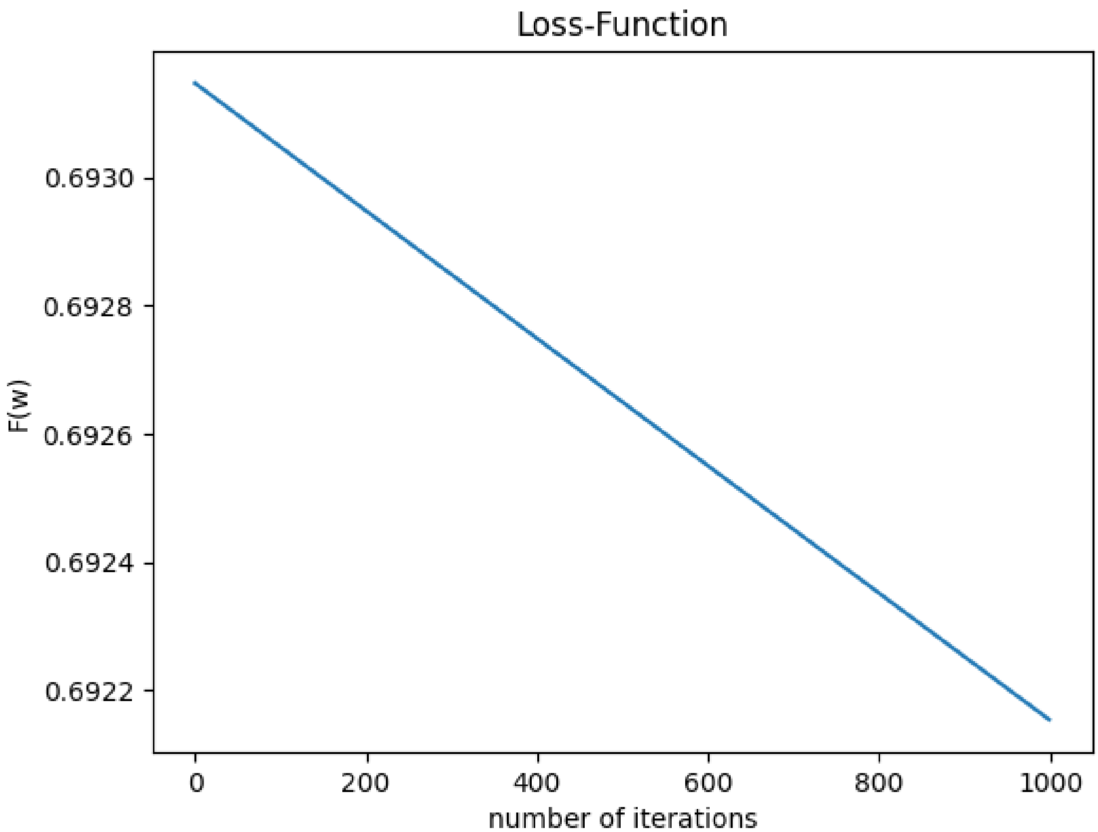
```
def function(weights:[float], map:{int:np.array}) -> float:
    sum = 0
    for i in [-1,1]:
        for x_i in map[i]:
            exponent = float(np.dot(weights, x_i)) * i * (-1)
            inner_expression = 1 + math.exp(exponent)
            sum += math.log(inner_expression)
    return sum / (len(map[-1]) + len(map[1]))


def gradient(weights:[float], map:{int:np.array}) -> np.array:
    gradient_array = np.zeros(len(weights))
    N = len(weights)
    for w_index in range(N):
        for i in [-1,1]:
            for x_i in map[i]:
                exponent = float(np.dot(weights, x_i)) * i * (-1)
                numerator = (-i) * x_i[w_index] * math.exp(exponent)
                denominator = 1 + math.exp(exponent)
                gradient_array[w_index] += numerator / denominator
        gradient_array[w_index] /= N
    return gradient_array

def optimized_gradient(weights:[float], map:{int:np.array}) -> np.array:
    gradient_array = np.zeros(len(weights))
    N = len(weights)
    for i in [-1,1]:
        for x_i in map[i]:
            exponent = float(np.dot(weights, x_i)) * i * (-1)
            numerator = (-i) * math.exp(exponent)
            denominator = 1 + math.exp(exponent)
            for w_index in range(N):
                gradient_array[w_index] += numerator * x_i[w_index] /
 ↪denominator
    return np.array([grad / N for grad in gradient_array])

def printf(weights:[float], map:{int:np.array}, i:int):
    if (i % 1000) == 0:
        print(f"iteration : {i}, F(w) = {function(weights, map)}")
    #else:
     #   pass
```

```python
        #print(f"iteration : {i}, F(w) = {function(weights, map)}")
def get_P(weights:[float], map:{int:np.array}):
    fw = optimized_gradient(weights, map)
    return np.sign(fw) * np.sum(np.abs(fw))

def get_P1(weights: np.array, map: {int: np.array}, count_vector):
    fw = optimized_gradient(weights, map)
    j_star = np.argmax(np.abs(fw))
    if count_vector is not None:
        count_vector[j_star] += 1
    p_t = np.zeros_like(fw)
    p_t[j_star] = np.sign(fw[j_star]) * np.max(np.abs(fw))

    return p_t
```

[26]:

```python
# 4 a
import math
lr = 1e-8
iteration_nbr = 10**3

weights = np.array([0 for _ in range(784)], dtype=np.float64)

x_values = []
y_values = []

for n in range(iteration_nbr):
    printf(weights, map, n)
    x_values.append(n)
    y_values.append(function(weights, map))
    weights -= lr * get_P(weights, map)
printf(weights, map, 1000)

plt.plot(x_values, y_values)
plt.xlabel('number of iterations')
plt.ylabel('F(w)')
plt.title('Loss-Function')
#plt.xticks(range(1, iteration_nbr + 1, 500))
plt.savefig('F(w).png', format='png', dpi=300)
plt.show()
```

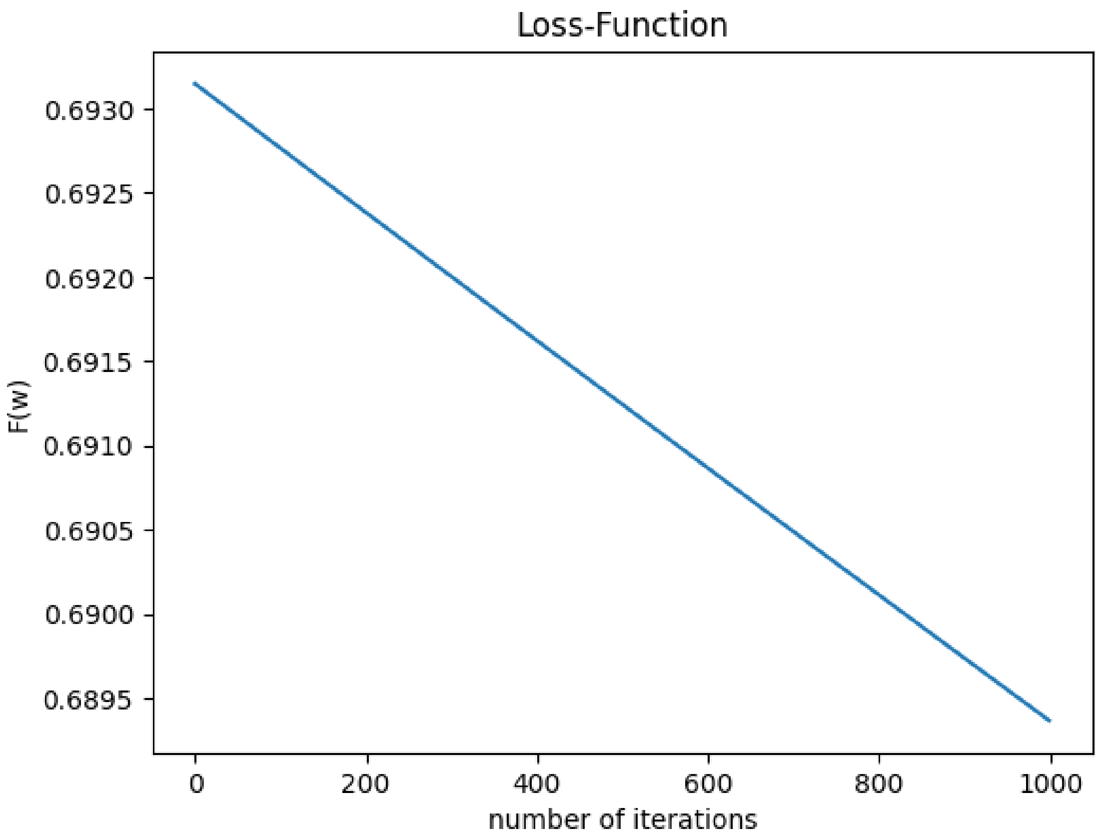
```
iteration : 0, F(w) = 0.6931471805599322
iteration : 1000, F(w) = 0.6921531799614858
```



[28]:
```
# implement gradient descent here
import math
lr = 1e-4
iteration_nbr = 10**3

weights = np.array([0 for _ in range(784)], dtype=np.float64)

x_values = []
y_values = []

for n in range(iteration_nbr):
    printf(weights, map, n)
    x_values.append(n)
    y_values.append(function(weights, map))
    weights -= lr * get_P1(weights, map,None)
printf(weights, map, 1000)

plt.plot(x_values, y_values)
plt.xlabel('number of iterations')
plt.ylabel('F(w)')
```

```
plt.title('Loss-Function')
#plt.xticks(range(1, iteration_nbr + 1, 500))
plt.savefig('F(w).png', format='png', dpi=300)
plt.show()
```

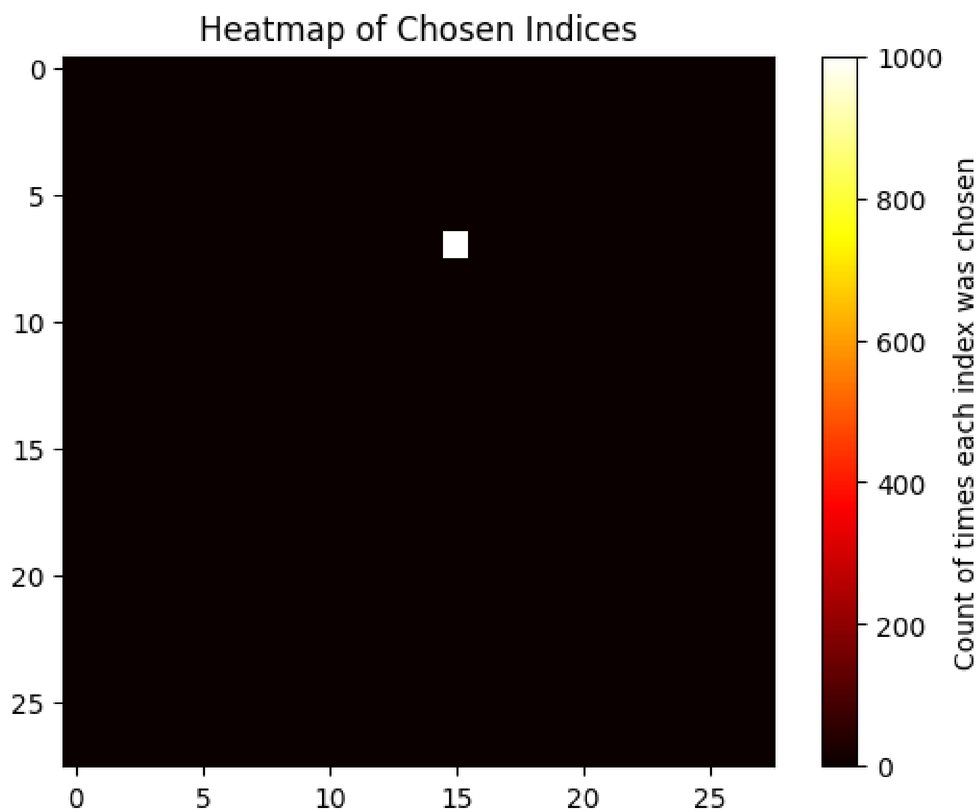
```
iteration : 0, F(w) = 0.6931471805599322
iteration : 1000, F(w) = 0.6893617393131152
```



[37]:
```
# implement gradient descent here
import math
lr = 1e-8
iteration_nbr = 10**3

weights = np.array([0 for _ in range(784)], dtype=np.float64)
count_vector = np.zeros(784, dtype=np.int32)
x_values = []
y_values = []

for n in range(iteration_nbr):
    printf(weights, map, n)
```

```
    x_values.append(n)
    y_values.append(function(weights, map))
    weights -= lr * get_P1(weights, map,count_vector)
printf(weights, map, 1000)

count_matrix = count_vector.reshape(28, 28)
plt.imshow(count_matrix, cmap='hot', interpolation='nearest')
plt.colorbar(label="Count of times each index was chosen")
plt.title("Heatmap of Chosen Indices")
plt.savefig('heatmap.png', format='png', dpi=300)
plt.show()
```

```
iteration : 0, F(w) = 0.6931471805599322
iteration : 1000, F(w) = 0.6931467970697871
```



6 (4c)

the GD from hw3.5 performs best. I think it is because the step size and number of iterations is optimized the best in that GD.

I do think the performance would change with different stepsizes. A big size can mean faster convergence, but might also mean that it is harder to find the exact minimizer

7 (4d)

The dot coordinate, when converted to an index, points to the specific entry in the weight vector that, if modified, would result in the greatest reduction in loss. This particular weight entry is consistently paired with the same feature in the input vector, specifically at positions corresponding to pixels that differ most between the two classes (e.g., pixels in the digits “4” and “9” that typically appear bright in one class and dark in the other). This suggests that these pixels are the key distinguishing features between the classes.