Assignment 1. Music Century Classification

Assignment Responsible: Natalie Lang.

In this assignment, we will build models to predict which **century** a piece of music was released. We will be using the "YearPredictionMSD Data Set" based on the Million Song Dataset. The data is available to download from the UCI Machine Learning Repository. Here are some links about the data:

- https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd
- http://millionsongdataset.com/pages/tasks-demos/#yearrecognition

Note that you are note allowed to import additional packages **(especially not PyTorch)**. One of the objectives is to understand how the training procedure actually operates, before working with PyTorch's autograd engine which does it all for us.

Question 1. Data (21%)

Start by setting up a Google Colab notebook in which to do your work. Since you are working with a partner, you might find this link helpful:

- https://colab.research.google.com/github/googlecolab/colabtools/blob/master/notebooks/colab-github-demo.ipynb

The recommended way to work together is pair coding, where you and your partner are sitting together and writing code together.

To process and read the data, we use the popular `pandas` package for data analysis.

```
import pandas
import numpy as np
import matplotlib.pyplot as plt
```

Now that your notebook is set up, we can load the data into the notebook. The code below provides two ways of loading the data: directly from the internet, or through mounting Google Drive. The first method is easier but slower, and the second method is a bit involved at first, but can save you time later on. You will need to mount Google Drive for later assignments, so we recommend figuring how to do that now.

Here are some resources to help you get started:

- http.://colab.research.google.com/notebooks/io.ipynb

```
load_from_drive = False
```

```
if not load_from_drive:
  csv_path = "http://archive.ics.uci.edu/ml/machine-learning-databases/00203/YearPredictio
else:
  from google.colab import drive
  drive.mount('/content/gdrive')
  csv_path = '/content/gdrive/My Drive/YearPredictionMSD.txt.zip' # TODO - UPDATE ME WITH

t_label = ["year"]
x_labels = ["var%d" % i for i in range(1, 91)]
df = pandas.read_csv(csv_path, names=t_label + x_labels)
```

Now that the data is loaded to your Colab notebook, you should be able to display the Pandas DataFrame `df` as a table:

```
df
```

	year	var1	var2	var3	var4	var5	var6	var7
0	2001	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202
1	2001	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630
2	2001	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872
3	2001	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097
4	2001	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207
...	...	...	...	...	...	...	...	...
515340	2006	51.28467	45.88068	22.19582	-5.53319	-3.61835	-16.36914	2.12652
515341	2006	49.87870	37.93125	18.65987	-3.63581	-27.75665	-18.52988	7.76108
515342	2006	45.12852	12.65758	-38.72018	8.80882	-29.29985	-2.28706	-18.40424
515343	2006	44.16614	32.38368	-3.34971	-2.49165	-19.59278	-18.67098	8.78428
515344	2005	51.85726	59.11655	26.39436	-5.46030	-20.69012	-19.95528	-6.72771

515345 rows × 91 columns

To set up our data for classification, we'll use the "year" field to represent whether a song was released in the 20-th century. In our case `df["year"]` will be 1 if the year was released after 2000, and 0 otherwise.

```
df["year"] = df["year"].map(lambda x: int(x > 2000))
```

```
df.head(20)
```

	year	var1	var2	var3	var4	var5	var6	var7	
0	1	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12
1	1	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0
2	1	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2
3	1	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10
4	1	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9
5	1	50.54767	0.31568	92.35066	22.38696	-25.51870	-19.04928	20.67345	-5
6	1	50.57546	33.17843	50.53517	11.55217	-27.24764	-8.78206	-12.04282	-9
7	1	48.26892	8.97526	75.23158	24.04945	-16.02105	-14.09491	8.11871	-1
8	1	49.75468	33.99581	56.73846	2.89581	-2.92429	-26.44413	1.71392	-0
9	1	45.17809	46.34234	-40.65357	-2.47909	1.21253	-0.65302	-6.95536	-12
10	1	39.13076	-23.01763	-36.20583	1.67519	-4.27101	13.01158	8.05718	-8
11	1	37.66498	-34.05910	-17.36060	-26.77781	-39.95119	-20.75000	-0.10231	-0
12	1	26.51957	-148.15762	-13.30095	-7.25851	17.22029	-21.99439	5.51947	3
13	1	37.68491	-26.84185	-27.10566	-14.95883	-5.87200	-21.68979	4.87374	-18
14	0	39.11695	-8.29767	-51.37966	-4.42668	-30.06506	-11.95916	-0.85322	-8
15	1	35.05129	-67.97714	-14.20239	-6.68696	-0.61230	-18.70341	-1.31928	-9
16	1	33.63129	-96.14912	-89.38216	-12.11699	13.77252	-6.69377	-33.36843	-24
17	0	41.38639	-20.78665	51.80155	17.21415	-36.44189	-11.53169	11.75252	-7
18	0	37.45034	11.42615	56.28982	19.58426	-16.43530	2.22457	1.02668	-7
19	0	39.71092	-4.92800	12.88590	-11.87773	2.48031	-16.11028	-16.40421	-8

20 rows × 91 columns

Part (a) -- 7%

The data set description text asks us to respect the below train/test split to avoid the "producer effect". That is, we want to make sure that no song from a single artist ends up in both the training and test set.

Explain why it would be problematic to have some songs from an artist in the training set, and other songs from the same artist in the test set. (Hint: Remember that we want our test accuracy to predict how well the model will perform in practice on a song it hasn't learned about.)

```
df_train = df[:463715]
df_test = df[463715:]

# convert to numpy
train_xs = df_train[x_labels].to_numpy()
train_ts = df_train[t_label].to_numpy()
test_xs = df_test[x_labels].to_numpy()
test_ts = df_test[t_label].to_numpy()

# Write your explanation here

# When we train the data with a specific artist, and then we use songs from the same artis
# Our test set should validate the model, and show it's accuracy - and using the same arsi
# Specificly, we don't want to overfit the data accurding to this artist.
```

Part (b) -- 7%

It can be beneficial to **normalize** the columns, so that each column (feature) has the *same* mean and standard deviation.

```
feature_means = df_train.mean()[1:].to_numpy() # the [1:] removes the mean of the "year" f
feature_stds  = df_train.std()[1:].to_numpy()

train_norm_xs = (train_xs - feature_means) / feature_stds
test_norm_xs = (test_xs - feature_means) / feature_stds
```

link textNotice how in our code, we normalized the test set using the *training data means and standard deviations*. This is *not* a bug.

Explain why it would be improper to compute and use test set means and standard deviations. (Hint: Remember what we want to use the test accuracy to measure.)

```
# Write your explanation here

# In theory, both training and test sets should be large enough so that the probability of
# will converge to a normal probability due to the law of large numbers.
# In practice, the validation set is much larger than the test set,
# so we assume the probability factors of the training set are much closer to ideal.

# Hence, if we assume the training and test sets should converge to the same normal variab
# we should use the means and standard deviation calculating using the training set to nor
```

Part (c) -- 7%

Finally, we'll move some of the data in our training set into a validation set.

Explain why we should limit how many times we use the test set, and that we should use the validation set during the model building process.

```
# shuffle the training set
reindex = np.random.permutation(len(train_xs))
train_xs = train_xs[reindex]
train_norm_xs = train_norm_xs[reindex]
train_ts = train_ts[reindex]

# use the first 50000 elements of `train_xs` as the validation set
train_xs, val_xs           = train_xs[50000:], train_xs[:50000]
train_norm_xs, val_norm_xs = train_norm_xs[50000:], train_norm_xs[:50000]
train_ts, val_ts           = train_ts[50000:], train_ts[:50000]

# Write your explanation here

# We should use validation data to evaluate the performences of our model, and to adjust i
# Without validation, the model that was built using only the training set is much more pr
# To avoid that situation we devide the overall set of samples to three seperate data sets
# We test our NN (with the test set) only after finishing the training and validation step
# If we use the test set too many times, it will be learned by the model, without being va
# causing the overal model to be overfitted to the test set data.
```

Part 2. Classification (79%)

We will first build a *classification* model to perform decade classification. These helper functions are written for you. All other code that you write in this section should be vectorized whenever possible (i.e., avoid unnecessary loops).

```
def sigmoid(z):
  return 1 / (1 + np.exp(-z))

def cross_entropy(t, y):
  e = 10e-20  # stabling factor - so we won't have log(0)
  return -t * np.log(y + e) - (1 - t) * np.log(1 - y + e)

def cost(y, t):
  return np.mean(cross_entropy(t, y))

def get_accuracy(y, t):
  acc = 0
  N = 0
  for i in range(len(y)):
    N += 1
    if (y[i] >= 0.5 and t[i] == 1) or (y[i] < 0.5 and t[i] == 0):
      acc += 1
  return acc / N
```

Part (a) -- 7%

Write a function `pred` that computes the prediction `y` based on logistic regression, i.e., a single layer with weights `w` and bias `b`. The output is given by:

$$y = \sigma(\mathbf{w}^T \mathbf{x} + b),$$

where the value of y is an estimate of the probability that the song is released in the current century, namely $\text{year} = 1$.

```
def pred(w, b, X):
  """
  Returns the prediction `y` of the target based on the weights `w` and scalar bias `b`.

  Preconditions: np.shape(w) == (90,)
                 type(b) == float
                 np.shape(X) = (N, 90) for some N

  >>> pred(np.zeros(90), 1, np.ones([2, 90]))
  array([0.73105858, 0.73105858]) # It's okay if your output differs in the last decimals
  """
  # Your code goes here
  X_T = X.transpose();
  z = np.dot(w, X_T) + b;
  return sigmoid(z);

pred(np.zeros(90), 1, np.ones([2, 90]))

     array([0.73105858, 0.73105858])
```

Part (b) -- 7%

Write a function `derivative_cost` that computes and returns the gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ and $\frac{\partial \mathcal{L}}{\partial b}$. Here, `X` is the input, `y` is the prediction, and `t` is the true label.

```
def derivative_cost(X, y, t):
  """
  Returns a tuple containing the gradients dLdw and dLdb.

  Precondition: np.shape(X) == (N, 90) for some N
                np.shape(y) == (N,)
                np.shape(t) == (N,)

  Postcondition: np.shape(dLdw) = (90,)
           type(dLdb) = float
  """
  # Your code goes here
  x = X.transpose();
  N = len(X)
  dw = 1/N*(np.matmul(np.transpose(X), y-t));
  db = 1/N*(np.sum(y-t));
  return(dw, db);
```

Explenation on Gradients

Add here an explaination on how the gradients are computed:

The cross entropy: $-t\log(y)-(1-t)\log(1-y)$

remember: $y=\sigma(z)=\frac{1}{1-e^{-z}}$

and: $z^i=(\mathbf{w}^T\mathbf{x^i}+b)$

so: $\frac{dy}{dw_i}=\sigma(\mathbf{w}^T\mathbf{x^i}+b)(1-\sigma(\mathbf{w}^T\mathbf{x^i}+b))x_i=y(1-y)x_i$

and: $\frac{dy}{db}=\sigma(\mathbf{w}^T\mathbf{x^i}+b)(1-\sigma(\mathbf{w}^T\mathbf{x^i}+b))=y(1-y)$

$\mathcal{L}$ is defined as the mean of the cross entropy:
$\mathcal{L}=\frac{1}{N}\sum_{i=1}^{N}(-t\log(y)-(1-t)\log(1-(y)))$

so $\frac{\partial l}{\partial y}=\frac{-t}{y}+\frac{1-t}{1-y}$

We'll find the derivative using partial derivatives:

$\triangledown_w\mathcal{L}=\frac{1}{N}\sum_{i=1}^{N}\triangledown_w l=\frac{1}{N}\sum_{i=1}^{N}\frac{\partial l}{\partial w}=\frac{1}{N}\sum_{i=1}^{N}\frac{\partial l}{\partial y}\frac{\partial y}{\partial w}=\frac{1}{N}X^T(\frac{-t}{y}+\frac{1-t}{1-y})y(1-y)=$

$\triangledown_b\mathcal{L}=\frac{1}{N}\sum_{n=1}^{N}\triangledown_b l=\frac{1}{N}\sum_{n=1}^{N}\frac{\partial l}{\partial b}=\frac{1}{N}\sum_{n=1}^{N}\frac{\partial l}{\partial y}\frac{\partial y}{\partial b}=\frac{1}{N}\sum_{n=1}^{N}(\frac{-t}{y}+\frac{1-t}{1-y})y(1-y)=$

Part (c) -- 7%

We can check that our derivative is implemented correctly using the finite difference rule. In 1D, the finite difference rule tells us that for small h, we should have

$$\frac{f(x+h)-f(x)}{h}\approx f'(x)$$

Show that $\frac{\partial\mathcal{L}}{\partial b}$ is implement correctly by comparing the result from `derivative_cost` with the empirical cost derivative computed using the above numerical approximation.

```
# Your code goes here
X=np.array([2])
t=np.array([4])
w=np.array([5])
b=np.array([5])
h = 10e-4

y=pred(w, b, X)
y_der=pred(w, b+h, X)

dw, db = derivative_cost(X, y, t)

r1 = db
```

```
r2 = ((cross_entropy(t, y_der) - cross_entropy(t, y))/h)[0]
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
```

```
     The analytical results is - -3.000000305902227
     The algorithm results is -  -3.0000000143601824
```

Part (d) -- 7%

Show that $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ is implement correctly.

```
# Your code goes here. You might find this below code helpful: but it's
# up to you to figure out how/why, and how to modify the code

X=np.array([2])
t=np.array([4])
w=np.array([5])
b=np.array([5])
h = 10e-4

y=pred(w, b, X)
y_der=pred(w+h, b, X)

dw, db = derivative_cost(X, y, t)

r1 = dw
r2 = ((cross_entropy(t, y_der) - cross_entropy(t, y))/h)[0]
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
```

```
     The analytical results is - -6.000000611804454
     The algorithm results is -  -6.0000006454075105
```

Part (e) -- 7%

Now that you have a gradient function that works, we can actually run gradient descent. Complete the following code that will run stochastic: gradient descent training:

```
def run_gradient_descent(w0, b0, mu=0.1, batch_size=100, max_iters=100):
  """Return the values of (w, b) after running gradient descent for max_iters.
  We use:
    - train_norm_xs and train_ts as the training set
    - val_norm_xs and val_ts as the test set
    - mu as the learning rate
    - (w0, b0) as the initial values of (w, b)

  Precondition: np.shape(w0) == (90,)
                type(b0) == float
  Postcondition: np.shape(w) == (90,)
                 type(b) == float
  """
  global train_ts, val_ts, train_norm_xs, val_norm_xs
  w = w0
  b = b0
  iter = 0

  all_costs = []
  all_acc = []

  # y_val = pred(w, b, val_norm_xs)

  while iter < max_iters:
    # shuffle the training set (there is code above for how to do this)
    reindex = np.random.permutation(len(train_norm_xs))
    train_norm_xs = train_norm_xs[reindex]
    train_ts = train_ts[reindex]

    for i in range(0, len(train_norm_xs), batch_size): # iterate over each minibatch
      # minibatch that we are working with:
      X = train_norm_xs[i:(i + batch_size)]
      t = train_ts[i:(i + batch_size), 0]

      # since len(train_norm_xs) does not divide batch_size evenly, we will skip over
      # the "last" minibatch
      if np.shape(X)[0] != batch_size:
        continue

      # compute the prediction
      y_train = pred(w, b, X)

      # update w and b
      dw, db = derivative_cost(X, y_train, t)
      w = w - mu*dw
      b = b - mu*db

      # increment the iteration count
      iter += 1

      # compute and print the *validation* loss and accuracy
      if (iter % 10 == 0):
        val_cost = 0
        val_acc = 0
        y_val = pred(w, b, val_norm_xs)

        for i_y, i_t in zip(y_val, map(lambda x: x[0], val_ts)):
          cost_i = cost(i_y, i_t)
          acc_i = get_accuracy([i_y], [i_t])
          val_cost += cost_i/len(val_ts)
          val_acc += acc_i/len(val_ts)
        all_costs += [val_cost]
        all_acc += [val_acc]
        print("Iter %d. [Val Acc %.0f%%, Loss %f]" % (
                iter, val_acc * 100, val_cost))

      # if iter >= max_iters:
      #   x_axis = np.arange(10, max_iters + 10, 10)
      #   costs = plt.plot(x_axis, all_costs, "-", label="loss val")
      #   plt.legend()
      #   plt.title(f'Validation set Loss vs Iteration mu={mu}')
      #   plt.xlabel('Iteration')
      #   plt.ylabel('Loss')
      #   plt.show()
      #   break
      if iter >= max_iters:
        break

      # Think what parameters you should return for further use

  return w, b, all_costs
```

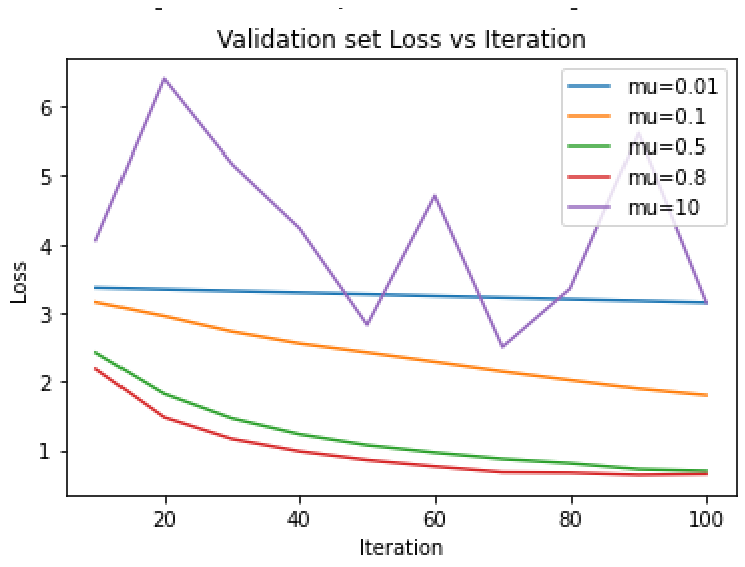
Part (f) -- 7%

Call `run_gradient_descent` with the weights and biases all initialized to zero. Show that if the learning rate μ is too small, then convergence is slow. Also, show that if μ is too large, then the optimization algorirthm does not converge. The demonstration should be made using plots showing these effects.

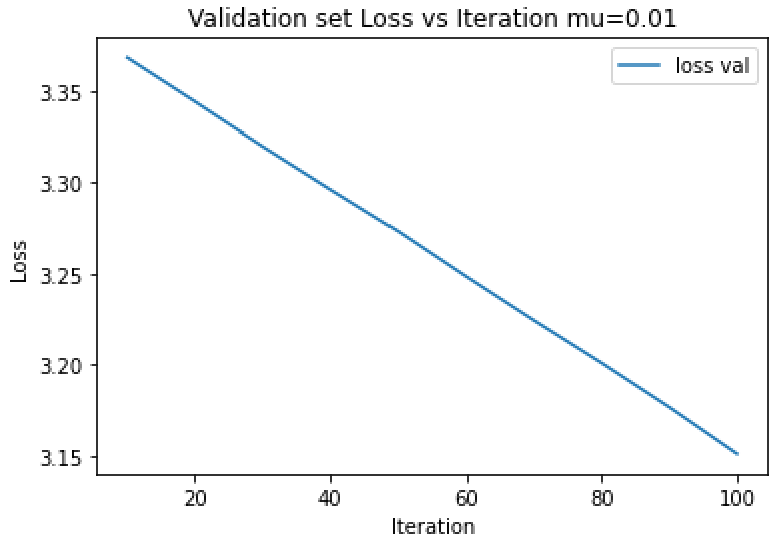
```
w0 = np.random.randn(90)
b0 = np.random.randn(1)[0]

# Write your code here
mu_arr= [0.01, 0.1, 0.5, 0.8, 10]
max_iters = 100
costs = [] # all costs from each miu will be stored here
for mu in mu_arr:
  print(f"\n Those are the cost and accuracy results for mu = {mu}:")
  w, b, all_costs = run_gradient_descent(w0,b0, mu, max_iters=max_iters)
  costs += [all_costs]

# plot the results to analyze which mu is the stable
x_axis = np.arange(10, max_iters + 10, 10)
for i in range(len(mu_arr)):
  plt.plot(x_axis, costs[i], "-", label=f"mu={mu_arr[i]}")
  plt.legend()
  plt.title(f"Validation set Loss vs Iteration")

plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()

print("Zoom in for the mu that is too small:")
```

```
plt.plot(x_axis, costs[0], "-", label="loss val")
plt.legend()
plt.title(f'Validation set Loss vs Iteration mu={mu_arr[0]}')

plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()

print("Zoom in for the mu that is too big:")

plt.plot(x_axis, costs[-1], "-", label="loss val")
plt.legend()
plt.title(f'Validation set Loss vs Iteration mu={mu_arr[-1]}')

plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()


print("Zoom in for the mu that is stable:")
for i in range(len(mu_arr)-3):
  plt.plot(x_axis, costs[i+2], "-", label=f"mu={mu_arr[i+2]}")
  plt.legend()
  plt.title(f"Validation set Loss vs Iteration")

plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```

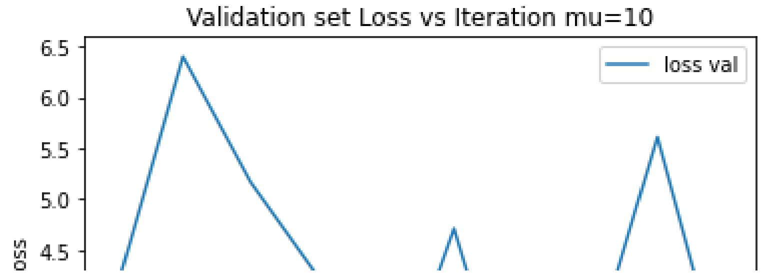
```
 Those are the cost and accuracy results for mu = 0.01:
Iter 10. [Val Acc 50%, Loss 3.368057]
Iter 20. [Val Acc 51%, Loss 3.344022]
Iter 30. [Val Acc 51%, Loss 3.319544]
Iter 40. [Val Acc 51%, Loss 3.296034]
Iter 50. [Val Acc 51%, Loss 3.272965]
Iter 60. [Val Acc 51%, Loss 3.248202]
Iter 70. [Val Acc 51%, Loss 3.224116]
Iter 80. [Val Acc 51%, Loss 3.200745]
Iter 90. [Val Acc 51%, Loss 3.176491]
Iter 100. [Val Acc 51%, Loss 3.150915]

 Those are the cost and accuracy results for mu = 0.1:
Iter 10. [Val Acc 51%, Loss 3.156330]
Iter 20. [Val Acc 52%, Loss 2.955304]
Iter 30. [Val Acc 52%, Loss 2.730533]
Iter 40. [Val Acc 53%, Loss 2.556268]
Iter 50. [Val Acc 53%, Loss 2.424059]
Iter 60. [Val Acc 54%, Loss 2.287857]
Iter 70. [Val Acc 55%, Loss 2.148989]
Iter 80. [Val Acc 55%, Loss 2.022494]
Iter 90. [Val Acc 56%, Loss 1.898655]
Iter 100. [Val Acc 57%, Loss 1.803417]

 Those are the cost and accuracy results for mu = 0.5:
Iter 10. [Val Acc 53%, Loss 2.420582]
Iter 20. [Val Acc 57%, Loss 1.824847]
Iter 30. [Val Acc 60%, Loss 1.465780]
Iter 40. [Val Acc 61%, Loss 1.232585]
Iter 50. [Val Acc 63%, Loss 1.073511]
Iter 60. [Val Acc 65%, Loss 0.964664]
Iter 70. [Val Acc 66%, Loss 0.872891]
Iter 80. [Val Acc 66%, Loss 0.811012]
Iter 90. [Val Acc 67%, Loss 0.724872]
Iter 100. [Val Acc 68%, Loss 0.698109]

 Those are the cost and accuracy results for mu = 0.8:
Iter 10. [Val Acc 55%, Loss 2.185978]
Iter 20. [Val Acc 60%, Loss 1.480366]
Iter 30. [Val Acc 62%, Loss 1.165243]
Iter 40. [Val Acc 65%, Loss 0.983356]
Iter 50. [Val Acc 67%, Loss 0.857446]
Iter 60. [Val Acc 68%, Loss 0.764138]
Iter 70. [Val Acc 69%, Loss 0.681682]
Iter 80. [Val Acc 69%, Loss 0.673806]
Iter 90. [Val Acc 69%, Loss 0.641854]
Iter 100. [Val Acc 69%, Loss 0.655101]

 Those are the cost and accuracy results for mu = 10:
Iter 10. [Val Acc 60%, Loss 4.059306]
Iter 20. [Val Acc 55%, Loss 6.402479]
Iter 30. [Val Acc 60%, Loss 5.167423]
Iter 40. [Val Acc 66%, Loss 4.233577]
Iter 50. [Val Acc 65%, Loss 2.827117]
Iter 60. [Val Acc 59%, Loss 4.710502]
Iter 70. [Val Acc 66%, Loss 2.505232]
Iter 80. [Val Acc 64%, Loss 3.355278]
Iter 90. [Val Acc 61%, Loss 5.611240]
Iter 100. [Val Acc 67%, Loss 3.151018]
```



```
Zoom in for the mu that is too small:
```



```
Zoom in for the mu that is too big:
```



Explain and discuss your results here:

Part (g) -- 7%

Find the optimial value of $\mathbf{w}$ and b using your code. Explain how you chose the learning rate μ and the batch size. Show plots demostrating good and bad behaviours.

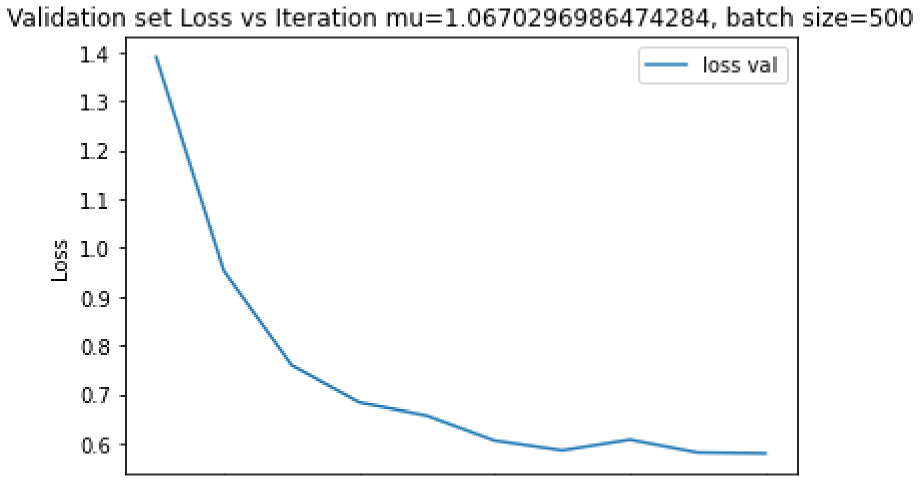
```
w0 = np.random.randn(90)
b0 = np.random.randn(1)[0]

# Write your code here

# find the best mu for gradient descent by minimizing cost function on last iteration
```

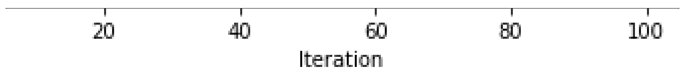
```
mu_arr = np.random.rand(5)+0.2
mu_arr.sort()
batch_arr = [20, 50, 100, 300, 500]
min_cost = 10000

for mu in mu_arr:
  for batch in batch_arr:
    w, b, all_cost = run_gradient_descent(w0, b0, mu, batch_size=batch)
    if all_cost[-1] < min_cost:  # if we got better results, update the values to the opti
      min_cost = all_cost[-1]
      mu_opt = mu
      batch_opt = batch


w_opt, b_opt, cost_arr = run_gradient_descent(w0, b0, mu_opt, batch_size=batch_opt)

print(f"The optimal w is: {w_opt} \n")
print(f"The optimal b is: {b_opt} \n")

print("Desired behavior, for the optimal w and b for good mu and batch size:")

plt.plot(x_axis, cost_arr, "-", label="loss val")
plt.legend()
plt.title(f'Validation set Loss vs Iteration mu={mu_opt}, batch size={batch_opt}')

plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()

w_opt_bad, b_opt_bad, cost_arr = run_gradient_descent(w0, b0, mu=10, batch_size=20)

print("Bad behavior, for the optimal w and b for good mu and batch size:")

plt.plot(x_axis, cost_arr, "-", label="loss val")
plt.legend()
plt.title(f'Validation set Loss vs Iteration mu=10, batch size=20')

plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```

```
Iter 10. [Val Acc 55%, Loss 2.190687]
Iter 20. [Val Acc 56%, Loss 1.946122]
Iter 30. [Val Acc 58%, Loss 1.676955]
Iter 40. [Val Acc 59%, Loss 1.456433]
Iter 50. [Val Acc 59%, Loss 1.322140]
Iter 60. [Val Acc 62%, Loss 1.229570]
Iter 70. [Val Acc 62%, Loss 1.097423]
Iter 80. [Val Acc 64%, Loss 1.057269]
Iter 90. [Val Acc 63%, Loss 1.066296]
Iter 100. [Val Acc 65%, Loss 0.932937]
Iter 10. [Val Acc 54%, Loss 2.059753]
Iter 20. [Val Acc 56%, Loss 1.711361]
Iter 30. [Val Acc 58%, Loss 1.480843]
Iter 40. [Val Acc 61%, Loss 1.245441]
Iter 50. [Val Acc 62%, Loss 1.122445]
Iter 60. [Val Acc 63%, Loss 1.041616]
Iter 70. [Val Acc 64%, Loss 0.952251]
Iter 80. [Val Acc 65%, Loss 0.904571]
Iter 90. [Val Acc 67%, Loss 0.851825]
Iter 100. [Val Acc 67%, Loss 0.806309]
Iter 10. [Val Acc 54%, Loss 2.028004]
Iter 20. [Val Acc 57%, Loss 1.689658]
Iter 30. [Val Acc 59%, Loss 1.441447]
Iter 40. [Val Acc 60%, Loss 1.270769]
Iter 50. [Val Acc 62%, Loss 1.111770]
Iter 60. [Val Acc 63%, Loss 1.008736]
Iter 70. [Val Acc 65%, Loss 0.926803]
Iter 80. [Val Acc 66%, Loss 0.858901]
Iter 90. [Val Acc 67%, Loss 0.798159]
Iter 100. [Val Acc 67%, Loss 0.766747]
Iter 10. [Val Acc 55%, Loss 2.052207]
Iter 20. [Val Acc 57%, Loss 1.675674]
Iter 30. [Val Acc 59%, Loss 1.427194]
Iter 40. [Val Acc 61%, Loss 1.240272]
Iter 50. [Val Acc 62%, Loss 1.090084]
Iter 60. [Val Acc 64%, Loss 0.978661]
Iter 70. [Val Acc 65%, Loss 0.895249]
Iter 80. [Val Acc 67%, Loss 0.822416]
Iter 90. [Val Acc 68%, Loss 0.769157]
Iter 100. [Val Acc 68%, Loss 0.734631]
Iter 10. [Val Acc 55%, Loss 2.018112]
Iter 20. [Val Acc 57%, Loss 1.653620]
Iter 30. [Val Acc 59%, Loss 1.410354]
Iter 40. [Val Acc 61%, Loss 1.220637]
Iter 50. [Val Acc 62%, Loss 1.071695]
Iter 60. [Val Acc 64%, Loss 0.972071]
Iter 70. [Val Acc 65%, Loss 0.885778]
Iter 80. [Val Acc 66%, Loss 0.820689]
Iter 90. [Val Acc 68%, Loss 0.772576]
Iter 100. [Val Acc 68%, Loss 0.735482]
Iter 10. [Val Acc 55%, Loss 2.188355]
Iter 20. [Val Acc 57%, Loss 1.729809]
Iter 30. [Val Acc 60%, Loss 1.492567]
Iter 40. [Val Acc 61%, Loss 1.332966]
Iter 50. [Val Acc 62%, Loss 1.203149]
Iter 60. [Val Acc 64%, Loss 1.046438]
Iter 70. [Val Acc 64%, Loss 1.044050]
Iter 80. [Val Acc 65%, Loss 0.983220]
Iter 90. [Val Acc 64%, Loss 1.010144]
Iter 100. [Val Acc 64%, Loss 0.982293]
Iter 10. [Val Acc 55%, Loss 1.989316]
Iter 20. [Val Acc 57%, Loss 1.645943]
Iter 30. [Val Acc 59%, Loss 1.376986]
Iter 40. [Val Acc 61%, Loss 1.214402]
Iter 50. [Val Acc 63%, Loss 1.072099]
Iter 60. [Val Acc 63%, Loss 0.984983]
Iter 70. [Val Acc 66%, Loss 0.908662]
Iter 80. [Val Acc 66%, Loss 0.853420]
Iter 90. [Val Acc 67%, Loss 0.794742]
Iter 100. [Val Acc 68%, Loss 0.759220]
Iter 10. [Val Acc 55%, Loss 1.956448]
Iter 20. [Val Acc 58%, Loss 1.562562]
Iter 30. [Val Acc 59%, Loss 1.329391]
Iter 40. [Val Acc 62%, Loss 1.138595]
Iter 50. [Val Acc 63%, Loss 1.019568]
Iter 60. [Val Acc 66%, Loss 0.900968]
Iter 70. [Val Acc 66%, Loss 0.836896]
Iter 80. [Val Acc 68%, Loss 0.769641]
Iter 90. [Val Acc 69%, Loss 0.740946]
Iter 100. [Val Acc 69%, Loss 0.705703]
Iter 10. [Val Acc 55%, Loss 1.961765]
Iter 20. [Val Acc 58%, Loss 1.542352]
Iter 30. [Val Acc 60%, Loss 1.278996]
Iter 40. [Val Acc 62%, Loss 1.110018]
Iter 50. [Val Acc 64%, Loss 0.969878]
Iter 60. [Val Acc 66%, Loss 0.885694]
Iter 70. [Val Acc 67%, Loss 0.812014]
Iter 80. [Val Acc 68%, Loss 0.756019]
Iter 90. [Val Acc 68%, Loss 0.715209]
Iter 100. [Val Acc 69%, Loss 0.681300]
Iter 10. [Val Acc 55%, Loss 1.958995]
Iter 20. [Val Acc 58%, Loss 1.546937]
Iter 30. [Val Acc 60%, Loss 1.285630]
Iter 40. [Val Acc 62%, Loss 1.107555]
Iter 50. [Val Acc 64%, Loss 0.975485]
Iter 60. [Val Acc 66%, Loss 0.879712]
Iter 70. [Val Acc 67%, Loss 0.805439]
Iter 80. [Val Acc 68%, Loss 0.751081]
Iter 90. [Val Acc 68%, Loss 0.715399]
Iter 100. [Val Acc 69%, Loss 0.680231]
Iter 10. [Val Acc 55%, Loss 1.802145]
Iter 20. [Val Acc 59%, Loss 1.634087]
Iter 30. [Val Acc 61%, Loss 1.364995]
Iter 40. [Val Acc 64%, Loss 1.261604]
Iter 50. [Val Acc 61%, Loss 1.408433]
Iter 60. [Val Acc 63%, Loss 1.225350]
Iter 70. [Val Acc 64%, Loss 1.076092]
Iter 80. [Val Acc 66%, Loss 1.008466]
Iter 90. [Val Acc 64%, Loss 1.132607]
Iter 100. [Val Acc 65%, Loss 1.051857]
Iter 10. [Val Acc 58%, Loss 1.616824]
Iter 20. [Val Acc 59%, Loss 1.326463]
Iter 30. [Val Acc 62%, Loss 1.115628]
Iter 40. [Val Acc 65%, Loss 0.927351]
Iter 50. [Val Acc 69%, Loss 0.818858]
Iter 60. [Val Acc 68%, Loss 0.765618]
Iter 70. [Val Acc 66%, Loss 0.805501]
Iter 80. [Val Acc 69%, Loss 0.692289]
Iter 90. [Val Acc 68%, Loss 0.728446]
Iter 100. [Val Acc 68%, Loss 0.709028]
Iter 10. [Val Acc 56%, Loss 1.635211]
Iter 20. [Val Acc 62%, Loss 1.201918]
Iter 30. [Val Acc 63%, Loss 0.982892]
Iter 40. [Val Acc 67%, Loss 0.812912]
Iter 50. [Val Acc 68%, Loss 0.742387]
Iter 60. [Val Acc 68%, Loss 0.719945]
Iter 70. [Val Acc 69%, Loss 0.686169]
Iter 80. [Val Acc 70%, Loss 0.664902]
Iter 90. [Val Acc 70%, Loss 0.636678]
Iter 100. [Val Acc 66%, Loss 0.741781]
Iter 10. [Val Acc 58%, Loss 1.580851]
Iter 20. [Val Acc 62%, Loss 1.166062]
Iter 30. [Val Acc 64%, Loss 0.924932]
Iter 40. [Val Acc 67%, Loss 0.793649]
Iter 50. [Val Acc 68%, Loss 0.715004]
Iter 60. [Val Acc 70%, Loss 0.657340]
Iter 70. [Val Acc 71%, Loss 0.633181]
Iter 80. [Val Acc 71%, Loss 0.608407]
Iter 90. [Val Acc 71%, Loss 0.597457]
Iter 100. [Val Acc 72%, Loss 0.589599]
Iter 10. [Val Acc 57%, Loss 1.586397]
Iter 20. [Val Acc 61%, Loss 1.147363]
Iter 30. [Val Acc 65%, Loss 0.900287]
Iter 40. [Val Acc 67%, Loss 0.778024]
Iter 50. [Val Acc 69%, Loss 0.704272]
Iter 60. [Val Acc 70%, Loss 0.653753]
Iter 70. [Val Acc 70%, Loss 0.632869]
Iter 80. [Val Acc 72%, Loss 0.605229]
Iter 90. [Val Acc 72%, Loss 0.593931]
Iter 100. [Val Acc 72%, Loss 0.586724]
Iter 10. [Val Acc 56%, Loss 2.004390]
Iter 20. [Val Acc 61%, Loss 1.563599]
Iter 30. [Val Acc 65%, Loss 1.265027]
Iter 40. [Val Acc 66%, Loss 1.217730]
Iter 50. [Val Acc 66%, Loss 1.219920]
Iter 60. [Val Acc 67%, Loss 1.203891]
Iter 70. [Val Acc 67%, Loss 1.165311]
Iter 80. [Val Acc 67%, Loss 1.205220]
Iter 90. [Val Acc 69%, Loss 1.107670]
Iter 100. [Val Acc 66%, Loss 1.037114]
Iter 10. [Val Acc 58%, Loss 1.532576]
Iter 20. [Val Acc 64%, Loss 1.088990]
Iter 30. [Val Acc 67%, Loss 1.006833]
Iter 40. [Val Acc 67%, Loss 0.903590]
Iter 50. [Val Acc 69%, Loss 0.782728]
Iter 60. [Val Acc 67%, Loss 0.770494]
Iter 70. [Val Acc 70%, Loss 0.712854]
Iter 80. [Val Acc 62%, Loss 1.009213]
Iter 90. [Val Acc 69%, Loss 0.725946]
Iter 100. [Val Acc 70%, Loss 0.695290]
Iter 10. [Val Acc 60%, Loss 1.499267]
Iter 20. [Val Acc 64%, Loss 1.040172]
Iter 30. [Val Acc 67%, Loss 0.861426]
Iter 40. [Val Acc 69%, Loss 0.784031]
Iter 50. [Val Acc 68%, Loss 0.752629]
Iter 60. [Val Acc 68%, Loss 0.719874]
Iter 70. [Val Acc 69%, Loss 0.678217]
Iter 80. [Val Acc 68%, Loss 0.694415]
Iter 90. [Val Acc 70%, Loss 0.645744]
Iter 100. [Val Acc 70%, Loss 0.650951]
```

```
        Iter 10. [Val Acc 59%, Loss 1.386531]
        Iter 20. [Val Acc 64%, Loss 0.960876]
        Iter 30. [Val Acc 68%, Loss 0.783618]
        Iter 40. [Val Acc 69%, Loss 0.701176]
        Iter 50. [Val Acc 71%, Loss 0.643543]
        Iter 60. [Val Acc 69%, Loss 0.638857]
        Iter 70. [Val Acc 72%, Loss 0.600128]
        Iter 80. [Val Acc 70%, Loss 0.634579]
        Iter 90. [Val Acc 73%, Loss 0.578789]
        Iter 100. [Val Acc 72%, Loss 0.586537]
        Iter 10. [Val Acc 59%, Loss 1.358570]
        Iter 20. [Val Acc 65%, Loss 0.944808]
        Iter 30. [Val Acc 67%, Loss 0.770250]
        Iter 40. [Val Acc 69%, Loss 0.683118]
        Iter 50. [Val Acc 71%, Loss 0.628982]
        Iter 60. [Val Acc 72%, Loss 0.606957]
        Iter 70. [Val Acc 72%, Loss 0.599072]
        Iter 80. [Val Acc 72%, Loss 0.585897]
        Iter 90. [Val Acc 72%, Loss 0.590346]
        Iter 100. [Val Acc 73%, Loss 0.572410]
        Iter 10. [Val Acc 58%, Loss 1.809289]
        Iter 20. [Val Acc 62%, Loss 1.438647]
        Iter 30. [Val Acc 60%, Loss 1.394396]
        Iter 40. [Val Acc 64%, Loss 1.209657]
        Iter 50. [Val Acc 66%, Loss 1.177424]
        Iter 60. [Val Acc 64%, Loss 1.262970]
        Iter 70. [Val Acc 65%, Loss 1.251831]
        Iter 80. [Val Acc 67%, Loss 1.162288]
        Iter 90. [Val Acc 65%, Loss 1.183568]
        Iter 100. [Val Acc 62%, Loss 1.419244]
        Iter 10. [Val Acc 58%, Loss 1.664426]
        Iter 20. [Val Acc 61%, Loss 1.274831]
        Iter 30. [Val Acc 66%, Loss 1.013321]
        Iter 40. [Val Acc 65%, Loss 0.975667]
        Iter 50. [Val Acc 69%, Loss 0.832449]
        Iter 60. [Val Acc 67%, Loss 0.783837]
        Iter 70. [Val Acc 69%, Loss 0.734702]
        Iter 80. [Val Acc 63%, Loss 0.877282]
        Iter 90. [Val Acc 69%, Loss 0.719624]
        Iter 100. [Val Acc 59%, Loss 1.237729]
        Iter 10. [Val Acc 58%, Loss 1.542489]
        Iter 20. [Val Acc 62%, Loss 1.098141]
        Iter 30. [Val Acc 66%, Loss 0.872165]
        Iter 40. [Val Acc 68%, Loss 0.756461]
        Iter 50. [Val Acc 70%, Loss 0.687719]
        Iter 60. [Val Acc 67%, Loss 0.721336]
        Iter 70. [Val Acc 68%, Loss 0.651107]
        Iter 80. [Val Acc 69%, Loss 0.646593]
        Iter 90. [Val Acc 70%, Loss 0.646031]
        Iter 100. [Val Acc 70%, Loss 0.635115]
        Iter 10. [Val Acc 59%, Loss 1.404507]
        Iter 20. [Val Acc 65%, Loss 0.950363]
        Iter 30. [Val Acc 68%, Loss 0.770373]
        Iter 40. [Val Acc 69%, Loss 0.682717]
        Iter 50. [Val Acc 71%, Loss 0.647218]
        Iter 60. [Val Acc 71%, Loss 0.618623]
        Iter 70. [Val Acc 71%, Loss 0.635205]
        Iter 80. [Val Acc 71%, Loss 0.597471]
        Iter 90. [Val Acc 71%, Loss 0.613755]
        Iter 100. [Val Acc 71%, Loss 0.606581]
        Iter 10. [Val Acc 59%, Loss 1.348778]
```

```
Iter 20. [Val Acc 65%, Loss 0.953157]
Iter 30. [Val Acc 67%, Loss 0.769828]
Iter 40. [Val Acc 68%, Loss 0.687085]
Iter 50. [Val Acc 70%, Loss 0.634212]
Iter 60. [Val Acc 71%, Loss 0.608376]
Iter 70. [Val Acc 71%, Loss 0.604338]
Iter 80. [Val Acc 72%, Loss 0.590251]
Iter 90. [Val Acc 71%, Loss 0.601807]
Iter 100. [Val Acc 72%, Loss 0.582371]
Iter 10. [Val Acc 59%, Loss 1.389879]
Iter 20. [Val Acc 65%, Loss 0.952868]
Iter 30. [Val Acc 68%, Loss 0.759939]
Iter 40. [Val Acc 70%, Loss 0.683698]
Iter 50. [Val Acc 69%, Loss 0.656940]
Iter 60. [Val Acc 72%, Loss 0.606222]
Iter 70. [Val Acc 72%, Loss 0.586225]
Iter 80. [Val Acc 70%, Loss 0.608038]
Iter 90. [Val Acc 72%, Loss 0.581753]
Iter 100. [Val Acc 72%, Loss 0.580081]
The optimal w is: [ 1.40400194e+00 -8.89342925e-01 -9.07169903e-02 -2.31901676e-01
 -1.20887730e-01 -5.75052590e-01 -2.88882143e-02 -2.00240513e-01
 -7.68141687e-02 -6.25417730e-02 -1.25055095e-01 -1.02984219e-03
  2.35164311e-01  3.31809082e-01 -1.20590955e-01 -1.10070052e-01
  4.96244773e-02  7.83891284e-02  1.66418862e-01  2.00165212e-01
  6.13820891e-02 -2.69759448e-01  1.05553209e+00  1.03909090e-01
 -1.38081971e-01 -1.08567114e-01  1.64023335e-01 -4.34679700e-02
 -6.14596118e-02  1.26435080e-01 -2.80859381e-02 -3.48934455e-02
 -1.03751158e-01 -2.33481812e-02 -5.90581325e-02 -6.16809537e-03
  1.54413661e-04  1.07673548e-01  1.12553222e-01 -7.44103018e-03
 -2.73410723e-01 -7.08685708e-02 -5.67231209e-02 -2.46350493e-03
  9.96103274e-04  5.38521373e-02 -2.46479845e-02 -9.26668511e-02
  3.59986524e-02 -2.99937798e-02  2.21673895e-02 -9.78184074e-02
  3.17957419e-02  2.24979735e-03 -1.44884156e-01 -5.42182353e-02
 -1.34752276e-01  3.72636786e-01 -7.42042141e-02 -1.15565864e-01
  1.87217831e-03  5.60709979e-02 -2.14372426e-01  6.84275889e-02
 -1.08594316e-01  1.46129575e-02  1.42722981e-02  5.71635918e-02
 -1.14468949e-01 -5.74076463e-02 -3.56692791e-02  4.12175846e-03
  4.27414361e-03  4.96849583e-02  1.94921539e-02 -3.95920320e-02
  4.83043875e-02 -8.25579546e-02  1.49707045e-01  2.23849929e-02
 -8.81712953e-03 -1.42133279e-02  1.49363781e-01 -5.88886139e-02
  1.15445722e-01 -6.47765446e-02  5.70285428e-02 -2.58885732e-01
 -3.35751083e-02  1.03954124e-01]

The optimal b is: 0.34797278473773213

Desired behavior, for the optimal w and b for good mu and batch size:
```





```
Iter 10. [Val Acc 55%, Loss 9.260579]
Iter 20. [Val Acc 60%, Loss 8.304836]
Iter 30. [Val Acc 62%, Loss 6.829526]
Iter 40. [Val Acc 65%, Loss 6.276969]
Iter 50. [Val Acc 59%, Loss 9.577724]
Iter 60. [Val Acc 64%, Loss 7.151208]
Iter 70. [Val Acc 63%, Loss 7.136384]
Iter 80. [Val Acc 62%, Loss 6.844744]
Iter 90. [Val Acc 66%, Loss 6.389411]
Iter 100. [Val Acc 64%, Loss 7.730463]
Bad behavior, for the optimal w and b for good mu and batch size:
```

Explain and discuss your results here:

Part (h) -- 15%

Using the values of `w` and `b` from part (g), compute your training accuracy, validation accuracy, and test accuracy. Are there any differences between those three values? If so, why?

```
# Write your code here

train_y = pred(w_opt,b_opt,train_norm_xs)
train_acc = get_accuracy(train_y,train_ts)
val_y = pred(w_opt,b_opt,val_norm_xs)
val_acc = get_accuracy(val_y,val_ts)
test_y = pred(w_opt,b_opt,test_norm_xs)
test_acc = get_accuracy(test_y,test_ts)

print('train_acc = ', train_acc, ' val_acc = ', val_acc, ' test_acc = ', test_acc)


# There is little difference between the values. It makes sense that the test set
# has the worst accuracy, for two reasons:
# 1) As we explained before, the values are computed with the train data so they do not
#    fit perfectly to the test data.
# 2) It is the smallest set.
```

```
train_acc =  0.7183785939596099  val_acc =  0.7209  test_acc =  0.7088514429595196
```

Explain and discuss your results here:

Part (i) -- 15%

Writing a classifier like this is instructive, and helps you understand what happens when we train a model. However, in practice, we rarely write model building and training code from scratch. Instead, we typically use one of the well-tested libraries available in a package.

Use `sklearn.linear_model.LogisticRegression` to build a linear classifier, and make predictions about the test set. Start by reading the API documentation here.

Compute the training, validation and test accuracy of this model.

```
import sklearn.linear_model

model = sklearn.linear_model.LogisticRegression()
model.fit(train_norm_xs,train_ts)

train_acc = model.score(train_norm_xs,train_ts)
val_acc = model.score(val_norm_xs,val_ts)
test_acc = model.score(test_norm_xs,test_ts)

print('train_acc = ', train_acc, ' val_acc = ', val_acc, ' test_acc = ', test_acc)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: DataConversi
  y = column_or_1d(y, warn=True)
train_acc =  0.7325380998996894  val_acc =  0.73588  test_acc =  0.726748014720124
```

This parts helps by checking if the code worked. Check if you get similar results, if not repair your code