In [2]:

```
import pandas
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(1)
```

In [3]:

```
load_from_drive = False

if not load_from_drive:
  csv_path = "http://archive.ics.uci.edu/ml/machine-learning-databases/00203/YearPredicti
onMSD.txt.zip"
else:
  from google.colab import drive
  drive.mount('/content/gdrive/')
  csv_path = '/content/gdrive/MyDrive/Colab Notebooks/YearPredictionMSD.txt'

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)
```

In [ ]:

```
df
```

Out[ ]:

	year	var1	var2	var3	var4	var5	var6	var7	var8	var9	...	var81	var
0	2001	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12.23257	7.83089	...	13.01620	-54.405
1	2001	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0.92019	18.76548	...	5.66812	-19.680
2	2001	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2.35035	16.07017	...	3.03800	26.058
3	2001	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10.34124	3.55005	...	34.57337	171.707
4	2001	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9.37636	12.63699	...	9.92661	-55.957
...	...	...	...	...	...	...	...	...	...	...	...	...	
515340	2006	51.28467	45.88068	22.19582	-5.53319	-3.61835	-16.36914	2.12652	5.18160	-8.66890	...	4.81440	-3.759
515341	2006	49.87870	37.93125	18.65987	-3.63581	-27.75665	-18.52988	7.76108	3.56109	-2.50351	...	32.38589	-32.755
515342	2006	45.12852	12.65758	-38.72018	8.80882	-29.29985	-2.28706	-18.40424	-22.28726	-4.52429	...	-18.73598	-71.159
515343	2006	44.16614	32.38368	-3.34971	-2.49165	-19.59278	-18.67098	8.78428	4.02039	-12.01230	...	67.16763	282.776
515344	2005	51.85726	59.11655	26.39436	-5.46030	-20.69012	-19.95528	-6.72771	2.29590	10.31018	...	-11.50511	-69.182

515345 rows × 91 columns

In [4]:

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

In [5]:

```
df.head(25)
```

```
df.head(25)
```

Out[5]:

	year	var1	var2	var3	var4	var5	var6	var7	var8	var9	...	var81	var82
0	1	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12.23257	7.83089	...	13.01620	-54.40548
1	1	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0.92019	18.76548	...	5.66812	-19.68073
2	1	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2.35035	16.07017	...	3.03800	26.05866
3	1	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10.34124	3.55005	...	34.57337	-171.70734
4	1	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9.37636	12.63699	...	9.92661	-55.95724
5	1	50.54767	0.31568	92.35066	22.38696	-25.51870	-19.04928	20.67345	-5.19943	3.63566	...	6.59753	-50.69577
6	1	50.57546	33.17843	50.53517	11.55217	-27.24764	-8.78206	-12.04282	-9.53930	28.61811	...	11.63681	25.44182
7	1	48.26892	8.97526	75.23158	24.04945	-16.02105	-14.09491	8.11871	-1.87566	7.46701	...	18.03989	-58.46192
8	1	49.75468	33.99581	56.73846	2.89581	-2.92429	-26.44413	1.71392	-0.55644	22.08594	...	18.70812	5.20391
9	1	45.17809	46.34234	-40.65357	-2.47909	1.21253	-0.65302	-6.95536	-12.20040	17.02512	...	-4.36742	-87.55285
10	1	39.13076	-23.01763	-36.20583	1.67519	-4.27101	13.01158	8.05718	-8.41088	6.27370	...	32.86051	-26.08461
11	1	37.66498	-34.05910	-17.36060	-26.77781	-39.95119	-20.75000	-0.10231	-0.89972	-1.30205	...	11.18909	45.20614
12	1	26.51957	-148.15762	-13.30095	-7.25851	17.22029	-21.99439	5.51947	3.48418	2.61738	...	23.80442	251.76360
13	1	37.68491	-26.84185	-27.10566	-14.95883	-5.87200	-21.68979	4.87374	-18.01800	1.52141	...	-67.57637	234.27192
14	0	39.11695	-8.29767	-51.37966	-4.42668	-30.06506	-11.95916	-0.85322	-8.86179	11.36680	...	42.22923	478.26580
15	1	35.05129	-67.97714	-14.20239	-6.68696	-0.61230	-18.70341	-1.31928	-9.46370	5.53492	...	10.25585	94.90539
16	1	33.63129	-96.14912	-89.38216	-12.11699	13.77252	-6.69377	-33.36843	-24.81437	21.22757	...	49.93249	-14.47489
17	0	41.38639	-20.78665	51.80155	17.21415	-36.44189	-11.53169	11.75252	-7.62428	-3.65488	...	50.37614	-40.48205
18	0	37.45034	11.42615	56.28982	19.58426	-16.43530	2.22457	1.02668	-7.34736	-0.01184	...	-22.46207	-25.77228
19	0	39.71092	-4.92800	12.88590	-11.87773	2.48031	-16.11028	-16.40421	-8.29657	9.86817	...	11.92816	-73.72412
20	0	37.22392	-88.45128	-1.54036	-15.89576	3.81949	-6.59919	-2.17960	15.62713	12.33636	...	-25.59803	-22.08026
21	0	41.37983	0.14209	2.34359	-11.16799	2.92536	-5.85021	-2.94938	16.48136	15.27529	...	-31.82900	-48.93128
22	1	42.15927	-36.86085	-27.15064	-16.19360	41.07527	-6.85163	-23.05788	-3.13688	14.19481	...	-2.86694	-122.84548
23	0	33.24221	-95.01277	37.87976	1.49067	5.48344	3.13076	-10.08094	16.65756	19.21635	...	20.52507	-92.65642
24	0	44.21532	15.52530	-8.38455	-5.72959	-5.72018	-8.56578	2.67454	-2.46999	14.02056	...	4.16200	-123.03059

25 rows × 91 columns

In [6]:

```
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()
```

Part (a): The reason we want to avoid the songs of a single artist to appear in both the training set and the test set is because this might cause the model to be unreliable. The model should be able to verify the year the song was release, not according to a set of specific artist. It should be independent to a chosen artist. Therefore, in order to make sure the model won't relay on the artist set given in the training, the model will be tested by a songs of a different set of artists. In other words, we want to generelize our model as much as possible. An artist usually releases songs in a specific period of time (a decade or two), an one's songs might sound like eachother, becauses artists has a specific style. That way, we avoid facing similar songs both in training and test sets, that reduces the generalization power of the model.

In [7]:

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

Part (b): It is not a mistake that the normalization of the test data set was made according to the means and the standard deviations of the training set, since there is an assumption that the whole data set (both the training and the test set) have the same distribution. Moreover, in order to test the model more accurately we want to test the model by samples of which we know nothing about them. Therefore, if we normalized the test set by its own means and standard deviations, it comes out that we do know something about the data or its first and second moments. In order to avoid this, the test set will be normalized by the data we learned in the past.

In [8]:

```
# 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
val_size = 50000
train_xs, val_xs           = train_xs[val_size:], train_xs[:val_size]
train_norm_xs, val_norm_xs = train_norm_xs[val_size:], train_norm_xs[:val_size]
train_ts, val_ts           = train_ts[val_size:], train_ts[:val_size]
```

Part (c): The reason we should limit how many times we use the test set is avoiding overfitting, and increasing our generalization power. Overfitting means that the model is trained exactly according to the samples given in the train set. This is unwanted since the model unfortunately cannot perform accurately against unseen data, defeating its purpose. Moreover,the reason for using the validation set during the model building process is making comparisons whlie tuning the model, i.e. selecting the optimal values of hyperparameters, and only then we use the test set to evalute our results with unseen samples and in an unbiased way.

In [9]:

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

def cross_entropy(t, y):
  if type(t) != int:
    t = t.reshape((np.shape(y)))
```

```
  return -t * np.log(y+10**-8) - (1 - t) * np.log(1 - y+10**-8)

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
```

In [10]:

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

  y=σ(wTx+b)

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

  """
  z  = np.dot(np.transpose(w),np.transpose(X))+b

  return sigmoid(z)
```

In [11]:

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

Out[11]:

```
array([0.73105858, 0.73105858])
```

In [12]:

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

Out[12]:

```
(2,)
```

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.

In [13]:

```
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
```

```
    dL_db = y-t
    #Converting (y-t) to the same size of X, and then multiplying:
    dL_dw =np.transpose(X)*(np.shape(X)[1]*[y-t])

    return(dL_dw,dL_db)
```

Explenation on Gradients

Add here an explaination on how the gradients are computed :

Write your explanation here. Use Latex to write mathematical expressions. [Here is a brief tutorial on latex for notebooks.]()

Explenation:

Let's take a look on the gradient calculation.

In order to derive the Loss function with respect to $\mathbf{w}$
and b
, we use chain rule for derivaties:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial \mathbf{w}}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial b}$$

Now, knowing the cross-entropy loss function, we write:

$$\mathcal{L}(y,t) = -tlog(y) - (1-t)log(1-y)$$

and when deriving:

$$\frac{\partial \mathcal{L}}{\partial y} = -\frac{t}{y} + \frac{1-t}{1-y} = \frac{y-t}{y(1-y)}$$

and now for sigmoid function:

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

we get:

$$\frac{\partial y}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z)) = y(1-y)$$

and now, using the connection $z = \mathbf{w}^T\mathbf{x} + \mathrm{b}$
, we get:

$$\frac{\partial z}{\partial b} = 1$$

, and $\frac{\partial z}{\partial \mathbf{w}} = \mathbf{x}$

multiplying everythimg together yields:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial \mathbf{w}} = \frac{y-t}{y(1-y)}\frac{y(1-y)}{1}\mathbf{x} = (y-t)\mathbf{x}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial b} = \frac{y-t}{y(1-y)}\frac{y(1-y)}{1} = (y-t)$$

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.

In [14]:

```
# Your code goes here
N=90
# Creating simple values
w = 0.0*np.ones(N)
b = 1
h = 10**-8   # Taking an arbitrary small positive number
X = np.random.rand(1,N)
t = 0

# Creating y from function in Ex. 2a.
y =pred(w,b,X)

# Calculating analytical result:
r1 = derivative_cost(X,y,t)
print("The analytical results is -\n",  r1[1],'\n')

# Now for numerical derivative:
y_h = pred(w,b+h,X)
r2 = (cross_entropy(t,y_h)-cross_entropy(t,y))/h
print("The algorithm results is - \n", r2)
print("The difference is: ", r2-r1[1], "\n Close enough...")
```

```
The analytical results is -
 [0.73105858]

The algorithm results is -
 [0.73105852]
The difference is:  [-5.38023169e-08]
 Close enough...
```

Part (d) -- 7%

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

In [15]:

```
# 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

w = 0*np.random.rand(N)
b = 0
h = 10**-8
X = np.random.rand(1,N)
t = 1
y = pred(w,b,X)

# Calculating analytical result:
r1 = derivative_cost(X,y,t)[0]  # The first element is the dw vector.
#print("The analytical results is -\n",  r1,'\n')


# Now for numerical derivative:
r2=np.zeros(np.shape(r1))
```

```
for i in range(N):
  w_h=np.ones(np.shape(w))*w
  w_h[i]=w[i]+h
  y_h=pred(w_h,b, X)
  r2[i,:]=(cross_entropy(t, y_h)-cross_entropy(t, y))/h

#y_h = pred(w+h,b,X)
#r2 = (cross_entropy(t,y_h)-cross_entropy(t,y))/h
print("The algorithm results is - \n", r2)
print("\n The differences are:\n ", r2-r1, "\n Close enough...")
```

```
The algorithm results is -
 [[-0.12497174]
 [-0.30297999]
 [-0.34908026]
 [-0.26332164]
 [-0.15221278]
 [-0.03031266]
 [-0.28710486]
 [-0.30493198]
 [-0.01596718]
 [-0.09663981]
 [-0.13337551]
 [-0.15799384]
 [-0.44017742]
 [-0.0592282 ]
 [-0.15862935]
 [-0.06403142]
 [-0.09851175]
 [-0.10768186]
 [-0.05256295]
 [-0.19961082]
 [-0.10969323]
 [-0.24339322]
 [-0.48961558]
 [-0.20285496]
 [-0.29327741]
 [-0.35494797]
 [-0.28387194]
 [-0.29326924]
 [-0.30436084]
 [-0.12554229]
 [-0.33048025]
 [-0.36723565]
 [-0.06164322]
 [-0.13147692]
 [-0.22103224]
 [-0.2835307 ]
 [-0.34703426]
 [-0.48104692]
 [-0.00346687]
 [-0.22112632]
 [-0.16067851]
 [-0.29088553]
 [-0.22414419]
 [-0.1417007 ]
 [-0.20192772]
 [-0.18121362]
 [-0.32110971]
 [-0.42977114]
 [-0.20905591]
 [-0.0300616 ]
 [-0.33055408]
 [-0.48266841]
 [-0.39497693]
 [-0.38875784]
 [-0.46212453]
 [-0.21685362]
 [-0.17124102]
 [-0.4322609 ]
 [-0.40909943]
 [-0.25425726]
```

```
[-0.01392375]
[-0.41499121]
[-0.09930059]
[-0.47510792]
[-0.32444661]
[-0.06644165]
[-0.0782354 ]
[-0.15757866]
[-0.10275765]
[-0.32526487]
[-0.28399724]
[-0.0622806 ]
[-0.2594431 ]
[-0.28097918]
[-0.45332308]
[-0.13722983]
[-0.42220675]
[-0.15992805]
[-0.01397749]
[-0.30779937]
[-0.3698407 ]
[-0.35406317]
[-0.28730861]
[-0.11807217]
[-0.11411592]
[-0.19363822]
[-0.02133393]
[-0.36316216]
[-0.49670584]
[-0.42245093]]

The differences are:
 [[-8.41174375e-09]
[ 1.31970211e-08]
[ 1.88436128e-09]
[ 6.81526874e-09]
[-7.89288584e-09]
[-1.28221851e-08]
[ 5.19735310e-09]
[ 1.27754192e-08]
[-1.10972867e-08]
[-7.28435234e-09]
[ 7.41036510e-10]
[ 5.57733604e-09]
[ 9.46277401e-09]
[ 6.05733969e-09]
[ 1.41226519e-09]
[-6.88844548e-09]
[ 7.27495908e-09]
[-9.34523797e-09]
[-4.18758767e-09]
[-2.75887480e-10]
[-8.29426128e-09]
[-3.33259714e-09]
[ 5.27190036e-09]
[ 1.79405368e-09]
[-1.55688140e-09]
[ 1.55435927e-08]
[ 1.13497312e-08]
[ 1.69637729e-08]
[ 4.70748984e-09]
[-1.25862175e-08]
[ 1.63143196e-08]
[ 6.64724648e-09]
[-1.16132032e-08]
[ 3.08109993e-09]
[-3.90153254e-09]
[ 1.88151999e-08]
[ 1.52266185e-08]
[ 2.08008921e-08]
[ 3.93246885e-09]
[ 4.70396333e-10]
```

```
 [-3.49849161e-09]
 [ 6.12042433e-09]
 [-1.01538036e-08]
 [ 4.34369873e-09]
 [-8.47361070e-10]
 [ 8.11272571e-09]
 [ 1.48078976e-08]
 [ 1.15489721e-08]
 [-2.03254324e-09]
 [ 7.00426367e-09]
 [ 5.97273814e-09]
 [ 1.08874562e-08]
 [ 3.51077151e-09]
 [ 1.45632205e-08]
 [ 1.01903178e-08]
 [-5.46527895e-09]
 [-4.34467157e-09]
 [ 1.14483295e-09]
 [ 1.89269781e-08]
 [-1.31162063e-08]
 [ 4.03238309e-09]
 [ 1.16120742e-08]
 [ 5.27046756e-09]
 [ 2.00937883e-08]
 [ 1.90429572e-08]
 [-9.63427560e-09]
 [ 7.23301130e-09]
 [-6.46283693e-09]
 [-7.93461880e-09]
 [ 4.16913043e-09]
 [ 3.04046988e-09]
 [-1.06155802e-08]
 [-2.77303791e-10]
 [ 8.46955583e-09]
 [ 1.26400028e-08]
 [ 7.23003701e-09]
 [ 5.64730668e-09]
 [ 2.80079093e-09]
 [-1.04320044e-08]
 [ 1.54955109e-08]
 [ 1.97207956e-08]
 [ 1.05496953e-08]
 [-4.82549778e-10]
 [-1.56827007e-09]
 [-7.22077076e-09]
 [-1.37648509e-09]
 [-9.94616495e-09]
 [ 7.24031141e-09]
 [ 3.81766019e-09]
 [ 7.86723153e-09]]
Close enough...
```

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:

In [18]:

```
def run_gradient_descent(train_norm_xs, train_ts, val_norm_xs, val_ts,w0, b0, mu=0.1, ba
tch_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
  """
  w = w0
  b = b0
  iter = 0
  val_costs = []
  val_accs = []
  train_cost =[]
  train_acc = []


  while iter < max_iters:

      # At the beggining of every epoch, we shuffle the training set
      # Thats what *Stochastic* in the SGD
      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 = pred(w,b,X)

        # update w and b
        w_tag,b_tag = derivative_cost(X,y,t)
        w = w-mu*np.mean(w_tag, axis=1) # Each iteration is taken from a batch, we're av
eraging on each column.
        b = b-mu*np.mean(b_tag) # Each iteration is taken from a batch, we're averaging
on all values.


        # increment the iteration count - happaning every epoch.
        # We indented the next line left in order to make it more logical.
      iter += 1
        # compute and print the *validation* loss and accuracy
      if (iter % 10 == 0):
        #evaluating on the validation:
        y_val = pred(w,b,val_norm_xs) #normalized for anlyzing
        val_costs.append(cost(y_val,val_ts))
        val_accs.append(get_accuracy(y_val,val_ts))
        print("Iter %d. [Val Acc %.0f%%, Loss %f]" %(iter, val_accs[-1]*100, val_costs[-
1]))


        # for further exercises, taking the training loss.
        y = pred(w,b,train_norm_xs)
        train_cost.append(cost(y,train_ts))
        train_acc.append(get_accuracy(y,train_ts))


  return w, b, val_costs, val_accs ,train_cost, train_acc
```

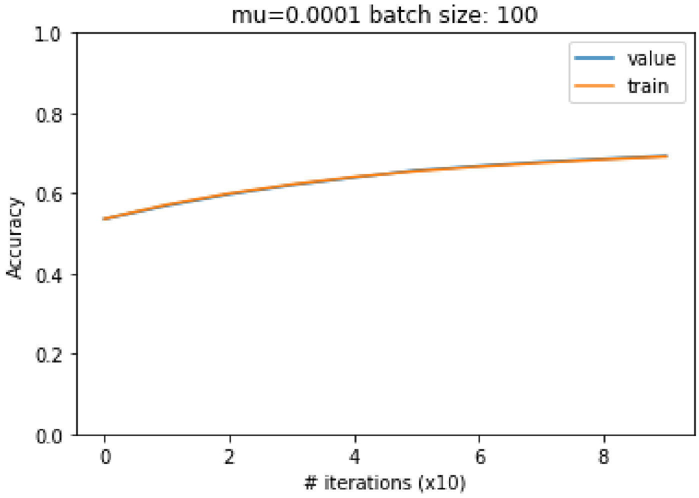
Part (f) -- 7%

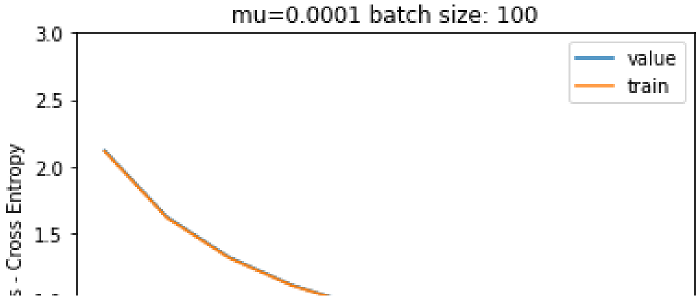
Call `run_gradient_descent` with the weights and biases all initialized to zero. Show that if the learning rate μ is too small, then convergence is slow. Also, show that if μ is too large, then the optimization algorirthm does not converge. The demonstration should be made using plots showing these effects.

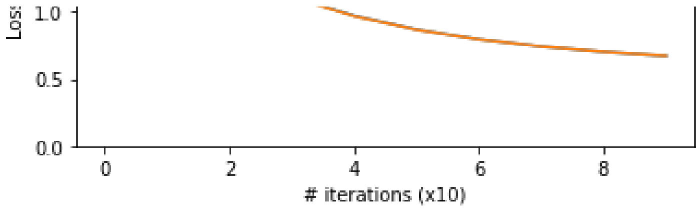
In [22]:

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

# Taking two artitrary values of mu fulfilling the requierments:
# According to the results, we know how good these values are.

mus = [0.0001,10]

for mu in mus:
  w, b, val_costs, val_accs ,train_costs, train_accs = run_gradient_descent(train_norm_x
s,train_ts,val_norm_xs,val_ts,w0,b0,mu)
  plt.plot(range(len(val_accs)),val_accs)
  plt.plot(range(len(train_accs)),train_accs)
  plt.xlabel('# iterations (x10)')
  plt.ylabel('Accuracy')
  plt.title('mu=' + str(mu) +' batch size: 100')
  plt.ylim([0,1])
  plt.legend(['value','train'])
  plt.show()

  plt.plot(range(len(val_costs)),val_costs)
  plt.plot(range(len(train_costs)),train_costs)
  plt.xlabel('# iterations (x10)')
  plt.ylabel('Loss - Cross Entropy')
  plt.title('mu=' + str(mu)+' batch size: 100')
  if (mu == 0.0001):
    plt.ylim([0,3])
  if (mu == 10):
    plt.ylim([0,5])
  plt.legend(['value','train'])
  plt.show()
```

```
Iter 10. [Val Acc 54%, Loss 2.120527]
Iter 20. [Val Acc 57%, Loss 1.623045]
Iter 30. [Val Acc 60%, Loss 1.321360]
Iter 40. [Val Acc 62%, Loss 1.114785]
Iter 50. [Val Acc 64%, Loss 0.969954]
Iter 60. [Val Acc 66%, Loss 0.867686]
Iter 70. [Val Acc 67%, Loss 0.794399]
Iter 80. [Val Acc 68%, Loss 0.741090]
Iter 90. [Val Acc 69%, Loss 0.701650]
Iter 100. [Val Acc 69%, Loss 0.671991]
```



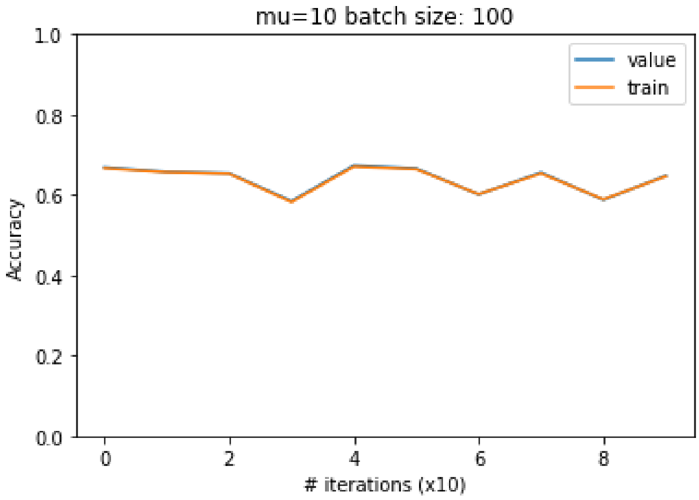


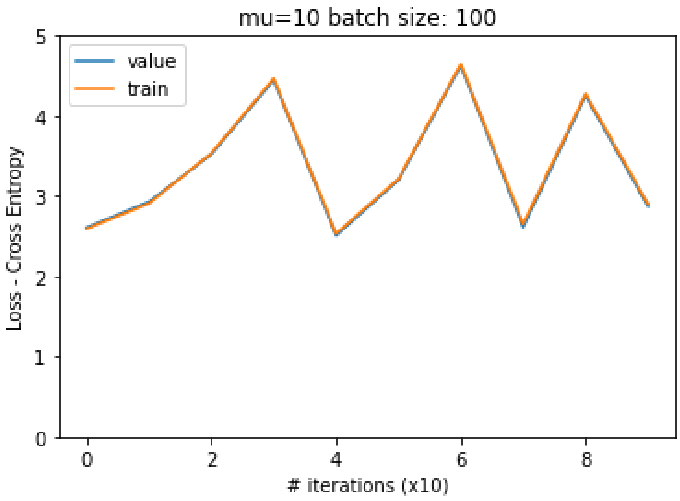



```
Iter 10. [Val Acc 67%, Loss 2.607009]
Iter 20. [Val Acc 66%, Loss 2.926539]
Iter 30. [Val Acc 65%, Loss 3.519260]
Iter 40. [Val Acc 58%, Loss 4.445630]
Iter 50. [Val Acc 67%, Loss 2.509323]
Iter 60. [Val Acc 67%, Loss 3.198608]
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: overflow
encountered in exp
```

```
Iter 70. [Val Acc 60%, Loss 4.625099]
Iter 80. [Val Acc 66%, Loss 2.612666]
Iter 90. [Val Acc 59%, Loss 4.250221]
Iter 100. [Val Acc 65%, Loss 2.873432]
```





Explain and discuss your results here: Too low learning rate: For μ = 0.0001 and the default batch size it can be seen that the accuray is growing in a general increament. Though this improvement is happening very slow and therefore converges very slowly. For similar reason the loss is decreasing very slowly. Moreover, since the SGD method is used, the decreasing of the loss and the increasing of the accuracy is not constantly but a bit varying. This is happening because of the stochastic trend in the SGD.

Too high learning rate: For μ = 10 and the default batch size it can be seen that the accuray is very noisy. Meaning that accuracy is neither generally increasing nor decreasing. It is varying, sometimes improving and sometimes not. It means that the model is not converging with a too high learning rate. The weights are bouncing around a given point in the loss surface and therefore the model is not converging.

For both, the large μ and the small μ, we might see that our model fits to the validation. In the loss figures the

training curve and the validation curve are almost merging, therfore we have no overfitiing in our model. In the next exercise we will find an optimal value for μ.

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.

In [25]:

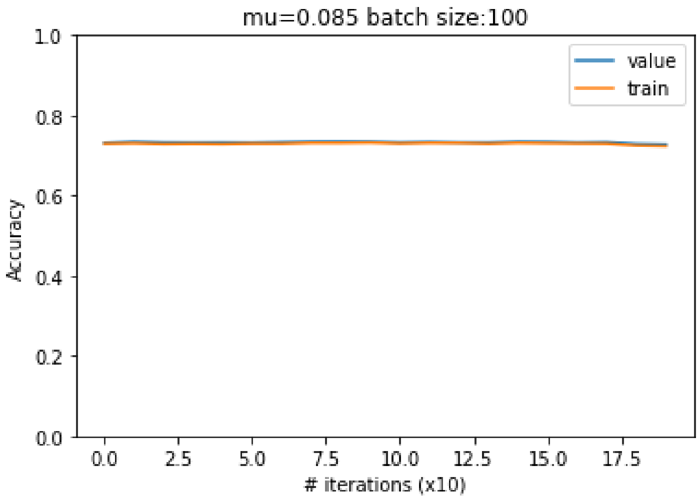
```
def find_best(batch_size):
  mus = [0.085,0.5,0.7,1] # [i//50 for i in range(50)]

  for mu in mus:
    w, b, val_costs, val_accs ,train_costs, train_accs = run_gradient_descent(train_norm
_xs,train_ts,val_norm_xs,val_ts,w0,b0,mu,batch_size, max_iters=200)
    plt.plot(range(len(val_accs)),val_accs)
    plt.plot(range(len(train_accs)),train_accs)
    plt.xlabel('# iterations (x10)')
    plt.ylabel('Accuracy')
    plt.title('mu=' + str(mu)+' batch size:'+str(batch_size))
    plt.ylim([0,1])
    plt.legend(['value','train'])
    plt.show()

    plt.plot(range(len(val_costs)),val_costs)
    plt.plot(range(len(train_costs)),train_costs)
    plt.xlabel('# iterations (x10)')
    plt.ylabel('Loss - Cross Entropy')
    plt.title('mu=' + str(mu)+' batch size:'+str(batch_size))
    plt.ylim([0,1.5])
    plt.legend(['value','train'])
    plt.show()
  return
```

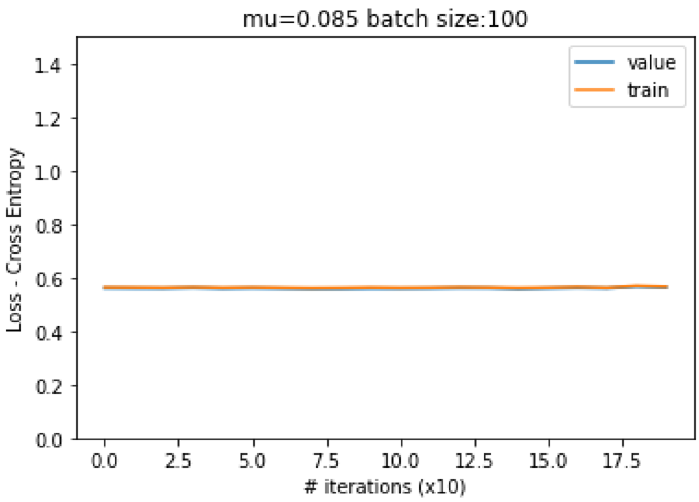
In [26]:

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


# 0.0001 and 10 as learning rate values were not suitable as seen above
# Therefore we will choose a mu in between statring with 0.5 and batch_size 100

# We will run a grid search over many MUs.
# In order to keep the document short, we will present only 4 MUs for each iteration.
# Mean while, we will make a grid search over batch sizes.

# The results are presented here:

batch_size = 100
find_best(batch_size)

batch_size = 500
find_best(batch_size)


batch_size = 2000
find_best(batch_size)


batch_size = 20000
find_best(batch_size)


#Rerun the model with optimal values found for later use:
```

```
w_opt, b_opt ,val_costs, val_accs ,train_costs, train_accs = run_gradient_descent(train_norm_xs,train_ts,val_norm_xs,val_ts,w0,b0,0.085,20000)
```

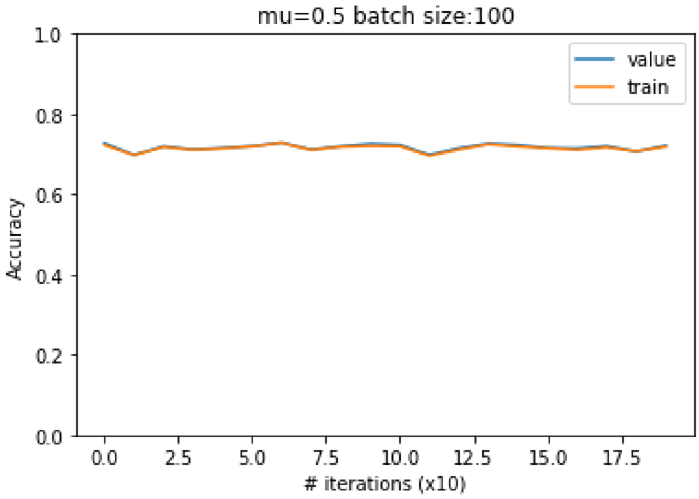
```
Iter 10. [Val Acc 73%, Loss 0.560598]
Iter 20. [Val Acc 73%, Loss 0.559926]
Iter 30. [Val Acc 73%, Loss 0.559605]
Iter 40. [Val Acc 73%, Loss 0.561612]
Iter 50. [Val Acc 73%, Loss 0.559205]
Iter 60. [Val Acc 73%, Loss 0.560407]
Iter 70. [Val Acc 73%, Loss 0.559369]
Iter 80. [Val Acc 73%, Loss 0.558203]
Iter 90. [Val Acc 74%, Loss 0.558375]
Iter 100. [Val Acc 73%, Loss 0.559357]
Iter 110. [Val Acc 73%, Loss 0.559230]
Iter 120. [Val Acc 73%, Loss 0.559618]
Iter 130. [Val Acc 73%, Loss 0.560862]
Iter 140. [Val Acc 73%, Loss 0.560588]
Iter 150. [Val Acc 73%, Loss 0.557997]
Iter 160. [Val Acc 73%, Loss 0.559868]
Iter 170. [Val Acc 73%, Loss 0.561720]
Iter 180. [Val Acc 73%, Loss 0.560186]
Iter 190. [Val Acc 73%, Loss 0.566331]
Iter 200. [Val Acc 73%, Loss 0.563408]
```

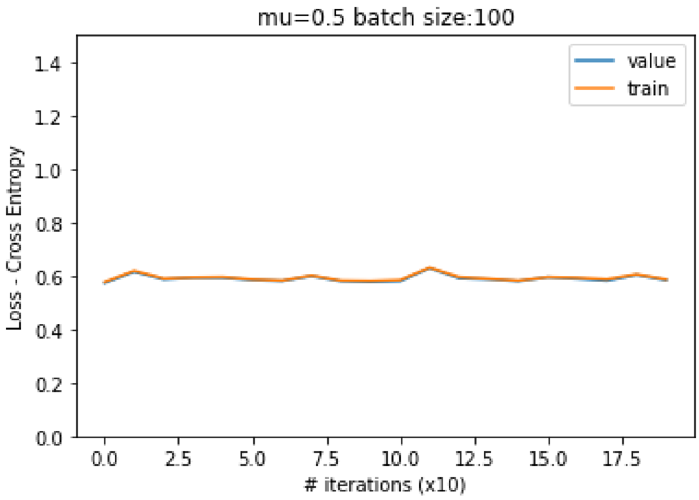




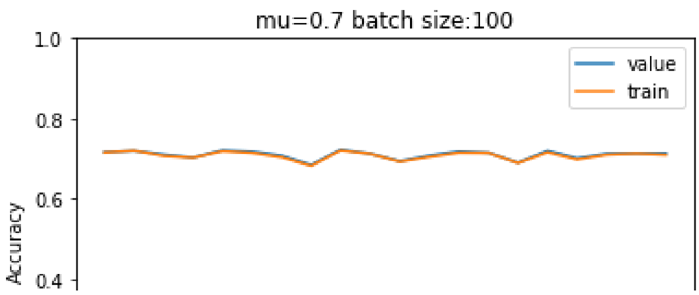

```
Iter 10. [Val Acc 73%, Loss 0.572709]
Iter 20. [Val Acc 70%, Loss 0.614940]
Iter 30. [Val Acc 72%, Loss 0.586680]
Iter 40. [Val Acc 71%, Loss 0.592901]
Iter 50. [Val Acc 72%, Loss 0.592625]
Iter 60. [Val Acc 72%, Loss 0.584327]
Iter 70. [Val Acc 73%, Loss 0.580489]
Iter 80. [Val Acc 71%, Loss 0.598675]
Iter 90. [Val Acc 72%, Loss 0.579599]
Iter 100. [Val Acc 73%, Loss 0.576745]
Iter 110. [Val Acc 72%, Loss 0.579484]
Iter 120. [Val Acc 70%, Loss 0.628781]
Iter 130. [Val Acc 71%, Loss 0.590736]
Iter 140. [Val Acc 73%, Loss 0.587010]
```

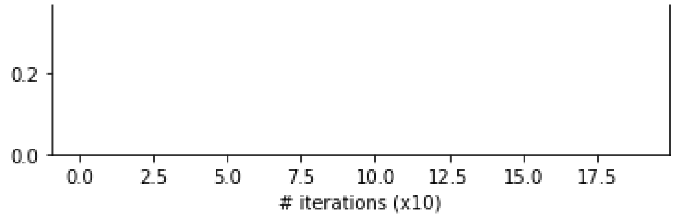
```
Iter 150. [Val Acc 72%, Loss 0.579768]
Iter 160. [Val Acc 72%, Loss 0.594530]
Iter 170. [Val Acc 71%, Loss 0.588883]
Iter 180. [Val Acc 72%, Loss 0.581735]
Iter 190. [Val Acc 71%, Loss 0.602794]
Iter 200. [Val Acc 72%, Loss 0.583729]
```

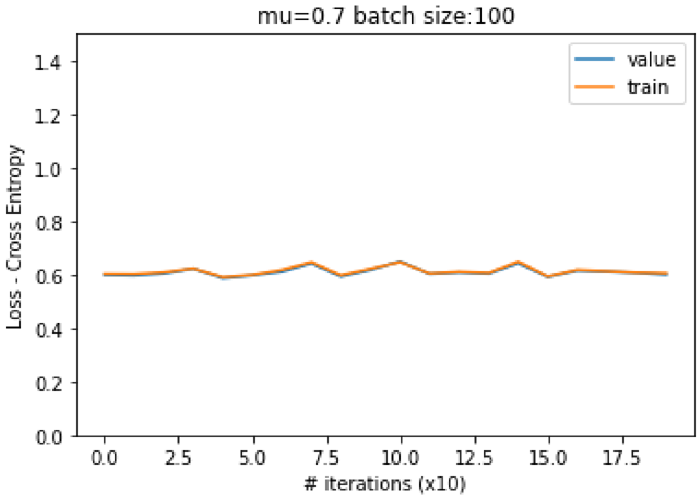




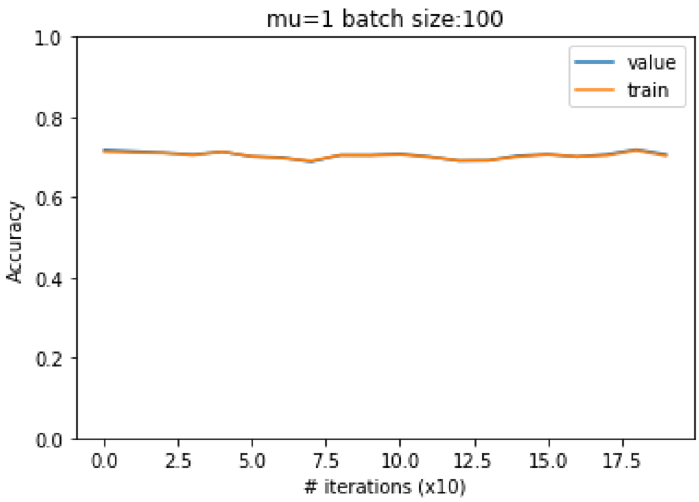

```
Iter 10. [Val Acc 72%, Loss 0.598198]
Iter 20. [Val Acc 72%, Loss 0.596140]
Iter 30. [Val Acc 71%, Loss 0.603737]
Iter 40. [Val Acc 70%, Loss 0.621814]
Iter 50. [Val Acc 72%, Loss 0.586713]
Iter 60. [Val Acc 72%, Loss 0.596773]
Iter 70. [Val Acc 71%, Loss 0.610595]
Iter 80. [Val Acc 68%, Loss 0.641298]
Iter 90. [Val Acc 72%, Loss 0.592542]
Iter 100. [Val Acc 71%, Loss 0.617201]
Iter 110. [Val Acc 69%, Loss 0.648526]
Iter 120. [Val Acc 71%, Loss 0.602404]
Iter 130. [Val Acc 72%, Loss 0.607959]
Iter 140. [Val Acc 71%, Loss 0.603386]
Iter 150. [Val Acc 69%, Loss 0.641987]
Iter 160. [Val Acc 72%, Loss 0.591251]
Iter 170. [Val Acc 70%, Loss 0.614830]
Iter 180. [Val Acc 71%, Loss 0.611161]
Iter 190. [Val Acc 71%, Loss 0.606473]
Iter 200. [Val Acc 71%, Loss 0.599224]
```

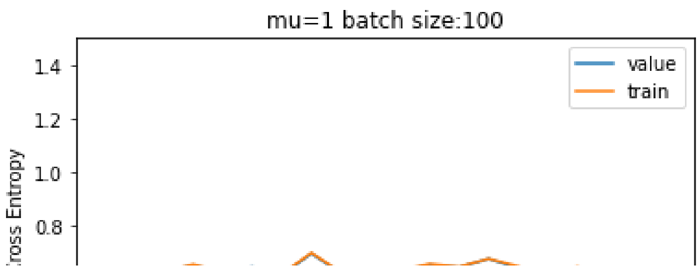


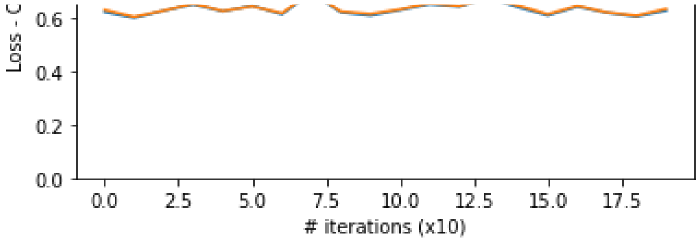




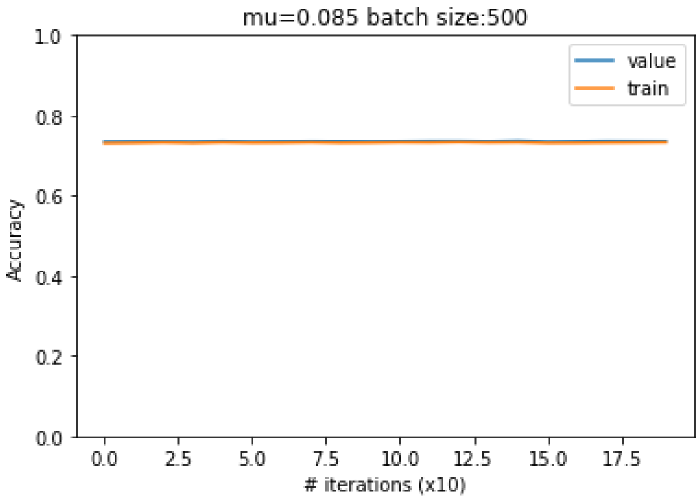

```
Iter 10. [Val Acc 72%, Loss 0.620780]
Iter 20. [Val Acc 71%, Loss 0.598986]
Iter 30. [Val Acc 71%, Loss 0.625587]
Iter 40. [Val Acc 71%, Loss 0.649563]
Iter 50. [Val Acc 71%, Loss 0.623469]
Iter 60. [Val Acc 70%, Loss 0.643882]
Iter 70. [Val Acc 70%, Loss 0.612335]
Iter 80. [Val Acc 69%, Loss 0.693272]
Iter 90. [Val Acc 71%, Loss 0.619433]
Iter 100. [Val Acc 71%, Loss 0.607791]
Iter 110. [Val Acc 71%, Loss 0.628378]
Iter 120. [Val Acc 70%, Loss 0.650324]
Iter 130. [Val Acc 69%, Loss 0.641664]
Iter 140. [Val Acc 69%, Loss 0.673244]
Iter 150. [Val Acc 70%, Loss 0.638661]
Iter 160. [Val Acc 71%, Loss 0.608008]
Iter 170. [Val Acc 70%, Loss 0.641026]
Iter 180. [Val Acc 71%, Loss 0.618061]
Iter 190. [Val Acc 72%, Loss 0.604159]
Iter 200. [Val Acc 71%, Loss 0.625646]
```

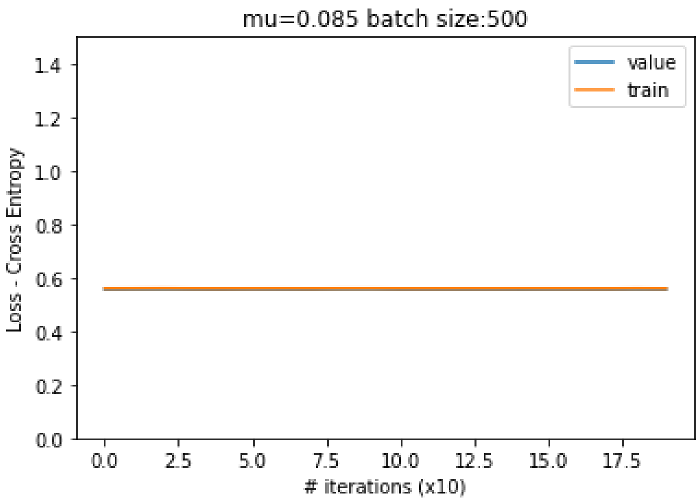






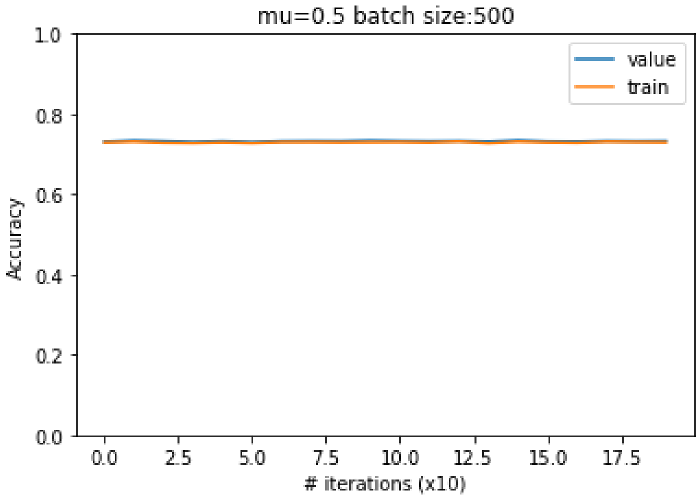

```
Iter 10. [Val Acc 74%, Loss 0.556594]
Iter 20. [Val Acc 74%, Loss 0.556940]
Iter 30. [Val Acc 74%, Loss 0.557213]
Iter 40. [Val Acc 74%, Loss 0.556677]
Iter 50. [Val Acc 74%, Loss 0.556377]
Iter 60. [Val Acc 74%, Loss 0.556892]
Iter 70. [Val Acc 74%, Loss 0.556754]
Iter 80. [Val Acc 74%, Loss 0.556431]
Iter 90. [Val Acc 74%, Loss 0.557268]
Iter 100. [Val Acc 74%, Loss 0.557357]
Iter 110. [Val Acc 74%, Loss 0.556481]
Iter 120. [Val Acc 74%, Loss 0.556539]
Iter 130. [Val Acc 74%, Loss 0.556550]
Iter 140. [Val Acc 74%, Loss 0.556552]
Iter 150. [Val Acc 74%, Loss 0.556480]
Iter 160. [Val Acc 74%, Loss 0.556755]
Iter 170. [Val Acc 74%, Loss 0.556585]
Iter 180. [Val Acc 74%, Loss 0.556430]
Iter 190. [Val Acc 74%, Loss 0.556989]
Iter 200. [Val Acc 74%, Loss 0.556617]
```

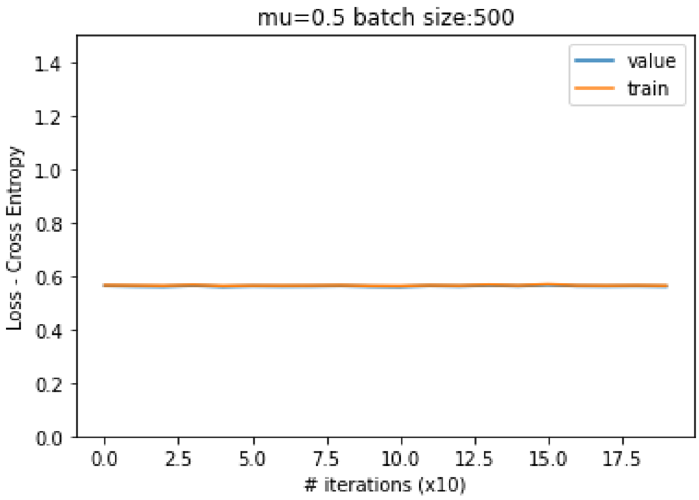




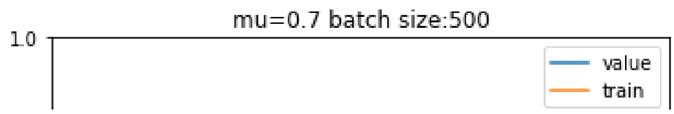

```
Iter 10. [Val Acc 73%, Loss 0.562498]
Iter 20. [Val Acc 73%, Loss 0.560652]
Iter 30. [Val Acc 73%, Loss 0.559485]
Iter 40. [Val Acc 73%, Loss 0.563096]
Iter 50. [Val Acc 73%, Loss 0.559094]
Iter 60. [Val Acc 73%, Loss 0.561259]
Iter 70. [Val Acc 73%, Loss 0.560490]
Iter 80. [Val Acc 73%, Loss 0.560873]
```

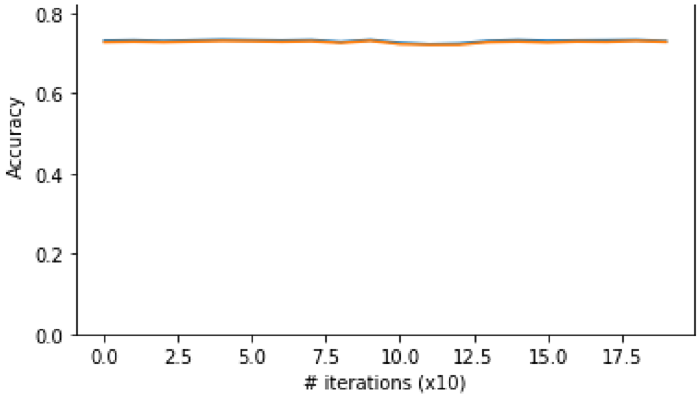
```
Iter 90. [Val Acc 73%, Loss 0.562237]
Iter 100. [Val Acc 73%, Loss 0.559844]
Iter 110. [Val Acc 73%, Loss 0.558837]
Iter 120. [Val Acc 73%, Loss 0.562370]
Iter 130. [Val Acc 73%, Loss 0.560763]
Iter 140. [Val Acc 73%, Loss 0.563782]
Iter 150. [Val Acc 74%, Loss 0.561272]
Iter 160. [Val Acc 73%, Loss 0.565264]
Iter 170. [Val Acc 73%, Loss 0.561435]
Iter 180. [Val Acc 73%, Loss 0.560826]
Iter 190. [Val Acc 73%, Loss 0.561553]
Iter 200. [Val Acc 73%, Loss 0.560317]
```



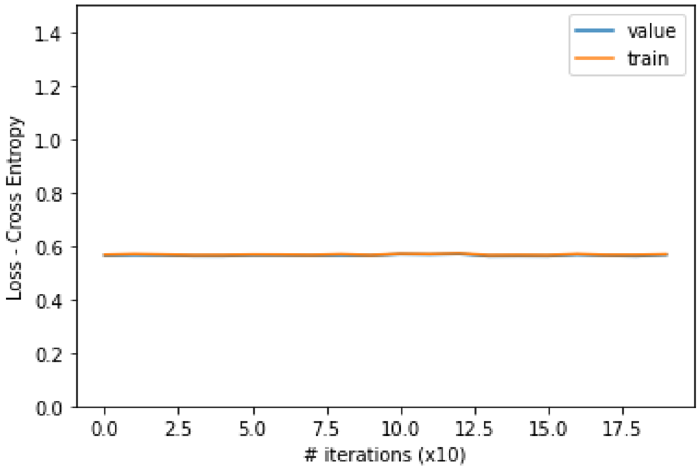



```
Iter 10. [Val Acc 73%, Loss 0.563488]
Iter 20. [Val Acc 73%, Loss 0.565192]
Iter 30. [Val Acc 73%, Loss 0.564315]
Iter 40. [Val Acc 73%, Loss 0.561941]
Iter 50. [Val Acc 73%, Loss 0.561892]
Iter 60. [Val Acc 73%, Loss 0.564310]
Iter 70. [Val Acc 73%, Loss 0.564096]
Iter 80. [Val Acc 73%, Loss 0.562974]
Iter 90. [Val Acc 73%, Loss 0.564433]
Iter 100. [Val Acc 73%, Loss 0.562866]
Iter 110. [Val Acc 73%, Loss 0.568731]
Iter 120. [Val Acc 72%, Loss 0.567438]
Iter 130. [Val Acc 72%, Loss 0.569575]
Iter 140. [Val Acc 73%, Loss 0.561532]
Iter 150. [Val Acc 73%, Loss 0.562084]
Iter 160. [Val Acc 73%, Loss 0.561697]
Iter 170. [Val Acc 73%, Loss 0.566389]
Iter 180. [Val Acc 73%, Loss 0.563139]
Iter 190. [Val Acc 73%, Loss 0.561630]
Iter 200. [Val Acc 73%, Loss 0.565984]
```



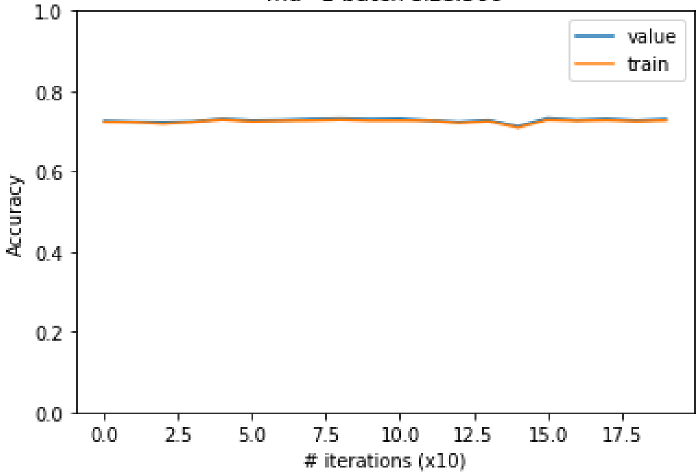


mu=0.7 batch size:500

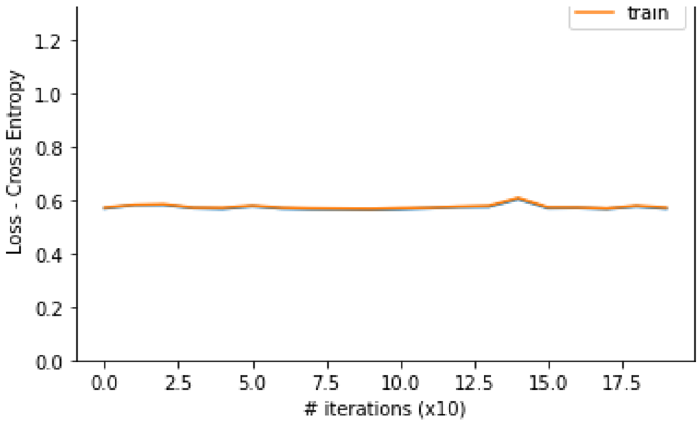



```
Iter 10. [Val Acc 73%, Loss 0.567660]
Iter 20. [Val Acc 72%, Loss 0.579587]
Iter 30. [Val Acc 72%, Loss 0.579853]
Iter 40. [Val Acc 72%, Loss 0.568961]
Iter 50. [Val Acc 73%, Loss 0.566058]
Iter 60. [Val Acc 73%, Loss 0.575509]
Iter 70. [Val Acc 73%, Loss 0.567189]
Iter 80. [Val Acc 73%, Loss 0.564989]
Iter 90. [Val Acc 73%, Loss 0.564234]
Iter 100. [Val Acc 73%, Loss 0.562729]
Iter 110. [Val Acc 73%, Loss 0.564960]
Iter 120. [Val Acc 73%, Loss 0.568598]
Iter 130. [Val Acc 72%, Loss 0.572793]
Iter 140. [Val Acc 73%, Loss 0.573616]
Iter 150. [Val Acc 71%, Loss 0.602264]
Iter 160. [Val Acc 73%, Loss 0.568471]
Iter 170. [Val Acc 73%, Loss 0.569589]
Iter 180. [Val Acc 73%, Loss 0.565771]
Iter 190. [Val Acc 73%, Loss 0.574753]
Iter 200. [Val Acc 73%, Loss 0.566927]
```

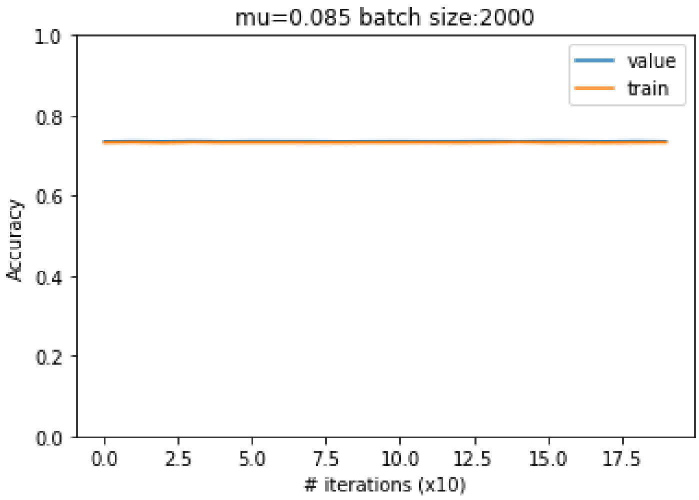
mu=1 batch size:500

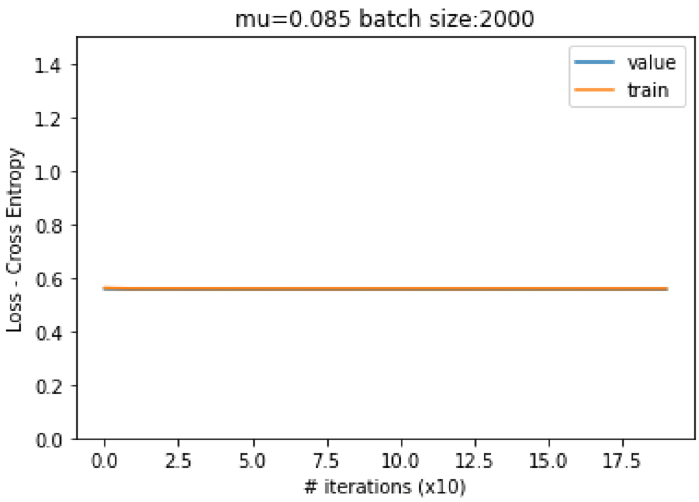


mu=1 batch size:500



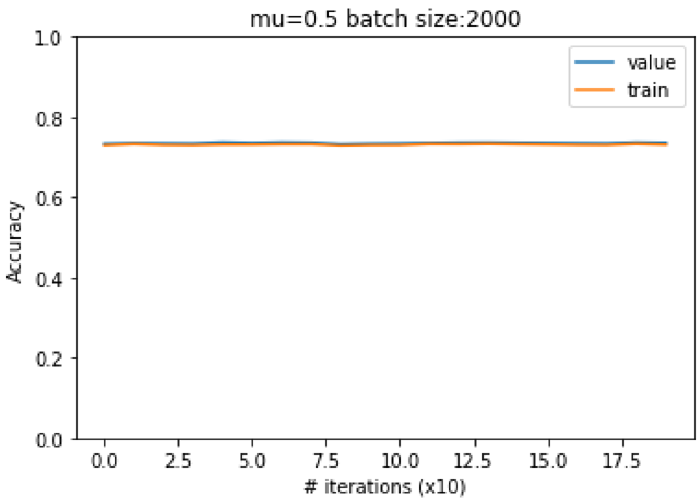

```
Iter 10. [Val Acc 74%, Loss 0.558037]
Iter 20. [Val Acc 74%, Loss 0.556020]
Iter 30. [Val Acc 74%, Loss 0.556283]
Iter 40. [Val Acc 74%, Loss 0.556160]
Iter 50. [Val Acc 74%, Loss 0.556250]
Iter 60. [Val Acc 74%, Loss 0.556030]
Iter 70. [Val Acc 74%, Loss 0.556126]
Iter 80. [Val Acc 74%, Loss 0.556055]
Iter 90. [Val Acc 74%, Loss 0.556019]
Iter 100. [Val Acc 74%, Loss 0.556209]
Iter 110. [Val Acc 74%, Loss 0.556145]
Iter 120. [Val Acc 74%, Loss 0.556058]
Iter 130. [Val Acc 74%, Loss 0.556057]
Iter 140. [Val Acc 74%, Loss 0.556097]
Iter 150. [Val Acc 74%, Loss 0.556172]
Iter 160. [Val Acc 74%, Loss 0.556048]
Iter 170. [Val Acc 74%, Loss 0.556243]
Iter 180. [Val Acc 74%, Loss 0.556154]
Iter 190. [Val Acc 74%, Loss 0.556197]
Iter 200. [Val Acc 74%, Loss 0.556161]
```

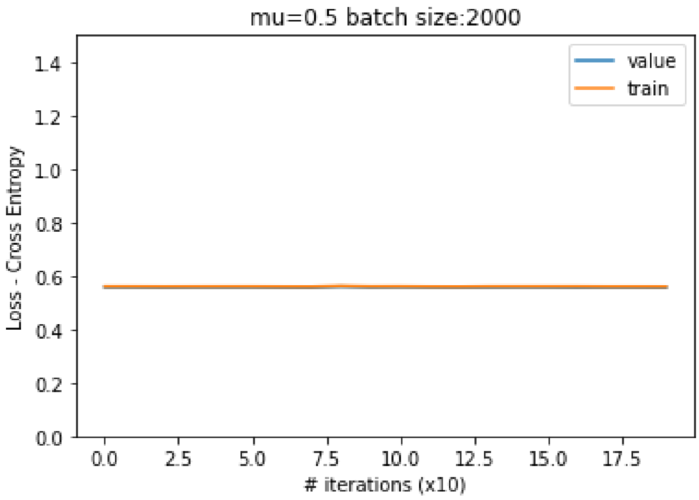





```
Iter 10. [Val Acc 73%, Loss 0.557619]
Iter 20. [Val Acc 74%, Loss 0.557337]
```

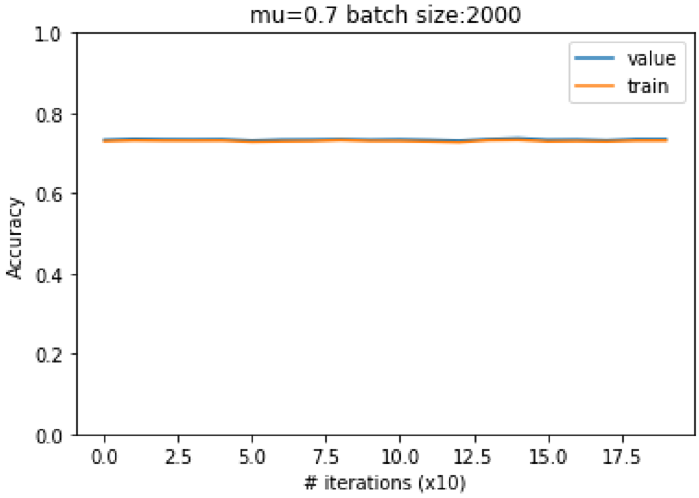
```
Iter 30. [Val Acc 73%, Loss 0.556847]
Iter 40. [Val Acc 73%, Loss 0.557273]
Iter 50. [Val Acc 74%, Loss 0.557246]
Iter 60. [Val Acc 74%, Loss 0.557306]
Iter 70. [Val Acc 74%, Loss 0.556810]
Iter 80. [Val Acc 74%, Loss 0.556546]
Iter 90. [Val Acc 73%, Loss 0.559647]
Iter 100. [Val Acc 73%, Loss 0.557356]
Iter 110. [Val Acc 73%, Loss 0.557841]
Iter 120. [Val Acc 74%, Loss 0.557072]
Iter 130. [Val Acc 74%, Loss 0.556910]
Iter 140. [Val Acc 74%, Loss 0.557120]
Iter 150. [Val Acc 74%, Loss 0.557444]
Iter 160. [Val Acc 74%, Loss 0.557272]
Iter 170. [Val Acc 74%, Loss 0.556764]
Iter 180. [Val Acc 73%, Loss 0.557002]
Iter 190. [Val Acc 74%, Loss 0.556935]
Iter 200. [Val Acc 74%, Loss 0.556626]
```

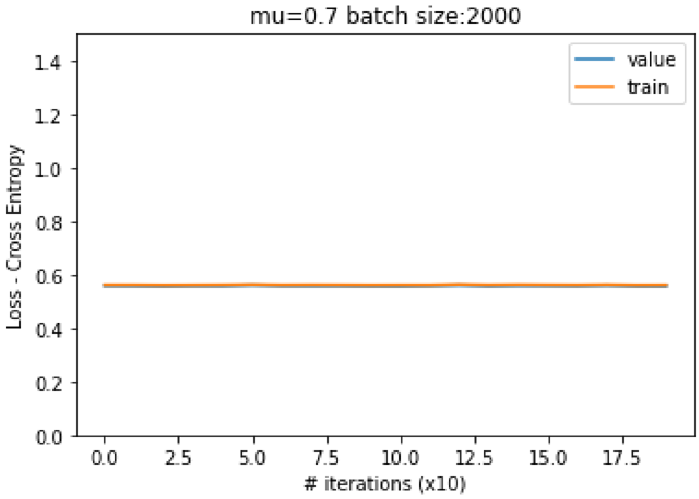




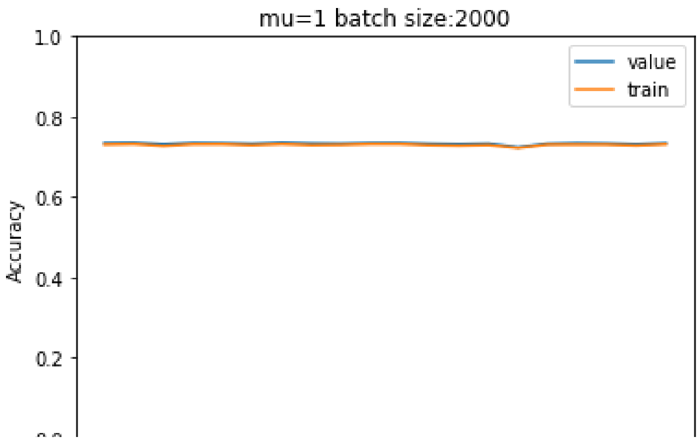

```
Iter 10. [Val Acc 73%, Loss 0.558321]
Iter 20. [Val Acc 74%, Loss 0.557926]
Iter 30. [Val Acc 74%, Loss 0.557176]
Iter 40. [Val Acc 73%, Loss 0.558019]
Iter 50. [Val Acc 74%, Loss 0.557867]
Iter 60. [Val Acc 73%, Loss 0.560106]
Iter 70. [Val Acc 73%, Loss 0.558066]
Iter 80. [Val Acc 73%, Loss 0.557954]
Iter 90. [Val Acc 74%, Loss 0.557947]
Iter 100. [Val Acc 73%, Loss 0.557400]
Iter 110. [Val Acc 73%, Loss 0.557251]
Iter 120. [Val Acc 73%, Loss 0.558055]
Iter 130. [Val Acc 73%, Loss 0.559951]
Iter 140. [Val Acc 74%, Loss 0.557547]
Iter 150. [Val Acc 74%, Loss 0.558726]
Iter 160. [Val Acc 73%, Loss 0.558343]
Iter 170. [Val Acc 73%, Loss 0.557837]
Iter 180. [Val Acc 73%, Loss 0.559338]
Iter 190. [Val Acc 74%, Loss 0.557386]
```

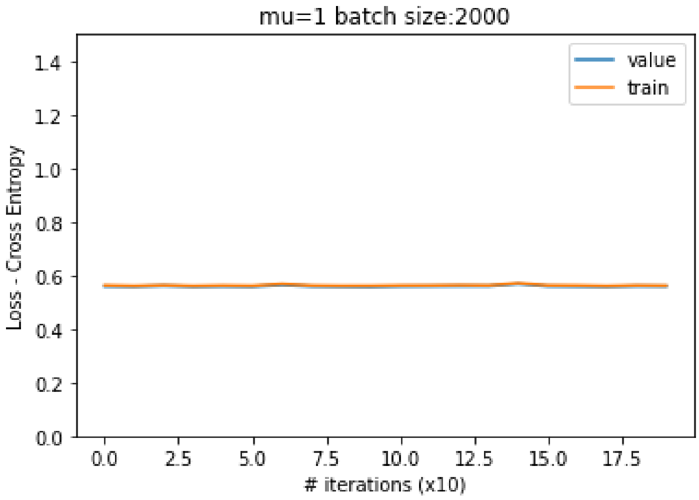
```
Iter 190. [Val Acc 74%, Loss 0.557380]
Iter 200. [Val Acc 74%, Loss 0.557281]
```



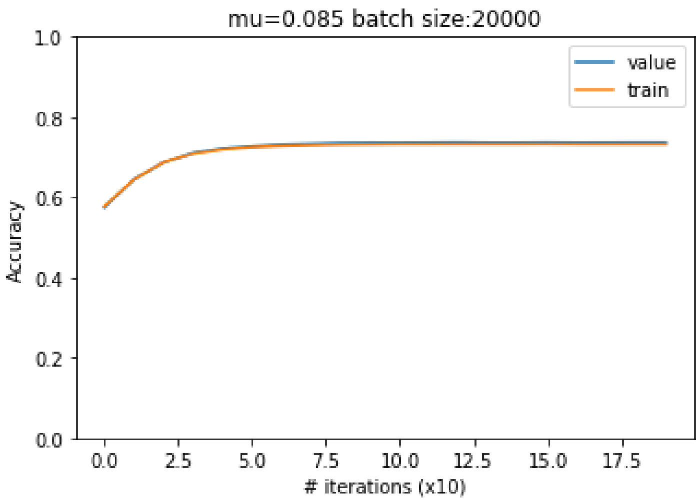


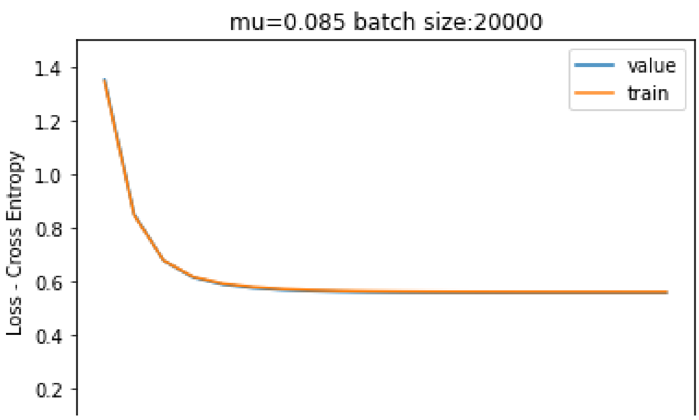

```
Iter 10. [Val Acc 73%, Loss 0.559365]
Iter 20. [Val Acc 74%, Loss 0.558406]
Iter 30. [Val Acc 73%, Loss 0.561009]
Iter 40. [Val Acc 73%, Loss 0.558215]
Iter 50. [Val Acc 73%, Loss 0.559236]
Iter 60. [Val Acc 73%, Loss 0.558224]
Iter 70. [Val Acc 74%, Loss 0.564164]
Iter 80. [Val Acc 73%, Loss 0.559261]
Iter 90. [Val Acc 73%, Loss 0.558308]
Iter 100. [Val Acc 73%, Loss 0.557640]
Iter 110. [Val Acc 73%, Loss 0.559375]
Iter 120. [Val Acc 73%, Loss 0.559759]
Iter 130. [Val Acc 73%, Loss 0.560213]
Iter 140. [Val Acc 73%, Loss 0.560191]
Iter 150. [Val Acc 72%, Loss 0.567952]
Iter 160. [Val Acc 73%, Loss 0.559680]
Iter 170. [Val Acc 73%, Loss 0.558897]
Iter 180. [Val Acc 73%, Loss 0.558126]
Iter 190. [Val Acc 73%, Loss 0.559920]
Iter 200. [Val Acc 73%, Loss 0.559557]
```

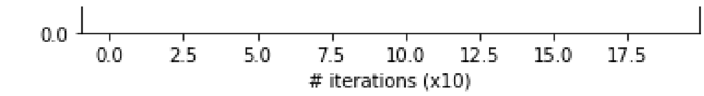




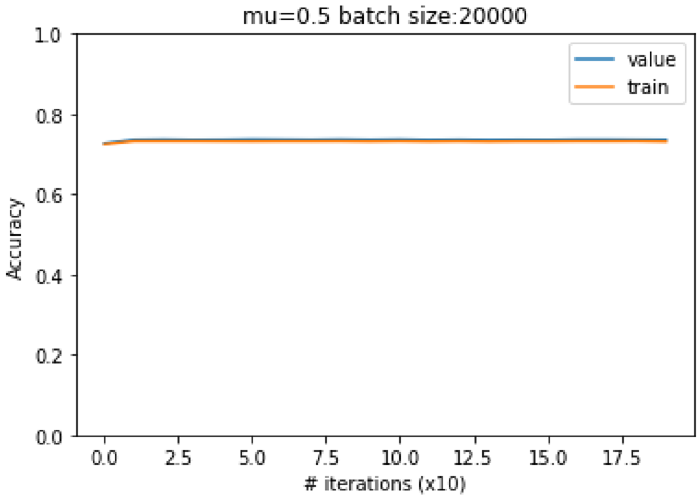

```
Iter 10. [Val Acc 57%, Loss 1.351139]
Iter 20. [Val Acc 64%, Loss 0.848494]
Iter 30. [Val Acc 69%, Loss 0.675021]
Iter 40. [Val Acc 71%, Loss 0.612569]
Iter 50. [Val Acc 72%, Loss 0.586461]
Iter 60. [Val Acc 73%, Loss 0.573556]
Iter 70. [Val Acc 73%, Loss 0.566397]
Iter 80. [Val Acc 73%, Loss 0.562308]
Iter 90. [Val Acc 74%, Loss 0.559776]
Iter 100. [Val Acc 74%, Loss 0.558306]
Iter 110. [Val Acc 74%, Loss 0.557387]
Iter 120. [Val Acc 74%, Loss 0.556889]
Iter 130. [Val Acc 74%, Loss 0.556532]
Iter 140. [Val Acc 74%, Loss 0.556336]
Iter 150. [Val Acc 74%, Loss 0.556234]
Iter 160. [Val Acc 74%, Loss 0.556123]
Iter 170. [Val Acc 74%, Loss 0.556085]
Iter 180. [Val Acc 74%, Loss 0.556068]
Iter 190. [Val Acc 74%, Loss 0.556032]
Iter 200. [Val Acc 74%, Loss 0.556024]
```

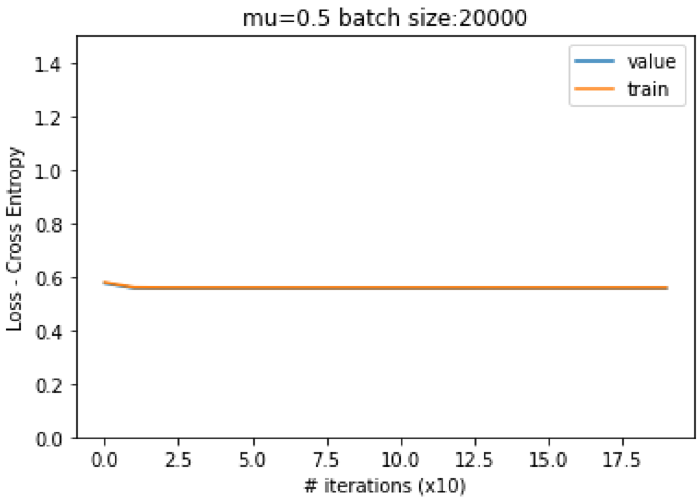






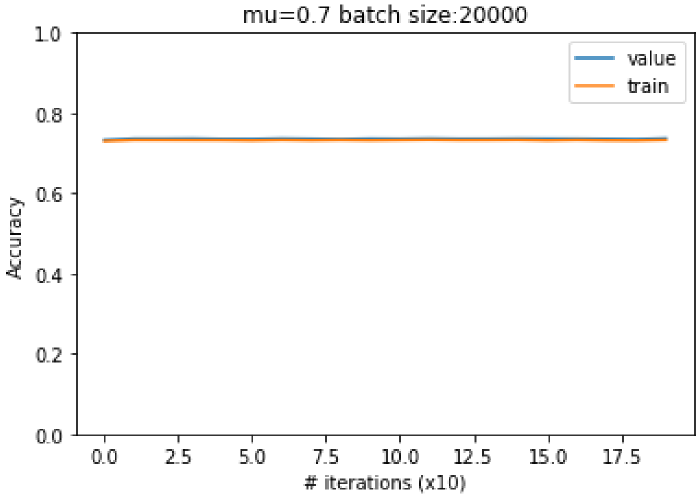

```
Iter 10. [Val Acc 73%, Loss 0.574624]
Iter 20. [Val Acc 74%, Loss 0.556974]
Iter 30. [Val Acc 74%, Loss 0.556082]
Iter 40. [Val Acc 74%, Loss 0.556140]
Iter 50. [Val Acc 74%, Loss 0.556085]
Iter 60. [Val Acc 74%, Loss 0.556170]
Iter 70. [Val Acc 74%, Loss 0.556100]
Iter 80. [Val Acc 74%, Loss 0.555963]
Iter 90. [Val Acc 74%, Loss 0.556114]
Iter 100. [Val Acc 74%, Loss 0.556025]
Iter 110. [Val Acc 74%, Loss 0.556050]
Iter 120. [Val Acc 74%, Loss 0.556139]
Iter 130. [Val Acc 74%, Loss 0.556005]
Iter 140. [Val Acc 74%, Loss 0.556228]
Iter 150. [Val Acc 74%, Loss 0.556105]
Iter 160. [Val Acc 74%, Loss 0.556255]
Iter 170. [Val Acc 74%, Loss 0.555998]
Iter 180. [Val Acc 74%, Loss 0.556018]
Iter 190. [Val Acc 74%, Loss 0.556037]
Iter 200. [Val Acc 74%, Loss 0.556105]
```

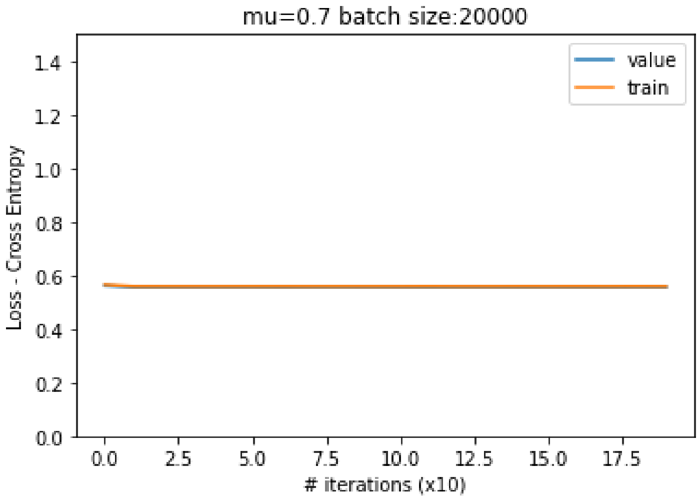




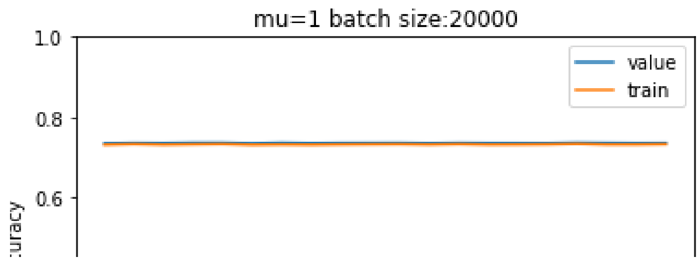

```
Iter 10. [Val Acc 73%, Loss 0.561805]
Iter 20. [Val Acc 74%, Loss 0.556077]
Iter 30. [Val Acc 74%, Loss 0.556114]
Iter 40. [Val Acc 74%, Loss 0.556044]
Iter 50. [Val Acc 74%, Loss 0.556109]
Iter 60. [Val Acc 74%, Loss 0.556248]
Iter 70. [Val Acc 74%, Loss 0.556012]
Iter 80. [Val Acc 74%, Loss 0.556199]
Iter 90. [Val Acc 74%, Loss 0.556257]
Iter 100. [Val Acc 74%, Loss 0.556166]
Iter 110. [Val Acc 74%, Loss 0.556046]
Iter 120. [Val Acc 74%, Loss 0.556124]
Iter 130. [Val Acc 74%, Loss 0.556125]
```

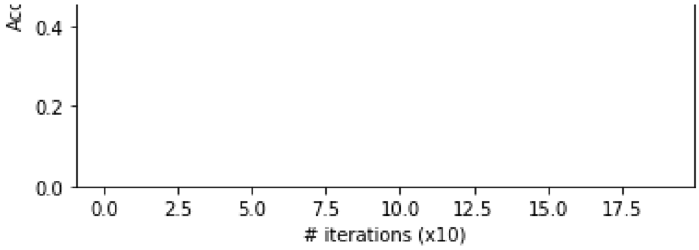
```
Iter 140. [Val Acc 74%, Loss 0.556099]
Iter 150. [Val Acc 74%, Loss 0.556083]
Iter 160. [Val Acc 74%, Loss 0.556176]
Iter 170. [Val Acc 74%, Loss 0.556097]
Iter 180. [Val Acc 74%, Loss 0.556076]
Iter 190. [Val Acc 74%, Loss 0.556479]
Iter 200. [Val Acc 74%, Loss 0.556191]
```

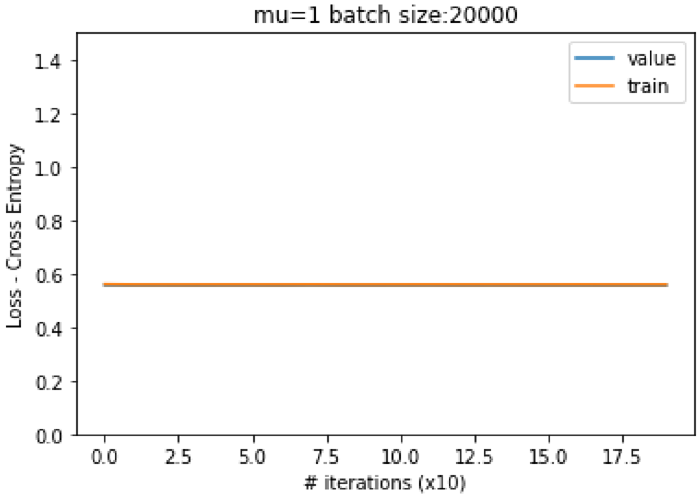





```
Iter 10. [Val Acc 74%, Loss 0.557174]
Iter 20. [Val Acc 74%, Loss 0.556288]
Iter 30. [Val Acc 74%, Loss 0.556237]
Iter 40. [Val Acc 74%, Loss 0.556020]
Iter 50. [Val Acc 74%, Loss 0.556263]
Iter 60. [Val Acc 74%, Loss 0.556350]
Iter 70. [Val Acc 74%, Loss 0.556080]
Iter 80. [Val Acc 74%, Loss 0.556759]
Iter 90. [Val Acc 74%, Loss 0.556210]
Iter 100. [Val Acc 74%, Loss 0.556183]
Iter 110. [Val Acc 74%, Loss 0.556239]
Iter 120. [Val Acc 74%, Loss 0.556511]
Iter 130. [Val Acc 74%, Loss 0.556217]
Iter 140. [Val Acc 74%, Loss 0.556343]
Iter 150. [Val Acc 74%, Loss 0.556205]
Iter 160. [Val Acc 74%, Loss 0.556107]
Iter 170. [Val Acc 74%, Loss 0.556378]
Iter 180. [Val Acc 74%, Loss 0.556075]
Iter 190. [Val Acc 74%, Loss 0.556248]
Iter 200. [Val Acc 74%, Loss 0.556225]
```








```
Iter 10. [Val Acc 57%, Loss 1.351215]
Iter 20. [Val Acc 64%, Loss 0.848292]
Iter 30. [Val Acc 69%, Loss 0.674930]
Iter 40. [Val Acc 71%, Loss 0.612587]
Iter 50. [Val Acc 72%, Loss 0.586497]
Iter 60. [Val Acc 73%, Loss 0.573532]
Iter 70. [Val Acc 73%, Loss 0.566403]
Iter 80. [Val Acc 73%, Loss 0.562297]
Iter 90. [Val Acc 73%, Loss 0.559800]
Iter 100. [Val Acc 74%, Loss 0.558284]
```

In [30]:

```
w_opt
```

Out[30]:

```
array([ 1.32065976e+00, -8.93504382e-01, -2.04117660e-01, -4.47951001e-02,
       -9.70920740e-02, -6.50344624e-01, -4.11712698e-02, -1.75271566e-01,
       -1.80832061e-01,  2.44163691e-02, -2.45451103e-01,  5.99244280e-02,
        2.15558953e-01,  1.72333863e-01, -5.04861157e-02,  1.47837608e-01,
        1.53374162e-03,  2.82519021e-01,  1.39882595e-01,  1.69299378e-01,
        3.43677564e-02,  5.89445419e-02,  3.07624009e-01,  1.00302722e-01,
       -1.33085521e-01,  3.33750993e-02,  1.44056526e-01,  1.85899148e-02,
        2.47867672e-03,  3.95145569e-02, -6.62252038e-03, -1.56413777e-02,
       -9.55763434e-02,  3.85489016e-02,  2.98053533e-03, -1.08597754e-01,
       -3.97872629e-02,  8.92411540e-02,  8.45080428e-02, -5.17545124e-02,
       -1.01724369e-01, -3.44760649e-02,  4.18775752e-03, -2.70354156e-02,
        2.00579095e-03,  5.66414914e-02,  2.64492602e-02, -1.10523464e-01,
        1.84548751e-02, -5.53329128e-03,  6.46993129e-03, -2.60179116e-02,
        6.74736430e-02, -2.26060172e-02, -5.68901364e-02, -1.80617375e-03,
       -1.40427449e-01,  9.65473398e-02, -4.27696975e-02, -3.56535936e-02,
       -3.75267292e-02, -2.36833655e-02, -1.27911359e-01,  9.41383374e-02,
       -1.31564726e-01, -5.51634232e-03,  8.20521317e-03,  3.60616819e-02,
       -9.80075140e-02, -1.83477991e-02, -5.79785281e-02,  1.69738671e-03,
        2.59936628e-02,  6.47010270e-02,  2.96109639e-02,  6.47495871e-02,
        4.30635949e-03, -1.09372204e-01, -6.20325902e-02, -5.43779593e-03,
       -1.87468309e-02,  1.56162566e-02,  1.74625023e-02,  6.74059510e-04,
        7.67093563e-02, -2.38909679e-02,  6.67403347e-02, -1.72826097e-01,
       -4.33835316e-02,  9.64986862e-03])
```

In [31]:

```
b_opt
```

Out[31]:

```
0.2897208078292643
```

Explain and discuss your results here:

Learining rate: As it can be seen in part(f) a too large rate causes the model to be very noisy and not to converge. In contrary, a too small rate causes the model to converge very slowly. Therefore, rates in between the rates choesn in last section were chosen in order to find the optimal learinig rate for our model.

Moreover, in this section the intial values of the weights and the bias are randomly initialized and not zero like in the last section. This may cause the low learning rate to be too low in order to converge, even if in genaral the learning rate is already higher than in the previous section. for this reason, we enlarged our iterations to 200.

Batch size: The role of the batch size is to control the accuracy of the estimate of the error gradient when training the model. There is a tradeoff between speed and stability of the learning process as the batch size varies. The purpose of the model is to estimate the weights and bias with help of SGD. However, according to the figures it can be seen that this estimation is often noisy (mostly according to the train curve). This variance of estimation is reduced by increasing the batch size. Meaning the model becomes more stable (the loss values are decreasing). In addtion to that, the presence of noise (as part of stochastic tend of SGD) helps to escape plateaus. Therefore the batch size is gradually increased for all values of learning rates we tested.

While the test we observed that as the batch size increases the noise that can be seen by the train curve is decreasing, therefore it is important not increase the batch size too much. Morover increasing it too much might also increase the runtime dramatically.

As the batch size was increased it can be seen that a smaller learning rate value is getting better results.

In summary the optimual learning rate that was chosen is: 0.085 with batchsize 20000. According to the values of the loss and the accuracy, by the above chosen rate and batch size the best result were accomplished.

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?

In [27]:

```
#Rerun the model with optimal values found for later use:

w_opt, b_opt ,val_costs, val_accs ,train_costs, train_accs = run_gradient_descent(train_
norm_xs,train_ts,val_norm_xs,val_ts,w0,b0,0.085,20000, max_iters=300)

y_train = pred(w_opt, b_opt, train_norm_xs)
train_acc = get_accuracy(y_train,train_ts)

y_val = pred(w_opt, b_opt, val_norm_xs)
val_acc = get_accuracy(y_val,val_ts)

y_test = pred(w_opt, b_opt, test_norm_xs)
test_acc = get_accuracy(y_test,test_ts)

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

```
Iter 10. [Val Acc 57%, Loss 1.351348]
Iter 20. [Val Acc 64%, Loss 0.848349]
Iter 30. [Val Acc 69%, Loss 0.674893]
Iter 40. [Val Acc 71%, Loss 0.612552]
Iter 50. [Val Acc 72%, Loss 0.586459]
Iter 60. [Val Acc 73%, Loss 0.573540]
Iter 70. [Val Acc 73%, Loss 0.566412]
Iter 80. [Val Acc 73%, Loss 0.562251]
Iter 90. [Val Acc 73%, Loss 0.559782]
Iter 100. [Val Acc 74%, Loss 0.558301]
Iter 110. [Val Acc 74%, Loss 0.557435]
Iter 120. [Val Acc 74%, Loss 0.556880]
Iter 130. [Val Acc 74%, Loss 0.556524]
Iter 140. [Val Acc 74%, Loss 0.556338]
```

```
Iter 150. [Val Acc 74%, Loss 0.556200]
Iter 160. [Val Acc 74%, Loss 0.556135]
Iter 170. [Val Acc 74%, Loss 0.556066]
Iter 180. [Val Acc 74%, Loss 0.556066]
Iter 190. [Val Acc 74%, Loss 0.556028]
Iter 200. [Val Acc 74%, Loss 0.556033]
Iter 210. [Val Acc 74%, Loss 0.556020]
Iter 220. [Val Acc 74%, Loss 0.556014]
Iter 230. [Val Acc 74%, Loss 0.556011]
Iter 240. [Val Acc 74%, Loss 0.556000]
Iter 250. [Val Acc 74%, Loss 0.556034]
Iter 260. [Val Acc 74%, Loss 0.556000]
Iter 270. [Val Acc 74%, Loss 0.556017]
Iter 280. [Val Acc 74%, Loss 0.556004]
Iter 290. [Val Acc 74%, Loss 0.555993]
Iter 300. [Val Acc 74%, Loss 0.556005]
train_acc =  0.7322262910457682  val_acc =  0.73632  test_acc =  0.726283168700368
```

Explain and discuss your results here:

There are small differences that can be seen in accuracy between the tree groups. The optimization values for the weights and the bias were chosen according to the trial and error we did in the previous section. The accuracy of the vallidation set is the best. While training the validation set was not used, therefore the better accuracy of the train set might implicate some overfitting. The test set and the train set are independet, as a result the accuracy of the train set is bit lower than the one of the train set. In general, all three groups have similar results which shows that our model managed to learn to classify songs according to release date. Therefore, also unlabled songs are expected to be classified in the right category with the accuracy of test set (72.6457).

Our results:

train_acc = 0.7322262910457682

val_acc = 0.73632

test_acc = 0.726283168700368

Sklearn results:

train_acc = 0.7323979067715698

val_acc = 0.73644

test_acc = 0.7265736974627155

The results of two models are similar.

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.

In [29]:

```
import sklearn.linear_model as sk

model = sk.LogisticRegression(max_iter =300,multi_class = 'multinomial').fit(train_norm_x
s,np.ravel(train_ts))
#multinomial is for SAGD
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)
```

```
train_acc =  0.7323979067715698  val_acc =  0.73644  test_acc =  0.7265736974627155
```

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