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.

In [1]:

```
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

In [2]:

```
load_from_drive = True

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', force_remount=True)
  csv_path = '/content/gdrive/My Drive/deep_learning/YearPredictionMSD.txt.zip' # TODO -
UPDATE ME WITH THE TRUE PATH!

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)
```

```
Mounted at /content/gdrive
```

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

In [3]:

```
df
```

Out[3]:

	year	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10	var11	
0	2001	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12.23257	7.83089	-2.46783	3.32136	2.
1	2001	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0.92019	18.76548	4.59210	2.21920	0.
2	2001	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2.35035	16.07017	1.39518	2.73553	0.
3	2001	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10.34124	3.55005	-6.36304	6.63016	3.
4	2001	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9.37636	12.63699	0.93609	1.60923	2.
...	...	...	...	...	...	...	...	...	...	...	...	...	
515340	2006	51.28467	45.88068	22.19582	-5.53319	-3.61835	-16.36914	2.12652	5.18160	-8.66890	2.67217	0.45234	2.
515341	2006	49.87870	37.93125	18.65987	-3.63581	-27.75665	-18.52988	7.76108	3.56109	-2.50351	2.20175	-0.58487	9.
515342	2006	45.12852	12.65758	-38.72018	8.80882	-29.29985	-2.28706	-18.40424	-22.28726	-4.52429	-11.46411	3.28514	1.
515343	2006	44.16614	32.38368	-3.34971	-2.49165	-19.59278	-18.67098	8.78428	4.02039	-12.01230	-0.74075	-1.26523	4.
515344	2005	51.85726	59.11655	26.39436	-5.46030	-20.69012	-19.95528	-6.72771	2.29590	10.31018	6.26597	-1.78800	6.

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.

In [4]:

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

In [5]:

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

Out[5]:

	year	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10	var11	va
0	1	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12.23257	7.83089	-2.46783	3.32136	-2.31
1	1	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0.92019	18.76548	4.59210	2.21920	0.34
2	1	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2.35035	16.07017	1.39518	2.73553	0.82
3	1	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10.34124	3.55005	-6.36304	6.63016	-3.35
4	1	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9.37636	12.63699	0.93609	1.60923	2.19

	year	var1	var2	var3	var4	var5	var6	var7	var8	var9	var10	var11	va
5	1	50.54767	0.31568	92.35066	22.38696	-25.51870	-19.04928	20.67345	-5.19943	3.63566	-4.69088	2.49578	-3.02
6	1	50.57546	33.17843	50.53517	11.55217	-27.24764	-8.78206	-12.04282	-9.53930	28.61811	8.25435	-0.43743	5.66
7	1	48.26892	8.97526	75.23158	24.04945	-16.02105	-14.09491	8.11871	-1.87566	7.46701	1.18189	1.46625	-6.34
8	1	49.75468	33.99581	56.73846	2.89581	-2.92429	-26.44413	1.71392	-0.55644	22.08594	7.43847	-0.03578	1.66
9	1	45.17809	46.34234	-40.65357	-2.47909	1.21253	-0.65302	-6.95536	-12.20040	17.02512	2.00002	-1.87785	9.85
10	1	39.13076	-23.01763	-36.20583	1.67519	-4.27101	13.01158	8.05718	-8.41088	6.27370	-7.81564	-12.29472	12.26
11	1	37.66498	-34.05910	-17.36060	-26.77781	-39.95119	-20.75000	-0.10231	-0.89972	-1.30205	-0.93041	-3.30157	-2.37
12	1	26.51957	-148.15762	-13.30095	-7.25851	17.22029	-21.99439	5.51947	3.48418	2.61738	2.51194	-0.53980	4.94
13	1	37.68491	-26.84185	-27.10566	-14.95883	-5.87200	-21.68979	4.87374	-18.01800	1.52141	-6.81668	6.80117	21.17
14	0	39.11695	-8.29767	-51.37966	-4.42668	-30.06506	-11.95916	-0.85322	-8.86179	11.36680	3.78199	1.54568	0.58
15	1	35.05129	-67.97714	-14.20239	-6.68696	-0.61230	-18.70341	-1.31928	-9.46370	5.53492	2.79989	3.34150	18.01
16	1	33.63129	-96.14912	-89.38216	-12.11699	13.77252	-6.69377	-33.36843	-24.81437	21.22757	0.26310	0.42982	-6.59
17	0	41.38639	-20.78665	51.80155	17.21415	-36.44189	-11.53169	11.75252	-7.62428	-3.65488	-5.08109	4.37624	-1.39
18	0	37.45034	11.42615	56.28982	19.58426	-16.43530	2.22457	1.02668	-7.34736	-0.01184	1.24013	2.57660	2.79
19	0	39.71092	-4.92800	12.88590	-11.87773	2.48031	-16.11028	-16.40421	-8.29657	9.86817	-0.17431	0.98765	7.37

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.)

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()

# Write your explanation here
# If some songs from an artist were to appear in both the training set and the test set, our test would not
# reflect the model's success accurately because a problem of overfitting would arise, as the mapping
# partially memorizes the training data. We therefore need to induct a bias, in this case we induct a
```

```
# bias of artists whose works appear only in the training set.
```

Part (b) -- 7%

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

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
```

Notice 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.)

In [8]:

```
# Since we do not have access to the true distribution, we can only access the distributi
on
# of the training set. By the law of large numbers, we can compute the empirical risk wit
h
# hopes that it converges to the true risk.
```

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.

In [9]:

```
# 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]

# We should limit how many times we use the test set because the test set should be
# unbiased. If we use the test set too many times and fit our model to it, it would
# become biased.
# In order to test the model without using the test set, we use the validation set in ord
er
# to tune the hyperparameters of the classifier.
```

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).

In [10]:

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

def cross_entropy(t, y):
  result = np.empty_like(y)
  for i in range(len(y)):
    if (0 < y[i] < 1):
      result[i] = -t[i] * np.log(y[i]) - (1 - t[i]) * np.log(1 - y[i])
    else:
      result[i] = y[i]
  return result

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 $\text{year} = 1$.

In [11]:

```python
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
  """
  return sigmoid(np.inner(w,X) + b)
```

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 [12]:

```python
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
    """

    dLdB = y-t
    dLdb = np.mean(dLdB)

    dLdw = np.mean(np.transpose([dLdB])*X,axis=0)

    return (dLdw,dLdb)
```

Explanation 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.]()

$$\frac{\partial y}{\partial \mathbf{b}} = y(1-y)$$

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

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 implemented correctly by comparing the result from `derivative_cost` with the empirical cost derivative computed using the above numerical approximation.

In [13]:

```
# Test Case

h = 0.0001
X = np.random.rand(2,90);
t = np.random.rand(2);
w = np.random.rand(90)
b = np.random.rand(1)

dLdB_algo = (np.mean(cross_entropy(t,pred(w,b + h,X)) - cross_entropy(t,pred(w,b,X))))/h

y = pred(w,b,X)

print("The analytical result is - ", dLdB_algo)
print("The algorithm result is - ", derivative_cost(X,y,t)[1])
```

```
The analytical result is -  0.3149694199322539
The algorithm result is -  0.3149941476013009
```

Part (d) -- 7%

Show that $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$

is implemented correctly.

In [14]:

```
# Test Case

h = 0.00001

H = np.eye(90) * h

dLdW = np.empty_like(w)

for i in range(len(w)):
  dLdW[i] = (cost(pred(w + H[i],b,X),t) - cost(pred(w,b,X),t)) / h

print("The analytical result is - ", dLdW)
print("The algorithm result is - ", derivative_cost(X,y,t)[0])
```

```
The analytical result is -  [0.14048729 0.15805553 0.28100441 0.15808494 0.1346605  0.064
41687
 0.15808494 0.0936974  0.14637271 0.02342427 0.1756238  0.07027297
 0.26929218 0.09952394 0.0468486  0.1756238  0.24586775 0.1288044
 0.20487513 0.14637271 0.21073121 0.06441687 0.05853143 0.14051661
 0.14634337 0.16394105 0.22829947 0.26343607 0.09952394 0.25172386
 0.22832877 0.19901906 0.19316298 0.12291908 0.13463122 0.1229483
 0.01171213 0.15219945 0.1990482  0.09955351 0.         0.1229483
 0.02342427 0.26929218 0.27514829 0.25172386 0.15219945 0.1288044
 0.21076042 0.08784129 0.1288044  0.14048729 0.09952394 0.0819557
 0.02342427 0.07609963 0.11123611 0.21661653 0.1346605  0.15219945
 0.21658729 0.09366786 0.0819557  0.12291908 0.26343607 0.14637271
 0.1346605  0.11123611 0.05853143 0.22829947 0.26343607 0.12291908
 0.16391162 0.05856078 0.26343607 0.11706302 0.1814799  0.21073121
 0.15805553 0.10540963 0.0936974  0.11706302 0.         0.16976771
 0.09366786 0.14051661 0.18733599 0.0175682  0.14048729 0.11709221]
The algorithm result is -  [0.13793026 0.1850681  0.2887772  0.17616711 0.16439354 0.0889
5369
 0.16473636 0.12682371 0.14541712 0.0554698  0.19488271 0.08775088
 0.27503683 0.08990733 0.06945843 0.18421215 0.23860291 0.12221356
 0.199091   0.16453657 0.20734639 0.09808216 0.09790108 0.12857579
 0.15393867 0.18139417 0.22988199 0.26981355 0.1227911  0.25690071
 0.26913366 0.21874627 0.20931198 0.12951481 0.11926136 0.10830195
 0.04510397 0.17314819 0.22490777 0.12070866 0.02055043 0.14950888
 0.04498557 0.25549811 0.2992753  0.25548182 0.18927534 0.13452985
 0.21820337 0.09912482 0.11566637 0.17311146 0.11701998 0.09788814
 0.05474666 0.08003114 0.10010031 0.21790736 0.17447324 0.16987536
 0.22361763 0.11458894 0.08563165 0.15628976 0.27017361 0.13112217
 0.11793849 0.15161747 0.06184811 0.25213489 0.28637557 0.11043108
 0.15480254 0.07870767 0.26607017 0.11260283 0.20315197 0.22335807
 0.19748535 0.11638683 0.13163908 0.13965714 0.04117202 0.18061028
 0.12637741 0.13461322 0.23040263 0.04761992 0.16311932 0.1602944 ]
```

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 [15]:

```
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
  """
  w = w0
  b = b0
  iter = 0
  global train_norm_xs, train_ts, val_norm_xs, val_ts

  val_accuracy = np.empty(10)
  val_accuracy_counter = 0

  while iter < max_iters:
    # shuffle the training set (there is code above for how to do this)

    # shuffle the training set
    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)
      gradient = derivative_cost(X, y, t)

      # update w and b
      b = b - mu * gradient[1]
      w = w - mu * gradient[0]

      # increment the iteration count
      iter += 1
      # compute and print the *validation* loss and accuracy
      if (iter % 10 == 0):

        val_cost = cost(pred(w,b,val_norm_xs),val_ts)
        val_acc = get_accuracy(pred(w,b,val_norm_xs),val_ts)

        val_accuracy[val_accuracy_counter] = val_acc
        val_accuracy_counter = val_accuracy_counter + 1

        print("Iter %d. [Val Acc %.0f%%, Loss %f]" % (
                iter, val_acc * 100, val_cost))

      if iter >= max_iters:
        break


      # Think what parameters you should return for further use

  return (val_accuracy,w,b)
```

Part (f) -- 7%

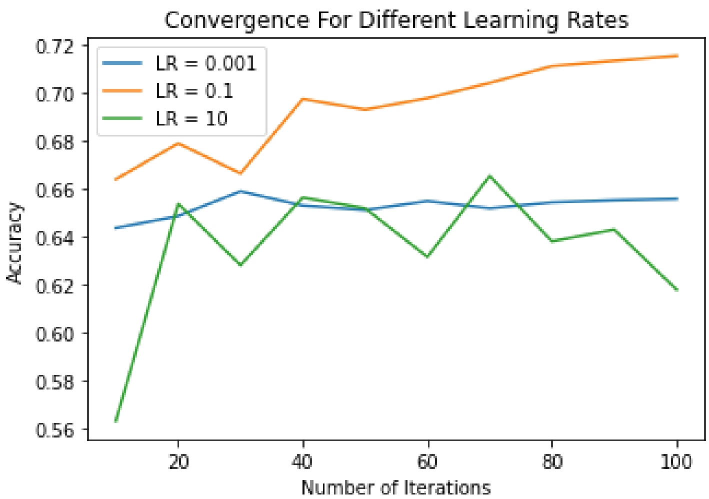
Call `run_gradient_descent` with the weights and biases all initialized to zero. Show that if the learning rate μ is too small, then convergence is slow. Also, show that if μ is too large, then the optimization algorirthm does not converge. The demonstration should be made using plots showing these effects.

In [16]:

```
# First we initialize the weights and biases to zero
w0 = np.zeros(90)
```

```python
b0 = 0

print(type(val_ts))
# Then we calculate the accuracy for different learning rates

small_mu_acc = run_gradient_descent(w0, b0, 0.001, 100, 100)[0]
normal_mu_acc = run_gradient_descent(w0, b0, 0.1, 100, 100)[0]
large_mu_acc = run_gradient_descent(w0, b0, 10, 100, 100)[0]

# And finally we plot those rates

x = np.arange(10,110,10)

plt.plot(x,small_mu_acc)
plt.plot(x,normal_mu_acc)
plt.plot(x,large_mu_acc)

plt.xlabel('Number of Iterations')
plt.ylabel('Accuracy')
plt.title('Convergence For Different Learning Rates')
plt.legend(['LR = 0.001', 'LR = 0.1', 'LR = 10'])      # LR = Learning Rate
plt.show()
```

```
<class 'numpy.ndarray'>
Iter 10. [Val Acc 64%, Loss 0.692394]
Iter 20. [Val Acc 65%, Loss 0.691844]
Iter 30. [Val Acc 66%, Loss 0.691171]
Iter 40. [Val Acc 65%, Loss 0.690394]
Iter 50. [Val Acc 65%, Loss 0.689641]
Iter 60. [Val Acc 65%, Loss 0.689086]
Iter 70. [Val Acc 65%, Loss 0.688392]
Iter 80. [Val Acc 65%, Loss 0.687809]
Iter 90. [Val Acc 66%, Loss 0.687134]
Iter 100. [Val Acc 66%, Loss 0.686463]
Iter 10. [Val Acc 66%, Loss 0.654766]
Iter 20. [Val Acc 68%, Loss 0.633320]
Iter 30. [Val Acc 67%, Loss 0.627454]
Iter 40. [Val Acc 70%, Loss 0.614422]
Iter 50. [Val Acc 69%, Loss 0.607764]
Iter 60. [Val Acc 70%, Loss 0.602455]
Iter 70. [Val Acc 70%, Loss 0.597488]
Iter 80. [Val Acc 71%, Loss 0.592926]
Iter 90. [Val Acc 71%, Loss 0.590670]
Iter 100. [Val Acc 72%, Loss 0.591654]
Iter 10. [Val Acc 56%, Loss 4.388864]
Iter 20. [Val Acc 65%, Loss 3.056586]
Iter 30. [Val Acc 63%, Loss 4.147658]
Iter 40. [Val Acc 66%, Loss 3.253839]
Iter 50. [Val Acc 65%, Loss 2.874682]
Iter 60. [Val Acc 63%, Loss 3.266279]
Iter 70. [Val Acc 67%, Loss 2.605236]
Iter 80. [Val Acc 64%, Loss 4.400185]
Iter 90. [Val Acc 64%, Loss 4.681177]
Iter 100. [Val Acc 62%, Loss 4.021326]
```



Explain and discuss your results here: We can see that only the orange line, which represents a learning rate of 0.1 actually manages to converge to the optimal accuracy.\ When the learning rate is large, the algorithm diverges as can be seen by the green graph.\ On the other hand, the blue graph seems to converge to the optimal value but much slower than the orange graph. The reason for that is the learning rate is too small.

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 [17]:

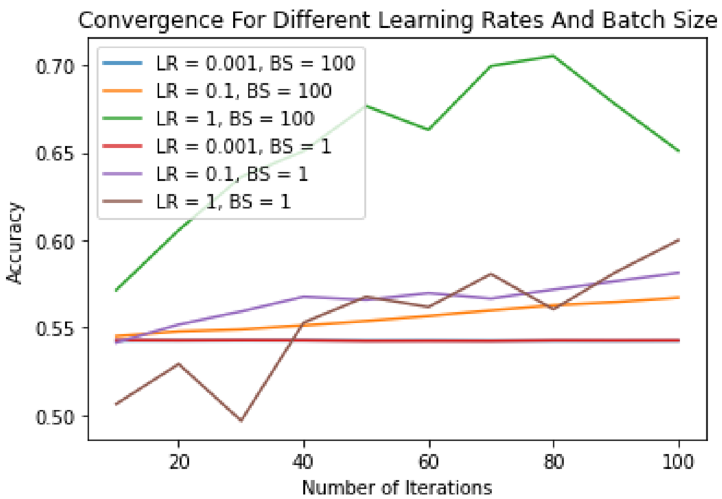
```
w0 = np.random.randn(90)
b0 = np.random.randn(1)[0]

# Then we calculate the accuracy for different learning rates

small_mu_large_batch_acc = run_gradient_descent(w0, b0, 0.001, 100, 100)[0]
normal_mu_large_batch_acc = run_gradient_descent(w0, b0, 0.1, 100, 100)[0]
large_mu_large_batch_acc = run_gradient_descent(w0, b0, 1, 100, 100)[0]
small_mu_small_batch_acc = run_gradient_descent(w0, b0, 0.001, 1, 100)[0]
normal_mu_small_batch_acc = run_gradient_descent(w0, b0, 0.1, 1, 100)[0]
large_mu_small_batch_acc = run_gradient_descent(w0, b0, 1, 1, 100)[0]

# And finally we plot those rates

x = np.arange(10,110,10)

plt.plot(x,small_mu_large_batch_acc)
plt.plot(x,normal_mu_large_batch_acc)
plt.plot(x,large_mu_large_batch_acc)
plt.plot(x,small_mu_small_batch_acc)
plt.plot(x,normal_mu_small_batch_acc)
plt.plot(x,large_mu_small_batch_acc)

plt.xlabel('Number of Iterations')
plt.ylabel('Accuracy')
plt.title('Convergence For Different Learning Rates And Batch Size')
plt.legend(['LR = 0.001, BS = 100', 'LR = 0.1, BS = 100', 'LR = 1, BS = 100','LR = 0.001,
BS = 1', 'LR = 0.1, BS = 1', 'LR = 1, BS = 1'])      # LR = Learning Rate
plt.show()

# Write your code here
return_value = run_gradient_descent(w0, b0, 0.96, 100, 100)
w = return_value[1]
b = return_value[2]
print("The optimal values are:")
print("W = ", w)
print("b = ", b)
```

```
Iter 10. [Val Acc 54%, Loss 3.235191]
Iter 20. [Val Acc 54%, Loss 3.232816]
Iter 30. [Val Acc 54%, Loss 3.230539]
Iter 40. [Val Acc 54%, Loss 3.227657]
Iter 50. [Val Acc 54%, Loss 3.225280]
Iter 60. [Val Acc 54%, Loss 3.222382]
Iter 70. [Val Acc 54%, Loss 3.220098]
Iter 80. [Val Acc 54%, Loss 3.217023]
Iter 90. [Val Acc 54%, Loss 3.214818]
Iter 100. [Val Acc 54%, Loss 3.212335]
Iter 10. [Val Acc 55%, Loss 3.004862]
Iter 20. [Val Acc 55%, Loss 2.812603]
Iter 30. [Val Acc 55%, Loss 2.660648]
Iter 40. [Val Acc 55%, Loss 2.536648]
Iter 50. [Val Acc 55%, Loss 2.431542]
Iter 60. [Val Acc 56%, Loss 2.327189]
Iter 70. [Val Acc 56%, Loss 2.242862]
```

```
Iter 80. [Val Acc 56%, Loss 2.147533]
Iter 90. [Val Acc 56%, Loss 2.066282]
Iter 100. [Val Acc 57%, Loss 1.979577]
Iter 10. [Val Acc 57%, Loss 1.949537]
Iter 20. [Val Acc 61%, Loss 1.385874]
Iter 30. [Val Acc 64%, Loss 1.051736]
Iter 40. [Val Acc 65%, Loss 0.883532]
Iter 50. [Val Acc 68%, Loss 0.761834]
Iter 60. [Val Acc 66%, Loss 0.735007]
Iter 70. [Val Acc 70%, Loss 0.652018]
Iter 80. [Val Acc 71%, Loss 0.634444]
Iter 90. [Val Acc 68%, Loss 0.692434]
Iter 100. [Val Acc 65%, Loss 0.795775]
Iter 10. [Val Acc 54%, Loss 3.239634]
Iter 20. [Val Acc 54%, Loss 3.232453]
Iter 30. [Val Acc 54%, Loss 3.230856]
Iter 40. [Val Acc 54%, Loss 3.221394]
Iter 50. [Val Acc 54%, Loss 3.215894]
Iter 60. [Val Acc 54%, Loss 3.215517]
Iter 70. [Val Acc 54%, Loss 3.213636]
Iter 80. [Val Acc 54%, Loss 3.202715]
Iter 90. [Val Acc 54%, Loss 3.200271]
Iter 100. [Val Acc 54%, Loss 3.199970]
Iter 10. [Val Acc 54%, Loss 3.091769]
Iter 20. [Val Acc 55%, Loss 2.985422]
Iter 30. [Val Acc 56%, Loss 2.794985]
Iter 40. [Val Acc 57%, Loss 2.737193]
Iter 50. [Val Acc 57%, Loss 2.836735]
Iter 60. [Val Acc 57%, Loss 2.776445]
Iter 70. [Val Acc 57%, Loss 2.859844]
Iter 80. [Val Acc 57%, Loss 2.684285]
Iter 90. [Val Acc 58%, Loss 2.631404]
Iter 100. [Val Acc 58%, Loss 2.367503]
Iter 10. [Val Acc 51%, Loss 7.699397]
```

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

```
Iter 20. [Val Acc 53%, Loss 11.918882]
Iter 30. [Val Acc 50%, Loss 13.434635]
Iter 40. [Val Acc 55%, Loss 9.644307]
Iter 50. [Val Acc 57%, Loss 9.374008]
Iter 60. [Val Acc 56%, Loss 7.132957]
Iter 70. [Val Acc 58%, Loss 7.134390]
Iter 80. [Val Acc 56%, Loss 6.741147]
Iter 90. [Val Acc 58%, Loss 8.653679]
Iter 100. [Val Acc 60%, Loss 7.194568]
```




```
Iter 10. [Val Acc 57%, Loss 2.049417]
Iter 20. [Val Acc 62%, Loss 1.416575]
Iter 30. [Val Acc 65%, Loss 1.060135]
Iter 40. [Val Acc 67%, Loss 0.889176]
Iter 50. [Val Acc 67%, Loss 0.805356]
Iter 60. [Val Acc 66%, Loss 0.750943]
```

```
Iter 70. [Val Acc 69%, Loss 0.699212]
Iter 80. [Val Acc 67%, Loss 0.739734]
Iter 90. [Val Acc 70%, Loss 0.647291]
Iter 100. [Val Acc 71%, Loss 0.635001]
The optimal values are:
W =  [ 1.5787773  -1.00942211 -0.10288704 -0.52399695 -0.07507961 -0.37435923
  0.06893695 -0.3443918  -0.32930763  0.3927501   0.08665824 -0.00739056
  0.18784511  0.29888572  0.12238551  0.21146638  0.15452929  0.79706374
  0.29962835  0.50521716  0.27893899 -0.56414149  0.15738637 -0.08651325
 -0.30482236  0.21891091 -0.22292385  0.04656066 -0.06811478  0.1325123
  0.12137567  0.00833851 -0.05260212  0.03042789 -0.01176423 -0.05436246
 -0.10133897  0.11993125  0.2842264  -0.09295781 -0.06458595  0.11353331
  0.01958267  0.12901566 -0.11461054  0.36816108  0.0692521  -0.0896005
  0.16337319 -0.05106412  0.06296432  0.11108292  0.20423958 -0.04744518
 -0.01750323 -0.18046763 -0.31721407 -0.11013632 -0.1848421  -0.13940567
 -0.1801845  -0.04405558 -0.43650721  0.0124486  -0.45756779 -0.17620873
 -0.06981331  0.12866875 -0.13153868 -0.23144316  0.07301761 -0.11911227
  0.12074864  0.20102293  0.15261097 -0.01426944  0.10819819 -0.14231425
 -0.22942227  0.06243781 -0.26110076 -0.13430122  0.16374106  0.02660366
  0.14144155  0.00257852  0.23381979 -0.40165724 -0.01293599  0.27012721]
b =  0.406990041444826
```

Explain and discuss your results here: We tried different values of Batch Size and Learning Rate.\ As can be seen in the above plot, a learning rate close to 1 is preferable, and a batch size of 100 is better than a batch size of 1.

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 [18]:

```
# Write your code here

train_acc = get_accuracy(pred(w,b,train_norm_xs),train_ts)
val_acc = get_accuracy(pred(w,b,val_norm_xs),val_ts)
test_acc = get_accuracy(pred(w,b,test_norm_xs),test_ts)

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

```
train_acc =  0.705072332402741  val_acc =  0.70538  test_acc =  0.7030989734650397
```

Explain and discuss your results here: There are very small differences between the values. That must indicate our algorithm is not biased.

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 [19]:

```
import sklearn.linear_model

model = sklearn.linear_model.SGDClassifier()
model.fit(train_norm_xs,train_ts.ravel())

train_acc = get_accuracy(model.predict(train_norm_xs),train_ts)
val_acc = get_accuracy(model.predict(val_norm_xs),val_ts)
```

```python
test_acc = get_accuracy(model.predict(test_norm_xs),test_ts)

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

```
train_acc =  0.7291227052439482  val_acc =  0.72864  test_acc =  0.7217121828394344
```

This parts helps by checking if the code worked. Check if you get similar results, if not repair your code We got similare results. yay!