Assignment 1. Music Century Classification

Deadline: Sunday, April 4th, by 9pm.

Submission: Submit a PDF export of the completed notebook as well as the ipynb file.

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 = False

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

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:

In [3]: `df`

Out[3]:

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

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
0	1	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12.23257	7.83089
1	1	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0.92019	18.76548
2	1	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2.35035	16.07017
3	1	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10.34124	3.55005
4	1	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9.37636	12.63699
5	1	50.54767	0.31568	92.35066	22.38696	-25.51870	-19.04928	20.67345	-5.19943	3.63566
6	1	50.57546	33.17843	50.53517	11.55217	-27.24764	-8.78206	-12.04282	-9.53930	28.61811
7	1	48.26892	8.97526	75.23158	24.04945	-16.02105	-14.09491	8.11871	-1.87566	7.46701
8	1	49.75468	33.99581	56.73846	2.89581	-2.92429	-26.44413	1.71392	-0.55644	22.08594
9	1	45.17809	46.34234	-40.65357	-2.47909	1.21253	-0.65302	-6.95536	-12.20040	17.02512
10	1	39.13076	-23.01763	-36.20583	1.67519	-4.27101	13.01158	8.05718	-8.41088	6.27370
11	1	37.66498	-34.05910	-17.36060	-26.77781	-39.95119	-20.75000	-0.10231	-0.89972	-1.30205
12	1	26.51957	-148.15762	-13.30095	-7.25851	17.22029	-21.99439	5.51947	3.48418	2.61738
13	1	37.68491	-26.84185	-27.10566	-14.95883	-5.87200	-21.68979	4.87374	-18.01800	1.52141
14	0	39.11695	-8.29767	-51.37966	-4.42668	-30.06506	-11.95916	-0.85322	-8.86179	11.36680
15	1	35.05129	-67.97714	-14.20239	-6.68696	-0.61230	-18.70341	-1.31928	-9.46370	5.53492
16	1	33.63129	-96.14912	-89.38216	-12.11699	13.77252	-6.69377	-33.36843	-24.81437	21.22757
17	0	41.38639	-20.78665	51.80155	17.21415	-36.44189	-11.53169	11.75252	-7.62428	-3.65488
18	0	37.45034	11.42615	56.28982	19.58426	-16.43530	2.22457	1.02668	-7.34736	-0.01184
19	0	39.71092	-4.92800	12.88590	-11.87773	2.48031	-16.11028	-16.40421	-8.29657	9.86817

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
'''
To evaluate our model correctly we want to know how well will the model will perform in pra
Using some songs from an artist in the training set, and other songs from the same art
can lead to a situation where the The model will learn specific featues of the singer
such as the singer's voice and will predict the decade according to the singer voice.
In that case we will get a good test accuracy because the singer's songs are in the te
but that does not mean the model will perfom the same on different singers
'''
```

Out[6]:
```
"\nTo evaluate our model correctly we want to know how well will the model will perform in
practice on a song it hasn't learned about.\nUsing some songs from an artist in the tr
aining set, and other songs from the same artist in the test set\ncan lead to a situat
ion where the The model will learn specific featues of the singer\nsuch as the singe
r's voice and will predict the decade according to the singer voice.\nIn that case we
will get a good test accuracy because the singer's songs are in the test data,hence th
e model will predict correctly,\nbut that does not mean the model will perfom the same
on different singers\n"
```

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 "ye
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]:
```
'''
The conceptual idea is that test data is supposed to mimic new, previously unseen data
If we normalize the test data separately we can cause data leakage,
meaing we are introducing future information into the training explanatory variables
Therefore, we should perform feature normalisation over the training data,
and then perform normalization on testing data me using the mean and variance of trai
In this way, we can test and evaluate whether our model can generalize well to new, u
'''
```

Out[8]:
```
'\nThe conceptual idea is that test data is supposed to mimic new, previously unseen d
ata after training. \nIf we normalize the test data separately we can cause data leaka
ge,\nmeaing we are introducing future information into the training explanatory variab
les (i.e. the mean and variance).\nTherefore, we should perform feature normalisation
over the training data,\nand then perform normalisation on testing data me using the m
ean and variance of training explanatory variables.\nIn this way, we can test and eval
uate whether our model can generalize well to new, unseen data points.\n'
```

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]

# Write your explanation here
'''
First , we should limit how many times we use the test set becaus we want to evaluate
If we use the test data and change our training according to the test data we will get
to unseen data, hence we sould use the validation data insead.
We use validation set during the model building proccess so we can give an estimate o
The validation dataset is different from the test dataset , both datasets are held bac
but the validation set used to give an unbiased estimate of the skill of the final tu
'''
```

Out[9]:
```
"\nFirst , we should limit how many times we use the test set becaus we want to evalua
te the model according to unseen data.\nIf we use the test data and change our trainin
g according to the test data we will get good test accuracy but it won't be generalize
\nto unseen data, hence we sould use the validation data insead.\nWe use validation se
t during the model building proccess so we can give an estimate of model skill while t
uning model's hyperparameters.\nThe validation dataset is different from the test data
set , both datasets are held back from the training of the model,\nbut the validation
set used to give an unbiased estimate of the skill of the final tuned model when compa
ring or selecting between final models.\n"
```

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

def cross_entropy(t, y):
  epsilon=1e-05
  return -t * np.log(y+epsilon) - (1 - t) * np.log(1 - y+epsilon)

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:

$$y = \sigma(\mathbf{w}^T\mathbf{x} + b), \quad (1)$$

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

In [11]:
```
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 decin
  """
  z=np.dot(X,w)+b
  y=sigmoid(z)
  return y
```

In [12]:
```
pred(np.zeros(90), 1, np.ones([2, 90]))
```

Out[12]:
```
array([0.73105858, 0.73105858])
```

In [13]:
```
pred(np.zeros(90), 1, train_norm_xs)
```

Out[13]:
```
array([0.73105858, 0.73105858, 0.73105858, ..., 0.73105858, 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.

In [14]:
```
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
  N=len(y)
  epsilon=1e-05
  dLdy=(-t/(y+epsilon)+(1-t)/(1-y+epsilon))/N
  sig_d=y*(1-y)
  dydb=sig_d
  dLdb=np.dot(dydb,dLdy)
  dydw=sig_d*X.T
  dLdw=np.dot(dydw,dLdy)
  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.

We calculate $\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}$ using the chain rule :

$L(w,b,X,t) = \frac{-1}{N}\sum_{n=1}^{N} t_n\log(y_n) + (1-t_n)\log(1-y_n) = \frac{-1}{N}(t\cdot\log(y) + (1-t)\cdot\log(1-y)$

So $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\cdot\frac{\partial y}{\partial w}$, and $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\cdot\frac{\partial y}{\partial b}$ where:

- $\frac{\partial L}{\partial y} = \frac{-1}{N}(\frac{t}{y} - \frac{1-t}{1-y}) = \frac{-1}{N}\frac{t-y}{y(1-y)}$
- $\frac{\partial y}{\partial b} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial b} = \sigma'(z)\cdot 1 = \sigma(z)(1-\sigma(z)) = y(1-y)$
- $\frac{\partial y}{\partial w} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial w} = \sigma'(z)X = (\sigma(z)(1-\sigma(z)))X = (y\cdot(1-y))X$

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 [15]:
```
# helper method to compute analytical deriviate according to parameter_to_check.
def compute_analytical_deriviate(w,b,t,X,parameter_to_check,h=1e-05):
  y= pred(w, b, X)
  L=cost(y,t)
  if parameter_to_check=='b':
    y_plus=pred(w,b+h,X)
    L_plus=cost(y_plus,t)
    deriviate=(L_plus-L)/h

  elif parameter_to_check=='w':
    L_plus=np.zeros(len(w))
    deriviate=np.zeros(len(w))
    for i in range(len(w)):
      # Compute cost of w[i] + h
```

```
our deriviate according to w : [-0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.268
93774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774
 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774 -0.26893774]
```

In [22]:
```
analytical_deriviate_2=compute_analytical_deriviate(w_2,b_2,t_2,X_2,'w')
print(make_bold("Second example:"))
print(make_underline("analytical_deriviate according to w :"), analytical_deriviate_2
print(make_underline("our deriviate according to w:"),dw_2)
```

```
Second example:
analytical_deriviate according to w : [-1.47644080e-02  5.43838112e-02 -6.19292034e-02
 -5.17513915e-02
  4.44657597e-01 -2.61039973e-02  1.00111588e-01  2.95271216e-01
 -1.07128801e-01  3.48930005e-01 -2.39404368e-01  6.97615545e-02
 -8.40610272e-02 -6.76292412e-02  3.63342705e-01 -3.43283860e-02
 -1.07756014e-01 -7.47970289e-02  2.39380916e-01  1.88978884e-02
  3.49757070e-02 -4.18057133e-04 -4.27618968e-02  1.03988501e-01
 -1.14824814e-01 -1.46854731e-01  3.01598029e-01 -1.74495599e-01
 -7.70476966e-02 -1.83014623e-02  6.05119196e-01 -1.43932629e-01
  1.27517959e-01 -8.95726896e-02  1.44363447e-01  2.75935237e-01
 -3.07927508e-02 -1.57032070e-01  7.60373407e-02 -1.57870494e-01
 -3.31004655e-02 -4.10890325e-01  1.05439660e-01 -1.66573491e-01
 -4.70736006e-01  2.16804910e-01 -2.11820110e-01  1.41601563e-01
  2.74942700e-01 -1.31709012e-02 -7.13387406e-02 -2.11779580e-01
 -4.44531102e-01 -4.21720436e-01 -1.30845725e-01 -4.63158233e-02
  6.41531406e-01  3.09861640e-01 -7.29608387e-02  2.50576874e-01
 -2.57759779e-02 -9.19629441e-03  1.73171908e-01  1.21600297e-01
  1.20540559e-01 -1.59917720e-01  1.83739652e-01 -5.98321048e-02
 -1.40434514e-01  2.01581991e-01 -1.88668850e-01 -5.89670431e-01
 -2.18524214e-01  5.03886190e-02  1.50346285e-02 -1.91744935e-02
  6.28534598e-02 -3.34416795e-01 -1.13901549e-01 -4.08810569e-02
 -2.91315103e-01  1.63299095e-02  9.83162302e-02 -9.96249113e-02
 -3.89829063e-01  9.80662512e-03 -2.58065593e-02 -1.18257845e-02
  2.29201220e-02  7.08526612e-02]
our deriviate according to w: [-1.47644291e-02  5.43837909e-02 -6.19292324e-02 -5.1751
4333e-02
  4.44656306e-01 -2.61040149e-02  1.00111524e-01  2.95270660e-01
 -1.07128885e-01  3.48929178e-01 -2.39404771e-01  6.97615275e-02
 -8.40610731e-02 -6.76292716e-02  3.63341858e-01 -3.43283780e-02
 -1.07756117e-01 -7.47970349e-02  2.39380514e-01  1.88978388e-02
  3.49756868e-02 -4.18059737e-04 -4.27619118e-02  1.03988420e-01
 -1.14824899e-01 -1.46854877e-01  3.01597324e-01 -1.74495807e-01
 -7.70477462e-02 -1.83014883e-02  6.05118947e-01 -1.43932775e-01
  1.27517840e-01 -8.95727544e-02  1.44363312e-01  2.75934753e-01
 -3.07928224e-02 -1.57032233e-01  7.60371770e-02 -1.57870665e-01
 -3.31005494e-02 -4.10891406e-01  1.05439566e-01 -1.66573745e-01
 -4.70737423e-01  2.16804607e-01 -2.11820400e-01  1.41601435e-01
  2.74942191e-01 -1.31709072e-02 -7.13387366e-02 -2.11779796e-01
 -4.44532388e-01 -4.21721569e-01 -1.30845841e-01 -4.63158434e-02
  6.41528780e-01  3.09860959e-01 -7.29608765e-02  2.50576474e-01
 -2.57759833e-02 -9.19629552e-03  1.73171717e-01  1.21600203e-01
  1.20540467e-01 -1.59919885e-01  1.83739430e-01 -5.98321436e-02
 -1.40434650e-01  2.01581731e-01 -1.88669115e-01 -5.89672637e-01
 -2.18523445e-01  5.03885756e-02  1.50345678e-02 -1.91744961e-02
  6.28534332e-02 -3.34417552e-01 -1.13901664e-01 -4.08810690e-02
 -2.91315642e-01  1.63298669e-02  9.83161201e-02 -9.96249788e-02
 -3.89830036e-01  9.80661307e-03 -2.58065641e-02 -1.18257951e-02
  2.29201128e-02  7.08526268e-02]
```

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 [23]:
```
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
  i=[]
  accuracies=[]
  losses=[]

  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_xs = train_xs[reindex]
    train_norm_xs_local = train_norm_xs[reindex]
    train_ts_local = 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_local[i:(i + batch_size)]
      t = train_ts_local[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
      dw,db=derivative_cost(X,y,t)
      w-=mu*dw
      b-=mu*db

      # increment the iteration count
      iter += 1
      # compute and print the *validation* loss and accuracy
      if (iter % 10 == 0):
        y_val=pred(w,b,val_norm_xs)
        val_cost = cost(y_val,val_ts.squeeze())
        val_acc = get_accuracy(y_val,val_ts.squeeze())
        print("Iter %d. [Val Acc %.0f%%, Loss %f]" % (
                iter, val_acc * 100, val_cost))
        losses.append(val_cost)
        accuracies.append(val_acc)

      if iter >= max_iters:
        break

  # we decided to return the losses according to iteration so we can se the loss conve

  return w,b,losses
```

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 algorithm does not converge. The demonstration should be made using plots showing these effects.

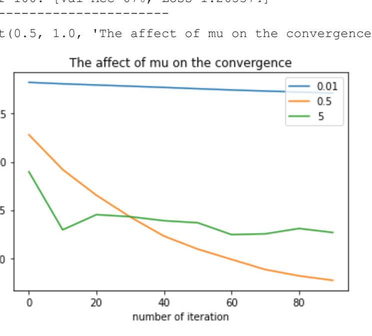
In [24]:
```
w0 = np.random.randn(90)
b0 = np.random.randn(1)[0]

mus=[0.01,0.5,5]
for mu in mus:
  print(make_bold("mu="),mu,":")
  w,b,losses=run_gradient_descent(w0,b0,mu)
  print("-------------------------")
  plt.plot(range(0,100,10),losses)

plt.legend(mus,loc='upper right')
plt.xlabel("number of iteration")
plt.ylabel("Loss")
plt.title("The affect of mu on the convergence")
```

```
mu= 0.01 :
Iter 10. [Val Acc 50%, Loss 2.825022]
Iter 20. [Val Acc 50%, Loss 2.810037]
Iter 30. [Val Acc 50%, Loss 2.797656]
Iter 40. [Val Acc 50%, Loss 2.785225]
Iter 50. [Val Acc 50%, Loss 2.772561]
Iter 60. [Val Acc 50%, Loss 2.759051]
Iter 70. [Val Acc 50%, Loss 2.746125]
Iter 80. [Val Acc 50%, Loss 2.734800]
Iter 90. [Val Acc 50%, Loss 2.724746]
Iter 100. [Val Acc 50%, Loss 2.712769]
-------------------------
mu= 0.5 :
Iter 10. [Val Acc 52%, Loss 2.281349]
Iter 20. [Val Acc 54%, Loss 1.919428]
Iter 30. [Val Acc 56%, Loss 1.652089]
Iter 40. [Val Acc 58%, Loss 1.426836]
Iter 50. [Val Acc 60%, Loss 1.228126]
Iter 60. [Val Acc 62%, Loss 1.092885]
Iter 70. [Val Acc 63%, Loss 0.986005]
Iter 80. [Val Acc 65%, Loss 0.880350]
Iter 90. [Val Acc 66%, Loss 0.814806]
Iter 100. [Val Acc 67%, Loss 0.768555]
-------------------------
mu= 5 :
Iter 10. [Val Acc 61%, Loss 1.897048]
Iter 20. [Val Acc 66%, Loss 1.293454]
Iter 30. [Val Acc 65%, Loss 1.450958]
Iter 40. [Val Acc 66%, Loss 1.429681]
Iter 50. [Val Acc 69%, Loss 1.388019]
Iter 60. [Val Acc 68%, Loss 1.366573]
Iter 70. [Val Acc 68%, Loss 1.243627]
Iter 80. [Val Acc 68%, Loss 1.251463]
Iter 90. [Val Acc 68%, Loss 1.307242]
Iter 100. [Val Acc 67%, Loss 1.265374]
-------------------------
```

Out[24]:
```
Text(0.5, 1.0, 'The affect of mu on the convergence')
```



Explain and discuss your results here:

Has we can see in the graph, if the learning rate is set too low, for example $\mu = 0.01$, the training progress very slowly because in each epoch we are making very tiny updates to the weights.

However, if the learning rate is set too high, for example $\mu = 5$ it cause undesirable divergent behavior in our loss function because the steps do oscillation and do not converge to the local minima.

For regular learning rate such as $\mu = 0.5$ we can tell that we got covergence and the covergence is not too slow.

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.

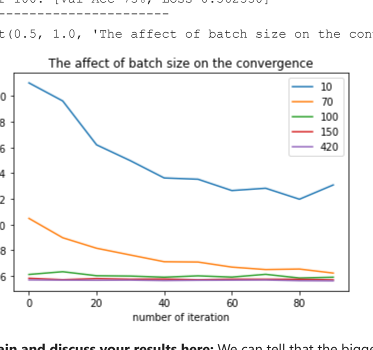
In [25]:
```
w0 = np.random.randn(90)
b0 = np.random.randn(1)[0]

mu=0.5
batches=[10,70,100,150,420]
for batch_size in batches:
  print(make_bold("batch_size="),batch_size,":")
  w,b,losses=run_gradient_descent(w0,b0,mu,batch_size)
  print("-------------------------")
  plt.plot(range(0,100,10),losses)

plt.legend(batches,loc='upper right')
plt.xlabel("number of iteration")
plt.ylabel("Loss")
plt.title("The affect of batch size on the convergence")
```

```
batch_size= 10 :
Iter 10. [Val Acc 56%, Loss 2.102463]
Iter 20. [Val Acc 55%, Loss 1.961835]
Iter 30. [Val Acc 60%, Loss 1.620595]
Iter 40. [Val Acc 59%, Loss 1.497217]
Iter 50. [Val Acc 61%, Loss 1.362714]
Iter 60. [Val Acc 63%, Loss 1.352913]
Iter 70. [Val Acc 64%, Loss 1.264761]
Iter 80. [Val Acc 63%, Loss 1.282819]
Iter 90. [Val Acc 65%, Loss 1.197332]
Iter 100. [Val Acc 66%, Loss 1.308506]
-------------------------
batch_size= 70 :
Iter 10. [Val Acc 67%, Loss 1.048324]
Iter 20. [Val Acc 68%, Loss 0.897608]
Iter 30. [Val Acc 69%, Loss 0.815982]
Iter 40. [Val Acc 69%, Loss 0.763307]
Iter 50. [Val Acc 70%, Loss 0.711019]
Iter 60. [Val Acc 69%, Loss 0.708562]
Iter 70. [Val Acc 70%, Loss 0.669150]
Iter 80. [Val Acc 70%, Loss 0.649959]
Iter 90. [Val Acc 70%, Loss 0.654027]
Iter 100. [Val Acc 71%, Loss 0.622528]
-------------------------
batch_size= 100 :
Iter 10. [Val Acc 71%, Loss 0.611853]
Iter 20. [Val Acc 71%, Loss 0.633214]
Iter 30. [Val Acc 71%, Loss 0.601980]
Iter 40. [Val Acc 72%, Loss 0.599402]
Iter 50. [Val Acc 72%, Loss 0.590583]
Iter 60. [Val Acc 72%, Loss 0.601073]
Iter 70. [Val Acc 72%, Loss 0.591435]
Iter 80. [Val Acc 70%, Loss 0.612864]
Iter 90. [Val Acc 72%, Loss 0.584292]
Iter 100. [Val Acc 72%, Loss 0.590514]
-------------------------
batch_size= 150 :
Iter 10. [Val Acc 72%, Loss 0.582124]
Iter 20. [Val Acc 73%, Loss 0.571310]
Iter 30. [Val Acc 73%, Loss 0.579775]
Iter 40. [Val Acc 72%, Loss 0.574862]
Iter 50. [Val Acc 73%, Loss 0.576129]
Iter 60. [Val Acc 73%, Loss 0.570809]
Iter 70. [Val Acc 72%, Loss 0.575014]
Iter 80. [Val Acc 73%, Loss 0.574661]
Iter 90. [Val Acc 73%, Loss 0.574354]
Iter 100. [Val Acc 73%, Loss 0.569938]
-------------------------
batch_size= 420 :
Iter 10. [Val Acc 73%, Loss 0.569512]
Iter 20. [Val Acc 73%, Loss 0.567949]
Iter 30. [Val Acc 73%, Loss 0.567201]
Iter 40. [Val Acc 73%, Loss 0.568040]
Iter 50. [Val Acc 73%, Loss 0.564597]
Iter 60. [Val Acc 73%, Loss 0.566801]
Iter 70. [Val Acc 73%, Loss 0.567701]
Iter 80. [Val Acc 72%, Loss 0.568870]
Iter 90. [Val Acc 73%, Loss 0.563493]
Iter 100. [Val Acc 73%, Loss 0.562330]
-------------------------
```

Out[25]:
```
Text(0.5, 1.0, 'The affect of batch size on the convergence')
```



Explain and discuss your results here: We can tell that the bigger the batch size we get fast convergence, which make sense because with bigger batches we need much fewer updates for the same accuracy.

However, even if it seems good to use a great batch size choosing too large of a vtch size will lead to poor generalization. hence 100 or even 70 will be good for our model.

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 [26]:
```
# Write your code here
best_w,best_b,_=run_gradient_descent(w0,b0,0.5,150)
train_ys=pred(best_w,best_b,train_norm_xs)
test_ys=pred(best_w,best_b,test_norm_xs)
val_ys=pred(best_w,best_b,val_norm_xs)


train_acc = get_accuracy(train_ys,train_ts)
val_acc =  get_accuracy(val_ys,val_ts)
test_acc =  get_accuracy(test_ys,test_ts)

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

```
Iter 10. [Val Acc 72%, Loss 0.579656]
Iter 20. [Val Acc 73%, Loss 0.572601]
Iter 30. [Val Acc 72%, Loss 0.577637]
Iter 40. [Val Acc 72%, Loss 0.581342]
Iter 50. [Val Acc 72%, Loss 0.576643]
Iter 60. [Val Acc 72%, Loss 0.579239]
Iter 70. [Val Acc 72%, Loss 0.574741]
Iter 80. [Val Acc 72%, Loss 0.577013]
Iter 90. [Val Acc 72%, Loss 0.586221]
Iter 100. [Val Acc 72%, Loss 0.572728]
train_acc =  0.7214483400408493  val_acc =  0.7232  test_acc =  0.7143714894441217
```

Explain and discuss your results here:

Train accuracy is a little bit better then the test_accuracy because the test data is an unseen data so it is very likely have lower accuracy on an unseen test dataset. The differnce here are not so great, it is a good sign it mean that the model is well generalized.

Val accuracy should be some where between test and train accuracy - it is unseen data but we use the validation set for model selection so we did choose our parameters so the val_acc will be as good as the train_acc

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 [27]:
```
import sklearn.linear_model

model = sklearn.linear_model.LogisticRegression()
model.fit(train_norm_xs,train_ts)
train_ys=model.predict(train_norm_xs)
test_ys=model.predict(test_norm_xs)
val_ys=model.predict(val_norm_xs)

train_acc = get_accuracy(train_ys,train_ts)
val_acc =  get_accuracy(val_ys,val_ts)
test_acc =  get_accuracy(test_ys,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:760: DataConversion
Warning: A column-vector y was passed when a 1d array was expected. Please change the
```

```
shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
train_acc =  0.7328402402620161  val_acc =  0.73344  test_acc =  0.7264574859577765
```

Converting To PDF

In [28]:
```
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
pandoc is already the newest version (1.19.2.4~dfsg-1build4).
texlive is already the newest version (2017.20180305-1).
texlive-latex-extra is already the newest version (2017.20180305-2).
texlive-xetex is already the newest version (2017.20180305-1).
0 upgraded, 0 newly installed, 0 to remove and 30 not upgraded.
Requirement already satisfied: pypandoc in /usr/local/lib/python3.7/dist-packages (1.
5)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (f
rom pypandoc) (54.2.0)
Requirement already satisfied: pip>=8.1.0 in /usr/local/lib/python3.7/dist-packages (f
rom pypandoc) (19.3.1)
Requirement already satisfied: wheel>=0.25.0 in /usr/local/lib/python3.7/dist-packages
(from pypandoc) (0.36.2)
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
!cp /content/drive/My\ Drive/Colab\ Notebooks/Assignment1.ipynb ./
```

In [ ]:
```
!jupyter nbconvert --to PDF "Assignment1.ipynb"
```

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