Assignment1

April 4, 2021

1 Assignment 1. Music Century Classification

Students: Amit Moses 312614324; Vladimir Gordon 312736481.

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.

1.1 Question 1. Data (21%)

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

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

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

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

[ ]:
```
import pandas
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

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

Here are some resources to help you get started:

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

[ ]:
```
load_from_drive = False

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

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

[ ]:
```
df
```

[ ]:

	year	var1	var2	...	var88	var89	var90
0	1	49.94357	21.47114	...	-1.82223	-27.46348	2.26327
1	1	48.73215	18.42930	...	12.04941	58.43453	26.92061
2	1	50.95714	31.85602	...	-0.05859	39.67068	-0.66345
3	1	48.24750	-1.89837	...	9.90558	199.62971	18.85382
4	1	50.97020	42.20998	...	7.88713	55.66926	28.74903
...	...	...	...	...	...	...	...
515340	1	51.28467	45.88068	...	3.42901	-41.14721	-15.46052
515341	1	49.87870	37.93125	...	12.96552	92.11633	10.88815
515342	1	45.12852	12.65758	...	-6.07171	53.96319	-8.09364
515343	1	44.16614	32.38368	...	20.32240	14.83107	39.74909
515344	1	51.85726	59.11655	...	-5.51512	32.35602	12.17352

[515345 rows x 91 columns]

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

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

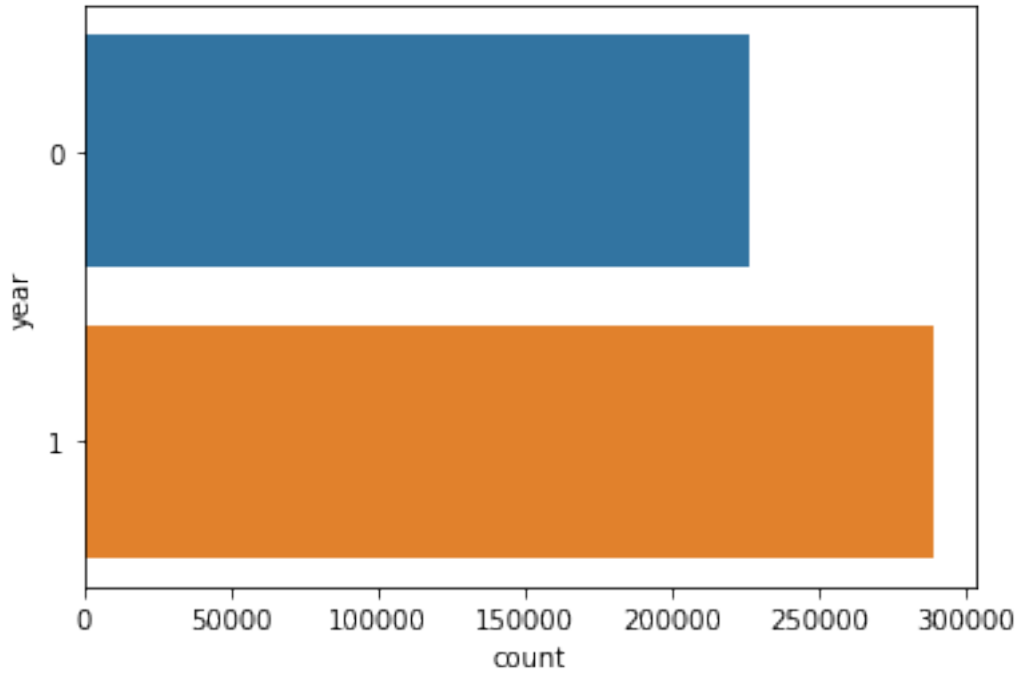
[ ]:
```
df.head(20)
```

[ ]:

	year	var1	var2	...	var88	var89	var90
0	1	49.94357	21.47114	...	-1.82223	-27.46348	2.26327
1	1	48.73215	18.42930	...	12.04941	58.43453	26.92061
2	1	50.95714	31.85602	...	-0.05859	39.67068	-0.66345
3	1	48.24750	-1.89837	...	9.90558	199.62971	18.85382
4	1	50.97020	42.20998	...	7.88713	55.66926	28.74903
5	1	50.54767	0.31568	...	5.00283	-11.02257	0.02263

```
6      1  50.57546   33.17843  ...   4.50056    -4.62739   1.40192
7      1  48.26892    8.97526  ...  -0.30633     3.98364  -3.72556
8      1  49.75468   33.99581  ...   5.48708    -9.13495   6.08680
9      1  45.17809   46.34234  ...  11.49326   -89.21804 -15.09719
10     1  39.13076  -23.01763  ...  -2.59543   109.19723  23.36143
11     1  37.66498  -34.05910  ...  -5.19009     8.83617  -7.16056
12     1  26.51957 -148.15762  ...  23.00230  -164.02536  51.54138
13     1  37.68491  -26.84185  ...  14.11648 -1030.99180  99.28967
14     0  39.11695   -8.29767  ...  -2.14942  -211.48202 -12.81569
15     1  35.05129  -67.97714  ...  20.73063  -562.07671  43.44696
16     1  33.63129  -96.14912  ...   3.44539   259.10825  10.28525
17     0  41.38639  -20.78665  ...   4.44627    58.16913  -0.02409
18     0  37.45034   11.42615  ...  25.73235   157.22967  38.70617
19     0  39.71092   -4.92800  ...   2.34002   -31.57015   1.58400

[20 rows x 91 columns]
```

[ ]:
```
sns.countplot(y="year", data=df);
```



1.1.1 Part (a) -- 7%

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

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

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

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

# Write your explanation here
'''
Answer:
Our goal when creating a trainig set is to create it versatile as much we can.
The reason for this is to let our model study different cases and different␣
 ↪variables.
The seperation is critical since the sole artist will probably have similar␣
 ↪recognizable patterns in the fetures,
outcomes with perfect prediction of the class, since the model will start␣
 ↪"memorizing" data, and thus, will effect the score of the model.
In the other hand, for unknown artist, it may predict poorly. Therefore, it␣
 ↪indicates on good score, while this score is irrelevant.
'''
```

1.1.2 Part (b) -- 7%

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

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

[ ]:
```
# Write your explanation here
'''
Answer: Test set is the "simulation data" for our model, for evaluation reasons␣
 ↪of the model.
Test set and Traning set may have different mean and standart deviation value.
```

```
So, it may create error. You can't evalute the model based on different values.␣
 ↪This will be improper.
Other way to look at this is that normalization of the Trainig set is important␣
 ↪so all the features
will have the same impact on the netwotk in the training phase - it is␣
 ↪equivalent to multiply each feature with a Weight.
So, each Weight should be applied to each feature in the Test set too.
Normalizing the Test set using the Test data mean and standard deviations will␣
 ↪cause different Weights.
'''
```

1.1.3 Part (c) -- 7%

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

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

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

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

# Write your explanation here
'''
Answer:
It's important to limit the times of usage if the test set,
because the test set used to provide an unbiased evaluation of the final model,
while the validation set used to provide evaluation of a model while tuning␣
 ↪model parameters.
In this example, validation set shuffled, meaning the validation set may␣
 ↪change,
and this means that the data within may change, effecting the model in a␣
 ↪different way.
'''
```

1.2 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).

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

def cross_entropy(t, y):
  return -t * np.log(y + 1e-30) - (1 - t) * np.log(1 - y + 1e-30)

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
```

1.2.1 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:

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

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

[ ]:
```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
  """
  # Your code goes here
  if np.shape(w)!=(90,):
    w = np.zeros(90) # TODO - change the functiolnality
  b = float(b)
  f = X @ w + b # @ is dot multiplication
  return sigmoid(f)
```

1.2.2 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.

[ ]:
```
# Vova
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
  dLdw = (X.T@(y-t))/np.shape(y)[0]
  dLdb = np.mean(y-t)
  return (dLdw, dLdb)
```

2 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. https://en.wikipedia.org/wiki/Wikipedia:LaTeX_symbols

The loss function:

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N} H(y,t)$$

and $H(y,t)$ defined as:

$$H(y,t) = -t * log(y) - (1-t) * log(1-y)$$

y defined as:

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

Let's define z as:

$$z \triangleq \mathbf{w}^T\mathbf{x} + b$$

Since y consist of a sigmoid function:

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

The gradient of the loss function $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$:

$$\frac{dL}{dw} = \frac{d}{dw}\frac{1}{N}\sum_{n=1}^{N}[-t * log(y) - (1-t) * log(1-y)] =$$

$$= \frac{1}{N}\sum_{n=1}^{N}\frac{d}{dw}[-t*log(y)-(1-t)*log(1-y)] =$$

$$= \frac{1}{N}\sum_{n=1}^{N}[-t\frac{d}{dw}log(y)+(1-t)\frac{d}{dw}log(1-y)]$$

Now, let's derivate $-t\frac{d}{dw}log(y)$:

$$-t\frac{d}{dw}log(y) = -t\frac{d}{dw}log(\sigma(\mathbf{w}^T\mathbf{x}+b)) =$$

$$= -t\frac{d}{dw}log(\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}) =$$

$$= t\frac{d}{dw}log(1+e^{-\mathbf{w}^T\mathbf{x}-b}) =$$

$$= t\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}\frac{d}{dw}(1+e^{-\mathbf{w}^T\mathbf{x}-b}) =$$

$$= t\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}e^{-\mathbf{w}^T\mathbf{x}-b}\frac{d}{dw}(-\mathbf{w}^T\mathbf{x}-b) =$$

$$\frac{te^{-\mathbf{w}^T\mathbf{x}-b}}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}(-x) =$$

$$= -t(1-\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}})x =$$

$$= -t(1-y)x$$

Now, let's derivate $(1-t)\frac{d}{dw}log(1-y)$:

$$(1-t)\frac{d}{dw}log(1-y) =$$

$$= (1-t)\frac{d}{dw}log(1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}) =$$

$$= (1-t)\frac{1}{1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}}\frac{d}{dw}(1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}})$$

To calculate $\frac{d}{dw}\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$:

$$\frac{d}{dw}\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}} = \frac{d}{dw}(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^{-1} =$$

$$= -(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^{-2}\cdot\frac{d}{dw}(1+e^{-(\mathbf{w}^T\mathbf{x}+b)}) =$$

$$= \frac{xe^{-(\mathbf{w}^T\mathbf{x}+b)}}{(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^2}$$

$$= \frac{x(-1+1+e^{-(\mathbf{w}^T\mathbf{x}+b)})}{(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^2} =$$

$$= x\cdot\sigma(\mathbf{w}^T\mathbf{x}+b)[1-\sigma(\mathbf{w}^T\mathbf{x}+b)] =$$

$$= xy(1-y)$$

So:

$$(1-t)\frac{d}{dw}log(1-y) =$$

$$= (1-t)\frac{1}{1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}}\frac{d}{dw}(1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}) =$$

$$(1-t)\frac{1-y}{1-y}xy =$$

$$= (1-t)xy$$

Therefore:

$$\frac{dL}{dw} = \frac{1}{N}[-t(1-y)x + (1-t)xy] =$$

$$= \frac{1}{N}[-tx + tyx + xy - txy] =$$

$$= \frac{1}{N}[-tx + xy] =$$

$$= \frac{1}{N}x(y-t)$$

The gradient of the loss function $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$, in the same principle, so let's derivate $-t\frac{d}{db}log(y)$:

$$-t\frac{d}{db}log(y) = -t\frac{d}{db}log(\sigma(\mathbf{w}^T\mathbf{x}+b)) =$$

$$= -t\frac{d}{db}log(\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}) = t\frac{d}{db}log(1+e^{-\mathbf{w}^T\mathbf{x}-b}) =$$

$$= t\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}\frac{d}{db}(1+e^{-\mathbf{w}^T\mathbf{x}-b}) =$$

$$= t\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}e^{-\mathbf{w}^T\mathbf{x}-b}\frac{d}{db}(-\mathbf{w}^T\mathbf{x}-b) =$$

$$= \frac{te^{-\mathbf{w}^T\mathbf{x}-b}}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}(-1) =$$

$$= -t(1-\frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}}) =$$

$$= -t(1-y) =$$

$$= t(y-1)$$

Now, let's derivate $(1-t)\frac{d}{db}log(1-y)$:

$$(1-t)\frac{d}{db}log(1-y) =$$

$$= (1-t)\frac{d}{db}log(1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}) =$$

$$= (1-t)\frac{1}{1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}}\frac{d}{db}(1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}})$$

To calculate $\frac{d}{db}\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$:

$$= \frac{d}{db}(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^{-1} =$$

$$= -(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^{-2}\frac{d}{db}(1+e^{-(\mathbf{w}^T\mathbf{x}+b)}) =$$

$$= \frac{e^{-(\mathbf{w}^T\mathbf{x}+b)}}{(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^2} =$$

$$= \frac{-1+1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}{(1+e^{-(\mathbf{w}^T\mathbf{x}+b)})^2} =$$

$$= \sigma(\mathbf{w}^T\mathbf{x}+b)[1-\sigma(\mathbf{w}^T\mathbf{x}+b)] =$$

$$= y(1-y)$$

So:

$$(1-t)\frac{d}{db}log(1-y) =$$

$$= (1-t)\frac{1}{1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}}\frac{d}{db}(1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}) =$$

$$= (1-t)\frac{1-y}{1-y}(-y) =$$

$$= (t-1)y$$

Therefore:

$$\frac{dL}{db} = \frac{1}{N}\sum_{n=1}^{N}[t(y-1)-(t-1)y] =$$

$$= \frac{1}{N}\sum_{n=1}^{N}[ty-t-ty+y] =$$

$$= \frac{1}{N}\sum_{n=1}^{N}[y-t]$$

In conclusion:

$$\frac{dL}{dw} = \frac{1}{N}[-t(1-y)x+(1-t)xy] = \frac{1}{N}x(y-t)$$

$$\frac{dL}{db} = \frac{1}{N}\sum_{n=1}^{N}[y-t]$$

2.0.1 Part (c) -- 7%

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

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

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

[ ]:
```
# Your code goes here

'''
r1 = ...
r2 = ...
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
'''

# Parameters
h = 0.0001
x = np.random.randn(1,90)
w = np.zeros(90,)
b = 0
yy = pred(w,b,x)
tt = np.zeros(1)


r1 = derivative_cost(x, yy, tt)
L = cost(pred(w, b, x), tt)
L_h = cost(pred(w, b+h, x), tt)
r2 = (L_h - L) / h
print("The analytical results is -", r1[1])
print("The algorithm results is - ", r2)
```

```
The analytical results is - 0.5
The algorithm results is -  0.5000125000009792
```

2.0.2 Part (d) -- 7%

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

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

'''
r1 = ...
r2 = ...
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
'''

# Parameters
''' Same parameters from (c) '''

r1 = derivative_cost(x, yy, tt)
r2 = np.zeros(90,)
```

```
for i in range (0, 90):
  h_i = np.zeros(90,)
  h_i[i] = h
  L = cost(pred(w, b, x), tt)
  L_h = cost(pred(w + h_i, b, x), tt)
  r2[i,] = (L_h - L) / h


print("The analytical results is -", r1[0])
print("The algorithm results is - ", r2)
```

```
The analytical results is - [-0.78971402 -0.69496745 -0.56474267 -0.41975221
0.71890818 -0.65042157
 -0.31412632  0.31523097 -0.44405119  1.32425402  0.19228957 -1.29947123
  0.68001227 -0.23018085 -0.62014546  0.08495674  0.20559117  0.28843452
 -0.45668896 -0.53930218 -0.29321848 -0.70192325 -0.28247781  0.45367609
  0.20825805  0.44485014 -0.34317127 -1.05494071 -0.60873208 -0.68155272
  0.27144286  0.34008606 -0.56155446 -0.74430716 -0.5190363  -0.48236845
  0.34282392 -0.31409985 -0.44785852 -0.34929071  0.21600445  0.34420671
  0.02558874  0.75439112  0.46013943 -0.46538159  0.10995521 -0.35650224
 -0.72350764  0.13477856  0.32862325 -1.29113574 -0.86578107 -0.2751956
  0.0283088  -0.79403969  0.15449433 -0.52289003 -0.25740942  0.25227907
 -0.81481032  0.00814862 -0.66476423 -0.39861494 -0.02220064 -0.87849933
 -1.19751039  0.03894978 -0.50795995  0.3535717  -0.55979153 -0.98091295
 -0.14772114 -0.17792988 -0.34727354  0.87799884 -0.47143183 -0.66312
  0.12841592  0.21153415 -0.06966958 -0.65378257 -0.4802785   0.00876059
 -0.0107075   0.34987811  0.50612699 -0.30991613  0.35766171  0.24263868]
The algorithm results is -  [-0.78968284 -0.6949433  -0.56472672 -0.4197434
0.71893402 -0.65040042
 -0.31412139  0.31523594 -0.44404133  1.3243417   0.19229142 -1.2993868
  0.68003539 -0.2301782  -0.62012623  0.0849571   0.20559328  0.28843868
 -0.45667854 -0.53928764 -0.29321418 -0.70189861 -0.28247382  0.45368638
  0.20826022  0.44486004 -0.34316538 -1.05488507 -0.60871355 -0.6815295
  0.27144654  0.34009185 -0.5615387  -0.74427946 -0.51902283 -0.48235682
  0.3428298  -0.31409492 -0.44784849 -0.3492846   0.21600678  0.34421263
  0.02558877  0.75441958  0.46015001 -0.46537076  0.10995581 -0.35649588
 -0.72348146  0.13477947  0.32862865 -1.29105239 -0.86574359 -0.27519182
  0.02830884 -0.79400816  0.15449553 -0.52287636 -0.25740611  0.25228225
 -0.81477713  0.00814862 -0.66474213 -0.39860699 -0.02220062 -0.87846075
 -1.19743869  0.03894986 -0.50794705  0.35357795 -0.55977587 -0.98086484
 -0.14772005 -0.17792829 -0.34726751  0.87803739 -0.47142072 -0.66309801
  0.12841674  0.21153639 -0.06966934 -0.6537612  -0.48026697  0.0087606
 -0.01070749  0.34988423  0.5061398  -0.30991133  0.3576681   0.24264162]
```

2.0.3 Part (e) -- 7%

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

[ ]:
```
def run_gradient_descent(train_norm_xs,train_ts,val_norm_xs,val_ts, w0, b0, mu,␣
↪batch_size, 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
 iter_disp_ratio = 10
 val_acc_end = np.zeros(int(max_iters/iter_disp_ratio),)


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

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

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

     # compute the prediction
     yy = pred(w, b, X)
     dL = derivative_cost(X, yy, t)

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

```
    # increment the iteration count
    iter += 1

    # compute and print the *validation* loss and accuracy
    if (iter % iter_disp_ratio == 0):
      y_val = pred(w, b, val_norm_xs)
      val_cost = cost(y_val, val_ts.transpose() )
      val_acc = get_accuracy(y_val, val_ts)
      print("Iter %d. [Val Acc %.0f%%, Loss %f]" % (
            iter, val_acc * 100, val_cost))
      # print(iter/iter_disp_ratio)
      val_acc_end[int(iter/iter_disp_ratio) - 1] = val_acc

    if iter >= max_iters:
      break

    # Think what parameters you should return for further use

  return w, b, val_acc_end
```

2.0.4 Part (f) -- 7%

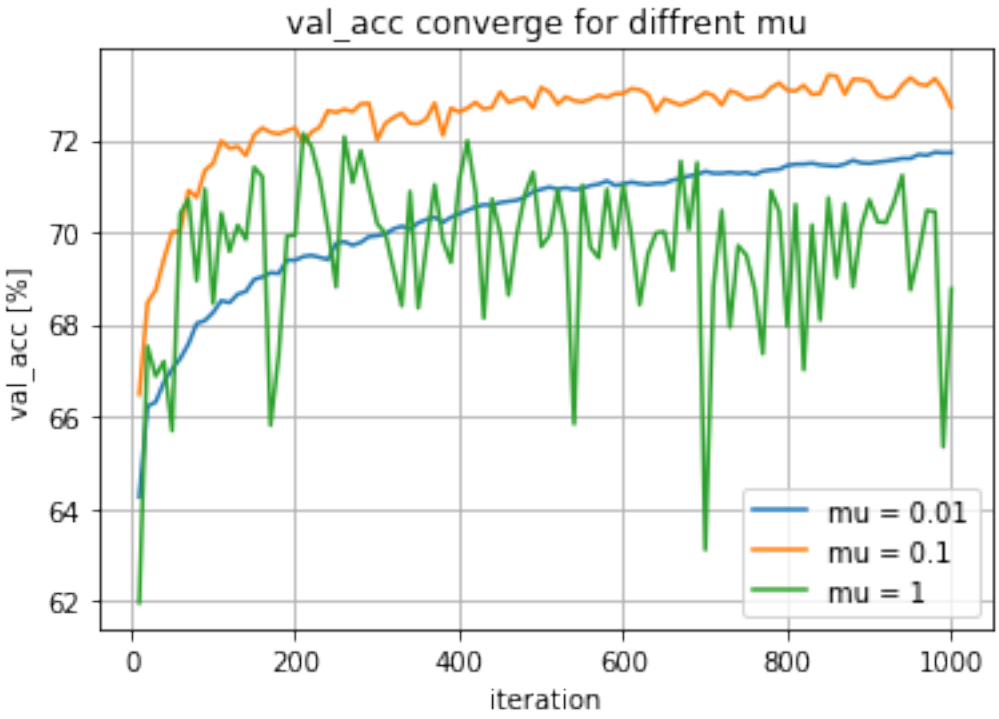
Call `run_gradient_descent` with the weights and biases all initialized to zero. Show that if the learning rate μ is too small, then convergence is slow. Also, show that if μ is too large, then the optimization algorirthm does not converge. The demonstration should be made using plots showing these effects.

[ ]:
```
## mu
w0 = np.zeros(90)
b0 = np.zeros(1)

# Write your code here
val_acc_mu = np.zeros([3,100])
w_new, b_new, val_acc_mu[0,] =␣
 ↪run_gradient_descent(train_norm_xs,train_ts,val_norm_xs,val_ts, w0, b0, mu=0.
 ↪01, batch_size=100, max_iters=1000)
w_new, b_new, val_acc_mu[1,] =␣
 ↪run_gradient_descent(train_norm_xs,train_ts,val_norm_xs,val_ts, w0, b0, mu=0.
 ↪1, batch_size=100, max_iters=1000)
w_new, b_new, val_acc_mu[2,] =␣
 ↪run_gradient_descent(train_norm_xs,train_ts,val_norm_xs,val_ts, w0, b0,␣
 ↪mu=1, batch_size=100, max_iters=1000)
print(val_acc_mu)
```

[ ]:
```
xx = np.arange(10,1010,10)
plt.plot(xx, val_acc_mu[0,] * 100)
plt.plot(xx, val_acc_mu[1,] * 100)
```

```
plt.plot(xx, val_acc_mu[2,] * 100)
plt.legend(['mu = 0.01','mu = 0.1','mu = 1'])
plt.grid()
plt.title('val_acc converge for diffrent mu')
plt.xlabel('iteration')
plt.ylabel('val_acc [%]')
plt.show
```

[ ]: <function matplotlib.pyplot.show>



Explain and discuss your results here:

As we can see in the plot above, μ too small (= 0.01) cause slow converge. In addition, for large μ (= 1) the optimization algorirthm does not converge.

The reason for the first claim is that μ represent the step size, and for small step size, the "path" through the minimum is longer. We can see that the blue (μ=0.01) graph follow the orange curve (μ=0.1, with the proper choice of μ) tendency. The reson for the second term is that the step size is too large, and the algorithm skips the minimun at each epoch. Therefore, the green graph will never converge and always be unstable.

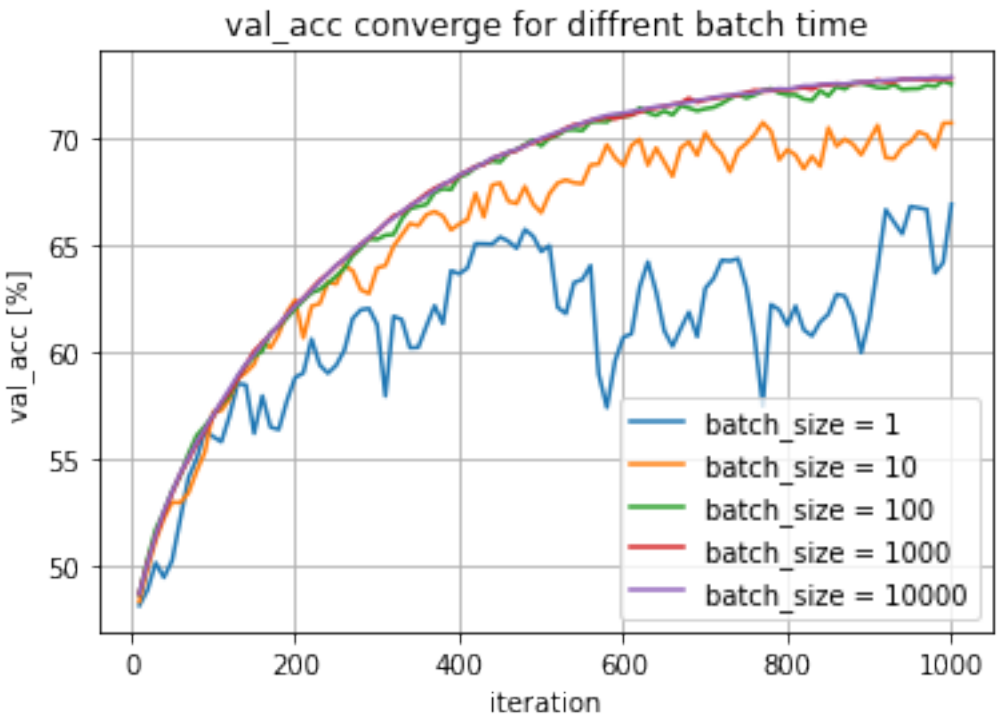
2.0.5 Part (g) -- 7%

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

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

# Write your code here

## batch-size comparison

import time
val_acc_bs = np.zeros([5,100])
runtime = np.zeros([5,])

for i in range (0,5):
  print (i)
  start_time = time.time()
  w_new, b_new, val_acc_bs[i,] =␣
 ↪run_gradient_descent(train_norm_xs,train_ts,val_norm_xs,val_ts, w0, b0, mu=0.
 ↪1, batch_size=10**i, max_iters=1000)
  runtime[i,] = (time.time() - start_time)
```

[ ]:
```
# Plot val_acc converge
xx = np.arange(10,1010,10)
plt.plot(xx, val_acc_bs[0,] * 100)
plt.plot(xx, val_acc_bs[1,] * 100)
plt.plot(xx, val_acc_bs[2,] * 100)
plt.plot(xx, val_acc_bs[3,] * 100)
plt.plot(xx, val_acc_bs[4,] * 100)
plt.legend(['batch_size = 1','batch_size = 10','batch_size = 100','batch_size =␣
 ↪1000','batch_size = 10000'])
plt.grid()
plt.title('val_acc converge for diffrent batch time')
plt.xlabel('iteration')
plt.ylabel('val_acc [%]')
plt.show
```

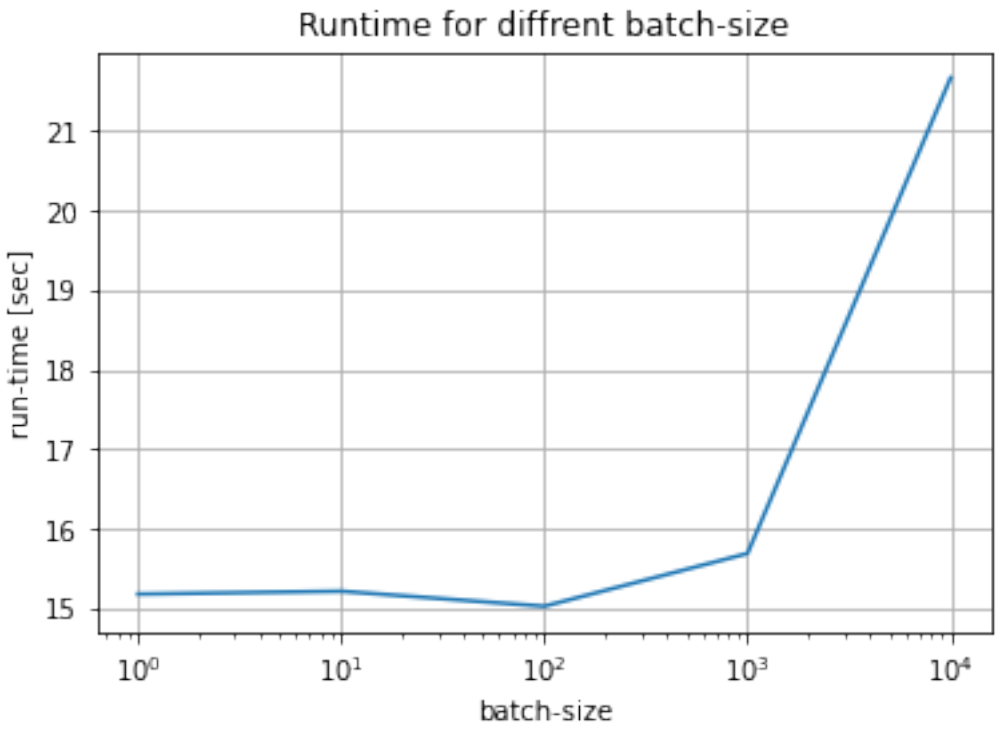
[ ]:
```
<function matplotlib.pyplot.show>
```



[ ]:
```
# Plot runtime
for i in range(0,5):
  print("Run time for batch_size = {}: {} sec".format(10**i,runtime[i,]))

plt.plot([1, 10, 100, 1000, 10000], runtime)
plt.xscale("log")
plt.grid()
plt.title('Runtime for diffrent batch-size')
plt.xlabel('batch-size')
plt.ylabel('run-time [sec]')
```

```
Run time for batch_size = 1: 15.17621922492981 sec
Run time for batch_size = 10: 15.21213150024414 sec
Run time for batch_size = 100: 15.023170471191406 sec
Run time for batch_size = 1000: 15.686829090118408 sec
Run time for batch_size = 10000: 21.665571212768555 sec
```

[ ]:
```
Text(0, 0.5, 'run-time [sec]')
```



[ ]:
```
## optimial value of  w  and  b
w_new, b_new, val_acc =␣
 ↪run_gradient_descent(train_norm_xs,train_ts,val_norm_xs,val_ts, w0, b0, mu=0.
 ↪1, batch_size=100, max_iters=1000)
print(" b = %f" %(b_new) )
print(" w = {}".format(w_new))
```

Explain and discuss your results here:

We discussed in the previous section on the consideration of choosing the step size (μ). The graph in the previous section show various results (good and bad) behavior of the model according to μ selection. Good behavior equivalent to fast converges, like in the orange curve ($\mu = 0.1$) in the "val_acc converge for diffrent mu" graph. Bad behavior equivalent to no conerges at all, like in the green curve in the same graph.

For choosing the right batch size, we run the algorithm for diffrent values and we look at two characteristics: 1. Validation accuracy:

As we can see in the "Validation accuracy to iteration" graph, small batch size cause slow and noisy/unstable converge - Bad behavior.

From $batch - size = 100$ and above, all the batch-sizes have similar behavior. 2. Converge time

From the run-time plot the minimun is at $batch - size = 100$. If it is too small, the algoritm will have to do more for loops, so the run time is a little bit slower. If the batch size is too large, the SGD will be signifly more complex, that will cause long run time.

Therefore, from the explnations above, the optimal results will outcome with those chosen values:

$$\mu = 0.1$$

$$Batch - size = 100$$

2.0.6 Part (h) -- 15%

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

[ ]:
```
# Write your code here
train_ys = pred(w_new, b_new, train_norm_xs)
val_ys = pred(w_new, b_new, val_norm_xs)
test_ys = pred(w_new, b_new, test_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)
```

```
train_acc =  0.7183036631497528  val_acc =  0.71988  test_acc =
0.7129382142165408
```

Explain and discuss your results here:

Those three values are different parameters since the calculation of the accuracy is based on different part of the dataset. The difference between those three values, based on the pourpose of each dataset. - The test accuracy used for final model evaluation - how good the model can handle "new" data.

- The train accuracy is the accuracy that calculated on the data set that used for the fitting of the model. This accuracy indicates how well the model handles the data it was trained on. If the training accuracy is subjectively far higher than test accuracy, this indicates on over-fitting of the model.
- The validation accuracy is the accuracy that calculated on the data set that not used for training of the model, but used during the training process for validating the generalisation ability of the model. This accuracy is importent during the model's parameters optimization.

The results of the model: Although the received accuracy of 71% is not so good result, the results are almost similar, with a difference of less than 1%. This means that the data was balanced well, and the model is general as it can be.

Since the results of the SKlrean API (in the next section) of this exact method (Logistic regression) outcome with almost similar results, there is not much to do to improve the results of this model. Changing the classification method may outcome with better accuracy result.

2.0.7 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.

[ ]:
```
from sklearn.linear_model import LogisticRegression;

model = LogisticRegression(random_state=0).fit(train_norm_xs, train_ts);

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

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

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760:
DataConversionWarning: 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 =  0.7325888594805603  val_acc =  0.73446  test_acc =
0.7267286461359674
```

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

[2]:
```
# This part used to save this notebook as PDF.
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/Colab Notebooks/
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
 ↪texlive-generic-recommended
!jupyter nbconvert --to pdf Assignment1.ipynb
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/MyDrive/Colab Notebooks
Reading package lists... Done
Building dependency tree
Reading state information... Done
texlive-fonts-recommended is already the newest version (2017.20180305-1).
texlive-generic-recommended is already the newest version (2017.20180305-1).
texlive-xetex is already the newest version (2017.20180305-1).
```

```
0 upgraded, 0 newly installed, 0 to remove and 30 not upgraded.
[NbConvertApp] Converting notebook Assignment1.ipynb to pdf
[NbConvertApp] Support files will be in Assignment1_files/
[NbConvertApp] Making directory ./Assignment1_files
[NbConvertApp] Making directory ./Assignment1_files
[NbConvertApp] Making directory ./Assignment1_files
[NbConvertApp] Making directory ./Assignment1_files
[NbConvertApp] Writing 103924 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: [u'xelatex', u'./notebook.tex',
'-quiet']
[NbConvertApp] Running bibtex 1 time: [u'bibtex', u'./notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 178392 bytes to Assignment1.pdf
```