Assignment1

November 27, 2022

Alexander Fok 308669944

Avi Dvir 204423735

Gal Cohen 204675805

1 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.

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.

[3]:
```
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

[4]:
```
load_from_drive = True

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/IntroDeepLearning2022Data/
 ↪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:

[5]:
```
df
```

[5]:
```
        year     var1     var2     var3     var4      var5      var6  \
0       2001 49.94357 21.47114 73.07750  8.74861 -17.40628 -13.09905
1       2001 48.73215 18.42930 70.32679 12.94636 -10.32437 -24.83777
2       2001 50.95714 31.85602 55.81851 13.41693  -6.57898 -18.54940
3       2001 48.24750 -1.89837 36.29772  2.58776   0.97170 -26.21683
4       2001 50.97020 42.20998 67.09964  8.46791 -15.85279 -16.81409
…        …      …        …        …        …        …         …
515340  2006 51.28467 45.88068 22.19582 -5.53319  -3.61835 -16.36914
515341  2006 49.87870 37.93125 18.65987 -3.63581 -27.75665 -18.52988
515342  2006 45.12852 12.65758 -38.72018 8.80882 -29.29985  -2.28706
515343  2006 44.16614 32.38368 -3.34971 -2.49165 -19.59278 -18.67098
515344  2005 51.85726 59.11655 26.39436 -5.46030 -20.69012 -19.95528

            var7      var8     var9  …    var81     var82     var83  \
0      -25.01202 -12.23257  7.83089  …  13.01620 -54.40548  58.99367
1        8.76630  -0.92019 18.76548  …   5.66812 -19.68073  33.04964
2       -3.27872  -2.35035 16.07017  …   3.03800  26.05866 -50.92779
```

```
3         5.05097 -10.34124   3.55005  …  34.57337 -171.70734  -16.96705
4       -12.48207  -9.37636  12.63699  …   9.92661  -55.95724   64.92712
…             …         …       …   …        …          …          …
515340    2.12652   5.18160  -8.66890  …   4.81440   -3.75991  -30.92584
515341    7.76108   3.56109  -2.50351  …  32.38589  -32.75535  -61.05473
515342  -18.40424 -22.28726  -4.52429  … -18.73598  -71.15954 -123.98443
515343    8.78428   4.02039 -12.01230  …  67.16763  282.77624   -4.63677
515344   -6.72771   2.29590  10.31018  … -11.50511  -69.18291   60.58456

            var84     var85     var86      var87     var88      var89  \
0        15.37344   1.11144 -23.08793   68.40795  -1.82223  -27.46348
1        42.87836  -9.90378 -32.22788   70.49388  12.04941   58.43453
2        10.93792  -0.07568  43.20130 -115.00698  -0.05859   39.67068
3       -46.67617 -12.51516  82.58061  -72.08993   9.90558  199.62971
4       -17.72522  -1.49237  -7.50035   51.76631   7.88713   55.66926
…             …         …         …        …         …          …
515340   26.33968  -5.03390  21.86037 -142.29410   3.42901  -41.14721
515341   56.65182  15.29965  95.88193  -10.63242  12.96552   92.11633
515342  121.26989  10.89629  34.62409 -248.61020  -6.07171   53.96319
515343  144.00125  21.62652 -29.72432   71.47198  20.32240   14.83107
515344   28.64599  -4.39620 -64.56491  -45.61012  -5.51512   32.35602

            var90
0         2.26327
1        26.92061
2        -0.66345
3        18.85382
4        28.74903
…             …
515340  -15.46052
515341   10.88815
515342   -8.09364
515343   39.74909
515344   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.

[6]:
```
df["year"] = df["year"].map(lambda x: int(x > 2000))
```

[7]:
```
df.head(20)
```

[7]:
```
   year       var1       var2      var3      var4       var5       var6  \
0     1   49.94357   21.47114  73.07750   8.74861  -17.40628  -13.09905
```

1	1	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777
2	1	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940
3	1	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683
4	1	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409
5	1	50.54767	0.31568	92.35066	22.38696	-25.51870	-19.04928
6	1	50.57546	33.17843	50.53517	11.55217	-27.24764	-8.78206
7	1	48.26892	8.97526	75.23158	24.04945	-16.02105	-14.09491
8	1	49.75468	33.99581	56.73846	2.89581	-2.92429	-26.44413
9	1	45.17809	46.34234	-40.65357	-2.47909	1.21253	-0.65302
10	1	39.13076	-23.01763	-36.20583	1.67519	-4.27101	13.01158
11	1	37.66498	-34.05910	-17.36060	-26.77781	-39.95119	-20.75000
12	1	26.51957	-148.15762	-13.30095	-7.25851	17.22029	-21.99439
13	1	37.68491	-26.84185	-27.10566	-14.95883	-5.87200	-21.68979
14	0	39.11695	-8.29767	-51.37966	-4.42668	-30.06506	-11.95916
15	1	35.05129	-67.97714	-14.20239	-6.68696	-0.61230	-18.70341
16	1	33.63129	-96.14912	-89.38216	-12.11699	13.77252	-6.69377
17	0	41.38639	-20.78665	51.80155	17.21415	-36.44189	-11.53169
18	0	37.45034	11.42615	56.28982	19.58426	-16.43530	2.22457
19	0	39.71092	-4.92800	12.88590	-11.87773	2.48031	-16.11028

	var7	var8	var9	…	var81	var82	var83	\
0	-25.01202	-12.23257	7.83089	…	13.01620	-54.40548	58.99367	
1	8.76630	-0.92019	18.76548	…	5.66812	-19.68073	33.04964	
2	-3.27872	-2.35035	16.07017	…	3.03800	26.05866	-50.92779	
3	5.05097	-10.34124	3.55005	…	34.57337	-171.70734	-16.96705	
4	-12.48207	-9.37636	12.63699	…	9.92661	-55.95724	64.92712	
5	20.67345	-5.19943	3.63566	…	6.59753	-50.69577	26.02574	
6	-12.04282	-9.53930	28.61811	…	11.63681	25.44182	134.62382	
7	8.11871	-1.87566	7.46701	…	18.03989	-58.46192	-65.56438	
8	1.71392	-0.55644	22.08594	…	18.70812	5.20391	-27.75192	
9	-6.95536	-12.20040	17.02512	…	-4.36742	-87.55285	-70.79677	
10	8.05718	-8.41088	6.27370	…	32.86051	-26.08461	-186.82429	
11	-0.10231	-0.89972	-1.30205	…	11.18909	45.20614	53.83925	
12	5.51947	3.48418	2.61738	…	23.80442	251.76360	18.81642	
13	4.87374	-18.01800	1.52141	…	-67.57637	234.27192	-72.34557	
14	-0.85322	-8.86179	11.36680	…	42.22923	478.26580	-10.33823	
15	-1.31928	-9.46370	5.53492	…	10.25585	94.90539	15.95689	
16	-33.36843	-24.81437	21.22757	…	49.93249	-14.47489	40.70590	
17	11.75252	-7.62428	-3.65488	…	50.37614	-40.48205	48.07805	
18	1.02668	-7.34736	-0.01184	…	-22.46207	-25.77228	-322.42841	
19	-16.40421	-8.29657	9.86817	…	11.92816	-73.72412	16.19039	

	var84	var85	var86	var87	var88	var89	var90
0	15.37344	1.11144	-23.08793	68.40795	-1.82223	-27.46348	2.26327
1	42.87836	-9.90378	-32.22788	70.49388	12.04941	58.43453	26.92061
2	10.93792	-0.07568	43.20130	-115.00698	-0.05859	39.67068	-0.66345
3	-46.67617	-12.51516	82.58061	-72.08993	9.90558	199.62971	18.85382

4	-17.72522	-1.49237	-7.50035	51.76631	7.88713	55.66926	28.74903
5	18.94430	-0.33730	6.09352	35.18381	5.00283	-11.02257	0.02263
6	21.51982	8.17570	35.46251	11.57736	4.50056	-4.62739	1.40192
7	46.99856	-4.09602	56.37650	-18.29975	-0.30633	3.98364	-3.72556
8	17.22100	-0.85210	-15.67150	-26.36257	5.48708	-9.13495	6.08680
9	76.57355	-7.71727	3.26926	-298.49845	11.49326	-89.21804	-15.09719
10	113.58176	9.28727	44.60282	158.00425	-2.59543	109.19723	23.36143
11	2.59467	-4.00958	-47.74886	-170.92864	-5.19009	8.83617	-7.16056
12	157.09656	-27.79449	-137.72740	115.28414	23.00230	-164.02536	51.54138
13	-362.25101	-25.55019	-89.08971	-891.58937	14.11648	-1030.99180	99.28967
14	-103.76858	39.19511	-98.76636	-122.81061	-2.14942	-211.48202	-12.81569
15	-98.15732	-9.64859	-93.52834	-95.82981	20.73063	-562.07671	43.44696
16	58.63692	8.81522	27.28474	5.78046	3.44539	259.10825	10.28525
17	-7.62399	6.51934	-30.46090	-53.87264	4.44627	58.16913	-0.02409
18	-146.57408	13.61588	92.22918	-439.80259	25.73235	157.22967	38.70617
19	9.79606	9.71693	-9.90907	-20.65851	2.34002	-31.57015	1.58400

[20 rows x 91 columns]

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.)

[8]:
```
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
#----------------------------------------------------------------------------------------------
# It would be problematic to have the some songs from an artist in the training␣
↪set and other songs from the same artist in the test set
# since in this specific case, we use dataset of songs, artists have common␣
↪ground that can be repeated in other songs
# from the same artist. so if different songs from same artist will show up in␣
↪the training and validation sets
```

```
# it could affect our model projection, it can tilt the classification to␣
↪correct answer duo to
# same artist style song that was in our training set.
# The main problem here is that it is very likely that most of the songs of the␣
↪same artist are written in the same centure.
# So if we have songs from the same artist in the training and test data set,␣
↪we actually do not test the model performance on the unknown songs.
```

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.

[9]:
```
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.)

[10]:
```
# Write your explanation here
#----------------------------------------------------------------------------------------------------
# it would be improper to compute and use test set means and standart deviation␣
↪for a few reasons:
# 1) We want to train the model based on the training set distribution
#       We can not be sure that the test set distribution statistical parameters␣
↪are the same as of the test set.
# 2) while we are training our model the data normalization based on␣
↪independent and identically distributed samples (dynamic range)
# 3) when we put our model under test, the input data should be the same type␣
↪so it can be captured in the right way by our model
# 4) the test samples or its statistic values are not exposed to us at␣
↪inference phase
#     so during the training phase, we assume no knowledge about the test␣
↪dataset.
#----------------------------------------------------------------------------------------------------
```

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.

[11]:
```
# shuffle the training set
reindex = np.random.permutation(len(train_xs))
train_xs = train_xs[reindex]
train_norm_xs = train_norm_xs[reindex]
train_ts = train_ts[reindex]

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

# Write your explanation here
#-----------------------------------------------------------------------------------
# we should limit how many times we use the test set, and that we should use␣
↪the validation set during the model building process
# due to the biased evaluations and risk of model overfitting.
# It means that if we use test set many times, the trained model performance␣
↪can 'fit' the test set distribution and eventually will perform bad with␣
↪wild data.
# Additional use of the validation set is for tuning model hyperparameters,␣
↪such as learning rate of mini batchsize.
# The model has to be trained with totally different training dataset.
# in case we don't enforce it, we might affect the model generalization measure␣
↪and it will lead to biased evaluations.
# This is the reason we need to use the validation set, in our case its a␣
↪little group of songs from training set
# and not used while training our model, and with that we evaluate our model␣
↪perfomance and if learning curve is correct while examine the overfitting␣
↪status.
#-----------------------------------------------------------------------------------
↪
```

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).

[12]:
```
def sigmoid(z):
  return 1 / (1 + np.exp(-z))

def cross_entropy(t, y):
  eps=1e-9
```

```python
  return -t * np.log(y + eps) - (1 - t) * np.log(1 - y + eps)

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 $= 1$.

[13]:
```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
  y = np.dot(X,w) + b
  return sigmoid(y)

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

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

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.

[14]:
```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
    """
    # Your code goes here
    errors = y - t
    # Calculate gradiant over w
    grad_w = np.dot(X.T, errors)
    grad_w /= X.shape[0]
    # Calculate gradiant over b
    grad_b = np.mean(errors)
    return grad_w, grad_b
```

2 Explenation on Gradients

$\mathbf{y}, \mathbf{t}, \mathbf{b} \in \mathbb{R}^{\mathbf{N}},\ \mathbf{X} \in \mathbb{R}^{\mathbf{N} \times \mathbf{d}},\ \mathbf{w} \in \mathbb{R}^{\mathbf{d}}.$

$$\mathcal{L}CE(\mathbf{y}, \mathbf{t}) = \frac{\mathbf{1}}{\mathbf{N}} \sum \mathbf{n} = \mathbf{1}^{\mathbf{N}} \big(-\mathbf{t_n} \cdot \log(\mathbf{y_n}) - (\mathbf{1} - \mathbf{t_n}) \cdot \log(\mathbf{1} - \mathbf{y_n}) \big)$$

$$\frac{\partial \mathcal{L}CE}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}CE}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{w}} \in \mathbb{R}^{1 \times d}$$

$$\frac{\partial \mathcal{L}CE}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}CE}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{b}} \in \mathbb{R}^{1 \times N}$$

Finding:

$$\left[\frac{\partial \mathcal{L}CE}{\partial \mathbf{w}}\right]_i \text{ and } \left[\frac{\partial \mathcal{L}CE}{\partial \mathbf{b}}\right]_i$$

$$\frac{\partial \mathcal{L}CE}{\partial \mathbf{y}} = \left[\frac{\partial \mathcal{L}CE}{\partial y_1}, \ \ldots, \ \frac{\partial \mathcal{L}_{CE}}{\partial y_N}\right] \in \mathbb{R}^{1 \times N}$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}CE}{\partial y_i} &= \frac{\partial}{\partial y_i}\frac{1}{N}\sum n = 1^N\big(-t_n \cdot \log(y_n) - (1-t_n)\cdot \log(1-y_n)\big) & (2)\\
&= \frac{1}{N}\left(\frac{-t_i}{y_i} - \frac{-(1-t_i)}{1-y_i}\right) & (3)\\
&= \frac{1}{N}\frac{y_i - t_i}{y_i(1-y_i)} & (4)
\end{aligned}
$$

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

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

Then we get:

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

for example:

$$
\left[\frac{\partial y}{\partial z}\right]_i = y_i(1-y_i)
$$

note

$\mathbf{z} = \mathbf{X}\mathbf{w} + \mathbf{b}$

$$
\begin{aligned}
\frac{\partial \mathbf{z}}{\partial \mathbf{w}} &= \mathbf{X} \in \mathbb{R}^{N\times d} & (5)\\
\frac{\partial \mathbf{z}}{\partial \mathbf{b}} &= \mathbf{I}_N \in \mathbb{R}^{N\times N} & (6)
\end{aligned}
$$

note

$\mathbf{X} = [\mathbf{x}_1^T, \ldots, \mathbf{x}_N^T]^T$ and $\mathbf{I}_N = [\mathbf{e}_1^T, \ldots, \mathbf{e}_N^T]^T$, where $\mathbf{x}_i, \mathbf{e}_i \in \mathbb{R}^d$.

$$
\begin{aligned}
\left[\frac{\partial \mathcal{L}CE}{\partial \mathbf{w}}\right]_i &= \frac{\partial \mathcal{L}CE}{\partial y_i}\cdot\left[\frac{\partial bfy}{\partial \mathbf{z}}\right]_i\cdot\left[\frac{\partial bfz}{\partial \mathbf{w}}\right]_i & (7)\\
&= \frac{1}{N}\frac{y_i - t_i}{y_i(1-y_i)}\cdot y_i(1-y_i)\cdot \mathbf{x_i} & (8)\\
&= \frac{1}{N}(y_i - t_i)\cdot \mathbf{x}_i & (9)\\
\left[\frac{\partial \mathcal{L}CE}{\partial \mathbf{b}}\right]_i &= \frac{\partial \mathcal{L}CE}{\partial y_i}\cdot\left[\frac{\partial bfy}{\partial \mathbf{z}}\right]_i\cdot\left[\frac{\partial bfz}{\partial \mathbf{b}}\right]_i & (10)\\
&= \frac{1}{N}\frac{y_i - t_i}{y_i(1-y_i)}\cdot y_i(1-y_i)\cdot \mathbf{x_i} & (11)\\
&= \frac{1}{N}(y_i - t_i)\cdot \mathbf{e}_i & (12)
\end{aligned}
$$

$$\frac{\partial \mathcal{L}CE}{\partial \mathbf{w}} = \left[\frac{\partial \mathcal{L}CE}{\partial w_1}, \ldots, \frac{\partial \mathcal{L}_{CE}}{\partial w_d}\right] = \frac{1}{N}(\mathbf{y}-\mathbf{t})\mathbf{X} \in \mathbb{R}^{1\times d} \tag{13}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \left[\frac{\partial \mathcal{L}CE}{\partial b_1}, \ldots, \frac{\partial \mathcal{L}CE}{\partial b_N}\right] = \frac{1}{N}(\mathbf{y}-\mathbf{t}) \in \mathbb{R}^{1\times N} \tag{14}$$

Now, since we are required in the function `derivative_cost` have to return $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$ scalar, we can write $\mathbf{b} = \beta\mathbf{1}$, where $\beta \in \mathbb{R}$ is a salar and $\mathbf{1} \in \mathbb{R}^{N\times 1}$ is a vector of ones.

$\frac{\partial \mathcal{L}}{\partial \beta}$:

$$\frac{\partial \mathbf{b}}{\partial \beta} = \frac{\partial \beta\mathbf{1}}{\partial \beta} = \mathbf{1} \in \mathbb{R}^{N\times 1} \tag{15}$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = \frac{\partial \mathcal{L}}{\partial \mathbf{b}} \cdot \frac{\partial \mathbf{b}}{\partial \beta} = \frac{1}{N}(\mathbf{y}-\mathbf{t})\mathbf{1} = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i) \in \mathbb{R} \tag{16}$$

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.

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.

[15]:
```
# Your code goes here
# Random seed
np.random.seed(100)
w = 0.01*np.random.randn(90)
b = 1.0
i = 1
h = 1e-9
batch_size = 10
# Pick test datachunk
X = train_norm_xs[i:(i + batch_size)]
t = train_ts[i:(i + batch_size), 0]
```

```python
# Calculate predictions
y = pred(w, b, X)
y_h = pred(w, b+h, X)
# Evaluate costs
r1 = (cost(y_h,t) - cost(y,t))/h
_, r2 = derivative_cost(X, y, t)

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

```
The analytical results is - 0.427001323188847
The algorithm results is -  0.4270012676283928
```

2.0.2 Part (d) -- 7%

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

[16]:
```python
# Your code goes here. You might find this below code helpful: but it's
# up to you to figure out how/why, and how to modify the code

w = np.zeros(90)
b = 1.0
i = 1
h = 1e-9
batch_size = 10
# Pick test datachunk
X = train_norm_xs[i:(i + batch_size)]
t = train_ts[i:(i + batch_size), 0]

# Calculate predictions
r1 = np.zeros(X.shape[1])
y = pred(w, b, X)
for j in range(X.shape[1]):
  eps = np.zeros(X.shape[1])
  eps[j] = h
  y_h = pred(w + eps, b, X)
  # Evaluate costs
  r1[j] = (cost(y_h,t) - cost(y,t))/h

r2, _ = derivative_cost(X, y, t)

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

```
The analytical results is - [ 0.12654811  0.3168199   0.1081335  -0.32482128
-0.16608559 -0.11990564
```

```
 -0.27350699  0.02671241  0.02925149  0.03246781  0.133068    0.3472258
 -0.146676   -0.36502379 -0.25764235 -0.34160919 -0.06213785 -0.28676261
 -0.15724133 -0.38691783 -0.12411916 -0.23055979 -0.35665404 -0.13828894
 -0.12188006  0.10482704  0.03167711  0.18111268  0.03718958 -0.10964873
  0.28550007 -0.19212631  0.21292257 -0.00886335  0.30269387 -0.13725665
 -0.07340817  0.11727219  0.05606293  0.00063727 -0.09536238 -0.11899259
  0.03508083  0.06230705 -0.03031397 -0.34701708 -0.19158475 -0.18646751
  0.19588819  0.26412206 -0.21799251 -0.05948042 -0.17734236 -0.01350142
 -0.06344658 -0.08764123  0.00518208 -0.1379179  -0.10237255 -0.12146217
 -0.04314904  0.0213809  -0.24017899 -0.07941137  0.05804646 -0.29632941
 -0.09511392 -0.10565349  0.08083312 -0.16351831  0.14524981  0.02701883
 -0.05330336  0.01557954 -0.06499157  0.03919487 -0.12067991 -0.01551514
  0.09023826 -0.00818856  0.04140999  0.13172197 -0.05929324 -0.16408785
  0.00981415  0.13651835 -0.0560556  -0.05535439 -0.06262946 -0.05745937]
The algorithm results is -  [ 0.12654773  0.31681967  0.10813339 -0.32482133
-0.16608581 -0.11990588
 -0.27350716  0.02671216  0.02925137  0.03246736  0.13306767  0.34722559
 -0.14667602 -0.36502388 -0.25764271 -0.34160962 -0.06213833 -0.28676284
 -0.15724156 -0.38691775 -0.12411922 -0.23055983 -0.35665409 -0.13828915
 -0.12188046  0.10482706  0.03167687  0.18111267  0.03718951 -0.1096489
  0.28549986 -0.19212642  0.21292234 -0.0088633   0.30269369 -0.13725662
 -0.07340841  0.11727185  0.05606284  0.00063694 -0.09536252 -0.11899292
  0.03508055  0.06230674 -0.03031421 -0.34701706 -0.19158512 -0.18646784
  0.19588799  0.26412197 -0.21799278 -0.05948068 -0.17734241 -0.0135015
 -0.06344685 -0.08764129  0.00518196 -0.13791819 -0.10237268 -0.12146242
 -0.04314933  0.02138076 -0.24017917 -0.07941146  0.05804657 -0.29632943
 -0.09511402 -0.10565361  0.08083291 -0.16351855  0.14524971  0.02701857
 -0.05330334  0.01557958 -0.06499176  0.03919452 -0.12068018 -0.01551514
  0.09023822 -0.00818877  0.04140968  0.13172179 -0.05929318 -0.16408819
  0.00981402  0.13651824 -0.05605585 -0.05535464 -0.06262945 -0.0574595 ]
```

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:

[17]:
```python
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_xs, train_norm_xs, train_ts
  val_costs = []
  val_accs = []


  for iter in range(max_iters):
    # 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]


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

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

      # compute the prediction
      y = pred(w, b, X)
      # update w and b
      dw, db = derivative_cost(X, y, t)
      w -= mu*dw
      b -= mu*db

    # compute and print the *validation* loss and accuracy
    X_val = val_norm_xs
    t_val = val_ts[:, 0]
    y_val = pred(w, b, X_val)
    val_cost = cost(y_val, t_val)
    val_acc = get_accuracy(y_val, t_val)
    val_costs.append(val_cost)
    val_accs.append(val_acc)
    if (iter % 2 == 0):
```

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

        # Think what parameters you should return for further use
    return w, b, val_costs, val_accs
```

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.

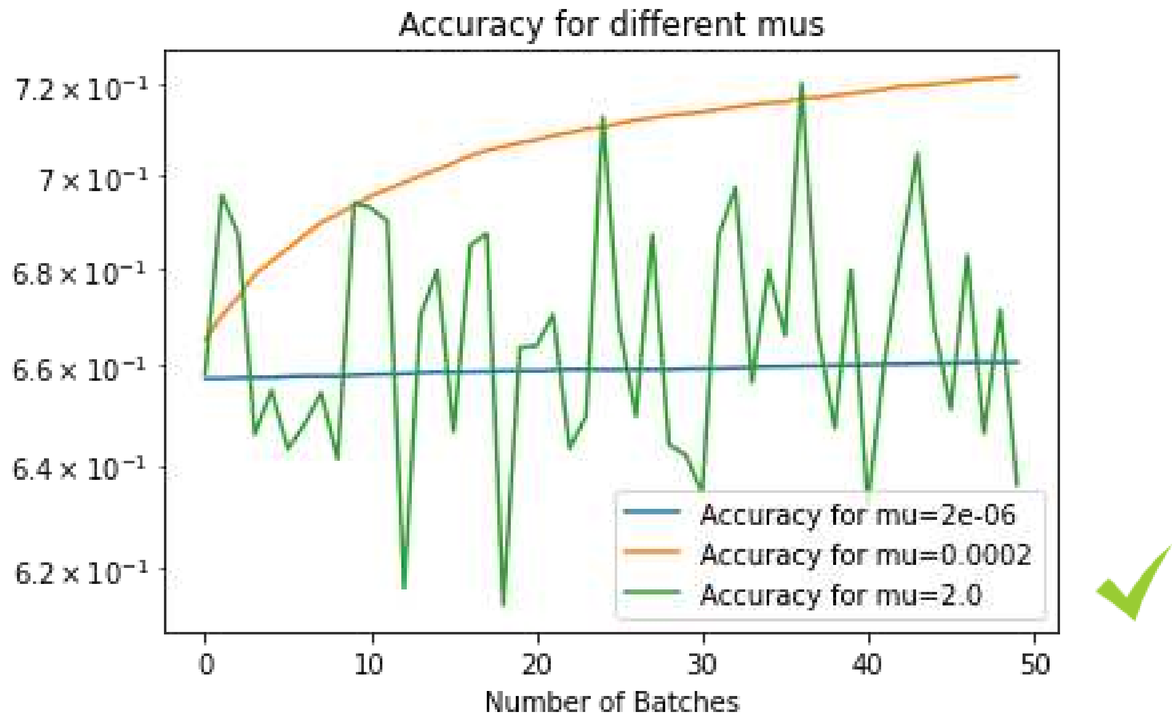
[18]:
```
# plot function - used to draw plots
def plot(title, data_1, legend_1, xlabel = "Number of Iterations", ylabel =␣
 ↪"Error", data_2 = None, legend_2 = None, data_3 = None, legend_3 = None,␣
 ↪loc="upper right"):
    plt.semilogy(data_1, label=legend_1)
    if data_2 is not None:
        plt.semilogy(data_2, label=legend_2)
    if data_3 is not None:
        plt.semilogy(data_3, label=legend_3)
    plt.legend(loc=loc)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    plt.show()
```

[19]:
```
#w0 = np.random.randn(90)
#b0 = np.random.randn(1)[0]
w0 = np.zeros(90)
b0 = np.zeros(1)[0]
mus=[2e-6, 2e-4, 2.0]
# Write your code here
val_costss = []
val_accss = []
for mu in mus:
  w, b, val_costs, val_accs = run_gradient_descent(w0, b0, mu, batch_size=500,␣
 ↪max_iters=50)
  val_costss.append(val_costs)
  val_accss.append(val_accs)
  # Plot the results
#  plot(f"Accuracy for mu={mu}", val_accs, "val_accs", xlabel = "Number of␣
 ↪Batches", ylabel = "%")
```

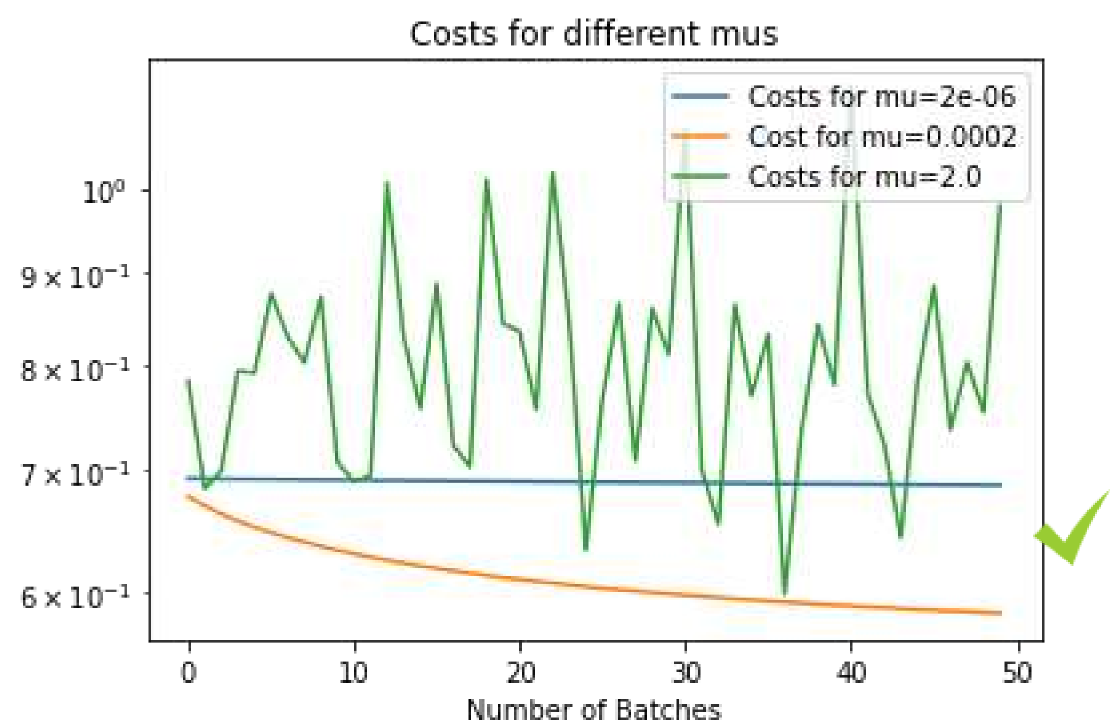
```python
#  plot(f"Costs for mu={mu}", val_costs, "val_costs", xlabel = "Number of␣
↪Batches", ylabel = "%")
plot(f"Accuracy for different mus", val_accss[0], f"Accuracy for mu={mus[0]}",␣
↪xlabel = "Number of Batches", ylabel = "", data_2=val_accss[1],␣
↪legend_2=f"Accuracy for mu={mus[1]}", data_3=val_accss[2],␣
↪legend_3=f"Accuracy for mu={mus[2]}", loc="lower right")
plot(f"Costs for different mus", val_costss[0], f"Costs for mu={mus[0]}",␣
↪xlabel = "Number of Batches", ylabel = "", data_2=val_costss[1],␣
↪legend_2=f"Cost for mu={mus[1]}", data_3=val_costss[2], legend_3=f"Costs for␣
↪mu={mus[2]}")
```

```
Iter 0. [Val Acc 66%, Loss 0.693020]
Iter 2. [Val Acc 66%, Loss 0.692768]
Iter 4. [Val Acc 66%, Loss 0.692517]
Iter 6. [Val Acc 66%, Loss 0.692267]
Iter 8. [Val Acc 66%, Loss 0.692019]
Iter 10. [Val Acc 66%, Loss 0.691772]
Iter 12. [Val Acc 66%, Loss 0.691526]
Iter 14. [Val Acc 66%, Loss 0.691281]
Iter 16. [Val Acc 66%, Loss 0.691038]
Iter 18. [Val Acc 66%, Loss 0.690796]
Iter 20. [Val Acc 66%, Loss 0.690555]
Iter 22. [Val Acc 66%, Loss 0.690315]
Iter 24. [Val Acc 66%, Loss 0.690077]
Iter 26. [Val Acc 66%, Loss 0.689840]
Iter 28. [Val Acc 66%, Loss 0.689604]
Iter 30. [Val Acc 66%, Loss 0.689369]
Iter 32. [Val Acc 66%, Loss 0.689136]
Iter 34. [Val Acc 66%, Loss 0.688904]
Iter 36. [Val Acc 66%, Loss 0.688672]
Iter 38. [Val Acc 66%, Loss 0.688442]
Iter 40. [Val Acc 66%, Loss 0.688213]
Iter 42. [Val Acc 66%, Loss 0.687986]
Iter 44. [Val Acc 66%, Loss 0.687759]
Iter 46. [Val Acc 66%, Loss 0.687534]
Iter 48. [Val Acc 66%, Loss 0.687309]
Iter 0. [Val Acc 67%, Loss 0.677525]
Iter 2. [Val Acc 67%, Loss 0.662643]
Iter 4. [Val Acc 68%, Loss 0.651723]
Iter 6. [Val Acc 69%, Loss 0.643121]
Iter 8. [Val Acc 69%, Loss 0.636058]
Iter 10. [Val Acc 70%, Loss 0.630101]
Iter 12. [Val Acc 70%, Loss 0.624980]
Iter 14. [Val Acc 70%, Loss 0.620514]
Iter 16. [Val Acc 70%, Loss 0.616578]
Iter 18. [Val Acc 71%, Loss 0.613074]
Iter 20. [Val Acc 71%, Loss 0.609928]
```

```
Iter 22. [Val Acc 71%, Loss 0.607086]
Iter 24. [Val Acc 71%, Loss 0.604502]
Iter 26. [Val Acc 71%, Loss 0.602140]
Iter 28. [Val Acc 71%, Loss 0.599968]
Iter 30. [Val Acc 71%, Loss 0.597964]
Iter 32. [Val Acc 72%, Loss 0.596108]
Iter 34. [Val Acc 72%, Loss 0.594381]
Iter 36. [Val Acc 72%, Loss 0.592772]
Iter 38. [Val Acc 72%, Loss 0.591266]
Iter 40. [Val Acc 72%, Loss 0.589855]
Iter 42. [Val Acc 72%, Loss 0.588527]
Iter 44. [Val Acc 72%, Loss 0.587277]
Iter 46. [Val Acc 72%, Loss 0.586096]
Iter 48. [Val Acc 72%, Loss 0.584980]
Iter 0. [Val Acc 66%, Loss 0.783830]
Iter 2. [Val Acc 69%, Loss 0.699722]
Iter 4. [Val Acc 65%, Loss 0.791944]
Iter 6. [Val Acc 65%, Loss 0.828709]
Iter 8. [Val Acc 64%, Loss 0.871666]
Iter 10. [Val Acc 69%, Loss 0.690476]
Iter 12. [Val Acc 62%, Loss 1.007000]
Iter 14. [Val Acc 68%, Loss 0.757162]
Iter 16. [Val Acc 69%, Loss 0.722689]
Iter 18. [Val Acc 61%, Loss 1.011236]
Iter 20. [Val Acc 66%, Loss 0.834598]
Iter 22. [Val Acc 64%, Loss 1.019830]
Iter 24. [Val Acc 71%, Loss 0.633106]
Iter 26. [Val Acc 65%, Loss 0.864502]
Iter 28. [Val Acc 64%, Loss 0.859224]
Iter 30. [Val Acc 63%, Loss 1.079111]
Iter 32. [Val Acc 70%, Loss 0.654011]
Iter 34. [Val Acc 68%, Loss 0.769511]
Iter 36. [Val Acc 72%, Loss 0.599105]
Iter 38. [Val Acc 65%, Loss 0.842001]
Iter 40. [Val Acc 63%, Loss 1.138255]
Iter 42. [Val Acc 68%, Loss 0.724019]
Iter 44. [Val Acc 67%, Loss 0.782657]
Iter 46. [Val Acc 68%, Loss 0.736886]
Iter 48. [Val Acc 67%, Loss 0.753650]
```



Accuracy for different mus
7.2×10^{-1}
7×10^{-1}
6.8×10^{-1}
6.6×10^{-1}
6.4×10^{-1}
6.2×10^{-1}
0
10
20
30
40
50
Number of Batches
Accuracy for mu=2e-06
Accuracy for mu=0.0002
Accuracy for mu=2.0



Costs for different mus
10^{0}
9×10^{-1}
8×10^{-1}
7×10^{-1}
6×10^{-1}
0
10
20
30
40
50
Number of Batches
Costs for mu=2e-06
Cost for mu=0.0002
Costs for mu=2.0

Explain and discuss your results here: After analyzing the results, we can see that: 1. When we use small learning rate (mu) the training results in a slow learning process that converges slowly. For example the loss is changing slowly. 2. When we use the good learning rate, we see the stable learning curve with loss monotonically decreasing and accuracy monotonically increasing over training iterations. 3. When we use large learning rate the training doesn't converge at all, and the loss and accuracy values are bouncing around the same numbers.

2.0.5 Part (g) -- 7%

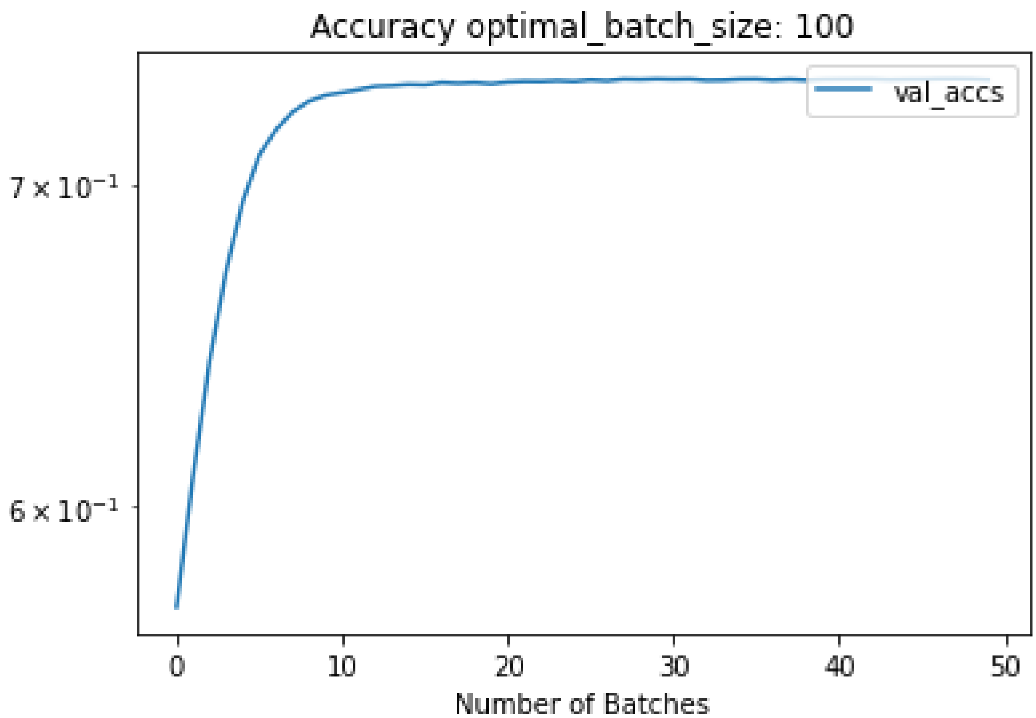
Find the optimial value of **w** and b using your code. Explain how you chose the learning rate μ and the batch size. Show plots demostrating good and bad behaviours.

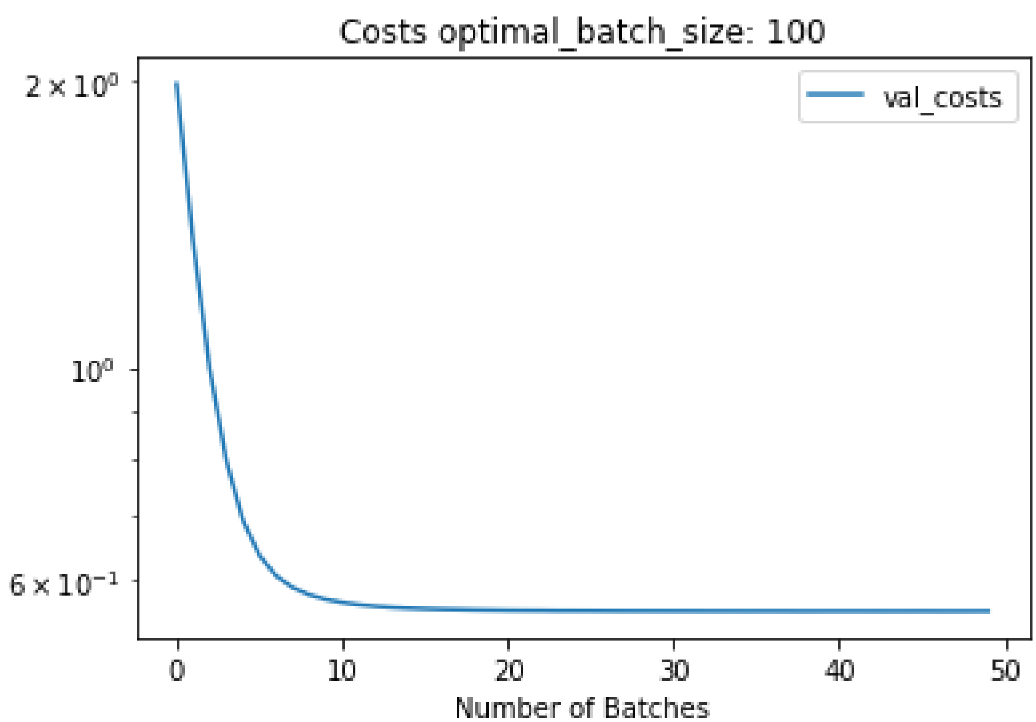
[20]:
```
w0 = np.random.randn(90)
b0 = np.random.randn(1)[0]
#mus=[2e-3, 2e-2, 2.0]
mu=2e-2
mu=2e-3
minimal_cost = 0
minimal_cost_diff = 0

max_iters=50

optimal_w = w0
optimal_b = b0
optimal_batch_size = 0
optimal_val_costs = []
optimal_val_accs = []

# Write your code here
for batch_size in range(100, 1000, 300):
  w, b, val_costs, val_accs = run_gradient_descent(w0, b0, mu, batch_size,␣
↪max_iters)
  cost_diff = val_costs[0] - val_costs[len(val_costs)-1]
  if (minimal_cost < val_costs[len(val_costs)-1] and cost_diff >␣
↪minimal_cost_diff):
    print(f"minimal_cost: {minimal_cost_diff}, cost_diff: {cost_diff},␣
↪batch_size: {batch_size}")
    minimal_cost_diff = cost_diff
    minimal_cost = val_costs[len(val_costs)-1]
#  if (np.mean(val_costs) < minimal_cost):
#    print(f"minimal_cost: {minimal_cost}, np.mean(val_costs): {np.
↪mean(val_costs)}, batch_size: {batch_size}")
#    minimal_cost = np.mean(val_costs)
    optimal_w = w
    optimal_b = b
    optimal_batch_size = batch_size
    optimal_val_costs = val_costs
```

```python
    optimal_val_accs = val_accs

print(f"optimal_batch_size: {optimal_batch_size}")
# Plot the results
plot(f"Accuracy optimal_batch_size: {optimal_batch_size}", optimal_val_accs,
 ↪"val_accs", xlabel = "Number of Batches", ylabel = "")
plot(f"Costs optimal_batch_size: {optimal_batch_size}", optimal_val_costs,
 ↪"val_costs", xlabel = "Number of Batches", ylabel = "")
```

```
Iter 0. [Val Acc 57%, Loss 1.984227]
Iter 2. [Val Acc 64%, Loss 0.996037]
Iter 4. [Val Acc 69%, Loss 0.691916]
Iter 6. [Val Acc 72%, Loss 0.604679]
Iter 8. [Val Acc 73%, Loss 0.577591]
Iter 10. [Val Acc 73%, Loss 0.566800]
Iter 12. [Val Acc 73%, Loss 0.561645]
Iter 14. [Val Acc 73%, Loss 0.559059]
Iter 16. [Val Acc 73%, Loss 0.557620]
Iter 18. [Val Acc 73%, Loss 0.556753]
Iter 20. [Val Acc 73%, Loss 0.556155]
Iter 22. [Val Acc 74%, Loss 0.555879]
Iter 24. [Val Acc 74%, Loss 0.555621]
Iter 26. [Val Acc 74%, Loss 0.555432]
Iter 28. [Val Acc 74%, Loss 0.555315]
Iter 30. [Val Acc 74%, Loss 0.555243]
Iter 32. [Val Acc 74%, Loss 0.555147]
Iter 34. [Val Acc 74%, Loss 0.555077]
Iter 36. [Val Acc 74%, Loss 0.555090]
Iter 38. [Val Acc 74%, Loss 0.554966]
Iter 40. [Val Acc 74%, Loss 0.555023]
Iter 42. [Val Acc 74%, Loss 0.554997]
Iter 44. [Val Acc 74%, Loss 0.554975]
Iter 46. [Val Acc 74%, Loss 0.554957]
Iter 48. [Val Acc 74%, Loss 0.555008]
minimal_cost: 0, cost_diff: 1.4291998445364875, batch_size: 100
Iter 0. [Val Acc 60%, Loss 0.722162]
Iter 2. [Val Acc 71%, Loss 0.591857]
Iter 4. [Val Acc 74%, Loss 0.563267]
Iter 6. [Val Acc 74%, Loss 0.557013]
Iter 8. [Val Acc 74%, Loss 0.555596]
Iter 10. [Val Acc 74%, Loss 0.555216]
Iter 12. [Val Acc 74%, Loss 0.555108]
Iter 14. [Val Acc 74%, Loss 0.555053]
Iter 16. [Val Acc 74%, Loss 0.555032]
Iter 18. [Val Acc 74%, Loss 0.555011]
Iter 20. [Val Acc 74%, Loss 0.554985]
Iter 22. [Val Acc 74%, Loss 0.554980]
```

```
Iter 24. [Val Acc 74%, Loss 0.554963]
Iter 26. [Val Acc 74%, Loss 0.554969]
Iter 28. [Val Acc 74%, Loss 0.554957]
Iter 30. [Val Acc 74%, Loss 0.554955]
Iter 32. [Val Acc 74%, Loss 0.554947]
Iter 34. [Val Acc 74%, Loss 0.554948]
Iter 36. [Val Acc 74%, Loss 0.554952]
Iter 38. [Val Acc 74%, Loss 0.554950]
Iter 40. [Val Acc 74%, Loss 0.554950]
Iter 42. [Val Acc 74%, Loss 0.554926]
Iter 44. [Val Acc 74%, Loss 0.554934]
Iter 46. [Val Acc 74%, Loss 0.554932]
Iter 48. [Val Acc 74%, Loss 0.554947]
Iter 0. [Val Acc 55%, Loss 0.784721]
Iter 2. [Val Acc 66%, Loss 0.652703]
Iter 4. [Val Acc 71%, Loss 0.596071]
Iter 6. [Val Acc 73%, Loss 0.572325]
Iter 8. [Val Acc 74%, Loss 0.562424]
Iter 10. [Val Acc 74%, Loss 0.558291]
Iter 12. [Val Acc 74%, Loss 0.556529]
Iter 14. [Val Acc 74%, Loss 0.555749]
Iter 16. [Val Acc 74%, Loss 0.555401]
Iter 18. [Val Acc 74%, Loss 0.555227]
Iter 20. [Val Acc 74%, Loss 0.555151]
Iter 22. [Val Acc 74%, Loss 0.555082]
Iter 24. [Val Acc 74%, Loss 0.555060]
Iter 26. [Val Acc 74%, Loss 0.555040]
Iter 28. [Val Acc 74%, Loss 0.555020]
Iter 30. [Val Acc 74%, Loss 0.555011]
Iter 32. [Val Acc 74%, Loss 0.555003]
Iter 34. [Val Acc 74%, Loss 0.554990]
Iter 36. [Val Acc 74%, Loss 0.554983]
Iter 38. [Val Acc 74%, Loss 0.554974]
Iter 40. [Val Acc 74%, Loss 0.554977]
Iter 42. [Val Acc 74%, Loss 0.554965]
Iter 44. [Val Acc 74%, Loss 0.554959]
Iter 46. [Val Acc 74%, Loss 0.554959]
Iter 48. [Val Acc 74%, Loss 0.554950]
optimal_batch_size: 100
```


Accuracy optimal_batch_size: 100
val_accs
7×10^{-1}
6×10^{-1}
0
10
20
30
40
50
Number of Batches


Costs optimal_batch_size: 100
val_costs
2×10^{0}
10^{0}
6×10^{-1}
0
10
20
30
40
50
Number of Batches

Explain and discuss your results here: We ran the training process many times and each time saw similar results. It emphisizes the training process stability. To find optimal training step mu, we ran the training process with small to large mu ranges. The optimal training process happened for mu = 2e-3. To find optimal batch size, we used fixed (optimal) mu and ran the training process for several batch sizes. The optimal batch size happened for batch_size = 100.

How we compared the training results to decide which training process is better? After several experiments and their analysis, we concluded that training cost differences between the first training cost and the last training cost provides the best training process score. cost_diff = val_costs[0] - val_costs[len(val_costs)-1]

To make sure the choosen training process really converges to the lowest cost, we added checked that the training cost is really lower than the previous training cost: minimal_cost < val_costs[len(val_costs)-1]

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?

[21]:
```python
def calc_acc(w, b, data_norm_xs, data_ts):
  """Return the value of data_acc.
  We use:
    - data_norm_xs, data_ts as the data set
    - (w, b)
  """
  # compute the given model accuracy on a dataset
  # the model is defined by its parameters: (w,b)
  X = data_norm_xs
  t = data_ts[:, 0]
  y = pred(w, b, X)
  data_acc = get_accuracy(y, t)
  return data_acc
```

[22]:
```python
# Write your code here

train_acc = calc_acc(optimal_w, optimal_b, train_norm_xs, train_ts)
val_acc = calc_acc(optimal_w, optimal_b, val_norm_xs, val_ts)
test_acc = calc_acc(optimal_w, optimal_b, test_norm_xs, test_ts)

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

```
train_acc =  0.7326057793408506  val_acc =  0.73612  test_acc =
0.7271353864032539
```

Explain and discuss your results here: The fact that the training and validation accuracy are very close numbers, means that the training model fits well the training data set and validated well with the validation set. The fact that the test accuracy is also very close to the training and validation accuracy, means that the model is not overfitten to the training dataset, and there is a good chance that it will generalize well to the whole data distribution and perform well on unknown data samples. In the next paragraph (Part (i)), we used builtin sklearn LogisticRegression implementation sklearn.linear_model.LogisticRegression to verify our training process. We see that sklearn LogisticRegression provides similar training accuracy: train_acc = 0.7329465936695552 val_acc = 0.73248 test_acc = 0.7269223319775324.

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.

[23]:
```
import sklearn.linear_model

from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=300).fit(train_norm_xs, train_ts[:, 0])
train_acc = model.score(train_norm_xs, train_ts[:, 0])
val_acc = model.score(val_norm_xs, val_ts[:, 0])
test_acc = model.score(test_norm_xs, test_ts[:, 0])

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

```
train_acc =  0.7325260142851964  val_acc =  0.73606  test_acc =
0.7270966492349409
```

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

[25]:
```
!cp './Assignment1.pdf' 'drive/My Drive/Colab Notebooks'
```

[24]:
```
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc

from google.colab import drive
drive.mount('/content/drive')
!cp 'drive/My Drive/Colab Notebooks/Assignment1.ipynb' ./

!jupyter nbconvert --to PDF "Assignment1.ipynb"
!ls -la
!cp './Assignment1.pdf' 'drive/My Drive/Colab Notebooks'
```

```
#%%capture
#!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
#from colab_pdf import colab_pdf
```

```
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).
The following package was automatically installed and is no longer required:
  libnvidia-common-460
Use 'apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 5 not upgraded.
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: pypandoc in /usr/local/lib/python3.7/dist-
packages (1.10)
Mounted at /content/drive
[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] Writing 97679 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', './notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', './notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 176092 bytes to Assignment1.pdf
ls: cannot access 'gdrive': Transport endpoint is not connected
total 352
drwxr-xr-x 1 root root   4096 Nov 27 00:31 .
drwxr-xr-x 1 root root   4096 Nov 27 00:08 ..
-rw------- 1 root root 163522 Nov 27 00:31 Assignment1.ipynb
-rw-r--r-- 1 root root 176092 Nov 27 00:31 Assignment1.pdf
drwxr-xr-x 4 root root   4096 Nov 22 00:13 .config
drwx------ 6 root root   4096 Nov 27 00:31 drive
d????????? ? ?    ?         ?            ? gdrive
drwxr-xr-x 1 root root   4096 Nov 22 00:14 sample_data
```