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 (https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd)
- http://millionsongdataset.com/pages/tasks-demos/#yearrecognition (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 (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 [5]:

```
import pandas
import numpy as np
import matplotlib.pyplot as plt
```

In [7]:

```
load_from_drive = True

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

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)
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, ca
ll drive.mount("/content/gdrive", force_remount=True).
```

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

In [8]:

```
df
```

Out[8]:

	year	var1	var2	var3	var4	var5	var6	var7	va
0	2001	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12.232
1	2001	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0.920
2	2001	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2.350
3	2001	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10.341
4	2001	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9.376
...	...	...	...	...	...	...	...	...	
515340	2006	51.28467	45.88068	22.19582	-5.53319	-3.61835	-16.36914	2.12652	5.181
515341	2006	49.87870	37.93125	18.65987	-3.63581	-27.75665	-18.52988	7.76108	3.561
515342	2006	45.12852	12.65758	-38.72018	8.80882	-29.29985	-2.28706	-18.40424	-22.287
515343	2006	44.16614	32.38368	-3.34971	-2.49165	-19.59278	-18.67098	8.78428	4.020
515344	2005	51.85726	59.11655	26.39436	-5.46030	-20.69012	-19.95528	-6.72771	2.295

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 [9]:

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

In [10]:

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

Out[10]:

	year	var1	var2	var3	var4	var5	var6	var7	var8
0	1	49.94357	21.47114	73.07750	8.74861	-17.40628	-13.09905	-25.01202	-12.23257
1	1	48.73215	18.42930	70.32679	12.94636	-10.32437	-24.83777	8.76630	-0.92019
2	1	50.95714	31.85602	55.81851	13.41693	-6.57898	-18.54940	-3.27872	-2.35035
3	1	48.24750	-1.89837	36.29772	2.58776	0.97170	-26.21683	5.05097	-10.34124
4	1	50.97020	42.20998	67.09964	8.46791	-15.85279	-16.81409	-12.48207	-9.37636
5	1	50.54767	0.31568	92.35066	22.38696	-25.51870	-19.04928	20.67345	-5.19943
6	1	50.57546	33.17843	50.53517	11.55217	-27.24764	-8.78206	-12.04282	-9.53930
7	1	48.26892	8.97526	75.23158	24.04945	-16.02105	-14.09491	8.11871	-1.87566
8	1	49.75468	33.99581	56.73846	2.89581	-2.92429	-26.44413	1.71392	-0.55644
9	1	45.17809	46.34234	-40.65357	-2.47909	1.21253	-0.65302	-6.95536	-12.20040
10	1	39.13076	-23.01763	-36.20583	1.67519	-4.27101	13.01158	8.05718	-8.41088
11	1	37.66498	-34.05910	-17.36060	-26.77781	-39.95119	-20.75000	-0.10231	-0.89972
12	1	26.51957	-148.15762	-13.30095	-7.25851	17.22029	-21.99439	5.51947	3.48418
13	1	37.68491	-26.84185	-27.10566	-14.95883	-5.87200	-21.68979	4.87374	-18.01800
14	0	39.11695	-8.29767	-51.37966	-4.42668	-30.06506	-11.95916	-0.85322	-8.86179
15	1	35.05129	-67.97714	-14.20239	-6.68696	-0.61230	-18.70341	-1.31928	-9.46370
16	1	33.63129	-96.14912	-89.38216	-12.11699	13.77252	-6.69377	-33.36843	-24.81437
17	0	41.38639	-20.78665	51.80155	17.21415	-36.44189	-11.53169	11.75252	-7.62428
18	0	37.45034	11.42615	56.28982	19.58426	-16.43530	2.22457	1.02668	-7.34736
19	0	39.71092	-4.92800	12.88590	-11.87773	2.48031	-16.11028	-16.40421	-8.29657

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 [11]:

```
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 because the tested data should represent the real unknown world, the main idea behind this assigment is to find a correlation between the songs and the century, therefore we would expect that the correlation will be in the melody or bit rate and so on, but when we provide two songs of the same artist which some of them is placed in the train and some in the test set, the network can make predictions based on features that are related to the artist and it will not serve the porpuse of the assignemnt and will provide the wrong classification. In general speaking the tested data should include minimal correlation with the training data for getting a reliability that reflects the reality.

Part (b) -- 7%

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

In [12]:

```
feature_means = df_train.mean()[1:].to_numpy() # the [1:] removes the mean of the "year" fi
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 [13]:

```
# Write your explanation here
```

We will mention 2 reasons why it would be improper to compute and use test set means and standard deviations.

1. In most cases we prefer to give a larger percentage of our data for the training set because we want to learn from a wide range of examples (a common method of division is 80% for the training set, 10% for the

validation set and 10% for the test set), assuming that the data is i.i.d, it will be more accuarte to calculate these parameters from the training set and not from the test set which is much smaller (in accordance with the law of large numbers).

2. Consistency: We have made manipulations on our training set, it means that we modified the input set to our model (preprocessing), and the results are depend on those manipulations, therefore if we use an input X, the first stage,before we insert the inputs to the model, is modifying it to be f(X) (for example subtract mean and divide by std) and the expected result is Y, if we would use the same X in the test-set after the manipulation (mean and std that were calculated according to the test set and not from the training set) it will become g(X) because it uses mean and std of the test-set, the output will be calculated according to g(X), which is not the expected behaviour.

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 [14]:

```
# 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 the number of times because if we use the test data to check the performance of the model and to choose the best hyper- parameters accordingly, again, we are impacting the reliability of the test data as a representation of unseen data. There is a chance that we will choose the hyperparameters to overfit on the test data, but when we test new unseen data, the performance will decrease, and we will damage the generalization of the model.

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 [15]:

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

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

def cost(y, t):
  return np.mean(cross_entropy(t, y))

def get_accuracy(y, t):
  acc = 0
  N = 0
  for i in range(len(y)):
    N += 1
    if (y[i] >= 0.5 and t[i] == 1) or (y[i] < 0.5 and t[i] == 0):
      acc += 1
  return acc / N
```

Part (a) -- 7%

Write a function `pred` that computes the prediction `y` based on logistic regression, i.e., a single layer with weights `w` and bias `b`. The output is given by:

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

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

In [16]:

```
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
  return sigmoid(np.dot(X, w) + b)
```

In [17]:

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

Out[17]:

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

Part (b) -- 7%

Write a function `derivative_cost` that computes and returns the gradients $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$. Here, `X` is the input, `y` is the prediction, and `t` is the true label.

In [18]:

```
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)
  dl_db = (1/N)*np.sum(y - t)
  dl_dw = (1/N)*np.dot(X.T, y - t)
  return (dl_dw, dl_db)
```

Explenation on Gradients

Add here an explaination on how the gradients are computed:

The loss measure that we are using is cross entropy , if we have $\{x_i\}_{i=1}^N$ that represent the samples (each x_i represent vector of size 90). The prediction of the model is $\{y_i\}_{i=1}^N$ accordingly to each $\{x_i\}_{i=1}^N$. The true labels is $\{t_i\}_{i=1}^N$. We know that the empirical risk is:

$$EmpiricalRisk = \frac{1}{N}\sum_{i=1}^{N} -(t_i log(y_i) + (1-t_i) log(1-y_i))$$

$$\frac{\partial \text{empiricalrisk}}{\partial b} = \frac{1}{N}\frac{\partial \sum_{i=1}^{N} -(t_i \log(y_i) + (1-t_i)\log(1-y_i))}{\partial b}$$

$$= \frac{1}{N}\sum_{i=1}^{N} -(t_i \frac{\partial \log(y_i)}{\partial b} + (1-t_i)\frac{\partial \log(1-y_i)}{\partial b})$$

Now we will simplify the two expressions $\frac{\partial \log(y_i)}{\partial b}$, $\frac{\partial \log(1-y_i)}{\partial b}$ in order to compute the gradients.

$$\frac{\partial \log(y_i)}{\partial b} = \frac{1}{y_i}\frac{\partial (y_i)}{\partial b} = \frac{1}{y_i}\frac{\partial (\sigma(z_i))}{\partial b} = \frac{1}{y_i}\sigma'(z_i)\frac{\partial (z_i)}{\partial b}$$

$$= \frac{1}{y_i}\sigma(z_i)(1-\sigma(z_i))\frac{\partial (z_i)}{\partial b}$$

$$= \frac{1}{y_i}\sigma\left(z_i\right)\left(1-\sigma\left(z_i\right)\right)$$

$$= \frac{1}{y_i}y_i\left(1-y_i\right)$$

$$= \left(1-y_i\right)$$

$$\frac{\partial \log(1-\mathrm{y_i})}{\partial b} = \frac{1}{1-y_i}\frac{\partial\left(1-y_i\right)}{\partial b} = \frac{-1}{1-y_i}\frac{\partial\left(\sigma\left(z_i\right)\right)}{\partial b} = \frac{-1}{1-y_i}\sigma'\left(z_i\right)\frac{\partial\left(z_i\right)}{\partial b}$$

$$= \frac{-1}{1-y_i}\sigma\left(z_i\right)\left(1-\sigma\left(z_i\right)\right)\frac{\partial\left(z_i\right)}{\partial b}$$

$$= \frac{-1}{1-y_i}\sigma\left(z_i\right)\left(1-\sigma\left(z_i\right)\right)$$

$$= \frac{-1}{1-y_i}y_i\left(1-y_i\right)$$

$$= \left(-y_i\right)$$

After placing the two expressions above in the expression we will get:

$$= \frac{1}{N}\sum_{i=1}^{N} -(t_i\frac{\partial \log(\mathrm{y_i})}{\partial b} \ + \ (1-t_i)\frac{\partial \log(1-\mathrm{y_i})}{\partial b})$$

$$= \frac{1}{N}\sum_{i=1}^{N} -(t_i(1-y_i) \ + \ (1-t_i)(-y_i)) =$$

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - t_i) = np.mean(y-t)$$

##

$$\frac{\partial \mathrm{empiricalrisk}}{\partial w_j} = \frac{1}{N}\frac{\partial \sum_{\mathrm{i}=1}^{\mathrm{N}} -(\mathrm{t_i}\log\left(\mathrm{y_i}\right) \ + \ (1-\mathrm{t_i})\log\left(1-\mathrm{y_i}\right))}{\partial w_j}$$

$$= \frac{1}{N}\sum_{i=1}^{N} -(t_i\frac{\partial \log(\mathrm{y_i})}{\partial w_j} \ + \ (1-t_i)\frac{\partial \log(1-\mathrm{y_i})}{\partial w_j})$$

Now we will simplify the two expressions $\frac{\partial \log(\mathrm{y_i})}{\partial w_j}$, $\frac{\partial \log(1-\mathrm{y_i})}{\partial w_j}$ in order to compute the gradients.

$$\frac{\partial \log(\mathrm{y_i})}{\partial w_j} = \frac{1}{y_i}\frac{\partial\left(y_i\right)}{\partial w_j} = \frac{1}{y_i}\frac{\partial\left(\sigma\left(z_i\right)\right)}{\partial w_j} = \frac{1}{y_i}\sigma'\left(z_i\right)\frac{\partial\left(z_i\right)}{\partial w_j}$$

$$= \frac{1}{y_i}\sigma\left(z_i\right)\left(1-\sigma\left(z_i\right)\right)\frac{\partial\left(z_i\right)}{\partial w_j}$$

$$= \frac{1}{y_i}\sigma\left(z_i\right)\left(1-\sigma\left(z_i\right)\right)x_i[j]$$

$$= \frac{1}{y_i}y_i\left(1-y_i\right)x_i[j]$$

$$= \left(1-y_i\right)x_i[j]$$

$$\frac{\partial \log(1-\mathrm{y_i})}{\partial w_j} = \frac{1}{1-y_i}\frac{\partial\left(1-y_i\right)}{\partial w_j} = \frac{-1}{1-y_i}\frac{\partial\left(\sigma\left(z_i\right)\right)}{\partial w_j} = \frac{-1}{1-y_i}\sigma'\left(z_i\right)\frac{\partial\left(z_i\right)}{\partial w_j}$$

$$= \frac{-1}{1-y_i}\sigma\left(z_i\right)\left(1-\sigma\left(z_i\right)\right)x_i[j]$$

$$= \frac{-1}{1-y_i}y_i\left(1-y_i\right)x_i[j]$$

$$= \left(-y_i\right)x_i[j]$$

After placing the two expressions above in the expression we will get:

$$= \frac{1}{N}\sum_{i=1}^{N} -(t_i\frac{\partial \log(\mathrm{y_i})}{\partial w_j} \; + \; (1-t_i)\frac{\partial \log(1-\mathrm{y_i})}{\partial w_j})$$

$$= \frac{1}{N}\sum_{i=1}^{N} -(t_i(1-y_i)x_i[j] \; + \; (1-t_i)(-y_i)x_i[j]) =$$

$$\frac{1}{N}\sum_{i=1}^{N}(y_i-t_i)x_i[j] = \frac{1}{N}x^T(y-t)$$

Part (c) -- 7%

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

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

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

In [19]:

```
# define random inputs
epsilon = 1e-4
b = np.random.randn(1)
w = np.random.randn(90)
X = np.random.rand(200, 90)
t = np.ones(200)

b_plus_epsilon = b + epsilon
y_b = pred(w, b, X)
y_b_plus_epsilon = pred(w, b_plus_epsilon, X)

r1 = (cost(y_b_plus_epsilon, t)-cost(y_b, t))/epsilon
_, r2 = derivative_cost(X, y_b, t)
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
```

```
The analytical results is - -0.568507124973916
The algorithm results is -  -0.5685124093296305
```

Part (d) -- 7%

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

In [20]:

```
# We will generate a random example, compute the analytical and algorithm partial
# derivatives, and compare the results.
# We have the vector w that we want to calculate for each element its derivative.
# In order to do this we took each time the vector w and added to it in the
# appropriate position epsilon
# (appropriate position is the index that we want to calc derivative), while to
# the other elements we did not add anything.
# The epsilon we add every time does not have to be equal to each element,
# so we made it random.

epsilon = np.random.rand(90)*0.001
b = np.random.rand(1)
w = np.random.rand(90)
X = np.random.rand(200, 90)
t = np.ones(200)

r1 = []
for i in range(90):
  w_new = w.copy()
  w_new[i] += epsilon[i]
  y_w = pred(w, b, X)
  y_wi_plus_epsilon = pred(w_new, b, X)
  derivative_wi = (cost(y_wi_plus_epsilon, t)-cost(y_w, t))/epsilon[i]
  r1.append(derivative_wi)

r1 = np.asarray(r1)
r2, _ = derivative_cost(X, y_w, t)
print("The analytical results is -", r1)
print("The algorithm results is - ", r2)
```

```
The analytical results is - [-3.41682249e-10 -5.00962542e-10 -2.86807301e-10
-3.86472664e-10
 -3.36911005e-10 -4.32575498e-10 -4.53131143e-10 -4.10376395e-10
 -3.29902417e-10 -3.80645112e-10 -4.19425804e-10 -4.65811725e-10
 -4.18579229e-10 -4.00894648e-10 -4.45068378e-10 -4.14243948e-10
 -4.15388451e-10 -4.45074250e-10 -3.32826659e-10 -3.48885064e-10
 -4.65523579e-10 -4.81919065e-10 -3.31570932e-10 -4.73655900e-10
 -3.66086416e-10 -3.66494197e-10 -4.31241797e-10 -3.39228841e-10
 -3.63857134e-10 -4.08922138e-10 -3.84000048e-10 -4.32407896e-10
 -4.45034196e-10 -3.69349494e-10 -4.07129808e-10 -3.85655098e-10
 -4.09908519e-10 -3.80492880e-10 -3.51305450e-10 -3.38884445e-10
 -4.58789758e-10 -3.82067960e-10 -4.83353522e-10 -4.79191885e-10
 -4.55699087e-10 -4.01595330e-10 -4.05799392e-10 -4.28864951e-10
 -3.57657596e-10 -4.23739054e-10 -3.92321129e-10 -4.44364339e-10
 -3.54576983e-10 -4.10133695e-10 -4.00613755e-10 -3.35460843e-10
 -4.35262477e-10 -3.77998443e-10 -3.99081500e-10 -3.20432489e-10
 -4.27938023e-10 -3.39582220e-10 -3.14728225e-10 -4.57266111e-10
 -4.71391181e-10 -3.89565764e-10 -3.06035234e-10 -4.68184837e-10
 -3.72575886e-10 -4.18797334e-10 -4.20831887e-10 -3.80570748e-10
 -4.28080172e-10 -3.59459484e-10 -3.91046384e-10 -3.52009712e-10
 -3.76840039e-10 -3.36337739e-10 -4.32628692e-10 -4.35480308e-10
 -2.80709438e-10 -4.30922159e-10 -4.13980148e-10 -3.20181603e-10
 -3.46367061e-10 -2.70638034e-10 -3.50082703e-10 -3.88701378e-10
 -3.62840118e-10 -4.14749129e-10]
The algorithm results is -  [-3.41701983e-10 -5.01083317e-10 -2.86865033e-10
-3.86536460e-10
 -3.36964679e-10 -4.32529639e-10 -4.53100181e-10 -4.10486751e-10
 -3.29940978e-10 -3.80755357e-10 -4.19494184e-10 -4.65887850e-10
 -4.18640405e-10 -4.00958341e-10 -4.45116137e-10 -4.14236542e-10
```

```
-4.15460318e-10 -4.45182768e-10 -3.32862491e-10 -3.48933155e-10
-4.65562624e-10 -4.82069422e-10 -3.31589155e-10 -4.73698547e-10
-3.65841830e-10 -3.66548018e-10 -4.31369608e-10 -3.39025031e-10
-3.63910793e-10 -4.09027053e-10 -3.84094106e-10 -4.32436002e-10
-4.45097473e-10 -3.69271872e-10 -4.07203107e-10 -3.85676722e-10
-4.09896147e-10 -3.80509641e-10 -3.51361465e-10 -3.38892119e-10
-4.58890671e-10 -3.82074954e-10 -4.83445085e-10 -4.79312240e-10
-4.55850771e-10 -4.01501528e-10 -4.05793753e-10 -4.28962464e-10
-3.57678074e-10 -4.23846643e-10 -3.92357045e-10 -4.44481793e-10
-3.54573463e-10 -4.10171643e-10 -4.00640594e-10 -3.35543382e-10
-4.35260689e-10 -3.78010339e-10 -3.99199101e-10 -3.20412680e-10
-4.27284518e-10 -3.39615814e-10 -3.14658540e-10 -4.57340615e-10
-4.71456772e-10 -3.89656417e-10 -3.06079033e-10 -4.68246086e-10
-3.72581276e-10 -4.18767263e-10 -4.20864443e-10 -3.80663975e-10
-4.28183506e-10 -3.59533211e-10 -3.91027402e-10 -3.52006248e-10
-3.76949441e-10 -3.36428780e-10 -4.32761376e-10 -4.35514395e-10
-2.80749154e-10 -4.31033367e-10 -4.14037144e-10 -3.20221790e-10
-3.46449842e-10 -2.70694551e-10 -3.50087358e-10 -3.88754963e-10
-3.62870351e-10 -4.14846239e-10]
```

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 [64]:

```
def run_gradient_descent(w0, b0, train_norm_xs, train_ts, val_norm_xs,
                         val_norm_ts, 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
  val_acc_list = []
  val_cost_list = []

  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
      y_pred = pred(w, b, X)
      dl_dw, dl_db = derivative_cost(X, y_pred, t)
      # update w and b
      w = w - mu*dl_dw
      b = b - mu*dl_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[:, 0])
        val_acc = get_accuracy(y_val, val_ts[:, 0])
        val_cost_list.append(val_cost)
        val_acc_list.append(val_acc)
        print("Iter %d. [Val Acc %.0f%%, Loss %f]" % (iter, val_acc * 100,
                                                      val_cost))

      if iter >= max_iters:
        break

      # Think what parameters you should return for further use
```

```
    return w, b, val_cost_list, val_acc_list
```

Now we are checking our validation accuracy with μ = 0.1 and batch_size = 100, Note that the empirical risk on the validation set decreased as function of the epochs(desirable behavior).

In [56]:

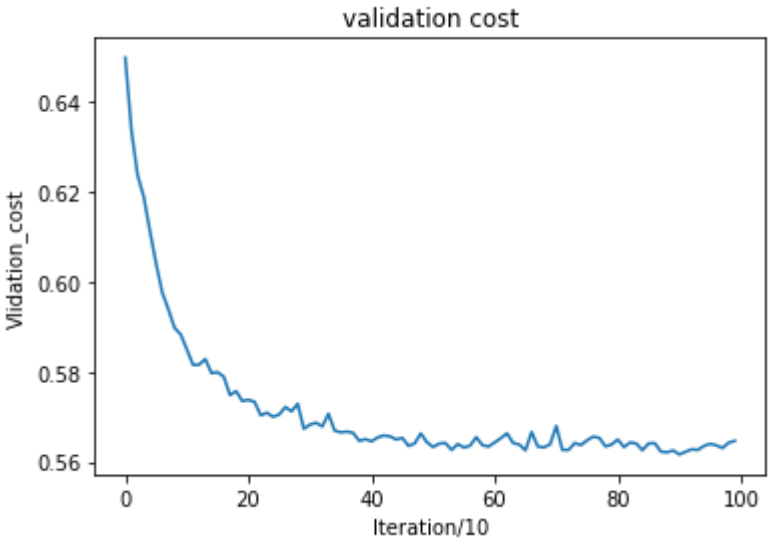
```
mu = 0.1
batch_size = 100
w0 = np.zeros(90)
b0 = 0
w_optimal, b_optimal, val_cost_list, val_acc_list = run_gradient_descent(w0, b0,
train_norm_xs, train_ts, val_norm_xs, val_ts,mu=mu, batch_size=batch_size,
max_iters=1000)
```

```
Iter 10. [Val Acc 66%, Loss 0.649679]
Iter 20. [Val Acc 67%, Loss 0.633457]
Iter 30. [Val Acc 68%, Loss 0.623587]
Iter 40. [Val Acc 68%, Loss 0.618795]
Iter 50. [Val Acc 70%, Loss 0.611197]
Iter 60. [Val Acc 70%, Loss 0.604003]
Iter 70. [Val Acc 71%, Loss 0.597574]
Iter 80. [Val Acc 71%, Loss 0.593837]
Iter 90. [Val Acc 71%, Loss 0.589716]
Iter 100. [Val Acc 71%, Loss 0.588285]
Iter 110. [Val Acc 72%, Loss 0.584954]
Iter 120. [Val Acc 72%, Loss 0.581539]
Iter 130. [Val Acc 72%, Loss 0.581562]
Iter 140. [Val Acc 71%, Loss 0.582827]
Iter 150. [Val Acc 72%, Loss 0.579701]
Iter 160. [Val Acc 72%, Loss 0.579879]
Iter 170. [Val Acc 72%, Loss 0.578884]
Iter 180. [Val Acc 72%, Loss 0.574830]
Iter 190. [Val Acc 72%, Loss 0.575699]
Iter 200. [Val Acc 72%, Loss 0.573502]
Iter 210. [Val Acc 72%, Loss 0.573768]
Iter 220. [Val Acc 72%, Loss 0.573312]
Iter 230. [Val Acc 72%, Loss 0.570344]
Iter 240. [Val Acc 72%, Loss 0.570893]
Iter 250. [Val Acc 72%, Loss 0.570001]
Iter 260. [Val Acc 73%, Loss 0.570508]
Iter 270. [Val Acc 72%, Loss 0.572176]
Iter 280. [Val Acc 72%, Loss 0.571207]
Iter 290. [Val Acc 72%, Loss 0.572892]
Iter 300. [Val Acc 73%, Loss 0.567358]
Iter 310. [Val Acc 72%, Loss 0.568267]
Iter 320. [Val Acc 72%, Loss 0.568670]
Iter 330. [Val Acc 72%, Loss 0.567865]
Iter 340. [Val Acc 73%, Loss 0.570673]
Iter 350. [Val Acc 72%, Loss 0.566928]
Iter 360. [Val Acc 73%, Loss 0.566557]
Iter 370. [Val Acc 73%, Loss 0.566714]
Iter 380. [Val Acc 73%, Loss 0.566431]
Iter 390. [Val Acc 73%, Loss 0.564669]
Iter 400. [Val Acc 72%, Loss 0.565022]
Iter 410. [Val Acc 73%, Loss 0.564558]
Iter 420. [Val Acc 73%, Loss 0.565418]
Iter 430. [Val Acc 73%, Loss 0.565853]
Iter 440. [Val Acc 73%, Loss 0.565648]
Iter 450. [Val Acc 73%, Loss 0.564959]
Iter 460. [Val Acc 73%, Loss 0.565310]
Iter 470. [Val Acc 73%, Loss 0.563541]
Iter 480. [Val Acc 73%, Loss 0.564089]
Iter 490. [Val Acc 73%, Loss 0.566305]
Iter 500. [Val Acc 73%, Loss 0.564344]
```

```
Iter 510. [Val Acc 73%, Loss 0.563306]
Iter 520. [Val Acc 72%, Loss 0.564058]
Iter 530. [Val Acc 72%, Loss 0.564175]
Iter 540. [Val Acc 73%, Loss 0.562659]
Iter 550. [Val Acc 73%, Loss 0.563947]
Iter 560. [Val Acc 73%, Loss 0.563178]
Iter 570. [Val Acc 73%, Loss 0.563596]
Iter 580. [Val Acc 72%, Loss 0.565479]
Iter 590. [Val Acc 73%, Loss 0.563716]
Iter 600. [Val Acc 73%, Loss 0.563398]
Iter 610. [Val Acc 73%, Loss 0.564304]
Iter 620. [Val Acc 73%, Loss 0.565264]
Iter 630. [Val Acc 72%, Loss 0.566317]
Iter 640. [Val Acc 73%, Loss 0.564187]
Iter 650. [Val Acc 73%, Loss 0.563842]
Iter 660. [Val Acc 73%, Loss 0.562550]
Iter 670. [Val Acc 72%, Loss 0.566623]
Iter 680. [Val Acc 73%, Loss 0.563392]
Iter 690. [Val Acc 73%, Loss 0.563234]
Iter 700. [Val Acc 73%, Loss 0.563876]
Iter 710. [Val Acc 72%, Loss 0.567955]
Iter 720. [Val Acc 73%, Loss 0.562655]
Iter 730. [Val Acc 73%, Loss 0.562648]
Iter 740. [Val Acc 73%, Loss 0.564174]
Iter 750. [Val Acc 73%, Loss 0.563738]
Iter 760. [Val Acc 73%, Loss 0.564771]
Iter 770. [Val Acc 73%, Loss 0.565597]
Iter 780. [Val Acc 73%, Loss 0.565321]
Iter 790. [Val Acc 73%, Loss 0.563432]
Iter 800. [Val Acc 72%, Loss 0.563858]
Iter 810. [Val Acc 72%, Loss 0.564961]
Iter 820. [Val Acc 73%, Loss 0.563295]
Iter 830. [Val Acc 73%, Loss 0.564300]
Iter 840. [Val Acc 73%, Loss 0.564065]
Iter 850. [Val Acc 73%, Loss 0.562580]
Iter 860. [Val Acc 73%, Loss 0.564087]
Iter 870. [Val Acc 73%, Loss 0.564136]
Iter 880. [Val Acc 73%, Loss 0.562265]
Iter 890. [Val Acc 73%, Loss 0.562126]
Iter 900. [Val Acc 73%, Loss 0.562518]
Iter 910. [Val Acc 73%, Loss 0.561642]
Iter 920. [Val Acc 73%, Loss 0.562240]
Iter 930. [Val Acc 73%, Loss 0.562781]
Iter 940. [Val Acc 73%, Loss 0.562661]
Iter 950. [Val Acc 73%, Loss 0.563529]
Iter 960. [Val Acc 73%, Loss 0.564009]
Iter 970. [Val Acc 73%, Loss 0.563667]
Iter 980. [Val Acc 73%, Loss 0.563057]
Iter 990. [Val Acc 73%, Loss 0.564255]
Iter 1000. [Val Acc 73%, Loss 0.564669]
```

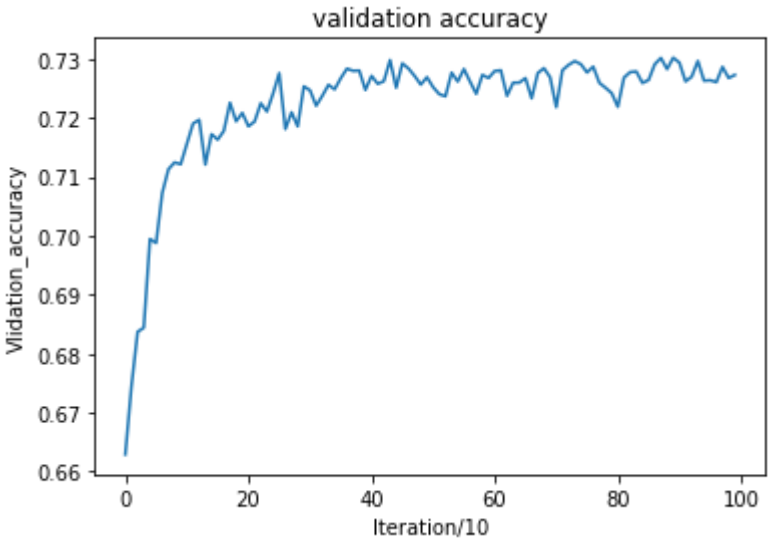
In [60]:

```
plt.plot(val_cost_list)
plt.title("validation cost")
plt.xlabel('Iteration/10')
plt.ylabel('Vlidation_cost')
plt.show()
```



In [61]:

```
plt.plot(val_acc_list)
plt.title("validation accuracy")
plt.xlabel('Iteration/10')
plt.ylabel('Vlidation_accuracy')
plt.show()
```



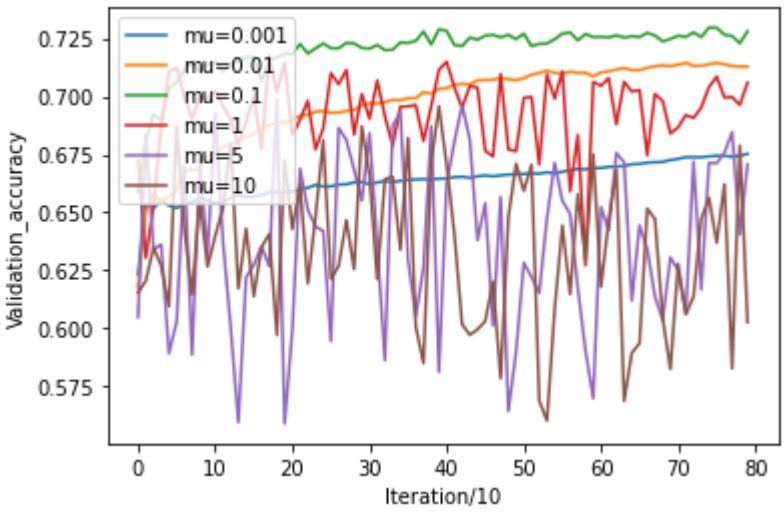
Part (f) -- 7%

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

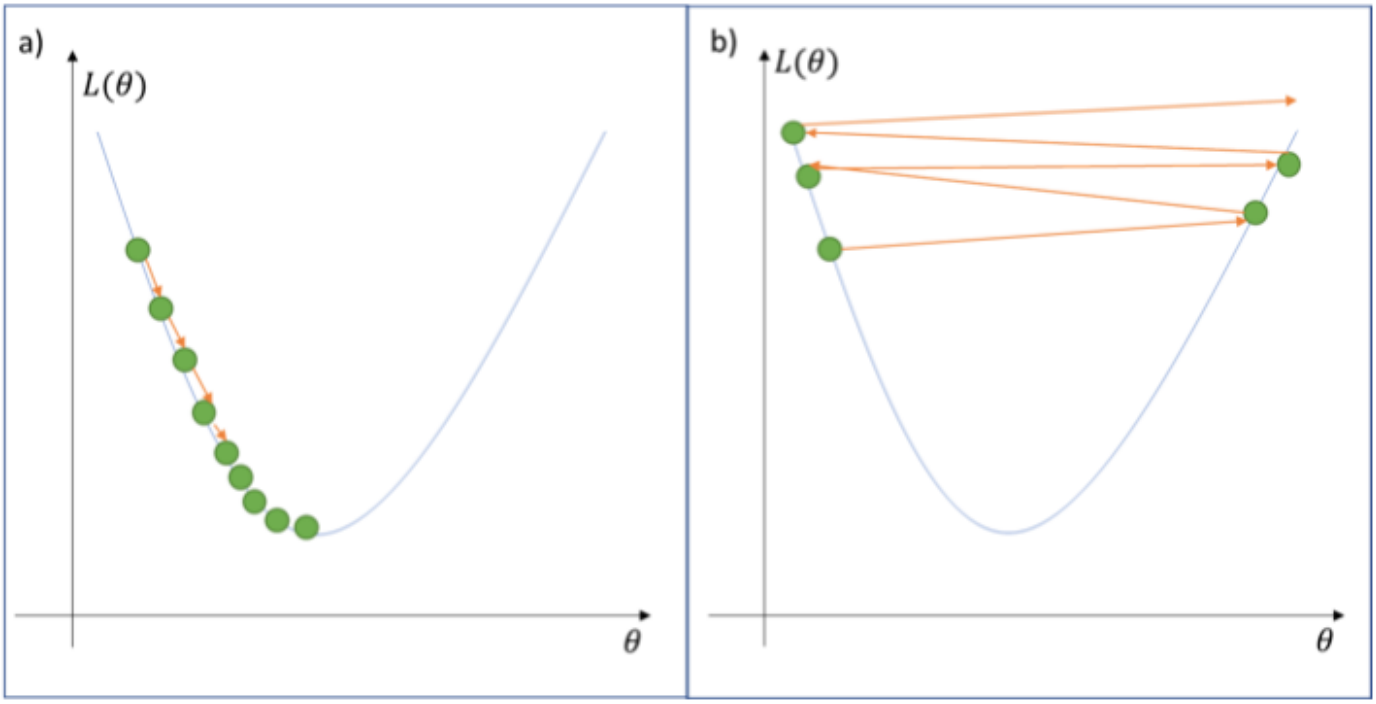
In [25]:

```
w0 = np.zeros(90)
b0 = 0

mu_list = [0.001, 0.01, 0.1, 1, 5, 10]
for mu in mu_list:
  w_optimal, b_optimal, val_cost_list, val_acc_list = run_gradient_descent(w0,
  b0, train_norm_xs, train_ts, val_norm_xs, val_ts, mu=mu, batch_size=100,
   max_iters=800)
  plt.plot(val_acc_list, label = 'mu=' + str(mu))
plt.xlabel('Iteration/10')
plt.ylabel('Validation_accuracy')
plt.legend(loc="upper left")
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: RuntimeWarni
ng: divide by zero encountered in log
  """
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: RuntimeWarni
ng: invalid value encountered in multiply
  """
```



We can describe the cost function as a function of the weights, and as well as the accuracy function. We will receive the best results when the weights converge to the global minimum of the cost function (In most cases the curve that describes the cost function won't be a convex function so it includes local minimas and plateaus) using the SGD algorithm, we are changing the weights at each iteration in the negative direction of the slope towards the local minimum. Hopefully, it will converge to the global minimum after enough epochs. When using a very big learning rate mu, the weights will "jump" over the value that gives the minimum; this phenomenon is called overshooting. The problem of using very small weights is that it will take much time for the algorithm to converge and, the weights can get stuck in a local minimum, as we learned in class. In the figure above, we see exactly those phenomenons on the convergence of the accuracy, and for a very small learning rate the accuracy improves very slow, and for a very high learning rate, the accuracy is not converging at all.



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 [50]:

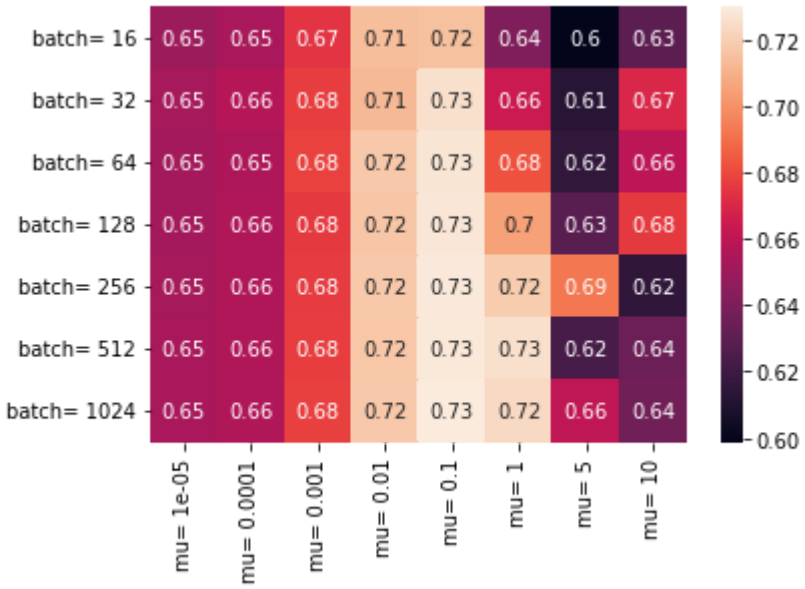
```
val_acc_max = 0
w_optimal = np.zeros(90)
b_optimal = 0

batch_size_list = [16, 32, 64, 128, 256, 512, 1024]
mu_list = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 5, 10]
Accuracy_matrix = np.zeros((len(batch_size_list), len(mu_list)))

for i in range(len(batch_size_list)):
  for j in range(len(mu_list)):
    # We initialize the weights of the network to be 0 in order to start from
    # the same reference point for each batch size and mu
    # (of course it is possible that if we initialize the network weights
    # differently we will reach different performance,
    # of course we do not want our optimization to be affected from our starting
    # point) for every batch size and learning rate we

    w0 = np.zeros(90)
    b0 = 0
    print(f'Starting train with hyper-parameters: mu={mu_list[j]},
     batch_size={batch_size_list[i]}')
    w0, b0, val_cost_list, val_acc_list = run_gradient_descent(w0, b0,
    train_norm_xs, train_ts, val_norm_xs, val_ts, mu=mu_list[j],
     batch_size=batch_size_list[i], max_iters=1000)
    if val_acc_list[-1] > val_acc_max:
      val_acc_max = val_acc_list[-1]
      w_optimal = w0
      b_optimal = b0
      best_batch_size = batch_size_list[i]
      best_mu = mu_list[j]
    Accuracy_matrix[i, j] = val_acc_list[-1]

print(f'The best accuracy is: {val_acc_max}, mu={best_mu},
 batch size = {best_batch_size} ')
```

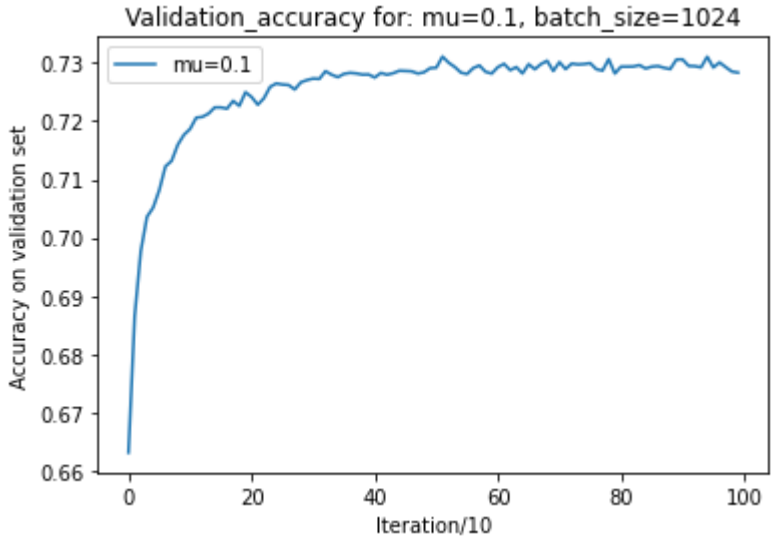
```
Starting train with hyper-parameters: mu=1e-05, batch_size=16
Starting train with hyper-parameters: mu=0.0001, batch_size=16
Starting train with hyper-parameters: mu=0.001, batch_size=16
Starting train with hyper-parameters: mu=0.01, batch_size=16
Starting train with hyper-parameters: mu=0.1, batch_size=16
Starting train with hyper-parameters: mu=1, batch_size=16

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: RuntimeWarni
ng: divide by zero encountered in log
  """
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: RuntimeWarni
ng: invalid value encountered in multiply
  """

Starting train with hyper-parameters: mu=5, batch_size=16

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


Starting train with hyper-parameters: mu=10, batch_size=16
Starting train with hyper-parameters: mu=1e-05, batch_size=32
```

```
Starting train with hyper-parameters: mu=0.0001, batch_size=32
Starting train with hyper-parameters: mu=0.001, batch_size=32
Starting train with hyper-parameters: mu=0.01, batch_size=32
Starting train with hyper-parameters: mu=0.1, batch_size=32
Starting train with hyper-parameters: mu=1, batch_size=32
Starting train with hyper-parameters: mu=5, batch_size=32
Starting train with hyper-parameters: mu=10, batch_size=32
Starting train with hyper-parameters: mu=1e-05, batch_size=64
Starting train with hyper-parameters: mu=0.0001, batch_size=64
Starting train with hyper-parameters: mu=0.001, batch_size=64
Starting train with hyper-parameters: mu=0.01, batch_size=64
Starting train with hyper-parameters: mu=0.1, batch_size=64
Starting train with hyper-parameters: mu=1, batch_size=64
Starting train with hyper-parameters: mu=5, batch_size=64
Starting train with hyper-parameters: mu=10, batch_size=64
Starting train with hyper-parameters: mu=1e-05, batch_size=128
Starting train with hyper-parameters: mu=0.0001, batch_size=128
Starting train with hyper-parameters: mu=0.001, batch_size=128
Starting train with hyper-parameters: mu=0.01, batch_size=128
Starting train with hyper-parameters: mu=0.1, batch_size=128
Starting train with hyper-parameters: mu=1, batch_size=128
Starting train with hyper-parameters: mu=5, batch_size=128
Starting train with hyper-parameters: mu=10, batch_size=128
Starting train with hyper-parameters: mu=1e-05, batch_size=256
Starting train with hyper-parameters: mu=0.0001, batch_size=256
Starting train with hyper-parameters: mu=0.001, batch_size=256
Starting train with hyper-parameters: mu=0.01, batch_size=256
Starting train with hyper-parameters: mu=0.1, batch_size=256
Starting train with hyper-parameters: mu=1, batch_size=256
Starting train with hyper-parameters: mu=5, batch_size=256
Starting train with hyper-parameters: mu=10, batch_size=256
Starting train with hyper-parameters: mu=1e-05, batch_size=512
Starting train with hyper-parameters: mu=0.0001, batch_size=512
Starting train with hyper-parameters: mu=0.001, batch_size=512
Starting train with hyper-parameters: mu=0.01, batch_size=512
Starting train with hyper-parameters: mu=0.1, batch_size=512
Starting train with hyper-parameters: mu=1, batch_size=512
Starting train with hyper-parameters: mu=5, batch_size=512
Starting train with hyper-parameters: mu=10, batch_size=512
Starting train with hyper-parameters: mu=1e-05, batch_size=1024
Starting train with hyper-parameters: mu=0.0001, batch_size=1024
Starting train with hyper-parameters: mu=0.001, batch_size=1024
Starting train with hyper-parameters: mu=0.01, batch_size=1024
Starting train with hyper-parameters: mu=0.1, batch_size=1024
Starting train with hyper-parameters: mu=1, batch_size=1024
Starting train with hyper-parameters: mu=5, batch_size=1024
Starting train with hyper-parameters: mu=10, batch_size=1024
The best accuracy is: 0.72992, mu=0.1, batch size = 1024
```

In [65]:

```
import seaborn as sns
xticklabels = []
yticklabels = []
# define our columns names of the dataframe
for i in range(len(mu_list)):
  xticklabels.append('mu= '+ str(mu_list[i]))
for i in range(len(batch_size_list)):
  yticklabels.append('batch= ' + str(batch_size_list[i]))
ax = sns.heatmap(Accuracy_matrix, xticklabels=xticklabels,
                 yticklabels=yticklabels, annot=True)
plt.show()

# plot showing the percentage of accuracy on a validation set versus the number
# of epoches for the hyper parameters that led to the highest percentages
# of accuracy
w0 = np.zeros(90)
b0 = 0
w_optimal, b_optimal, val_cost_list, val_acc_list = run_gradient_descent(w0, b0,
train_norm_xs, train_ts, val_norm_xs, val_ts, mu=best_mu,
 batch_size=best_batch_size, max_iters=1000)
plt.plot(val_acc_list, label = 'mu=' + str(best_mu))
plt.xlabel('Iteration/10')
plt.ylabel('Accuracy on validation set')
plt.title(f'Validation_accuracy for: mu={best_mu},
 batch_size={best_batch_size}')
plt.legend(loc="upper left")
plt.show()
```





Explain and discuss your results here: Since we are searching for the optimal hyper-parameters of only two parameters (learnig rate and batch size), we performed a grid search over different learning rates and batch sizes. The heatmap above shows the accuracy for the differet parmeters we took in the search. Overall we can see the taking large batch size (1024) was benficial. In addition, we can see that 1000 iterations are not engough for small learning rates (or that the cost function is stuck in local minmum during the gradient descent algorithm). we can see the training process of the worst hyperparameters and the best hyperparameters, and therfore demostrating the good and bad behaviours of them on the training process. For the worst hyperparameters we can see that no learing is done at all.

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 [52]:

```
# Write your code here

y_pred_train = pred(w_optimal, b_optimal, train_norm_xs)
y_pred_val = pred(w_optimal, b_optimal, val_norm_xs)
y_pred_test = pred(w_optimal, b_optimal, test_norm_xs)

train_acc = get_accuracy(y_pred_train, train_ts)
val_acc = get_accuracy(y_pred_val, val_ts)
test_acc = get_accuracy(y_pred_test, test_ts)

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

```
train_acc =  0.7328668286139008  val_acc =  0.73016  test_acc =  0.724927367
8094131
```

First of all, the main goal of the validation set is to prevent overfitting. We can see from this case that there is a tiny difference between the accuracies. As we see the best accuracy we got from validation set which is make sense becuase we choose the hyper-parameters according to the validation accuracy. the lower accuracy we got from the test set which is also make sense because this is an unseen data set that we would like to predict. We can conclude that we manage to build a model without overfitting since the delta between the accuracies is relativly low.

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 (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html).

Compute the training, validation and test accuracy of this model.

In [53]:

```
import sklearn.linear_model as linear_model

model = linear_model.LogisticRegression(penalty='none').fit(train_norm_xs,
                                                             train_ts)

train_predict = model.predict(train_norm_xs)
train_acc = get_accuracy(train_predict, train_ts)

val_predict = model.predict(val_norm_xs)
val_acc = get_accuracy(val_predict, val_ts)

test_predict = model.predict(test_norm_xs)
test_acc = get_accuracy(test_predict, 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: Data
ConversionWarning: A column-vector y was passed when a 1d array was expecte
d. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)

train_acc =  0.7331858888365179  val_acc =  0.73036  test_acc =  0.726728646
1359674
```

It can be seen that we obtained similar results similar to the model Logistic_regression that we built above with the selected hyper-parameters. This confirms that we obtained optimal results using a logistic regression model.