Midterm Project Report

20180396 오지오

Task 1: Iris-dataset

1.

In multinomial logistic regression, we try to classify the target into more than two classes and in order to do this we need to compute the probability of y being in each classes, p(y=c|x). This multinomial classifier uses a generalization of the sigmoid function, which is the softmax function. Softmax function takes the input and maps them into a probability distribution, with all values in the range of (0,1) and summing to 1.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \le i \le k$$

The input to the softmax will be the dot product between the weight vector and the input vector plus the bias, thus the probability for each class will be denoted as the following.

$$p(y = c|x) = \frac{e^{w_c x + b_c}}{\sum_{j=1}^{k} e^{w_j x + b_j}}$$

Based on this knowledge, we can now compute the loss function, which has a slightly different structure than the binary logistic regression, since it uses the softmax function for a classifier instead of the sigmoid function. Basically, we use a cross-entropy loss (negative loglikelihood loss), where we compute how much our prediction, yhat, differs from the true data, y. It prefers the correct class labels of the training samples to be more likely. For a single example x, we can design the loss function as the sum of the logs of the output of the K classes.

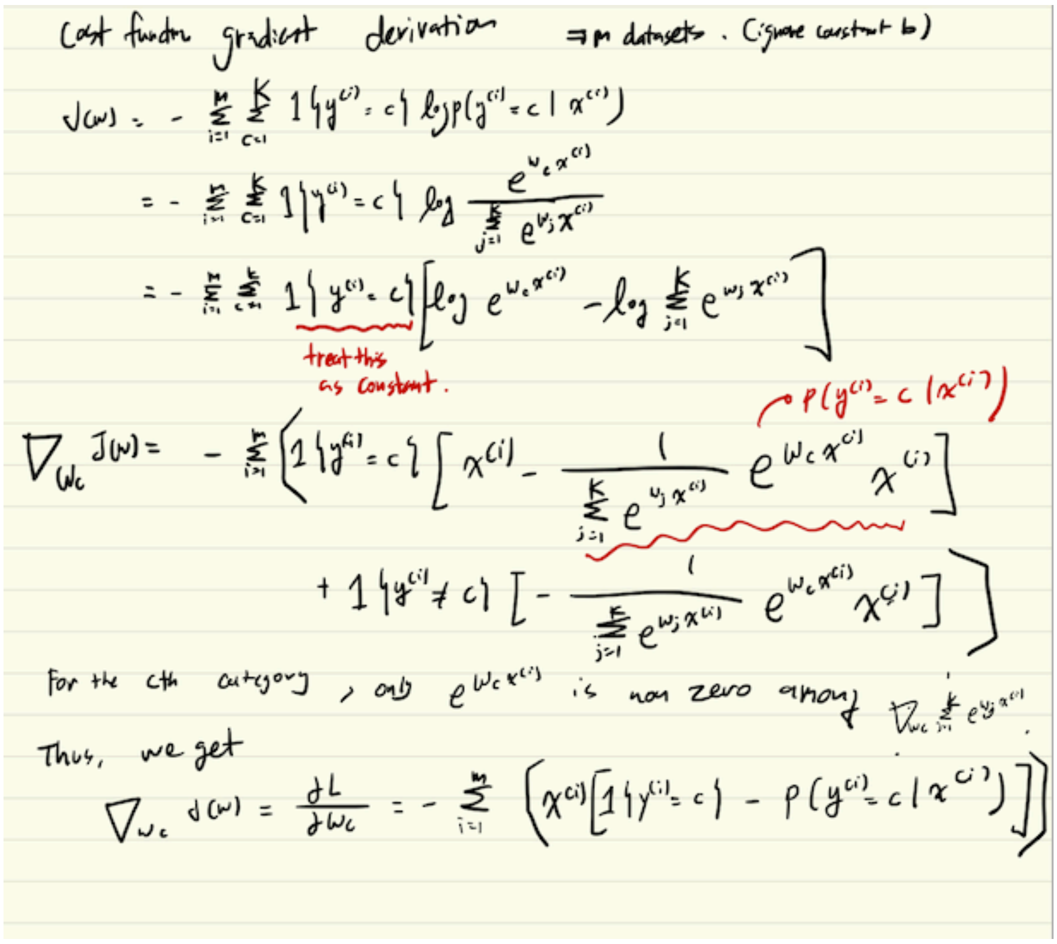
Loss = $L(y, \hat{y}) = -\sum_{c=1}^{K} 1\{y = c\} \log p(y = c|x) = -\sum_{c=1}^{K} 1\{y = c\} log \frac{e^{w_c x + b_c}}{\sum_{j=1}^{k} e^{w_j x + b_j}}$,

where 1{y=c} returns 1 when the condition in the bracket is true, else returns 0.

Now, based on this loss function, we should train the model using the gradient descent. Then, what is gradient descent? We try to use the loss function's slope/gradient to compute the smallest point of the function. Starting at a random point on a function, if we move the point in the direction of the gradient (in a negative direction by subtraction, since the gradient itself tends to go the maximum value) and repeat the process again several times, we can obtain the minimum point. We would like to minimize the loss in machine learning so the method can be denoted as $w_{i+1} = w - \eta \frac{\partial L}{\partial w_i}$. Here, η is the hyperparameter we should set in order to scale the movement. Now we should compute the gradient itself. It is expressed as the difference between the value for the true class c and the probability the classifier outputs for that particular class weighted by the value of the input x_c. It can be written as the following.

$$\frac{\partial L}{\partial w_c} = -\left(1\{y = c\} - \frac{e^{w_c x + b_c}}{\sum_{j=1}^{k} e^{w_j x + b_j}}\right) x_c$$

Now by setting the learning rate and the number of iterations (epoch), we can train our model using the gradient descent algorithm,



Cost function gradient derivation ⇒ m datasets. (ignore constant b)

$$J(w) = -\sum_{i=1}^{m}\sum_{c=1}^{K} 1\{y^{(i)}=c\}\log p(y^{(i)}=c \mid x^{(i)})$$

$$= -\sum_{i=1}^{m}\sum_{c=1}^{K} 1\{y^{(i)}=c\}\log \frac{e^{w_c x^{(i)}}}{\sum_{j=1}^{K} e^{w_j x^{(i)}}}$$

$$= -\sum_{i=1}^{m}\sum_{c=1}^{K} \underbrace{1\{y^{(i)}=c\}}_{\text{treat this as constant.}}\left[\log e^{w_c x^{(i)}} - \log \sum_{j=1}^{K} e^{w_j x^{(i)}}\right]$$

$$\nabla_{w_c} J(w) = -\sum_{i=1}^{m}\left\{1\{y^{(i)}=c\}\left[x^{(i)} - \overbrace{\frac{1}{\sum_{j=1}^{K} e^{w_j x^{(i)}}} e^{w_c x^{(i)}}}^{p(y^{(i)}=c \mid x^{(i)})} x^{(i)}\right]\right.$$

$$\left. + 1\{y^{(i)}\neq c\}\left[-\frac{1}{\sum_{j=1}^{K} e^{w_j x^{(i)}}} e^{w_c x^{(i)}} x^{(i)}\right]\right\}$$

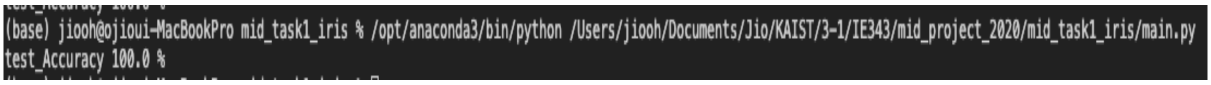
For the cth category, only $e^{w_c x^{(i)}}$ is non zero among $\nabla_{w_c} \sum_{j=1}^{K} e^{w_j x^{(i)}}$.

Thus, we get

$$\nabla_{w_c} J(w) = \frac{\partial L}{\partial w_c} = -\sum_{i=1}^{m}\left(x^{(i)}\left[1\{y^{(i)}=c\} - p(y^{(i)}=c \mid x^{(i)})\right]\right)$$

3.

```
import numpy as np
import pandas as pd
import math

class LogisticRegressor():
    #It initializes the values for the logistic regressor model.
    def __init__(self, w=None):
        # We initialize the weights to the weight that we got as the input. If the user didn't give any information about the weights, we just set is as None
        self.w = w
        #For this test, we tend to classify the data into 3 parts, thus we just initialize this for convenience.
        # In order for the model to be expanded we should change this fixed property
        self.num_class=3

    #METHOD: It trains the model
    def fit(self, X, y,lr,epoch_num):
        #We save the number of inputs as a varibale num_input for convenience.
        num_input = np.shape(X)[1]
        #We initialize the weights having a similar structure as the Xavier initialization
        self.w = np.random.randn(num_input,self.num_class)/np.sqrt(num_input * self.num_class)
        #UNUSED: We initialize the constant having a similar structure as the Xavier_initialization
        #self.b = np.random.randn(self.num_class)/np.sqrt(self.num_class)

        #In order to prevent some errorsome happening, copy the values of the instance variable w to local variable w
        w = self.w

        #for the number of epochs, given by the user, we iterate the process
        for _ in range(epoch_num):
            #Save the weight that was saved before.
            w_prev = np.copy(w)

            #Compute the scores for each data that by doing the dot operation of X and w.
            score = X @ w

            #After getting the scores for the data, put it into the softmax function so that we can get the probabilities of each data for each class.
            yhat = self.softmax(score)

            #Now, we apply cross entropy and get the loss as well as the gradient value at that certain point.
            loss,grad   = self.cross_entropy(X,yhat,y)

            #We apply stochastic gradient descent with the given learning rate by the user and save it as w.
            w = self.SGD(w, grad, lr)

            #Then, we save the weights as the instance variable.
            self.w =w

            #If the weights are the same within the tolerance values, that means that the further iterations are useless,
            #since it is repeating the same process again and again, thus exit the loop
            if np.allclose(w, w_prev):
                break




    #UNUSED METHOD: For a matrix z, it returns the column index that has the largest value for each row.
    def get_class(self,z):
        return np.argmax(z,axis= 1)

    #METHOD: Returns the score for the given input X based on the current model's weight saved as the instance variable (self.w)
    def predict(self, X):
        return X@self.w

    #METHOD: Returns a matrix that applied softmax function for each of the rows.
    def softmax(self,z):
        #Regularization
        #In order to avoid overflow when performing e^z, subtract the maximum value in the input matrix.
        #This doesn't change the function's result since each element is divided by the sum of the elements in the row that the element is a part of.
        z -= np.max(z)

        #We apply the exponential function for every element in the matrix
        z_exp = np.exp(z)
        #We divide each element by the some of the elements in the that the element is a part of, inorder to compute the probability.
        #Then the matrix will have rows consisting of probabilities of each datum for each class.
        #For example, First row <=> First datum / First column <=> Probability of the datum to be classified as class 0.
        for i in range(z_exp.shape[0]):
            z_exp[i] = z_exp[i]/np.sum(z_exp[i])

        #Return the matrix, sigmoid function applied.
        return z_exp

    #METHOD: Returns the weight, in which stochastic gradient descent is applied.
    def SGD(self,w, grad, lr):
        #From the original weight we subtract the product of the learning rate and the gradient, to get a better-performing weight.
        w -= lr*grad
        return w

    #METHOD: Returns the matrix so that it has the form of one-hot-row-vector
    def one_hot_encoding(self,y):
        #ENCODING
        #y currently has elements with the "number" of the class for each datum.
        #Encode the class labels so that we could relatively easy to work with when dealing with the loss function (CE)
        #For instance, if y is [0,2] change it to [[1,0,0],[0,0,1]]
        y_one_hot_encoded = (np.arange(np.max(y) + 1) == y[:, None]).astype(float)
        return y_one_hot_encoded

    #METHOD: The loss function, using the cross entropy loss function
    #          Returns the loss value and the gradient
    def cross_entropy(self,x,yhat, y):
        #Saves the number of data a variable (size) for further use.
        size = x.shape[0]

        #Encode y to get a one-hot-vector in every row.
        y_one_hot_encoded= self.one_hot_encoding(y)

        #Get the loss value, though it is unused in this project, inserted it, since it might be used when the model is enlarged
        loss= - np.sum(np.log(yhat) * y_one_hot_encoded, axis=1)

        #Compute the gradient
        #Multiplied -1 to size, since the gradient should have a -1 multiplied value but no negative signs exist in the nominator
        #(put it at the denominator just for easier interpretation)
        grad = x.T@(y_one_hot_encoded - yhat) / (-1*size)

        return loss, grad
```

<RESULT>


```
(base) jiooh@ojioui-MacBookPro mid_task1_iris % /opt/anaconda3/bin/python /Users/jiooh/Documents/Jio/KAIST/3-1/IE343/mid_project_2020/mid_task1_iris/main.py
test_Accuracy 100.0 %
```

The accuracy of the model was 100% for the learning rate 0.01 and the epoch of 5000. This dataset seems to be simple, thus possible to get high accuracy, though in the real world, a model with 100% accuracy is almost impossible.

<Xavier Initialization>
In the above program, I initialized the weights as using similar methods to Xavier initialization. So, it is quite essential to select the right initialization value for the weights in order to perform an optimal training process. The most basic and simple way to initialize the weight is to assign zero. Frankly, in logistic regression the function is convex, so the initialization to zero, doesn't really matter, since we get to eventually obtain the optimal parameters. However, it is terrible for neural networks. In my model, I just initialized the weights mimicking the Xavier Initialization, used commonly in deep networks, which divides the random weight, sampled by the univariate Gaussian distribution, by the square root of the nodes. In order to mimic this, I just divided the product of the number of inputs and classes.

4.
I definitely believe that binary logistic regression models without being expanded to softmax regression can be used to multi-class problems. Here, I would like to propose two methods to solve this case.

To begin with, I've thought of the case where I could split the multi-class classification model into multiple binary classification models. The predictions in this case will be derived by the most confident model, among them. It sees the samples of that particular class as positive, while sees the others as negative. For instance, if we have three classes labelled as 0, 1, and 2, we classify the three classes as 0 vs [1,2]; 1 vs [0,2]; 2 vs [0,1]. Thus, if we put the inputs (samples, labels, and the training algorithm), we can get the output of K trained classifiers, where K indicates the number of classes. It has classifiers for each class and the class that gives the maximum score will be selected. I believe that this might cause some problems, since even though the training set is balanced, the binary classification learners might see some unbalanced distributions because there will be much more negative sets than the positive sets. This happens because of the classification method we use: one versus the rest. Moreover, it needs models for each class, which might be a problem for heavy datasets. Lastly, the confidence range or values might vary between each classifier.

Secondly, in this case we also split the multi-class classification model into multiple binary classification models, but here we split it quite differently. We split the dataset into one class versus the other class. For example, if we have 3 classes as 0, 1, 2, we classify the dataset as 0 vs 1; 0 vs 2; 1 vs 2. We choose two of the K classes, where K is the total number of classes, so we can see that there will be $\binom{K}{2}$= K*(K-1)/2 models for the problem. Each of the binary classifications will predict the class among the two, and the class that has been chosen the most (voted the most) by the binary classifiers will be selected. If the classifier returns the probability (score), the argmax of the sum of the scores, will be selected. I believe that this will have similar problems as the above one. We need bunch of models, thus inefficient for heavy datasets and if some class gets the same number of votes, it is ambiguous, since we don't know which class to select.

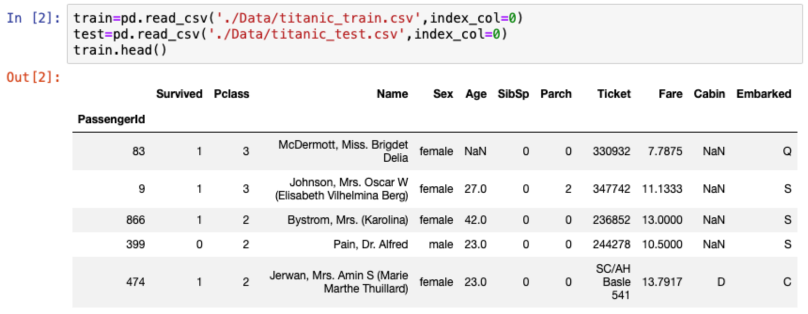
Task 2: Titanic dataset

2. Data Preprocessing

<1> Theoretical Background
Briefly, data preprocessing is a data mining technique, where we transform the raw data available in the internet into a format that is understandable. We modify the data so that we fill out lacking values or remove data that

seems irrelevant, plus introduce new variables so that we could explain the model more expressively. We exclude the outliers in order to put data into models.

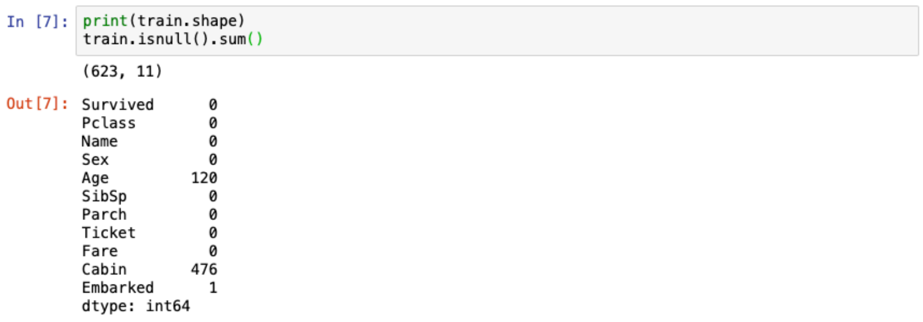
<2> My implementation
I used Jupyter in order to check which data might be useful for classifying survivors. Starting off, this is the basic structure of our data. (PassengerId column is used as the index, thus it isn't in the training set in our code)



```
In [2]: train=pd.read_csv('./Data/titanic_train.csv',index_col=0)
        test=pd.read_csv('./Data/titanic_test.csv',index_col=0)
        train.head()
```

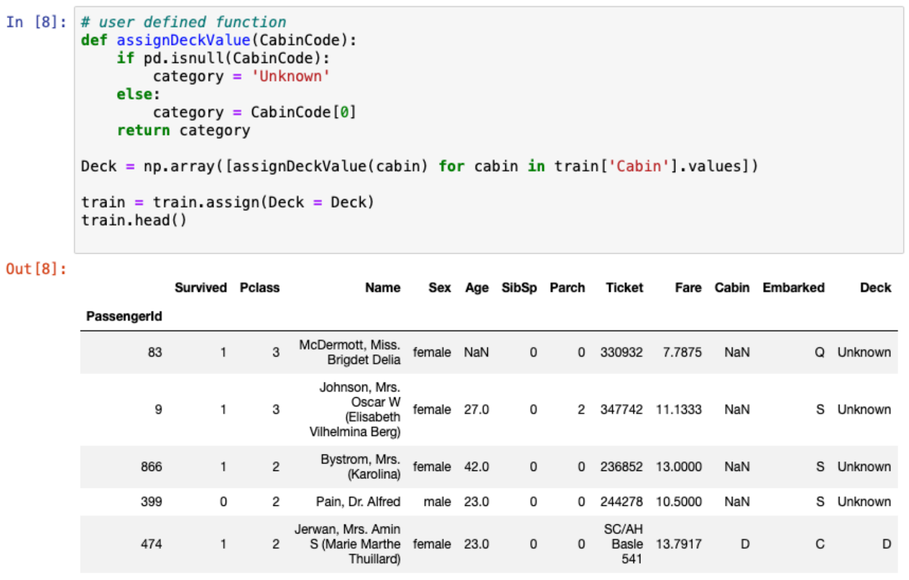
Out[2]:

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId											
83	1	3	McDermott, Miss. Bridget Delia	female	NaN	0	0	330932	7.7875	NaN	Q
9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	NaN	S
866	1	2	Bystrom, Mrs. (Karolina)	female	42.0	0	0	236852	13.0000	NaN	S
399	0	2	Pain, Dr. Alfred	male	23.0	0	0	244278	10.5000	NaN	S
474	1	2	Jerwan, Mrs. Amin S (Marie Marthe Thuillard)	female	23.0	0	0	SC/AH Basle 541	13.7917	D	C



```
In [7]: print(train.shape)
        train.isnull().sum()

        (623, 11)

Out[7]: Survived      0
        Pclass        0
        Name          0
        Sex           0
        Age         120
        SibSp         0
        Parch         0
        Ticket        0
        Fare          0
        Cabin       476
        Embarked      1
        dtype: int64
```

Among 623 training data 476 data for cabin information is lacking, thus we might think that dropping the information of the cabins for passengers might be efficient. When I closely examined the cabin column in our training data, I found that the character at the beginning denoted the deck number for the remaining approximately 150 data. It might be useful for some cases, so I assigned 476 missing data as Unknown and classified the decks for the rest of the data.

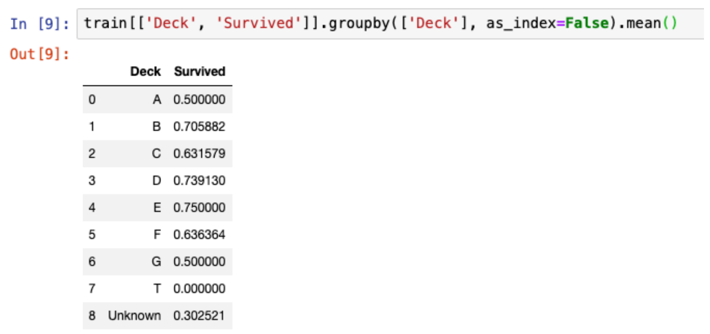


```
In [8]: # user defined function
        def assignDeckValue(CabinCode):
            if pd.isnull(CabinCode):
                category = 'Unknown'
            else:
                category = CabinCode[0]
            return category

        Deck = np.array([assignDeckValue(cabin) for cabin in train['Cabin'].values])

        train = train.assign(Deck = Deck)
        train.head()
```

Out[8]:

	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Deck
PassengerId												
83	1	3	McDermott, Miss. Bridget Delia	female	NaN	0	0	330932	7.7875	NaN	Q	Unknown
9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	NaN	S	Unknown
866	1	2	Bystrom, Mrs. (Karolina)	female	42.0	0	0	236852	13.0000	NaN	S	Unknown
399	0	2	Pain, Dr. Alfred	male	23.0	0	0	244278	10.5000	NaN	S	Unknown
474	1	2	Jerwan, Mrs. Amin S (Marie Marthe Thuillard)	female	23.0	0	0	SC/AH Basle 541	13.7917	D	C	D

Now I need to check if this deck data is needed, so I checked the rate of survival for each deck.

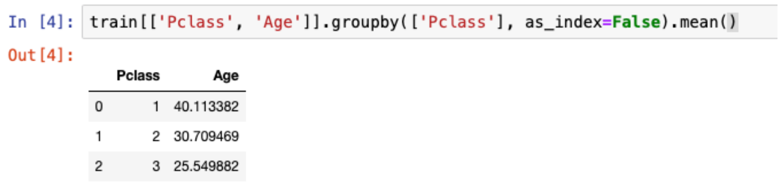


```
In [9]: train[['Deck', 'Survived']].groupby(['Deck'], as_index=False).mean()
```

Out[9]:

	Deck	Survived
0	A	0.500000
1	B	0.705882
2	C	0.631579
3	D	0.739130
4	E	0.750000
5	F	0.636364
6	G	0.500000
7	T	0.000000
8	Unknown	0.302521

There is not much difference in survival for decks A through G (since the unknown decks really don't give us much information about correlation between the decks and survival), thus I decided to drop the Cabin column. Now, I thought that age will be an important factor when we classify the survival rate and it took a long time for me to think of how to use this age factor.

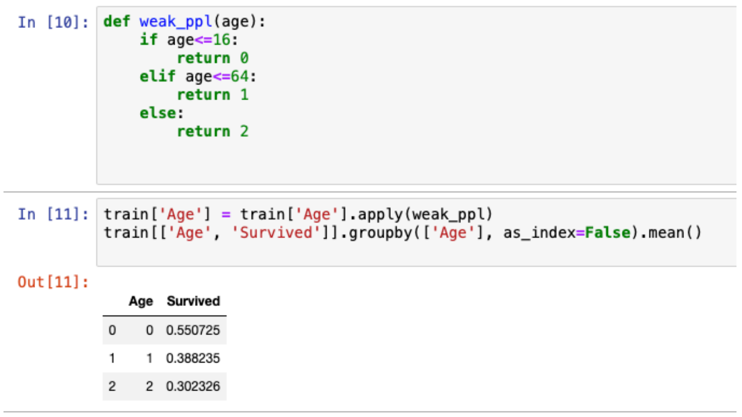


```
In [4]: train[['Pclass', 'Age']].groupby(['Pclass'], as_index=False).mean()
```

Out[4]:

	Pclass	Age
0	1	40.113382
1	2	30.709469
2	3	25.549882

Before starting, I needed to fill out the 120 lacked age data. Dropping out the people with no age data, will lose too much training data, so I decided to fill out the age factor by putting the mean of the age of the people of each passenger classes. Since, I thought that more age might be related to higher social class, which was true.
Now, will it be okay to consider age as a quantitative value? At beginning I thought that the children and old people will have a high survival rate, since they had the priority to escape the boat. So, I thought that I might change the "age" factor as a qualitative factor so that I classify if a person is an elderly, children, or an adult. By doing this, we classify the whole "age unknown people" to the same category, thus the above process, might be unneeded (it would be more efficient to put fill the blanks as 20, but I just left it, since my classification via age might modified in the future.



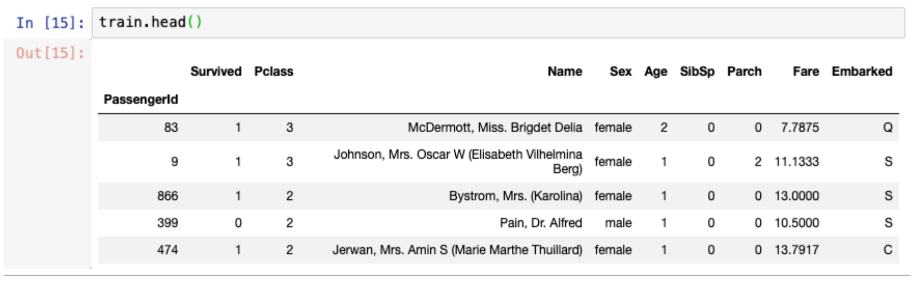
```
In [10]: def weak_ppl(age):
             if age<=16:
                 return 0
             elif age<=64:
                 return 1
             else:
                 return 2
```

```
In [11]: train['Age'] = train['Age'].apply(weak_ppl)
         train[['Age', 'Survived']].groupby(['Age'], as_index=False).mean()
```

Out[11]:

	Age	Survived
0	0	0.550725
1	1	0.388235
2	2	0.302326

The result of survival was a little different from my expectations, since the elder people survival rate was lower than any other classes. Even though they had the priority to get off the boat, I think that their inability to move quick might have caused this phenomenon. Some significant differences between classes, have been found, so I think that age factor will work well for this classification model.
The next thing that I thought that it wasn't useful was the ticket number, the ticket number is assigned randomly, so we should drop the column.

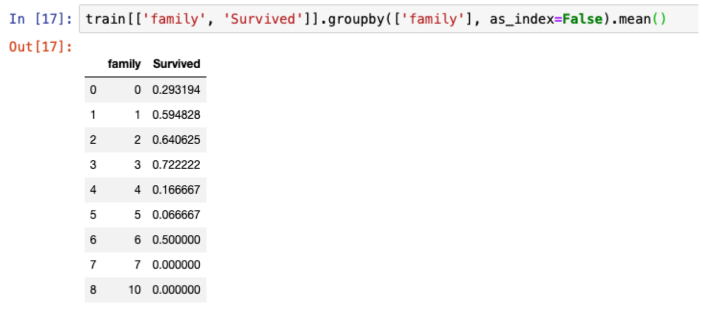


```
In [15]: train.head()
```

Out[15]:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Fare	Embarked
83	1	3	McDermott, Miss. Bridget Delia	female	2	0	0	7.7875	Q
9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	1	0	2	11.1333	S
866	1	2	Bystrom, Mrs. (Karolina)	female	1	0	0	13.0000	S
399	0	2	Pain, Dr. Alfred	male	1	0	0	10.5000	S
474	1	2	Jerwan, Mrs. Amin S (Marie Marthe Thuillard)	female	1	0	0	13.7917	C

Now we have these features left in our model. It is better to have less parameters in order to prevent overfitting, I thought about what to do in order to get the optimal solution. I saw some relationship between Parch (# of parent, child in the same boat) and the SibSp (# of siblings in the same boat). Both factors are families and I believed that if there are some family members left in the boat the individual might have gone back to the sinking boat and might have less probability to survive. On the other hand, there might be some cases in which the family member has saved one's life. So, I summed up the SibSp and Parch as the variable family, indicating the number of family members in the same boat.

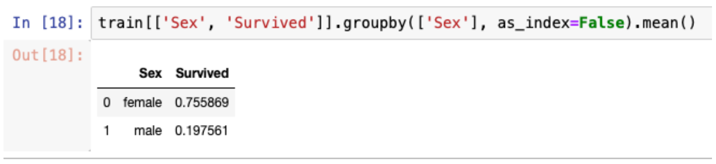


```
In [17]: train[['family', 'Survived']].groupby(['family'], as_index=False).mean()
Out[17]:
```

	family	Survived
0	0	0.293194
1	1	0.594828
2	2	0.640625
3	3	0.722222
4	4	0.166667
5	5	0.066667
6	6	0.500000
7	7	0.000000
8	10	0.000000

The number of families seemed to be related to the survival rate, since it showed clearly different survival rate for each number of families on board, thus it seems to be a good parameter.

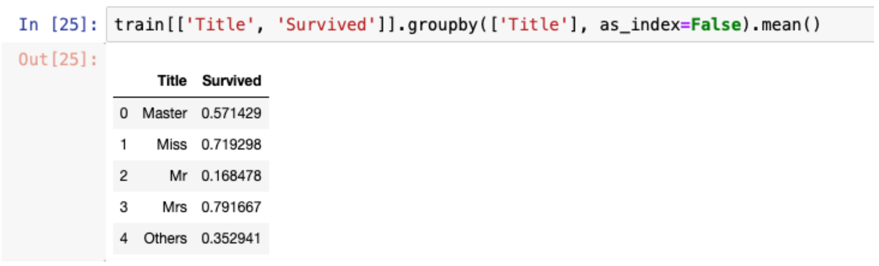
For the next step, I considered the name variable. At the beginning, I thought that name wasn't really important and dropped the column. However, it seemed their status or title in the society might give some effect in the survival rate. Especially since women had the priority to escape the boat. There is the "Sex" factor that classifies this part, but using this might even strengthen this feature in the model. It seemed that the sex of a person was an essential factor in survival since the survival rate for women was extremely higher than that of men. Thus, strengthening this factor didn't seem like a bad idea.



```
In [18]: train[['Sex', 'Survived']].groupby(['Sex'], as_index=False).mean()
Out[18]:
```

	Sex	Survived
0	female	0.755869
1	male	0.197561

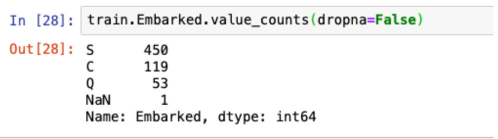
To begin with, I extracted the titles in the person's name and checked how many titles are there as well as how many people are classified as the corresponding title. Then, I classified most men and some minor titles as Others as well as classifying Miss, Mrs. differently (since there were large numbers for them), and classify masters as a separate group. I did this because I thought at that time masters might indicate a high social degree, in which they might have the priority to escape the sunken ship.



```
In [25]: train[['Title', 'Survived']].groupby(['Title'], as_index=False).mean()
Out[25]:
```

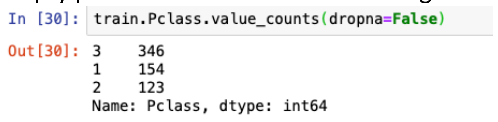
	Title	Survived
0	Master	0.571429
1	Miss	0.719298
2	Mr	0.168478
3	Mrs	0.791667
4	Others	0.352941

As we can see above, the title factor also had some significant effects on the rate of survival. So, this processing with the "name" factor was beneficial. After dropping this "name" factor, I found that there is one lacking information about the "Embarked" factor.



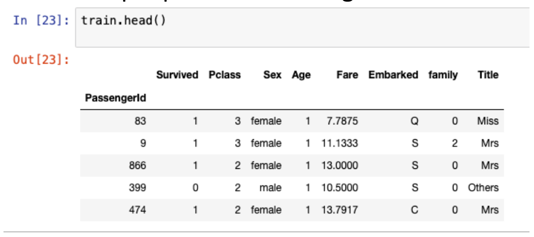
```
In [28]: train.Embarked.value_counts(dropna=False)
Out[28]: S      450
         C      119
         Q       53
         NaN      1
         Name: Embarked, dtype: int64
```

Since almost the majority of the people on board have departed from Southampton, I just simply filled out the empty part as "S" in order to maintain generality.



```
In [30]: train.Pclass.value_counts(dropna=False)
Out[30]: 3    346
         1    154
         2    123
         Name: Pclass, dtype: int64
```

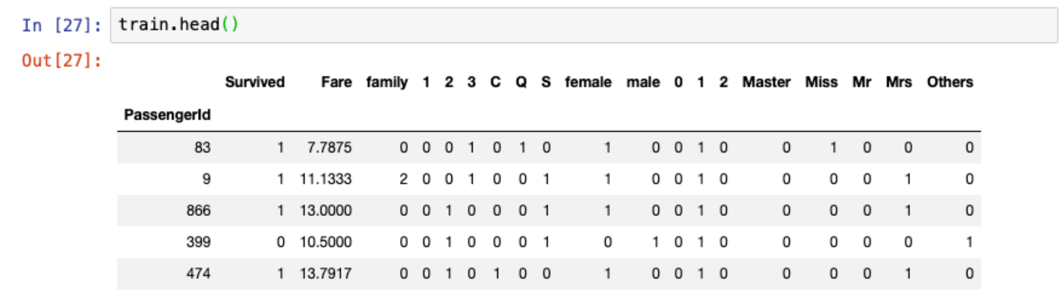
Plus, I filled out the fare column as 13.314 in order to be ready for the lack of the information of fares, since fare is a quantitative value that might cause lots of trouble, if just computed as 0. This feature doesn't lack in training sets, but there might be some probabilities that this might happen in some cases. I filled out 13.314, since almost half of the people in the training set had 3rd class seats, and the average fare for the 3rd class was 13.314.



```
In [23]: train.head()

Out[23]:
```

PassengerId	Survived	Pclass	Sex	Age	Fare	Embarked	family	Title
83	1	3	female	1	7.7875	Q	0	Miss
9	1	3	female	1	11.1333	S	2	Mrs
866	1	2	female	1	13.0000	S	0	Mrs
399	0	2	male	1	10.5000	S	0	Others
474	1	2	female	1	13.7917	C	0	Mrs

Now we finished most of the preprocessing, and the only thing left is the last process, of creating dummy variables for qualitative variables. This is an essential step, since regression itself are well known for using continuous variables. However, for quantitative variables, for instance, we assigned the "Embarked" factor among S, Q, and C and let's say we mapped these into 0, 1, and 2. If we just use this variable, without any modification, there might be some misinterpretation since there is no such thing as an increase in the factor "Embarked". We can't say that C is higher than Q or S, thus cannot give them a score. In order to solve this, we make a separate column for each category (C, Q, S) so that the model treat it as having a high score (1) or a low score (0) for each category. This process creates dummy variables for classification. We have quantitative factors, "Pclass, Sex, Age, Embarked, Title" In our model, so we do the above process, which leaves us the final data.



```
In [27]: train.head()
Out[27]:
```

PassengerId	Survived	Fare	family	1	2	3	C	Q	S	female	male	0	1	2	Master	Miss	Mr	Mrs	Others
83	1	7.7875	0	0	0	1	0	1	0	1	0	0	1	0	0	1	0	0	0
9	1	11.1333	2	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0
866	1	13.0000	0	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0
399	0	10.5000	0	0	1	0	0	0	1	0	1	0	1	0	0	0	0	0	1
474	1	13.7917	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0	1	0

Now that we are done, we copy the survived column, which is the output to an additional variable y and add an additional column for constants and we are ready to train our model. (Plus, all the same process is done for the test data.)

3. Logistic Regression

<1> Theoretical Background

In logistic regression in this project, we try to train the system using the AdaGrad and the BCE and test is by computing p(C|x) and return the higher probability label, for instance if it is bigger than .5, y is 1 and 0 if not. We have a goal to train a classifier that can make a binary decision about the class of a new input.
So, in order to perform logistic regression, we try to find a score function, which is consisted of weights and bias, that gives us a score when we put the input features x to the function. ($z = b + w^T x$) Using this score, we tend to calculate the probability of each cases, we'll pass z through the sigmoid function. ($\sigma(z) = \frac{1}{1+e^{-z}}$) This indicates the probability of each type to be in class 1 and if we subtract this value from 1, that will indicate the probability of that data to be in class 0. This is because the sigmoid function itself calculates the probability of each data being in one of the classes, which I denoted as class 1 above.
Here, the loss function is written as the following.

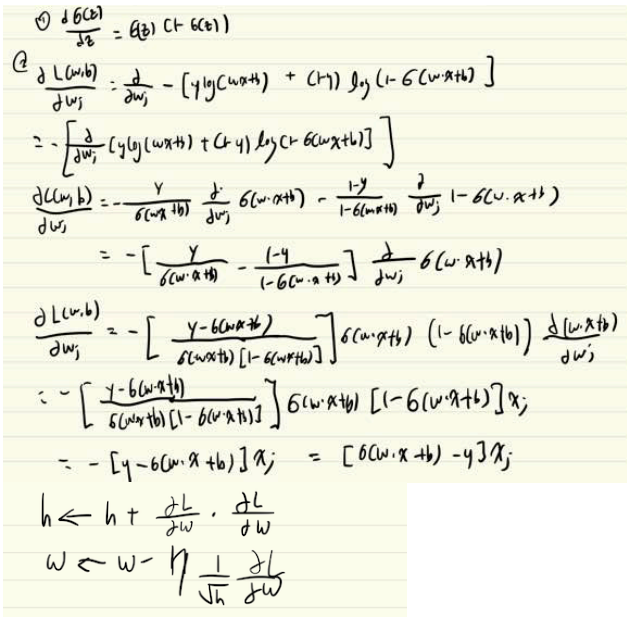
Since only two outcomes exist → Bernoulli distribution

$p(y|x) = \hat{y}^{y}(1-\hat{y})^{1-y}$ $\qquad \hat{y} = \sigma(w \cdot x + b) = \sigma(z)$

$\log p(y|x) = y \log \hat{y} + (1-y)\log(1-\hat{y})$

$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$

Using, this loss function we try to train our model using a similar concept of AdaGrad, which will be explained a bit later. Here, we compute the predicted y_hat and compute the loss for each prediction. Then, compute the gradient of the loss function with respect to the weights of the score function, which tells us how to move the weight to maximize the loss. Then, we go the other way, in other words, we subtract the product of the hyperparameter (learning rate) and the gradient from the original weight divided by the new variable h, which is the sum of all gradients for the points that was used until now, and get a new weight. We iterate this over and over to optimize the model. The following picture indicates the process and derivation of gradients. This only shows the gradient of one weight and datum, in our code this is done by matrix multiplication in order to handle numerous data.



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

② $\frac{\partial L(w,b)}{\partial w_j} = \frac{\partial}{\partial w_j} - [y \log(w \cdot x + b) + (1-y) \log(1-\sigma(w \cdot x + b)]$

$= -\left[\frac{\partial}{\partial w_j} (y \log(w \cdot x + b) + (1-y) \log(1-\sigma(w \cdot x + b))]\right]$

$\frac{\partial L(w,b)}{\partial w_j} = -\frac{y}{\sigma(w \cdot x + b)} \frac{\partial}{\partial w_j} \sigma(w \cdot x + b) - \frac{1-y}{1-\sigma(w \cdot x + b)} \frac{\partial}{\partial w_j} 1-\sigma(w \cdot x + b))$

$= -\left[\frac{y}{\sigma(w \cdot x + b)} - \frac{1-y}{1-\sigma(w \cdot x + b)}\right] \frac{\partial}{\partial w_j} \sigma(w \cdot x + b)$

$\frac{\partial L(w,b)}{\partial w_j} = -\left[\frac{y-\sigma(w \cdot x + b)}{\sigma(w \cdot x + b)[1-\sigma(w \cdot x + b)]}\right] \sigma(w \cdot x + b)\,(1-\sigma(w \cdot x + b))\,\frac{\partial (w \cdot x + b)}{\partial w_j}$

$= -\left[\frac{y-\sigma(w \cdot x + b)}{\sigma(w \cdot x + b)[1-\sigma(w \cdot x + b)]}\right] \sigma(w \cdot x + b)\,[1-\sigma(w \cdot x + b)]\,x_j$

$= -[y-\sigma(w \cdot x + b)]\,x_j = [\sigma(w \cdot x + b) - y]\,x_j$

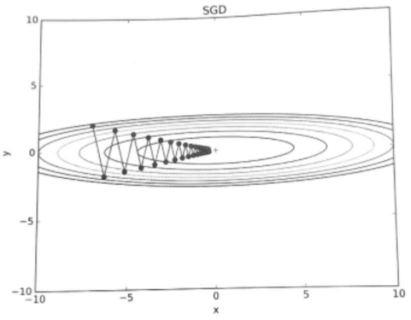
$h \leftarrow h + \frac{\partial L}{\partial W} \cdot \frac{\partial L}{\partial W}$

$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$

<AdaGrad>

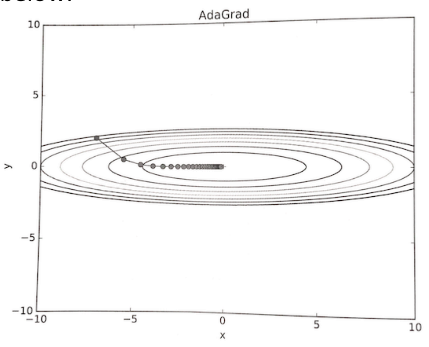
So, what is AdaGrad, and why do we use it? Stochastic gradient descent (SGD) can be expressed as $w_{i+1} = w - \eta \frac{\partial L}{\partial w_i}$. This is simple and its implementation is quite simple, but the problem is that it might be in efficient.

In some unlucky cases, when finding the optimal solution, it shows inefficiency as the picture below, especially for functions, where the slope differs from point to point.



Due to this inefficiency, when I used SGD as my optimizer, I found out that I got different results in every iteration and the accuracy for the model was ranging from 75~81. I needed to increase the number of iterations in order to shrink the range, but it seemed it is a waste of time to do this. So, in order to prevent this inefficiency, there were three optimizers that I could use in hand (Momentum, AdaGrad, Adam), and the AdaGrad seemed to perform very well with the least time and error.

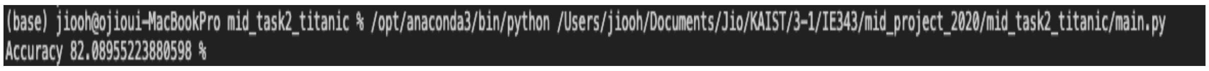
Now, back to our main point, in training, learning rate is crucial. Too high learning rates lead to divergence, while too small learning rates takes too much time. This learning rate is a fixed hyperparameter, in other words cannot be changed during the training. We try to overcome this shorthand, by decaying the learning rate using a new variable h. We decrease the learning rate adaptively as we increase the number of iterations. We increase the value of h by adding the square of gradients at every iteration and divide the square root of h to the learning rate so that we get, $h_{i+1} = h_i + \frac{\partial L}{\partial W} \times \frac{\partial L}{\partial W}$ (the multiplication sign indicates the multiplication of each element in the matrix) and $w_{i+1} = w - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial w_i}$. This not only reduces the movement as we increase the iteration number, but also reduces the movement when the gradient is very high, compared to other points. It works as a regularization factor. Using this method, the optimal point is found with the motion described in the figure below.



In the real experiment that I have done, the result came out very fast, with high constant accuracy of 82.1% due to the condition where the weight is same as the previous weight, it breaks out the iteration, which means that the decay of learning rate worked efficiently.

<2> My implementation

```
import numpy as np
import pandas as pd
import math

class LogisticRegressor():
    #It initializes the values for the logistic regressor model.
    def __init__(self, w=None):
        # We initialize the weights to the weight that we got as the input. If the
user didn't give any information about the weights, we just set is as None
        self.w = w
        # We initialize the h variable, which will be used in AdaGrad.
        self.h = 0

    #Computes the sigmoid function for all elements in the matrix, this is used for
logistic regression.
    def sigmoid(self, a):
        return 1 / (1 + np.exp(-1 * a))

    #METHOD: It trains the model
    def fit(self, X, y, lr, epoch_num):
        #We initialize the weights having a similar structure as the Xavier
initialization
        self.w = np.random.randn(np.size(X,1))/np.sqrt(np.size(X,1))
        #In order to prevent some errorsome happening, copy the values of the
instance variable w to local variable w
        w = self.w

        #Reshape the y as a column vector for convenience in matrix multiplication
        y = y.reshape(-1,1)

        #for the number of epochs, given by the user, we iterate the process
        for _ in range(epoch_num):
            #Save the weight that was saved before.
            w_prev = np.copy(w)
            #Get the score function for the given input data
            y_hat = self.proba(X)

            #Compute the gradient based on the loss function, binary cross entropy
            grad = X.T @ (y_hat-y)

            #Update the parameter based on the AdaGrad method with the learning
rate given by the user
            w = self.AdaGrad(w, grad, lr)

            #If the weights are the same within the tolerance values, that means
that the further iterations are useless,
            #since it is repeating the same process again and again, thus exit the
loop
            if np.allclose(w, w_prev):
                break

            #Then, we save the weights as the instance variable.
            self.w = w

    #METHOD: Applies the AdaGrad method using the grad computed in the fit
function, and the learning rate given by the user
    def AdaGrad(self,w, grad, lr):
        # update the regularization variable h by adding the square of the grad of
that certain point
        # This variable is an instance variable, so it increases rapidly as
iterations increase
        # Punishes based on iteration numbers and gradient size
        self.h = self.h+ grad*grad

        #Change the weight into a column vector for matrix multiplication
        w = self.w.reshape(-1,1)

        #Apply the AdaGrad -> w <- w - lr *grad/sqrt(h) +1e-7 is added, in order to
prevent div by zero
        w = w-lr*grad/(np.sqrt(self.h)+ 1e-7)
        return w

    #METHOD: Based on the input, return the score function based on the input
    def proba(self, X):

        #Change the weight into a column vector for matrix multiplication
        w = self.w.reshape(-1,1)

        #Score Function dot product of X and W
        return self.sigmoid(X @ w)

    #METHOD: Predicting the score based on the input -> Same as the proba function,
but used in the main function for predicting the value of test inputs
    def predict(self, X):
        return self.proba(X)
    #UNUSED METHOD: Returns the cross entropy function loss value
    #Might be used for future use.
    def BCE(self,y, y_hat):
        BCE = -1 * np.mean(y * np.log(y_hat + 1e-7) + (1-y) * np.log(1 - y_hat +
1e-7))
        return BCE
```

<RESULT>


```
(base) jiooh@ojioui-MacBookPro mid_task2_titanic % /opt/anaconda3/bin/python /Users/jiooh/Documents/Jio/KAIST/3-1/IE343/mid_project_2020/mid_task2_titanic/main.py
Accuracy 82.08955223880598 %
```

The accuracy of the result was about 82%, when the learning rate is 0.01 with epoch of 2000000. Honestly, with much smaller epoch or big learning rates, the results were same thanks to AdaGrad, which penalizes huge gradients and increase in iterations, however, for new datasets, we cannot predict what can happen, thus I just left the values as it is. (It seems that the iteration is escaped by np.allclose() since the execution time is significantly small.)

(Extra) Further implementation/Limits
The training wasn't perfect since 18% of the data wasn't classified correctly. I proposed some qualities or features that might be changed to enhance the performance.
(1) Forward Selection
The model might be encountering overfitting problems, so using forward or backward selection might be a good idea to reduce the error.

(2) Age Classification
I classified the ages range into 3 groups, based on my own inference. Maybe if I have a better classification method for this, I might have a better performance.