Final Term Project Report

20180396오지오

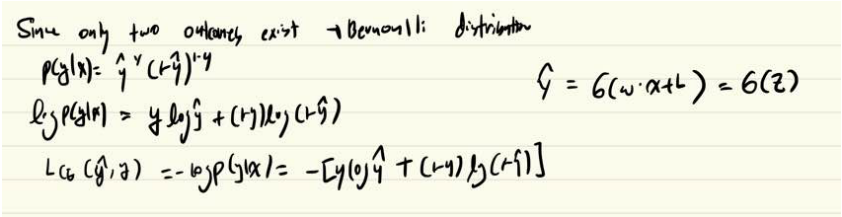
(1)

<Logistic Regression>

A discriminative classifier tries to learn how to distinguish the classes and they do not care about the general characteristics of each class. They try to compute the $P(C_1|x)$ directly. There are four basic components in a classification system, a feature representation of the input, the classification function that determines the y_hat, objective function, and the algorithm to optimize the objective function. In logistic regression in this project, we try to train the system using the gradient descent 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. Simply speaking, we calculate the score function with the linear function denoted as $z=\sum_{i=1}^{d} \theta_i X_i + \theta_0$ for each class and put in the sigmoid function, where we get the probability and derive the expectations based on the standard. With that expectation we train the model so that we get the optimal weight, indicated as θ.

Here, the loss function is written as the following.



In logistic regression, with the score function of $z = X\theta$ (as we denoted it as in the codes) in matrix form, each of the letters have dimensions as the following. Firstly, X will have the dimension of (n, d+1) and θ will have the dimension of (d+1,1), which is a column vector. For both X and θ we have one more column or row than the number of parameters, i.e. we have the term d+1, since we added an additional bias term to the model. Lastly, obvious y and $\hat{y}$, will all have dimension (n,1)

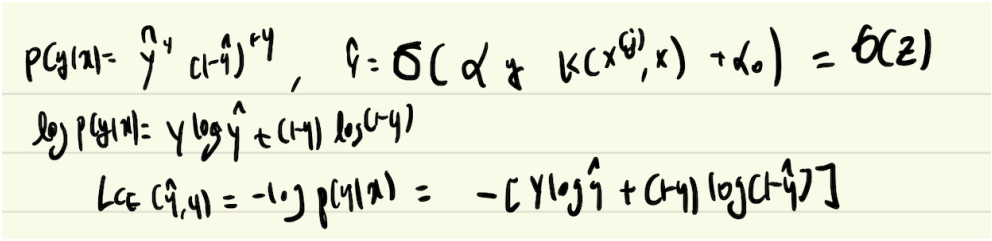
<Kernel Logistic Regression>

The kernel logistic regression uses kernels to construct the model with the help of the methods that we used for the SVM kernels. Kernels allow us to represent the data only though a set of pairwise similarity comparisons between the original data observations, with the original coordinates in lower dimensional spaces, instead of explicitly applying the transformations and representing the data by these transformed coordinates in the higher dimensional feature spaces. We do this with the help of kernel functions, where they accept inputs in the lower dimensional space and returns the dot product of the transformed vectors in higher dimensional spaces. By the representor theorem, we can show that for the hyperplane $z=\sum_{i=1}^{d} \theta_i \phi(X_i) + \theta_0$, we can derive the optimal θ as $\sum_{j=1}^{N} \alpha_j y^{(j)} \phi(X^{(j)})$. (d: number of parameters, N: number of datapoints) We define the kernel function K as $K(X^{(i)}, X^{(j)}) = < \phi(X^{(i)}), \phi(X^{(j)}) >$, where <..> indicates the inner product of the two components. From this, we can change the current P(Y|X) as the following.

$$P(Y=1|X) = \frac{1}{1+\exp\left(< (\theta_0, \theta), concatenate\big(1, \phi(X)\big) >\right)}$$

$$= \frac{1}{1+\exp\left(< (\theta_0, \sum_{j=1}^{N} \alpha_j y^{(j)} \phi(X^{(j)})), concatenate\big(1, \phi(X)\big) >\right)}$$

$$= \frac{1}{1+\exp\left(\sum_{j=1}^{N} \alpha_j y^{(j)} < \phi(X^{(j)}), \phi(X) > + \alpha_0\right)} = \frac{1}{1+\exp\left(\sum_{j=1}^{N} \alpha_j y^{(j)} K(X^{(j)}, X) + \alpha_0\right)}$$



Based on this result we do the process we did above, by deciding whether the current result will be classified to which class, using the same loss function as above. (bigger than 0.5->class 1, smaller than 0.5->class 0)

Here we tend train this alpha to minimize the loss. On the above equation, $\alpha_j, y^{(j)}, K(X^{(j)}, X)$ are all scalar values, however we can express this in a matrix form combining all the values so that, the dimension of alpha is (n, 1). K(X,X) (X: training data, X': test data) has a dimension of (n, n) ((n,n) matrix is made since, we calculate and merge the inner product from one training point (that is transformed via the kernel function) to another), and lastly y has the dimension of (n,1). Additionally, when in the stage of testing, K(X,X') will have a dimension of (# of test data points, n), since we are calculating and merging the inner product of test points and training points (that are transformed via the kernel function). Additionally, bias terms will be concatenated into matrices for convenience in the codes, giving a slightly different dimension, which will be explained in the codes. When, deriving the score function, (which will be explained afterwards), I added temporarily added 1 in the y, in order to match the dimensions and preserve the bias term, since if I add 0, the bias term won't be considered.

(2)

In normal logistic regression, we have a linear score function. Thus, if the data aren't separable with a linear boundary, the logistic regression will underperform. We might use feature engineering by using polynomial functions to solve this and when we use them, these transformations and applications are costly and impractical. We can solve these problems using the kernels. But kernel functions allow us to operate in the original feature space without computing the coordinates of a data in a higher dimensional space. Kernels allow us to separate data that are only linearly separable as $\phi(x)$ in higher dimensional spaces, without having to calculate or even know about anything. That is why we use kernels for logistic regression, where it allows us to work in very high or infinite dimensional spaces.

(3)

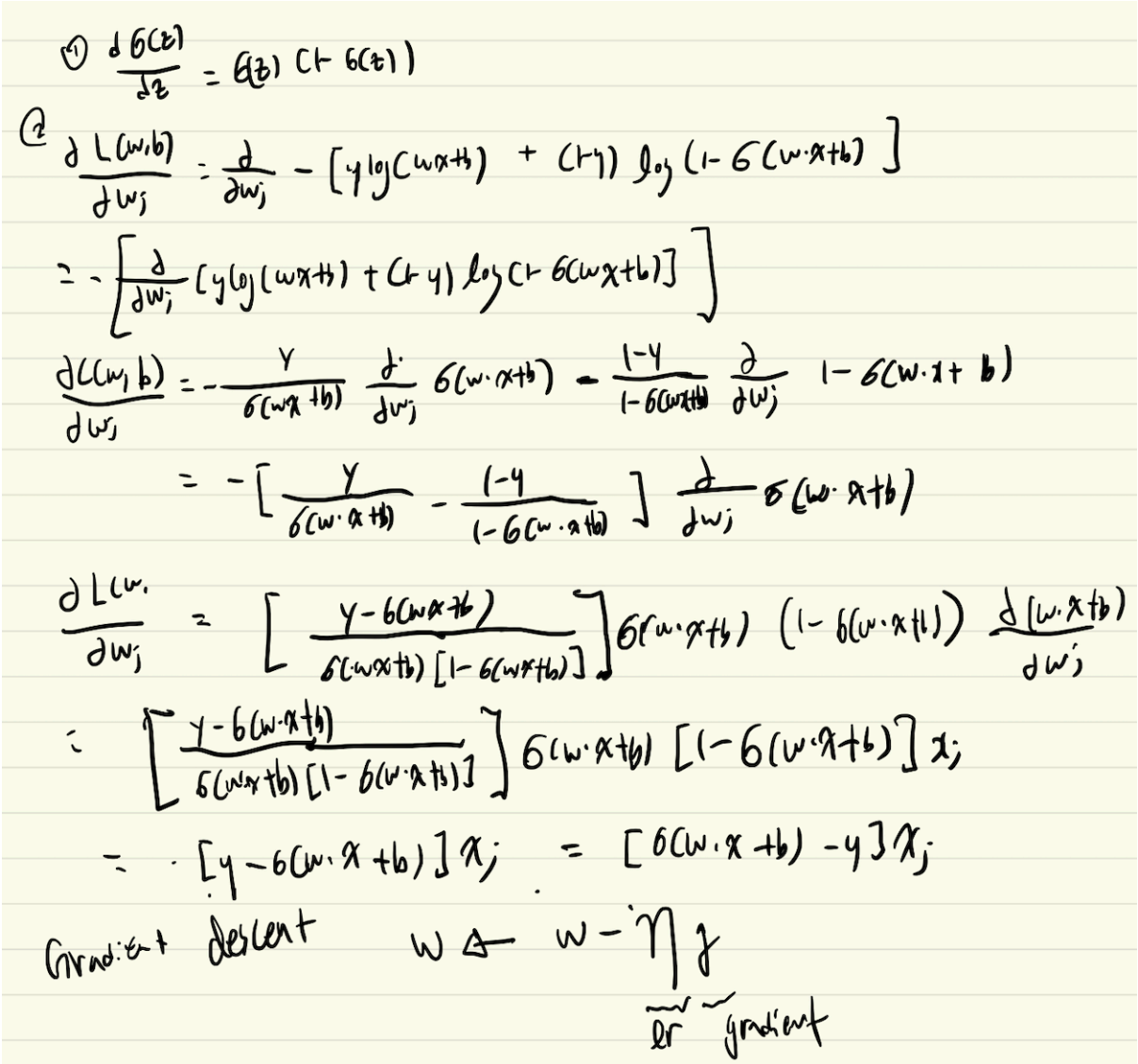
Logistic regression is a widely used technique for classification. It is widely used since it is simple to implement (without hyperparameters, unless we use LASSO or ridge regression) and interpretative and is effective. Plus, unlike some other methods, it outputs a probabilistic interpretation, rather than just giving a dichotomic result, which might be very advantageous is various cases. It also allows people to control overfitting (LASSO, ridge regression) and do subset selection (LASSO). Moreover, we can extend this method easily to multiclass problems. However, it gives us a linear boundary, so it shows poor performance on non-linear data, such as images, or data that requires nonlinear boundaries. (Though this can be solved by kernels or extensions to higher order polynomial functions.) Plus, when the number of observations is lesser than the number of features, logistic regression overfits, leading to bad performance. Moreover, when classes are completely separable, the estimation of parameters become unstable in logistic regression due to the use of logistic function, which then becomes close to a step function and forces the derivatives to be infinite, becoming computationally unstable. Due to low flexibility, it isn't optimal to capture some complex relationships.

Kernel logistic regression, is an extension to the logistic regression making use of the representor function. It was mentioned above that the logistic regression has limitations due to linearity and underperforms when decision boundaries aren't linear. Plus, kernel functions allow us to operate in the original feature space without computing the coordinates of a data in a higher dimensional space outperforming those, using high dimensional vectors for nonlinear boundaries. This is very efficient in ways of computation. However, kernel logistic regression has disadvantages. To begin with, kernels are hard to interpret and is not intuitive. It is hard to explain, which features affect the result the most. Plus, since we are boosting up the dimensions and crushing it down internally, there is no way to deeply analyze the model. Moreover, choosing the right kernel could be a difficult task. There are lots of kernels that we can use such as RBF kernels (used for this project), polynomial kernels, hyperbolic tangent kernels, Laplacian kernels, spline kernels, and so on. Since, data distributions are all different from one another, selecting the appropriate kernel might be a challenging task. Not only that, after deciding the kernel, we should select the well-fitting hyperparameters for the model via cross validation or other methods, which also takes lots of time and effort. Hence, the computational time and complexity is much bigger than that of normal logistic regression. Lastly, overfitting might happen. When we use a complex kernel for the model, such as the RBF kernel, that boosts up a gigantic number of dimensions, it might tend to overfit. This happens for our

model in this project also: training accuracy shows 96% of accuracy, while the test error gives us an accuracy among 67%, which is a huge difference. However, in normal logistic regression training accuracy is about 78% and the test accuracy is about 69%, which indicates that overfitting isn't happening. The error might be the error coming from the bias in the normal ones. In order to solve this problem, we can use a similar method as the LASSO or Ridge regression by introducing a new term that gives penalty to the value of the coefficients. Without these regularization terms, it is prone to face with overfitting.

(4)

To begin with, let's take a look at how the weight update is done for the original logistic regression. Using the loss function that we derived above, we try to train our model using the concept of gradient descent. Here, we compute the predicted y_hat and compute the loss for each of the predictions. Then we 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. After acquiring this gradient value, we go the other way, in other words, we subtract the product of the hyperparameter (the learning rate) and the gradient value from the original weight and get a new weight. We do this, in order to find the minimum point. We iterate this over and over to optimize the model. The following indicates the process and derivation of gradients. This only shows the gradient of one weight and datum. In our code, we use matrix forms, which is no different in terms of meaning.



The weight updating process in kernel logistic regression is very similar to the original one. The only difference is the score function, which leads to a slight difference in the gradient value. The process is written below.

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

② $\frac{\partial L(\alpha,\alpha_0)}{\partial \alpha_j} = \frac{\partial}{\partial \alpha_j} - [y \log \sigma(\alpha_y K(x^{(i)}, x) + \alpha_0) + (1-y)\log(1-\sigma(\alpha_y K(x^{(i)}, x) + \alpha_0))]$

$= -\left[\frac{\partial}{\partial \alpha_j}(y \log \sigma(\alpha_y K(x^{(i)}, x) + \alpha_0) + (1-y)\log(1-\sigma(\alpha_y K(x^{(i)}, x) + \alpha_0))\right]$

$\frac{\partial L(\alpha,\alpha_0)}{\partial \alpha_j} = -\frac{y}{\sigma(\alpha_y K(x^{(i)}, x) + \alpha_0)} \frac{\partial}{\partial \alpha_j} \sigma(\alpha_y K(x^{(i)}, x) + \alpha_0) - \frac{1-y}{1-\sigma(\alpha_y K(x^{(i)}, x) + \alpha_0)} \frac{\partial}{\partial \alpha_j}(1-\sigma(\alpha_y K(x^{(i)}, x) + \alpha_0))$

$= -\left[\frac{y}{\sigma(\alpha_y K(x^{(i)}, x) + \alpha_0)} - \frac{1-y}{1-\sigma(\alpha_y K(x^{(i)}, x) + \alpha_0)}\right] \frac{\partial}{\partial \alpha_j} \sigma(\alpha_y K(x^{(i)}, x) + \alpha_0)$

z

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

$\frac{\partial \sigma(z)}{\partial z} = \frac{-e^{-z}}{(1+e^{-z})^2} = -\frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = -\sigma(z)(1-\sigma(z))$

$\frac{\partial L(\alpha,\alpha_0)}{\partial \alpha_j} = \left[\frac{y-\sigma(z)}{\sigma(z)[1-\sigma(z)]}\right] \sigma(z)(1-\sigma(z)) \frac{\partial(\alpha_y K(x^{(i)}, x) + \alpha_0)}{\partial \alpha_j}$

$= [y - \sigma(z)]\, y_j K(x^{(i)}, x^{(j)})$ (= g)

Gradient descent $w \leftarrow w - \eta g$

learning rate, gradient

Before we move on, let's take a brief look at the RBF Kernels, which might also be an additional information for the question 1. Gaussian RBF kernels are the most widely used kernels in kernel tricks. This can be represented as the following equations. The following also allows us to see a glimpse of the effects of this kernel.

Gaussian RBF Kernel

$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{\sigma^2}\right)$

How can we get this function with $\phi(x)$

Assume $d=1, \sigma=1$.

$K(x_i, x_j) = \exp(-x_i^2 + 2x_i x_j - x_j^2) = \exp(-x_i^2)\exp(2x_i x_j)\exp(-x_j^2)$

Thus, $\phi(x_i) = \exp(-x_i^2) z_i$ with $z_i z_j = \exp(2x_i x_j)$

$\Rightarrow$ z_i is infinite-dimensional!

$\exp(2x_i x_j) = \sum_{k=0}^{\infty} \frac{2^k x_i^k x_j^k}{k!}$

$\Rightarrow \phi(x_i) = \exp(-x_i^2)\left[1 \sqrt{\frac{2^0}{0!}} x_i \sqrt{\frac{2^1}{1!}} x_i^2 \sqrt{\frac{2^2}{2!}} x_i^3 \cdots\right]$

This shows that using the Gaussian kernel we can expand the dimensions infinitely.

(5)

Then, how can we expand this to multiclass problems?
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.

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

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. The w matrix consisted of all weights for classes (each column corresponding to each weight and w_j being the jth column) and the b matrix will contain all the biases.

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

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

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

Then, what is it about multiclass problems for kernel logistic regression? We will use the same softmax function, as we've seen above in order to compute the probability of y being in each class, p(y|x). This input to the softmax function will be different as the score function itself is different. Using the representor theorem stated above, p(y=c|x) will be expressed as the following.

Alpha and y is the matrix with all alphas and y combined, for instance with α_j being the jth column as well as the coefficient of the jth class. Those two terms are multiplied component-wise, rather than inner products. Plus, $\alpha_0$is the matrix(vector) consisted of all bias terms. X is a matrix consisting of training points.

$$p(y = c|x) = \frac{e^{\sum_{i=1}^{N} \alpha_{ic} y_i K(x^{(i)}, x) + \alpha_{0c}}}{\sum_{j=1}^{k} e^{\sum_{i=1}^{N} \alpha_{ij} y_i K(x^{(i)}, x) + \alpha_{0j}}} = \frac{e^{(\alpha_c \circ y)^T K(X, x) + \alpha_{0c}}}{\sum_{j=1}^{k} e^{(\alpha_j \circ y)^T K(X, x) + \alpha_{0j}}}$$

Based on this we can compute the loss function, which will have a similar looking structure as the loss function we derived for multiclass problems.

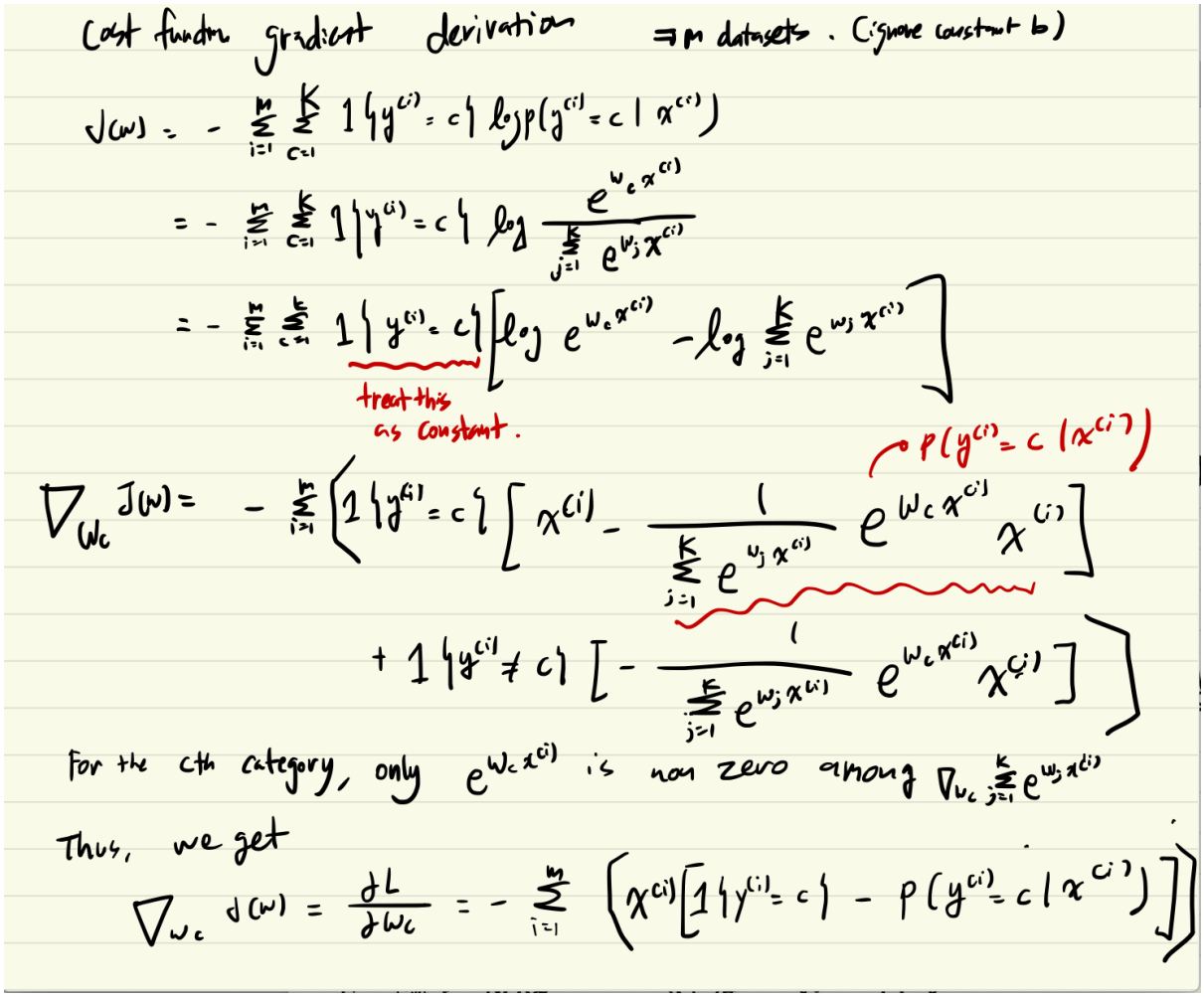
$$L(y, \hat{y}) = -\log\left(Y, \hat{Y}\right) = -\sum_{c=1}^{K} 1\{y = c\} \log p(y = c|x) = -\sum_{c=1}^{K} 1\{y = c\} log \frac{e^{(\alpha_c \circ y)^T K(X, x) + \alpha_{0c}}}{\sum_{j=1}^{k} e^{(\alpha_j \circ y)^T K(X, x) + \alpha_{0j}}},$$

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

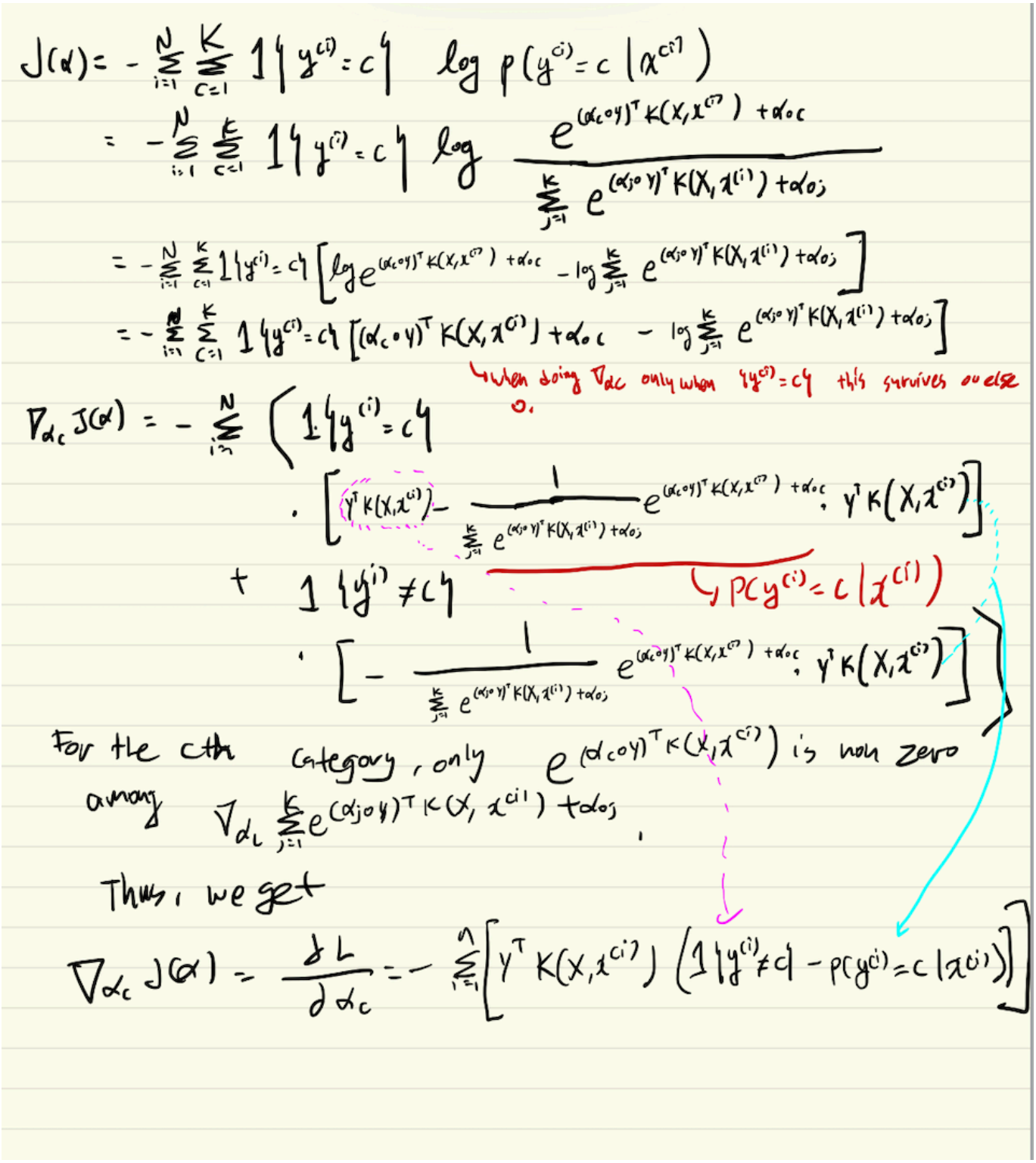
(6)
Now, based on this loss function, we should train the model using the gradient descent. Then, what is gradient descent? We try to use the loss function's slope/gradient to compute the smallest point of the function. Starting at a random point on a function, if we move the point in the direction of 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. The following is the derivation process of the gradient.



For Multiclass Kernel regression, we can follow the same steps, which is depicted in the following figure.



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

(7)

```
import numpy as np
import pandas as pd
import math
class LogisticRegressor():
```

```
    #INITIALIZATION OF THE MODEL
    def __init__(self,w=None):
        #First initialize the weight to a none value
        self.w = w
    def fit(self, X, y,lr,epoch_num):
        """
        TRAINING
        X dimension is (623,11)
        X dimension is (N,D) => We will change this to (N,D+1) adding the bias term
        w dimension is (D+1,1)
        y dimension is (N,1) so the dimension is (623,1)
        model.fit(train_X, train_y,lr,epoch_num) will be called.
        """
        #ADDING THE BIAS TERM
        #matrix consisted of ones so that we could add it to X
        ones = np.ones((X.shape[0],1))
        #Add the matrix of ones to the X so that we have X E R (N*D+1)
        X = np.concatenate((ones, X),axis=1)

        #Initialization of the weights to zero.
        self. w= np.zeros(np.shape(X[1]))
        #TRAINING PROCESS (TRAINS "epoch_num" times)
        #epoch_num is a hyperparameter
        for i in range(epoch_num):
            #y_hat = The probability value that we got from the sigmoid function
            y_hat = self.sigmoid(X)
            #Getting the gradient
            #FYI: Here the gradient is actual gradient * (-1)
            #The real gradient is -X.T@(y_hat-y) but just to make it pretty, just
left it like that
            gradient = X.T @(y_hat-y)

            #Perform the gradient descent
            #since the gradient variable is has opposite sign of the real gradient-
> add it
            self.w += gradient *lr

            #Printing the training error every 100 iterations.
            if(i%100==0):
                self.accuracy(y,y_hat)



    def predict(self, X):
        """
        Here we predict the probability value for the test data.
        Input dimension (N,d)
        Output dimension (N,1)
        """
        #ADDING THE BIAS TERM
        #matrix consisted of ones so that we could add it to X
```

```
        ones = np.ones((X.shape[0],1))

        #Add the matrix of ones to the X so that we have X E R (N*D+1)
        X = np.concatenate((ones, X),axis=1)

        #Call the sigmoid function to get the probability value and return it to
the caller.
        return self.sigmoid(X)

    def sigmoid(self,X):
        """
        SIGMOID FUNCTION (MORE OF A COMPUTING SCORE +SIGMOID)
        FIRST using the given X, we compute the score.
        Then we put this in the sigmoid function 1/1+e^z and return it
        input dimension is (N,d+1)
        z/output dimension is (N,1)
        """
        #X:(N,d+1), self.w: (d+1,1), z:(N,1)
        #Computing the score value. z = XW (bias term included)
        z = X@ self.w

        #returning the sigmoid function value, which is the probability value.
        #We put the score function value into the sigmoid function.
        return 1/(1+np.exp(z))


    def accuracy(self,true_y,pred_y):
        '''
        Computing the accuracy by comparing the predicted y and the true y value
        When we get the probability value from the sigmoid function,
        we check if it is bigger than 0.5-> if yes we put that to class 1
        if not to class 0
        '''
        #If the predicted probability is smaller than 0.5 -> class 0
        pred_y[pred_y<0.5]=0

        #If the predicted probability is bigger than 0.5 -> class 1
        pred_y[pred_y>=0.5]=1

        #Check how many true y and predicted y are equal among the total
datapoints.
        accuracy=np.sum(true_y==pred_y)/len(true_y)*100

        #PRINTING The accuracy
        print('Training Accuracy: ', accuracy,"%" )
```

(8)

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

```
class LogisticRegressor():
    #INITIALIZATION OF THE MODEL
    def __init__(self,w=None, y=None):
        #First initialize the weight to a none value
        self.w = w
        #self.Y is the value of training y. This will be used in gradients + score
functions
        #Initialize to zero.
        self.Y=y
    def fit(self, X, y,lr,epoch_num):
        """
        TRAINING (623,623)    ->(623,624) By including the bias term
                (N,N)          -> (N,N+1)
        y dimension is (N,1) so the dimension is (623,1)
        self.w -> (N+1,1)
        model.fit(train_X, train_y,lr,epoch_num) will be called.
        """
        #Here, we save the y value that is used for training
        self.Y =y

        #ADDING THE BIAS TERM
        #matrix consisted of ones so that we could add it to X
        ones = np.ones((X.shape[0],1))

        #Add the matrix of ones to the X so that we have X E R (N*N+1)
        X = np.concatenate((ones, X),axis=1)

        #initialize the weights to one and so that it has the dimension,  (N+1,1)
        self. w= np.ones(np.shape(X[1]))

        #Temporarily add one element of one, to use it in the gradient
        #       => Should match the dimension
        #I inserted 1, since, I should make the bias term valid.
        Y = np.insert(self.Y,0,1)

        #TRAINING PROCESS (TRAINS "epoch_num" times)
        #epoch_num is a hyperparameter
        for i in range(epoch_num):
            #y_hat = The probability value that we got from the sigmoid_changed
function
            y_hat = self.sigmoid_changed(X)

            #Getting the gradient
            #FYI: Here the gradient is actual gradient * (-1)
            #The real gradient is -Y*X.T@(y_hat-y) but just to make it pretty, just
left it like that
            #Plus, multiplying Y is not essential since, we multiply Y in the
changed sigmoid function.
            #Just did to make it sure.
            gradient = Y*(X.T @(y_hat-y))
```

```
            #Perform the gradient descent
            #since the gradient variable is has opposite sign of the real gradient-
> add it
            self.w += gradient *lr
            #Printing the training error every 100 iterations.
            if(i%100==0):
                self.accuracy(y,y_hat)

    def predict(self, X):
        print(self.w.shape)
        """
        Input dimension (N,N)
        Output dimension (N,1)
        """
        #ADDING THE BIAS TERM
        #matrix consisted of ones so that we could add it to X
        ones = np.ones((X.shape[0],1))
         #Add the matrix of ones to the X so that we have X E R (N*D+1)
        X = np.concatenate((ones, X),axis=1)
        #Call the sigmoid_changed function to get the probability value and return
it to the caller.
        return self.sigmoid_changed(X)

    def sigmoid_changed(self,X):
        """
        SIGMOID_CHANGED FUNCTION (MORE OF A COMPUTING SCORE +SIGMOID_CHANGED)
        FIRST using the given X, we compute the score.
        Then we put this in the sigmoid function 1/1+e^z and return it
        input dimension is (N,N+1)
        z/output dimension is (N,1)
        """
        #Temporarily add one element of one, to use it in the gradient
        #       => Should match the dimension
        #I inserted 1, since, I should make the bias term valid.
        Y = np.insert(self.Y,0,1)
        #Calculating the score function
        t = self.w*Y
        z = X@t
        #returning the sigmoid function value, which is the probability value.
        #We put the score function value into the sigmoid function.
        return 1/(1+np.exp(z))

"""(HEAD) UNUSED IN THIS TASK"""
    def sigmoid(self,X):
        """
        input dimension is (N,d)
        z/output dimension is (N,1)
        """
       # print("IAM",X.shape)
        z = X@ self.w
        return 1/(1+np.exp(z))
```

```
"""UNUSED IN THIS TASK(TAIL)"""

    '''
        Computing the accuracy by comparing the predicted y and the true y value
        When we get the probability value from the sigmoid_changed function,
        we check if it is bigger than 0.5-> if yes we put that to class 1
        if not to class 0
    '''
    def accuracy(self,true_y,pred_y):
        #If the predicted probability is smaller than 0.5 -> class 0
        pred_y[pred_y<0.5]=0
        #If the predicted probability is bigger than 0.5 -> class 1
        pred_y[pred_y>=0.5]=1
        #Check how many true y and predicted y are equal among the total
datapoints.
        accuracy=np.sum(true_y==pred_y)/len(true_y)*100
        #PRINTING The accuracy
        print('Training Accuracy: ', accuracy,"%" )
```

main.py

```
if __name__ == "__main__":
    #Getting the train/test data
    train_X, train_y, test_X, test_y = getData()
    #Constructing the RBF kernel consisting of parameters of all one.
    #No regularization used in this model.
    kernel = RBF(np.ones(train_X.shape[-1]+1))
    #The input of all inner products between training points via Kernel.
    K = kernel(train_X, train_X)
    # Fit
    #Model for logistic regression
    model = LogisticRegressor()
    #learning rate & epochs initializaiton
    lr=0.005
    epoch_num=5000
    # Training (Learning) the model with the input derived from above
    # Put three inputs : matrix of all inner products between training points using ke
rnel
    #                   ,lr, epoch
    model.fit(K, train_y,lr,epoch_num)
    #For testing, inner products between test and training points, with dimensions bo
osted up
    #with the kernel function.
    K2 = kernel(test_X,train_X)
    # Predicting the probability of (class being in class 1)each data points
    pred_y = model.predict(K2)
    # Evaluating the accuracy
    accuracy(test_y, pred_y)
```