Machine Learning
Home Assignment 1

Zelin Li(qzj475)

1 Make Your Own

1. For instance, the profile information can be 5 other courses' grade (as positive real number $\mathbb{R}$) of the students who finished the Machine Learning course, and their study program (different program can be marked as different natural number). In this case the sample space $\mathcal{X} = \mathbb{R}^5 \times \mathbb{N}$.

2. The label space $\mathcal{Y}$ would be final grade (as positive real number $\mathbb{R}$) of the students in the Machine Learning course; therefore $\mathcal{Y} = \mathbb{R}$.

3. Since $\mathcal{Y} = \mathbb{R}$, this is a regression problem; the loss function can be defined as square loss: $\ell(Y', Y) = (Y' - Y)^2$.

4. Use a testing set that not include in the sample space to evaluate the performance (through average loss). Each element in the testing set have a real label Y_i; using the input X_i to compute Y_i', then calculate the average loss, that is: $\frac{\sum_{i=1}^{n}(Y_i'-Y_i)^2}{n}$, where n is the volume of the testing set.

5. Over-fitting and sampling bias are issues. So, using an independent and identically distributed large sample for training is significant.

2 Illustration of Markov's and Chebyshev's Inequalities

Implementation can be found in the notebook `Assignment1Question2ZelinLi.ipynb`.

2.a

1. To make 1,000,000 repetitions of the experiment of drawing 20 i.i.d. Bernoulli random variables $X_1, X_2, ..., X_{20}$ with bias $\frac{1}{2}$. Since repeating n times i.i.d. Bernoulli experiment with having probability of an event is p; the total count of occurrence of the event is X, then the distribution of X fits the binomial distribution. So that code can be like this (generating random binomial matrix):

```
import numpy as np
x = np.random.binomial(n=20, p=prob, size=sample_size)
```

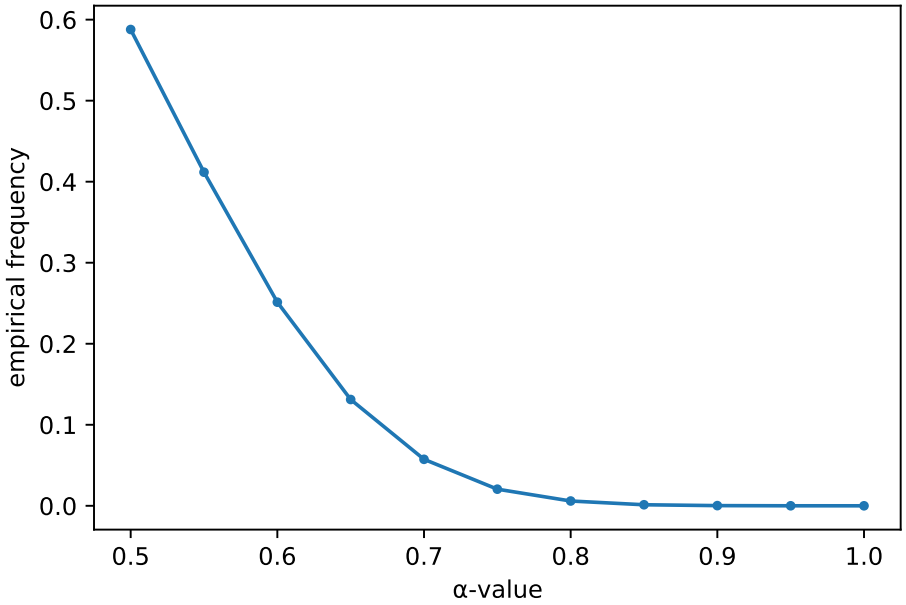

Figure 1: empirical frequency plot

2. The plot can be like Fig.1 (a random experiment result). This is calculate from the count of each average that greater than α divided by 1000000.

```
step_num=int(round((1-prob)/0.05,1)+1)
#prob is equal to 0.5 here, this is to count how many 0.05 steps it needs
    from bias probability to reach 1.0
count=np.array(np.arange(step_num))
x = x/20
for o in range(step_num):
    count[o]=sum(x>=round(prob+0.05*o,2)) #this is the count of average
   that greater than alpha
for i in range(step_num):
    y1.append(count[i]/sample_size) #this is the empirical frequency
```

3. Using granularity of 0.05 is sufficient because we already can get enough count to figure out the curvature of the frequency changes, that is because for each experiment we have 20 i.i.d. variables, to get an average, so we only care about each "step" to form the average, and the minimum "step" is 1/20 (0.05). Also, using smaller granularity like 0.01 will consume much more computing time.

4. The Markov's bound is shown on the Fig.2, this is calculate through the Markov's inequality

$$\mathbb{P}(X \geq \alpha) \leq \frac{E[X]}{\alpha}, \ \alpha = \{0.5, 0.55, ..., 0.95, 1.0\}$$

, for exact code:

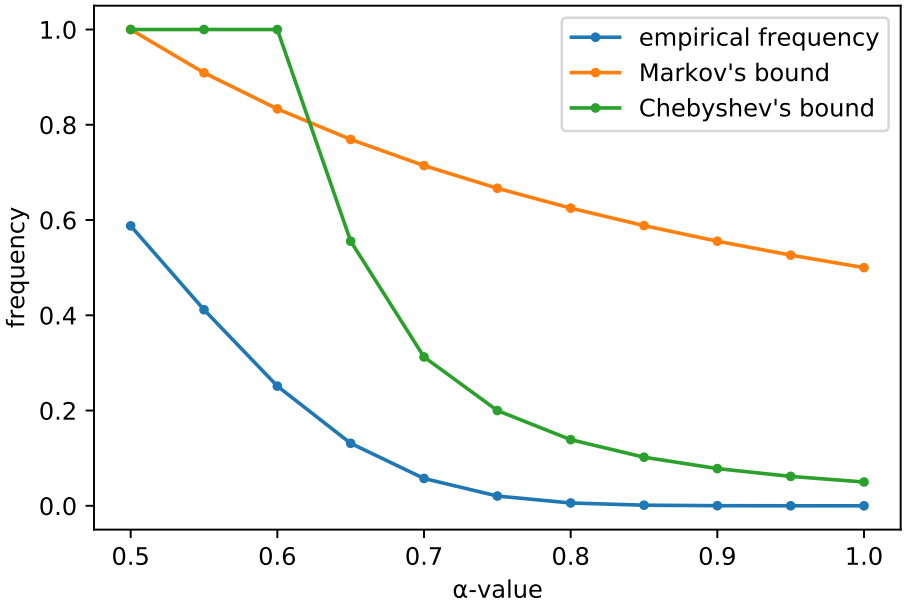

Figure 2: full plot

```
E=prob
#the expectation is equal to the probability in each one Bernoulli
    experiment
for i in range(step_num):
    x.append(prob+0.05*i)
    y2.append(E/(prob+0.05*i)) #this is the Markov's bound
```

5. The Chebyshev's bound is also shown on the Fig.2, this is calculate through the Chebyshev's inequality

$$\mathbb{P}(|X - E[X]| \geq k) \leq \frac{Var[X]}{k^2} = \frac{p(1-p)}{Nk^2}, \ \alpha = \{0.5, 0.55, ..., 0.95, 1.0\}$$

so that if we consider the transformation from α to k, we can easily find out that the left side of bound can rewrite like this

$$\mathbb{P}(X \geq k + E[X])$$

, which is similar to the left side of Markov's inequality, thus $k = \alpha - E[X]$. The exact code:

```
V=prob*(1-prob)/20 #V is Var[X], this from the binomial distribution
for i in range(step_num):
    k=prob+0.05*i-E
```

```
    #this is from Chebyshev's inequality, we need to solve k, so similar
    to Markov's inequality, move the E(X) to the right hand side of the
    equation, we got k=alpha-E(X)
    if(k==0):
        y3.append(1.0)
    elif(pow((1/k),2)*V>=1):
        y3.append(1.0)
    else:
        y3.append(pow((1/k),2)*V) #this is the Chebyshev's bound
```

6. From Fig.2, it's easy to figure out that generally Markov's bound is more loosely than Chebyshev's bound, especially for cases that close to 1.0. But in specific cases like 0.5,0.55 and 0.6, the Chebyshev's bound is greater than 0; larger than Markov's bound.
7. Since we have more than one experiment, the probability is binomial distributed, so the exact probability:

$$P(\frac{1}{20}\sum_{i=1}^{20} X_i \geq \alpha) = C_{\frac{\alpha}{0.05}}^{20} p^{\frac{\alpha}{0.05}} (1-p)^{20-\frac{\alpha}{0.05}}$$

, we know that p =0.5 and N =20, so that when $\alpha = 0.95$, the exact probability is $20 * (0.5)^{20} = 0.00001907348$; when $\alpha = 1$, the exact probability is $1 * (0.5)^{20} = 0.000000953674316$

2.b

When bias=0.1, it comes out the results which are shown on Fig.3. For question 1-5, the code is same just change some variables.
6. In this plot all the lines are more steep than previous bias=0.5 plot (also might can say the bounds are more "strict", especially for the Chebyshev's bound.), but basically the three lines are at the same relative position.
7. In this case, similar to 2.a.7,

$$P(\frac{1}{20}\sum_{i=1}^{20} X_i \geq 0.95) = C_{19}^{20}(0.1)^{19}(1-0.1)^1 = 1.8 * 10^{-18}$$

$$P(\frac{1}{20}\sum_{i=1}^{20} X_i \geq 1) = C_{20}^{20}(0.1)^{20}(1-0.1)^0 = 10^{-20}$$

2.c

Generally speaking, the empirical frequency is greatly influenced by the bias, when the bias close to 0.5, the overall distribution and the curve is more ease, on the other hand, when it's close to 0 or 1, the curve will be more steep. Also, the Markov's bound is less strict than Chebyshev's bound; Chebyshev's bound can be greater than 1.0 in some cases.

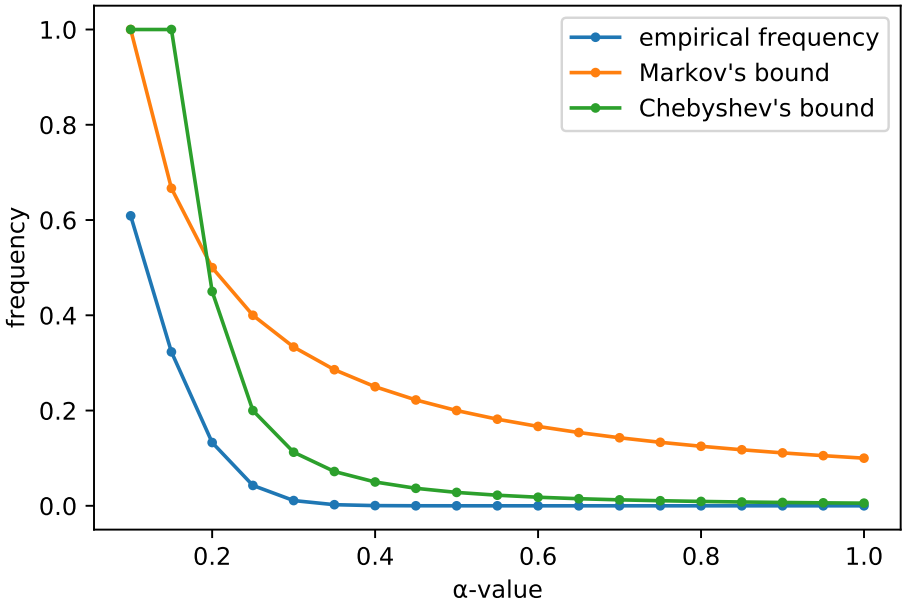

Figure 3: full plot while bias=0.1

3 Tightness of Markov's Inequality

Since we want to prove

$$\mathbb{P}(X \geq \varepsilon^*) = \frac{E[X]}{\varepsilon^*}$$

, and ε^* is fixed.

Assume X can be randomly assigned as ε^* or 0; This is a binomial distribution and its $n = 1$ (Bernoulli distribution), and in this distribution $\mathbb{P}(X > \varepsilon^*) = 0$, so,

$$\mathbb{P}(X \geq \varepsilon^*) = \mathbb{P}(X > \varepsilon^* \vee X = \varepsilon^*) = \mathbb{P}(X > \varepsilon^*) + \mathbb{P}(X = \varepsilon^*) = \mathbb{P}(X = \varepsilon^*)$$

, Bernoulli distribution has a property that is $E[X] = n * \mathbb{P}(X = n)$, so that

$$\mathbb{P}(X \geq \varepsilon^*) = \mathbb{P}(X = \varepsilon^*) = \frac{E[X]}{\varepsilon^*}$$

, thus the original equation is proved (in this equation we have example like: $\mathbb{P}(X = \varepsilon^*) = \frac{1}{\varepsilon^{*2}}$).

4 Digits Classification with Nearest Neighbors

Notice that:

Sorry, I have misunderstood the meanings, I already implement the multiclass classification but not the binary classification. These take great amount of computation time. And it's seem that not enough time remain, So I haven't change the answer of Question 4. The answer below just shown the correct rate of multiclass classification of two digits. Which is:

$$correct\ rate:\ R_{AB} = 1 - freq(digit\ A\ error) - freq(digit\ B\ error)$$

In the three example below, I just using the whole training data set for training, but use the selected (select two digits) testing or validation data set for testing or validation.

I implemented the K-NN using the Euclidean distance, My implementation can be found in the notebook `Assignment1Question4ZelinLi.ipynb`.

The core code is:

```
def knn(k,ts_x,tr_x,label):
    squre_dist=np.square(ts_x-tr_x)
    #this is square Euclidean distance
    total_dist=np.sqrt(squre_dist.sum(axis=1))
    #this is the total distance (add up all 784 units) from test element
   to each training element
    sort_dist=total_dist.argsort(axis=0)
    #sort the total distance from nearest to farest
    tr_lb_stats={}
    for i in range(k):
        lb=label[sort_dist[i]]
        #find the element label from the training label set
        tr_lb_stats[lb]=tr_lb_stats.get(lb,0)+1
        #add count to the label that have been found in k distance, store
    the count just in position "lb"
    ts_predict_lb=sorted(tr_lb_stats.items(),key=lambda d:d[1],reverse=
   True) #find the label that with highest count
    return ts_predict_lb[0][0]
```

For validation, I first separate the test data and make each kind of graph forming a segmentation data of their own, I use code like this:

```
def seg(dt,lb,char):
    seg_dt=[]
    for i in range(lb.shape[0]):
        if(lb[i]==char): #find the character label (or called number)
            seg_dt.append(dt[i])
    seg_dt=np.array(seg_dt)
    return seg_dt
```

Then I can do the validation and testing through calculate the mean of error rate (one minus it will get the correct rate), here I put the code for "Correct rate for digits "5" and "6" in multiclass classification" code as example, other two are pretty much same with it.

4.1 Correct rate for digits "5" and "6" in multiclass classification

First, count the testing correct rate, the calculation is: when it encounter a label that's not the same as test label, count as an error, the final correct rate equal to 1 minus the mean of "5" and "6" error rate (separately count), notice that k have 16 value, so there is a loop for counting each k correct rate:

```
ts5=seg(ts_dt,ts_lb,5)
ts5_lb=5
ts6=seg(ts_dt,ts_lb,6)
ts6_lb=6
ts56=np.vstack((ts5,ts6))
rate56=[]
for K in range(1,34,2): #k range from 1 to 33
    lb=[]
    for i in range(ts56.shape[0]):
        lb.append(knn(K,ts56[i],tr_dt,tr_lb)) #predicted label calculated
    from knn
    lb=np.array(lb) #change predicted label to array, so that it can
   compare with test set label
    rate56.append(1-np.mean(lb[0:ts5.shape[0]]!=ts5_lb)-np.mean(lb[ts5.
   shape[0]:ts56.shape[0]]!=ts6_lb)) #when label isn't same as test
   label, count the error rate, correct rate=1-error rate
```

Then, calculate the validation correct rate (using the first 8000 as training set and use the last 2000 as validation set):

```
ts5=seg(tr_dt[8000:10000],tr_lb[8000:10000],5)
ts6=seg(tr_dt[8000:10000],tr_lb[8000:10000],6)
ts56=np.vstack((ts5,ts6))
valid_rate56=[]
for K in range(1,34,2): #k range from 1 to 33
    lb=[]
    for i in range(ts56.shape[0]):
        lb.append(knn(K,ts56[i],tr_dt[0:8000],tr_lb[0:8000]))
    lb=np.array(lb)
    valid_rate56.append(1-np.mean(lb[0:ts5.shape[0]]==ts6_lb)-np.mean(lb[
   ts5.shape[0]:ts56.shape[0]]==ts5_lb))
    print(valid_rate56)
```

The final plot is Fig.4, which shows the correct rate of digits 5 and 6 in multiclass classification results.

4.2 Correct rate for digits "0" and "8" in multiclass classification

Fig.5 shows the correct rate of digits 0 and 8 in multiclass classification results.

4.3 Correct rate for digits "0" and "1" in multiclass classification

Fig.6 shows the correct rate of digits 0 and 1 in multiclass classification results.

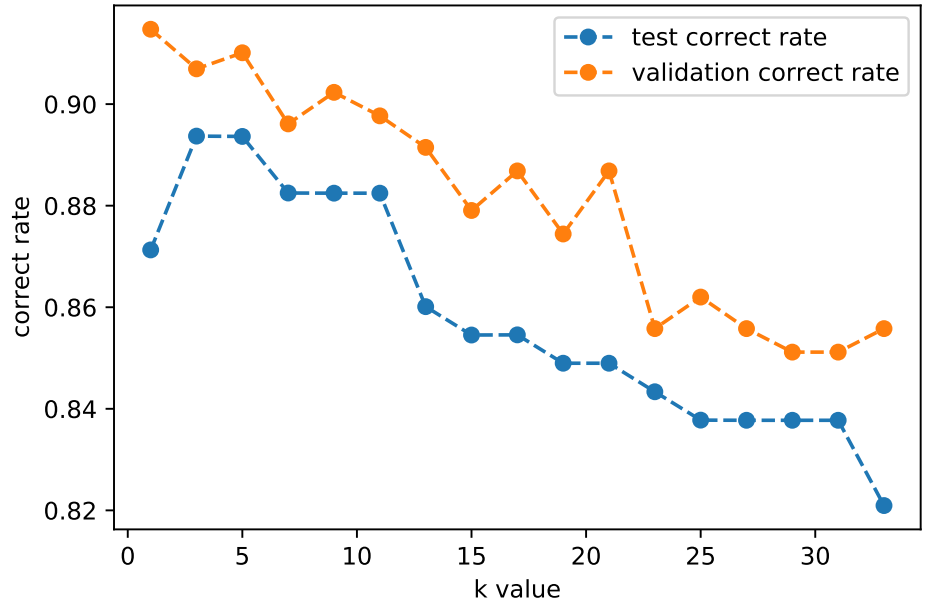

Figure 4: This figure shows the correct rate of digits 5 and 6 in total 10 digits classification results

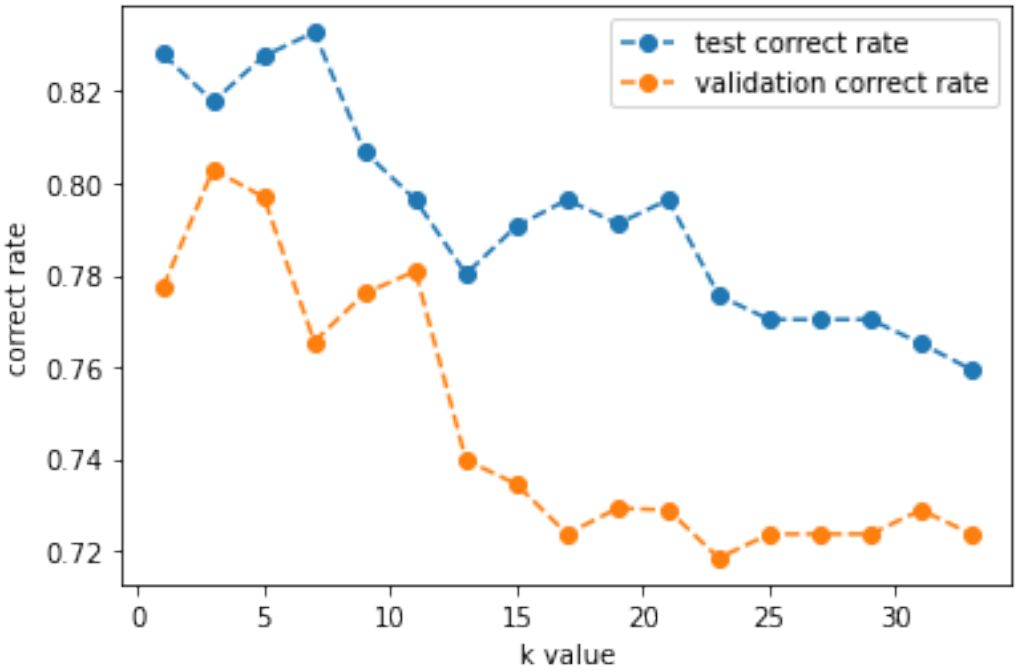

Figure 5: This figure shows the correct rate of digits 0 and 8 in total 10 digits classification results

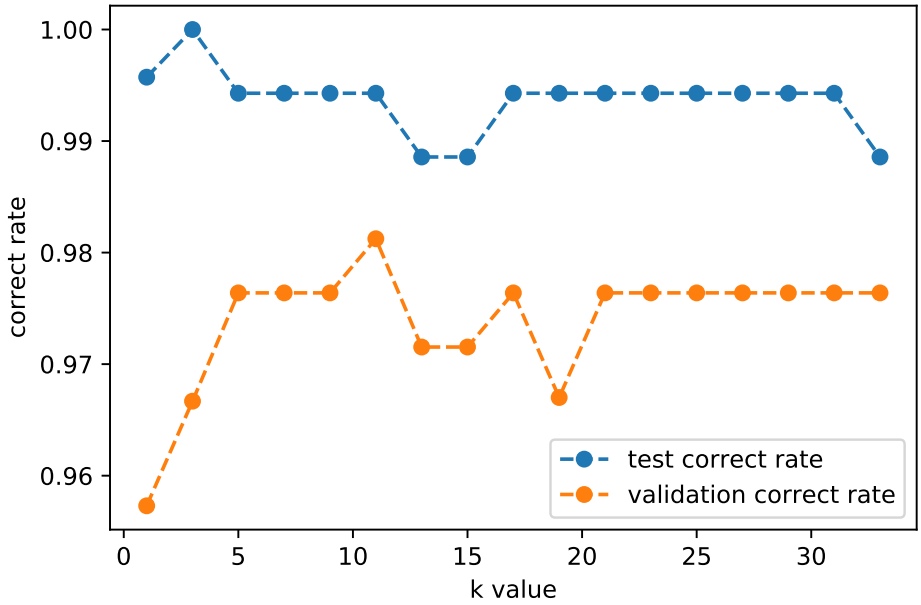

Figure 6: This figure shows the correct rate of digits 0 and 1 in total 10 digits classification results

4.4 Discussion

In conclusion, the error rate for digits 0 and 8 is high, but for digits 0 and 1 are not that high, which shows that it is digits 8 error rate that influenced the sum error rate of 0 and 8. Apparently, digits 8 have much similarity with multiple digits such as 0,9,6 even 4 or 5. On the other hand, like digits 1, it is pretty hard for an error because it has the fewest number of strokes.

Also, From observing the correct rate and k value gradient, from all of three plot it is obvious that correct rate will goes down along as the increasing k, but it is not always the case. Generally the highest correct rate happened when k is 3 or 5.

5 Nearest Neighbors for Multiclass Classification

Sorry, in Question 4, I have misunderstood the meanings, I already implement the multiclass classification but not the binary classification, These take great amount of computation time. And now it's seem that not enough time remain, So I haven't change the answer of Question 4.

So, instead of answering how to make a multiclass classification I would like to answer how to change my multiclass classification to a binary classification.

Instead of using a training set having all the digits (or numbers, characters) data, simply

select only two kinds of digits element from original training set as new training, testing and validation sets.

6 Linear regression

6.1 Implementation of linear regression

I implemented linear regression using the pseudoinverse as introduced in the lecture. My implementation can be found in the notebook `Assignment1Question6ZelinLi.ipynb`.

I computed the parameters of a regression model in the following way:

```
for i in range(x.shape[0]):c[i]=1
c=np.array(np.arange(x.shape[0]))
x1 = np.c_[x,c] #add a column of 1 in matrix x after its last column
def reg_func(x,y):
    w=np.dot(np.dot(np.linalg.pinv(np.dot(x.T,x)),x.T),y)
    #this is the formula: w=(x^Tx)^(-1)x^Ty
    return w
w=reg_func(x1,y)
predict=np.dot(x1,w.T) #this is the regression prediction result of y
```

The key formula is $w = (x^T x)^{-1} x^T y$, which is corresponded to the 5th line of the code above, after that, the regression prediction of y is calculated as $y' = wx$, here w is a matrix; its first column refer to a and second column is refer to b (a and b is refer to the constant in $y' = ax + b$). In order to get w, martix x (only one column) has to add a column full of 1, which formed $x1$. Only $x1$ can participate in the matrix multiplication, and helping calculate the "b" part of the w.

The results are shown on Fig.7.

6.2 Building the first model

Fig.7 shows the loaded data. To fit a model of the form

$$h(x) = \exp(ax + b)$$

, with parameters $a, b \in \mathbb{R}$, I transformed the output data (the y values) by applying the natural logarithm first:

```
y2=np.log(y)
w2=reg_func(x1,y2)
predict2=np.dot(x1,w2.T)
```

The I used the algorithm described in Section 6.1 to model $\ln y$ by $ax + b$. I found $a = 1.55777052$ and $b = -1.45194395$.

Now, the new logarithm model I found $a = 0.25912824$ and $b = 0.03147247$. Fig.8 shows the logarithm plot.

Calculate the mean squared error using this code:

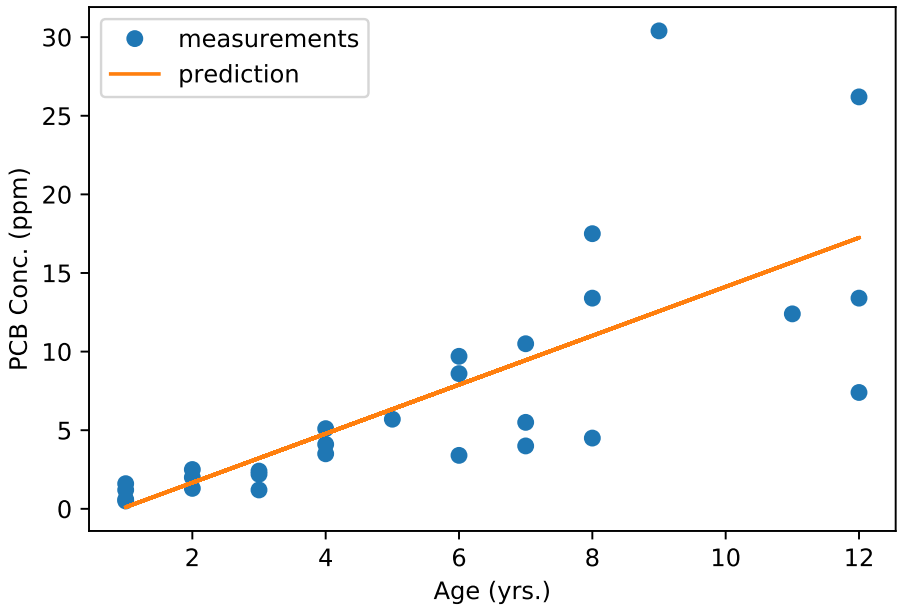

Figure 7: This figure shows the PCB concentration vs. the age of the fish in years

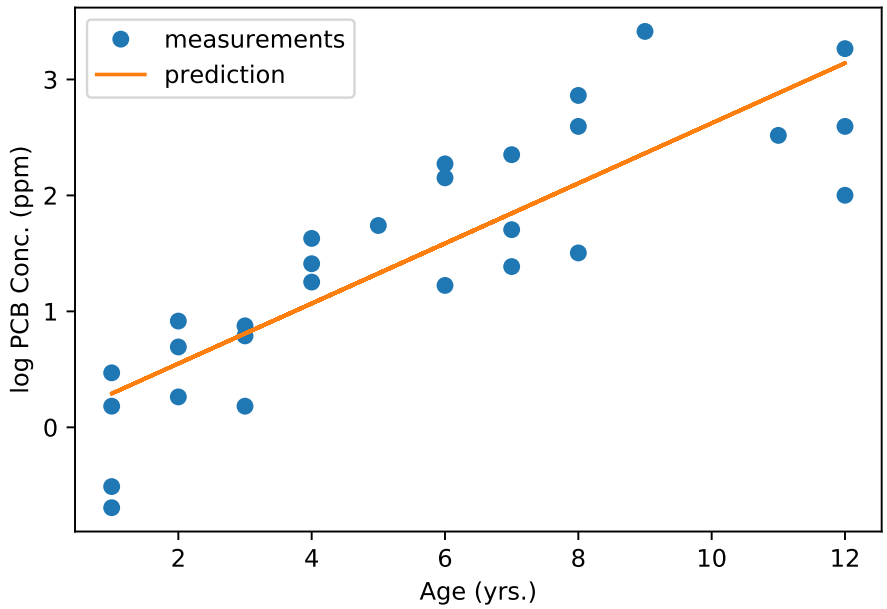

Figure 8: This figure shows the log PCB concentration vs. the age of the fish in years

```
def mean_error(a,predict_a):
    err = (np.square(a - predict_a)).mean(axis=0)
    return err
```

I got the mean squared error of original is 24.801064316570585, while the logarithm is 0.29853492489386174, which is much smaller than the original.

6.3 Plot the data and the regression line

The plot can be found at Fig.7 and Fig.8.

6.4 Compute the coefficient of determination

I calculate the R^2 using this code:

```
def R_func(a,predict_a):
    R=1-((np.square(a - predict_a)).sum(axis=0)/(np.square(a - np.mean(a)
   )).sum(axis=0))
    return R
```

I got R^2 of original is 0.5422279075635534, while the logarithm is 0.7313915439623446.

The larger the R^2, the better the regression function is; when $R^2 = 1$ it means the data are totally fit the regression function, if $R^2 = 0$, it means that the regression function has a error rate that is equal to $y = \overline{y}$. R^2 can be negative, in this case the regression function has a error rate that is larger than $y = \overline{y}$, which means the regression function really bad.

6.5 Build a non-linear model

To fit a model of the form

$$h(x) = \exp(a\sqrt{x} + b)$$

, with parameters $a, b \in \mathbb{R}$, First I applying the non-linear input transformation that is from x to $\sqrt{x}$; also I transformed the output data like Section 6.2, the transformation code:

```
x3=np.sqrt(x1)
y3=np.log(y)
w3=reg_func(x3,y3)
predict3=np.dot(x3,w3.T)
```

This model I found $a = 1.1986063$ and $b = -1.19475082$. Fig.9 shows the plot.

As for the mean squared error, it is 0.2377249626004808, while the R^2 is 0.7861056451320498. The mean squared error is lower than previous two regression function and the R^2 is higher than previous two regression function, which means this model is better.

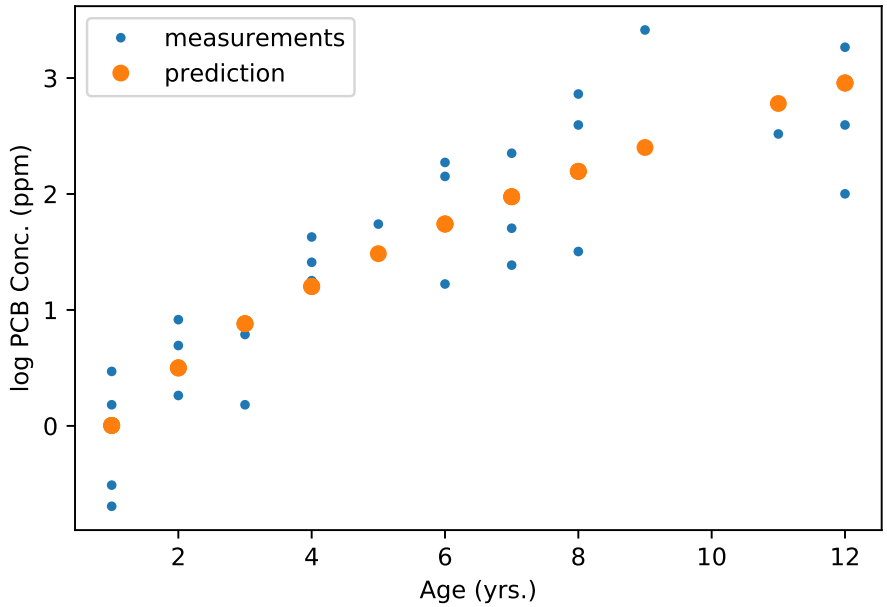

Figure 9: This figure shows the log PCB concentration vs. the age of the fish in years (x transformed to $\sqrt{x}$)