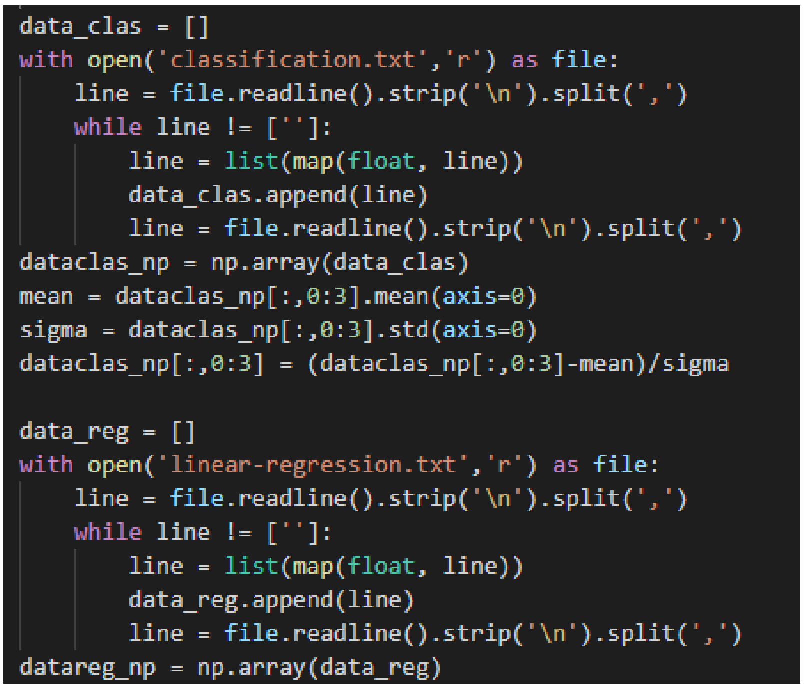
Programming Assignment 4: Classification and Regression

1. Group member
   a) Hao Yang, USCID: 5284267375. (Contributions: All)
2. Part 1: Implementation
   1. Programming language: Python 3.7
   2. Libraries used: random, numpy, matplotlib.pyplot, mpl_toolkits.mplot3d, copy
   3. Algorithm: Perceptron learning algorithm (and with pocket method), Logistic Regression, Linear Regression.
   4. Script description:
      1. Data processing: read txt file line by line, and save as list. Totally 2000 data points in classification.txt and 3000 data points in linear-regression.txt. Then do normalization and standardization:

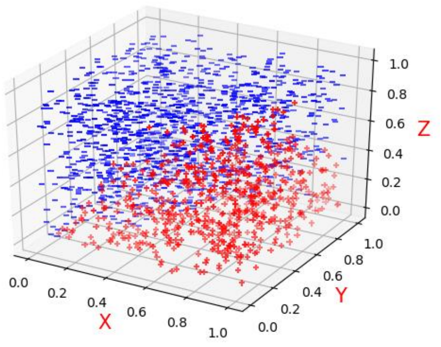


```
data_clas = []
with open('classification.txt','r') as file:
    line = file.readline().strip('\n').split(',')
    while line != ['']:
        line = list(map(float, line))
        data_clas.append(line)
        line = file.readline().strip('\n').split(',')
dataclas_np = np.array(data_clas)
mean = dataclas_np[:,0:3].mean(axis=0)
sigma = dataclas_np[:,0:3].std(axis=0)
dataclas_np[:,0:3] = (dataclas_np[:,0:3]-mean)/sigma

data_reg = []
with open('linear-regression.txt','r') as file:
    line = file.readline().strip('\n').split(',')
    while line != ['']:
        line = list(map(float, line))
        data_reg.append(line)
        line = file.readline().strip('\n').split(',')
datareg_np = np.array(data_reg)
```

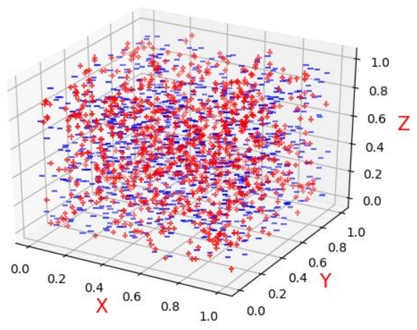
Data points in 3D space:

Points of classification.txt:

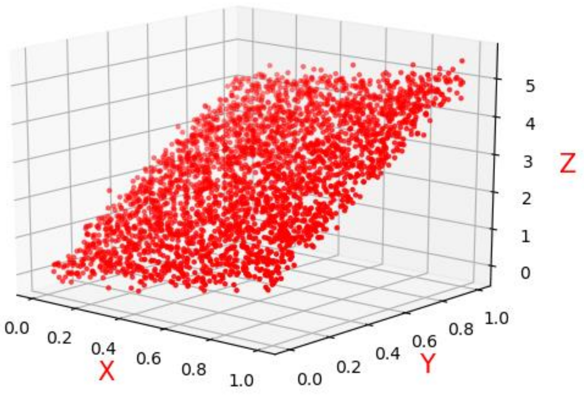
Column 4:



Column 5:

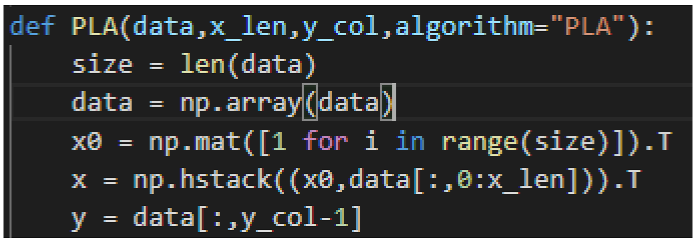


Points of linear-regression.txt:



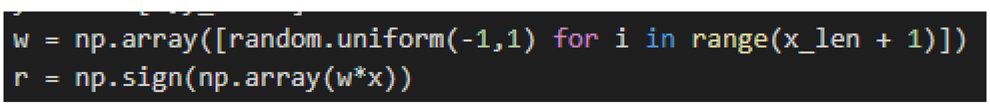
2. Implementation of Perceptron Learning Algorithm (PLA) and pocket:

Step 0: Data preprocessing, extend the dimension from x_1~x_n to x_0~x_n.

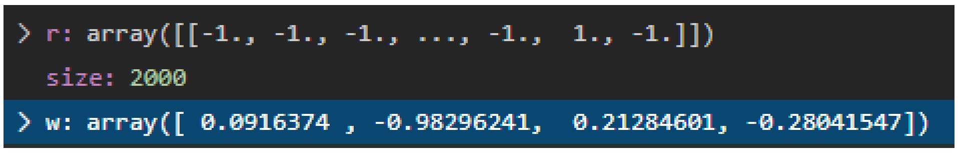


```
def PLA(data,x_len,y_col,algorithm="PLA"):
    size = len(data)
    data = np.array(data)
    x0 = np.mat([1 for i in range(size)]).T
    x = np.hstack((x0,data[:,0:x_len])).T
    y = data[:,y_col-1]
```

Step 1: Initialize the weight vector to a random small value in [-1,1]. Then calculate the prediction value of y as r:

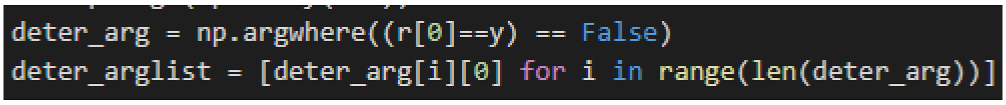


```
w = np.array([random.uniform(-1,1) for i in range(x_len + 1)])
r = np.sign(np.array(w*x))
```



```
> r: array([[-1., -1., -1., ..., -1.,  1., -1.]])
  size: 2000
> w: array([ 0.0916374 , -0.98296241,  0.21284601, -0.28041547])
```

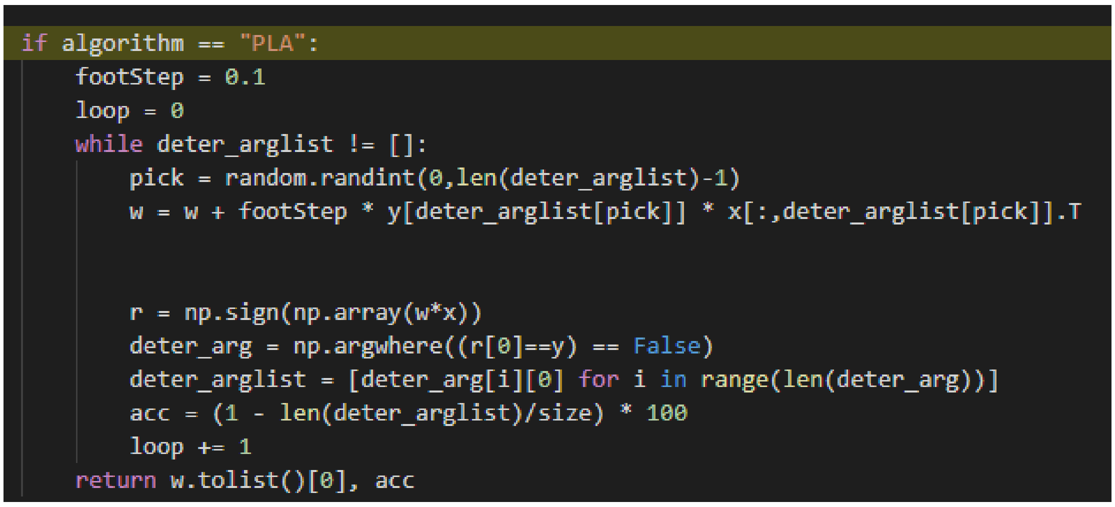
Get the mistake list of initial weight and their index:



```
deter_arg = np.argwhere((r[0]==y) == False)
deter_arglist = [deter_arg[i][0] for i in range(len(deter_arg))]
```

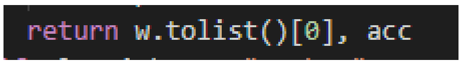
Step 2:

1. When implementing PLA:



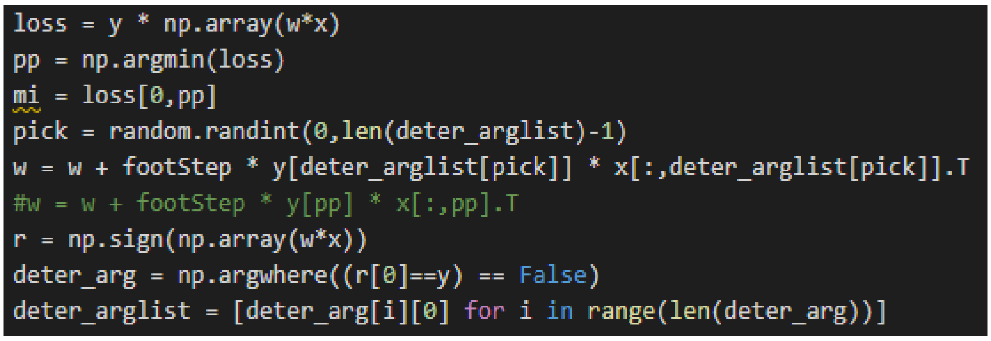
```
if algorithm == "PLA":
    footStep = 0.1
    loop = 0
    while deter_arglist != []:
        pick = random.randint(0,len(deter_arglist)-1)
        w = w + footStep * y[deter_arglist[pick]] * x[:,deter_arglist[pick]].T


        r = np.sign(np.array(w*x))
        deter_arg = np.argwhere((r[0]==y) == False)
        deter_arglist = [deter_arg[i][0] for i in range(len(deter_arg))]
        acc = (1 - len(deter_arglist)/size) * 100
        loop += 1
    return w.tolist()[0], acc
```

Pick a random error point from mistake list to modify weight, then recompute new mistake list with new weight until the mistake list is empty. Return the weight and accuracy rate.



```
return w.tolist()[0], acc
```

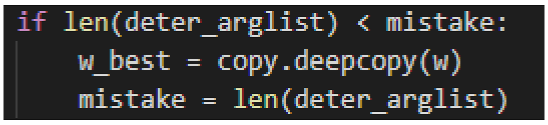
2. When implementing pocket method:



```
loss = y * np.array(w*x)
pp = np.argmin(loss)
mi = loss[0,pp]
pick = random.randint(0,len(deter_arglist)-1)
w = w + footStep * y[deter_arglist[pick]] * x[:,deter_arglist[pick]].T
#w = w + footStep * y[pp] * x[:,pp].T
r = np.sign(np.array(w*x))
deter_arg = np.argwhere((r[0]==y) == False)
deter_arglist = [deter_arg[i][0] for i in range(len(deter_arg))]
```

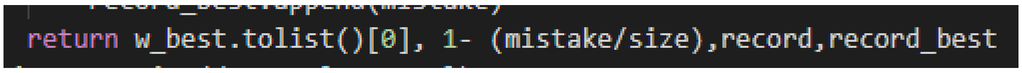
Pick a random error point from mistake list or pick error point with largest loss to modify weight, then recompute new mistake list with new weight until the mistake list is empty or reach 7000 iterations.

Check the length of mistake list in every loop, if the length of list is shorter than the best case we have, record the weight and length of mistake list as the best so far.



```python
if len(deter_arglist) < mistake:
    w_best = copy.deepcopy(w)
    mistake = len(deter_arglist)
```

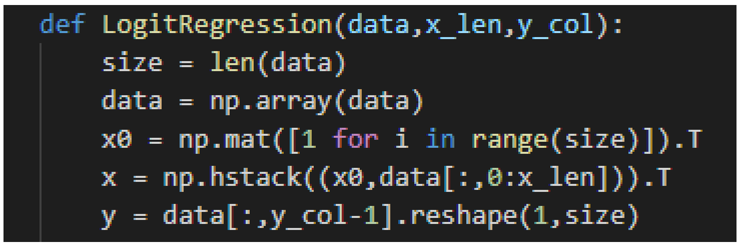
Return the weight, accuracy rate, and the change of mistake number for plotting.



```python
return w_best.tolist()[0], 1- (mistake/size),record,record_best
```

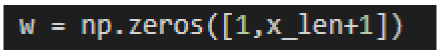
3. Implementation of Logistic Regression:

Step 0: Data preprocessing, extend the dimension from x1~xn to x0~xn.



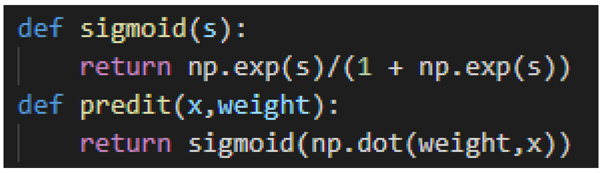
```python
def LogitRegression(data,x_len,y_col):
    size = len(data)
    data = np.array(data)
    x0 = np.mat([1 for i in range(size)]).T
    x = np.hstack((x0,data[:,0:x_len])).T
    y = data[:,y_col-1].reshape(1,size)
```

Step 1: Initialize the weight vector to 0 for all coordinates.

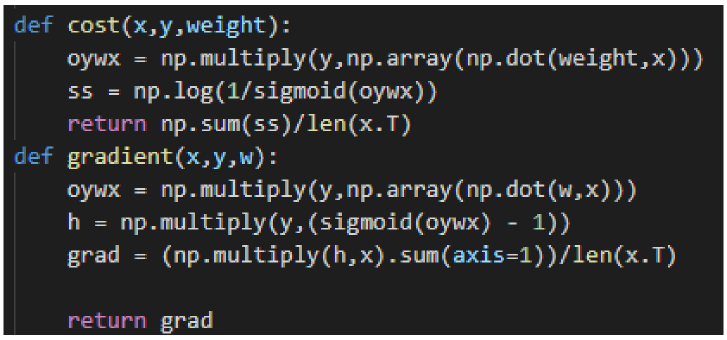


```python
w = np.zeros([1,x_len+1])
```

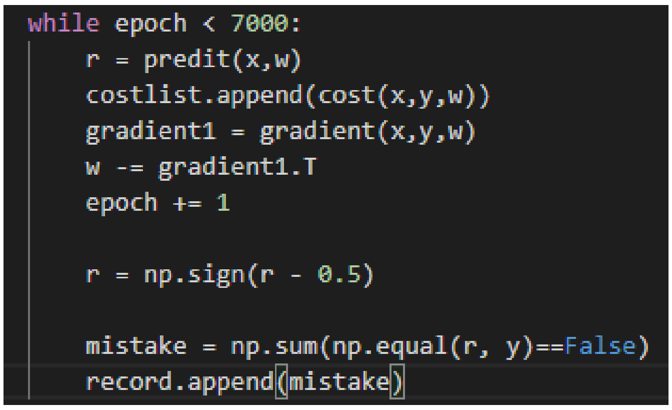
Step 2: For 7000 iterations, compute cost and its gradients to update the weight, and record the mistake number and cost changes:



```python
def sigmoid(s):
    return np.exp(s)/(1 + np.exp(s))
def predit(x,weight):
    return sigmoid(np.dot(weight,x))
```

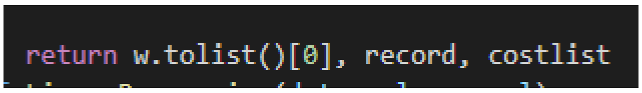


```
def cost(x,y,weight):
    oywx = np.multiply(y,np.array(np.dot(weight,x)))
    ss = np.log(1/sigmoid(oywx))
    return np.sum(ss)/len(x.T)
def gradient(x,y,w):
    oywx = np.multiply(y,np.array(np.dot(w,x)))
    h = np.multiply(y,(sigmoid(oywx) - 1))
    grad = (np.multiply(h,x).sum(axis=1))/len(x.T)

    return grad
```



```
while epoch < 7000:
    r = predit(x,w)
    costlist.append(cost(x,y,w))
    gradient1 = gradient(x,y,w)
    w -= gradient1.T
    epoch += 1

    r = np.sign(r - 0.5)

    mistake = np.sum(np.equal(r, y)==False)
    record.append(mistake)
```

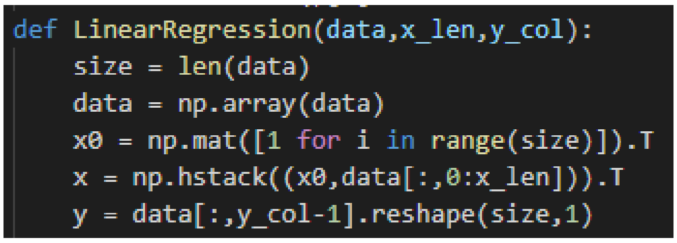
Return the weight and mistake changes and cost changes.



```
return w.tolist()[0], record, costlist
```

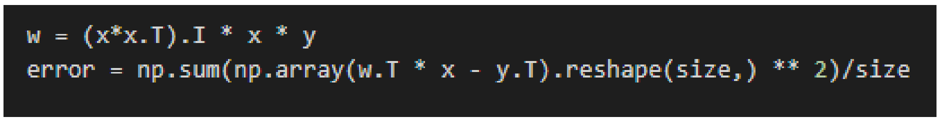
4. Implementation of Linear Regression:

Step 0: Data preprocessing, extend the dimension from x1~xn to x0~xn.



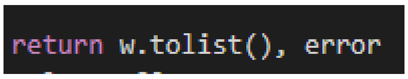
```
def LinearRegression(data,x_len,y_col):
    size = len(data)
    data = np.array(data)
    x0 = np.mat([1 for i in range(size)]).T
    x = np.hstack((x0,data[:,0:x_len])).T
    y = data[:,y_col-1].reshape(size,1)
```

Step 1: Compute the weight with dataset x and y based on the equation learn in class, and compute the error:



```
w = (x*x.T).I * x * y
error = np.sum(np.array(w.T * x - y.T).reshape(size,) ** 2)/size
```

Return the weight and error:



```
return w.tolist(), error
```

5. Optimizations

    1. Work data normalization and standardization first to clean data.
    2. Set the column number as part of parameter, so that can switch testing all PLA, pocket and Logistic Regression with both column 4 and 5 easier.
    3. For Pocket algorithm, use both picking mistake point randomly and picking mistake point with largest lost to modify weight. This can increase the speed of convergence.
    4. For Logistic regression, I recorded the cost in each iteration, which can track if it modify correctly.

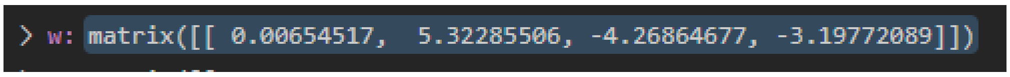
6. Challenges faced

    1. At first, I use 1.0 as the study rate (step length) and found that the weight can't converge and the number of mistake points keep in range of 100 ~ 400. Then I try many different study rate, found that 0.1 is much better than others, in convergent speed as well as accuracy of result.
    2. When implementing Pocket algorithm and Logistic regression for data of column 5 as y label. I found out that the accuracy rate is always around 50%~52% and there are still 940+ points misclassified. I start confusing whether my implementation have some mistake. But it works well in data of column 4 as y label. Then I try using sklearn package to test, and I found that the result is similar with mine (Provided at part 2 below). Therefore, I think using the data of column 5 as y label is not well-separable with both Pocket and Logistic regression.

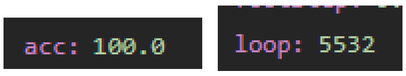
7. Result

1. PLA:

a) For column 4:

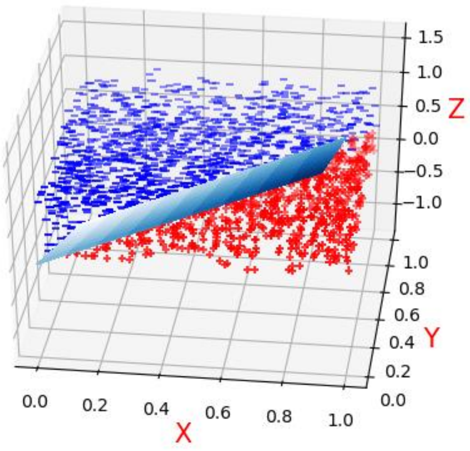


```
> w: matrix([[ 0.00654517,  5.32285506, -4.26864677, -3.19772089]])
```



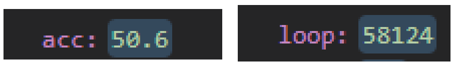
```
acc: 100.0
```

```
loop: 5532
```



b) For column 5:

Can't terminate because it's not linear separable. It keeps having around 50% accuracy rate even after 50,000+ iterations:

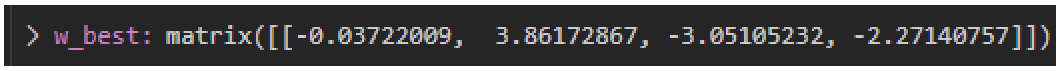


```
acc: 50.6
```

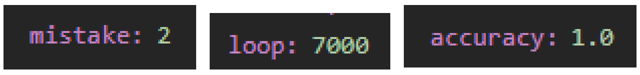
```
loop: 58124
```

2. Pocket method:

a) For column 4:



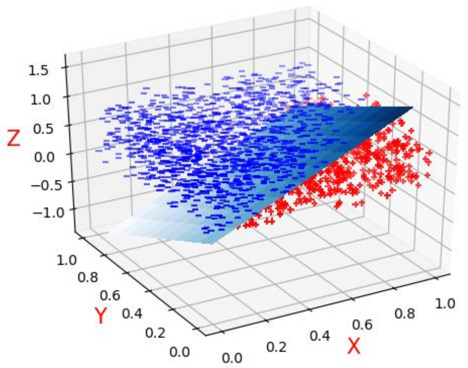
```
> w_best: matrix([[-0.03722009,  3.86172867, -3.05105232, -2.27140757]])
```

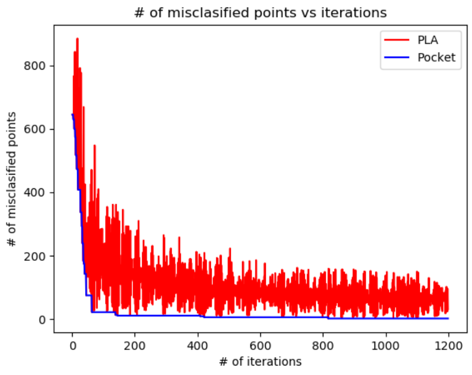


```
mistake: 2
```

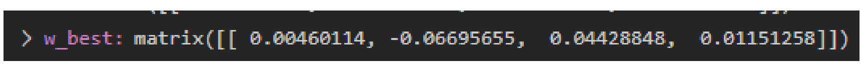
```
loop: 7000
```

```
accuracy: 1.0
```

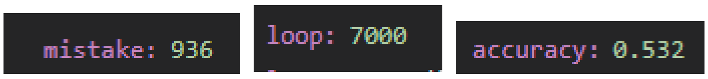




b) For column 5:



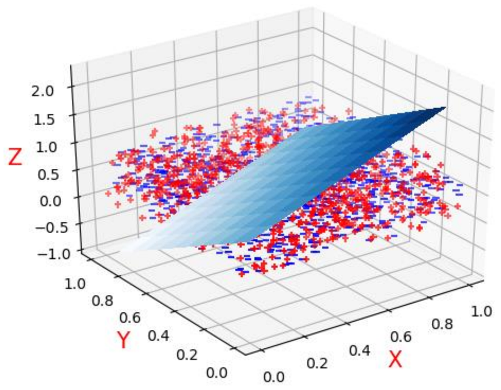
```
> w_best: matrix([[ 0.00460114, -0.06695655,  0.04428848,  0.01151258]])
```

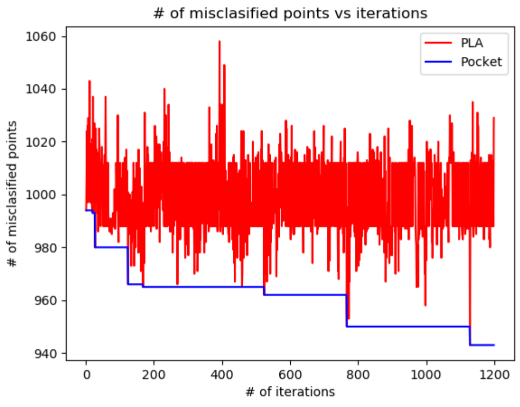


```
mistake: 936
```

```
loop: 7000
```

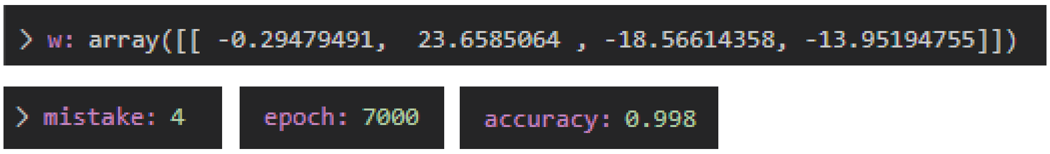
```
accuracy: 0.532
```





3. Logistic Regression:

a) For column 4:

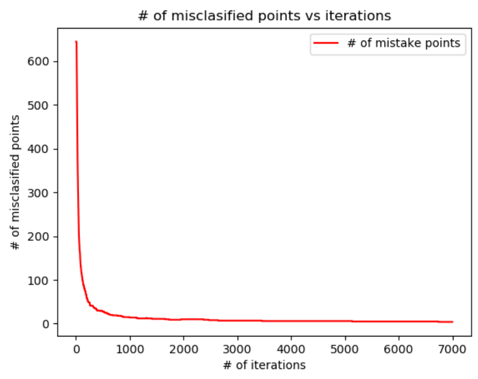


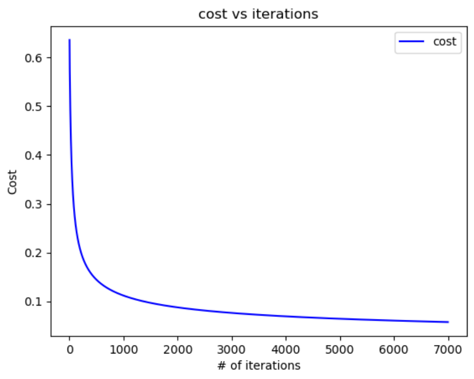
```
> w: array([[ -0.29479491,  23.6585064 , -18.56614358, -13.95194755]])
```

```
> mistake: 4
```

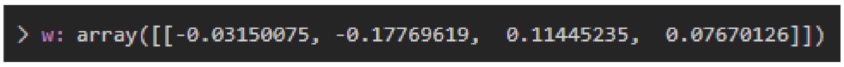
```
epoch: 7000
```

```
accuracy: 0.998
```

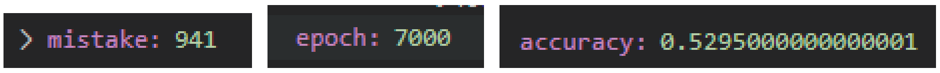




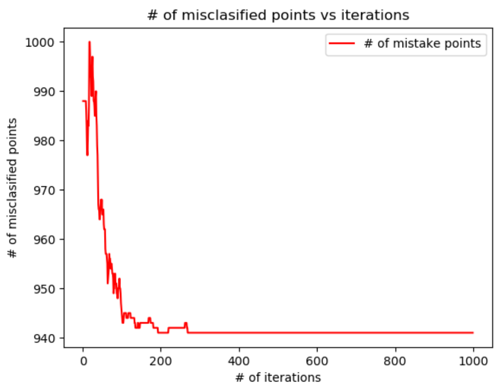
b) For column 5:

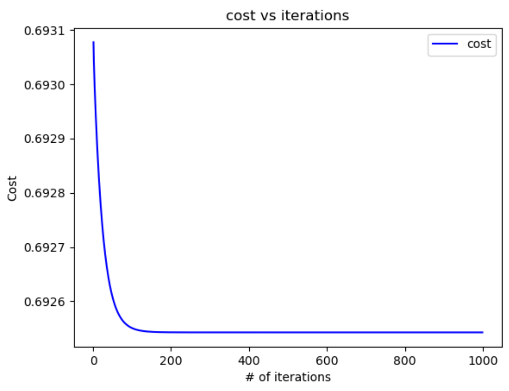


```
> w: array([[-0.03150075, -0.17769619,  0.11445235,  0.07670126]])
```

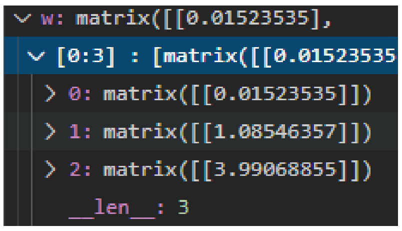


```
> mistake: 941
epoch: 7000
accuracy: 0.5295000000000001
```

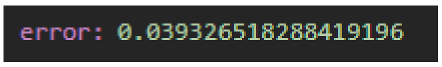




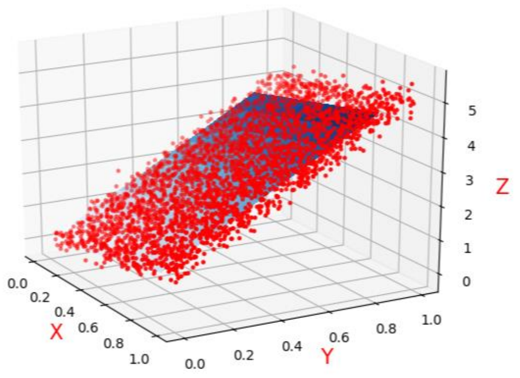
4. Linear Regression



```
v w: matrix([[0.01523535],
  v [0:3] : [matrix([[0.01523535
    > 0: matrix([[0.01523535]])
    > 1: matrix([[1.08546357]])
    > 2: matrix([[3.99068855]])
      __len__: 3
```



```
error: 0.039326518288419196
```



3. Part 2: Software familiarization

    1. Library function: sklearn.decomposition.Perceptron,

       sklearn.linear_model.logistic.LogisticRegression, sklearn.linear_model

    2. How to use:

        1. PLA:Simply use

```
clf=Perceptron(max_iter=7000,eta0=0.1)
clf.fit(dataclas_np[:,0:3],dataclas_np[:,4])
```

to set max iteration. Then get the weight vector (.coef and .intercept for w_0).

```
weights=clf.coef_

bias=clf.intercept_
```

Then output the accuracy

```
y_pre = clf.predict(XX)
d = np.equal(y_pre, yy)
dd = np.sum(np.equal(y_pre, yy)==True)
print('matchs:{0}/{1}'.format(np.sum(np.equal(y_pre, yy)==True), yy.shape[0]))
print(dd/yy.shape[0])
```

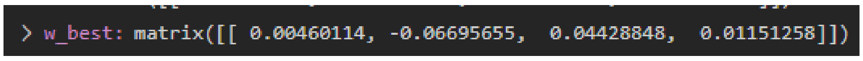
Result:

```
> bias: array([0.])
```

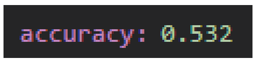
```
> weights: array([[-1.22987189,  0.00285867, -2.14257355]])
```

```
matchs:1004/2000
0.502
```

Compare with my implementation:



```
> w_best: matrix([[ 0.00460114, -0.06695655,  0.04428848,  0.01151258]])
```

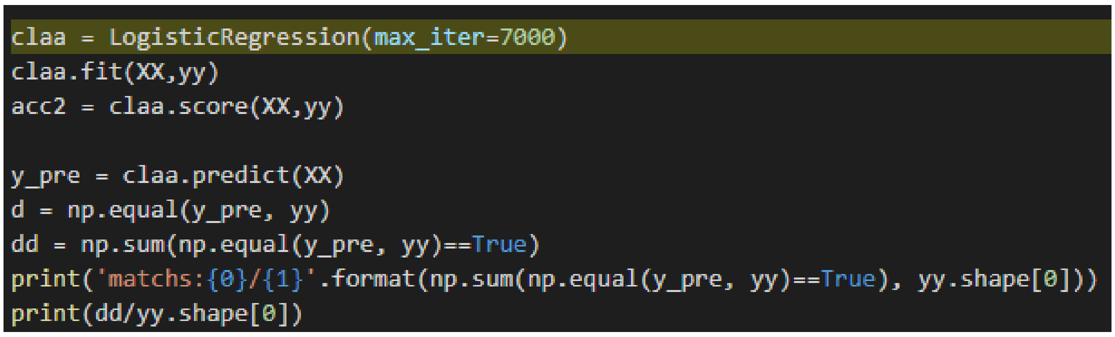


```
accuracy: 0.532
```

My implement even has greater accuracy rate.

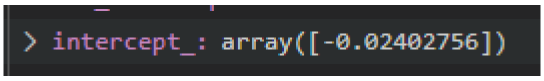
2. Logistic Regression:

   Similar with PLA/Pocket:

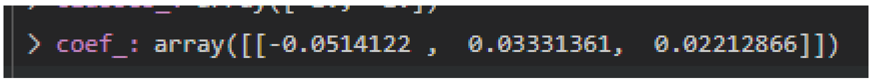


```
claa = LogisticRegression(max_iter=7000)
claa.fit(XX,yy)
acc2 = claa.score(XX,yy)

y_pre = claa.predict(XX)
d = np.equal(y_pre, yy)
dd = np.sum(np.equal(y_pre, yy)==True)
print('matchs:{0}/{1}'.format(np.sum(np.equal(y_pre, yy)==True), yy.shape[0]))
print(dd/yy.shape[0])
```

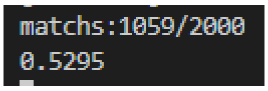
   Result:



```
> intercept_: array([-0.02402756])
```

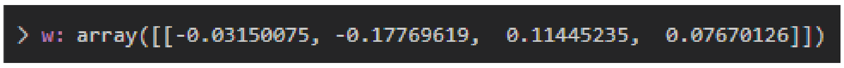


```
> coef_: array([[-0.0514122 ,  0.03331361,  0.02212866]])
```

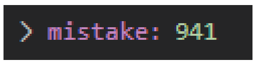


```
matchs:1059/2000
0.5295
```

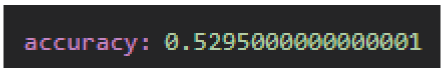
   Compare with my implement:



```
> w: array([[-0.03150075, -0.17769619,  0.11445235,  0.07670126]])
```



```
> mistake: 941
```

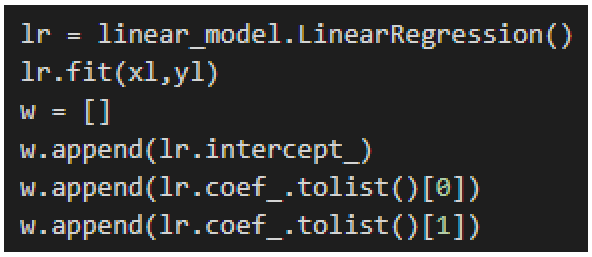


```
accuracy: 0.5295000000000001
```

   The weight is super closed and the classification result is exactly same.

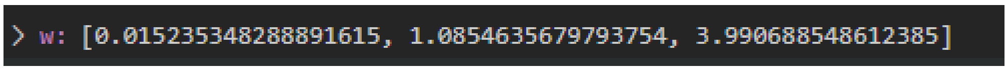
3. Linear Regression:

   Same as first two:



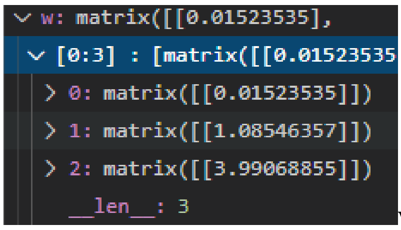
```
lr = linear_model.LinearRegression()
lr.fit(xl,yl)
w = []
w.append(lr.intercept_)
w.append(lr.coef_.tolist()[0])
w.append(lr.coef_.tolist()[1])
```

Result:



```
> w: [0.015235348288891615, 1.0854635679793754, 3.990688548612385]
```

Compare with my result:



```
v w: matrix([[0.01523535],
  v [0:3] : [matrix([[0.01523535
    > 0: matrix([[0.01523535]])
    > 1: matrix([[1.08546357]])
    > 2: matrix([[3.99068855]])
      __len__: 3
```

which is exactly same.

4. Comparison:

    1. In Sklearn package, PLA and pocket algorithm is within one library. It will always output the best result it gets like pocket algorithm. But it has more parameter can choose. For example, it can choose whether shuffle data after each epoch. This what my implementation doesn't have.
    2. Sklearn package doesn't pick error point to modify weight randomly, so it works faster them mine.
    3. All this three libraries can be used to deal with multi-classification problems, but my implementation can only solve two classification problems.

4. Part 3: Applications

PLA, pocket and logistic regression are all classification method, so it can be used in many real problems which need to classify something. For example, we can record patients' data like age, gender and number of disaster they have, to implement classification to see whether they are likely to get coronavirus, so that we can provide test to these group of people first.

Linear regression is a fitting method, which can be used in prediction model. For example, we can predict the GDP of a certain country with data set of its GDP in previous 20 years.