XinqianWang s4565489

1. (a)

My knowledge told me a regression MLP that uses identity activation functions for all neurons would have no difference with a OLS model. So I plan to build two Regression models using Pytouch and Sk-Learn respectively, and do iteration for 100 times to check if the things have happened.

```
import torch
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import torch.optim as optim
import torch.nn as nn
from torch.nn.modules.loss import MSELoss
def regression_data(n=500, d=2):
    X = torch.rand(n, d)
    w = torch.rand(d+1)
    Y = X @ w[1:] + w[0] + torch.rand(n) * 0.1
    return X, Y
def ols3(X, Y, niter=200, lr=0.5):
    K = []
    Y = Y.reshape(-1, 1) #Change to n*1
    net = nn.Linear(2, 1)
    optimizer = optim.SGD(net.parameters(), lr=lr, momentum=0)
    mse = MSELoss()
    for i in range(niter):
        optimizer.zero_grad()
        loss = mse(net(X), Y)
        loss.backward()
        optimizer.step()
    for param  in net.parameters():
        K.append(param.data)
    return K
ols3(X, Y)
```

```python
def Data_7703_1_a():
    X, Y = regression_data(d=2)

    reg = LinearRegression()
    reg.fit(X, Y)
    y_pred = reg.predict(X)
    SK_MSE = mean_squared_error(Y, y_pred)

    L = ols3(X, Y)
    coef = L[0]
    Intercept = L[1]
    tensor_y_predict = coef@X.T+0.2947
    coef = torch.tensor([[0.8489, 0.5036]], requires_grad=True)
    tensor_y_predict = tensor_y_predict[0]
    tensor_y_predict = tensor_y_predict.detach().numpy()
    NLP_MSE = mean_squared_error(Y, y_pred)

    if NLP_MSE - SK_MSE != 0:
        print("!!!")

for i in range(1,101):
    Data_7703_1_a()
```

I failed to output a better value that better than OLS, my understanding is that normally, it would not out put lower MSE, may be in some extreme situation resulted by computer's calculation it would have a little difference. Basicly the data scientist's announcement is not right.

1. (b)

$$\omega_{t+1} = \omega_t - \eta_t \bigtriangledown L_\lambda(\omega_t)$$

$$\bigtriangledown L_\lambda(\omega_t) = \bigtriangledown L(\omega_t) + \lambda\omega_t$$

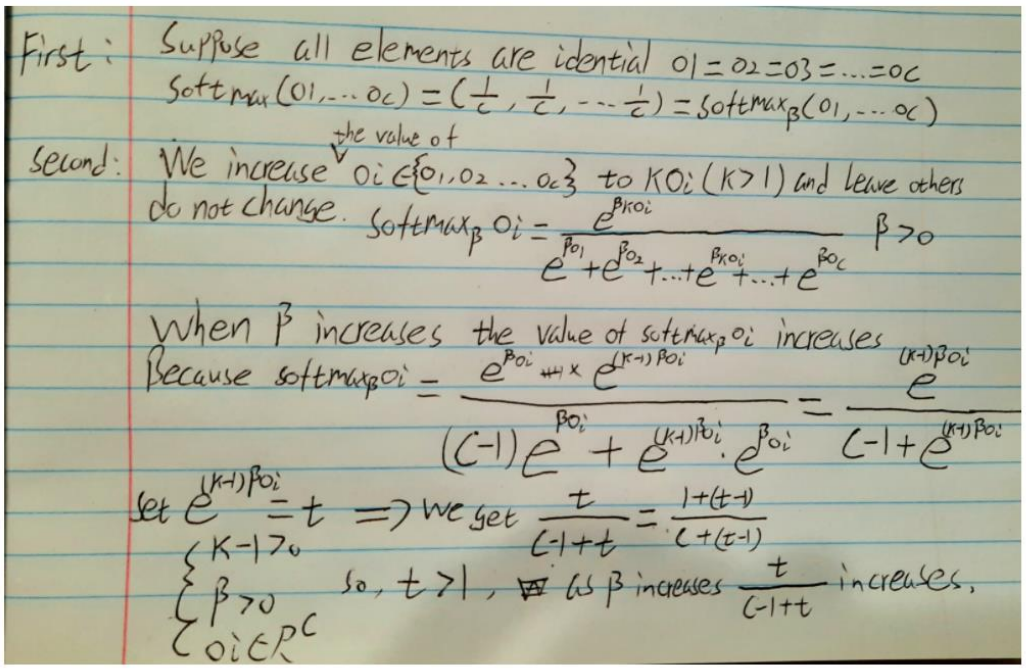
$$\bigtriangledown L_\lambda(\omega_t) > \bigtriangledown L(\omega_t)$$

$$\omega_{t+1} = \omega_t - \eta_t(\bigtriangledown L(\omega_t) + \lambda\omega_t) = (1-\lambda)\omega_t - \eta_t \bigtriangledown L(\omega_t$$

We can see by using the L2 regularization the new weight t+1 would decrease, which is equivalent to first multiply ω by a constant value λ.

Compared to L(ω), the new loss function would decrease more in a gradient descent step. And we can also observe the weight decrease by λ times.

1. (c)



First: Suppose all elements are idential $o1 = o2 = o3 = \dots = oc$

$softmax(o1, \dots oc) = (\frac{1}{c}, \frac{1}{c}, \dots \frac{1}{c}) = softmax_\beta(o1, \dots oc)$

Second: We increase the value of $oi \in \{o1, o2 \dots oc\}$ to Koi $(K>1)$ and leave others do not change. $softmax_\beta\, oi = \frac{e^{\beta K oi}}{e^{\beta o_1} + e^{\beta o_2} + \dots + e^{\beta K o_i} + \dots + e^{\beta o_c}}$ $\beta > 0$

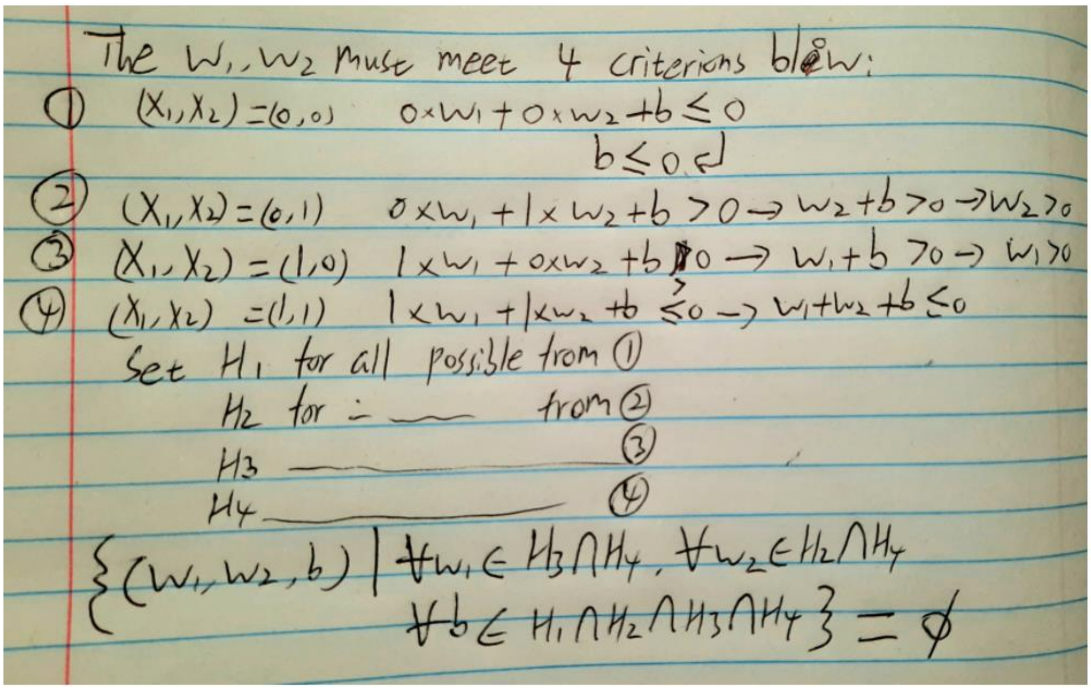
When β increases the value of $softmax_\beta\, oi$ increases

Because $softmax_\beta\, oi = \frac{e^{\beta o_i} \times e^{(K-1)\beta o_i}}{(C-1)e^{\beta o_i} + e^{(K-1)\beta o_i} \cdot e^{\beta o_i}} = \frac{e^{(K-1)\beta o_i}}{C-1+e^{(K-1)\beta o_i}}$

Set $e^{(K-1)\beta o_i} = t \Rightarrow$ we get $\frac{t}{C-1+t} = \frac{1+(t-1)}{C+(t-1)}$

$\begin{cases} K-1>0 \\ \beta>0 \\ oi \in R^C \end{cases}$ so, $t>1$, as β increases $\frac{t}{C-1+t}$ increases.

1. (d)



The w_1, w_2 must meet 4 criterions blow:

① $(X_1, X_2) = (0,0)$ $\quad 0 \times w_1 + 0 \times w_2 + b \le 0$

$b \le 0$ ↵

② $(X_1, X_2) = (0,1)$ $\quad 0 \times w_1 + 1 \times w_2 + b > 0 \rightarrow w_2 + b > 0 \rightarrow w_2 > 0$

③ $(X_1, X_2) = (1,0)$ $\quad 1 \times w_1 + 0 \times w_2 + b > 0 \rightarrow w_1 + b > 0 \rightarrow w_1 > 0$

④ $(X_1, X_2) = (1,1)$ $\quad 1 \times w_1 + 1 \times w_2 + b \le 0 \rightarrow w_1 + w_2 + b \le 0$

Set H_1 for all possible from ①

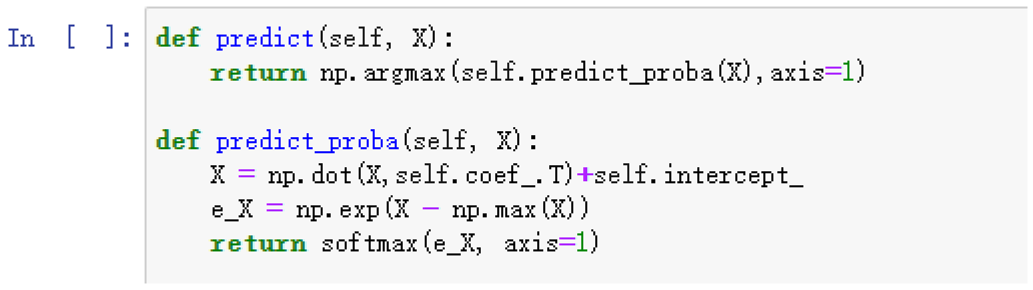
H_2 for ——— from ②

H_3 ——— ③

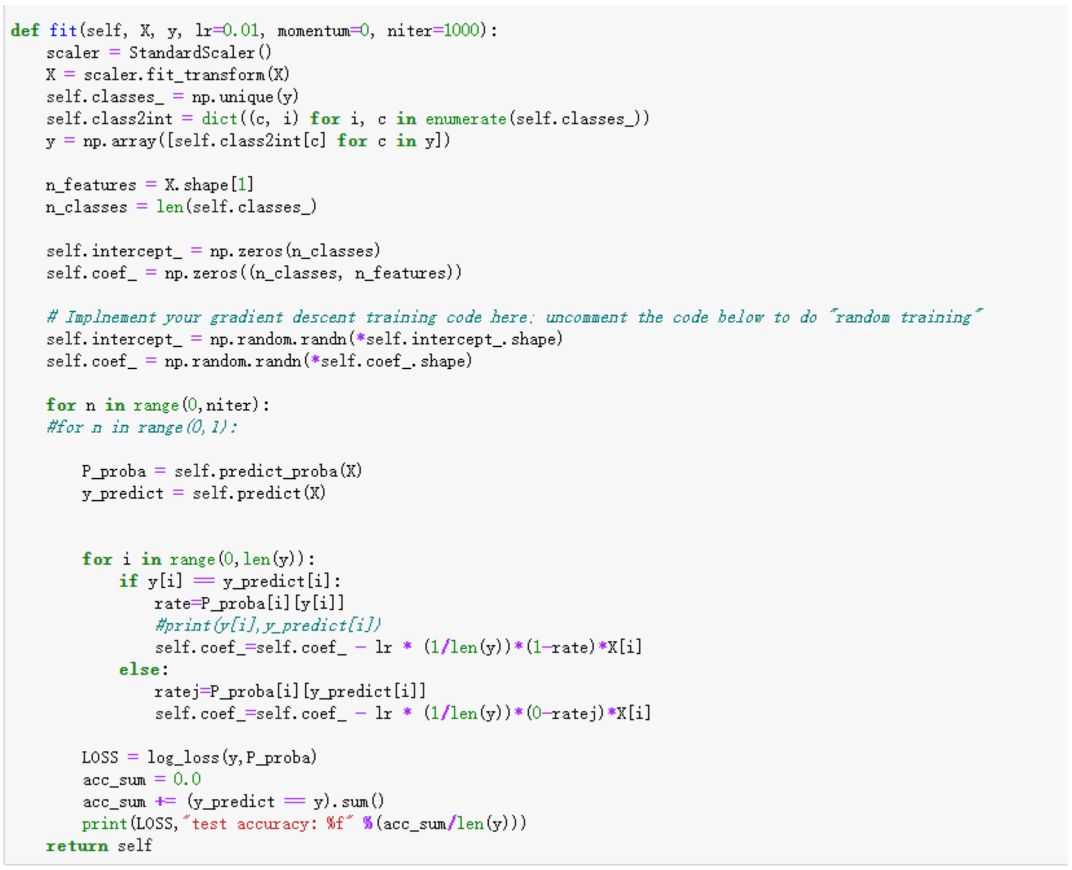
H_4 ——— ④

$\{(w_1, w_2, b) \mid \forall w_1 \in H_3 \cap H_4, \forall w_2 \in H_2 \cap H_4, \forall b \in H_1 \cap H_2 \cap H_3 \cap H_4\} = \emptyset$

2. (b),(c)



```
In [ ]: def predict(self, X):
            return np.argmax(self.predict_proba(X),axis=1)

        def predict_proba(self, X):
            X = np.dot(X,self.coef_.T)+self.intercept_
            e_X = np.exp(X - np.max(X))
            return softmax(e_X, axis=1)
```



```
def fit(self, X, y, lr=0.01, momentum=0, niter=1000):
    scaler = StandardScaler()
    X = scaler.fit_transform(X)
    self.classes_ = np.unique(y)
    self.class2int = dict((c, i) for i, c in enumerate(self.classes_))
    y = np.array([self.class2int[c] for c in y])

    n_features = X.shape[1]
    n_classes = len(self.classes_)

    self.intercept_ = np.zeros(n_classes)
    self.coef_ = np.zeros((n_classes, n_features))

    # Implnement your gradient descent training code here; uncomment the code below to do "random training"
    self.intercept_ = np.random.randn(*self.intercept_.shape)
    self.coef_ = np.random.randn(*self.coef_.shape)

    for n in range(0,niter):
    #for n in range(0,1):

        P_proba = self.predict_proba(X)
        y_predict = self.predict(X)


        for i in range(0,len(y)):
            if y[i] == y_predict[i]:
                rate=P_proba[i][y[i]]
                #print(y[i],y_predict[i])
                self.coef_=self.coef_ - lr * (1/len(y))*(1-rate)*X[i]
            else:
                ratej=P_proba[i][y_predict[i]]
                self.coef_=self.coef_ - lr * (1/len(y))*(0-ratej)*X[i]

        LOSS = log_loss(y,P_proba)
        acc_sum = 0.0
        acc_sum += (y_predict == y).sum()
        print(LOSS,"test accuracy: %f" %(acc_sum/len(y)))
    return self
```

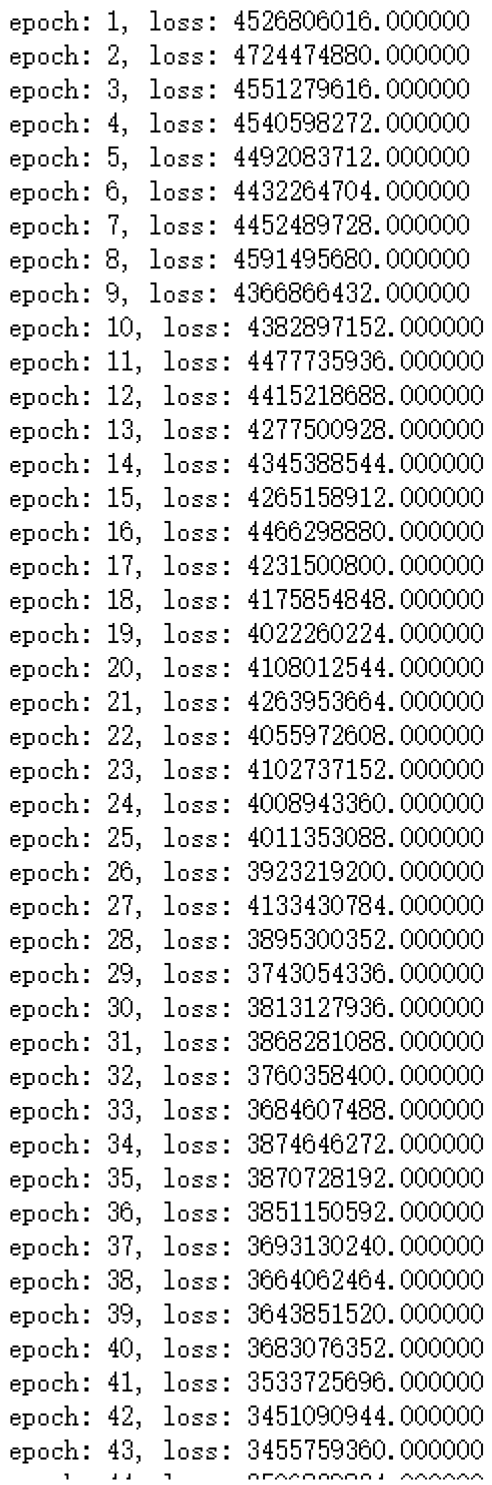
My numpy function worked not well so I change to pytorch.

I deduced because of I have not do Standardlization to X, the distribution of X would got a lot of same value at first, but the arg max seems not functioned well on them. I think this is the reason. ¶

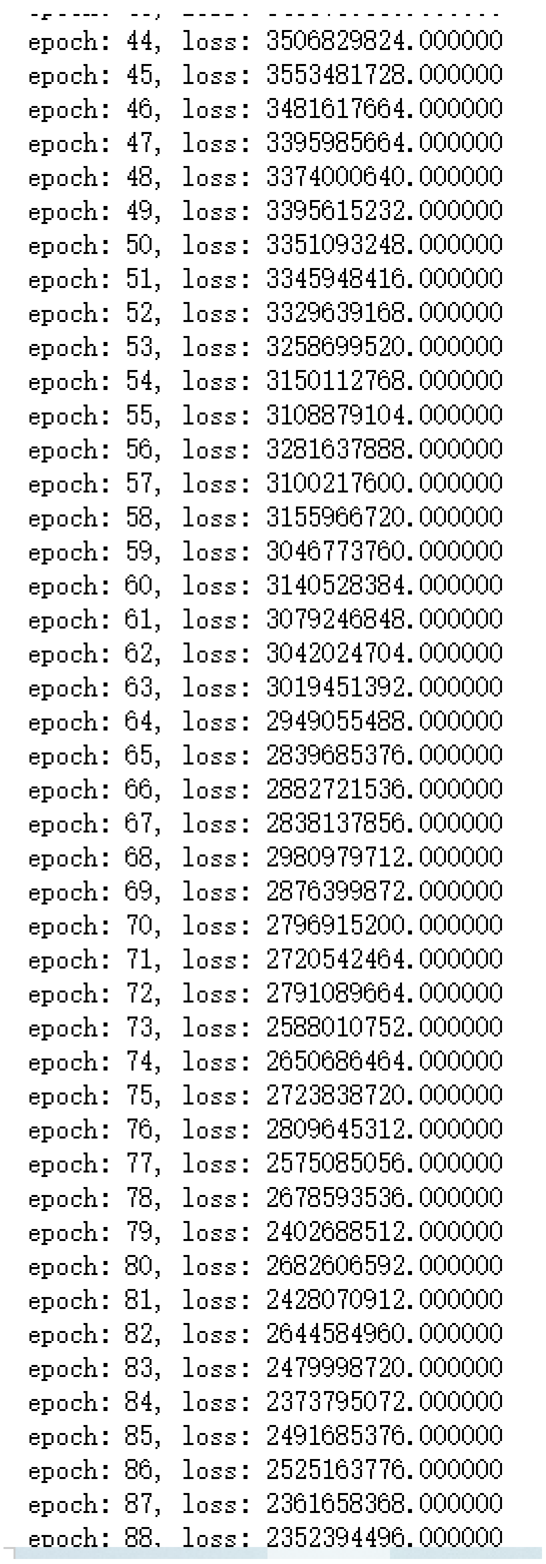
2. (d),(e),(f),(g)

[76]:

```python
import numpy as np
import torch.nn as nn
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_covtype
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler

import torch
import torch.optim as optim
import torch.utils.data as Data

class Logstic_Regression(nn.Module):
    def __init__(self, X,y):
        super(Logstic_Regression, self).__init__()
        scaler = StandardScaler()
        X = scaler.fit_transform(X)
        K_train = np.max(X)
        X = X - K_train

        self.classes_ = np.unique(y)
        self.class2int = dict((c, i) for i, c in enumerate(self.classes_))
        y = np.array([self.class2int[c] for c in y])
        print(np.unique(y))
        n_features = X.shape[1]
        n_classes = len(self.classes_)

        self.w = nn.Parameter(torch.randn(n_classes, n_features))
        self.b = nn.Parameter(torch.zeros(n_classes))

        self._X = torch.from_numpy(X).type(torch.FloatTensor)
        self._y = torch.from_numpy(y).type(torch.LongTensor)

        self.net = nn.Sequential(
            nn.LogSoftmax()
        )

    def fit(self, lr=10, momentum=0.9, niter=100,BATCH_SIZE=100):
        LOSS_FUNC = nn.CrossEntropyLoss()
        OPTIMIZER = torch.optim.SGD([self.w, self.b], lr=lr,momentum=momentum)
        train_set = Data.TensorDataset(self._X,self._y)
        train_loader = Data.DataLoader(dataset=train_set,batch_size=BATCH_SIZE,shuffle=True)
        for epoch in range(1,niter+1):
            loss_sum = 0.0
            for step,(x,y) in enumerate(train_loader):
                y_pred = self.predict_proba(x)
                y = y.squeeze()
                loss = LOSS_FUNC(y_pred,y)
                loss_sum += loss
                OPTIMIZER.zero_grad()
                loss.backward()
                OPTIMIZER.step()
            print("epoch: %d, loss: %f" %(epoch,loss_sum/BATCH_SIZE))



    def predict_proba(self, X):
        X = torch.mm(X, self.w.T) + self.b.T
        return self.net(X)

    def predict(self, X):
        X = self.predict_proba(X)
        return X.argmax(dim=1)

if __name__ == '__main__':
    X, y = fetch_covtype(return_X_y=True)
    X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, test_size=0.3, random_state=5)

    LR = Logstic_Regression(X_tr,y_tr)
    LR.fit()
```



```
epoch: 1, loss: 4526806016.000000
epoch: 2, loss: 4724474880.000000
epoch: 3, loss: 4551279616.000000
epoch: 4, loss: 4540598272.000000
epoch: 5, loss: 4492083712.000000
epoch: 6, loss: 4432264704.000000
epoch: 7, loss: 4452489728.000000
epoch: 8, loss: 4591495680.000000
epoch: 9, loss: 4366866432.000000
epoch: 10, loss: 4382897152.000000
epoch: 11, loss: 4477735936.000000
epoch: 12, loss: 4415218688.000000
epoch: 13, loss: 4277500928.000000
epoch: 14, loss: 4345388544.000000
epoch: 15, loss: 4265158912.000000
epoch: 16, loss: 4466298880.000000
epoch: 17, loss: 4231500800.000000
epoch: 18, loss: 4175854848.000000
epoch: 19, loss: 4022260224.000000
epoch: 20, loss: 4108012544.000000
epoch: 21, loss: 4263953664.000000
epoch: 22, loss: 4055972608.000000
epoch: 23, loss: 4102737152.000000
epoch: 24, loss: 4008943360.000000
epoch: 25, loss: 4011353088.000000
epoch: 26, loss: 3923219200.000000
epoch: 27, loss: 4133430784.000000
epoch: 28, loss: 3895300352.000000
epoch: 29, loss: 3743054336.000000
epoch: 30, loss: 3813127936.000000
epoch: 31, loss: 3868281088.000000
epoch: 32, loss: 3760358400.000000
epoch: 33, loss: 3684607488.000000
epoch: 34, loss: 3874646272.000000
epoch: 35, loss: 3870728192.000000
epoch: 36, loss: 3851150592.000000
epoch: 37, loss: 3693130240.000000
epoch: 38, loss: 3664062464.000000
epoch: 39, loss: 3643851520.000000
epoch: 40, loss: 3683076352.000000
epoch: 41, loss: 3533725696.000000
epoch: 42, loss: 3451090944.000000
epoch: 43, loss: 3455759360.000000
```



```
epoch: 44, loss: 3506829824.000000
epoch: 45, loss: 3553481728.000000
epoch: 46, loss: 3481617664.000000
epoch: 47, loss: 3395985664.000000
epoch: 48, loss: 3374000640.000000
epoch: 49, loss: 3395615232.000000
epoch: 50, loss: 3351093248.000000
epoch: 51, loss: 3345948416.000000
epoch: 52, loss: 3329639168.000000
epoch: 53, loss: 3258699520.000000
epoch: 54, loss: 3150112768.000000
epoch: 55, loss: 3108879104.000000
epoch: 56, loss: 3281637888.000000
epoch: 57, loss: 3100217600.000000
epoch: 58, loss: 3155966720.000000
epoch: 59, loss: 3046773760.000000
epoch: 60, loss: 3140528384.000000
epoch: 61, loss: 3079246848.000000
epoch: 62, loss: 3042024704.000000
epoch: 63, loss: 3019451392.000000
epoch: 64, loss: 2949055488.000000
epoch: 65, loss: 2839685376.000000
epoch: 66, loss: 2882721536.000000
epoch: 67, loss: 2838137856.000000
epoch: 68, loss: 2980979712.000000
epoch: 69, loss: 2876399872.000000
epoch: 70, loss: 2796915200.000000
epoch: 71, loss: 2720542464.000000
epoch: 72, loss: 2791089664.000000
epoch: 73, loss: 2588010752.000000
epoch: 74, loss: 2650686464.000000
epoch: 75, loss: 2723838720.000000
epoch: 76, loss: 2809645312.000000
epoch: 77, loss: 2575085056.000000
epoch: 78, loss: 2678593536.000000
epoch: 79, loss: 2402688512.000000
epoch: 80, loss: 2682606592.000000
epoch: 81, loss: 2428070912.000000
epoch: 82, loss: 2644584960.000000
epoch: 83, loss: 2479998720.000000
epoch: 84, loss: 2373795072.000000
epoch: 85, loss: 2491685376.000000
epoch: 86, loss: 2525163776.000000
epoch: 87, loss: 2361658368.000000
epoch: 88, loss: 2352394496.000000
```

:
```
y_ts = y_ts-1
scaler = StandardScaler()
X_ts = scaler.fit_transform(X_ts)
X_ts = torch.from_numpy(X_ts).type(torch.FloatTensor)
y_ts = torch.from_numpy(y_ts).type(torch.LongTensor)
acc_sum = 0.0
acc_sum += (LR.predict(X_ts) == y_ts.squeeze()).sum()
print("test accuracy: %f" %(acc_sum/len(y_ts)))
```

```
test accuracy: 0.594811
```

```
C:\Users\Andy\Anaconda3\lib\site-packages\torch\nn\modules\container.py:117: UserWarning: Implicit dimension choice for log_softmax has been
deprecated. Change the call to include dim=X as an argument.
  input = module(input)
```

3. (b),(c)

Sorry for late submission, I unconsciously delete the code for (b) and (c), I just need to redo again. Blew is the redo answer, for me, may be a little messy.

2]:
```
import numpy as np

from sklearn.base import clone
from sklearn.datasets import load_boston
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression, RANSACRegressor, TheilSenRegressor
from sklearn.model_selection import train_test_split
from sklearn.utils import check_random_state

def corrupt(X, y, outlier_ratio=0.1, random_state=None):
    random = check_random_state(random_state)

    n_samples = len(y)
    n_outliers = int(outlier_ratio*n_samples)

    W = X.copy()
    z = y.copy()

    mask = np.ones(n_samples).astype(bool)
    outlier_ids = random.choice(n_samples, n_outliers)
    mask[outlier_ids] = False

    W[~mask, 4] *= 0.1

    return W, z
```

```
class ENOLS:
    def __init__(self, n_estimators=100, sample_size='auto'):
        '''
        Parameters
        ----------
        n_estimators: number of OLS models to train
        sample_size: size of random subset used to train the OLS models, default to 'auto'
            - If 'auto': use subsets of size n_features+1 during training
            - If int: use subsets of size sample_size during training
            - If float: use subsets of size ceil(n_sample*sample_size) during training
        '''

        self.n_estimators = n_estimators
        self.sample_size = sample_size
```

```python
def fit(self, X, y, random_state=None):
    '''
    Train ENOLS on the given training set.

    Parameters
    ----------

    X: an input array of shape (n_sample, n_features)
    y: an array of shape (n_sample,) containing the values for the input examples

    Return
    ------

    self: the fitted model
    '''
    if self.sample_size=='auto':
        S_size = X.shape[1]+1
    elif isinstance(self.sample_size,int):
        S_size = self.sample_size
    elif isinstance(self.sample_size,float):
        S_size = X.shape[0]*self.sample_size
    else:
        xxxxxxx


    # use random instead of np.random to sample random numbers below
    random = check_random_state(random_state)

    # add all the trained OLS models to this list
    self.estimators_ = []

    # write your training code below. your code should support the
    # n_estimators and sample_size hyper-parameters described in the
    # documentation for the __init__ function
    self.base_estimator_ = LinearRegression()
    for n in range(0,self.n_estimators):
        estimator = clone(self.base_estimator_)
        X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=S_size, random_state=random)
        estimator.fit(X_train, y_train)
        self.estimators_.append(estimator)
    return self
```

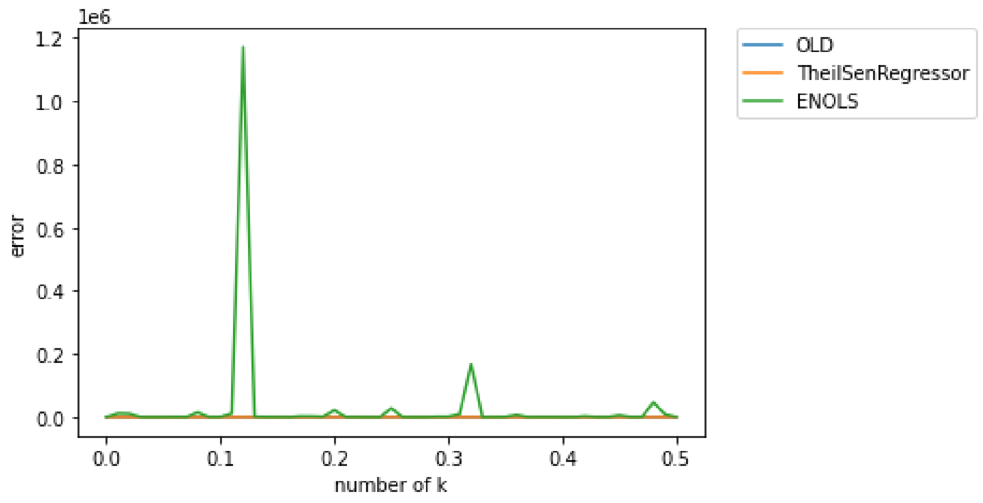
```
def predict(self, X, method='average'):
    '''
    Parameters
    ----------
    X: an input array of shape (n_sample, n_features)
    method: 'median' or 'average', corresponding to predicting median and
        mean of the OLS models' predictions respectively.

    Returns
    -------
    y: an array of shape (n_samples,) containig the predicted values
    '''

    if method == 'average':
        MEAN = []
        for SE in self.estimators_:
            P = SE.predict(X)
            MEAN.append(P)
        MEAN = np.array(MEAN)
        return np.mean(MEAN,axis=0)
    elif method == 'median':
        MEDIAN = []
        for SE in self.estimators_:
            P = SE.predict(X)
            MEDIAN.append(P)
        MEDIAN = np.array(MEDIAN)
        return np.median(MEDIAN,axis=0)
    else:
        xxxxxxxx
```

```
if __name__ == '__main__':
    X, y = load_boston(return_X_y=True)
    X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, test_size=0.3, random_state=42)
    W, z = corrupt(X_tr, y_tr, outlier_ratio=0.1, random_state=42)
    '''
    reg = LinearRegression()
    reg.fit(X_tr, y_tr)
    print(mean_squared_error(y_ts, reg.predict(X_ts)))

    reg = LinearRegression()
    reg.fit(W, z)
    print(mean_squared_error(y_ts, reg.predict(X_ts)))

    EL = ENOLS()
    EL.fit(X_tr,y_tr)
    EL.predict(X_ts)
    '''
    OLD = []
    TSR = []
    ELS = []
    for i in range(0,51):
        p = 0.01*i
        W, z = corrupt(X_tr, y_tr, outlier_ratio=p, random_state=42)

        LR = LinearRegression()
        TR = TheilSenRegressor()
        ES = ENOLS()

        LR.fit(W, z)
        TR.fit(W, z)
        ES.fit(W, z)

        OLD.append(mean_squared_error(y_ts, LR.predict(X_ts)))
        TSR.append(mean_squared_error(y_ts, TR.predict(X_ts)))
        ELS.append(mean_squared_error(y_ts, ES.predict(X_ts,method='median')))
```

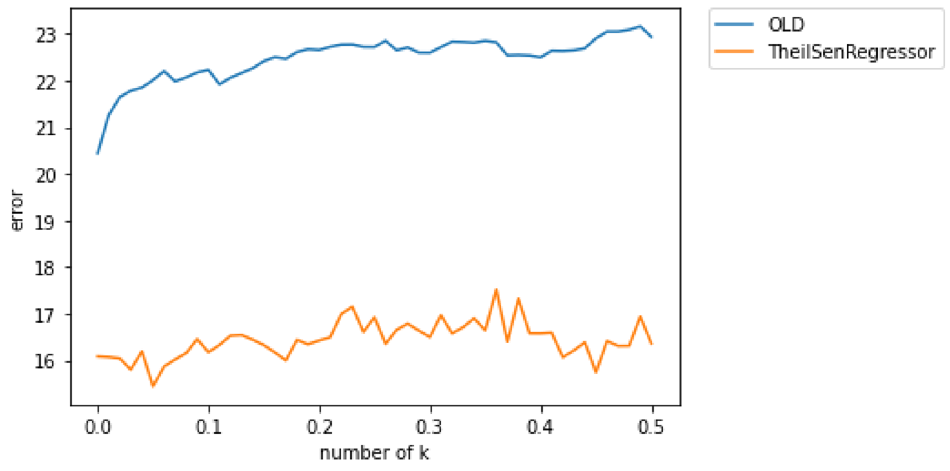
3 (d)

ENOLS seems to be less stable than OLD and TheilSenRegressor

```
import matplotlib.pyplot as plt
k = [i*0.01 for i in range(0,51)]
fig, ax = plt.subplots()
plt.plot(k, OLD,label="OLD")
plt.plot(k, TSR,label="TheilSenRegressor")
plt.plot(k, ELS,label="ENOLS")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
ax.set_xlabel('number of k')
ax.set_ylabel('error')
plt.show()
```



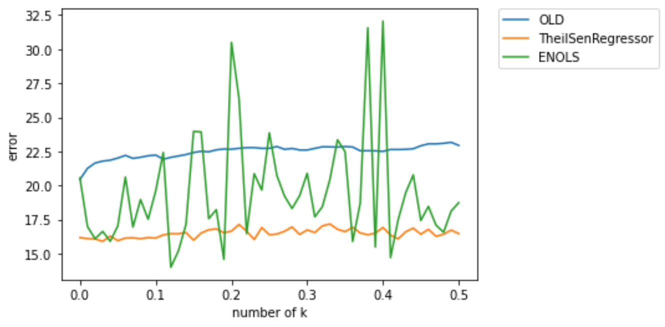
```
: import matplotlib.pyplot as plt
k = [i*0.01 for i in range(0,51)]
fig, ax = plt.subplots()
plt.plot(k, OLD,label="OLD")
plt.plot(k, TSR,label="TheilSenRegressor")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
ax.set_xlabel('number of k')
ax.set_ylabel('error')
plt.show()
```



3 (e)

ENOLS still not stable than others and TheilSenRegressor Algorithm has the highest robustness.

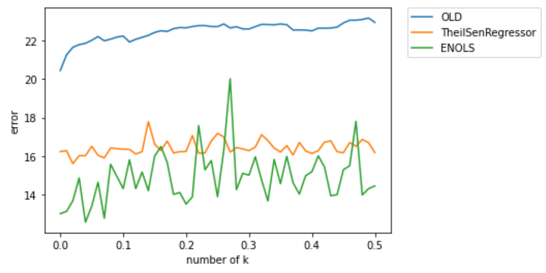
```
]: import matplotlib.pyplot as plt
k = [i*0.01 for i in range(0,51)]
fig, ax = plt.subplots()
plt.plot(k, OLD,label="OLD")
plt.plot(k, TSR,label="TheilSenRegressor")
plt.plot(k, ELS,label="ENOLS")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
ax.set_xlabel('number of k')
ax.set_ylabel('error')
plt.show()
```



3 (f)

When the number of estimators increases to a big number. The ENLOS gets to a highest stable condition compared to the other two algorithms.

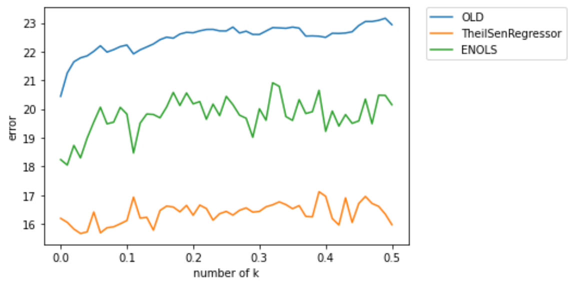
```
[180]: import matplotlib.pyplot as plt
k = [i*0.01 for i in range(0,51)]
fig, ax = plt.subplots()
plt.plot(k, OLD,label="OLD")
plt.plot(k, TSR,label="TheilSenRegressor")
plt.plot(k, ELS,label="ENOLS")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
ax.set_xlabel('number of k')
ax.set_ylabel('error')
plt.show()
```



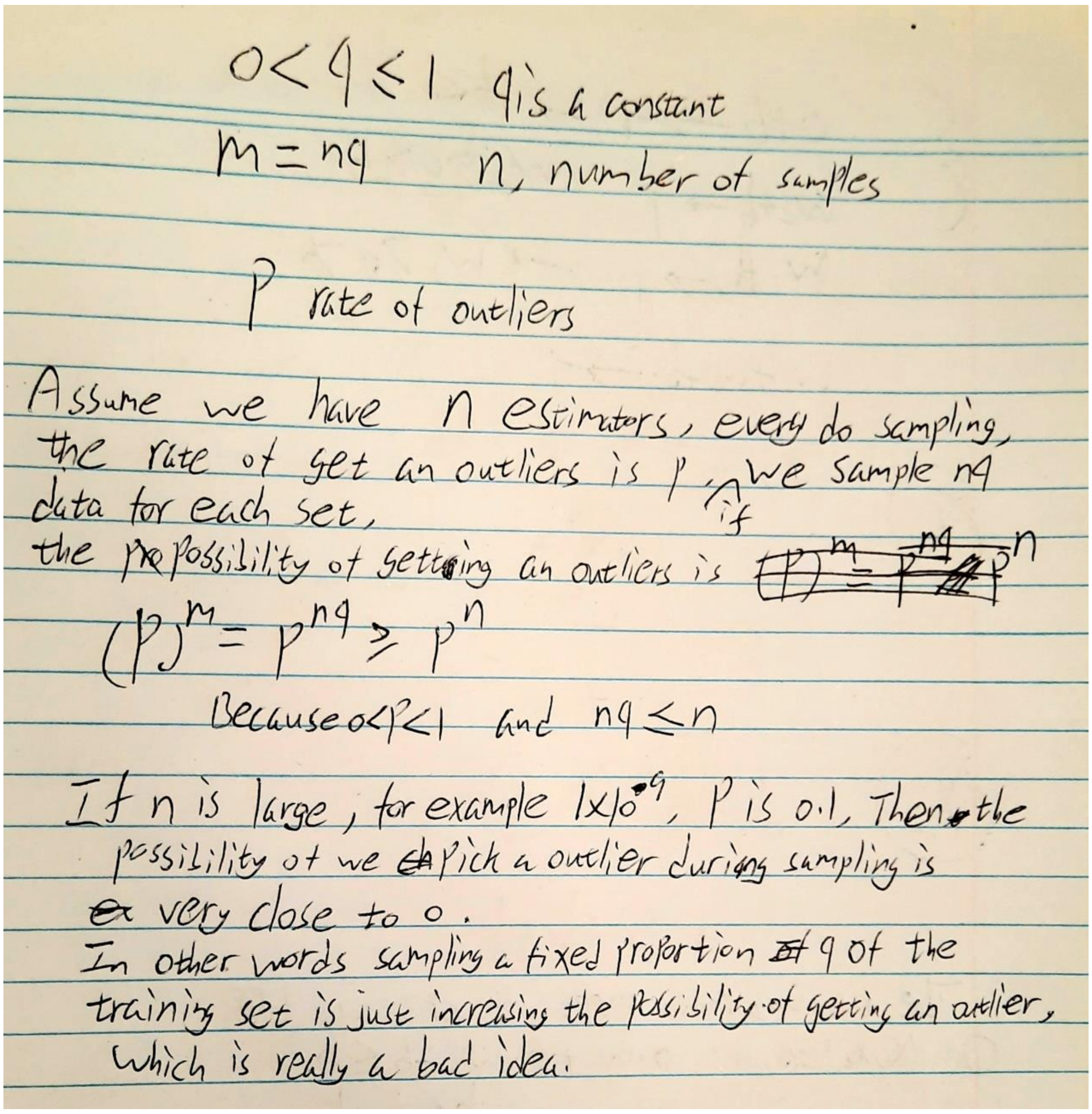
3 (g)

The fluctuation of ENOLS tends to become a little more placid than before pictures, but the robustness seems to be not as good as the previous ENOLS model, probably because we decrease the number of subset.

```
: import matplotlib.pyplot as plt
k = [i*0.01 for i in range(0,51)]
fig, ax = plt.subplots()
plt.plot(k, OLD,label="OLD")
plt.plot(k, TSR,label="TheilSenRegressor")
plt.plot(k, ELS,label="ENOLS")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
ax.set_xlabel('number of k')
ax.set_ylabel('error')
plt.show()
```



3(h)



$0 < q \leq 1$ q is a constant

$m = nq$ n, number of samples

P rate of outliers

Assume we have n estimators, every do sampling, the rate of get an outliers is P, if we sample nq data for each set,

the ~~pro~~ possibility of getting an outliers is ~~$(P)^m = P^{nq}$~~ P^n

$$(P)^m = P^{nq} \geq P^n$$

Because $0 < P < 1$ and $nq \leq n$

If n is large, for example 1×10^9, P is 0.1, Then the possibility of we ~~ch~~ pick a outlier during sampling is ~~ex~~ very close to 0.

In other words sampling a fixed proportion ~~of~~ q of the training set is just increasing the possibility of getting an outlier, which is really a bad idea.