# TERM PROJECT [ROBT407]

Performed by Anuar Maratkhan, Ainura Karabayeva

In [ ]:
```
% pylab inline
```

# 1. Linear Regression

Method of Linear Regression is based on minimization of square root error:

$$\min_{w} \sum_{n=1}^{N} [y_n - (w^T x_n)]^2$$

We have used the following algorithm:
1: Constructing the matrix $X$ (input data matrix) and vector $y$ (target vector) from the data set, including $x_0 = 1$ (the column of 1s)
2: Compute the pseudo-inverse $X^t = (X^T X)^{-1} X^T$ of the matrix $X$
3: Return $w_{lin} = X^t y$

For 1st task we implemented four functions: generating random data set, finding error of the model, pocket algorithm, and linesr regression.

In [ ]:
```
# generation random data set with giving size
def generate_data(size):
    w_true = random.randint(1,10,size=2) #coefficients of Xs
    b_true = random.randint(1,10) #bias term

    d = 2
    X = random.uniform(-1,1,(size,d))*10

    Y = zeros(20)

#calculating final hypothesis
#here the parameters of target function can be extracted:
#slope=-w_true[0]/w_true[1], bias=-b_true/w_true[1]

    h = X.dot(w_true) + b_true
    labels = (h > 0)*1

    # randomly choose 10
    indexes = random.randint(size, size=10)
    # flip chosen Ys
    for i in indexes:
        if (labels[i] == 1):
            labels[i] = 0
        else:
            labels[i] = 1
    Y = labels

    a=ones(size)     #to get bias coordinate x_0=1
    a1=a.reshape(size,1)
    X1=hstack((a1,X)) #input data matrix

    return X1, Y, w_true, b_true
```

In [ ]:
```
def find_error(x, w, y):
    E = 0.0
    for i in range(len(x)):
        ypred = dot(x[i],w)
        if(ypred > 0 and y[i] == -1): #predicted wrongly as 1st class
            E += 1
        elif(ypred <= 0 and y[i] == 1): #predicted wrongly as 2nd class
            E += 1
    return E/x.size #find Error term by finding mean value
```

In [ ]:

```
# oursize is the number of obsevations((x1, x2)), ntimes is the number of iterations of pocket algorithm
#w is the initial weights

def pocket(x, y, w, ntimes):

    t = [] #array of number of updates
    tt = 0 #current update number
    #w= zeros(3)
    w_new=[]
    E=[]
    E_val=0.0

    for ii in range (ntimes):

        y_wr_collection = [] #saves indeces of the wrong predicted y
        ypredict=[]

        for i in range(len(x)):

            if dot(x[i],w)>0:
                ypredict.append(1)
            else: ypredict.append(0)

            if ypredict[i] != y[i]:
                y_wr_collection.append(i)

        #randomly choose the index of wrong classified x
        pick_random = np.random.randint(len(y_wr_collection))

        #new weight
        w_new = w + (y[pick_random]-ypredict[pick_random]) * x[pick_random]

        E_old = find_error(x, w, y)
        E_new = find_error(x, w_new, y)

        if (E_new < E_old): #compare old and new Error

            E.append(E_new)
            t.append(tt)
            tt += 1
            w = copy(w_new) #update weight to new weight
            E_val=E_new


    return w
```

In [ ]:
```
def lin_reg(x, y):

    #to reshape in order to be able to compute dot product
    y_new=y.reshape(len(y),1)

    INV = np.linalg.inv(dot(x.T,x)) #inverse
    pseudo_inv_x=dot(INV, x.T)
    w=dot(pseudo_inv_x,y_new)

    E=find_error(x, w, y)
    w1=w.reshape(3)

    return w1
```

1) Generating a training data set of size 100:

In [ ]:
```
x_train,y_train,w_true_train, b_true_train=generate_data(100)
```

2) Generating a test data set of size 1000 of the identical nature of 100.

In [ ]:
```
x_test,y_test,w_true_test, b_true_test=generate_data(1000)
```

3) Run of Pocket algorithm:

In [ ]:
```
w=zeros(3) #set initial weight vector of 0s
w_pocket=pocket(x_train,y_train,w,1000)
print ('Weights are: ',w_pocket)
```

4) Run of Linear Regression algorithm:

In [ ]:
```
w_lin=lin_reg(x_train, y_train)
print ('Weights are: ',w_lin)
```

In [ ]:
```
slope_true=-w_true_train[0]/w_true_train[1]
bias_true=-b_true_train/w_true_train[1]
print('Slope of target function is ',slope_true, 'bias term is ',bias_true)
```

In [ ]:
```
bias_pocket = -w_pocket[1]/w_pocket[2]
slope_pocket = -w_pocket[0]/w_pocket[2]
print ('Final hypothesis of Pocket algorithm is ', 'g=',slope_pocket,'x','+',bias_pocket)
```

In [ ]:
```
a_lin = -w_lin[1]/w_lin[2]
b_lin = -w_lin[0]/w_lin[2]
print (a_lin,b_lin)
```

Here can be clearly seen that the slope values of both Pocket and Linear Reagression algorithms are very close to the slope of target function. However, the bias terms are different for both. It means that the lines approximated by two different algorithms have nearly equal slope but the location with respect to the origin are different.

5. Calculating $E_{test}(w_{pocket})$ and $E_{test}(w_{lin})$ of the test set:

In [ ]:
```
E_pocket=find_error(x_test, w_pocket, y_test)
E_pocket
```

In [ ]:
```
E_lin=find_error(x_test, w_lin, y_test)
E_pocket
```

6) Repeat the experiment 100 times with new data set:

In [ ]:
```
x_new,y_new,w_true_new, b_true_new=generate_data(1000)
```

In [ ]:
```
E_pocket_new=zeros(100)
E_lin_new=zeros(100)
w_new=zeros(3)
n=zeros(100)

for i in range (100):

    E_pocket_new[i]=find_error(x_new, w_pocket, y_new)
    E_lin_new[i]=find_error(x_new, w_lin, y_test)

    n[i]=i
```

In [ ]:
```
scatter(n,E_pocket_new*100, c='red')
scatter(n,E_lin_new*100, c="green")
```

In [ ]:
```
scatter(E_pocket_new*100,E_lin_new*100)
```

It can be clearly seen that after repeating experiment 100 times, the algorithm of Linear Regression is more efficient than Pocket. However, both algorithms gives very low error terms (less than 5%).

**Pocket Algorithm**
*Results:*

1. It was seen that Pocket Algorithm generates very close parameters (slope and bias) to the target function.
2. The Error terms were relatively low, no more than 10%.
3. Error term obtained from test set (n=1000) by implemeting weights from training set is identical to the Error term of Linear Regression.

*Problems:*

1. The data is not completely separable, that is why it is impossible to get E_pocket = 0
2. Pocket algorithm is more time consuming than Linear Regression.

**Linear Regression**
*Results:*

1. Efficiently fast and easy calculate weights with low Error terms.
2. Get very close parameters to the target function. However, the bias is much more different. It means that the obtained line is located in a different place than target function. Line of Pocket algorithm is closer.

*Problems:*

1. Calculating pseudo-inverse require matrix being invertible which might not be true for some data sets. Thus, various other function might be implemented, but to find common function which will work for all cases is very hard.
2. Linear Regression is limited to linear relationship. We first assume that classification might be done by estimating a line.

# 2. Gradient Descent for Logistic Regression

In [ ]:
```
from sklearn.datasets import load_iris
iris = load_iris()
```

In [ ]:
```
import random

def train_test_split(data):
    X = iris.data
    y = iris.target

    # random sampling
    train_size = int(0.8*X.shape[0])
    train_ind = random.sample(range(0,150),120) # unique random indices
    X_train = X[train_ind]
    y_train = y[train_ind]

    test_ind = list(set(range(0,150))-set(train_ind))
    X_test = X[test_ind]
    y_test = y[test_ind]

    return X_train, y_train, X_test, y_test
```

In [ ]:
```
X_train, y_train, X_test, y_test = train_test_split(iris)
```

In [ ]:
```
train_samples, d = X_train.shape
test_samples = X_test.shape[0]

train_ind = random.sample(range(0,120),120)
test_ind = random.sample(range(0,30),30)

epochs = 2000
learning_rate = 0.0001

weights = np.zeros(d+1)

E_in = []
E_out = []

for epoch in range(epochs):
    E_in_total = 0
    for ind in train_ind:
        extended_X_train = np.append(X_train[ind],1)
        cross_entropy_error = np.log(1+np.exp(-y_train[ind]*2*np.dot(weights.T, extended_X_train)))
        weights = weights + learning_rate * cross_entropy_error # does not work with minus sign!

        E_in_total += cross_entropy_error
    E_in_average = E_in_total / train_samples
    E_in.append(E_in_average)

    E_out_total = 0
    for ind in test_ind:
        extended_X_test = np.append(X_test[ind],1)
        cross_entropy_error = np.log(1+np.exp(-y_test[ind]*2*np.dot(weights.T, extended_X_test)))

        E_out_total += cross_entropy_error
    E_out_average = E_out_total / test_samples
    E_out.append(E_out_average)
```

In [ ]:
```
figure(figsize=(10,10))
plt.plot(range(epochs), E_in, 'b', label='train')
plt.plot(range(epochs), E_out, 'r', label='test')
plt.legend()
```

**Logistic Regression**

*Results:*

1. By repeating experiment the Error of train set decreases faster (but not always the case) than the Error of test set. It can be explained by that the training set is bigger than test set, so the reduction of Error occures faster.

2. The Error is higher than average Error of Linear Regression and Pocket algorithm.
3. The early stopping would be perfect here because the error converges approximately after 100 epochs

*Limitations:*

1. Might not perform well for very large feature space. We see that the algorithm shows higher Error.

# 3. Practical design of a learning algorithm

In [ ]:
```
from sklearn import datasets
digits = datasets.load_digits()
```

In [ ]:
```
print(digits.images.shape)
print(digits.target.shape)
```

In [ ]:
```
X = digits.data
y = digits.target
```

Error based on Euclidean distance from each point to prediction line:

In [ ]:
```
def find_error_lin(x, w, y):
    E = 0.0
    E=dot(y.T,y)-2*dot(w.T,dot(x.T,y))+dot(dot(w.T, x.T),dot(x,w))
    return E
```

In [ ]:
```
def lin_reg(X_train, X_test, y_train, y_test):
    w = np.linalg.pinv(X_train.T.dot(X_train)).dot(X_train.T).dot(y_train)

    return find_error_lin(X_test, w, y_test)
```

In [ ]:

```
import random

def log_reg(X_train, X_test, y_train, y_test):
    train_samples, d = X_train.shape
    test_samples = X_test.shape[0]

    train_ind = random.sample(range(0,train_samples),train_samples)
    test_ind = random.sample(range(0,test_samples),test_samples)

    epochs = 100
    learning_rate = 0.0001

    weights = np.zeros(d+1)

    E_in = []

    for epoch in range(epochs):
        E_in_total = 0
        for ind in train_ind:
            extended_X_train = np.append(X_train[ind],1)
            cross_entropy_error = np.log(1+np.exp(-y_train[ind]*2*np.dot(weights.T, extended_X_train)))
            weights = weights + learning_rate * cross_entropy_error # does not work with minus sign!

            E_in_total += cross_entropy_error
        E_in_average = E_in_total / train_samples
        E_in.append(E_in_average)

    E_out_total = 0
    for ind in test_ind:
        extended_X_test = np.append(X_test[ind],1)
        cross_entropy_error = np.log(1+np.exp(-y_test[ind]*2*np.dot(weights.T, extended_X_test)))

        E_out_total += cross_entropy_error
    E_out_average = E_out_total / test_samples

    return E_out_average
```

In [ ]:
```
def split_folds(X, y, folds=10):
    fold_size = int(X.shape[0] / folds)
    folds_dict = {}
    X_list = []
    y_list = []
    for i in range(folds-1):
        X_fold = X[fold_size*i : fold_size*i+fold_size]
        y_fold = y[fold_size*i : fold_size*i+fold_size]
        X_list.append(X_fold)
        y_list.append(y_fold)

    # take rest of the data
    X_fold = X[fold_size*(folds-1):]
    y_fold = y[fold_size*(folds-1):]
    X_list.append(X_fold)
    y_list.append(y_fold)

    folds_dict['X'] = X_list
    folds_dict['y'] = y_list

    return folds_dict
```

## 10 fold cross validation

In [ ]:
```
folds = split_folds(X, y)
```

In [ ]:
```
X_folds = np.array(folds['X'])
y_folds = np.array(folds['y'])

E_total_linear = []
E_total_logistic = []

for fold_num in range(len(folds['X'])):
    X_test = X_folds[fold_num]
    y_test = y_folds[fold_num]

    X_temp = np.delete(X_folds, fold_num)
    y_temp = np.delete(y_folds, fold_num)

    X_train = np.concatenate([X_temp[0]])
    for i in range(1,len(X_temp)):
        X_train = np.concatenate([X_train, X_temp[i]])

    y_train = np.concatenate([y_temp[0]])
    for i in range(1,len(y_temp)):
        y_train = np.concatenate([y_train, y_temp[i]])

    E_out_linear = lin_reg(X_train, X_test, y_train, y_test)
    E_total_linear.append(E_out_linear)

    E_out_logistic = log_reg(X_train, X_test, y_train, y_test)
    E_total_logistic.append(E_out_logistic)
```

In [ ]:
```
E_total_linear
```

As we see, the error is pretty variative. That is because we have different data in each cross validation step. The model errors are different for tests on each of the folds by being trained on rest 9 folds. Thus, using cross validation is suggested because it will show more accurate data (an estimate of the generalization error).

In [ ]:
```
E_total_logistic
```

In [ ]:
```
sum(E_total_linear)/len(E_total_linear)
```

In [ ]:
```
sum(E_total_logistic)/len(E_total_logistic)
```

Important note: errors are different because error function are different. Linear regression uses Euclidean distance based error function, and logistic regression uses cross entropy error function.

## 5 fold cross validation

In [ ]:
```
folds = split_folds(X, y, folds=5)
```

In [ ]:
```
X_folds = np.array(folds['X'])
y_folds = np.array(folds['y'])

E_total_linear = []
E_total_logistic = []

for fold_num in range(len(folds['X'])):
    X_test = X_folds[fold_num]
    y_test = y_folds[fold_num]

    X_temp = np.delete(X_folds, fold_num)
    y_temp = np.delete(y_folds, fold_num)

    X_train = np.concatenate([X_temp[0]])
    for i in range(1,len(X_temp)):
        X_train = np.concatenate([X_train, X_temp[i]])

    y_train = np.concatenate([y_temp[0]])
    for i in range(1,len(y_temp)):
        y_train = np.concatenate([y_train, y_temp[i]])

    E_out_linear = lin_reg(X_train, X_test, y_train, y_test)
    E_total_linear.append(E_out_linear)

    E_out_logistic = log_reg(X_train, X_test, y_train, y_test)
    E_total_logistic.append(E_out_logistic)
```

In [ ]:
```
sum(E_total_linear)/len(E_total_linear)
```

In [ ]:
```
sum(E_total_logistic)/len(E_total_logistic)
```

From the observations, we see that the less folds we have, the higher the error is.

## 20 fold cross validation

In [ ]:
```
folds = split_folds(X, y, folds=20)
```

In [ ]:
```
X_folds = np.array(folds['X'])
y_folds = np.array(folds['y'])

E_total_linear = []
E_total_logistic = []

for fold_num in range(len(folds['X'])):
    X_test = X_folds[fold_num]
    y_test = y_folds[fold_num]

    X_temp = np.delete(X_folds, fold_num)
    y_temp = np.delete(y_folds, fold_num)

    X_train = np.concatenate([X_temp[0]])
    for i in range(1,len(X_temp)):
        X_train = np.concatenate([X_train, X_temp[i]])

    y_train = np.concatenate([y_temp[0]])
    for i in range(1,len(y_temp)):
        y_train = np.concatenate([y_train, y_temp[i]])

    E_out_linear = lin_reg(X_train, X_test, y_train, y_test)
    E_total_linear.append(E_out_linear)

    E_out_logistic = log_reg(X_train, X_test, y_train, y_test)
    E_total_logistic.append(E_out_logistic)
```

In [ ]:
```
sum(E_total_linear)/len(E_total_linear)
```

In [ ]:
```
sum(E_total_logistic)/len(E_total_logistic)
```

From the observations, we can clearly see that the more folds we have, the less error is.

## Leave one out cross validation

In [ ]:
```
folds = split_folds(X, y, folds=X.shape[0])
```

In [ ]:
```
X_folds = np.array(folds['X'])
y_folds = np.array(folds['y'])

E_total_linear = []
E_total_logistic = []

for fold_num in range(len(folds['X'])):
    X_test = X_folds[fold_num]
    y_test = y_folds[fold_num]

    X_temp = np.delete(X_folds, fold_num, axis=0)
    y_temp = np.delete(y_folds, fold_num, axis=0)

    X_train = np.concatenate([X_temp[0]])
    for i in range(1,len(X_temp)):
        X_train = np.concatenate([X_train, X_temp[i]])

    y_train = np.concatenate([y_temp[0]])
    for i in range(1,len(y_temp)):
        y_train = np.concatenate([y_train, y_temp[i]])

    E_out_linear = lin_reg(X_train, X_test, y_train, y_test)
    E_total_linear.append(E_out_linear)

    E_out_logistic = log_reg(X_train, X_test, y_train, y_test)
    E_total_logistic.append(E_out_logistic)
```

In [ ]:
```
sum(E_total_linear)/len(E_total_linear)
```

In [ ]:
```
sum(E_total_logistic)/len(E_total_logistic)
```

## Task 2

In [ ]:
```
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import GridSearchCV, validation_curve
```

In [ ]:
```
model = LinearRegression()
model.fit(X, y)
max(cross_val_score(model, X, y, cv=10))
```

In [ ]:
```
C = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
param_grid = { 'C' : C}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=10)
grid.fit(X, y)
print(grid.best_params_)
print(grid.best_score_)
```

In [ ]:
```
train_score, test_score = validation_curve(LogisticRegression(), X, y, "C", C, cv=10)

plt.plot(C, np.median(train_score, 1), color='blue', label='training score')
plt.plot(C, np.median(test_score, 1), color='red', label='validation score')
plt.xscale('log')
plt.legend(loc='best')
plt.xlabel('C (log scaled)')
plt.ylabel('score')
```

In [ ]:
```
C = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
param_grid = { 'C' : C}
grid = GridSearchCV(LinearSVC(), param_grid, cv=10)
grid.fit(X, y)
print(grid.best_params_)
print(grid.best_score_)
```

In [ ]:
```
train_score, test_score = validation_curve(LinearSVC(), X, y, "C", C, cv=10)

plt.plot(C, np.median(train_score, 1), color='blue', label='training score')
plt.plot(C, np.median(test_score, 1), color='red', label='validation score')
plt.xscale('log')
plt.legend(loc='best')
plt.xlabel('C (log scaled)')
plt.ylabel('score')
```

In [ ]:
```
degree = np.arange(10)
param_grid = {'degree' : degree}
grid = GridSearchCV(SVC(kernel='poly'), param_grid, cv=10)
grid.fit(X, y)
print(grid.best_params_)
print(grid.best_score_)
```

In [ ]:
```
train_score, test_score = validation_curve(SVC(kernel='poly'), X, y, "degree", degree, cv=10)

plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(test_score, 1), color='red', label='validation score')
plt.legend(loc='best')
plt.xlabel('degree')
plt.ylabel('score')
```

As we see from experiments above, the results are as follows:

Linear Regression score: 0.6717421507323188

Best performed Logistic Regression score: 0.9376739009460211

Best performed Linear SVM score: 0.9393433500278241

Best performed polynomial kernel SVM score: 0.9788536449638287

So, to conclude, the experiments show the lowest results for Linear Regression (which does not have any parameters), Logistic Regression and Linear SVM are showing approximately same good results, but the best results are obtained by utilizing SVM with polynomial kernel, which is nearly 98%. However, there is a little tradeoff between execution time of each cell above, and their efficiencies: the best performed model, polynomial SVM is being trained a lot more in terms of time relative to other models.

Note: the probable limitation in terms of computation time of each cells above may be large dataset size, with some number of parameters, and cross validations. That means model has to be evaluated that much times. In other words, computation complexity grows as each of the features (cross validation folds, number of parameters, dataset size) grows.

In terms of bias-variance tradeoff, only logistic regression is being overfitted on digits dataset because as we see, the higher C parameter, the lower validation accuracy, which is not seen from training accuracy. And if validation accuracy starts lowering relative to training accuracy as complexity of the model grows, we say that the model has overfitted.

Other two models, which are Support Vector Machines, does not overfit digits data. That may be seen from the two approximately same curves: training and validation. As training grows, validation also grows, and as training lowers, validation also lowers, which is perfect and does not overfit or underfit the data.

Statement: each student has contributed to the project equally by implementing each of the parts individually. After individual implementation, we just compared the results, and have taken the best of the implementations. However, Ainura has contributed more to the report of first two sections, and Anuar has contributed more to the last section of tasks.

In [ ]: