# Code for Final Project Fall 2020

EE 660 Course Project

**Project Type:** (1) Design a system based on real-world data

**Number of student authors:** 1

Yijing Jiang, yijingji@usc.edu

12/06/2020

**Code structure:**

- **[utils.py]**: load data, preprocessing, feature struction
- **[modules.py]**: regression models
- **[main.py]**: load the saved trained model, predict test dataset on it
- **[pretraining.py]**: pre-training process including data split, data plot and models' pre-training
- **[training.py]**: train the regression models, select the best model, train the best model using all the training data and save the parameters

## [utils.py]

```
import numpy as np
import csv
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.decomposition import PCA

#load data from original csv: delete missing data and change cate to num
def csv_to_X_and_y(filename):
    read = pd.read_csv(filename)
    csv_list = read.values.tolist()
    csv_array = np.zeros((1436,10))
    csv_array[:,0:3] = np.array([row[0:3] for row in csv_list])
    csv_array[:,4:10] = np.array([row[4:10] for row in csv_list])
    #delete missing data: nan in 'MetColor' feature
    missing_row_index = []
    for i in range(0,1436):
        if (csv_array[i][5] != 0 and csv_array[i][5] != 1) or not
(csv_array[i][4]<200):
            missing_row_index.append(i)
        #define fuel type as Petrol-3, Diesel-2, other(CNG)-1
        if csv_list[i][3] == 'Petrol': csv_array[i][3] = 3
        elif csv_list[i][3] == 'Diesel': csv_array[i][3] = 2
        else: csv_array[i][3] = 1
    csv_array = np.delete(csv_array, np.array(missing_row_index), 0)
    #D = csv_array[:,1:]
    #y = csv_array[:,0]
    #(num_D, num_feature) = D.shape
    return csv_array

# divide data_delete_missing into pretraining, training and testing sets
def divide_D(filename):
    read = pd.read_csv(filename)
    csv_list = read.values.tolist()
    csv_array = np.array(csv_list)
    D = csv_array[:,2:]
    y = csv_array[:,1]
    #divide the dataset into pretraining, training and test set.
    D_model, D_pt, y_model, y_pt = train_test_split(D, y, test_size = 0.1)
    D_prime, D_test, y_prime, y_test = train_test_split(D_model, y_model,
test_size = 0.2)
    return((D_pt, y_pt), (D_prime, y_prime), (D_test, y_test))

# load dataset from csv
def load(filename):
    read = pd.read_csv(filename)
    csv_list = read.values.tolist()
    csv_array = np.array(csv_list)
    D = csv_array[:,2:]
```

```
    y = csv_array[:,1]
    return (D, y)


#https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
def normalizing(X_train, X_test):
    scaler = MinMaxScaler()
    scaler.fit(X_train)
    X_train_normalized = scaler.transform(X_train)
    X_test_normalized = scaler.transform(X_test)
    return(X_train_normalized, X_test_normalized)

#https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler
def standardizing(X_train, X_test):
    scaler = StandardScaler()
    scaler.fit(X_train)
    X_train_standard = scaler.transform(X_train)
    X_test_standard = scaler.transform(X_test)
    return(X_train_standard, X_test_standard)

#https://towardsdatascience.com/machine-learning-polynomial-regression-with-python-5328e4e8a386
#https://scikit-learn.org/stable/modules/linear_model.html#polynomial-regression-extending-linear-models-with-basis-functions
def polynomial(d, X_train, X_test):
    poly = PolynomialFeatures(degree = d)
    X_train_poly = poly.fit_transform(X_train)
    X_test_poly = poly.fit_transform(X_test)
    return(X_train_poly, X_test_poly)

# use PCA to transform the features(linear dimensionality reduction)
def doPCA(c, train, test):
    pca = PCA(n_components = c)
    pca.fit(train)
    train_PCA = pca.transform(train)
    test_PCA = pca.transform(test)
    return (pca.explained_variance_ratio_, train_PCA, test_PCA)
```

## [modules.py]

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import StratifiedKFold
from itertools import combinations
import utils
from sklearn.linear_model import Ridge, Lasso
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor

# constant model:
def constant(X_train, y_train, X_test, y_test):
    y_pred = np.mean(y_train)
    y_train_pred = np.full(y_train.shape, y_pred)
    y_test_pred = np.full(y_test.shape, y_pred)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return(y_pred, error_train, error_test)

# Baseline for LR: MLE
def MLE(X_train, y_train, X_test, y_test):
    MLE = LinearRegression()
    MLE.fit(X_train, y_train)
    coef = [MLE.coef_, MLE.intercept_] # coef is wi, intercept is w0
    y_train_pred = MLE.predict(X_train)
    y_test_pred = MLE.predict(X_test)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return(coef, y_train_pred, y_test_pred, error_train, error_test)

# MLE + polynomial (d = 2)
def MLE_poly(d, X_train, y_train, X_test, y_test):
    (X_train_poly, X_test_poly) = utils.polynomial(d, X_train, X_test)
    (coef, y_train_pred, y_test_pred, error_train, error_test) =
MLE(X_train_poly, y_train, X_test_poly, y_test)
    return(coef, y_train_pred, y_test_pred, error_train, error_test)

# MLE + PCA (c = 7)
def MLE_pca(c, X_train, y_train, X_test, y_test):
    (pca_ratio, X_train_pca, X_test_pca) = utils.doPCA(c, X_train, X_test)
    (coef, y_train_pred, y_test_pred, error_train, error_test) =
MLE(X_train_pca, y_train, X_test_pca, y_test)
    return(pca_ratio, coef, y_train_pred, y_test_pred, error_train,
error_test)

# Ridge regression (l is the lambda in course;)
def ridge(l, X_train, y_train, X_test, y_test):
    clf = Ridge(alpha=l)
    clf.fit(X_train, y_train)
    coef = [clf.coef_, clf.intercept_]
```

```
    y_train_pred = clf.predict(X_train)
    y_test_pred = clf.predict(X_test)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return(coef, y_train_pred, y_test_pred, error_train, error_test)

# Lasso regression (l is the lambda in course;)
def lasso(l, X_train, y_train, X_test, y_test):
    lasso = Lasso(alpha=l)
    lasso.fit(X_train, y_train)
    coef = [lasso.coef_, lasso.sparse_coef_, lasso.intercept_]
    y_train_pred = lasso.predict(X_train)
    y_test_pred = lasso.predict(X_test)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return(coef, y_train_pred, y_test_pred, error_train, error_test)

#
https://scikit-learn.org/stable/auto_examples/neighbors/plot_regression.html
#sphx-glr-auto-examples-neighbors-plot-regression-py
def knn(k, weights, X_train, y_train, X_test, y_test):
    neigh = KNeighborsRegressor(n_neighbors=k, weights = weights)
    neigh.fit(X_train, y_train)
    y_train_pred = neigh.predict(X_train)
    y_test_pred = neigh.predict(X_test)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return(y_train_pred, y_test_pred, error_train, error_test)

#https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTree
Regressor.html#sklearn.tree.DecisionTreeRegressor
#https://scikit-learn.org/stable/auto_examples/tree/plot_tree_regression.htm
l#sphx-glr-auto-examples-tree-plot-tree-regression-py
def cart(max_leaf_nodes, min_impurity_decrease, X_train, y_train, X_test,
y_test):
    tree = DecisionTreeRegressor(criterion="mse", splitter = "best",
max_leaf_nodes = max_leaf_nodes,
min_impurity_decrease=min_impurity_decrease)
    tree.fit(X_train, y_train)
    y_train_pred = tree.predict(X_train)
    y_test_pred = tree.predict(X_test)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return( tree.get_depth(), tree.get_n_leaves(),
tree.feature_importances_, y_train_pred, y_test_pred, error_train,
error_test)

#https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomFo
restRegressor.html
def random_forest(N_trees, N_samples, max_leaf_nodes, min_impurity_decrease,
X_train, y_train, X_test, y_test):
    rf = RandomForestRegressor(n_estimators = N_trees, criterion = "mse",
max_leaf_nodes=max_leaf_nodes, min_impurity_decrease=min_impurity_decrease,
bootstrap=True, random_state=None, max_samples = N_samples)
```

```
    rf.fit(X_train, y_train)
    y_train_pred = rf.predict(X_train)
    y_test_pred = rf.predict(X_test)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return(rf.feature_importances_, y_train_pred, y_test_pred, error_train,
error_test)

def GDboosting(N_boosting, N_depth, tol, X_train, y_train, X_test, y_test):
    gdb = GradientBoostingRegressor(loss='ls', n_estimators = N_boosting,
max_depth = N_depth, tol = tol, subsample = 1.0)
    gdb.fit(X_train,y_train)
    y_train_pred = gdb.predict(X_train)
    y_test_pred = gdb.predict(X_test)
    error_train = np.sqrt(mean_squared_error(y_train, y_train_pred))
    error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))
    return(gdb.feature_importances_, y_train_pred, y_test_pred, error_train,
error_test)
```

## [main.py]

```
import numpy as np
import utils
import pickle
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error

(X_training, y_training) =
utils.load("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/training.csv")
(X_test, y_test) =
utils.load("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/testing.csv")
(X_training_norm, X_test_norm) = utils.normalizing(X_training, X_test)

#load trained model
#https://machinelearningmastery.com/save-load-machine-learning-models-python
-scikit-learn/
filename =
'/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/final_model.sav'
loaded_model = pickle.load(open(filename, 'rb'))
y_training_pred = loaded_model.predict(X_training_norm)
y_test_pred = loaded_model.predict(X_test_norm)
error_training = np.sqrt(mean_squared_error(y_training, y_training_pred))
error_test = np.sqrt(mean_squared_error(y_test, y_test_pred))

print("Final result with Gradient boosting model:")
print("RMSE on training dataset = " + str(error_training))
print("RMSE on test dataset = " + str(error_test))
print("Features importance: ")
print(loaded_model.feature_importances_)
```

## [pretraining.py]

```
import numpy as np
import pandas as pd
import utils
import modules
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

'''''
split the data and save into files
'''

#create csv files to save the data (delete missing data and convert catogary to numerical)
data_delete_missing = utils.csv_to_X_and_y('/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/cars_missing.csv')
#https://www.geeksforgeeks.org/convert-a-numpy-array-into-a-csv-file/
# convert array into dataframe
DF = pd.DataFrame(data_delete_missing)
# save the dataframe as a csv file
DF.to_csv("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/data_delete_missing.csv")

#create csv files to save pretraining, training, testing data
((D_pt, y_pt), (D_prime, y_prime), (D_test, y_test)) = utils.divide_D("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/data_delete_missing.csv")
pretraining = np.append(np.reshape(y_pt, (125,1)), D_pt, axis=1)
PT = pd.DataFrame(pretraining)
PT.to_csv("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/pretraining.csv")
training = np.append(np.reshape(y_prime, (893,1)), D_prime, axis=1)
TR = pd.DataFrame(training)
TR.to_csv("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/training.csv")
testing = np.append(np.reshape(y_test, (224,1)), D_test, axis=1)
TE = pd.DataFrame(testing)
TE.to_csv("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/testing.csv")

'''''
----------------------------------------------------------------------
Using pre-training dataset to look into the data and try models.
'''
# load pretraining data
(D_pt, y_pt) = utils.load("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/pretraining.csv")


# view the data in standardize form (for LR)
(D_pt, D) = utils.standardizing(D_pt, D_pt)
plt.subplot(331)
plt.scatter(D_pt[:,0], y_pt)
```

```
plt.title('age')
plt.subplot(332)
plt.scatter(D_pt[:,1], y_pt)
plt.title('km')
plt.subplot(333)
plt.scatter(D_pt[:,2], y_pt)
plt.title('Fuel Type')
plt.subplot(334)
plt.scatter(D_pt[:,3], y_pt)
plt.title('Engine Power')
plt.subplot(335)
plt.scatter(D_pt[:,4], y_pt)
plt.title('MetColor')
plt.subplot(336)
plt.scatter(D_pt[:,5], y_pt)
plt.title('Gear type')
plt.subplot(337)
plt.scatter(D_pt[:,6], y_pt)
plt.title('Engine CC')
plt.subplot(338)
plt.scatter(D_pt[:,7], y_pt)
plt.title('Doors')
plt.subplot(339)
plt.scatter(D_pt[:,8], y_pt)
plt.title('weight')
plt.show()


#splite the pretraining data into training and test set
(Dpt_tr, Dpt_test, ypt_tr, ypt_test) = train_test_split(D_pt, y_pt,
test_size = 0.1)

'''''
------------------------------------------------------------
using linear regression related model: all need to standardize the data
'''

#standardize pretraining data for linear model
(Dpt_tr_std, Dpt_test_std) = utils.standardizing(Dpt_tr, Dpt_test)

#MLE baseline with all features:
(coef, y_train_pred, y_test_pred, error_train, error_test) =
modules.MLE(Dpt_tr_std, ypt_tr, Dpt_test_std, ypt_test)
print("MLE")
print(coef, error_train, error_test)
# plot result with feature 0
x= np.array([-2, -1, 0, 1, 2])
y_pred = coef[0][0] * x + coef[1]
plt.plot(x,y_pred, 'r')
plt.scatter(Dpt_test_std[:,0].reshape(-1,1), ypt_test)
plt.show()


```

```
#MLE + one feature: feature 0 performs best
one_feature_error_train = []
one_feature_error_test = []
all_feature_error_train = []
all_feature_error_test = []
for i in range(0,9):
    result = modules.MLE(Dpt_tr_std[:,i].reshape(-1,1), ypt_tr,
Dpt_test_std[:,i].reshape(-1,1), ypt_test)
    one_feature_error_train.append(result[3])
    one_feature_error_test.append(result[4])
    all_feature_error_train.append(error_train)
    all_feature_error_test.append(error_test)
print("MLE for one feature")
print(one_feature_error_train)
print(one_feature_error_test)

plt.plot(all_feature_error_train, 'r', label = 'E_train on MLE with all
feature')
plt.plot(all_feature_error_test, 'b', label = 'E_test on MLE with all
feature')
plt.plot(one_feature_error_train, 'y', label = 'E_train on MLE with one
feature')
plt.plot(one_feature_error_test, 'g', label = 'E_test on MLE with one
feature')
plt.legend()
plt.show()
#MLE + polynomial features: finally choose d = 2 for all features; 0 feature
to plot
#all features: N_pretraining only has 125 data
for d in range(2,5):
    (Dpt_tr_std_poly, Dpt_test_std_poly) = utils.polynomial(d, Dpt_tr_std,
Dpt_test_std)
    (coef, y_train_pred, y_test_pred, error_train, error_test) =
modules.MLE(Dpt_tr_std_poly, ypt_tr, Dpt_test_std_poly, ypt_test)
    print("-----")
    print(d)
    print(coef, error_train, error_test)
#MLE + PCA: choose c = 7
etrain = []
etest = []
for c in range(1,9):
    (pca_ratio, Dpt_tr_std_pca, Dpt_test_std_pca) = utils.doPCA(c,
Dpt_tr_std, Dpt_test_std)
    (coef, y_train_pred, y_test_pred, error_train, error_test) =
modules.MLE(Dpt_tr_std_pca, ypt_tr, Dpt_test_std_pca, ypt_test)
    print("-------")
    print(c)
    print(coef, error_train, error_test)
    etrain.append(error_train)
    etest.append(error_test)

plt.plot([1,2,3,4,5,6,7,8], etrain, 'g', label = "train")
plt.plot([1,2,3,4,5,6,7,8], etest, 'b', label = "test")
plt.legend()
```

```
plt.show()
'''''
#give some constraints on weights to less fit on the noise
# find a range of lamba: lambda larger, stronger regularization, weight will shrink more
# b is sparcity of weight. b smaller, more sparce, centered at (0,0) and axes
'''
# Ridge regression: l = [0.1, 1, 5, 10, 20, 40, 60, 80, 100, 150, 200] choose the best
L = [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 20, 30, 40, 50, 60, 100, 150, 200]
etrain = []
etest = []
for l in L:
    (coef, y_train_pred, y_test_pred, error_train, error_test) = modules.ridge(l, Dpt_tr_std, ypt_tr, Dpt_test_std, ypt_test)
    etrain.append(error_train)
    etest.append(error_test)

plt.plot(np.log10(L), etrain, 'g', label = "train")
plt.plot(np.log10(L), etest, 'b', label = "test")
plt.legend()
plt.show()
# Lasso regression: l = [1,...,1000]
L = [1, 5, 10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
etrain = []
etest = []
for l in L:
    (coef, y_train_pred, y_test_pred, error_train, error_test) = modules.lasso(l, Dpt_tr_std, ypt_tr, Dpt_test_std, ypt_test)
    etrain.append(error_train)
    etest.append(error_test)

plt.plot(np.log10(L), etrain, 'g', label = "train")
plt.plot(np.log10(L), etest, 'b', label = "test")
plt.legend()
plt.show()


#normalize pretraining data for non linear model
(Dpt_tr_norm, Dpt_test_norm) = utils.normalizing(Dpt_tr, Dpt_test)

'''''
------------------------------------------------
Another baseline: KNN
'''
#KNN: weights = ['uniform', 'distance'], k = 1 to 10
for k in range (1,10):
    (y_train_pred, y_test_pred, error_train, error_test) = modules.knn(k, 'distance', Dpt_tr_norm, ypt_tr, Dpt_test_norm, ypt_test)
    print("----")
    print(k)
    print(error_train, error_test)
```

```
'''''
-----------------------------------------------
using ABM model (tree based)
'''
#CART: min_impurity_decrease = [0.001, 0.1, 1, 10, 100, 1000] and plot --- find not overfitting region
#       then try similar leafs around and plot, choose simplist model around +- 1 std Etest
min_impurity_decrease = 10
max_leaf_nodes = 90
result = modules.cart(max_leaf_nodes, min_impurity_decrease, Dpt_tr_norm, ypt_tr, Dpt_test_norm, ypt_test)
print("-----")
print(max_leaf_nodes)
print(result[0])
print(result[1])
print(result[2])
print(result[5:7])
plt.plot(np.arange(90,105,1), etrain, 'b', label = 'Etrain')
plt.plot(np.arange(90,105,1), etest, 'r', label = 'Etest')
plt.legend()
plt.show()

#RF: N_trees=[10, 50, 100, 200, 400, 800, 1000]; N_samples=[2,3,4,5,6,7,8]
# other parameters from best choice of CART
for d in range(3, 9):
    result = modules.random_forest(100, d, 99, 10, Dpt_tr_norm, ypt_tr, Dpt_test_norm, ypt_test)
    print("------")
    print(d)
    print(result[2:4])

#GradiantBoostingRegression: N_boosting = [50, 100, 200, 300, 500, 800, 1000]; N_depth = [1,2,3,5]
N_boosting = 100
N_depth = 2
tol = 1e-4
result = modules.GDboosting(N_boosting, N_depth, tol, Dpt_tr_norm, ypt_tr, Dpt_test_norm, ypt_test)
print("----")
print(result[0])
print(result[3:5])
```

## [training.py]

```
import numpy as np
import pandas as pd
import utils
import modules
import matplotlib.pyplot as plt
from sklearn.model_selection import StratifiedKFold
from mpl_toolkits.mplot3d import Axes3D
import pickle
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.semi_supervised import LabelPropagation
from sklearn.metrics import accuracy_score


# load training data
(X, y) =
utils.load("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/training.csv")

'''''
----------------------------------------------------------------------
Using training dataset to find the best parameters of each modules, and
to choose the best module by comparing between different modules.
'''

# MLE on all features[baseline], 0138 features, all features with poly (d =
2), all features with PCA (c = 7)
T = 10
k = 5
skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
#save the result:(coef, Etrain, Eval)
result_all = []
result_4 = []
result_all_poly = []
result_all_pca = []
for t in range(T):
    for tr_index, v_index in skf.split(X, y):
        #training data
        X_tr = X[tr_index,:]
        y_tr = y[tr_index]
        #validation data
        X_val = X[v_index,:]
        y_val = y[v_index]
        #standardize
        (X_tr_std, X_val_std) = utils.standardizing(X_tr, X_val)
        #MLE on all features:
        result1 = modules.MLE(X_tr_std, y_tr, X_val_std, y_val)
        result_all.append(result1)
        #MLE on four features:
        f = [0,1,3,8]
        result2 = modules.MLE(X_tr_std[:,f], y_tr, X_val_std[:,f], y_val)
        result_4.append(result2)
        #MLE on all features with poly 2:
```

```
            d = 2
            result3 = modules.MLE_poly(d, X_tr_std, y_tr, X_val_std, y_val)
            result_all_poly.append(result3)
            #MLE on all features with pca 7:
            c = 7
            result4 = modules.MLE_pca(c, X_tr_std, y_tr, X_val_std, y_val)
            result_all_pca.append(result4)
    print("-------")
    print("MLE on all features: mean var")
    print("Etrain " + str(np.mean(np.array(result_all)[:,3])) + " " +
    str(np.var(np.array(result_all)[:,3])))
    print("Eval " + str(np.mean(np.array(result_all)[:,4])) + " " +
    str(np.var(np.array(result_all)[:,4])))
    print("-------")
    print("MLE on 0138 features: mean var")
    print("Etrain " + str(np.mean(np.array(result_4)[:,3])) + " " +
    str(np.var(np.array(result_4)[:,3])))
    print("Eval " + str(np.mean(np.array(result_4)[:,4])) + " " +
    str(np.var(np.array(result_4)[:,4])))
    print("-------")
    print("MLE on all features with poly 2: mean var")
    print("Etrain " + str(np.mean(np.array(result_all_poly)[:,3])) + " " +
    str(np.var(np.array(result_all_poly)[:,3])))
    print("Eval " + str(np.mean(np.array(result_all_poly)[:,4])) + " " +
    str(np.var(np.array(result_all_poly)[:,4])))
    print("-------")
    print("MLE on all features with pca 7: mean var")
    print("Etrain " + str(np.mean(np.array(result_all_pca)[:,4])) + " " +
    str(np.var(np.array(result_all_pca)[:,4])))
    print("Eval " + str(np.mean(np.array(result_all_pca)[:,5])) + " " +
    str(np.var(np.array(result_all_pca)[:,5])))

    # Ridge regression: choose best lambda
    L = [0.1, 1, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 80, 100]
    T = 10
    k = 5
    skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
    #save the result:
    result = []
    for t in range(T):
        for tr_index, v_index in skf.split(X, y):
            #training data
            X_tr = X[tr_index,:]
            y_tr = y[tr_index]
            #validation data
            X_val = X[v_index,:]
            y_val = y[v_index]
            #standardize
            (X_tr_std, X_val_std) = utils.standardizing(X_tr, X_val)
            #Ridge regression on all features:
            result_L = []
            for l in L:
                result_l = modules.ridge(l, X_tr_std, y_tr, X_val_std, y_val)
                result_L.append(result_l)
```

```
            result.append(result_L)
    mean_Etrain = np.mean(np.array(result)[:,:,3], axis = 0)
    var_Etrain = np.var(np.array(result)[:,:,3], axis = 0)
    mean_Eval = np.mean(np.array(result)[:,:,4], axis = 0)
    var_Eval = np.var(np.array(result)[:,:,4], axis = 0)
    best_lambda = L[np.argmin(mean_Eval)]
    print("Ridge Regression best lambda: " + str(best_lambda)+"
    "+str(mean_Etrain[np.argmin(mean_Eval)])+"
    "+str(mean_Eval[np.argmin(mean_Eval)]))

    plt.plot(np.log10(L), mean_Etrain, 'b', label = 'Etrain')
    plt.plot(np.log10(L), mean_Eval, 'r', label = 'Eval')
    plt.title("Ridge Regression best lambda: " + str(best_lambda)+"
    "+str(mean_Etrain[np.argmin(mean_Eval)])+"
    "+str(mean_Eval[np.argmin(mean_Eval)]))
    plt.legend()
    plt.show()

    # Lasso regression (l is the lambda in course;)
    L = [1, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 80, 100]
    T = 10
    k = 5
    skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
    #save the result:
    result = []
    for t in range(T):
        for tr_index, v_index in skf.split(X, y):
            #training data
            X_tr = X[tr_index,:]
            y_tr = y[tr_index]
            #validation data
            X_val = X[v_index,:]
            y_val = y[v_index]
            #standardize
            (X_tr_std, X_val_std) = utils.standardizing(X_tr, X_val)
            #Ridge regression on all features:
            result_L = []
            for l in L:
                result_l = modules.lasso(l, X_tr_std, y_tr, X_val_std, y_val)
                result_L.append(result_l)
            result.append(result_L)
    mean_Etrain = np.mean(np.array(result)[:,:,3], axis = 0)
    var_Etrain = np.var(np.array(result)[:,:,3], axis = 0)
    mean_Eval = np.mean(np.array(result)[:,:,4], axis = 0)
    var_Eval = np.var(np.array(result)[:,:,4], axis = 0)
    best_lambda = L[np.argmin(mean_Eval)]
    print("Lasso Regression best lambda: " + str(best_lambda)+"
    "+str(mean_Etrain[np.argmin(mean_Eval)])+"
    "+str(mean_Eval[np.argmin(mean_Eval)]))

    plt.plot(np.log10(L), mean_Etrain, 'b', label = 'Etrain')
    plt.plot(np.log10(L), mean_Eval, 'r', label = 'Eval')
```

```
plt.title("Lasso Regression best lambda: " + str(best_lambda)+"
"+str(mean_Etrain[np.argmin(mean_Eval)])+"
"+str(mean_Eval[np.argmin(mean_Eval)]))
plt.legend()
plt.show()

# KNN (baseline)
N_K = np.arange(1, 10, 1)
weights = ['uniform', 'distance']
T = 10
k = 5
skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
#save the result:
result = []
for t in range(T):
    for tr_index, v_index in skf.split(X, y):
        #training data
        X_tr = X[tr_index,:]
        y_tr = y[tr_index]
        #validation data
        X_val = X[v_index,:]
        y_val = y[v_index]
        #standardize
        (X_tr_norm, X_val_norm) = utils.normalizing(X_tr, X_val)
        #Ridge regression on all features:
        result_L = []
        for w in weights:
            for n in N_K:
                result_l = modules.knn(n, w, X_tr_norm, y_tr, X_val_norm,
y_val)
                result_L.append(result_l)
        result.append(result_L)

mean_Etrain = np.mean(np.array(result)[:,:,2], axis = 0)
var_Etrain = np.var(np.array(result)[:,:,2], axis = 0)
mean_Eval = np.mean(np.array(result)[:,:,3], axis = 0)
var_Eval = np.var(np.array(result)[:,:,3], axis = 0)
best_n_uniform = N_K[np.argmin(mean_Eval[0:9])]
best_n_distance = N_K[np.argmin(mean_Eval[9:18])]
print("KNN_uniform best k : " + str(best_n_uniform)+"
"+str(mean_Etrain[np.argmin(mean_Eval[0:9])])+"
"+str(mean_Eval[np.argmin(mean_Eval[0:9])]))
print("KNN_distance best k : " + str(best_n_distance)+"
"+str(mean_Etrain[9+np.argmin(mean_Eval[9:18])])+"
"+str(mean_Eval[9+np.argmin(mean_Eval[9:18])]))

plt.plot(N_K, mean_Etrain[0:9],  label = 'Etrain_uniform')
plt.plot(N_K, mean_Eval[0:9],   label = 'Eval_uniform')
plt.plot(N_K, mean_Etrain[9:18],   label = 'Etrain_distance')
plt.plot(N_K, mean_Eval[9:18],   label = 'Eval_distance')
plt.title("KNN (weights = 'uniform' or 'distance'; k = 1 to 9)")
plt.legend()
plt.show()

```

```
# CART (baseline for tree based module)
decrease = [1000, 2000, 5000, 6000, 7000, 8000, 9000, 10000, 15000, 20000,
25000, 50000]
T = 10
k = 5
skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
#save the result:
result = []
for t in range(T):
    for tr_index, v_index in skf.split(X, y):
        #training data
        X_tr = X[tr_index,:]
        y_tr = y[tr_index]
        #validation data
        X_val = X[v_index,:]
        y_val = y[v_index]
        #standardize
        (X_tr_norm, X_val_norm) = utils.normalizing(X_tr, X_val)
        #cart on different min_impurity_decrerase
        result_L = []
        for d in decrease:
            result_l = modules.cart(None, d, X_tr_norm, y_tr, X_val_norm,
y_val)
            result_L.append(result_l)
        result.append(result_L)
mean_Etrain = np.mean(np.array(result)[:,:,5], axis = 0)
var_Etrain = np.var(np.array(result)[:,:,5], axis = 0)
mean_Eval = np.mean(np.array(result)[:,:,6], axis = 0)
var_Eval = np.var(np.array(result)[:,:,6], axis = 0)
mean_depth = np.mean(np.array(result)[:,:,0], axis = 0)
mean_leafs = np.mean(np.array(result)[:,:,1], axis = 0)
'''''
'''
best_lambda = L[np.argmin(mean_Eval)]
print("Lasso Regression best lambda: " + str(best_lambda)+"
"+str(mean_Etrain[np.argmin(mean_Eval)])+"
"+str(mean_Eval[np.argmin(mean_Eval)]))
'''''
'''
plt.subplot(221)
plt.plot(np.log10(decrease), mean_Etrain, 'b', label = 'mean_Etrain')
plt.plot(np.log10(decrease), mean_Eval, 'r', label = 'mean_Eval')
plt.legend()
#plt.title("Lasso Regression best lambda: " + str(best_lambda)+"
"+str(mean_Etrain[np.argmin(mean_Eval)])+"
"+str(mean_Eval[np.argmin(mean_Eval)]))
plt.subplot(222)
plt.plot(np.log10(decrease), var_Etrain, 'b', label = 'var_Etrain')
plt.plot(np.log10(decrease), var_Eval, 'r', label = 'var_Eval')
plt.legend()
plt.subplot(223)
plt.plot(np.log10(decrease), mean_depth, label = 'mean_depth')
plt.legend()
plt.subplot(224)
```

```
plt.plot(np.log10(decrease), mean_leafs, label = 'mean_leafs')
plt.legend()
plt.suptitle("CART with different min_impurity_decrease")
plt.show()


decrease = 10**3.5
leafs = np.arange(10,100,2)
T = 10
k = 5
skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
#save the result:
result = []
for t in range(T):
    for tr_index, v_index in skf.split(X, y):
        #training data
        X_tr = X[tr_index,:]
        y_tr = y[tr_index]
        #validation data
        X_val = X[v_index,:]
        y_val = y[v_index]
        #standardize
        (X_tr_norm, X_val_norm) = utils.normalizing(X_tr, X_val)
        #cart on different min_impurity_decrerase
        result_L = []
        for l in leafs:
            result_l = modules.cart(l, decrease, X_tr_norm, y_tr,
X_val_norm, y_val)
            result_L.append(result_l)
        result.append(result_L)
mean_Etrain = np.mean(np.array(result)[:,:,5], axis = 0)
var_Etrain = np.var(np.array(result)[:,:,5], axis = 0)
mean_Eval = np.mean(np.array(result)[:,:,6], axis = 0)
var_Eval = np.var(np.array(result)[:,:,6], axis = 0)
mean_depth = np.mean(np.array(result)[:,:,0], axis = 0)
mean_leafs = np.mean(np.array(result)[:,:,1], axis = 0)
#one standard error of module:
#https://stats.stackexchange.com/questions/80268/empirical-justification-for
-the-one-standard-error-rule-when-using-cross-validat
min_Eval = np.min(mean_Eval)
std_err = np.sqrt(var_Eval[np.argmin(mean_Eval)])/np.sqrt(T*k)
bound_Eval = min_Eval+std_err
simplest_module = 0
while(mean_Eval[simplest_module] > bound_Eval):
    simplest_module = simplest_module + 1
print("simplest module with leafs = " + str(leafs[simplest_module]))
print("Etrain = " + str(mean_Etrain[simplest_module]))
print("Eval = " + str(mean_Eval[simplest_module]))

plt.subplot(221)
plt.plot(leafs, mean_Etrain, 'b', label = 'mean_Etrain')
plt.plot(leafs, mean_Eval, 'r', label = 'mean_Eval')
plt.legend()
```

```
plt.subplot(222)
plt.plot(leafs, var_Etrain, 'b', label = 'var_Etrain')
plt.plot(leafs, var_Eval, 'r', label = 'var_Eval')
plt.legend()
plt.subplot(223)
plt.plot(leafs, mean_depth, label = 'mean_depth')
plt.legend()
plt.subplot(224)
plt.plot(leafs, mean_leafs, label = 'mean_leafs')
plt.legend()
plt.suptitle("CART with different max_leaf_nodes")
plt.show()

# Random Forest: use the choosen tree from CART
decrease = 10**3.5
leafs = 34
N_trees=[50,100, 200, 300, 400, 500]
N_samples= [3,4,5,6,7,8]
T = 10
k = 5
skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
#save the result:
result = []
for t in range(T):
    for tr_index, v_index in skf.split(X, y):
        #training data
        X_tr = X[tr_index,:]
        y_tr = y[tr_index]
        #validation data
        X_val = X[v_index,:]
        y_val = y[v_index]
        #standardize
        (X_tr_norm, X_val_norm) = utils.normalizing(X_tr, X_val)
        #cart on different min_impurity_decrerase
        result_L = []
        for j in N_samples:
            for i in N_trees:
                result_ij = modules.random_forest(i, j, leafs, decrease,
X_tr_norm, y_tr, X_val_norm, y_val)
                result_L.append(result_ij)
        result.append(result_L)

mean_Etrain = np.mean(np.array(result)[:,:,3], axis = 0)
var_Etrain = np.var(np.array(result)[:,:,3], axis = 0)
mean_Eval = np.mean(np.array(result)[:,:,4], axis = 0)
var_Eval = np.var(np.array(result)[:,:,4], axis = 0)
best = np.argmin(mean_Eval)
print("----")
print(best)
print("mean_Etrain: " + str(mean_Etrain[best]))
print("mean_Etest: " + str(mean_Eval[best]))
print("var_Etrain: " + str(var_Etrain[best]))
print("var_Etest: " + str(var_Eval[best]))
```

```
mean_Etr = np.reshape(mean_Etrain, (6,6))
var_Etr = np.reshape(var_Etrain, (6,6))
mean_Ete = np.reshape(mean_Eval, (6,6))
var_Ete = np.reshape(var_Eval, (6,6))


plt.figure(0)
for i in range(3):
    for j in range(2):
        ax = plt.subplot2grid((3,2), (i,j))
        if j == 0:
            ax.plot(N_trees, mean_Etr[:,i].reshape(-1,1),
label='mean_Etrain')
            ax.plot(N_trees, mean_Ete[:,i].reshape(-1,1), label='mean_Eval')
            plt.legend()
        else:
            ax.plot(N_trees, var_Etr[:,i].reshape(-1,1), label='var_Etrain')
            ax.plot(N_trees, var_Ete[:,i].reshape(-1,1), label='var_Eval')
            plt.legend()
plt.show()

plt.figure(1)
for i in range(3,6):
    for j in range(2):
        ax = plt.subplot2grid((3,2), (i-3,j))
        if j == 0:
            ax.plot(N_trees, mean_Etr[:,i].reshape(-1,1),
label='mean_Etrain')
            ax.plot(N_trees, mean_Ete[:,i].reshape(-1,1), label='mean_Eval')
            plt.legend()
        else:
            ax.plot(N_trees, var_Etr[:,i].reshape(-1,1), label='var_Etrain')
            ax.plot(N_trees, var_Ete[:,i].reshape(-1,1), label='var_Eval')
            plt.legend()
plt.show()

# Gradiant Boosting Regression:
N_estimate=[10, 100, 300, 400, 500, 600, 700, 800, 2000]
N_depth=1
tol = 1e-4
T = 10
k = 5
skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
#save the result:
result = []
for t in range(T):
    for tr_index, v_index in skf.split(X, y):
        #training data
        X_tr = X[tr_index,:]
        y_tr = y[tr_index]
        #validation data
        X_val = X[v_index,:]
```

```
            y_val = y[v_index]
            #standardize
            (X_tr_norm, X_val_norm) = utils.normalizing(X_tr, X_val)
            #cart on different min_impurity_decrerase
            result_L = []
            for n in N_estimate:
                result_n = modules.GDboosting(n, N_depth, tol, X_tr_norm, y_tr,
    X_val_norm, y_val)
                result_L.append(result_n)
            result.append(result_L)

    mean_Etrain = np.mean(np.array(result)[:,:,3], axis = 0)
    var_Etrain = np.var(np.array(result)[:,:,3], axis = 0)
    mean_Eval = np.mean(np.array(result)[:,:,4], axis = 0)
    var_Eval = np.var(np.array(result)[:,:,4], axis = 0)
    # one standard error
    min_Eval = np.min(mean_Eval)
    std_err = np.sqrt(var_Eval[np.argmin(mean_Eval)])/np.sqrt(T*k)
    bound_Eval = min_Eval+std_err
    simplest_module = 0
    while(mean_Eval[simplest_module] > bound_Eval):
        simplest_module = simplest_module + 1
    print("simplest module with boosting = " + str(N_estimate[simplest_module]))
    print("mean_Etrain = " + str(mean_Etrain[simplest_module]))
    print("mean_Eval = " + str(mean_Eval[simplest_module]))
    print("var_Etrain = " + str(var_Etrain[simplest_module]))
    print("var_Eval = " + str(var_Eval[simplest_module]))

    plt.subplot(121)
    plt.plot(np.log10(N_estimate), mean_Etrain, label='mean_Etrain')
    plt.plot(np.log10(N_estimate), mean_Eval, label='mean_Eval')
    plt.legend()
    plt.subplot(122)
    plt.plot(np.log10(N_estimate), var_Etrain, label='var_Etrain')
    plt.plot(np.log10(N_estimate), var_Eval, label='var_Eval')
    plt.legend()
    plt.show()


    '''''
    ----------------------------------------------------------------------
    Using training dataset to choose the best module by comparing between
    different modules.
    '''

    # using cross-validation to compare between different modules:
    T = 10
    k = 5
    skf = StratifiedKFold(n_splits=k, shuffle=True,random_state=None)
    #save the result:
    result_const = []
    result_MLE = []
    result_MLE_pca = []
```

```
result_ridge = []
result_lasso = []
result_knn_uniform = []
result_knn_dist = []
result_CART = []
result_RF = []
result_boosting = []
for t in range(T):
    for tr_index, v_index in skf.split(X, y):
        #training data
        X_tr = X[tr_index,:]
        y_tr = y[tr_index]
        #validation data
        X_val = X[v_index,:]
        y_val = y[v_index]

        #####Constant module#####
        result0 = modules.constant(X_tr, y_tr, X_val, y_val)
        result_const.append(result0)

        #####Linear Regression Module#####
        #standardize
        (X_tr_std, X_val_std) = utils.standardizing(X_tr, X_val)
        #MLE on all features[baseline]
        result1 = modules.MLE(X_tr_std, y_tr, X_val_std, y_val)
        result_MLE.append(result1)
        #MLE with pca = 7
        c = 7
        result2 = modules.MLE_pca(c, X_tr_std, y_tr, X_val_std, y_val)
        result_MLE_pca.append(result2)
        #Ridge Regression with lambda = 25
        l1 = 25
        result3 = modules.ridge(l1, X_tr_std, y_tr, X_val_std, y_val)
        result_ridge.append(result3)
        #Lasso Regression with lambda = 35
        l2 = 35
        result4 = modules.lasso(l2, X_tr_std, y_tr, X_val_std, y_val)
        result_lasso.append(result4)


        #####Other Regreession Module#####
        #normalize
        (X_tr_norm, X_val_norm) = utils.normalizing(X_tr, X_val)
        #KNN (uniform) with k = 3
        k1 = 3
        w1 = 'uniform'
        result5 = modules.knn(k1, w1, X_tr_norm, y_tr, X_val_norm, y_val)
        result_knn_uniform.append(result5)
        #KNN (distance) with k = 4
        k2 = 4
        w2 = 'distance'
        result6 = modules.knn(k2, w2, X_tr_norm, y_tr, X_val_norm, y_val)
        result_knn_dist.append(result6)
```

```
            #CART with max_leaf_nodes = 34, min_impurity_decrease = 10^3.5
            leafs = 34
            decrease = 10**3.5
            result7 = modules.cart(leafs, decrease, X_tr_norm, y_tr, X_val_norm,
    y_val)
            result_CART.append(result7)
            #Random Forest with N_trees = 500, N_samples = 6
            trees = 500
            samples = 6
            result8 = modules.random_forest(trees, samples, leafs, decrease,
    X_tr_norm, y_tr, X_val_norm, y_val)
            result_RF.append(result8)
            #Gradian Boosting Regression with boosting = 400, depth = 1
            boosting = 400
            depth = 1
            tol = 1e-4
            result9 = modules.GDboosting(boosting, depth, tol, X_tr_norm, y_tr,
    X_val_norm, y_val)
            result_boosting.append(result9)

    #Constant Model[baseline]
    print("-------")
    print("Constant Model: ")
    print(" Etrain[mean var] " + str(np.mean(np.array(result_const)[:,1])) + " "
    + str(np.var(np.array(result_const)[:,1])))
    print(" Eval[mean var] " + str(np.mean(np.array(result_const)[:,2])) + " " +
    str(np.var(np.array(result_const)[:,2])))
    print(" constant of best Eval: " )
    print(np.array(result_const)[np.argmin(np.array(result_const)[:,2])][0])
    #MLE on all features[baseline]
    print("-------")
    print("MLE on all features: ")
    print(" Etrain[mean var] " + str(np.mean(np.array(result_MLE)[:,3])) + " " +
    str(np.var(np.array(result_MLE)[:,3])))
    print(" Eval[mean var] " + str(np.mean(np.array(result_MLE)[:,4])) + " " +
    str(np.var(np.array(result_MLE)[:,4])))
    print(" coef of best Eval: " )
    print(np.array(result_MLE)[np.argmin(np.array(result_MLE)[:,4])][0])
    #MLE with pca = 7
    print("-------")
    print("MLE with PCA (c = 7): ")
    print(" Etrain[mean var] " + str(np.mean(np.array(result_MLE_pca)[:,4])) + "
    " + str(np.var(np.array(result_MLE_pca)[:,4])))
    print(" Eval[mean var] " + str(np.mean(np.array(result_MLE_pca)[:,5])) + " "
    + str(np.var(np.array(result_MLE_pca)[:,5])))
    print(" coef of best Eval: " )
    print(np.array(result_MLE_pca)[np.argmin(np.array(result_MLE_pca)[:,5])][1])
    print(" pca_ration: ")
    print(np.array(result_MLE_pca)[np.argmin(np.array(result_MLE_pca)[:,5])][0])
    #Ridge Regression with lambda = 25
    print("-------")
    print("Ridge Regression with lambda = 25: ")
    print(" Etrain[mean var] " + str(np.mean(np.array(result_ridge)[:,3])) + " "
    + str(np.var(np.array(result_ridge)[:,3])))
```

```
print(" Eval[mean var] " + str(np.mean(np.array(result_ridge)[:,4])) + " " + str(np.var(np.array(result_ridge)[:,4])))
print(" coef of best Eval: " )
print(np.array(result_ridge)[np.argmin(np.array(result_ridge)[:,4])][0])
#Lasso Regression with lambda = 35
print("-------")
print("Lasso Regression with lambda = 35: ")
print(" Etrain[mean var] " + str(np.mean(np.array(result_lasso)[:,3])) + " " + str(np.var(np.array(result_lasso)[:,3])))
print(" Eval[mean var] " + str(np.mean(np.array(result_lasso)[:,4])) + " " + str(np.var(np.array(result_lasso)[:,4])))
print(" coef of best Eval: " )
print(np.array(result_lasso)[np.argmin(np.array(result_lasso)[:,4])][0])
#KNN (uniform) with k = 3
print("-------")
print("KNN (uniform) with k = 3: ")
print(" Etrain[mean var] " + str(np.mean(np.array(result_knn_uniform)[:,2])) + " " + str(np.var(np.array(result_knn_uniform)[:,2])))
print(" Eval[mean var] " + str(np.mean(np.array(result_knn_uniform)[:,3])) + " " + str(np.var(np.array(result_knn_uniform)[:,3])))
#KNN (distance) with k = 4
print("-------")
print("KNN (distance) with k = 4: ")
print(" Etrain[mean var] " + str(np.mean(np.array(result_knn_dist)[:,2])) + " " + str(np.var(np.array(result_knn_dist)[:,2])))
print(" Eval[mean var] " + str(np.mean(np.array(result_knn_dist)[:,3])) + " " + str(np.var(np.array(result_knn_dist)[:,3])))
#CART with max_leaf_nodes = 34, min_impurity_decrease = 10^3.5
print("-------")
print("CART with max_leaf_nodes = 34, min_impurity_decrease = 10^3.5: ")
print(" Etrain[mean var] " + str(np.mean(np.array(result_CART)[:,5])) + " " + str(np.var(np.array(result_CART)[:,5])))
print(" Eval[mean var] " + str(np.mean(np.array(result_CART)[:,6])) + " " + str(np.var(np.array(result_CART)[:,6])))
print(" feature importance of best Eval: " )
print(np.array(result_CART)[np.argmin(np.array(result_CART)[:,6])][2])
#Random Forest with N_trees = 500, N_samples = 6
print("-------")
print("Random Forest with N_trees = 500, N_samples = 6: ")
print(" Etrain[mean var] " + str(np.mean(np.array(result_RF)[:,3])) + " " + str(np.var(np.array(result_RF)[:,3])))
print(" Eval[mean var] " + str(np.mean(np.array(result_RF)[:,4])) + " " + str(np.var(np.array(result_RF)[:,4])))
print(" feature importance of best Eval: " )
print(np.array(result_RF)[np.argmin(np.array(result_RF)[:,4])][0])
#Gradian Boosting Regression with boosting = 400, depth = 1
print("-------")
print("Gradian Boosting Regression with boosting = 400, depth = 1: ")
print(" Etrain[mean var] " + str(np.mean(np.array(result_boosting)[:,3])) + " " + str(np.var(np.array(result_boosting)[:,3])))
print(" Eval[mean var] " + str(np.mean(np.array(result_boosting)[:,4])) + " " + str(np.var(np.array(result_boosting)[:,4])))
print(" feature importance of best Eval: " )
```

```
print(np.array(result_boosting)[np.argmin(np.array(result_boosting)[:,4])][0
])


# train the final module with whole training data
(X_test, y_test) =
utils.load("/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/testing.csv")
(X_training_norm, X_test_norm) = utils.normalizing(X, X_test)

boosting = 400
depth = 1
tol = 1e-4
model = GradientBoostingRegressor(loss='ls', n_estimators = boosting,
max_depth = depth, tol = tol, subsample = 1.0)
model.fit(X_training_norm,y)

# save the model to disk
#
https://machinelearningmastery.com/save-load-machine-learning-models-python-
scikit-learn/
filename =
'/Users/mac-pro/Desktop/20Fall/EE660/HW/Final/code/final_model.sav'
pickle.dump(model, open(filename, 'wb'))
```