Boston House Price Forcast

# Libraries

import numpy as np

import matplotlib as mpl

import warnings

warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt

import pandas as pd

import scipy.stats as st

import seaborn as sns

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import GridSearchCV

from sklearn.metrics import r2\_score

from sklearn.datasets import load\_boston

from sklearn.linear\_model import RidgeCV, LassoCV, LinearRegression,ElasticNet

from sklearn.svm import SVR

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from xgboost import XGBRegressor

# Get Data

boston = load\_boston()

x= boston.data

y= boston.target

# Splitting Data

x\_train,x\_test,y\_train,y\_test = train\_test\_split(x,y,test\_size=0.2,random\_state=28)

ss= StandardScaler()

x\_train = ss.fit\_transform(x\_train)

x\_test = ss.transform(x\_test)

# Create & Evaluate model

#Set the model name.

names = ['LinerRegression',

'Ridge',

'Lasso',

'Random Forrest',

'GBDT',

'Support Vector Regression',

'ElasticNet',

'XgBoost']

#Define the model.

# cv is the cross-validation idea here.

models = [LinearRegression(),

RidgeCV (alphas=(0.001,0.1,1),cv=3),

LassoCV (alphas=(0.001,0.1,1),cv=5),

RandomForestRegressor(n\_estimators=10),

GradientBoostingRegressor(n\_estimators=30),

SVR(),

ElasticNet(alpha=0.001,max\_iter=10000),

XGBRegressor()]

# Output the R2 scores of all regression models.

#Define the R2 scoring function.

def R2(model,x\_train, x\_test, y\_train, y\_test):

model\_fitted = model.fit(x\_train,y\_train)

y\_pred = model\_fitted.predict(x\_test)

score = r2\_score(y\_test, y\_pred)

return score

#Traverse all models to score.

for name,model in zip(names,models):

score = R2(model,x\_train, x\_test, y\_train, y\_test)

print("{}: {:.6f}, {:.4f}". format (name,score.mean() ,score.std()))

# Use GridSearch to adjust hyperparemeters

## --- For example : SVR() ---

parameters = {

'kernel': ['linear', 'rbf'],

'C': [0.1, 0.5,0.9,1,5],

'gamma': [0.001,0.01,0.1,1]

}

#Use grid search and perform cross validation.

model = GridSearchCV(SVR(), param\_grid=parameters, cv=3)

model.fit(x\_train, y\_train)

print("Optimal parameter list:", xgb\_grid.best\_params\_)

print("Optimal model:", xgb\_grid. best\_estimator\_)

print("Optimal R2 value:", xgb\_grid.best\_score\_)

# Plotting Forcast

##Perform visualization.

ln\_x\_test = range(len(x\_test))

y\_predict = xgb\_grid.predict (x\_test)

#Set the canvas.

plt.figure (figsize=(16,8), facecolor='w')

#Draw with a red solid line.

plt.plot (ln\_x\_test, y\_test, 'r-', lw=2, label=u'Value')

#Draw with a green solid line.

plt.plot (ln\_x\_test, y\_predict, 'g-', lw = 3, label=u'Estimated value of the SVR algorithm, $RA2$=%.3f' %

(xgb\_grid.best\_score\_))

#Display in a diagram.

plt.legend (loc ='upper left')

plt.grid(True)

plt.title(u"Boston Housing Price Forecast Xgboost")

plt.xlim(0, 101)

plt.show()

Detail of linear regression

# --- Read Data --

#lmport the required modules, numpy for calculation, and Matplotlib for drawing

import numpy as np

import matplotlib.pyplot as plt

#This code is for jupyter Notebook only

%matplotlib inline

# define data, and change list to array

x = [3,21,22,34,54,34,55,67,89,99]

x = np.array(x)

y = [1,10,14,34,44,36,22,67,79,90]

y = np.array(y)

#Show the effect of a scatter plot

plt.scatter (x,y)

# --- Build The Linear Regression Model ---

#The basic linear regression model is wx+ b, and since this is a two-dimensional space, the model is

#ax+ b

def model(a, b, x):

return a\*x + b

#The most commonly used loss function of linear regression model is the loss function of mean

#variance difference

def loss\_function(a, b, x, y):

num = len(x)

prediction=model(a,b,x)

return (0.5/num) \* (np.square(prediction-y)).sum()

#The optimization function mainly USES partial derivatives to update two parameters a and b

def optimize(a,b,x,y):

num = len(x)

prediction = model(a,b,x)

#Update the values of A and B by finding the partial derivatives of the loss function on a and b

da = (1.0/num) \* ((prediction -y)\*x).sum()

db = (1.0/num) \* ((prediction -y).sum())

a=a -Lr\*da

b = b - Lr\*db

return a, b

#iterated function, return a and b

def iterate (a,b,x,y,times):

for i in range(times):

a,b = optimize (a,b,x,y)

return a,b

# --- Plotting –

#lInitialize parameters and display

a = np.random.rand(1)

print(a)

b = np.random.rand(1)

print(b)

Lr = 1e-4

#For the first iteration, the parameter values, losses, and visualization after the iteration are displayed

a,b = iterate(a,b,x,y,1)

prediction=model(a,b,x)

loss = loss\_function(a, b, x, y)

print(a,b,loss)

plt.scatter (x,y)

plt.plot(x,prediction)

Decision Tree Details

TensorFlow Basics

import tensorflow as tf

const\_a = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32)

tf.zeros(shape=[2, 3], dtype=tf.int32)

zeros\_like\_c = tf.zeros\_like(const\_a)

fill\_d = tf.fill([3,3], 8)

random\_e = tf.random.normal([5,5],mean=0,stddev=1.0, seed = 1)

#Create a list.

list\_f = [1,2,3,4,5,6]

tensor\_f = tf.convert\_to\_tensor(list\_f, dtype=tf.float32)

var\_1 = tf.Variable(tf.ones([2,3]))

var\_1

#Create a 4-dimensional tensor. The tensor contains four images. The size of each image is 100x100x3

tensor\_h = tf.random.normal([4,100,100,3])

#Obtain the pixel in the position [20,40] in the second channel of the first image.

tensor\_h[0][19][39][1]

#Extract the first, second, and fourth images from tensor\_h ([4,100,100,3]).

indices = [0,1,3]

tf.gather(tensor\_h,axis=0,indices=indices)

#Extract the pixel in [1,1] from the first dimension of the first image and the pixel in [2,2] from the

#first dimension of the second image in tensot\_h ([4,100,100,3]).

indices = [[0,1,1,0],[1,2,2,0]]

tf.gather\_nd(tensor\_h,indices=indices)

# Display Dimension

const\_d\_1 = tf.constant([[1, 2, 3, 4]],shape=[2,2], dtype=tf.float32)

print(const\_d\_1.shape)

print(const\_d\_1.get\_shape())

reshape\_1 = tf.constant([[1,2,3],[4,5,6]]) # 2\*3

tf.reshape(reshape\_1, (3,2))

#Generate a 100 x 100 x 3 tensor to represent a 100 x 100 three-channel color image.

expand\_sample\_1 = tf.random.normal([100,100,3], seed=1)

print("add a dimension before the first dimension (axis = 0): ",tf.expand\_dims(expand\_sample\_1,

axis=0) shape)

#Generate a 100 x 100 x 3 tensor to represent a 100 x 100 three-channel color image.

squeeze\_sample\_1 = tf.random.normal([1,100,100,3])

print("size of the original data:",squeeze\_sample\_1.shape)

squeezed\_sample\_1 = tf.squeeze(expand\_sample\_1)

transposed\_sample\_1 = tf.transpose(trans\_sample\_1)

# perm value is [0,2,1,3]. Perm is a parameter# initial data [4,100,200,3]

transposed\_sample\_2 = tf.transpose(trans\_sample\_2,[0,2,1,3])

Size of the transposed data: (4, 200, 100, 3)

# --- Operation on Tensors ---

a = tf.constant([[8, 5],

b = tf.constant([[1, 6],

tf.add(a, b) # Addition

tf.matmul (a,b) # Multiplication

tf.substract() ; tf.pow() ; tf.divide() ;

# --- Tensor Concatenation & Splitting ---

stack\_sample\_1 = tf.random.normal([100,100,3])

stack\_sample\_2 = tf.random.normal([100,100,3])

print("sizes of the original data: ",stack\_sample\_1.shape, stack\_sample\_2.shape)

#Dimensions increase after the concatenation. If axis is set to 0, a dimension is added before the first

# dimension.

stacked\_sample\_1 = tf.stack([stack\_sample\_1, stack\_sample\_2],axis=0)

print("size of the concatenated data:",stacked\_sample\_1.shape)

Output:

Sizes of the original data: (100, 100, 3) (100, 100, 3)

Size of the concatenated data: (2, 100, 100, 3)

concat\_sample\_1 = tf.random.normal([4,100,100,3])

concat\_sample\_2 = tf.random.normal([40,100,100,3])

concated\_sample\_1 = tf.concat([concat\_sample\_1,concat\_sample\_2],axis=0) #(4, 100, 100, 3) (40, 100, # 100, 3)

# --- Tensor Spliting ---

split\_sample\_1 = tf.random.normal([10,100,100,3])

splited\_sample\_2 = tf.split(split\_sample\_1, num\_or\_size\_splits=[3,5,2],axis=0)

np.shape(splited\_sample\_2[0]), # (3, 100, 100, 3)

np.shape(splited\_sample\_2[1]), # (5, 100, 100, 3)

np.shape(splited\_sample\_2[2])) # (2,100, 100, 3)

# --- Tensor Sorting ---

sorted\_sample\_1 = tf.sort(sort\_sample\_1, direction="ASCENDING") # [1879654230]

sorted\_sample\_1.numpy() # [012345678 9]

sorted\_sample\_2 = tf.argsort(sort\_sample\_1,direction="ASCENDING")

sorted\_sample\_2.numpy() # [907865421 3] indexes

# Get top 5

values, index = tf.nn.top\_k(sort\_sample\_1,5)

values.numpy() # [9 8 7 6 5]

index.numpy() # [3 1 2 4 5]