Proje-1: Regresyon

Bu rapor Notion kullanarak yazılmıştır. Buradan Notion sayfasına ulaşabilir ve raporu daha düzenli okuyabilirsiniz.

Link : https://dawn-squash-710.notion.site/Proje-1-Regresyon-08c0dec28ff349439db3dfe26f1b0b5d

Tüm işlemler google colab kullanarak yapıldı. Kodlara ve .ipynb dosyasına ulaşmak için :

Link : https://github.com/Pilestin/My_ML_Adventure/tree/master/Makine Öğrenmesine Giriş/Proje 1 - Regresyon

Veri Seti : `https://www.kaggle.com/datasets/harlfoxem/housesalesprediction`

a) *Yaşam alanı (sqft_living) ve arsa (sqft_lot) özniteliklerini (feature, attribute) kullanarak evin fiyatını tahmin edecek basit bir regresyon modeli geliştiriniz.*

- İlk olarak yapmamız gerekenler gerekli kütüphaneleri import etmek olacak. Tabi ki modelimi eğiteceğimiz algoritmamızı kendimiz yazacağız. Fakat verileri uygun veri yapılarında tutmayı, görselleştirmeyi veya arçalamayı uygun kütüphaneleri kullanarak yapacağız.

Başlangıç olarak nump, pandas ve maplotlib kütüphanelerini import ettik. Yeri geldiğinde bir kaç import işlemi daha yapılacak

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Verilerimizi .csv dosyasından almamız gerek. Bunun için pandas kütüphanesinden read_csv metodunu kullanıyoruz. Buradan bir pandas data frame tipinde nesne dönecektir ve bu nesne tüm tabloyu tutmakta.

İlk olarak bu data frame nesnesinin shape(boyut) bilgisine bakıyoruz.

Ardından head ile ilk 5 satırı görmekteyiz.

```
data_frame = pd.read_csv("kc_house_data.csv")
print(data_frame.shape)
data_frame.head()
```

```
(21613, 21)
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_basement	yr_built	yr_renc
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...	7	1180	0	1955	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...	7	2170	400	1951	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...	6	770	0	1933	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	...	7	1050	910	1965	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	...	8	1680	0	1987	

5 rows × 21 columns

- Verilerimizi inceledikten sonra artık train ve test olarak ayırma işlemine geçebiliriz.

Öncelikle ayırma işlemi için sklearn.model_selection içerisinden train_test_split metodunu import ettik. X 'e attribute'larımızı aldık. Y ise çıkış değeri olan fiyat biligisini tutacak.

Bu metodun içerisine X , Y ve train_size bilgisini verip verilerimizi ayırdık.

x_train ve y_train ile verilerimizi eğğiteceğiz. x_test ve y_test ile ise doğrulama ve maliyet değerlerimizi kontrol edeceğiz.

Burada x ve y parçalarımızın her biri pandas Dataframe tipinde.

```
from sklearn.model_selection import train_test_split


X = data_frame[['sqft_living','sqft_lot']]
Y = data_frame[["price"]]
x_train, x_test, y_train, y_test =  train_test_split(X,Y,train_size=0.7) # %70 ini test için ayır.

print(len(x_train.values))
```

- Sıradaki işlemimize geçmeden önce verilerimize normalizasyon uygulamalıyız. Aksi halde verilerimizi eğitmemiz çok daha uzun sürecektir.

Bu işlemi kendimiz de yazabilirdik fakat burada hazır bulunan sklearn.preprocessing içerisindeki MinMaxScaler kullanımı tercih edildi.

Arkaplanda aşağıdaki formül her x değeri için uygulanmakta.

$$(X - min(x))/(max(x) - mix(x))$$

Normalize edilen veirler ise tekrar x_train ve x_test değişkenlerine verildi.

Şu anda bu değişkenler **numpy.ndaray** tipinde. Bunların üzerinde işlem yapmamız artık daha kolay olacak.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(x_train)
# x train scale edilecek. Bu yüzden uygun hale getirildi.

x_train = scaler.transform(x_train)
x_test  = scaler.transform(x_test)


print(type(y_train)) # pandas tipi
type(x_train)        # numpy tipi
```

```
<class 'pandas.core.frame.DataFrame'>
numpy.ndarray
```

X verileri

	sqft_living	sqft_lot
0	1180	5650
1	2570	7242
2	770	10000
3	1960	5000
4	1680	8080
...	...	...
21608	1530	1131
21609	2310	5813
21610	1020	1350
21611	1600	2388
21612	1020	1076

Y verileri

	price
0	221900.0
1	538000.0
2	180000.0
3	604000.0
4	510000.0
...	...
21608	360000.0
21609	400000.0
21610	402101.0
21611	400000.0
21612	325000.0

- Şimdi elimizde hazır verilerimiz bulunduğuna göre kendi fonksiyonlarımızı yazabiliriz. İlk olarak maliyet fonksiyonumuzu görelim.

Maliyet Fonksiyonu - 1

- Maliyet fonksiyonumuz x ve y aralarındaki hata oranını bize göstermekteydi. Ayrıca optimum'a ne kadar yaklaştığımızı bulabilmekteydik.

1. soru için J fonksiyonumuz sadece Q0 , Q1 , Q2 değerleri ile çalışacağı için direk bu parametreleri alabilir. Fakat 2. soruda bunları bir liste içerisinde tutacağız.

Bu fonksiyon ilk olarak bir toplam değişkeni belirlemekte. Ardından X 'in boyutu kadar döngüde parantez içerisinde verilen işlemi uygulamakta.

Bu işlem aslında Hq(x) hipotez değerimizdir. Bunu 2. soruda ayrı bir fonksiyon olarak ayıracağız fakat şu an için bu şekilde devam edilecek.

Bu hipotez ile doğrusal bir eğri uydurmuş ve bu eğri ile gerçek Y değeri arasındaki farkı bulmuş oluruz. Yani hatamız. Ardından bu değerlerin her satır için bulunup karelerini alarak toplamaktayız.

Son olarak ise 1/2m yani veri sayısının 2 katına sonucumuzu bölmekteyiz.

Maliyet fonksiyonumuz tamamlandı.

```
def J(X, Y, Q0=1, Q1=1, Q2=1 ):
    """ VERİLEN Q DEĞERLERİ İÇİN TÜM TABLOYU DENER VE GERÇEK Y İLE FARKIN KARELERİNİ TOPLAR. BÖYLECE MALİYETİ ÖLÇER """
    # X : x_test.values
        # X[i][0] : sqft_living
        # X[i][1] : sqft_lot

    # Y : y_test.values
    # hx = Q0 + Q1*x1 + Q2x2

    # print("Ben J : ",Q0,Q1,Q2)
    total = 0
    m = len(X)
    for i in range(m):
        total = total + ( Q0 + Q1*X[i][0] + Q2*X[i][1] - Y[i][0])**2
    return 1/(2*m) * total
```

- Şimdi gradient descent içerisinde kullandığımız maliyet fonksiyonunun türevini yapan fonksiyonu yazabiliriz.

Burada gördüklerimiz de yukarıda yaptığımız işlemlerin neredeyse aynısı. Farkımız türev alındığı için kare bölümü çarpan olarak gelip 1/2m değerini 1/m 'e çevirmekte.

Ayrıca parantez içerisinin de türevi alınacağı için (Dikkat Q'ya göre türev alınmakta. Bilinmeyen Q) Q'ların önündeki katsayılar çarpan olarak gelmekte.

Son olarakta toplam değer 1/m ile çarpılmakta.

```
def J_derivate(X, Y, Q0=1, Q1=1, Q2=1, k=1 ):
    """GRADİENT D. ALGORİTMASI İÇİN MALİYETİN, TÜREVİNİN ALINDIKTAN SONRA, HESAPLANDIĞI KISIM"""
    # X : x_test.values
        # X[i][0] : sqft_living
        # X[i][1] : sqft_lot

    # Y : y_test.values
    # hx = Q0 + Q1*x1 + Q2x2
    total = 0
    result = 0
    m = len(X)
    if(k==0):
        for i in range(m):

            total = total + (  Q0 + Q1*X[i][0] + Q2*X[i][1] - Y[i][0] )
    else:
        for i in range(m):

            total = total + ( Q0 + Q1*X[i][0] + Q2*X[i][1] - Y[i][0] )*X[i][k-1]

    result = (1/m) * total
    return result
```

- Şimdi asıl eğitme aşamasını yapacağımız GradientDescent algoritmamızı yazabiliriz. Ama öncelikle Q değerlerimizi oluşturalım.

Başlangıç için Q değerlerini 1,1,1 seçebiliriz. Fakat bu sayılar optimum değerlere doğru değişeceği için doğru seçim yapmak önemlidir. İlk uygulamada Q değerleri bazı sayılara yakınsamaktadır. Bu sayıların yakın olduğu değerleri vererek eğitme aşamasını kısaltmayı ve maliyet değerlerini düşürmeyi hedefleyebiliriz.

Ayrıca cost listesi oluşturarak her iterasyonda maliyetin ne kadar olduğunu tutabiliriz.

```
[ ]  Q     = [50000,12000,16000]
     # maliyetlerimizi bir listede tutalım
     cost = []
     Q

     [50000, 12000, 16000]
```

GradientDescent Algoritması

Şimdi tüm parametrelerimizi değiştirdiğimiz ve maliyeti gördüğümüz aşamaya geldik.

Burada ilk olarak bir epoch değerimiz var. Bu değer kadar x'leri eğitmekteyiz.

Ardından her adımda bir maliyet fonksiyonunu kullanarak hatamızı cost listesi içerisine atmaktayız. Burada hatalarımızı ezberleme olmaması için x_test ve y_test ile hesaplamaktayız.

Sonrasında her Q değerini q0,q1,q2 gibi değişkenlerde tutarak hesaplamaktayız. Bunu Q değerinden alfa (varsayılan 0.1 seçildi) öğrenme katsayısı * J_derivate değerini çıkararak yapmaktayız. Matematiksel olarak

$$Qj := Qj - a * ((1/m) * \sum(hq(x) - y) * x$$

işlemini yapmakta. Burada J_derivate ile gerekli veriler dışında k değerini de veriyoruz. Bu k değeri türev ifadesinin ne olacağını belirlemektedir.

Bu işlem sonucunda Q[i] değeri "hatanın türevi * alfa" kadar miktar uzaklaşıp yakınlaşmakta. Q[i+1] değerinde bu değişim hesaplanırken yine Q[i] kullanıldığı için değişim tüm Q[] değerlerinin değişim hesaplandıktan sonra yapılmaktadır.

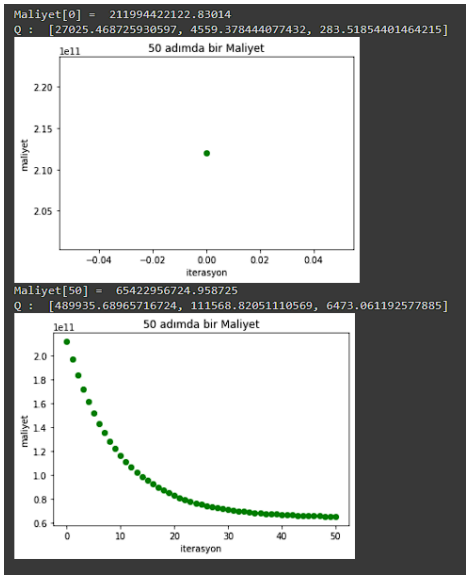
Yani eşzamanlı olarak değiştirilmekte ve böylece hesaplama hatası oluşmamaktadır.

Son olarak ise her adımda maliyet bastırmak yerine her 50 adımda maliyet değeri bastırılmakta ve bu sonuçlar tablo olarak çizdirilmekte.

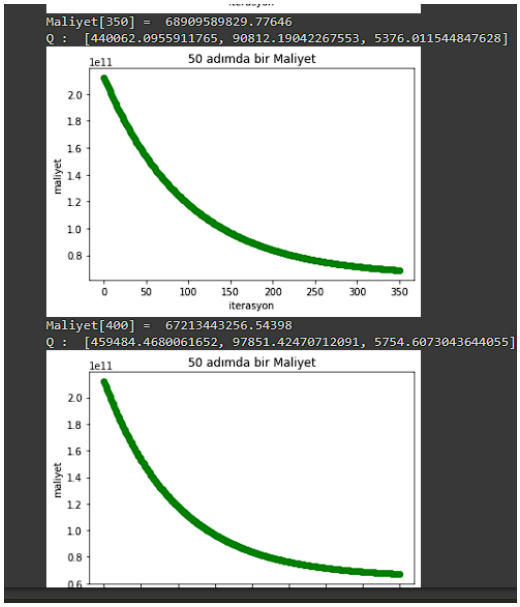
Hemen alt satırda ise bu fonksiyon çalıştırılmakta. Farklı değerler için sonuçları çalıştırıp görelim.

```
def GradientDescent(Q, X , Y , alfa=0.1, epoch=10000):
    # epoch sayısı kadar tablomuzu tekrar tekrar gezip, modelimizi eğiteceğiz.
    for i in range(epoch+1):
        # Her dongude gecici olarak maliyeti hesaplayalım ve bu değeri cost listemize atalım
        tempCost = J(X=x_test, Y=y_test.values, Q0=Q[0], Q1=Q[1], Q2=Q[2])
        cost.append(tempCost)

        # Paramtere hesaplama kismi
        q0 = Q[0] - alfa * J_derivate(X=X, Y=Y, Q0=Q[0], Q1=Q[1], Q2=Q[2], k=0)
        q1 = Q[1] - alfa * J_derivate(X=X, Y=Y, Q0=Q[0], Q1=Q[1], Q2=Q[2], k=1)
        q2 = Q[2] - alfa * J_derivate(X=X, Y=Y, Q0=Q[0], Q1=Q[1], Q2=Q[2], k=2)

        # Paramterlerin eşzamanlı güncellenmesi
        Q[0] , Q[1] , Q[2] = q0,q1,q2
```

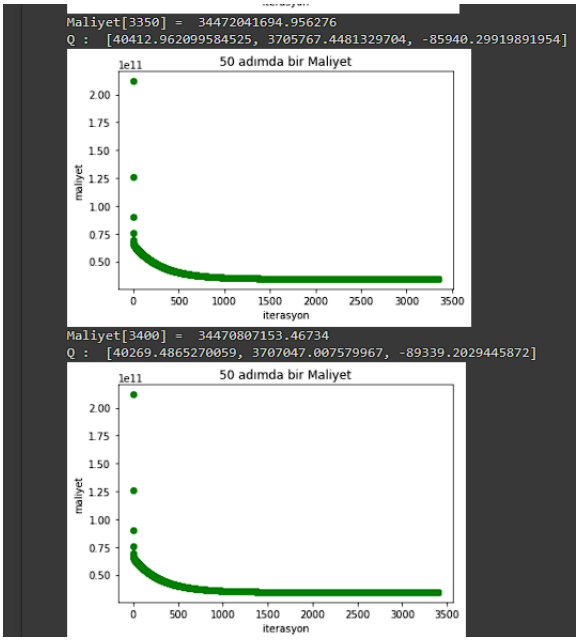
```
        # her 50 adımda maliyetin yazılması ve tablo çizilmesi
        if i % 50 == 0:
            print(f"Maliyet[{i}] = ",cost[i] )
            plt.scatter(x = range(len(cost)) ,y = cost, color="green")
            # tanım
            plt.xlabel("iterasyon")
            plt.ylabel("maliyet")
            plt.title("50 adımda bir Maliyet")
            plt.show()


GradientDescent(Q= Q, X=x_train , Y=y_train.values, alfa=0.1 , epoch=15000)
```



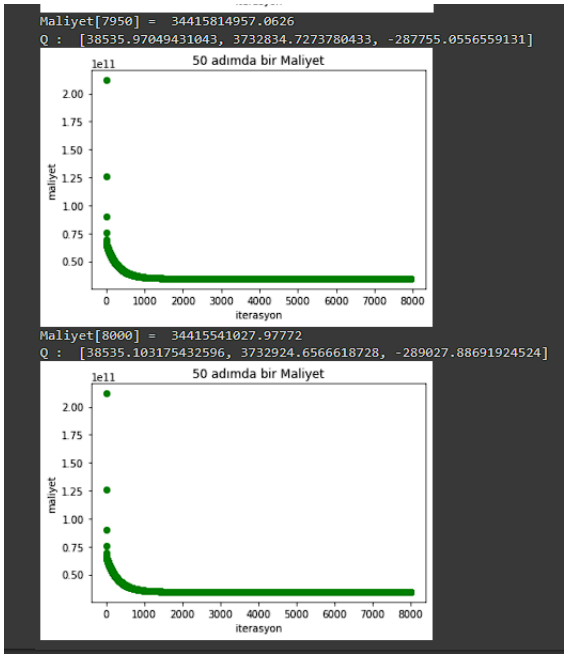
epoch 8000 , alfa =0.005



epoch 8000 , alfa =0.015



epoch 8000 , alfa =0.35, iterasyon 3400



epoch 8000 , alfa =0.35 , iterasyon = 8000

- Epoch 8000 ve alfa 0.35 için Q değerlerimizi görelim.

```
[63]  print(f"Q değerleri {Q}")

      Q değerleri [38535.103175432596, 3732924.6566618728, -289027.88691924524]
```

- Son olarak eğittiğimiz modelimizden Q değerlerini bulduk. Şimdi bu değerleri x_test ve y_test üzerinden Q değerlerimizi deneyelim ve yüzde kaç hata yaptığımızı her satır için bulalım

Öncelikle her x_test - y_test arasındaki yüzdelik hatayı bulacağımız için bu değerleri bir liste içerisinde tutmalıyız. Bu yüzden hatalar listesi tanımladık

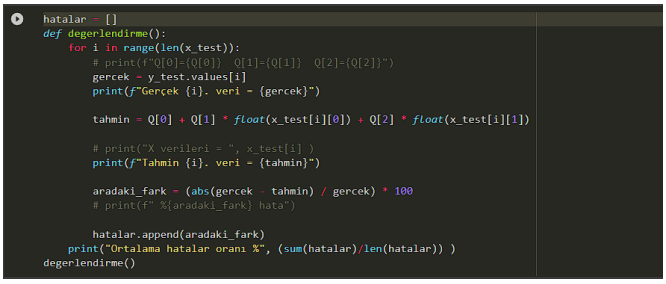
Ardından degerlendirme fonksiyonunda her x_test satırını görmek için döngü yazdık. Bu döngüde gercek değişkenimiz y_test.values[i] ile o satırdaki, değerini bildiğimiz y bilgisini tutacak.

tahmin değişkeni ise hipotezimizi uygulayacak.

Şimdi aradaki farkın mutlak değerini gercek veriye bölüp 100 ile çarparak

$$(|gercek - tahmin|/gercek) * 100$$

aradaki yüzdelik farkı bulmuş olduk. Bunu listemize ekleyebiliriz.



```
hatalar = []
def degerlendirme():
    for i in range(len(x_test)):
        # print(f"Q[0]={Q[0]}  Q[1]={Q[1]}  Q[2]={Q[2]}")
        gercek = y_test.values[i]
        print(f"Gerçek {i}. veri = {gercek}")

        tahmin = Q[0] + Q[1] * float(x_test[i][0]) + Q[2] * float(x_test[i][1])

        # print("X verileri = ", x_test[i] )
        print(f"Tahmin {i}. veri = {tahmin}")

        aradaki_fark = (abs(gercek - tahmin) / gercek) * 100
        # print(f" %{aradaki_fark} hata")

        hatalar.append(aradaki_fark)
    print("Ortalama hatalar oranı %", (sum(hatalar)/len(hatalar)) )
degerlendirme()
```

Döngü bittiğinde elimizde hataların hepsi bulunacak. Bunların aritmetik ortalamasını alarak ortalamada ne kadar hatamız olduğunu bulabiliriz.

Bu hücreyi de çalıştırırsak elde ettiğimiz sonuçlar şu şekilde (Epoch 8000 ve alfa 0.35 için)

```
Gerçek 6472. veri = [395000.]
Tahmin 6472. veri = 328488.86729938386
Gerçek 6473. veri = [505000.]
Tahmin 6473. veri = 462776.3678270633
Gerçek 6474. veri = [525000.]
Tahmin 6474. veri = 358681.52349790843
Gerçek 6475. veri = [415000.]
Tahmin 6475. veri = 1036148.2288286386
Gerçek 6476. veri = [557500.]
Tahmin 6476. veri = 490256.94511196786
Gerçek 6477. veri = [760000.]
Tahmin 6477. veri = 889638.4950295526
Gerçek 6478. veri = [280000.]
Tahmin 6478. veri = 406491.43798116944
Gerçek 6479. veri = [279900.]
Tahmin 6479. veri = 349455.7677062515
Gerçek 6480. veri = [349950.]
Tahmin 6480. veri = 350109.5141526079
Gerçek 6481. veri = [284000.]
Tahmin 6481. veri = 400826.545991171
Gerçek 6482. veri = [577500.]
Tahmin 6482. veri = 597364.4796692565
Gerçek 6483. veri = [258000.]
Tahmin 6483. veri = 412007.1623299532
Ortalama hatalar oranı % [35.77562956]
```

Ortalama olarak %35 hata oranına sahibiz.

Şimdi 2. sorumuza geçelim

b) Veri setindeki bütün öznitelikleri kullanarak çok değişkenli bir regresyon modeli ile evin fiyatını tah

Bu bölümde önceki fonksiyon ve algoritmaları kullanacağımız için detaya girilmeden genelleştirilmiş halleri anlatılacaktır.

İlk olarak verilerimizi X ve Y olarak tekrar parçalıyoruz. Burada evin fiyatını etkilemeyecek olan verileri ilik satırda atılmıştır. Fakat öncekinin aksine parametre sayısından bağımsız genelleşmiş bir uygulama olacağı için eklenmesinde de tasarım açısından sorun yoktur. Sadece yavaşlamaya neden olacaktır.

Bu yüzden bu değerler atılarak geri kalan 14 kolon attribute olarak seçildi.

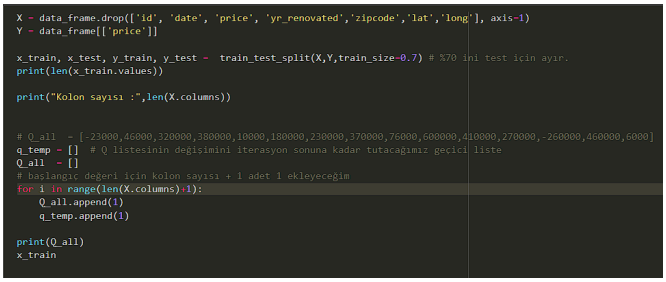
Hemen ardından veriler train ve test olarak %70 oranda parçalandı.

Q değişkenlerimizi tutmak için Q_all[ ] listesi oluşturuldu.

Her iterasyonda bu Q_all[*i*] bilgisi değişeceği için bu değişimler tek tek değişkenlerde tutulması yanlış olacaktır. Bu yüzden q_temp listesi oluşturuldu.

Bir döngü ile kolon sayısı + 1 kadar 1 değeri listelere eklendi.

Bunun sebebi x[*i*] kolon bilgilerine karşılık gelmekteydi. Örneğin sqft_living gibi. Fakat hipotezimizi $Q0 + Q1x1 + Q2x2 + ... + Qnxn$ şeklinde düşündük. Burada fazladan Q0 fazlalığı bulunduğu için 15 adet 1 ekledik.



```
X = data_frame.drop(['id', 'date', 'price', 'yr_renovated','zipcode','lat','long'], axis=1)
Y = data_frame[['price']]

x_train, x_test, y_train, y_test =  train_test_split(X,Y,train_size=0.7) # %70 ini test için ayır.
print(len(x_train.values))

print("Kolon sayısı :",len(X.columns))


# Q_all  = [-23000,46000,320000,380000,10000,180000,230000,370000,76000,600000,410000,270000,-260000,460000,6000]
q_temp = []  # Q listesinin değişimini iterasyon sonuna kadar tutacağımız geçici liste
Q_all  = []
# baslangıc değeri için kolon sayısı + 1 adet 1 ekleyeceğim
for i in range(len(X.columns)+1):
    Q_all.append(1)
    q_temp.append(1)

print(Q_all)
x_train
```

- Şimdi verilerimizi Normalize edebiliriz.

Burada yaptığımız işlem ilk soruda yaptığımız ile aynı

```
[ ]
scaler = MinMaxScaler()
scaler.fit(x_train)
# x train scale edilecek. Bu yüzden uygun hale getirildi.
print(x_train)
x_train = scaler.transform(x_train)
x_test  = scaler.transform(x_test)


print(type(y_train)) # pandas tipi
type(x_train)        # numpy tipi
x_train
```

- Önceki soruda hipotezimizi hep başka fonksiyonların içerisinde yazmıştık. Şimdi ise bunu ayıralım

Hq ile hipotezimizi bulacağız. Bunu

$$Q0 + Q1x1 + Q2x2 + ... + Qnxn$$

işlemi ile tanımlamıştık. Bu yüzden bu toplamı tutacak önce hq değişkenimizi oluşturuyoruz. Ardından her X satırı için bu formülü işletiyoruz. Burada for döngüsünde indisleri de kullanmak istediğimiz için enumerate ile X'leri key value haline getirdik.

Ayrıca Q[0] verimizin fazlalık olduğunu ve x çarpanı olmadığını belirtmiştik. Bu yüzden iterasyonu 1'den başlatıyoruz.

Q[0] 'ı ise en sonda sonucumuza ekleyip değeri döndürüyoruz.

```
def Hq(X , Q_all:list):
    hq = 0
    # Önce satırı al dizi olarak
    # Satırı enumerate ile (indis,veri) olarak al
    # Q[indis] * veri
    for j, attr in enumerate(X, start=1):
        hq = hq +  Q_all[j]*attr
    hq += Q_all[0]
    return hq
```

Maliyet Fonksiyonu - 2

- Artık daha genel maliyet ve gradiant fonksiyonları yazacağımız için bazı bölğmleri değiştireceğiz.

Burada satır sayımızı aldıktan sonra toplam ve result değişkenleri belirliyoruz.

Burada tekrar enumerate kullanarak X içerisindeki satırlara ulaşıyoruz ve her satırı sayabilir oluyoruz.

Artık ana işlemimiz olan farkın karelerini toplama işlemini yapabiliriz.

Burada satır bilgisi elimizde olduğu için (14 attribute'a sahip 1 tane X bilgisi) bunu ve Q bilgilerini Hq fonksiyonumuza verip tahmini y çıktısını buluyoruz.

Bunu Gerçek Y ile çıkartıp karesini alıyoruz ve sonuçları topluyoruz

En sonda bu toplamı 2*satır sayısına bölüyoruz.

Burası gördüğümüz maliyet fonksiyonunun aynısı.

```
def J_2(X, Y, Q_all:list ):
    """ VERİLEN Q DEĞERLERİ İÇİN TÜM TABLOYU DENER VE GERÇEK Y İLE FARKIN KARELERİNİ TOPLAR. BÖYLECE MALİYETİ ÖLÇER """

    # hx = Q0 + Q1*X1 + Q2x2  + . . .

    total = 0
    satir_sayisi = len(X)
    kolon_sayisi = X.shape[1]
    result = 0
    # hq = 0

    for i, satir in enumerate(X):
        # ------ bu kisim yukarıda Hq metoduna devredilmiştir ------------------
        # hq = 0
        # for j,attr in enumerate(satir , start=1):
        #     hq = hq +  Q[j]*attr
        # hq += Q[0]
        # ----------------------------------------------------------------------
        #
        # her satırı dizi olarak alıyorum ve Hq metoduna verip hipotezi uyguluyorum.

        total = total + ( Hq(X=satir,Q_all = Q_all) - Y[i][0])**2


    result = (1/2*satir_sayisi)*total
    print("toplam maliyet şu an bu : ",result)
    return result

# J_2(X=x_train , Y=y_train.values , Q_all=Q_all)
```

- Şimdi J_derivate ile türevli halini görelim

Burada ise yine aynı işlemleri yapıyor , Hq ile hipotezi hesaplayıp farkı buluyoruz. Öncesinde yaptığımız gibi yine X[i][k-1] katsayısı ile çarpıyoruz. (O Q[i] değerinin önündeki x attribute'u)

Ve en sonda toplam değeri satır sayısına bölüp döndürüyoruz.

```
def J_derivate_2(X, Y, Q_all:list , k:int ):
    """GRADİENT D. ALGORİTMASI İÇİN MALİYETİN, TÜREVİNİN ALINDIKTAN SONRA, HESAPLANDIĞI KISIM"""

    total = 0
    result = 0
    satir_sayisi = len(X)
    if(k==0):
        for i, satir in enumerate(X):
            total = total + ( Hq(X=satir,Q_all = Q_all) - Y[i][0] )
    else:
        for i, satir in enumerate(X):
            total = total + ( Hq(X=satir,Q_all = Q_all) - Y[i][0] )*X[i][k-1]

    result = (1/satir_sayisi) * total
    return result


print(J_derivate_2(X=x_train , Y=y_train.values , Q_all=Q_all, k=0))
J_derivate_2(X=x_train , Y=y_train.values , Q_all=Q_all, k=1)
```

GradientDescent Algoritması

- Modelimizi eğitme aşamasını genel bir şekilde yazmaya geçebiliriz.

Burada ilk olarak copy kütüphanesini import ettik. Yeri geldiğinde

bahsedilecek.

Ardından cost_all listesi oluşturuldu. Bu liste ile her iterasyonda maliyetimizi bu listeye ekleyeceğiz. Böylece listedeki verileri ve liste uzunluğunu kullanarak matplotlib'e tablo oluşturtacağız.

```
import copy

cost_all = []

def Gd_with_all(X , Y , Q_all:list, alfa=0.1, epoch= 100 ):

    satir_sayisi = len(X)
    kolon_sayisi = X.shape[1]


    for i in range(epoch):

        temp_cost = J_2(X=x_test, Y=y_test.values, Q_all=Q_all)
        cost_all.append(temp_cost)

        for j in range(kolon_sayisi+1):
            q_temp[j] = Q_all[j] - alfa * J_derivate_2(X=X, Y=Y, Q_all=Q_all, k=j)


        Q_all = copy.deepcopy(q_temp)
```

Ardından döngü ile verilen epoch sayısı kadar modelimizi eğiticez.

Her döngü içerisinde temp_cost ile maliyeti hesaplamaktayız ve bu veriyi cost_all listemize eklemekteyiz.

Ardından bir döngü ile modelin eğitildiği kısmı yapmaktayız. Burada kolon sayısının bir fazlası kadar döngü oluşturduk. Çünkü başta da belirttiğimiz gibi Q[0] yanında X çarpanı bulunmamakta.

Şimdi içerideki her iterasyonda Q[i] değerinin ne kadar değiştiği önceki soru ile benzer bir şekilde J_derivate_2 ile hesaplanacak. Ardından önceki sorudan farklı olarak bu geçici sonuç bir listede tutulacak. Bu işlem her Q değeri için yapıldıktan sonra içerideki döngümüz bitmiş olacak.

Döngü sonlandıktan sonra Q listemizi bulduğumuz geçici q_temp ile değiştirebiliriz. Fakat burada Q_all = q_temp yaparsak python liste özelliğinden dolayı shallow copy yapmış olacağız ve referans kopyalanması sorunu oluşacak. Bu yüzden import ettiğimiz copy ile deepcopy metodunu kullanarak değerleri kopyalıyoruz.

Bu iterasyonu her i değeri için görmek uzun bir süreç olacağı için her 50 adımda sonucu

göreceğimiz bir ifade ekliyoruz. Burada ayrıca plt.scatter ile cost_all listemizi verip maliyetimizin nasıl bir grafik izlediğini de çizdiriyoruz.

Sonuçları ve eğitmeyi en sonda görelim

```
        if i % 50 == 0:
            print(f"{i}. Maliyetim = ",temp_cost )
            plt.scatter(x = range(len(cost_all)) ,y = cost_all , color="red")
            # tanım
            plt.xlabel("epoch")
            plt.ylabel("cost")
            plt.title("Epoch - Cost")
            plt.show()


    print(Q_all)
```

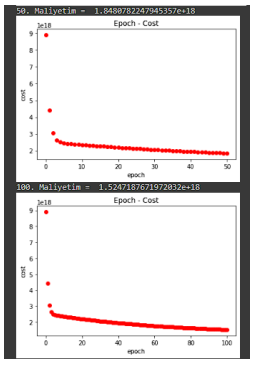
- Şimdi ilk soruda yaptığımız yüzdelik hata bulmayı revize edelim ve tekrar deneyelim.

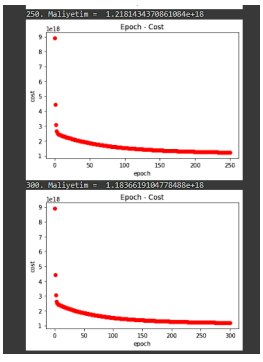
Burada hataları tutacak bir listemiz var. Bu listeye her iterasyonda gerçek değer ile tahmin değer arasındaki yüzdelik farkı buluyoruz. Tahmini ise x_test ve Q değerlerini hipotez fonksiyonumuza vererek yaptık.

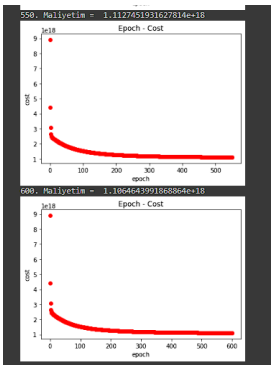
```
hatalar = []
def percent_error():
    for i in range(len(x_test)):
        # print(f"Q[0]={Q[0]}  Q[1]={Q[1]}  Q[2]={Q[2]}")
        gercek = y_test.values[i]
        print(f"Gerçek {i}. veri = {gercek}")


        tahmin = Hq(X = x_test[i], Q_all = Q_all)

        # print("X ver  Loading...  x_test[i] )
        print(f"Tahmin {i}. veri = {tahmin}")

        aradaki_fark = (abs(gercek - tahmin) / gercek) * 100
        # print(f" %{aradaki_fark} hata")

        hatalar.append(aradaki_fark)
    print("Ortalama hatalar oranı %", (sum(hatalar)/len(hatalar)) )

percent_error()
```

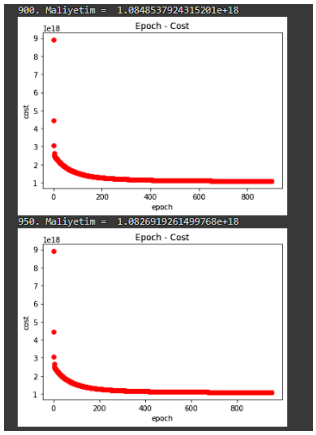
Şimdi sonuçlarımızı görelim.

(alfa=0.20 , epoch=1000)







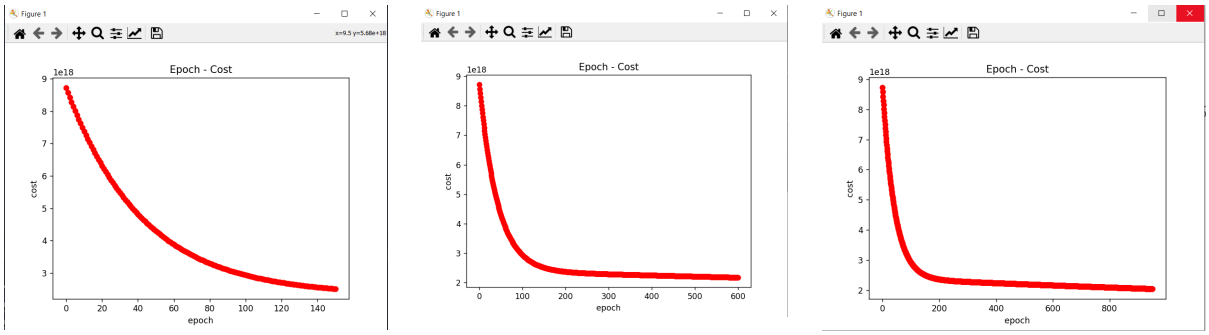


Q değerleri = [-154535.40088235194, -23601.05417902612, 382231.68690385355, 521983.1666968419, -8862.907305400215, 132577.3125777927, 471541.3542879963, 233011.74856392056, 69281.84846283196, 959837.6930524483, 592346.8325346287, 367966.8133860462, -404452.1103062461, 463869.4666701383, -40863.99019222211]

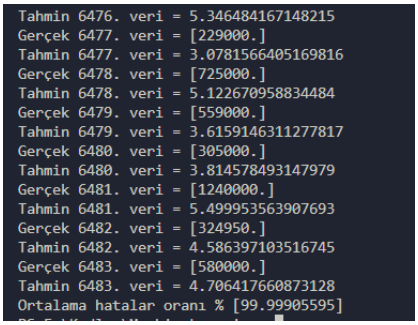
%'lik hata

```
Gerçek 6471. veri = [459900.]
Tahmin 6471. veri = 3.765191853975859
Gerçek 6472. veri = [855000.]
Tahmin 6472. veri = 4.843637596200022
Gerçek 6473. veri = [128000.]
Tahmin 6473. veri = 3.0754673561204813
Gerçek 6474. veri = [440000.]
Tahmin 6474. veri = 3.613020573941843
Gerçek 6475. veri = [250000.]
Tahmin 6475. veri = 3.192896740342808
Gerçek 6476. veri = [570000.]
Tahmin 6476. veri = 4.230998149875649
Gerçek 6477. veri = [343566.]
Tahmin 6477. veri = 2.8421227535890425
Gerçek 6478. veri = [455000.]
Tahmin 6478. veri = 2.9623055683055055
Gerçek 6479. veri = [320000.]
Tahmin 6479. veri = 4.33449364704969
Gerçek 6480. veri = [590000.]
Tahmin 6480. veri = 4.390575345168148
Gerçek 6481. veri = [460000.]
Tahmin 6481. veri = 2.9666266063877056
Gerçek 6482. veri = [565000.]
Tahmin 6482. veri = 4.161179000932275
Gerçek 6483. veri = [360000.]
Tahmin 6483. veri = 2.6302809437812686
Ortalama hatalar oranı % [99.99908495]
```

(alfa=0.005 , epoch=1000)



%lik hata



Sonuç :

Veriler x_train ve y_train ile eğitilmiş ve x_test ve y_test ile maliyet değerleri kontrol edilip , hata oranları bulunmuştur.

İlk soruda 2 parametre için alfa ve epoch değerleri ile oynanarak %30 - % 50 arasında hata oranları alınmıştır

Fakat 2. soruda tüm parametrelerin verilmesinde hata oranları %99.99 oranında olmuştur.