In [1]:
```
import sys
IN_COLAB = 'google.colab' in sys.modules
if IN_COLAB:
    !git clone https://github.com/cs357/demos-cs357.git
    !mv demos-cs357/figures/ .
    !mv demos-cs357/additional_files/ .
```

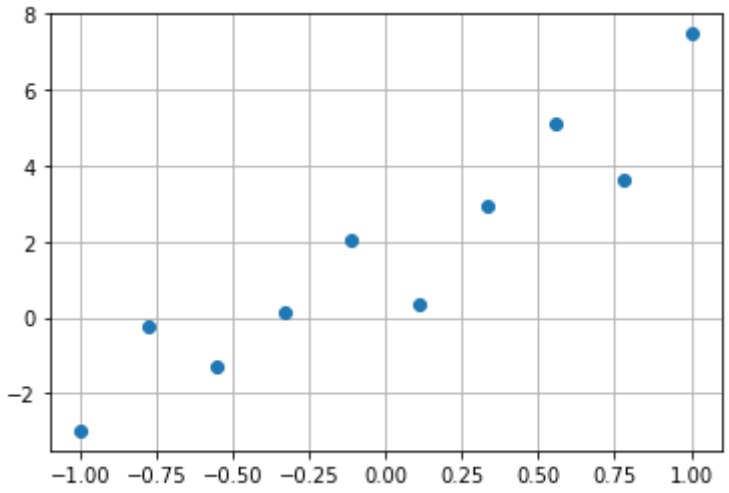
In [2]:
```
import numpy as np
import numpy.linalg as la
import scipy.linalg as sla
import matplotlib.pyplot as plt
%matplotlib inline
import random

import pandas as pd
from scipy import stats

import sys
sys.path.append('./additional_files')
```

A) Data Fitting with Least Squares: simple examples

1) Fit with a line

Suppose we want to fit the points below using a line:

In [3]:
```
n = 10
t = np.linspace(-1, 1, n)
y = 4*t  + np.random.randn(n)  + 2

plt.plot(t,y,'o')
plt.grid()
```



We want to use a fit function with the form:

$$y = x_0 + x_1 t$$

What's the system of equations for x_0 and x_1? We will solve the least squares problem by solving the Normal Equations

$$A^T A x = A^T b$$

Write the model matrix:

In [4]:
```
m = len(t)
n = 2
```

In [5]:
```
A = np.ones((m,n))
A[:,1] = t
```

Build the arrays needed to solve the normal equations

In [6]:
```
AtA = A.T@A
Atb = A.T@y
```

Solve the linear system of equations:

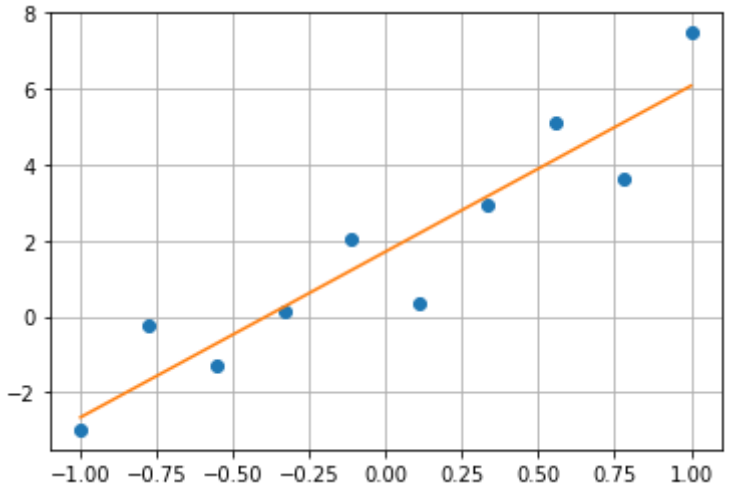
In [7]:
```
x = la.solve(AtA,Atb)
```

In [8]:
```
x
```

Out[8]:
```
array([1.71134979, 4.37036809])
```

Plot the fit:

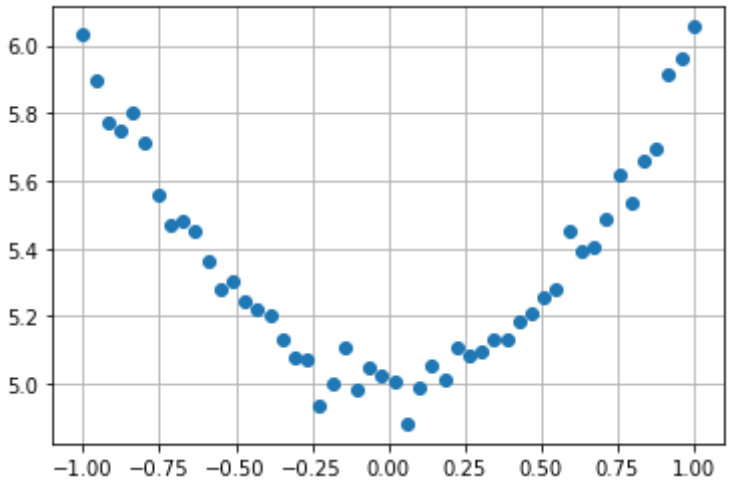
In [9]:
```
plt.plot(t, y, 'o')
plt.plot(t,  x[1] * t + x[0])
plt.grid()
```



2) Fit with a quadratic curve

What if the set of points looks like this?

In [10]:
```
n = 50
t = np.linspace(-1, 1, n)
y = t ** 2 + np.random.randn(n) * 0.05 + 5

plt.plot(t, y, 'o')
plt.grid()
```



We want to use a fit function with the form:

$$y = x_0 + x_1 t + x_2 t^2$$

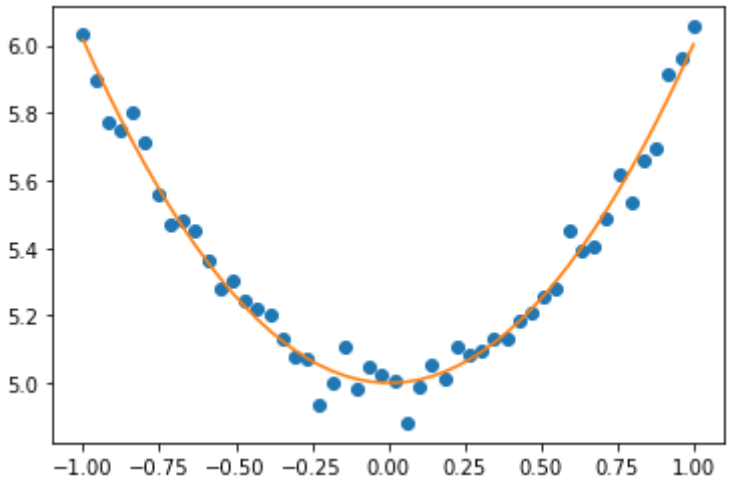
In [11]:
```
m = len(t)
n = 3 # we want to find three coefficients to fit with a quadratic funct
ion

A = np.ones((m,n))
A[:,1] = t
A[:,2] = t**2

#Build the arrays needed to solve the normal equations

AtA = A.T@A
Atb = A.T@y

#Solve the linear system of equations:

x = la.solve(AtA,Atb)
```

In [12]:
```
x
```

Out[12]:
```
array([ 5.00017417, -0.00679983,  1.00937144])
```

In [13]:
```
plt.plot(t, y, 'o')
plt.plot(t, x[2] * t ** 2 + x[1] * t + x[0])
```

Out[13]:
```
[<matplotlib.lines.Line2D at 0x1a18972810>]
```

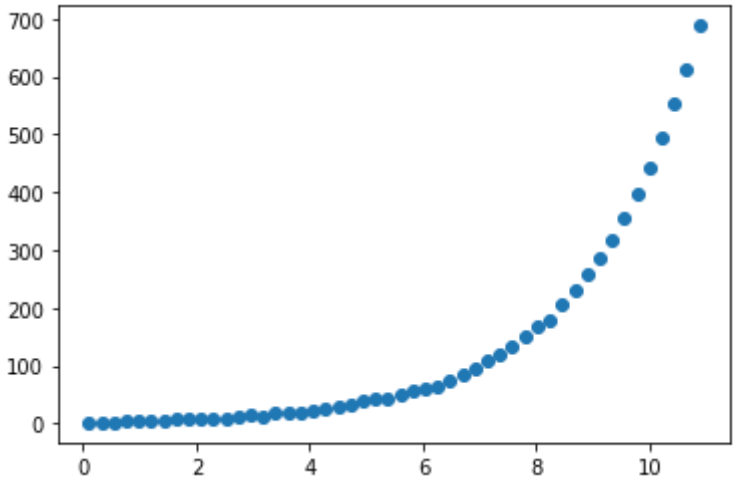


3) Fit an exponential curve

What if the set of points looks like this?

In [14]:
```
n = 50
t = np.linspace(0.1, np.e * 4, n)
y = np.exp(t * 0.5) * 3 + np.random.randn(n) * 2

plt.plot(t, y, 'o')
```

Out[14]:
```
[<matplotlib.lines.Line2D at 0x1a18f72850>]
```



We want to use a fit function with the form:

$$y = x_0\, e^{x_1\, t}$$

Noticed that we can't directly use linear least squares here since the function is not linear with respect to all the coefficients x_i. What if we take the natural log of both sides?

$$\ln y = \ln x_0 + x_1 t$$

with the change of variables $\bar{y} = \ln y$ and $\bar{x}_0 = \ln x_0$ we can re-write the above equation as:

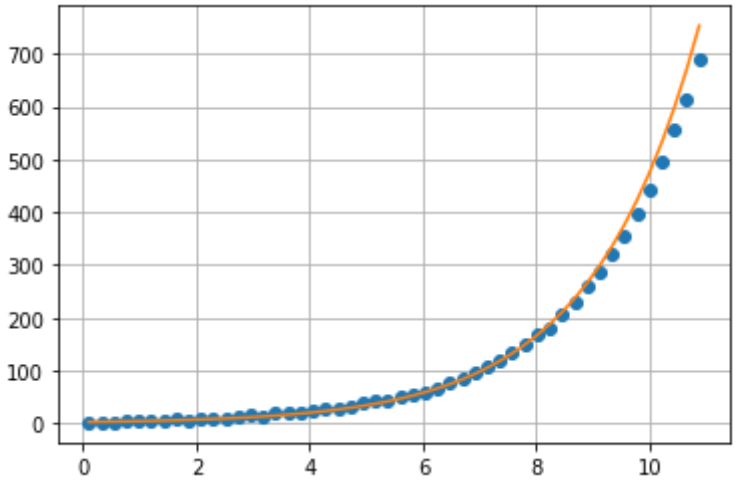
$$\bar{y} = \bar{x}_0 + x_1 t$$

In [15]:
```
m = len(t)
n = 2 # we want to find three coefficients to fit with a quadratic funct
ion

A = np.ones((m,n))
A[:,1] = t

#Build the arrays needed to solve the normal equations

AtA = A.T@A
Atb = A.T@np.log(y)

#Solve the linear system of equations:

x = la.solve(AtA,Atb)
```

In [16]:
```
x
```

Out[16]:
```
array([0.90112513, 0.52635894])
```

In [17]:
```
x0 = np.exp(x[0])
plt.plot(t, y, 'o')
plt.plot(t,  x0 * np.exp(x[1]*t))
plt.grid()
```



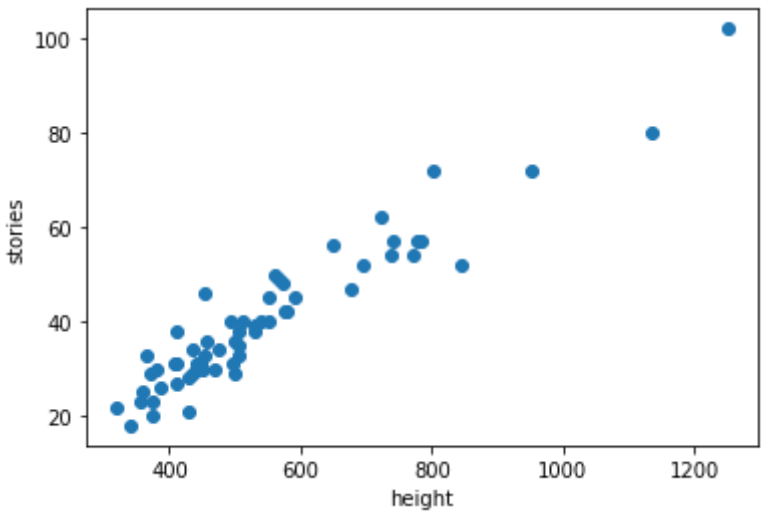
4) building stories vs height

In [18]:
```
data = pd.read_csv('./additional_files/bldgstories.txt', delim_whitespac
e=True)
```

In [19]:
```
year = data.values[:,0]
hght = data.values[:,1]
stories= data.values[:,2]
```

In [20]:
```
plt.plot(hght, stories, 'o')
plt.xlabel('height')
plt.ylabel('stories')
```

Out[20]: Text(0, 0.5, 'stories')



We want to use a fit function with the form:

$$y = x_0 + x_1 t$$

The model matrix is:

In [21]:
```
m = len(hght)
n = 2

A = np.ones((m,n))
A[:,1] = hght
```

Solving using normal equations:

In [22]:
```
AtA = A.T@A
Atb = A.T@stories

x = la.solve(AtA,Atb)
print(x)
```

```
[-3.33129846  0.08001458]
```

In [23]:
```
# Get the residual Euclidean norm:
print( la.norm(A@x-stories,2) )

# Squared of the Euclidean norm of the residual
print( la.norm(A@x-stories,2)**2 )
```

```
37.39104302480999
1398.0900984831917
```

Using the least-squares built-in function:

In [24]:
```
coeff, residual, rank, singvalues = la.lstsq(A,stories,rcond=None)
print(coeff)

print(residual)

print(rank)

print(singvalues)
```

```
[-3.33129846  0.08001458]
[1398.09009848]
2
[4.46706748e+03 2.48020755e+00]
```

In [25]:
```
u,s,vt = la.svd(A)
print(s)
```

```
[4.46706748e+03 2.48020755e+00]
```

Or using another built-in function for linear regression:

In [26]:
```
a, b, rvalue, pvalue, stderr = stats.linregress(hght, stories)
print(a)
print(b)
```

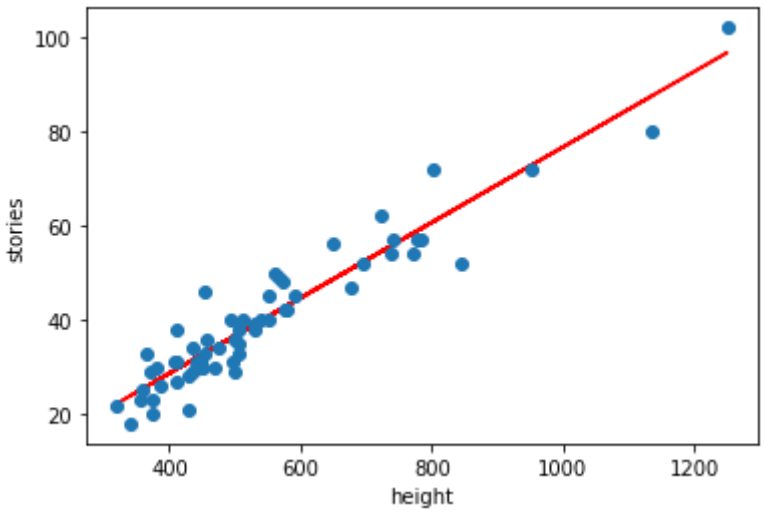
```
0.08001457924032182
-3.3312984582958265
```

In [27]:
```
print(rvalue)
```

```
0.9505548942627724
```

In [28]:
```
plt.plot(hght, x[0] + x[1]*hght , 'r-')
plt.plot(hght, stories, 'o')
plt.xlabel('height')
plt.ylabel('stories')
```

Out[28]: Text(0, 0.5, 'stories')



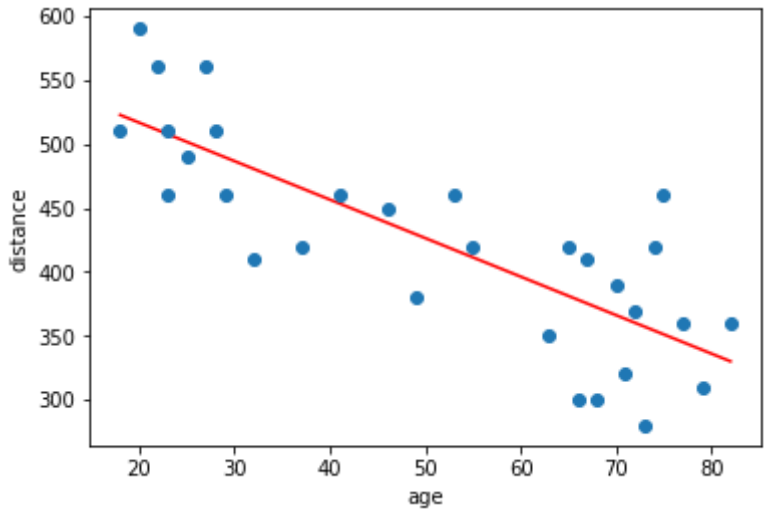
5) eye sight distance vs age

In [29]:
```
data = pd.read_csv('./additional_files/signdist.txt', delim_whitespace=True)
```

In [30]:
```
age = data.values[:,0]
distance = data.values[:,1]
```

In [31]:
```
a, b, rvalue, pvalue, stderr = stats.linregress(age, distance)
plt.plot(age, a*age + b, 'r-')
plt.plot(age, distance, 'o')
plt.xlabel('age')
plt.ylabel('distance')

print(abs(rvalue))
```

```
0.8012446509407871
```



6) height vs GPA

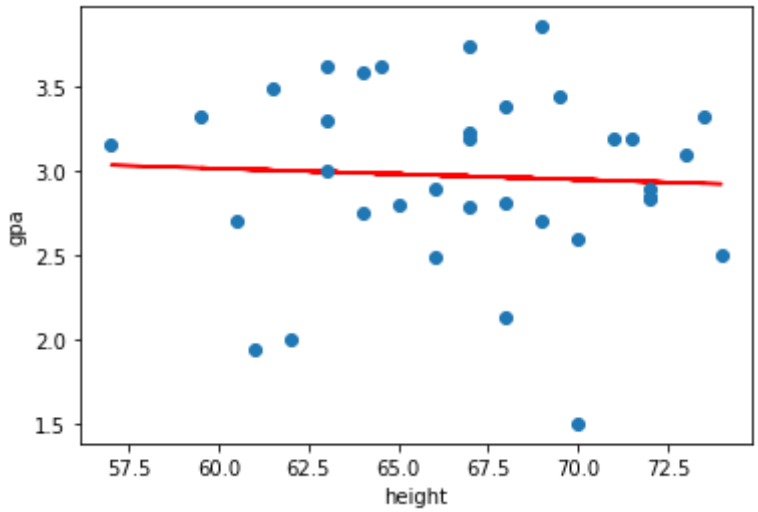
What?!

In [32]:
```
data = pd.read_csv('./additional_files/heightgpa.txt', delim_whitespace=
True)
```

In [33]:
```
height = data.values[:,0]
gpa = data.values[:,1]
```

In [34]:
```
a, b, rvalue, pvalue, stderr = stats.linregress(height, gpa)
plt.plot(height, a*height + b, 'r-')
plt.plot(height, gpa, 'o')
plt.xlabel('height')
plt.ylabel('gpa')

print(abs(rvalue))
```

```
0.05324125988691996
```



B) Modeling ice extent over time


Ice Extent

In addition to fitting functions to datapoints, we can use least squares to make predictions on events that may happen in the future. Here we have a dataset containing the extent of arctic sea ice over the years, which we can fit a least squares model to and predict the extent of arctic ice in future years.

This is based on data from:

http://ww2.amstat.org/publications/jse/v21n1/witt.pdf (http://ww2.amstat.org/publications/jse/v21n1/witt.pdf)

http://nsidc.org/research/bios/fetterer.html (http://nsidc.org/research/bios/fetterer.html)

ftp://sidads.colorado.edu/DATASETS/NOAA/G02135/north/monthly/data/N_08_extent_v3.0.csv

In [35]:
```
data = pd.read_csv('./additional_files/N_09_extent_v3.0.csv', dtype={'ye
ar': np.int32, 'extent': np.double})
```

In [36]:
```
data.head()
```

Out[36]:

	year	mo	data-type	region	extent	area
0	1979	9	Goddard	N	7.05	4.58
1	1980	9	Goddard	N	7.67	4.87
2	1981	9	Goddard	N	7.14	4.44
3	1982	9	Goddard	N	7.30	4.43
4	1983	9	Goddard	N	7.39	4.70

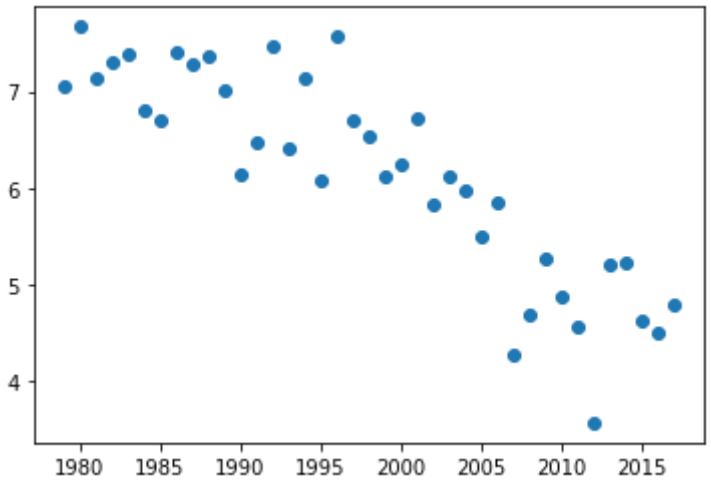
In [37]:
```
print(data.shape)
```

```
(39, 6)
```

In [38]:
```
year = data['year']
extent = data[' extent']
```

In [39]:
```
plt.figure(figsize=(6,4))
plt.plot(year, extent, 'o')
```

Out[39]:
```
[<matplotlib.lines.Line2D at 0x1a199025d0>]
```



Fitting with a line

In [40]:
```
npoints = data.shape[0]
print('number of data points = ', npoints)
```

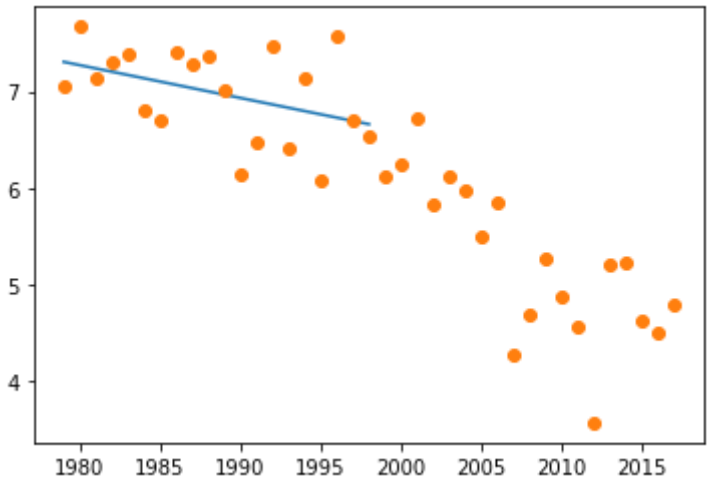
```
number of data points =  39
```

In [41]:
```
def fitfunction(t,coeffs):
    fit = 0
    for i,c in enumerate(coeffs):
        fit += c*t**i
    return  fit
```

In [42]:
```
ndata = 20 #use the first ndata points for the fit
year1 = year[:ndata]
extent1 = extent[:ndata]

A = np.array([
    1+0*year1,
    year1
    ]).T

b = np.array(extent1)

x = la.solve(A.T@A,A.T@b)
```

In [43]:
```
plt.plot(year1, fitfunction(year1,x))
plt.plot(year, extent, 'o')
```

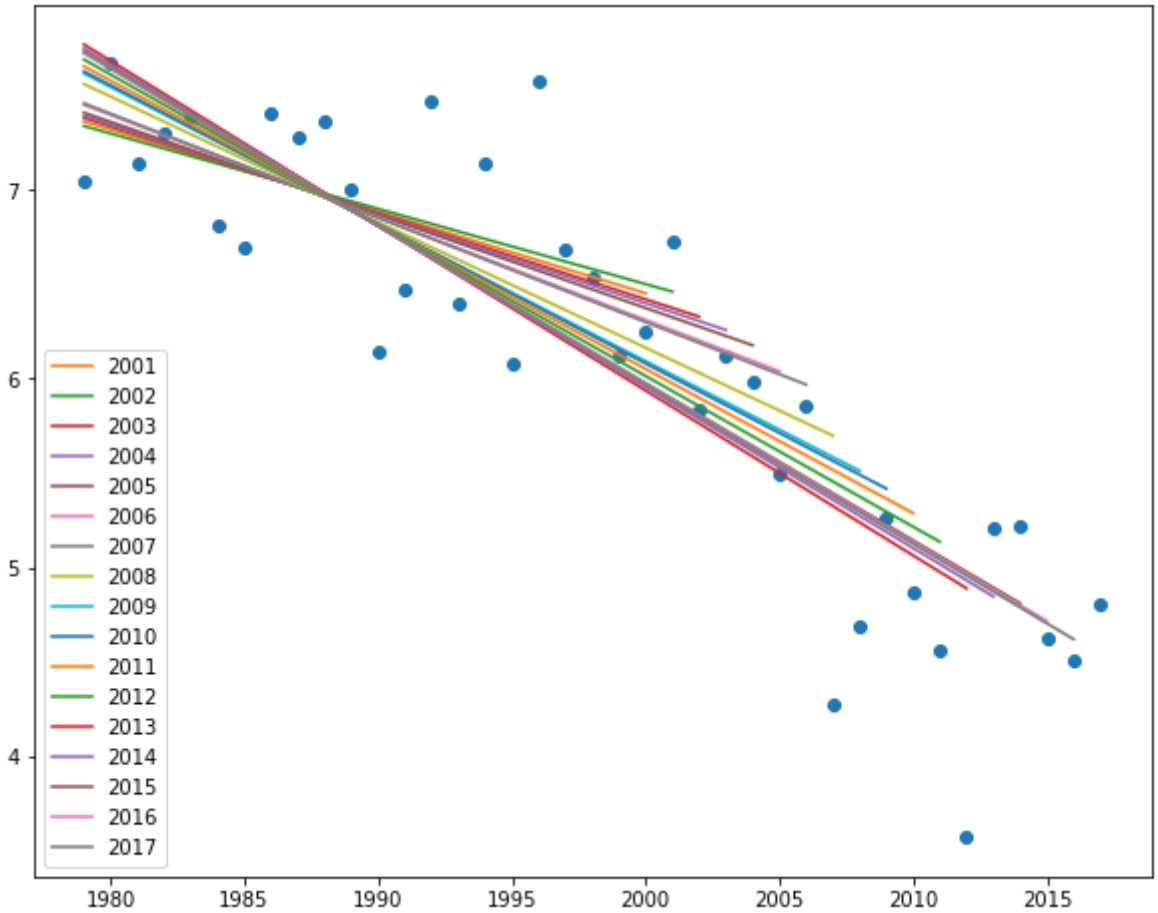
Out[43]:
```
[<matplotlib.lines.Line2D at 0x1a1993c310>]
```



How did the linear fit "fit" as time time progresses?

In [44]:
```
plt.figure(figsize=(10,8))
plt.plot(year, extent, 'o')

for ndata in range(22, npoints):

    year1 = year[:ndata]
    extent1 = extent[:ndata]

    A = np.array([
        1+0*year1,
        year1
        ]).T

    b = np.array(extent1)

    x = la.solve(A.T@A,A.T@b)

    plt.plot(year1, fitfunction(year1,x),  label='%d' % (year[0]+ndata))

plt.legend()
```

Out[44]: <matplotlib.legend.Legend at 0x1a19031b10>



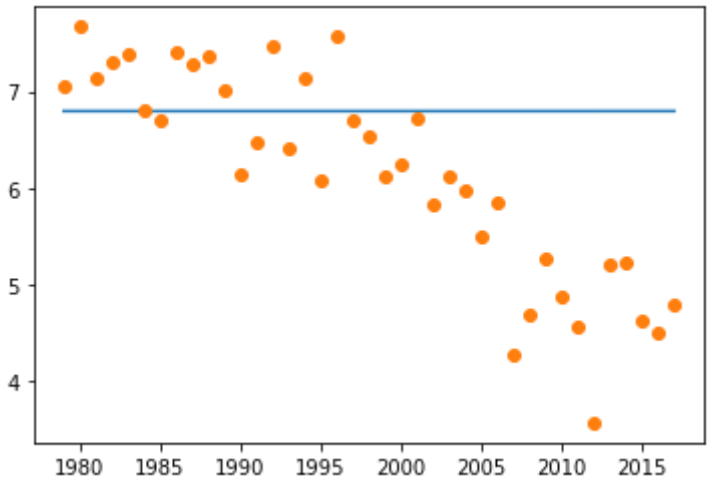
Let's try a quadratic fit

In [45]:
```
ndata = 26 #use the first ndata points for the fit
year1 = year[:ndata]
extent1 = extent[:ndata]

A = np.array([
    1+0*year1,
    year1,
    year1**2
    ]).T

b = np.array(extent1)

x = la.solve(A.T@A,A.T@b)

print(x)
```

```
[6.79221628e+00 7.00452986e-07 3.50158197e-10]
```

In [46]:
```
plt.plot(year, fitfunction(year,x))
plt.plot(year, extent, 'o')
```

Out[46]:
```
[<matplotlib.lines.Line2D at 0x1a18961c10>]
```

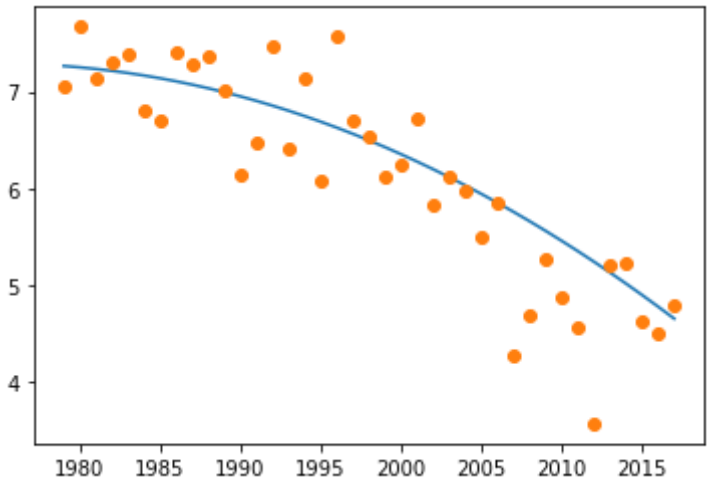


What went wrong?

Let's try to use the least square function from scipy

In [47]:
```
coeffs,residual,rank,sval=np.linalg.lstsq(A,b,rcond=None)

plt.plot(year, fitfunction(year,coeffs))
plt.plot(year, extent, 'o')

print(coeffs)
```

```
[-5.78703451e+03  5.86799957e+00 -1.48565324e-03]
```



Seems to work with `lstsq` ... what could be the issue with the Normal Equations method above?

Let's check the condition number of the matrix A

In [48]:
```
print(la.cond(A))
print(x)
print(la.norm(A@x-b))
```

```
313354558941.1675
[6.79221628e+00 7.00452986e-07 3.50158197e-10]
2.7991699750018357
```

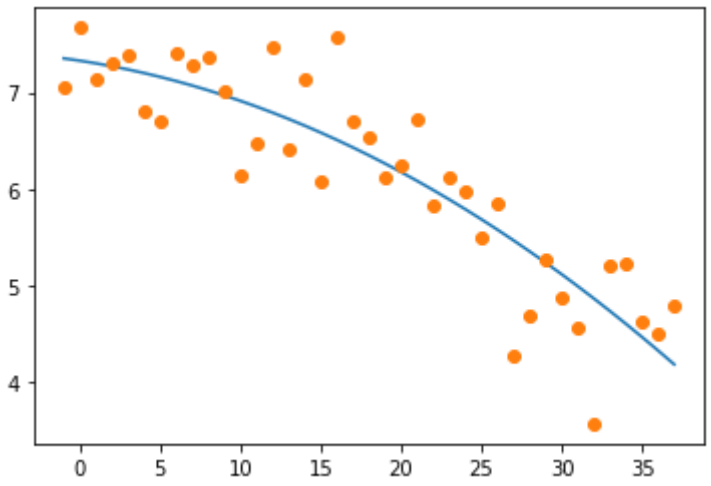
The matrix A becomes closer to singular as the number of columns increases (i.e., as the number of coefficients for the fit increase). We can scale the years, to mitigate this situation:

In [49]:
```
year2 = year - 1980
extent2 = extent

A = np.array([
    1+0*year2,
    year2,
    year2**2
    ]).T

b = np.array(extent2)

x = la.solve(A.T@A,A.T@b)


print(la.cond(A))
print(x)
```

```
1593.2774157903386
[ 7.32702301e+00 -2.55283521e-02 -1.60576890e-03]
```

In [50]:
```
plt.plot(year2, fitfunction(year2,x))
plt.plot(year2, extent, 'o')
```

Out[50]:
```
[<matplotlib.lines.Line2D at 0x10471d8d0>]
```



C) Linear Least Squares using SVD

The function below creates a random matrix and a random right-hand side vector, to use as input data for least squares. The arguments of the function are the shape of A, and the rank of A. You should run examples to investigate the following situations:

1) rankA = N (this is a full rank matrix, and hence solution is unique

2) rankA = N - 1 (this is a rank deficient matrix, and the solution is no longer unique

In [51]:
```
def creates_A_b(shap = (10,4), rankA=4):
    M,N = shap
    # Generating the orthogonal matrix U
    X = np.random.rand(M,M)
    U,R = sla.qr(X)
    # Generating the orthogonal matrix V
    Y = np.random.rand(N,N)
    V,R = sla.qr(Y)
    Vt = V.T
    # Generating the diagonal matrix Sigma
    singval = random.sample(range(1, 9), rankA)
    singval.sort()
    sigmavec = singval[::-1]
    sigma = np.zeros((M,N))
    for i,sing in enumerate(sigmavec):
        sigma[i,i] = sing
    A = U@sigma@Vt
    b = np.random.rand(M)
    return(A,b)
```

In [52]:
```
# Matrix shape
M = 10
N = 4
r = 2
A,b = creates_A_b((M,N),r)
```

In [53]:
```
print(la.cond(A))
```

```
4.625115885046109e+16
```

1) Using normal equations (unique solution, full rank)

In [54]:
```
xu = la.solve(A.T@A,A.T@b)
print(xu)
```

```
[ 0.01396977 -0.9037762  -0.00833452  0.78846146]
```

In [55]:
```
la.norm(A@xu-b,2)
```

Out[55]:
```
1.061029830914453
```

In [56]:
```
la.norm(xu,2)
```

Out[56]:
```
1.1994780179500095
```

2) Using SVD

In [57]:
```
UR,SR,VRt = la.svd(A,full_matrices=False)
print(SR)
```

```
[6.00000000e+00 3.00000000e+00 6.77057243e-16 1.29726479e-16]
```

In [58]:
```
# ub = (UR.T@b)
# x = np.zeros(N)
# for i,s in enumerate(SR):
#     if s > 1e-15:
#         x += VRt[i,:]*ub[i]/s
# print(x)
```

In [59]:
```
Sinv = np.zeros((N,N))
for i,si in enumerate(SR):
    if si > 1e-12:
        Sinv[i,i] = 1/si

x = VRt.T@Sinv@UR.T@b
print(x)
```

```
[-0.1534047   0.0949751   0.0667189   0.22137863]
```

In [60]:
```
la.norm(A@x-b,2)
```

Out[60]:
```
1.0610298309144532
```

In [61]:
```
la.norm(x,2)
```

Out[61]:
```
0.29328004036984134
```

In [62]:
```
## Side note (if we want to predict the residual based on the full SVD)
U,S,Vt = la.svd(A)
Sfullinv = np.zeros((N,M))
for i,si in enumerate(S):
    if si > 1e-12:
        Sfullinv[i,i] = 1/si

x = Vt.T@Sfullinv@U.T@b
print(x)

la.norm((U.T@b)[r:],2)
```

```
[-0.1534047   0.0949751   0.0667189   0.22137863]
```

Out[62]:
```
1.061029830914453
```

3) Using numpy.linalg Least Squares method

In [63]:
```
coeffs,residual,rank,sval=np.linalg.lstsq(A,b,rcond=None)
print(coeffs)
```

```
[-0.1534047   0.0949751   0.0667189   0.22137863]
```

In [64]:
```
la.norm(A@coeffs-b,2)
```

Out[64]:
```
1.0610298309144532
```

In [65]:
```
la.norm(coeffs,2)
```

Out[65]:
```
0.2932800403698414
```

In [66]:
```
rank
```

Out[66]:
```
2
```

D) Least Squares Predictor for Fantasy Football

In Fantasy Football, contestants choose from a pool of available (American) football players to build a team. Contestants' teams score points depending on how their chosen players performed in real-life. The more points scored, the better!

There are literally hundreds of websites and blogs dedicated to predicting who will have a good game. They use a variety of methodologies (including no methodology at all) to generate their predictions. We will try to develop a predictor using Linear Least Squares that will answer the question: "Should I pick this player?"

Bonus: This activity may help you with the "breast cancer MP", since you will be using similar data structures in that assignment.

There are two data sets, `FF-data-2018.csv` and `FF-data-2019.csv` that were collected using scoring from the Yahoo Fantasy Football platform. The 2018 data was collected from here (http://rotoguru1.com/cgi-bin/fyday.pl?week=16&year=2018&game=yh&scsv=1). You can choose other years going back to 2011 from a variety of platforms.

Let's read in the data and see what it looks like.

In [67]:
```
ff_2018 = pd.read_csv('./additional_files/FF-data-2018.csv')
ff_2018
```

Out[67]:

	Week	Year	GID	Name	Pos	Team	h/a	Oppt	YH points	YH salary
0	1	2018	1242	Fitzpatrick; Ryan	QB	tam	a	nor	42.28	25.0
1	1	2018	1151	Brees; Drew	QB	nor	h	tam	31.56	33.0
2	1	2018	1231	Rivers; Philip	QB	lac	h	kan	29.96	31.0
3	1	2018	1523	Mahomes II; Patrick	QB	kan	a	lac	28.34	27.0
4	1	2018	1252	Rodgers; Aaron	QB	gnb	h	chi	24.94	39.0
...	...	...	...	...	...	...	...	...	...	...
6350	16	2018	7013	Indianapolis	Def	ind	h	nyg	2.00	13.0
6351	16	2018	7010	Denver	Def	den	a	oak	1.00	16.0
6352	16	2018	7029	Tampa Bay	Def	tam	a	dal	1.00	10.0
6353	16	2018	7015	Kansas City	Def	kan	a	sea	-1.00	13.0
6354	16	2018	7012	Green Bay	Def	gnb	a	nyj	-2.00	15.0

6355 rows × 10 columns

There are 6,355 data points which have a number of fields. They are:

- **Week**: The NFL season features 17 weeks of games, and each team plays 16 games in this time period. This column tells you which week the player's game was. I didn't include week 17, because many of the best players take that week off.
- **Year**: Which year the game was played. For this data set, all the year values are equal to 2018.
- **GID**: A unique ID tag for each player. We'll ignore this column.
- **Name**: The actual name of the player. In the case of defenses, the defense of the entire team is included, so in that case, this is the name of a city.
- **Pos**: This is the position of the player. The available choices are quarterback (QB), running back (RB), wide receiver (WR), tight end (TE), and defense (Def).
- **Team**: An abbreviation that indicates which team the player belongs to. Ryan Fitzpatrick was a member of the Tampa Bay Buccaneers, so his Team value is "tam".
- **h/a**: Whether the player's game was played at home or on the road. The possible values are 'h' (home) and 'a' (away).
- **Oppt**: The opposing team that the player faced. Ryan Fitzpatrick played against the New Orleans Saints in week 1, so his Oppt value is "nor".
- **YH points**: The amount of points the player scored that week. Ryan Fitzpatrick scored a whopping 42.28 points in week 1.
- **YH salary**: On many Fantasy Football sites, you start with a certain budget, and select a team of players within the constraints of that budget. Ryan Fitzpatrick only took 25.0 "dollars" of your available budget if you selected him on your team. It gives an indication of how the platform judges the quality of a player.

We can access the labels and put them in a list:

In [68]:
```
labels = list(ff_2018.columns)
print(labels)
```

```
['Week', 'Year', 'GID', 'Name', 'Pos', 'Team', 'h/a', 'Oppt', 'YH point
s', 'YH salary']
```

We can print out the available values of the positions for the data set by passing the key `Pos` as a string to the data set.

In [69]:
```
print(ff_2018['Pos'].values)
```

```
['QB' 'QB' 'QB' ... 'Def' 'Def' 'Def']
```

To remove all the duplicates, we can call the function `numpy.unique` to access all distinct values. (Just like every other time you use a new function, review the documentation of `numpy.unique` ! You can do so by running a cell with the following command: `np.unique?`)

In [70]:
```
positions = np.unique(ff_2018['Pos'])
print(positions)
```

```
['Def' 'QB' 'RB' 'TE' 'WR']
```

Since the positions in football are so different, we really want to focus on one at a time. It would be very ambitious to try and create a general predictor for all positions. Let's focus on quarterbacks first.

How can we extract all the data for quarterbacks? We can find the rows in the dataframe that has position equal to `QB`

In [71]:
```
POS = 'QB'
ff_2018['Pos'] == POS
```

Out[71]:
```
0         True
1         True
2         True
3         True
4         True
         ...
6350     False
6351     False
6352     False
6353     False
6354     False
Name: Pos, Length: 6355, dtype: bool
```

We will create another (smaller) dataframe that has the rows referring to the quarterback position.

In [72]:
```
df_POS = ff_2018[ff_2018['Pos'] == POS].copy()
df_POS.head()
```

Out[72]:

	Week	Year	GID	Name	Pos	Team	h/a	Oppt	YH points	YH salary
0	1	2018	1242	Fitzpatrick; Ryan	QB	tam	a	nor	42.28	25.0
1	1	2018	1151	Brees; Drew	QB	nor	h	tam	31.56	33.0
2	1	2018	1231	Rivers; Philip	QB	lac	h	kan	29.96	31.0
3	1	2018	1523	Mahomes II; Patrick	QB	kan	a	lac	28.34	27.0
4	1	2018	1252	Rodgers; Aaron	QB	gnb	h	chi	24.94	39.0

We can access the names of all the quarterbacks by referring to the columns `Name`

In [73]:
```
df_POS['Name']
```

Out[73]:
```
0         Fitzpatrick; Ryan
1              Brees; Drew
2            Rivers; Philip
3       Mahomes II; Patrick
4            Rodgers; Aaron
                ...
5968           Allen; Kyle
5969         Sudfeld; Nate
5970         Mannion; Sean
5971          Hoyer; Brian
5972          Hill; Taysom
Name: Name, Length: 586, dtype: object
```

Linear Least Squares works with numerical data, not strings. Eventually, we will want our predictive models to incorporate whether the player played at home or on the road, or how good their opponent was. But the columns `h/a` and `Oppt` are strings:

In [74]:
```
df_POS['h/a']
```

Out[74]:
```
0       a
1       h
2       h
3       a
4       h
       ..
5968    h
5969    h
5970    a
5971    h
5972    h
Name: h/a, Length: 586, dtype: object
```

In [75]:
```
df_POS['Oppt']
```

Out[75]:
```
0       nor
1       tam
2       kan
3       lac
4       chi
       ...
5968    atl
5969    hou
5970    ari
5971    buf
5972    pit
Name: Oppt, Length: 586, dtype: object
```

At this point, we need to make decisions about what numerical values these should take. For the home/away column:

- let's make an array with the value +1.0 when the game is played at home, and -1.0 when the game is played away.
- store this array as another column in the pandas dataframe, with label `home_away`

In [76]:
```
df_POS['home_away'] = np.where(df_POS['h/a']=='a',-1,1)
df_POS
```

Out[76]:

	Week	Year	GID	Name	Pos	Team	h/a	Oppt	YH points	YH salary	home_away
0	1	2018	1242	Fitzpatrick; Ryan	QB	tam	a	nor	42.28	25.0	-1
1	1	2018	1151	Brees; Drew	QB	nor	h	tam	31.56	33.0	1
2	1	2018	1231	Rivers; Philip	QB	lac	h	kan	29.96	31.0	1
3	1	2018	1523	Mahomes II; Patrick	QB	kan	a	lac	28.34	27.0	-1
4	1	2018	1252	Rodgers; Aaron	QB	gnb	h	chi	24.94	39.0	1
...	...	...	...	...	...	...	...	...	...	...	...
5968	16	2018	1536	Allen; Kyle	QB	car	h	atl	1.52	0.0	1
5969	16	2018	1507	Sudfeld; Nate	QB	phi	h	hou	0.00	20.0	1
5970	16	2018	1484	Mannion; Sean	QB	lar	a	ari	-0.20	20.0	-1
5971	16	2018	1336	Hoyer; Brian	QB	nwe	h	buf	-0.20	20.0	1
5972	16	2018	1530	Hill; Taysom	QB	nor	h	pit	-1.00	20.0	1

586 rows × 11 columns

For the opponents, we need some kind of information about how many points they give up to a position on average. We have compiled that information in a separate file, called `team_rankings.py` . Importing this file will give us access to a collection of dictionaries that provides this information.

After importing this file, the number `vs_2018[Pos][team]` will give us a relevant ranking.

In [77]:
```
from team_rankings import *  # asterik just means we import everything f
rom that namespace
```

We can take a look at the keys in the dictionary:

In [78]:
```
print( vs_2018.keys() )
```

```
dict_keys(['QB', 'WR', 'RB', 'TE', 'Def'])
```

Note that the keys are just the player positions. Let's see the information for the key `QB` (we have been storing this string in the variable `POS`)

In [79]:
```
vs_2018[POS]
```

Out[79]:
```
{'ari': 28,
 'atl': 1.0,
 'bal': 29.0,
 'buf': 32.0,
 'car': 9.0,
 'chi': 31.0,
 'cin': 3.0,
 'cle': 13.0,
 'dal': 24.0,
 'den': 27.0,
 'det': 15.0,
 'gnb': 12.0,
 'hou': 19.0,
 'ind': 21.0,
 'jac': 23.0,
 'kan': 5.0,
 'lac': 25.0,
 'lar': 20.0,
 'mia': 10.0,
 'min': 30.0,
 'nor': 2.0,
 'nwe': 18,
 'nyg': 16.0,
 'nyj': 6.0,
 'oak': 8,
 'phi': 11.0,
 'pit': 17.0,
 'sea': 22.0,
 'sfo': 7.0,
 'tam': 4.0,
 'ten': 26.0,
 'was': 14}
```

In [80]:
```
print(vs_2018[POS]['atl'])
print(vs_2018[POS]['buf'])
```

```
1.0
32.0
```

There are 32 football teams in the NFL.

The fact that `vs_2018['QB']['atl']` has the value 1.0, means that the Atlanta Falcons gave up the **most** points to quarterbacks on average in the 2018 season.

Since `vs_2018['QB']['buf']` has the value 32.0, this means that the Buffalo Bills gave up the **least** points to quarterbacks on average in the 2018 season.

So, we would expect a better performance out of a quarterback if he is playing the Atlanta Falcons, compared to the Buffalo Bills.

The rankings can be very different for different positions:

In [81]:
```
print(vs_2018['RB']['atl'])
print(vs_2018['RB']['buf'])
print()
print(vs_2018['WR']['atl'])
print(vs_2018['WR']['buf'])
print()
print(vs_2018['TE']['atl'])
print(vs_2018['TE']['buf'])
print()
print(vs_2018['Def']['atl'])
print(vs_2018['Def']['buf'])
print()
```

```
4.0
7.0

6.0
29.0

20.0
32.0

21.0
2.0
```

For the quarterback position (POS = 'QB'), convert the strings in the column `Oppt` into their corresponding numerical values using the dictionary `vs_2018` . Store this as another column of the pandas dataframe `oppt_rank`

In [82]:
```
def get_rank(x):
    return vs_2018[POS][x]

df_POS['oppt_rank'] = df_POS['Oppt'].apply(get_rank)
df_POS
```

Out[82]:

	Week	Year	GID	Name	Pos	Team	h/a	Oppt	YH points	YH salary	home_away	oppt_ra
0	1	2018	1242	Fitzpatrick; Ryan	QB	tam	a	nor	42.28	25.0	-1	2
1	1	2018	1151	Brees; Drew	QB	nor	h	tam	31.56	33.0	1	4
2	1	2018	1231	Rivers; Philip	QB	lac	h	kan	29.96	31.0	1	5
3	1	2018	1523	Mahomes II; Patrick	QB	kan	a	lac	28.34	27.0	-1	25
4	1	2018	1252	Rodgers; Aaron	QB	gnb	h	chi	24.94	39.0	1	31
...	...	...	...	...	...	...	...	...	...	...	...	
5968	16	2018	1536	Allen; Kyle	QB	car	h	atl	1.52	0.0	1	
5969	16	2018	1507	Sudfeld; Nate	QB	phi	h	hou	0.00	20.0	1	19
5970	16	2018	1484	Mannion; Sean	QB	lar	a	ari	-0.20	20.0	-1	28
5971	16	2018	1336	Hoyer; Brian	QB	nwe	h	buf	-0.20	20.0	1	32
5972	16	2018	1530	Hill; Taysom	QB	nor	h	pit	-1.00	20.0	1	17

586 rows × 12 columns

Now, players' names will be repeated in the array `names` for every game they played. We will find it convenient to have another array collecting the names without these repeats. We'll use `pandas.Series.unique` to do this.

In [83]:
```
unique_players = df_POS['Name'].unique()
len(unique_players)
```

Out[83]: 73

So 73 quarterbacks played in 2018. But there are only 32 teams! Who are all these people?

In [84]:
```
print(unique_players[7])
print(unique_players[72])
```

```
Brady; Tom
Sudfeld; Nate
```

I know who Tom Brady is, but I've never heard of Nate Sudfeld. Let's count how many times a players played a game.

We can use `groupby` to group players by Name, and then count the number of times each player appears:

In [85]:
```
df_POS.groupby('Name')['Name'].count()
```

Out[85]:
```
Name
Allen; Brandon      1
Allen; Josh        11
Allen; Kyle         1
Anderson; Derek     2
Barkley; Matt       1
                   ..
Webb; Joe           2
Weeden; Brandon     1
Wentz; Carson      11
Wilson; Russell    15
Winston; Jameis    10
Name: Name, Length: 73, dtype: int64
```

We want to add the frequency (game count) back to the original dataframe, and for that we will use transform to return an aligned index.

In [86]:
```
df_POS['game_count'] = df_POS.groupby('Name')['Name'].transform('count')
df_POS
```

Out[86]:

	Week	Year	GID	Name	Pos	Team	h/a	Oppt	YH points	YH salary	home_away	oppt_ra
0	1	2018	1242	Fitzpatrick; Ryan	QB	tam	a	nor	42.28	25.0	-1	2
1	1	2018	1151	Brees; Drew	QB	nor	h	tam	31.56	33.0	1	4
2	1	2018	1231	Rivers; Philip	QB	lac	h	kan	29.96	31.0	1	5
3	1	2018	1523	Mahomes II; Patrick	QB	kan	a	lac	28.34	27.0	-1	25
4	1	2018	1252	Rodgers; Aaron	QB	gnb	h	chi	24.94	39.0	1	31
...	...	...	...	...	...	...	...	...	...	...	...	
5968	16	2018	1536	Allen; Kyle	QB	car	h	atl	1.52	0.0	1	
5969	16	2018	1507	Sudfeld; Nate	QB	phi	h	hou	0.00	20.0	1	19
5970	16	2018	1484	Mannion; Sean	QB	lar	a	ari	-0.20	20.0	-1	28
5971	16	2018	1336	Hoyer; Brian	QB	nwe	h	buf	-0.20	20.0	1	32
5972	16	2018	1530	Hill; Taysom	QB	nor	h	pit	-1.00	20.0	1	17

586 rows × 13 columns

Note that Nate Sudfeld only played in 1 game in 2018. He probably took over when the starter was injured, or when his team was involved in a lopsided game. We probably want to remove his data, since it won't be very helpful.

Let's us create an array of the names of all the players that are relevant to our analysis. For that, we will exclude the names for all the players that participated in less than `min_games` .

In [87]:
```
min_games = 5
relevant_players =  df_POS[df_POS['game_count']>=min_games]['Name'].uniq
ue()
print(len(relevant_players))
relevant_players
```

```
43
```

Out[87]:
```
array(['Fitzpatrick; Ryan', 'Brees; Drew', 'Rivers; Philip',
       'Mahomes II; Patrick', 'Rodgers; Aaron', 'Wilson; Russell',
       'Brady; Tom', 'Keenum; Case', 'Flacco; Joe', 'Luck; Andrew',
       'Cousins; Kirk', 'Smith; Alex', 'Newton; Cam', 'Dalton; Andy',
       'Goff; Jared', 'Tannehill; Ryan', 'Darnold; Sam', 'Bortles; Blak
e',
       'Trubisky; Mitchell', 'Watson; Deshaun', 'Stafford; Matthew',
       'Roethlisberger; Ben', 'Ryan; Matt', 'Carr; Derek',
       'Prescott; Dak', 'Manning; Eli', 'Allen; Josh', 'Jackson; Lama
r',
       'Gabbert; Blaine', 'Mariota; Marcus', 'Hill; Taysom',
       'Wentz; Carson', 'Mayfield; Baker', 'Rosen; Josh',
       'Beathard; C.J.', 'Winston; Jameis', 'Osweiler; Brock',
       'Daniel; Chase', 'Dobbs; Joshua', 'Kessler; Cody', 'Driskel; Jef
f',
       'Heinicke; Taylor', 'Mullens; Nick'], dtype=object)
```

Now we only consider 43 quarterbacks playing in 2018.

Let's put all of this together!

Write a function `prepare_data` that creates the dataframe `df_POS` for a given player position. The function also returns as an argument the list of relevant unique players.

In [88]:
```
def prepare_data(ff_data,POS,min_games):
    # returns (new_df,relevant_players) as described above
    #clear
    df_POS = ff_data[ff_data['Pos'] == POS].copy()
    df_POS['home_away'] = np.where(df_POS['h/a']=='a',-1,1)
    df_POS['oppt_rank'] = df_POS['Oppt'].apply(get_rank)
    df_POS['game_count'] = df_POS.groupby('Name')['Name'].transform('cou
nt')
    df_new = df_POS[df_POS['game_count']>=min_games].copy()
    relevant_players =  df_POS[df_POS['game_count']>=min_games]['Name'].
unique()
    return(df_POS, relevant_players)
```

Test out that your function works as expected:

In [89]:
```
df_test,players_test = prepare_data(ff_2018,'WR',3)
df_test
```

Out[89]:

	Week	Year	GID	Name	Pos	Team	h/a	Oppt	YH points	YH salary	home_away	oppt_rank
144	1	2018	5485	Hill; Tyreek	WR	kan	a	lac	38.8	28.0	-1	25.0
145	1	2018	5459	Thomas; Michael	WR	nor	h	tam	30.0	37.0	1	4.0
146	1	2018	3770	Jackson; DeSean	WR	tam	a	nor	29.1	14.0	-1	2.0
147	1	2018	5125	Cobb; Randall	WR	gnb	h	chi	24.7	15.0	1	31.0
148	1	2018	5212	Stills; Kenny	WR	mia	h	ten	24.6	17.0	1	26.0
...	...	...	...	...	...	...	...	...	...	...	...	..
6231	16	2018	5570	Cole; Keelan	WR	jac	a	mia	0.0	10.0	-1	10.0
6232	16	2018	5387	Hardy; Justin	WR	atl	a	car	0.0	10.0	-1	9.0
6233	16	2018	5692	Beebe; Chad	WR	min	a	det	0.0	10.0	-1	15.0
6234	16	2018	5595	Hall; Marvin	WR	atl	a	car	0.0	10.0	-1	9.0
6235	16	2018	5684	Moore; J'Mon	WR	gnb	a	nyj	-2.0	10.0	-1	6.0

2278 rows × 13 columns

Simple Model - Last n games

We'll start with a simple linear model. For now, we will keep using our example where we constructed a dataset for quarterbacks in the variable `df_POS` , along with `relevant_players`

The points scored in the previous n games will be the only data considered when making a prediction. Let's look at what the model would look like for only one player, say Andy Dalton, with $n = 3$.

In [90]:
```
pl = relevant_players[13]
pl_points = df_POS[df_POS['Name']==pl]['YH points'].values

print('Player:', pl)
print('Points:', pl_points)
```

```
Player: Dalton; Andy
Points: [17.52 26.6  18.08 25.78 13.92 17.16  8.92 20.2   8.92 19.34
9.1 ]
```

Andy Dalton played 11 games. So we could try to build a model that predicted the points he scored in his 4th game, based on his first 3, and similarly try to predict the points he scored in the 5th games based on games 2,3, and 4.

I.e. a "local" least squares system might look something like

$$\mathbf{A}\mathbf{x} \cong \mathbf{b}$$

where

$$\mathbf{A} = \begin{pmatrix} 17.52 & 26.6 & 18.08 \\ 26.6 & 18.08 & 25.78 \\ 18.08 & 25.78 & 13.92 \\ 25.78 & 13.92 & 17.16 \\ 13.92 & 17.16 & 8.92 \\ 17.16 & 8.92 & 20.2 \\ 8.92 & 20.2 & 8.92 \\ 20.2 & 8.92 & 19.34 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 25.78 \\ 13.92 \\ 17.16 \\ 8.92 \\ 20.2 \\ 8.92 \\ 19.34 \\ 9.1 \end{pmatrix}$$

This was with $n = 3$ games. If instead, we base our "local" least squares on the previous $n = 4$ games, then our system would instead look like:

$$\mathbf{A} = \begin{pmatrix} 17.52 & 26.6 & 18.08 & 25.78 \\ 26.6 & 18.08 & 25.78 & 13.92 \\ 18.08 & 25.78 & 13.92 & 17.16 \\ 25.78 & 13.92 & 17.16 & 8.92 \\ 13.92 & 17.16 & 8.92 & 20.2 \\ 17.16 & 8.92 & 20.2 & 8.92 \\ 8.92 & 20.2 & 8.92 & 19.34 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 13.92 \\ 17.16 \\ 8.92 \\ 20.2 \\ 8.92 \\ 19.34 \\ 9.1 \end{pmatrix}$$

Write a function that generates this local system for a given (relevant) player. Use the example above to debug your function (i.e., data for Andy Dalton)

In [91]:
```
def player_point_history(df, pl, n_games):
    # df: dataframe
    # rel_player (string): name of a player
    # n_games (int): number of games used for the prediction
    # clear
    pts = df[df['Name']==pl]['YH points'].values

    m = pts[n_games:].shape[0]
    A = np.zeros((m,n_games))
    for k in range(n_games):
        A[:,k] = pts[k:-n_games + k]
    b = pts[n_games:]

    return A,b

A,b = player_point_history(df_POS, relevant_players[13], 4)
print(A)
print(b)
```

```
[[17.52 26.6  18.08 25.78]
 [26.6  18.08 25.78 13.92]
 [18.08 25.78 13.92 17.16]
 [25.78 13.92 17.16  8.92]
 [13.92 17.16  8.92 20.2 ]
 [17.16  8.92 20.2   8.92]
 [ 8.92 20.2   8.92 19.34]]
[13.92 17.16  8.92 20.2   8.92 19.34  9.1 ]
```

Now, with this function, we can loop over the relevant players, generate their local systems, and "stack" them on top of each other to generate the global system. We'll do this with $n = 3$

In [92]:
```
n_games = 3

# empty array for right hand side of size M x 1
pts_scored = np.array([])

# empty array for matrix of size M x n_games.  We had to reshape to size
0 x n_games to allow for "stacking"
game_hist = np.array([]).reshape(0,n_games)

for pl in relevant_players:
    # generate local system
    a,c = player_point_history(df_POS,pl,n_games)

    # use numpy.append to append local system to global vector
    pts_scored = np.append(pts_scored,c)

    # use numpy.vstack (i.e. "vertical stack") to stack the global matri
x and the local matrix
    game_hist = np.vstack((game_hist,a))

print(pts_scored.shape)
print(game_hist.shape)
```

```
(383,)
(383, 3)
```

When should we start a player?

It would be an overly ambitious task to try to predict a players exact point total. What we can do instead is set a "threshold". I.e. if a player's points exceed this threshold, then we can deem them "startable". If they don't exceed this threshold, then we should look choose a different player.

What threshold should we use? That's debatable, but I've compiled the following dictionary based on additional data I collected from nfl.com.

In [93]:
```
start_threshold = {'QB': 19.3999, 'RB': 14.599, 'WR': 15.099, 'TE': 7.89
9, 'Def': 7.499}
```

So, if a quarterback scores more than 19.3999, we declare them startable. If a defense scores less than 7.499, then we should pick a different defense, etc.

We can finally set up our least squares system. Set the matrix `A` to the variable `game_hist` defined above. The components of the vector `b` should have a value of +1.0 if the corresponding component of `pts_scored` exceeds the threshold, and -1.0 if it lies below the threshold. (I chose the thresholds so that it is impossible for the points to equal the threshold).

Set up the right hand side vector, and solve the Linear Least Squares problem for $\mathbf{x}$. You can use `numpy.linalg.lstsq` to compute the least-squares solution. Then compute a numpy array `b_predict` that tests how this linear model performs on the data.

In [94]:
```
threshold = start_threshold[POS]
A = game_hist

# clear
b = np.sign(pts_scored - threshold)
LSTQ = la.lstsq(A,b,rcond=None)
x = LSTQ[0]
b_predict = np.sign(A@x)
```

We can have the following situations:

- The prediction tells you to start a player that ends up performing poorly (a "false positive")
- The prediction tells you to exclude a player that ends up performing well (a "false negative")
- The prediction tells you to start a player that ends up performing well (a correct prediction)

Compute the number of false positives, false negatives, and correct prediction. What percentage of each do we obtain on the data?

In [95]:
```
# clear
false_positive = np.sum(b_predict > b)
false_negative = np.sum(b > b_predict)
correct_prediction = np.sum(b == b_predict)

print(false_positive)
print(false_negative)
print(correct_prediction)
print()
print(false_positive/b.shape[0])
print(false_negative/b.shape[0])
print(correct_prediction/b.shape[0])
```

```
13
138
232

0.033942558746736295
0.360313315926893
0.6057441253263708
```

The model is only correct 60.57% of the time. However, it only return a "false positive" 3.39% of the time, which is very nice: if the model tells you to start a player, there's a good chance you will be happy with the results.

Let's put it all together into a single function. This will mostly be copying and pasting from above. The function should return the variables `A`, `b`, `x`.

In [96]:
```
def linear_predictor(ff_data, Pos, min_games, n_games, threshold):
    # clear

    df,relevant_players = prepare_data(ff_data,Pos,min_games)

    pts_scored = np.array([])
    game_hist = np.array([]).reshape(0,n_games)

    for pl in relevant_players:
        a,c = player_point_history(df,pl,n_games)
        pts_scored = np.append(pts_scored,c)
        game_hist = np.vstack((game_hist,a))

    A = game_hist
    b = np.sign(pts_scored - threshold)

    LSTQ = np.linalg.lstsq(A,b,rcond = None)
    x = LSTQ[0]
    #

    return A, b, x
```

We can call the routine for any position, and we can tweak the number of `min_games` and `n_games`. You can also tweak the threshold. Try changing the input variables and see how this affects model accuracy

In [97]:
```
Pos = 'WR'
min_games = 5
n_games = 3
threshold = start_threshold[Pos]

A, b, x = linear_predictor(ff_2018, Pos, min_games, n_games, threshold)

# clear
b_predict = np.sign(A@x)

false_negative = np.sum(b > b_predict)
false_positive = np.sum(b_predict > b)
correct_prediction = np.sum(b == b_predict)

print(false_negative)
print(false_positive)
print(correct_prediction)
print()
print(false_negative/b.shape[0])
print(false_positive/b.shape[0])
print(correct_prediction/b.shape[0])
```

```
201
144
1287

0.12316176470588236
0.08823529411764706
0.7886029411764706
```

Notice we didn't make use of the fact that a player is playing on home or on the road, or the ranking of the opponent. Let's try to enrich the features used in this problem to include this data. Let's go back to Andy Dalton:

In [98]:
```
pl = relevant_players[13]
pl_points = df_POS[df_POS['Name']==pl]['YH points'].values
pl_home_away = df_POS[df_POS['Name']==pl]['home_away'].values
pl_oppt_rank = df_POS[df_POS['Name']==pl]['oppt_rank'].values

print('Player:', pl)
print('Points:', pl_points)
print('Location:', pl_home_away)
print('Opp Rank:', pl_oppt_rank)
```

```
Player: Dalton; Andy
Points: [17.52 26.6  18.08 25.78 13.92 17.16  8.92 20.2   8.92 19.34
9.1 ]
Location: [-1  1 -1 -1  1  1 -1  1  1 -1  1]
Opp Rank: [21. 29.  9.  1. 10. 17.  5.  4.  2. 29. 13.]
```

When $n = 3$ we had the following system when we only took previous games played:

$$\mathbf{A} = \begin{pmatrix} 17.52 & 26.6 & 18.08 \\ 26.6 & 18.08 & 25.78 \\ 18.08 & 25.78 & 13.92 \\ 25.78 & 13.92 & 17.16 \\ 13.92 & 17.16 & 8.92 \\ 17.16 & 8.92 & 20.2 \\ 8.92 & 20.2 & 8.92 \\ 20.2 & 8.92 & 19.34 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 25.78 \\ 13.92 \\ 17.16 \\ 8.92 \\ 20.2 \\ 8.92 \\ 19.34 \\ 9.1 \end{pmatrix}$$

With the location and opponent data, it should now look like this:

$$\mathbf{A} = \begin{pmatrix} 17.52 & 26.6 & 18.08 & -1 & 1 \\ 26.6 & 18.08 & 25.78 & 1 & 10 \\ 18.08 & 25.78 & 13.92 & 1 & 17 \\ 25.78 & 13.92 & 17.16 & -1 & 5 \\ 13.92 & 17.16 & 8.92 & 1 & 4 \\ 17.16 & 8.92 & 20.2 & 1 & 2 \\ 8.92 & 20.2 & 8.92 & -1 & 29 \\ 20.2 & 8.92 & 19.34 & 1 & 13 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 25.78 \\ 13.92 \\ 17.16 \\ 8.92 \\ 20.2 \\ 8.92 \\ 19.34 \\ 9.1 \end{pmatrix}$$

Create an enriched linear regression, by adding these two extra columns to the matrix $\mathbf{A}$. The routine should return `A` with the two added columns. It should also return the right hand side `b` and least-squares solution `x`.

In [99]:
```
def linear_predictor_enriched(ff_data, Pos, min_games, n_games, threshold):
    # clear

    df,relevant_players = prepare_data(ff_data,Pos,min_games)

    pts_scored = np.array([])
    game_hist = np.array([]).reshape(0,n_games+2)

    for pl in relevant_players:
        a,c = player_point_history(df,pl,n_games)
        location = df[df['Name']==pl]['home_away'].values
        opponent = df[df['Name']==pl]['oppt_rank'].values
        last_two_columns = np.vstack((location[n_games:],opponent[n_games:])).T
        anew = np.hstack((a,last_two_columns ))

        pts_scored = np.append(pts_scored,c)
        game_hist = np.vstack((game_hist,anew))


    b = np.sign(pts_scored - threshold)
    A = game_hist

    LSTQ = np.linalg.lstsq(A,b,rcond = None)
    x = LSTQ[0]
    #

    return A, b, x
```

This enriched version is considerably better for running backs, with our standard inputs:

In [100]:
```
Pos = 'RB'
min_games = 5
n_games = 3
threshold = start_threshold[Pos]

A, b, x  = linear_predictor(ff_2018, Pos, min_games, n_games, threshold)

b_predict = np.sign(A@x)
false_negative = np.sum(b > b_predict)
false_positive = np.sum(b_predict > b)
correct_prediction = np.sum(b == b_predict)

print('Standard Model')
print('Fraction of false negatives:    ', false_negative/b.shape[0])
print('Fraction of false positives:    ', false_positive/b.shape[0])
print('Fraction of correct predictions:', correct_prediction/b.shape[0])
print()

A, b, x = linear_predictor_enriched(ff_2018, Pos, min_games, n_games, th
reshold)

b_predict = np.sign(A@x)

false_negative = np.sum(b > b_predict)
false_positive = np.sum(b_predict > b)
correct_prediction = np.sum(b == b_predict)
```

```
Standard Model
Fraction of false negatives:     0.161400512382579
Fraction of false positives:     0.161400512382579
Fraction of correct predictions: 0.677198975234842
```

But it's not very effective for quarterbacks:

In [101]:
```
Pos = 'WR'
min_games = 10
n_games = 1
threshold = start_threshold[Pos]

A, b, x  = linear_predictor(ff_2018, Pos, min_games, n_games, threshold)

b_predict = np.sign(A@x)
false_negative = np.sum(b > b_predict)
false_positive = np.sum(b_predict > b)
correct_prediction = np.sum(b == b_predict)

print('Standard Model')
print('Fraction of false negatives:    ', false_negative/b.shape[0])
print('Fraction of false positives:    ', false_positive/b.shape[0])
print('Fraction of correct predictions:', correct_prediction/b.shape[0])
print()

A, b, x = linear_predictor_enriched(ff_2018, Pos, min_games, n_games, th
reshold)

b_predict = np.sign(A@x)

false_negative = np.sum(b > b_predict)
false_positive = np.sum(b_predict > b)
correct_prediction = np.sum(b == b_predict)

print('Enriched Model')
print('Fraction of false negatives:    ', false_negative/b.shape[0])
print('Fraction of false positives:    ', false_positive/b.shape[0])
print('Fraction of correct predictions:', correct_prediction/b.shape[0])
print()
```

```
Standard Model
Fraction of false negatives:     0.13917216556688664
Fraction of false positives:     0.20695860827834434
Fraction of correct predictions: 0.6538692261547691

Enriched Model
Fraction of false negatives:     0.131373725254949
Fraction of false positives:     0.008998200359928014
Fraction of correct predictions: 0.859628074385123
```

The number of false positives has shot up dramatically. Despite the (slightly) better accuracy, I would probably avoid this one.

It seems that running backs are more "matchup-dependent" than quarterbacks. That is, where they are playing and how good the other team is are bigger factors in their performance compared to quarterbacks.

Validation set

Of course, you never want to conclude anything about your model based on the data you used to construct it. You should validate its accuracy on a different data set. We can do so on this years fantasy football data. We can also select the optimal **hyperparameters** (a fancy word for parameters) based on this validation set.

Some questions to ask as you test the model on the validation set:

- Should we include the home/away and opponent data or not?
- Is our decision to exclude players that have played less than 5 games a good one? Should we bump that number up to 7 games? Or down to 3?
- How many games should we include in our history? Is 3 games really the best choice? What about 5? What about just the last game?

I.e. the inclusion of the extra data, the minimum number of games, and the history length are the **hyperparameters** for this model.

In [102]:
```
ff_2019 = pd.read_csv('./additional_files/FF-data-2019.csv')

# position
Pos = 'WR'

# these are your hyperparameters
min_games = 5
n_games = 2
enriched = True

# build model on 2018 data and retrieve least squares solution x
if enriched:
    OUT_2018 = linear_predictor_enriched(ff_2018, Pos, min_games,n_games
,threshold)
    x = OUT_2018[2]
else:
    OUT_2018 = linear_predictor(ff_2018, Pos, min_games,n_games,threshol
d)
    x = OUT_2018[2]


# retrieve Data matrix A and outcomes vector b using 2019 data
if enriched:
    OUT_2019 = linear_predictor_enriched(ff_2019, Pos, min_games,n_games
,threshold)
    A,b = OUT_2019[0], OUT_2019[1]
else:
    OUT_2019 = linear_predictor(ff_2019, Pos, min_games,n_games,threshol
d)
    A,b = OUT_2019[0], OUT_2019[1]

# assess model
b_predict = np.sign(A@x)

false_negative = np.sum(b > b_predict)
false_positive = np.sum(b_predict > b)
correct_prediction = np.sum(b == b_predict)
print('Fraction of false negatives:    ', false_negative/b.shape[0])
print('Fraction of false positives:    ', false_positive/b.shape[0])
print('Fraction of correct predictions:', correct_prediction/b.shape[0])
print()
```

```
Fraction of false negatives:     0.10348258706467661
Fraction of false positives:     0.004975124378109453
Fraction of correct predictions: 0.891542288557214
```

In [ ]:

In [ ]:

In [ ]: