Lab: Simple Linear Regression

CMSE 381 - Spring 2024

In the today's lectures, we are focused on simple linear regression, that is, fitting models of the form

$$Y = \beta_0 + \beta_1 X_1 + \varepsilon$$

In this lab, we will use two different tools for linear regression.

- Scikit learn is arguably the most used tool for machine learning in python
- Statsmodels provides many of the statisitcial tests we've been learning in class

0. A note on datasets and ethics

For much of this course, we will follow the labs outlined in the textbook at the end of each section (albeit, translated into python). However, there are many portions of this book that rely on the `Boston` data set. Although this dataset has been a standard example for a long time, often used for teaching linear regression, it has some major issues with assumptions based around race and housing. An excellent in-depth description of issues in the data set can be found in this medium post from a few years ago. More recently, the data set has marked as deprecated in scikit-learn 1.0, which essentially means that anyone loading it will encounter a warning, and is marked for removal in version 1.2. For these reasons, we will not be using the dataset in this class.

1. The Dataset

In [1]:
```
# As always, we start with our favorite standard imports.

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

In this module, we will be using the `Diabetes` data set. while we could download a csv to put in the correct folder yadda yadda yadda, because this is a commonly used test data set, it's available in `scikit-learn` for us to use without any cleanup. Yay!

In [2]:
```
from sklearn.datasets import load_diabetes
```

In [3]:
```
diabetes = load_diabetes(as_frame=True)
```

In [4]:
```
# Notice that this loads in a lot of info into what is essentially a beastly dictiona
print(type(diabetes))
diabetes
```

```
<class 'sklearn.utils._bunch.Bunch'>
```

Out[4]:
```
{'data':           age       sex       bmi        bp        s1        s2        s3
₩
 0    0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
 1   -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
 2    0.085299  0.050680  0.044451 -0.005670 -0.045599 -0.034194 -0.032356
 3   -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
 4    0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142
 ..        ...       ...       ...       ...       ...       ...       ...
 437  0.041708  0.050680  0.019662  0.059744 -0.005697 -0.002566 -0.028674
 438 -0.005515  0.050680 -0.015906 -0.067642  0.049341  0.079165 -0.028674
 439  0.041708  0.050680 -0.015906  0.017293 -0.037344 -0.013840 -0.024993
 440 -0.045472 -0.044642  0.039062  0.001215  0.016318  0.015283 -0.028674
 441 -0.045472 -0.044642 -0.073030 -0.081413  0.083740  0.027809  0.173816

            s4        s5        s6
 0   -0.002592  0.019907 -0.017646
 1   -0.039493 -0.068332 -0.092204
 2   -0.002592  0.002861 -0.025930
 3    0.034309  0.022688 -0.009362
 4   -0.002592 -0.031988 -0.046641
 ..        ...       ...       ...
 437 -0.002592  0.031193  0.007207
 438  0.034309 -0.018114  0.044485
 439 -0.011080 -0.046883  0.015491
 440  0.026560  0.044529 -0.025930
 441 -0.039493 -0.004222  0.003064

 [442 rows x 10 columns],
 'target': 0      151.0
 1       75.0
 2      141.0
 3      206.0
 4      135.0
        ...
 437    178.0
 438    104.0
 439    132.0
 440    220.0
 441     57.0
 Name: target, Length: 442, dtype: float64,
 'frame':           age       sex       bmi        bp        s1        s2        s3
₩
 0    0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
 1   -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
 2    0.085299  0.050680  0.044451 -0.005670 -0.045599 -0.034194 -0.032356
 3   -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
 4    0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142
 ..        ...       ...       ...       ...       ...       ...       ...
 437  0.041708  0.050680  0.019662  0.059744 -0.005697 -0.002566 -0.028674
 438 -0.005515  0.050680 -0.015906 -0.067642  0.049341  0.079165 -0.028674
 439  0.041708  0.050680 -0.015906  0.017293 -0.037344 -0.013840 -0.024993
 440 -0.045472 -0.044642  0.039062  0.001215  0.016318  0.015283 -0.028674
 441 -0.045472 -0.044642 -0.073030 -0.081413  0.083740  0.027809  0.173816

            s4        s5        s6  target
 0   -0.002592  0.019907 -0.017646   151.0
 1   -0.039493 -0.068332 -0.092204    75.0
 2   -0.002592  0.002861 -0.025930   141.0
 3    0.034309  0.022688 -0.009362   206.0
 4   -0.002592 -0.031988 -0.046641   135.0
 ..        ...       ...       ...     ...
 437 -0.002592  0.031193  0.007207   178.0
 438  0.034309 -0.018114  0.044485   104.0
 439 -0.011080 -0.046883  0.015491   132.0
```

```
440  0.026560  0.044529 -0.025930   220.0
441 -0.039493 -0.004222  0.003064    57.0

 [442 rows x 11 columns],
 'DESCR': '.. _diabetes_dataset:\n\nDiabetes dataset\n----------------\n\nTen baseline variables, age, sex, body mass index, average blood\npressure, and six blood serum measurements were obtained for each of n =\n442 diabetes patients, as well as the response of interest, a\nquantitative measure of disease progression one year after baseline.\n\n**Data Set Characteristics:**\n\n  :Number of Instances: 442\n\n  :Number of Attributes: First 10 columns are numeric predictive values\n\n  :Target: Column 11 is a quantitative measure of disease progression one year after baseline\n\n  :Attribute Information:\n      - age     age in years\n      - sex\n      - bmi     body mass index\n      - bp      average blood pressure\n      - s1      tc, total serum cholesterol\n      - s2      ldl, low-density lipoproteins\n      - s3      hdl, high-density lipoproteins\n      - s4      tch, total cholesterol / HDL\n      - s5      ltg, possibly log of serum triglycerides level\n      - s6      glu, blood sugar level\n\nNote: Each of these 10 feature variables have been mean centered and scaled by the standard deviation times the square root of `n_samples` (i.e. the sum of squares of each column totals 1).\n\nSource URL:\nhttps://www4.stat.ncsu.edu/~boos/var.select/diabetes.html\n\nFor more information see:\nBradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least Angle Regression," Annals of Statistics (with discussion), 407-499.\n(https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)\n',
 'feature_names': ['age',
  'sex',
  'bmi',
  'bp',
  's1',
  's2',
  's3',
  's4',
  's5',
  's6'],
 'data_filename': 'diabetes_data_raw.csv.gz',
 'target_filename': 'diabetes_target.csv.gz',
 'data_module': 'sklearn.datasets.data'}
```

In [5]:

```python
# But we can immediately get it into a pandas data frame for ease of use as follows
diabetes_df = pd.DataFrame(diabetes.data, columns = diabetes.feature_names)
diabetes_df['target'] = pd.Series(diabetes.target)

diabetes_df
```

Out[5]:

	age	sex	bmi	bp	s1	s2	s3	s4	
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.0199
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.0683
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.0028
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.0226
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.0319
...	...	...	...	...	...	...	...	...	
437	0.041708	0.050680	0.019662	0.059744	-0.005697	-0.002566	-0.028674	-0.002592	0.0311
438	-0.005515	0.050680	-0.015906	-0.067642	0.049341	0.079165	-0.028674	0.034309	-0.0181
439	0.041708	0.050680	-0.015906	0.017293	-0.037344	-0.013840	-0.024993	-0.011080	-0.0468
440	-0.045472	-0.044642	0.039062	0.001215	0.016318	0.015283	-0.028674	0.026560	0.0445
441	-0.045472	-0.044642	-0.073030	-0.081413	0.083740	0.027809	0.173816	-0.039493	-0.0042

442 rows × 11 columns

Info about the data set

Look up the documentation about the dataset here:

From https://scikit-learn.org/stable/datasets/toy_dataset.html#diabetes-dataset

✅ **Q:**

- Write a brief description of the data set.

theres 442 samples of peoples diabetes with different columns

- What do the columns `s1` through `s6` correspond to?

s1 tc, total serum cholesterol

s2 ldl, low-density lipoproteins

s3 hdl, high-density lipoproteins

s4 tch, total cholesterol / HDL

s5 ltg, possibly log of serum triglycerides level

s6 glu, blood sugar level

- Which of the available variables are quantitative? Which are categorical?

i am not sure about it but continus data can be used

- What is the `target` that we are trying to predict?

targey is people with diabetes

Your answer here

---

2. Getting familiar with the data

The following command should show you the top of your data frame.

In [6]:
```
diabetes_df.head()
```

Out[6]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019907
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068332
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	-0.002592	0.002861
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022688
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031988

✅ **Q:** Do some basic data exploration. How many data points do we have? How many variables do we have? Are there any data points with missing data?

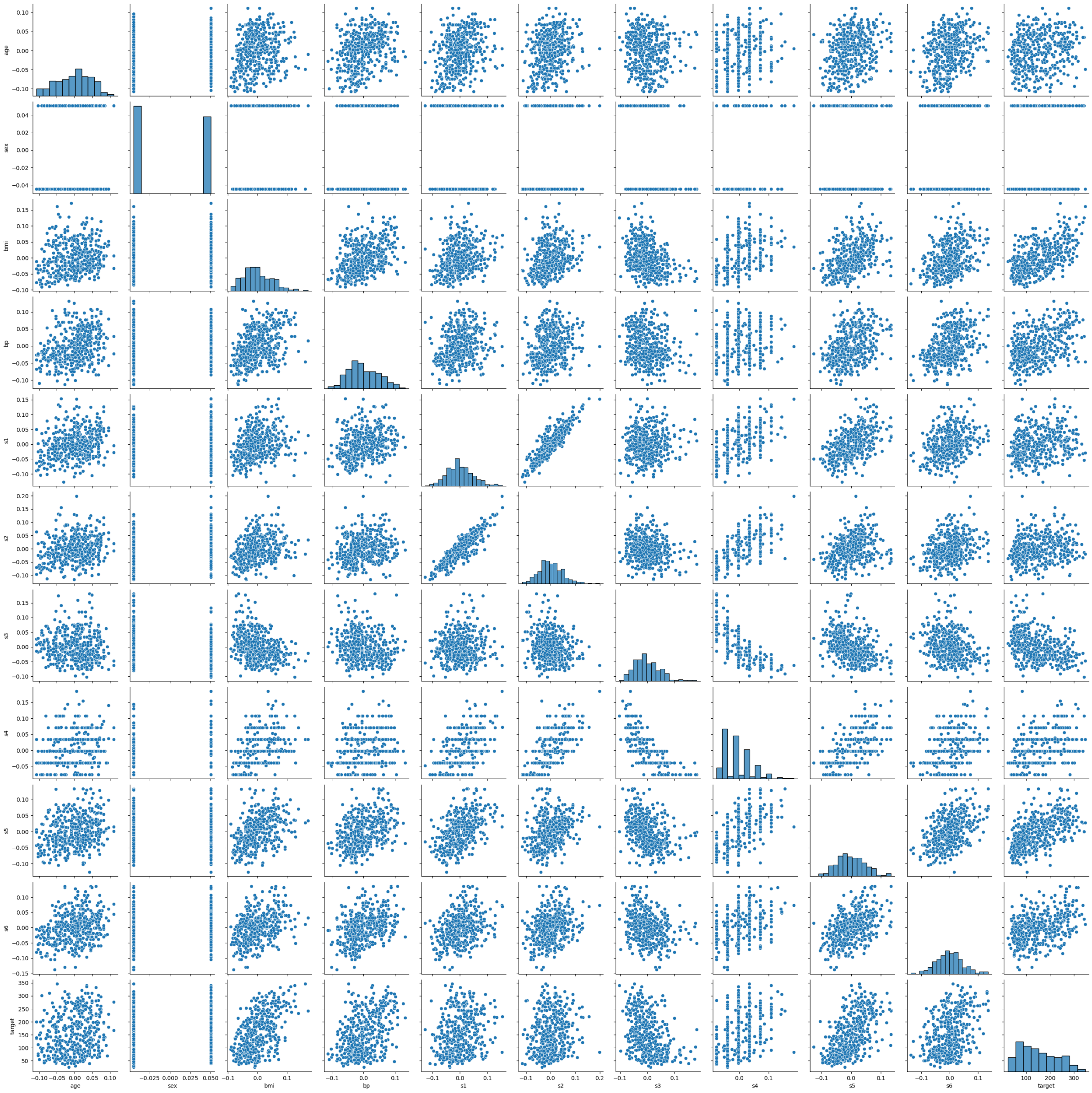
442 rows × 11 columns

✅ **Q:** Use the seaborn `sns.pairplot` command to look at relationships between the variables. Are there pairs of variables that appear to be related?

Your answer here

In [20]:
```
sns.pairplot(diabetes_df)
plt.show()
```

```
c:\Users\siony\anaconda3\Lib\site-packages\seaborn\axisgrid.py:118: UserWarning: The
figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```



---

3. Simple Linear Regression

We're now going to fig to a simple linear regression to the models

$$\texttt{target} = \beta_0 + \beta_1 \cdot \texttt{s1}$$

and

$$\texttt{target} = \beta_0 + \beta_1 \cdot \texttt{s5}$$

where the variables are

- `s1`: tc, total serum cholesterol
- `s5`: ltg, possibly log of serum triglycerides level.

Let's start by looking at using `s5` to predict `target`.

In [7]:
```
from sklearn.linear_model import LinearRegression
```

```
# sklearn actually likes being handed numpy arrays more than
# pandas dataframes, so we'll extract the bits we want and just pass it that.
X = diabetes_df['s5'].values
X = X.reshape([len(X),1])
y = diabetes_df['target'].values
y = y.reshape([len(y),1])

# This code works by first creating an instance of
# the linear regression class
reg = LinearRegression()
# Then we pass in the data we want it to use to fit.
reg.fit(X,y)
```

Out[7]:

```
▾ LinearRegression
LinearRegression()
```

What the fork, nothing seems to have happened? Well actually, we first created an instance of the regression class, which is just a collection of the model functionality waiting to be trained. When we run the `fit` command with data handed in, it actually figures out the best choice of coefficients for our particular data. Once they're found, we can extract them from the class as follows.

In [8]:

```
# We can find the intercept and coefficient information
# from the regression class as follows.

print(reg.coef_)
print(reg.intercept_)
```

```
[[916.13737455]]
[152.13348416]
```

✅ **Q:**

- What is the model using these coefficients? That is, write down the function $\hat{f}$ explicitly.
- What is the prediction by the model for $\texttt{s5} = 0.05$?

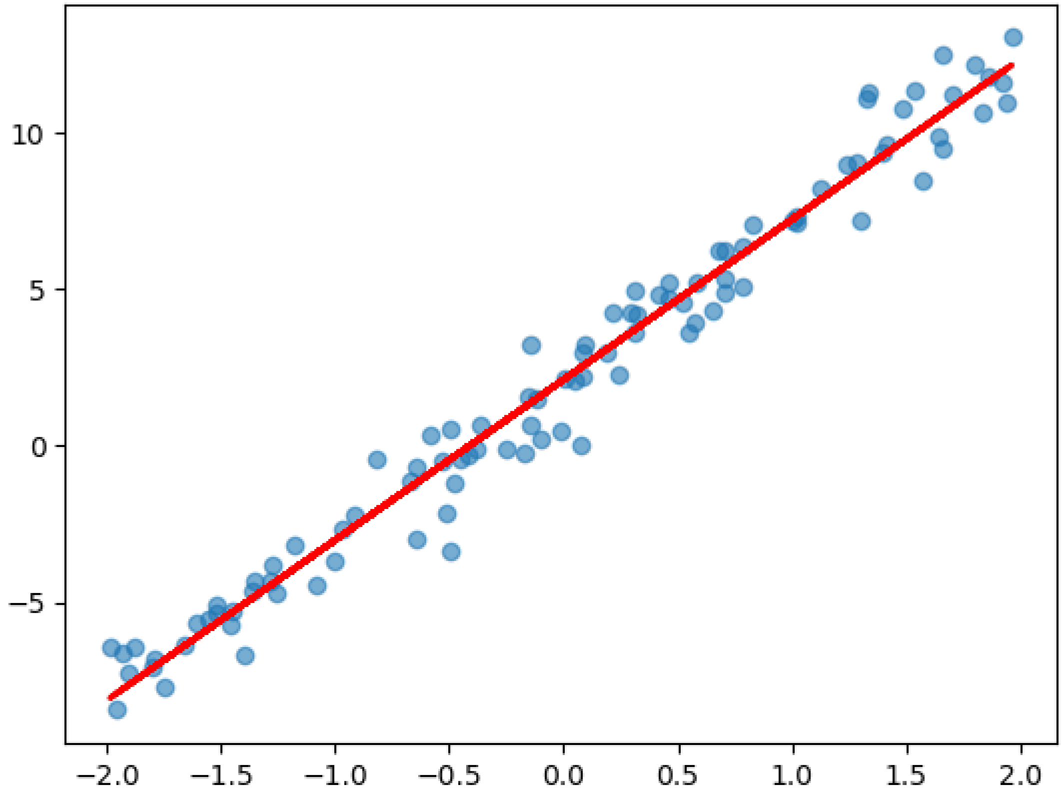
In [9]:

```
# Your answer here
```

✅ **Q:** Overlay a plot of your predicted model (your line) on a scatter plot of the data used. Does linear seem like a good assumption?

In [23]:

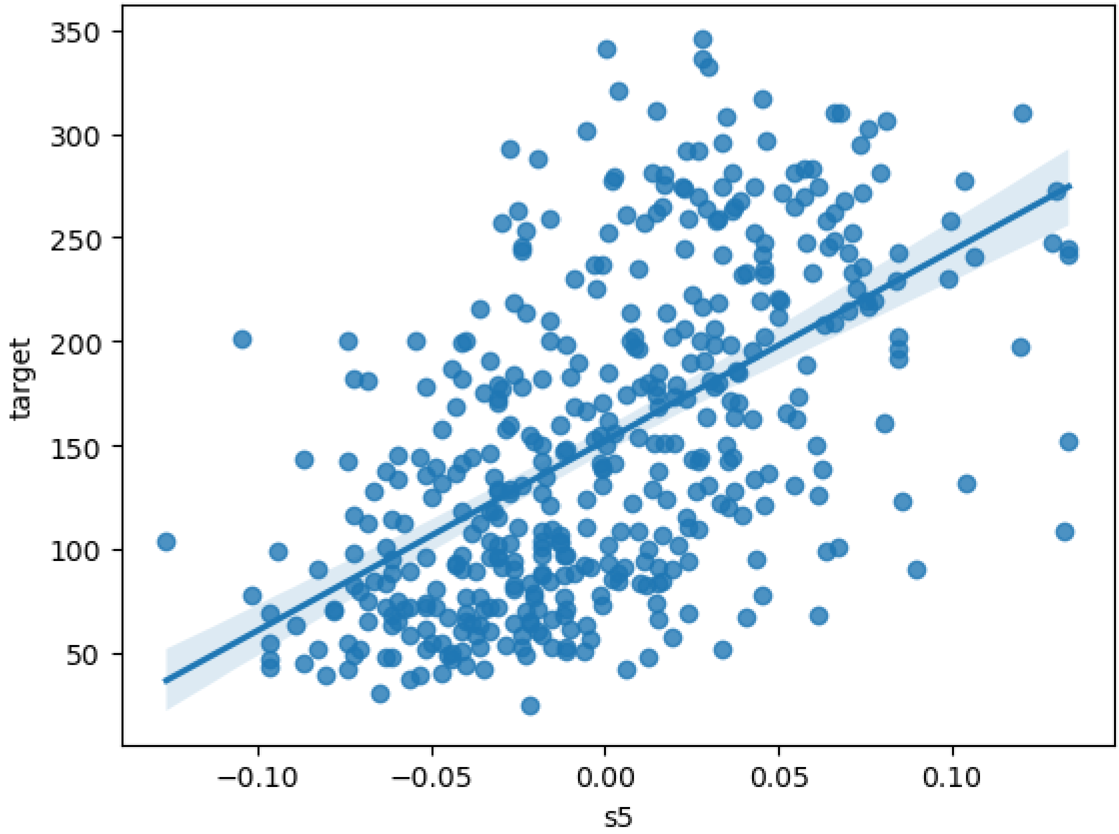
```
plt.scatter(X, y, label='Data', alpha=0.6)

plt.plot(X, reg.predict(X), color='red', linewidth=2, label='Linear Regression')
plt.show()
```



It turns out there is a bit of a cheap trick for plotting linear regression using seaborn. This command will actually both run the linear regression (that is, find the required β_i's) and plot it for you. The tradeoff is that this will only work for single variable linear regression; we'll have to work harder when we're doing multi-variable linear regression. They also do not provide any easy way to get the equation of the line out, so this isn't really the best tool to use for anything other than quick and dirty visualization.

In [11]:
```
# First easy version, but hard to get out the parameters....
sns.regplot(x = diabetes_df.s5,y = diabetes_df.target)
```

Out[11]: <Axes: xlabel='s5', ylabel='target'>



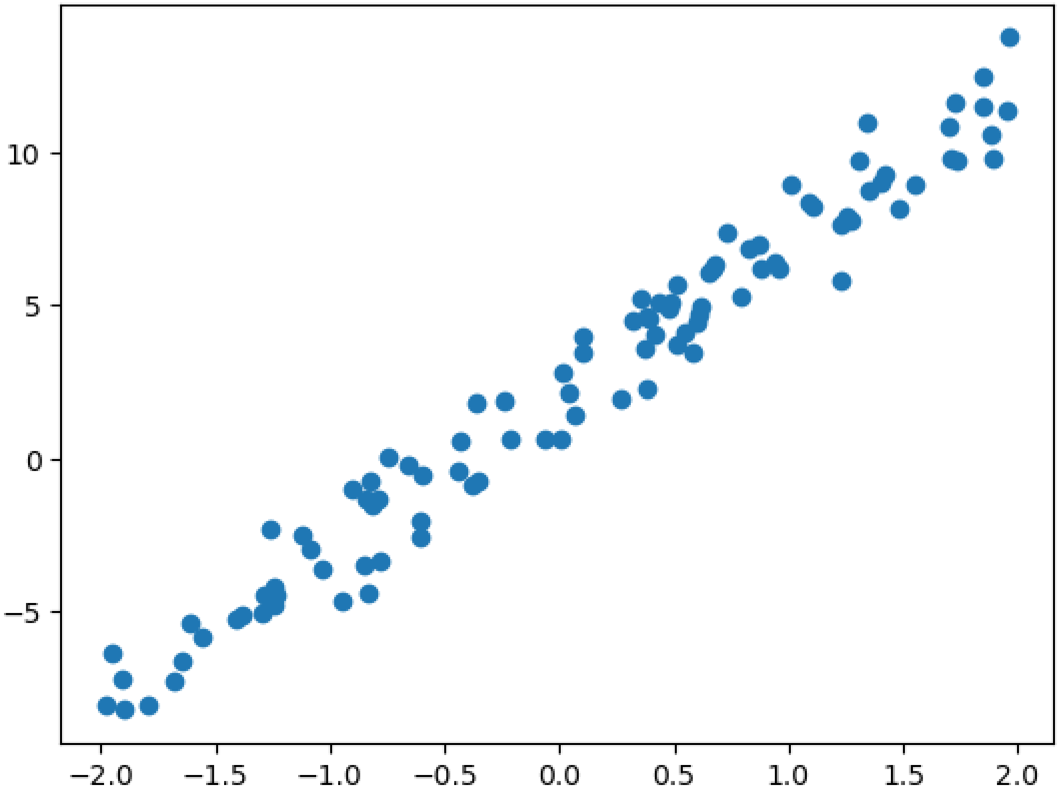
Simulating data

Ok, let's run an example like was shown in class where we see the distribution of possible values.

In [12]:
```
# Here's code that decides on my function
def myFunc(x, b0=2, b1=5):
    return b0 + b1*x


# Here's a command that generates 100 random data points from f(x) + epsilon
def makeData(n = 100):
    X = np.random.uniform(-2,2,n)
    y = myFunc(X) + np.random.normal(size = n)
    return X,y
```

In [13]:
```
# Everytime you run this cell, you get slightly different data

X,y = makeData()

plt.scatter(X,y)
```

Out[13]:
```
<matplotlib.collections.PathCollection at 0x1f76d41f090>
```



In [14]:
```
# Which means that every time you run this cell, you get a slightly different choice
# for the model learned

X,y = makeData()
X = X.reshape([len(X),1])
y = y.reshape([len(y),1])
reg = LinearRegression()
reg.fit(X,y)
print( 'y=' + str(round(reg.coef_[0,0],4)) +  "x_1 + " +  str(round(reg.intercept_[0
```

```
y=5.113x_1 + 1.8863
```

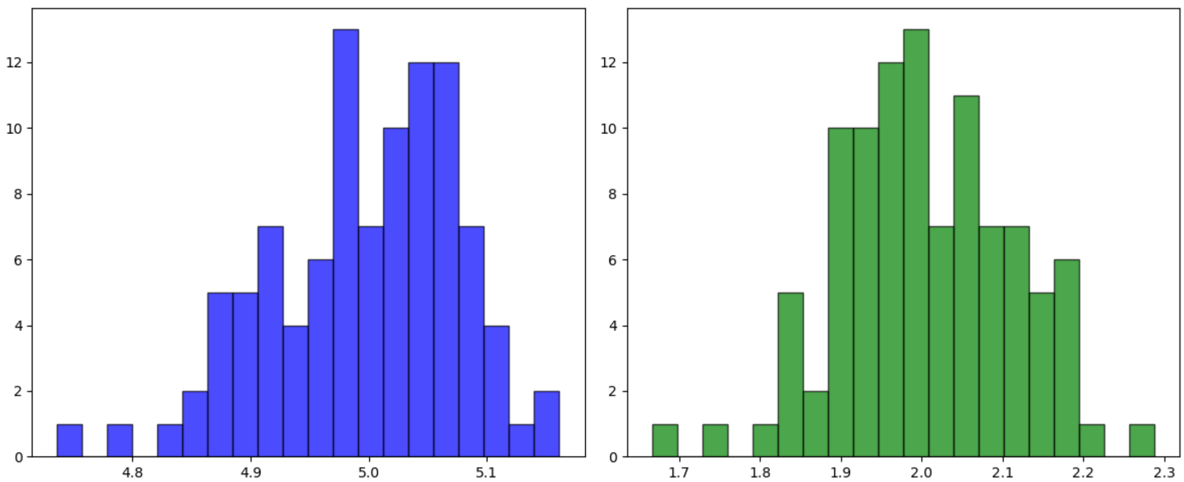
In [15]:
```
# So now, lets just train our linear model lots of times, and collect the resulting c

beta0_list = []
beta1_list = []
for i in range(100):
    X,y = makeData()
    X = X.reshape([len(X),1])
    y = y.reshape([len(y),1])
    reg = LinearRegression()
    reg.fit(X,y)
    beta1_list.append(reg.coef_[0,0])
    beta0_list.append(reg.intercept_[0])

print(beta1_list)
```

```
[4.841211861545089, 5.097051313582989, 5.057848247890258, 4.9829485798147655, 5.0665
11002092634, 5.161348879704815, 4.874671462945548, 4.96165871180912, 5.032295702002
758, 5.096085137883466, 5.025417841093661, 5.055555497445987, 4.984178103256O695, 5.
0390829576577705, 4.9903501376677, 4.990412279261721, 4.95262064929436, 5.0384654320
11232, 5.080796951438875, 4.990669010698016, 5.064378461061874, 5.003456965289418,
4.970470703853867, 5.071220616156822, 5.081750116883228, 4.949022721718189, 5.095692
7843366335, 4.900863577161655, 5.05694403998846, 4.898268643519275, 5.04883317710854
9, 5.063841923060865, 4.959679905516236, 5.012590935931785, 5.0135063597102505, 4.92
965895166306, 5.0319953721521875, 4.955915003090367, 4.9515234569326365, 4.932126857
306128, 5.033710035604536, 4.973169903233403, 4.798777241711958, 5.050724744253253 5,
5.073061477580604, 4.982346336362458, 4.983428902437007, 4.970795990953759, 4.992747
132790065, 4.923030304000648, 4.970545157717457, 5.023626746179707, 4.91389614739302
4, 5.001010307992415 5, 4.9372222362451 6, 4.884929706080533, 5.050962159047375, 5.042
463859393368, 5.037771277824578, 4.736347936600245, 5.030881008524958 5, 5.0378482602
7902, 5.161249244550261, 4.857771326954408, 4.925237590220081, 5.070232151642948, 4.
940728479361404, 5.035150536274977, 5.1209179224393, 4.973895016755378, 4.9909608555
09097, 4.884099697387743, 4.843200397757234, 4.914090807096782, 4.992359394857567,
4.919393652442564, 5.0778831887917875, 5.103531210623521, 4.885515320302331, 5.02525
6564663346, 4.923710224279448, 5.099587846519574, 5.027631425009961, 4.8950605820885
755, 5.00763363305008, 4.894618281507958, 5.071604391629822, 4.9141569240796965, 5.1
185033239786035, 5.041278734237349, 5.089042396669851, 5.064017704520412, 4.87926987
4851025, 5.036449568701099, 5.058317487317978, 5.014737043877317, 4.881433794272026,
5.046779493142074, 4.9957932142053725, 5.114599134072793]
```

Q: Make a histogram of `beta1_list` and separately, `beta0_list`. What do you notice about the distributions?

In [26]:

```python
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.hist(beta1_list, bins=20, edgecolor='black', color='blue', alpha=0.7)


plt.subplot(1, 2, 2)
plt.hist(beta0_list, bins=20, edgecolor='black', color='green', alpha=0.7)


plt.tight_layout()
plt.show()
```



Variance in estimation

To get the statistical test information, we will use the `statsmodels` package. You can take a look at the documentation here: www.statsmodels.org

In [17]:
```
import statsmodels.formula.api as smf
```

In [18]:
```
# Notice that the code is intentionally written to look
# more like R than like python, but it still works!
# Double check..... the coefficients here should be
# about the same as those found by scikit-learn
est = smf.ols('target ~ s5', diabetes_df).fit()
est.summary().tables[1]
```

Out[18]:

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	152.1335	3.027	50.263	0.000	146.185	158.082
s5	916.1374	63.634	14.397	0.000	791.072	1041.202

✅ **Q:** What is $SE(\hat{\beta}_0)$ and $SE(\hat{\beta}_1)$?

In [24]:
```
916.1374-152.1335
```

Out[24]:
```
764.0038999999999
```

✅ **Q:** If we instead use `s1` to predict the target, are $SE(\hat{\beta}_0)$ and $SE(\hat{\beta}_1)$ higher or lower than what you found for the `s5` prediction? Is this reasonable? Try plotting your predictions against scatter plots of the data to compare.

In [19]:
```
# Your code here.
```

Congratulations, we're done!