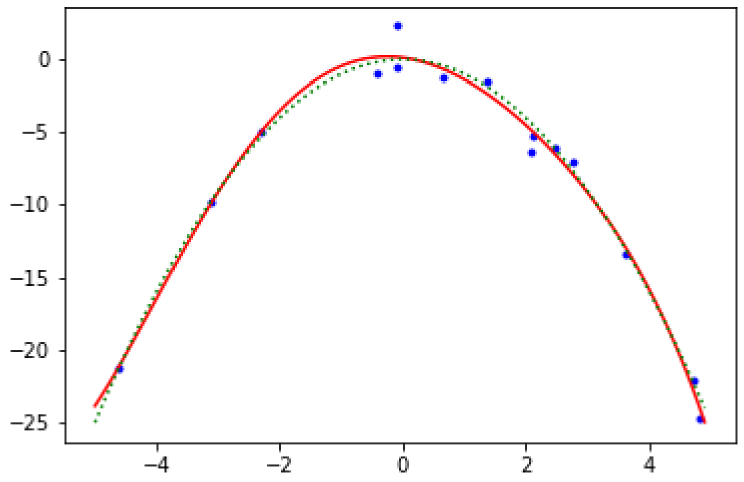
CSCI 303

Introduction to Data Science

6 - Machine Learning Beginnings



Looking Ahead

Tentative Plan

- We've learned just some basics of Python
  - We left out a lot!
    - Functions, modules, classes
    - NumPy, pandas
  - We will learn *some* of these in context of rest of class
- Next task is to start on some machine learning
  - Provide context for learning additional technologies
  - Introduce basic concepts we'll build on later

This Lecture

- Introduce basic supervised learning concepts
- Introduce NumPy & more matplotlib
- Introduce linear regression
- Explore linear regression using NumPy and matplotlib

Machine (or Statistical) Learning

- Learning structure and relationships in data
- Supervised learning
  - Goal is predicting outputs based on inputs
  - Learn from labeled exemplar data
  - E.g., regression, classification
- Unsupervised learning
  - Goal is reveal hidden structure in the data
  - Inputs but no labeled outputs
  - E.g., clustering

Regression

Motivation: consider a relatively simple prediction problem*:

- Inputs: advertising dollars spent (in various media)
- Outputs: total sales
- Problem: predict sales for new ad buy

*This example comes from D. James, et al., *An Introduction to Statistical Learning*

Terms

The *inputs* in this case are numerical values:

- Advertising dollars (for some given week)
  - TV
  - Radio
  - Newspaper

Inputs are often denoted as X, and go by various names:

- predictors
- independent variables
- features

The *outputs* are also numerical values in a regression problem:

- Total sales (for some given week)

The outputs are typically labeled Y, and are called:

- response variables
- dependent variables
- targets

The Model

In regression, we typically assume the existence of some hidden function f, such that

$$Y = f(X) + \epsilon$$

where ϵ represents a random error or *noise* term with zero mean.

Prediction

Given some data (e.g., historical sales data) that includes both the inputs and the outputs, can we make an informed guess of the output for some (previously unseen) input?

We write

$$\hat{Y} = \hat{f}(X)$$

where the "hat" on $\hat{Y}$ and $\hat{f}$ means these are approximate.

Essentially, since ϵ has zero mean, the best approximation we can obtain for some unobserved output y' associated with inputs x' is $f(x')$. However, since f itself is unavailable, we'll use an *estimate* of f.

Parametric Approximation

- There are different kinds of approximation techniques for estimating f.
- We'll study these in more depth later
- For now, focus on *parametric* approximation, in which:
  - $\hat{f}(X) = f(X, \theta)$
  - θ is a vector of *parameters* of some model of f
- More specifically, we'll look at *linear regression*:
  - $\hat{f}(X) = f(X, \theta) = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \ldots + \theta_k X_k$

Notation for Linear Regression

Let's rewrite a bit: given an input (which we assume to be a vector of inputs: $\mathbf{x}$)

- let ϕ represent the vector $(1, x_1, x_2, \ldots, x_k)$
- instead of θ for our parameter vector, we'll use $\mathbf{w}$
- Then

$$\begin{aligned}\hat{f}(\mathbf{x}) &= 1w_0 + x_1 w_1 + \ldots + x_k w_k \\ &= \phi \cdot \mathbf{w}\end{aligned}$$

Some Linear Algebra

Now suppose we have n examples.

We stack all of our ϕ vectors as rows in the matrix Φ:

$$\Phi = \begin{bmatrix} \phi_{00} & \phi_{01} & \cdots & \phi_{0k} \\ \phi_{10} & \phi_{11} & \cdots & \phi_{1k} \\ \vdots & \ddots & & \vdots \\ \phi_{n0} & \phi_{n1} & \cdots & \phi_{nk} \end{bmatrix}$$

Φ is often called the *design matrix*.

We similarly take all of our n outputs y_i to make a vector $\mathbf{y}$:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$\mathbf{y}$ is the *target vector*.

Then we are interested in finding vector $\mathbf{w}$ which as close as possible satisfies the equation

$$\Phi\mathbf{w} = \mathbf{y}$$

For the "ordinary least squares" (**OLS**) solution, we find

$$\mathbf{w} = \arg\min_{\mathbf{w}'} \|\mathbf{y} - \Phi\mathbf{w}'\|$$

which, it turns out, can be solved for using linear algebra to obtain

$$\mathbf{w} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{y}$$

$\mathbf{w}$ is the *weight vector*.

Linear Algebra in Python: NumPy

- Like MATLAB, NumPy provides wrappers onto specialized linear algebra libraries written in C/Fortran
- NumPy adds to Python a basic type: `ndarray`
  - Flexible, multidimensional array type
  - Supports scalar and vector/matrix math operations
  - Easily converts to/from Python sequence types
    - However, `ndarray` imposes container typing
- See the docs (Help menu in Jupyter notebook) for more info

OLS Example Using NumPy and matplotlib

For the rest of this lecture, we're going to:

- Generate (simulate) a regression problem (NumPy)
- Perform OLS regression on the problem (NumPy)
- Plot results (matplotlib)

Step 1: Generate a Problem

First, we need a function to be our "unknown" function to learn.

We're going to stick with polynomials in one variable for this example.

How about:

$$f(x) = 3 + 0.5n - n^2 + 0.15n^3$$

Let's visualize this function.

First, we need some points along the curve.

To do that, we need the NumPy library *imported* into Python:

```
In [19]: import numpy as np     # use np as an alias by convention
```

Now we can use NumPy's `ndarray` type and generate some inputs:

```
In [21]: X = np.arange(-5, 5, 0.1)
         X.shape

Out[21]: (100,)
```

Next, compute f(x) for each input:

In [22]:
```
Y = 3 + 0.5 * X - X**2 + 0.15 * X**3
Y
```

Out[22]:
```
array([-4.325000e+01, -4.110735e+01, -3.902880e+01, -3.701345e+01,
       -3.506040e+01, -3.316875e+01, -3.133760e+01, -2.956605e+01,
       -2.785320e+01, -2.619815e+01, -2.460000e+01, -2.305785e+01,
       -2.157080e+01, -2.013795e+01, -1.875840e+01, -1.743125e+01,
       -1.615560e+01, -1.493055e+01, -1.375520e+01, -1.262865e+01,
       -1.155000e+01, -1.051835e+01, -9.532800e+00, -8.592450e+00,
       -7.696400e+00, -6.843750e+00, -6.033600e+00, -5.265050e+00,
       -4.537200e+00, -3.849150e+00, -3.200000e+00, -2.588850e+00,
       -2.014800e+00, -1.476950e+00, -9.744000e-01, -5.062500e-01,
       -7.160000e-02,  3.304500e-01,  7.008000e-01,  1.040350e+00,
        1.350000e+00,  1.630650e+00,  1.883200e+00,  2.108550e+00,
        2.307600e+00,  2.481250e+00,  2.630400e+00,  2.755950e+00,
        2.858800e+00,  2.939850e+00,  3.000000e+00,  3.040150e+00,
        3.061200e+00,  3.064050e+00,  3.049600e+00,  3.018750e+00,
        2.972400e+00,  2.911450e+00,  2.836800e+00,  2.749350e+00,
        2.650000e+00,  2.539650e+00,  2.419200e+00,  2.289550e+00,
        2.151600e+00,  2.006250e+00,  1.854400e+00,  1.696950e+00,
        1.534800e+00,  1.368850e+00,  1.200000e+00,  1.029150e+00,
        8.572000e-01,  6.850500e-01,  5.136000e-01,  3.437500e-01,
        1.764000e-01,  1.245000e-02, -1.472000e-01, -3.016500e-01,
       -4.500000e-01, -5.913500e-01, -7.248000e-01, -8.494500e-01,
       -9.644000e-01, -1.068750e+00, -1.161600e+00, -1.242050e+00,
       -1.309200e+00, -1.362150e+00, -1.400000e+00, -1.421850e+00,
       -1.426800e+00, -1.413950e+00, -1.382400e+00, -1.331250e+00,
       -1.259600e+00, -1.166550e+00, -1.051200e+00, -9.126500e-01])
```

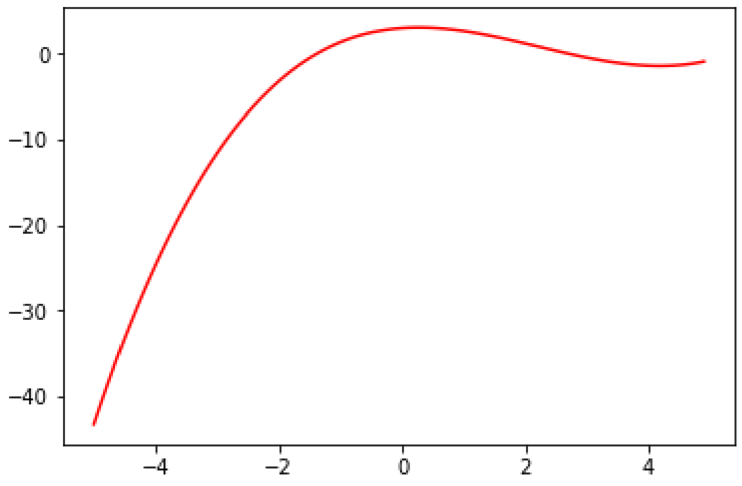
Note that we could simply do the math on the `ndarray` object.

By default, all math operations on ndarrays is element-wise.

Vector operations require other notation (as we'll see).

Let's plot our function:

In [24]:
```python
import matplotlib.pyplot as plt  # another convention
plt.plot(X, Y, "r")
plt.show()
```



Great! Now, our model is

$$Y = f(X) + \epsilon$$

and we assume we only have some data to guide us.

Let's generate some random inputs, then compute Y by adding random noise.

First, we'll make use of `np.random.random(n)` , which generates an array of n points in the interval [0, 1).

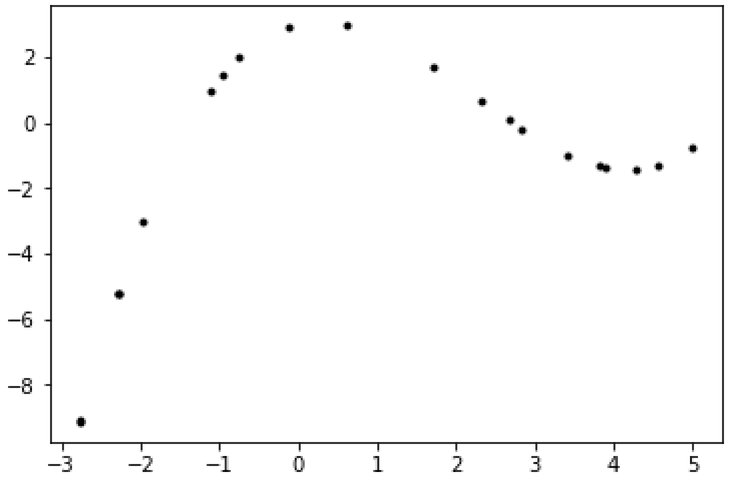
We have to scale and translate the points into the input range of interest - [-5, 5)

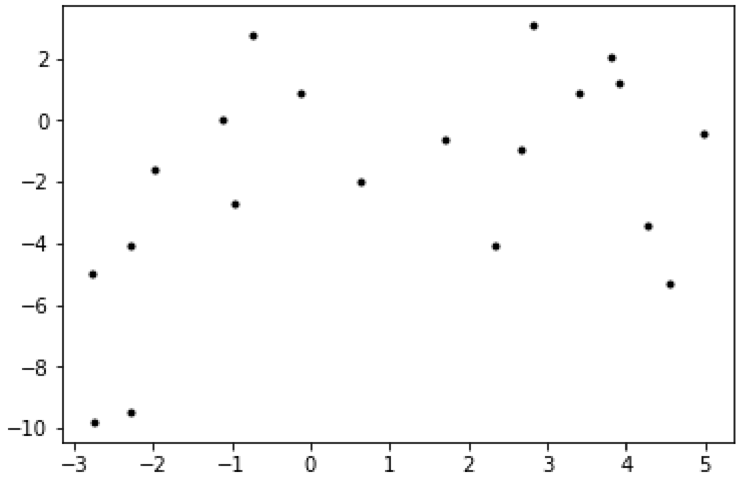
In [25]:
```python
trainX = np.random.random(20) * 10 - 5
trainX
```

Out[25]:
```
array([-2.75793715, -2.29140831,  4.56027452,  2.33229543,  2.83075862,
        4.97854684,  3.82665367,  3.41967435, -2.75018934,  4.27185256,
        2.66496086, -0.95786194, -1.96849765, -0.11363065, -0.74608437,
        1.71517923, -1.11927485,  3.90438685,  0.62703039, -2.29092591])
```

Next, compute $f(x)$ plus some zero mean Gaussian noise on each sample input:

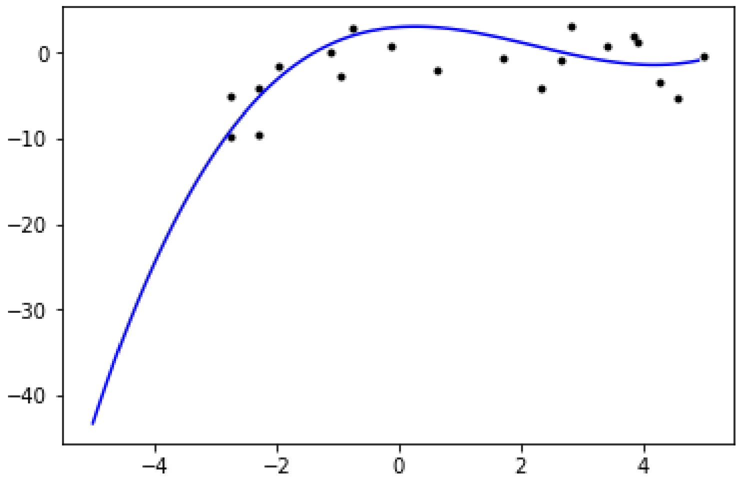
In [26]:
```
trainY = 3 + 0.5 * trainX - trainX**2 + 0.15 * trainX**3  # f(x)
plt.plot(trainX, trainY, 'k.')
plt.show()

trainY = trainY + (np.random.randn(trainY.size) * 3)     # noise

plt.plot(trainX, trainY, 'k.')
plt.show()
```





Adding the true function in, we can see better the noise term in action:

```
In [27]: plt.plot(trainX, trainY, 'k.')
         plt.plot(X, Y, 'b-')
         plt.show()
         trainX.shape
```



```
Out[27]: (20,)
```

Yay! Step 1 complete!

Step 2: Linear Regression

For this, we'll need some *features* to make Φ

- X data of the type we'd have as inputs
- We'll cheat and use polynomials of X

```
In [29]: Phi = np.array([trainX ** p for p in range(6)]).T
         Phi.shape
```

```
Out[29]: (20, 6)
```

Let's examine this a bit further.

To create a matrix in NumPy, use `np.array` on a sequence of sequences (rows):

E.g., the array

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

is created with

```
In [30]: A = np.array([[1,2,3],
                       [4,5,6]])
         A
```

```
Out[30]: array([[1, 2, 3],
                [4, 5, 6]])
```

We did:

```
In [31]: Phi = np.array([trainX ** p for p in range(6)]).T
```

Note the list comprehension applying exponentials to *ndarray* objects!

This generates rows of data which are powers of the training X inputs.

Finally, the `.T` at the end transposes the matrix so that these rows become columns as desired.

OK, we're ready to perform regression using OLS:

Remember, we want to obtain

$$\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$$

```
In [32]: w = np.linalg.inv(Phi.T @ Phi) @ Phi.T @ trainY
         w
```

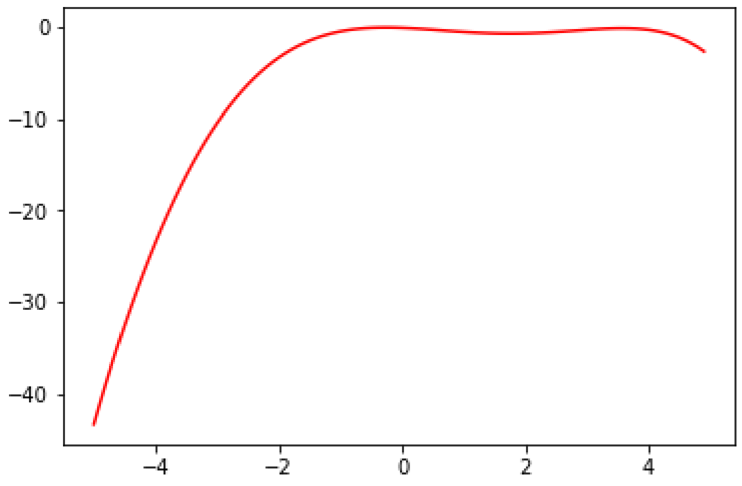
```
Out[32]: array([-0.07775026, -0.26769496, -0.39566406,  0.24369979, -0.02127521,
                -0.00289195])
```

Not bad! Remember, our true coefficients are 3, 0.5, -1, 0.15 and zeroes for the rest.

Step 3: Plot Results

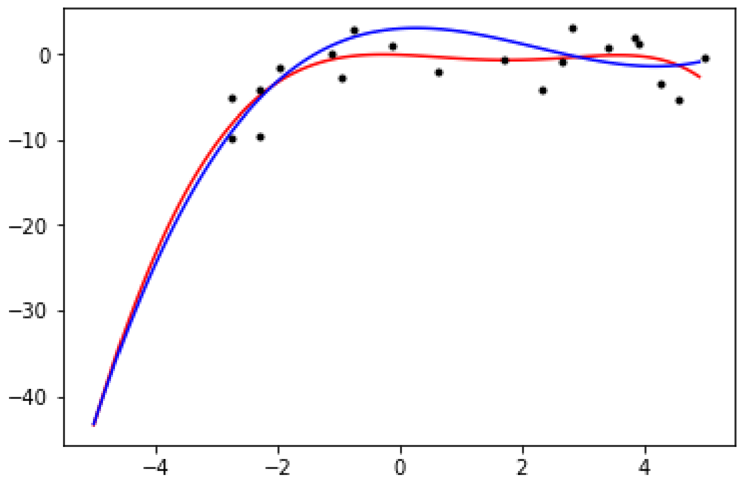
Let's use our learned weights to estimate (predict) Y across our range:

In [33]:
```python
Yhat = np.array([X ** p for p in range(6)]).T @ w
plt.plot(X, Yhat, 'r-')
plt.show()
```



Looks promising! Let's see it against the sample data and the true function:

In [34]:
```python
plt.plot(X, Yhat, 'r-', X, Y, 'b-', trainX, trainY, 'k.')
plt.show()
```



Next Steps

In the next lecture, we'll:

- We will continue exploring a few more machine learning concepts using our example
- Talk more about NumPy