2.1 Linear regression

By the end of this notebook you should be able to:

- Use matrices to formulate linear regression when there is more than one feature (*multiple* linear regression)
- Understand how to determine the optimal values of weights for multiple linear regression using:
    - the normal equations
    - an optimization algorithm
- Understand a basic python implementation of multiple linear regression (using optimization)

Recap of simple linear regression

Last time we looked at *linear* regression where there is only *one* single feature x:

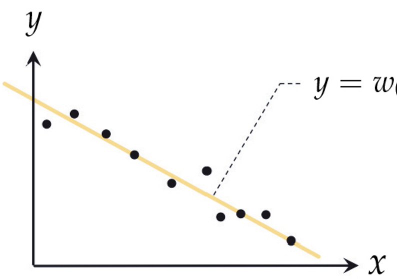
$$f(x) = w_0 + w_1 x$$

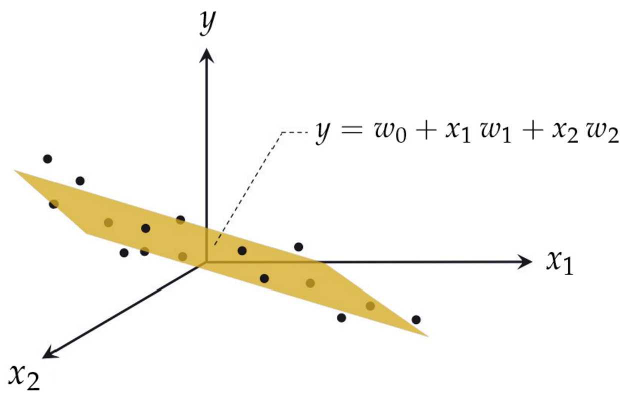
- This is *simple* linear regression.
- The model has only two parameters that we need fit: w_0 and w_1.
- If we use the mean squared error $\frac{1}{N}\sum_{i=1}^{N}(y_i - (w_0 + w_1 x_i))^2$ to fit the line, then we can differentiate this to determine the optimal values of w_0 and w_1 via the normal equations

Beyond simple linear regression

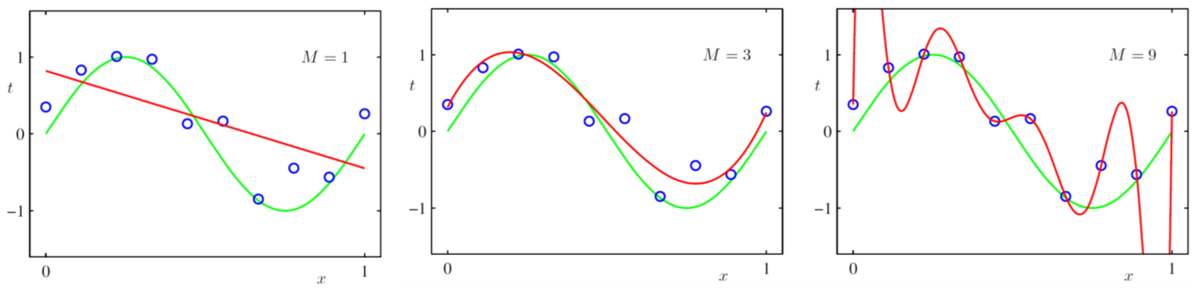
Simple linear regression is a useful starting point, but it is limiting in at least two ways:

1. We may have multiple features $x_1, x_2, x_3, \ldots, x_d$ for our problem, and we may want to include several (or all) of them in our model to predict the label y. This leads us to **multiple linear regression**.





2. Restricting ourselves to a linear function of the features may not be appropriate. For example:
    - In the case where we only one single feature x, the model $f(x) = w_0 + w_1 x$ (i.e. a straight line) will give a poor fit if the relationship between x and y is not linear
    - More generally, when we have multiple features, we don't want to be restricted to linear relationships between y and each feature x_i



From *Pattern Recognition and Machine Learning* by Chris Bishop, (2006)

We will cover multiple linear regression first: extending our model to cover multiple features.

In doing so, we will cover the mathematics necessary to consider non-linear responses from a linear model.

Multiple linear regression

- Suppose each example in our data-set has d features: $x_1, x_2, \ldots, x_d$
- We can represent the features of each example in our data-set as a vector. The features of the i-th example would be represented as

$$\mathbf{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{id} \end{bmatrix}$$

  i.e. we use x_{ij} to denote the j-th feature of example i
  - Don't confuse this with notation a_{ij} used last week for the element in the i-th row and j-th column of a matrix
- Suppose we have a data-set of N examples $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}$

We can construct a linear function of the d features as follows:

$$\begin{aligned} f(\mathbf{x}) &= w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_d x_d \\ &= w_0 + \sum_{i=1}^{d} w_i x_i \end{aligned}$$

This can be written compactly as $f(x) = w_0 + \mathbf{w}^T \mathbf{x}$ where:

the vector $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$ represents the features, and the vector $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$ represents the *feature touching* weights. Weight w_0 does not touch a feature and is referred to as the *bias*.

We can make things even neater by including the bias inside the weight vector however.

Suppose instead that the weight vector includes the bias term w_0, i.e. $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$

And suppose we also augment the feature vector with a 1, i.e. $\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$

Then $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_d x_d$

Representing the least squares cost function in terms of matrices

Recall that the least squares cost function is the mean of the sum of squared residuals in our model:

$$g(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(\mathbf{x}_i))^2$$

With our matrix formulation, $f(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i$ (assuming now the $\mathbf{x}_i$ is augmented with a 1), so this cost function can be written as:

$$g(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

This can then be simplified further as

$$g(\mathbf{w}) = \frac{1}{N}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})$$

where:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} & \ldots & x_{1d} \\ 1 & x_{21} & x_{22} & x_{23} & \ldots & x_{2d} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & x_{N2} & x_{N3} & \ldots & x_{Nd} \end{bmatrix}$$

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

Similar to before, we can differentiate $g(\mathbf{w}) = \frac{1}{N}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})$ with respect to $\mathbf{w}$, and derive normal equations for the case of multiple linear regression. Doing so gives us:

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

It is part of a coursework question to implement this.

Limitations of Normal equations

- Using the normal equations involves computing $(\mathbf{X}^T\mathbf{X})^{-1}$
- This is a $d \times d$ matrix, where d is the number of features.
- The computational complexity of inverting a $d \times d$ matrix is roughly d^3, although algorithms exist that take this down to about $d^{2.4}$
- Therefore, if number of feaures d increases by a factor of 10, then the computational complexity increases by a factor of $10^{2.4} \approx 250$
- For large enough d, the cost becomes prohibitive, and using the normal equations isn't feasible.

Optimization

- Using normal equations is not normal: most machine learning algorithms don't have closed-form solutions for their optimal parameter values.
- Optimization algorithms are almost always used to determine the best model parameters for a machine learning algorithm. We will look at optimization algorithms in more depth next week.
- For this week, we will just use a random search optimization algorithm as a black box, and look in more detail at how it works next week.

Implementing the Least Squares model as an optimization problem in Python

Step 1

- Read in our data into matrices $\mathbf{X}$ and $\mathbf{y}$
- Here we will just create some dummy data to test our code out as we go along

Better:

- Read in real data from a csv file into `X` and `y`

In [2]:
```
import numpy as np

X = np.array([[0.2,0.3],[0.4,0.2],[0.8,0.5],[0.6,0.3]])
y = np.array([[1],[2],[2],[4]])

print(X.shape)
```

```
(4, 2)
```

Step 2

Now we need to add an extra column of 1s to `X`

In [4]:
```
X_with_ones = np.hstack((np.ones((4,1)),X))
print(X_with_ones)
```

```
[[1.  0.2 0.3]
 [1.  0.4 0.2]
 [1.  0.8 0.5]
 [1.  0.6 0.3]]
```

Step 3

- Create a function called `model()` that:
  - takes as input a vector `x` and a weight vector `w`
  - returns as output the linear combination $f(\mathbf{x}) = w_0 + w_1x_1 + w_2x_2 + \ldots + w_dx_d$
- We need to be very careful as to whether `x` contains the added 1 or not:
  - If it does: $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = w_0 + w_1x_1 + w_2x_2 + \ldots + w_dx_d$
  - If it does not: $f(\mathbf{x}) = w_0 + \mathbf{w}_1^T\mathbf{x} = w_0 + w_1x_1 + w_2x_2 + \ldots + w_dx_d$

In [5]:
```
def model(x,w):
    val = w[0,0] + np.matmul(w[1:,].T,x)[0,0]
    return val
```

Step 4

Lets test our `model` function out with a dummy set of weights `w` , and a dummy input vector `x_new`

In [12]:
```
w = np.array([[1,2,3]]).T

x_new = np.array([[0.3,0.5]]).T

pred = model(x_new,w)
print(pred)
```

```
3.1
```

Step 5

Write our cost function `least_squares` that calculates the mean squared error for our model using

$$g(\mathbf{w}) = \frac{1}{N}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})$$

- The input to `least_squares` should be weights `w`
  - It will also use matrix `X` and vector `y` (not passed in)
- The function should return the value of `g(w)` as given by the formula above

In [13]:
```
def least_squares(w):
    Xw = np.matmul(X_with_ones,w)
    y_minus_Xw = y - Xw
    cost = np.matmul(y_minus_Xw.T,y_minus_Xw)/len(y)
    return cost[0,0]
```

Step 6

Test out the `least_squares` function by calling it with our data

In [15]:
```
mse = least_squares(w)
print(mse)
```

```
1.7674999999999994
```

Step 7

- In the cell below we will use the *random search* algorithm for optimization.
- Optimization will be covered next week - for this week we treat this function as a 'black box' that:
  - takes a cost function as an input `g`
  - takes an initial weight vector `w`
  - other input parameters will be explained next week

- and returns the history of the search for optimal parameters values (using the cost function we specified) as output
  - last value will be the best set of parameter values found

In [17]:

```
def random_search(g,alpha_choice,max_its,w,num_samples):
    # run random search
    weight_history = []         # container for weight history
    cost_history = []           # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice

        # record weights and cost evaluation
        weight_history.append(w.T)
        cost_history.append(g(w.T))

        # construct set of random unit directions
        directions = np.random.randn(num_samples,np.size(w))
        norms = np.sqrt(np.sum(directions*directions,axis = 1))[:,np.newaxis]
        directions = directions/norms

        ### pick best descent direction
        # compute all new candidate points
        w_candidates = w + alpha*directions

        # evaluate all candidates
        evals = np.array([g(w_val.T) for w_val in w_candidates])

        # if we find a real descent direction take the step in its direction
        ind = np.argmin(evals)
        if g(w_candidates[ind]) < g(w.T):
            # pluck out best descent direction
            d = directions[ind,:]

            # take step
            w = w + alpha*d

    # record weights and cost evaluation
    weight_history.append(w.T)
    cost_history.append(g(w.T))
    return weight_history,cost_history
```

In [18]:

```
wh, sh= random_search(least_squares,alpha_choice='diminishing',max_its=100,w=np.arra
```

In [2]:

```python
# Video below from https://github.com/jermwatt/machine_learning_refined
# Shared under the Creative Commons Attribution 4.0 International License (CC BY-NC-

from IPython.display import HTML
HTML("""
<video width="1000" height="400" controls loop>
  <source src="Videos/animation_2.mp4" type="video/mp4">
  </video>
""")
```

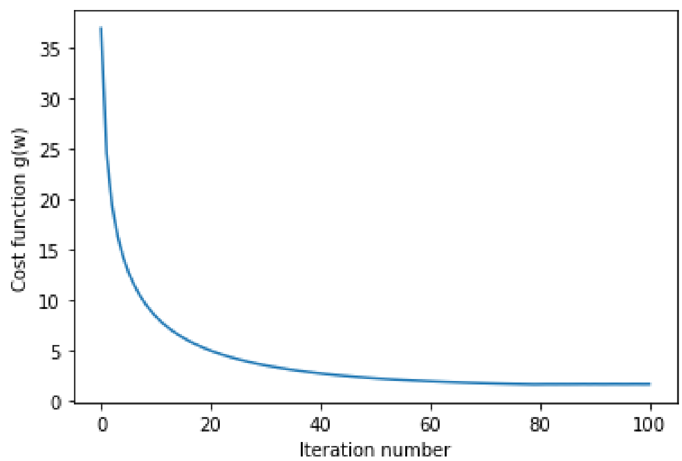
Out[2]:

0:00 / 0:05

In [19]:

```python
import matplotlib.pyplot as plt

xiter = np.arange(0,101)
plt.plot(xiter[0:],sh[0:])
plt.xlabel('Iteration number')
plt.ylabel('Cost function g(w)')
plt.show()
```



In [20]:

```
print('best weights found = ',wh[-1])
```

```
best weights found =  [[ 2.63047661]
 [-0.9905079 ]
 [-0.6078989 ]]
```

In [21]:

```
print('best cost function value = ',sh[-1])
```

```
best cost function value =  1.5996054132058242
```

Example of multiple linear regression

1. The Wine data-set that you looked at in labs last week contained multiple features, which could be used to predict the label of *quality*. **This will be one of the lab exercises today**.

2. Using the example from *First Course in Machine Learning*, of winning times in the men's 100m Olympics finals, we could have three features as follows:

   - x_1 = year
   - x_2 = wind speed at time of the race
   - x_3 = best time run so far by any competitor that year

   At the minute, we only have values for x_1 in our data-set for Olympics.

   - We could go search the internet for the values of x_2 and x_3 for each year
   - Or we could construct new features from the feature we currently have, i.e. x_1 (polynomial regression - next part of the lecture)