COMP755-Lect03

September 3, 2018

1 COMP 755

Plan for today

1. Review likelihood, log-likelihood, maximum likelihood estimates
2. Compute Maximum Likelihood Estimate for a Gaussian model
3. Introduce Linear Regression
4. Introduce Gradient Ascent/Descent methods

2 Last time -- distributions

Last time we reviewed probability distributions of one dimensional random variables: * Bernoulli (coin toss), Binomial (count of heads in multiple coin tosses) * Categorical (die roll), Multinomial (count of different die sides in multiple rolls)

Continuous distributions: * Gaussian * Laplace

3 Last time -- likelihood

We introduced **likelihood** function.

$$\mathcal{L}(\theta|\mathbf{x}) = p(\underbrace{\mathbf{x}}_{\text{Data}} | \underbrace{\theta}_{\text{parameters}}) = \prod_i p(x_i|\theta)$$

It tells us how well our model fits the data for particular parameters.

An example of model

$$p(x|\theta) = \theta^x(1-\theta)^{1-x}$$

data

$$\mathbf{x} = \{0, 1, 1, 1\}$$

and likelihood

$$\mathcal{L}(\theta|\mathbf{x}) = \prod_i p(x_i|\theta) = p(x_1|\theta)p(x_2|\theta)p(x_3|\theta)p(x_4|\theta) = (1-\theta)\theta^3$$

4 Last time -- log-likelihood and MLE

We introduced log-likelihood function

$$\log \mathcal{L}(\theta \big| \text{Data}).$$

An example: Given data Data $= \{0,0,0,1\}$ and model $p(x|\theta) = \theta^x(1-\theta)^{1-x}$

$$-2.25 = \log \mathcal{L}(0.25 \big| \mathbf{x}) > \log \mathcal{L}(0.99 \big| \text{Data}) = -13.83$$

```
In [1]: import numpy
        from __future__ import print_function
        def likelihood(theta,xs):
            p = 1.0
            for x in xs:
                p = p*theta**x * (1-theta)**(1-x)
            return p

        xs = [0,0,0,1]
        theta1 = 0.99
        theta2 = 0.25
        loglik1 = numpy.log(likelihood(theta1,xs))
        loglik2 = numpy.log(likelihood(theta2,xs))
        print("Log-Likelihood(",theta1,"|",xs, ")=",loglik1)
        print("Log-Likelihood(",theta2,"|",xs, ")=",loglik2)
```

```
Log-Likelihood( 0.99 | [0, 0, 0, 1] )= -13.8255608938
Log-Likelihood( 0.25 | [0, 0, 0, 1] )= -2.24934057848
```

5 Last time -- finding Maximum Likelihood Estimate (MLE)

We talked about finding a mazimizer of the log-likelihood called **Maximum Likelihood Estimate** (MLE)

$$\theta^{\text{MLE}} = \arg\max_{\theta} \log \mathcal{L}(\theta \big| \text{Data})$$

which can be interepreted as the parameter for which the data is most probable under the model.

We recalled that in order to maximize a functions, $f(x)$, we finding x for which

$$\frac{\partial}{\partial x} f(x) = 0.$$

We used this approach to find MLE for a Bernoulli model

$$p(x\big||\theta) = \theta^x(1-\theta)^{1-x}$$

and came up with

$$\theta^{\text{MLE}} = \frac{\overbrace{\sum_i [x_i = 1]}^{\text{count of 1s in data}}}{\underbrace{\sum_i [x_i = 1]}_{\text{count of 1s in data}} + \underbrace{\sum_i [x_i = 0]}_{\text{count of 0s in data}}}$$

6 Finding μ^{MLE} of Gaussian distribution

We left as an exercise a problem to come up with a maximum likelihood estimate for parameter μ of a Gaussian distribution

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

So we will do that now.

Likelihood function is

$$\mathcal{L}(\mu, \sigma^2|\mathbf{x}) = \prod_{i=1}^{N} p(x_i|\mu, \sigma^2) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x_i-\mu)^2}$$

Log-likelihood function is

$$\log \mathcal{L}(\mu, \sigma^2|\mathbf{x}) = \log \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x_i-\mu)^2} = \sum_{i=1}^{N} \left[-\frac{1}{2}\log 2\pi\sigma^2 - \frac{1}{2\sigma^2}(x_i - \mu)^2 \right]$$

7 Finding μ^{MLE} of Gaussian distribution

Our recipe is: 1. Take the function you want to maximize:

$$f(\mu) = \sum_{i=1}^{N} \left[-\frac{1}{2}\log 2\pi\sigma^2 - \frac{1}{2\sigma^2}(x_i - \mu)^2 \right]$$

2. Compute its first derivative: $\frac{\partial}{\partial\mu} f(\mu)$ 3. Equate that derivative to zero and solve: $\frac{\partial}{\partial\mu} f(\mu) = 0$

The first derivative is

$$\frac{\partial}{\partial\mu} f(\mu) = \sum_{i=1}^{N} \left[-\frac{1}{\sigma^2}(x_i - \mu) \right]$$

We equate it to zero and solve

$$\sum_{i=1}^{N} \left[-\frac{1}{\sigma^2}(x_i - \mu) \right] = 0$$

8 Finding μ^{MLE} of Gaussian distribution

$$\begin{aligned}
\sum_{i=1}^{N}\left[-\frac{1}{\sigma^2}(x_i-\mu)\right] &= 0\\
\sum_{i=1}^{N}\frac{1}{\sigma^2}(x_i-\mu) &= 0\\
\sum_{i=1}^{N}(x_i-\mu) &= 0\\
\sum_{i=1}^{N}x_i &= \sum_i \mu\\
\sum_{i=1}^{N}x_i &= \mu\sum_i 1\\
\frac{\sum_{i=1}^{N}x_i}{\sum_{i=1}^{N}1} &= \mu\\
\frac{\sum_{i=1}^{N}x_i}{N} &= \mu
\end{aligned}$$

9 Using MLE estimates in prediction

What can we do with maximum likelihood estimates? We can predict outcome of the next experiment: * Given data $\{1,0,1,1,0,0,0,0,0\}$ and assuming Bernoulli model,θ^{MLE} is $\frac{3}{3+5}=\frac{3}{8}$. Hence, probability that the next experiment will yield $x=1$ is $\frac{3}{8}$

- Given heights of 6th graders in inches $\{56,57,59,63,55,61\}$ and assuming Gaussian model, μ^{MLE} is $\frac{56+57+59+63+55+61}{6}=58.5$ and standard deviation is 3.082 . Hence, probability that the next 6th grader will be of height 58.5 ± 2*3.082 is 0.95.

Given realizations of the same random variable we can estimate what the next draw might look like.

Q: Suppose you wanted to predict a student's grade in COMP 755? What would you do?

10 Linear Regression

One of the simplest examples of supervised learning is **linear regression**.

An underlying assumption in **linear regression** is that the target variable, y, varies around a mean which is a linear combination of some set of features.

For example, we can try to predict a student's grade in COMP755 class using grades from prereq's and number of beers the student has each week

$$\text{COMP755} = 0.25 + 0.2*\text{COMP410} + 0.3*\text{MATH233} + 0.5*\text{STOR435} - 0.1*\text{beers} + \epsilon$$
$$\epsilon \sim \mathcal{N}(0, 0.1)$$

Formally, we would write

$$y = \beta_0 + \sum_j x_j\beta_j + \epsilon$$
$$\epsilon \sim \mathcal{N}\left(0,\sigma^2\right)$$

or more compactly

$$y|\mathbf{x} \sim \mathcal{N}\left(\beta_0 + \sum_j x_j\beta_j, \sigma^2\right)$$

11 Linear Regression

Taking a closer look at the model:

$$y|\mathbf{x} \sim \mathcal{N}\left(\beta_0 + \sum_{j=1}^{p} x_j\beta_j, \sigma^2\right)$$

* y is a target variable we are modeling * $\mathbf{x}$ is a vector of p features (aka predictors and covariates) * β_0 is a **bias** which does not depent on the features * $\beta_1, \ldots, \beta_p$ is a vector of weights (one per feature) * σ^2 is variance -- how far can we expect y to be away from $\beta_0 + \sum_{j=1}^{p} x_j\beta_j$

Q: Which of these items above are parameters? Hint: Which values can we adjust to get a better prediction?

$$\text{COMP755} = 0.25 + 0.2 * \text{COMP410} + 0.3 * \text{MATH233} + 0.5 * \text{STOR435} - 0.1 * \text{beers} + \epsilon$$

12 Linear Regression

Probability of target variable y

$$p(y|\mathbf{x}, \beta_0, \beta, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}\left(y_i - \underbrace{(\beta_0 + \sum_j x_j\beta_j)}_{\text{mean of the Gaussian}}\right)^2\right\}$$

In the case of the 6th grader's height, we made **the same** prediction for any other 6th grader (58.5 inches).

In our COMP 755 grade example, we compute a potentially different mean for every student.

$$\beta_0 + \beta_{\text{COMP410}} * \text{COMP410} + \beta_{\text{MATH233}} * \text{MATH233} + \beta_{\text{STOR435}} * \text{STOR435} + \beta_{\text{beers}} * \text{beers}$$

13 Linear Regression -- toy example

We can try to fit this model to some data.

Note that these are **not** real UNC student data.

$x_1 = \text{COMP410}$	$x_2 = \text{MATH233}$	$x_3 = \text{STOR345}$	$x_4 = \text{\#beers}$	$y = \text{COMP755}$
3	3	4	0	4
3	3	3	0	3
3	2	2	5	2
2	2	4	0	3
3	3	4	4	3

x_1 = COMP410	x_2 = MATH233	x_3 = STOR345	x_4 = #beers	y = COMP755
4	4	3	0	4
2	2	4	0	3
2	2	2	0	2
3	4	2	5	3
2	2	3	0	3
2	4	2	0	3
3	3	3	0	3
2	2	4	5	2
4	3	2	6	2
3	4	4	4	3
3	2	2	6	2
3	3	3	0	3
3	2	4	0	3
3	3	4	4	3
3	4	2	0	3

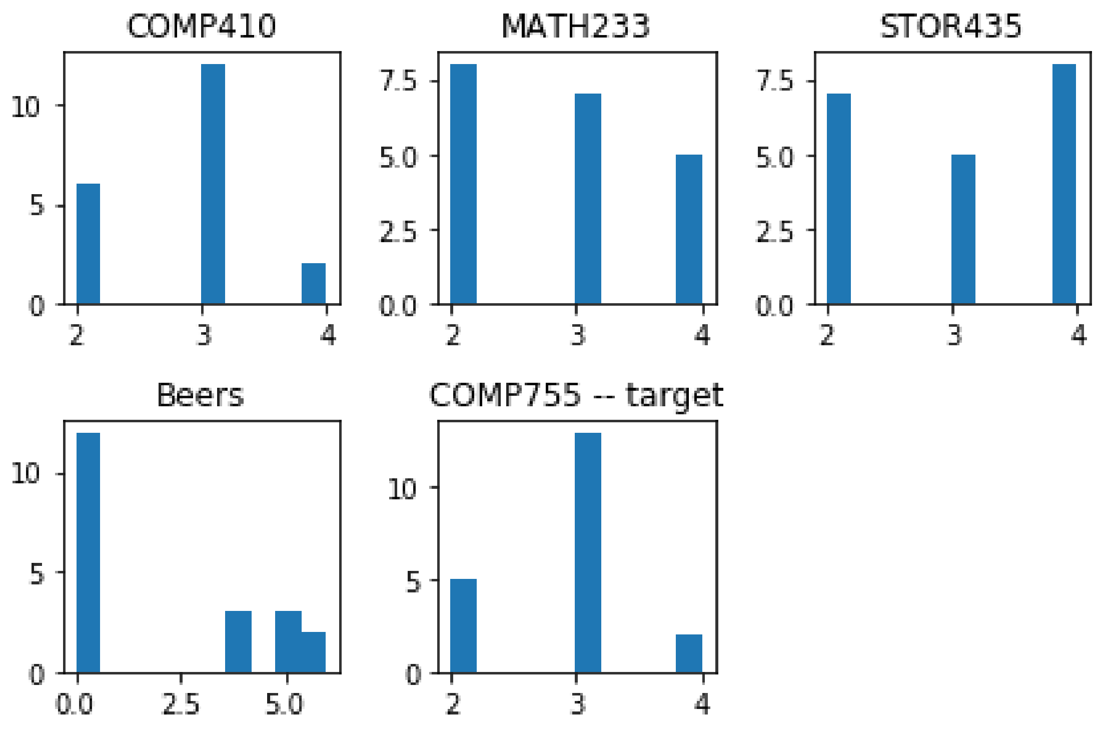
```
In [3]: X = numpy.asarray([[3.0,3.0,4.0,0.0],[3.0,3.0,3.0,0.0],[3.0,2.0,2.0,5.0],
        [2.0,2.0,4.0,0.0],[3.0,3.0,4.0,4.0],[4.0,4.0,3.0,0.0],
        [2.0,2.0,4.0,0.0],[2.0,2.0,2.0,0.0],[3.0,4.0,2.0,5.0],
        [2.0,2.0,3.0,0.0],[2.0,4.0,2.0,0.0],[3.0,3.0,3.0,0.0],
        [2.0,2.0,4.0,5.0],[4.0,3.0,2.0,6.0],[3.0,4.0,4.0,4.0],
        [3.0,2.0,2.0,6.0],[3.0,3.0,3.0,0.0],[3.0,2.0,4.0,0.0],
        [3.0,3.0,4.0,4.0],[3.0,4.0,2.0,0.0]])

        y = numpy.asarray([4.0,3.0,2.0,3.0,3.0,4.0,3.0,2.0,3.0,
        3.0,3.0,3.0,2.0,2.0,3.0,2.0,3.0,3.0,3.0,3.0])
```

```
In [4]: import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy
        features = ['COMP410','MATH233','STOR435','Beers']
        for j in range(X.shape[1]):
            plt.subplot(2,3,j+1)
            plt.hist(X[:,j])
            plt.title(features[j])

        plt.subplot(2,3,5)
        print(y)
        plt.hist(y)
        plt.title('COMP755 -- target')
        plt.tight_layout()  # ensure sensible layout of subplots
```

```
[ 4.  3.  2.  3.  3.  4.  3.  2.  3.  3.  3.  3.  2.  2.  3.  2.  3.  3.
  3.  3.]
```



14 Linear Regression -- toy example

Our goal is to fit the model that predicts COMP 755 grade.

Our prediction is based on

$$\text{COMP562} = \beta_0 + \beta_{\text{COMP410}} * \text{COMP410} + \beta_{\text{MATH233}} * \text{MATH233} + \beta_{\text{STOR435}} * \text{STOR435} + \beta_{\text{beers}} * \text{beers} + \epsilon$$

In order to make a prediction we need * β_0 -- bias or grade you get regardless of your other grades * β_{COMP410} * β_{MATH233} * β_{STOR435} * β_{beers}

Note that having σ^2 would enable us to give a range of grades that cover 95% of the probability.

15 Linear Regression -- likelihood

We start by writing out a likelihood for linear regression is

$$\mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} p(y|\mathbf{x}, \beta_0, \beta, \sigma^2) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{ -\frac{1}{2\sigma^2} \left(y_i - (\beta_0 + \sum_j x_j \beta_j) \right)^2 \right\}$$

Log-likelihood for linear regression is

$$\log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} \left[-\frac{1}{2} \log 2\pi\sigma^2 - \frac{1}{2\sigma^2} \left(y_i - (\beta_0 + \sum_j x_j \beta_j) \right)^2 \right].$$

Our goal is still to find β_0, β such that

$$\begin{aligned}
\frac{\partial}{\partial \beta_0} \log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) &= 0 \\
\frac{\partial}{\partial \beta_1} \log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) &= 0 \\
&\ldots \\
\frac{\partial}{\partial \beta_p} \log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) &= 0
\end{aligned}$$

because that will guarantee the parameters can not be further changed to improve the likelihood.

16 Introducing gradient ascent

Previously, we solved equations of type $\frac{\partial}{\partial \theta} \log \mathcal{L}(\theta | \text{Data}) = 0$ in a closed-form.

Here, we will develop a different approach using numerical optimization.

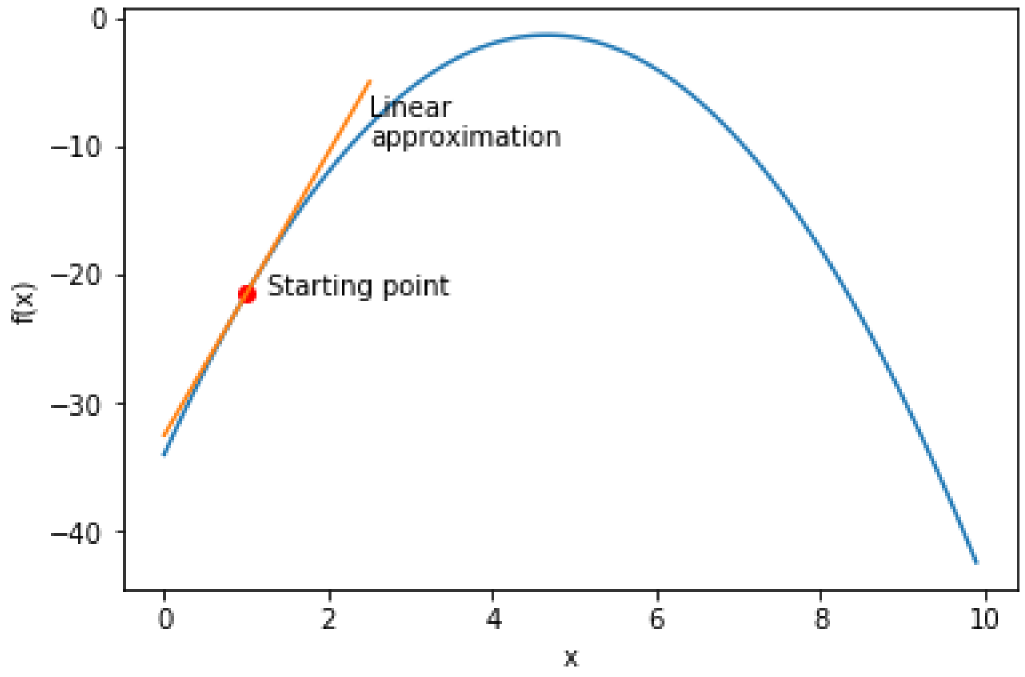
Let's first consider how we can maximize a univariate differentiable function $f(x)$ iteratively.

Q: For the function shown below, starting from the labeled point, should we increase or decrease x to get to the maximum of function $f(x)$? Why?

```
In [5]: x = numpy.arange(0.0,10.0,0.1)
        f = -(x-4.0)**2.0 - 0.5*(x-6.0)**2.0
        dfdx = -(x-4.0)*2 - (x-6.0)
        plt.plot(x,f)
        plt.xlabel('x')
        plt.ylabel('f(x)')
        plt.plot(x[10],f[10],'ro')
        plt.annotate('Starting point',(x[10]+0.25,f[10]))
        x1 = x[10]-1
        x2 = x[10]+1.5
        f1 = f[10]-dfdx[10]*1
        f2 = f[10]+dfdx[10]*1.5
        plt.plot([x1,x2],[f1,f2])
        plt.annotate('Linear\napproximation',(x2,f2-5))
```

```
Out[5]: Text(2.5,-10,'Linear\napproximation')
```

Topographical map of Pilot Mountain



Another way to see this is to consider a very simplified version of Taylor's theorem.

Theorem. Given a function $f(\cdot)$ which is smooth at x

$$f(x+d) = f(x) + f'(x)d + O(d^2)$$

where we say that $g(x) = O(h(x))$ iff $\lim_{x\to a} \left|\frac{f(x)}{g(x)}\right| = C < +\infty$.

In words, close to x function $f(\cdot)$ is very close to being a linear function of d

$$f(x+d) = f(x) + f'(x)d$$

Slope of the best linear approximation is $f'(x)$.

$f'(x)$ tells us in which direction function grows.

Multivariate functions are a bit harder to visualize.

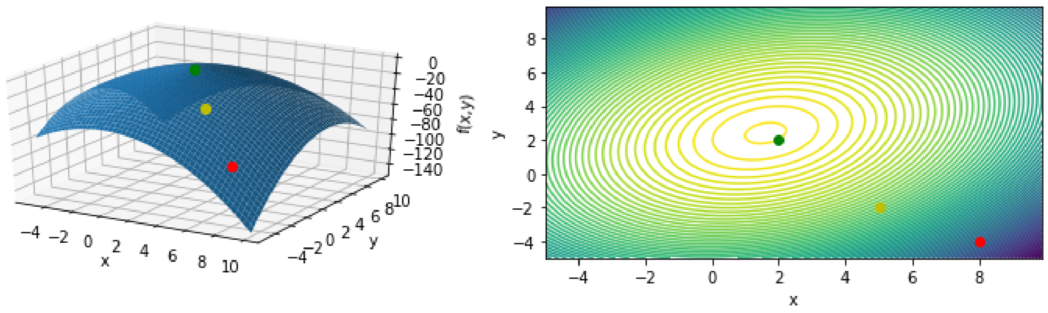
Hence we view them like topographical maps.

Contours connect nearby points with the same altitutde (function value).

```
In [6]: import matplotlib.cm as cm
        x = numpy.arange(-5.0,10.0,0.1)
        y = numpy.arange(-5.0,10.0,0.1)
        X,Y = numpy.meshgrid(x,y)
        F = -(X-1.0)**2.0 - (Y-2.0)**2.0 + 0.5*X*Y
        fig = plt.figure(figsize=plt.figaspect(0.35))
        from mpl_toolkits.mplot3d import Axes3D
```

```
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_surface(X,Y,F)
plt.tight_layout(6.0)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x,y)')
ax.plot([x[130]],[y[10]],[F[130,10]],'ro')
ax.plot([x[70]],[y[70]],[F[70,70]],'go')
ax.plot([x[100]],[y[30]],[F[100,30]],'yo')
ax = fig.add_subplot(1,2,2)
ax.contour(X,Y,F,levels=numpy.arange(numpy.min(F),numpy.max(F),2))
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.plot(x[130],y[10],'ro')
ax.plot(x[70],y[70],'go')
ax.plot(x[100],y[30],'yo')
```

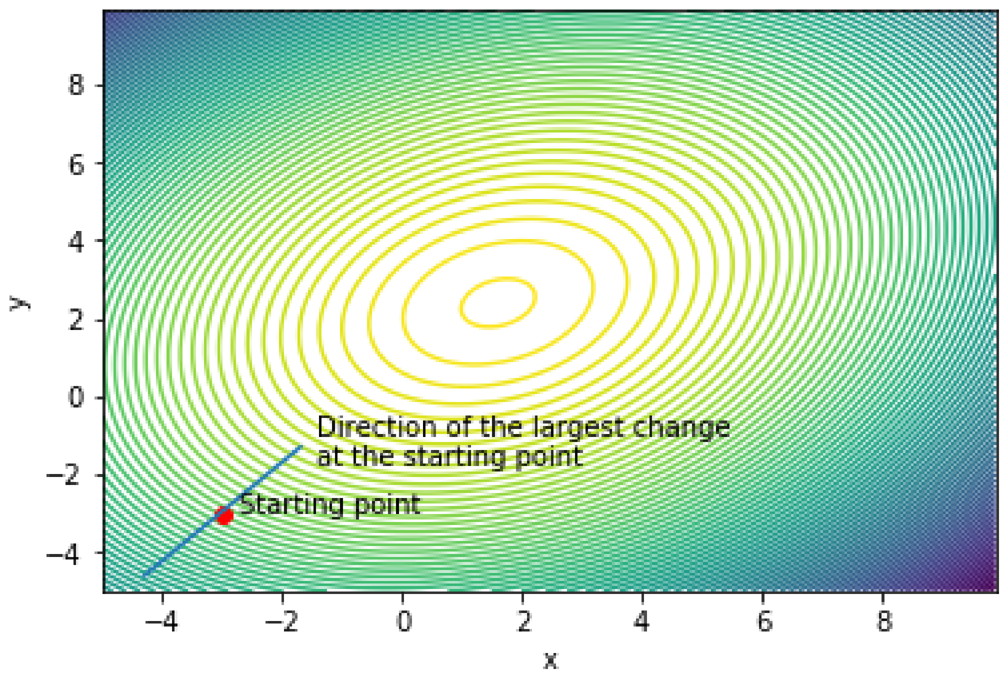
```
Out[6]: [<matplotlib.lines.Line2D at 0x11573e160>]
```



```
In [7]: import matplotlib.cm as cm
        x = numpy.arange(-5.0,10.0,0.1)
        y = numpy.arange(-5.0,10.0,0.1)
        X,Y = numpy.meshgrid(x,y)
        F = -(X-1.0)**2.0 - (Y-2.0)**2.0 + 0.5*X*Y
        dfdx = -2.0*(X-1.0) + 0.5*Y
        dfdy = -2.0*(Y-2.0) + 0.5*X
        plt.contour(X,Y,F,levels=numpy.arange(numpy.min(F),numpy.max(F),2))
        plt.annotate('Starting point',(x[20]+0.25,y[20]))
        plt.plot(x[20],y[20],'ro')
        x1 = x[20] -0.2*dfdx[20,20]
        y1 = y[20] -0.2*dfdy[20,10]
        x2 = x[20] +0.2*dfdx[20,20]
        y2 = y[20] +0.2*dfdy[20,20]
        plt.annotate('Direction of the largest change \nat the starting point',(x2+0.25,y2-0.5))
        plt.xlabel('x')
```

```
    plt.ylabel('y')
    plt.plot([x1,x2],[y1,y2])
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x1157b4748>]
```



Much as in the case of univariate function, the direction in which the function changes is described by derivatives.

A vector composed of partial derivatives of a function is called gradient

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_p} f(\mathbf{x}) \end{bmatrix}$$

For example,

$$f(x,y) = -(x-1)^2 - (y-2)^2 + \frac{1}{2}xy$$

has gradient

$$\nabla f(x,y) = \begin{bmatrix} -2(x-1) + \frac{1}{2}y \\ -2(y-2) + \frac{1}{2}x \end{bmatrix}$$

Gradient points in the direction of the largest change in the function.

Q: Can you come up with an algorithm that would use the gradients to maximize a function?

Gradients are important because: 1. They can tell us when we have reached an optimum ($\nabla f(\mathbf{x}) = 0$). 2. Point in the direction in which function is changing the most.

So, we shouldn't mess them up.

To check your gradient go back to the definition:

$$\frac{\partial}{\partial x} f(x_1, \ldots, x_p) = \lim_{\delta \to 0} \frac{f(x_1 + \delta/2, \ldots, x_p) - f(x_1 - \delta/2, \ldots, x_p)}{\delta}$$

Left side can be numerically approximated using a small $\delta = 10^{-6}$:

$$\frac{f(x_1 + 0.5\delta, \ldots, x_p) - f(x_1 - 0.5\delta, \ldots, x_p)}{\delta}$$

This is called **finite difference approximation**.

Let's see this in practice.

```
In [8]: def grad_check(f,xy0,delta=1e-6,tolerance=1e-7):
            f0,g0 = f(xy0)
            p = len(xy0)
            finite_diff = numpy.zeros(p)
            gradient_correct = True
            for i in range(p):
                xy1 = numpy.copy(xy0)
                xy2 = numpy.copy(xy0)
                xy1[i] = xy1[i] - 0.5*delta
                xy2[i] = xy2[i] + 0.5*delta
                f1,_ = f(xy1)
                f2,_ = f(xy2)
                finite_diff = (f2 - f1)/(delta)
                if (abs(finite_diff - g0[i])>tolerance):
                    print("Broken partial",i," Finite Diff: ",finite_diff," Partial: ",g0[i])
                    gradient_correct = False
            return gradient_correct

        def f_broken(xy):
            x = xy[0]
            y = xy[1]
            f = -(x-1.0)**2.0 - (y-2.0)**2.0 + 0.5*x*y
            dfdx = -2.0*(x-1.0) + 0.5*y
            dfdy = -2.0*(y-2.0) - 0.5*x
            grad = [dfdx,dfdy]
            return f,grad

        def f_correct(xy):
            x = xy[0]
            y = xy[1]
            f = -(x-1.0)**2.0 - (y-2.0)**2.0 + 0.5*x*y
            dfdx = -2.0*(x-1.0) + 0.5*y
            dfdy = -2.0*(y-2.0) + 0.5*x
            grad = [dfdx,dfdy]
            return f,grad

        xy = numpy.asarray([1.0,1.0])
        print("f_broken has correct gradient:", grad_check(f_broken,[1.0,1.0]))
        print("f_correct has correct gradient:", grad_check(f_correct,[1.0,1.0]))
```

```
Broken partial 1  Finite Diff:  2.50000000035  Partial:  1.5
f_broken has correct gradient: False
f_correct has correct gradient: True
```

Once we are sure that the gradients are correct, we can proceed to optimize the function.

The algorithm we introduce next is called **gradient ascent** or **gradient descent** depending on whether we are maximizing or minimizing a function.

```
In [9]: def gradient_ascent(f,x,init_step,iterations):
            f_val,grad = f(x)                           # compute function value and gradient
            f_vals = [f_val]
            for it in range(iterations):                # iterate for a fixed number of iteratio
                done = False                            # initial condition for done
                line_search_it = 0                      # how many times we tried to shrink the
                step = init_step                        # reset step size to the initial size
                while not done and line_search_it<100:  # are we done yet?
                    new_x = x + step*grad               # take a step along the gradient
                    new_f_val,new_grad = f(new_x)       # evaluate function value and gradient
                    if new_f_val<f_val:                 # did we go too far?
                        step = step*0.95                # if so, shrink the step-size
                        line_search_it += 1             # how many times did we shrank the step
                    else:
                        done = True                     # better than the last x, so we move on

                if not done:                            # did not find right step size
                    print("Line Search failed.")
                else:
                    f_val = new_f_val                   # ah, we are ok, accept the new x
                    x = new_x
                    grad = new_grad
                    f_vals.append(f_val)
                plt.plot(f_vals)
            plt.xlabel('Iterations')
            plt.ylabel('Function value')
            return f_val, x
```

17 Linear Regression -- likelihood

We start by writing out a likelihood for linear regression is

$$\mathcal{L}(\beta_0,\beta,\sigma^2|\mathbf{x},\mathbf{y}) = \prod_{i=1}^{N} p(y|\mathbf{x},\beta_0,\beta,\sigma^2) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}\left(y_i - (\beta_0 + \sum_j x_j\beta_j)\right)^2\right\}$$

Log-likelihood for linear regression is

$$\log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} \left[-\frac{1}{2} \log 2\pi\sigma^2 - \frac{1}{2\sigma^2} \left(y_i - (\beta_0 + \sum_j x_j \beta_j) \right)^2 \right].$$

18 Linear Regression -- gradient of log-likelihood

Partial derivatives

$$\frac{\partial}{\partial \beta_0} \log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} -\frac{1}{\sigma^2} \left(y_i - (\beta_0 + \sum_j x_j \beta_j) \right) (-1)$$
$$\frac{\partial}{\partial \beta_k} \log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) = \sum_{i=1}^{N} -\frac{1}{\sigma^2} \left(y_i - (\beta_0 + \sum_j x_j \beta_j) \right) (-x_k) \quad , k \in \{1, \ldots, p\}$$

We will refer to expression $y_i - (\beta_0 + \sum_j x_j \beta_j)$ as **residual**.
Hence gradient (with respect to βs)

$$\nabla \log \mathcal{L}(\beta_0, \beta, \sigma^2 | \mathbf{x}, \mathbf{y}) = \begin{bmatrix} \sum_{i=1}^{N} -\frac{1}{\sigma^2} \left(y_i - (\beta_0 + \sum_j x_j \beta_j) \right) (-1) \\ \sum_{i=1}^{N} -\frac{1}{\sigma^2} \left(y_i - (\beta_1 + \sum_j x_j \beta_j) \right) (-x_1) \\ \vdots \\ \sum_{i=1}^{N} -\frac{1}{\sigma^2} \left(y_i - (\beta_0 + \sum_j x_j \beta_j) \right) (-x_p) \end{bmatrix}$$

```
In [10]: X = numpy.asarray([[3.0,3.0,4.0,0.0],[3.0,3.0,3.0,0.0],[3.0,2.0,2.0,5.0],
         [2.0,2.0,4.0,0.0],[3.0,3.0,4.0,4.0],[4.0,4.0,3.0,0.0],
         [2.0,2.0,4.0,0.0],[2.0,2.0,2.0,0.0],[3.0,4.0,2.0,5.0],
         [2.0,2.0,3.0,0.0],[2.0,4.0,2.0,0.0],[3.0,3.0,3.0,0.0],
         [2.0,2.0,4.0,5.0],[4.0,3.0,2.0,6.0],[3.0,4.0,4.0,4.0],
         [3.0,2.0,2.0,6.0],[3.0,3.0,3.0,0.0],[3.0,2.0,4.0,0.0],
         [3.0,3.0,4.0,4.0],[3.0,4.0,2.0,0.0]])

         Y = numpy.asarray([4.0,3.0,2.0,3.0,3.0,4.0,3.0,2.0,3.0,
         3.0,3.0,3.0,2.0,2.0,3.0,2.0,3.0,3.0,3.0,3.0])

         def linear_regression_log_likelihood(Y,X,betas,sigma2=1.0):
             ll = 0
             beta0 = betas[0]
             beta = betas[1:]
             dlldbeta0 = 0
             dlldbeta = numpy.zeros(len(beta))
             for (x,y) in zip(X,Y):
                 ll = ll -0.5*numpy.log(2*numpy.pi*sigma2)
                 res = y - beta0 - numpy.sum(x*beta)
                 ll = ll - 1.0/(2.0*sigma2)*(res**2.0)
                 dlldbeta0 = dlldbeta0 - 1.0/sigma2*res*(-1)
                 dlldbeta = dlldbeta - 1.0/sigma2*(res*(-x))
```

```
            grad = numpy.zeros(len(beta)+1)
            grad[0] = dlldbeta0
            grad[1:] = dlldbeta
            return ll, grad

        init_beta = [0.1]*5
        f = lambda betas: linear_regression_log_likelihood(Y,X,betas)
        grad_check(f,init_beta)
```

```
Out[10]: True
```

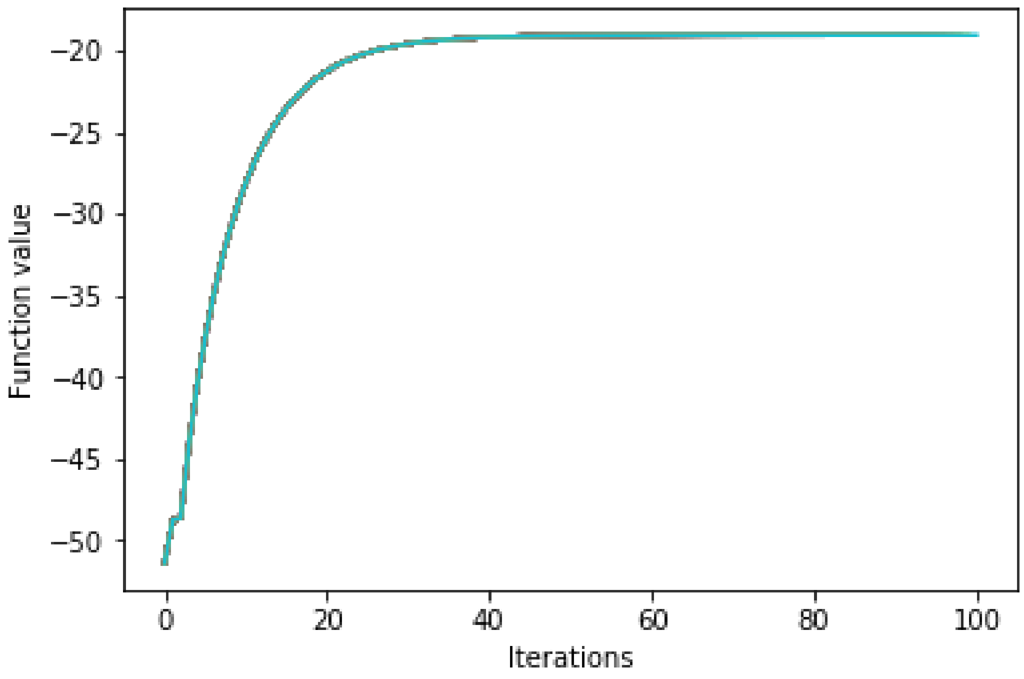
19 Fitting linear regression using gradient ascent

We are now ready to optimize our model.

We will aim to find maximum of the log-likelihood function using the gradient ascent algorithm we implemented.

```
In [11]: init_beta = [0.1]*5
         init_step = 0.01
         iterations = 1000
         f = lambda betas: linear_regression_log_likelihood(Y,X,betas)
         [f_best,betas_mle] = gradient_ascent(f,init_beta,0.01,100)
         names = ['0','COMP410','MATH233','STOR435','Beers']
         for (name,beta) in zip(names,betas_mle):
             print('Beta',name,'=',beta)
```

```
Beta 0 = 0.220839910747
Beta COMP410 = 0.278268720838
Beta MATH233 = 0.361256724745
Beta STOR435 = 0.347500790314
Beta Beers = -0.131046074777
```



20 Today

- Introduced Linear Regression
- Introduced Gradient Ascent/Descent
- Implemented Linear Regression log-likelihood and gradient ascent
- Fit a simple model for COMP 755 grades.