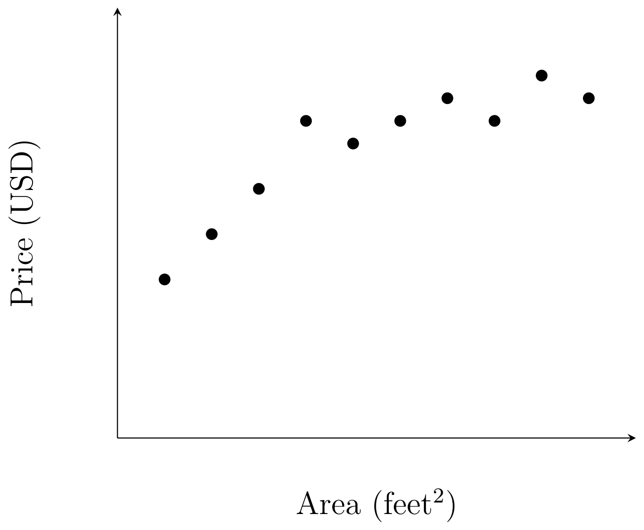
Lecture 1

1 Basic notations

- Arthur Samuel (1959). Machine Learning: field of study that gives computers the ability to learn without being explicitly programmed. He wrote checker program computer against himself. Computer learned how to play checkers better than Arthur Samuel.
- Tom Mitchell (1998) Well-posed learning problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

Let us consider the example: we will try to predict housing prices. Features: living area (feet^2); output: price. The question is to find the relationship between living area and price.

Living area (feet^2)	**USD** ($\times$1000)
2104	400
1416	232
1534	315
852	178
1940	240
...	...



Notations:

- m: number of training examples
- x: input variables or features (notice that x is a column vector)
- y: output variable or target
- (x, y): training example (sample)
- $(x^{(i)}, y^{(i)})$: i-th training example

$$\begin{tikzcd}\boxed{\text{Training set}} \arrow[r] & \boxed{\text{Learning algorithm}} \arrow[r] & \boxed{\text{Hypothesis } h}\end{tikzcd}$$

2 Linear regression

We will choose the hypothesis in the following form:

$$h(x) = \theta_0 + \theta_1 x$$

More complicated example: we add number of bedrooms in the houses, x_1 is a size, x_2 is a number of bedrooms, then the hypothesis:

$$h(x) = h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

For conciseness, define $x_0 = 1$, then

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x,$$

where n is a number of features, θ_i are **parameters**. The task of learning algorithm is to learn parameters from the training set. To learn the parameters we should choose the cost (error) function. The natural choice is

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

and we try to find

$$\min_\theta \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2.$$

2.1 Python implementation

There are a lot of useful packages in Python. To load them we use the following commands:

```
In [1]: from mpl_toolkits.mplot3d import Axes3D
        from matplotlib import cm
        import matplotlib.pyplot as plt
        import numpy as np
        import sklearn.datasets as ds
        from sklearn.linear_model import LinearRegression
```

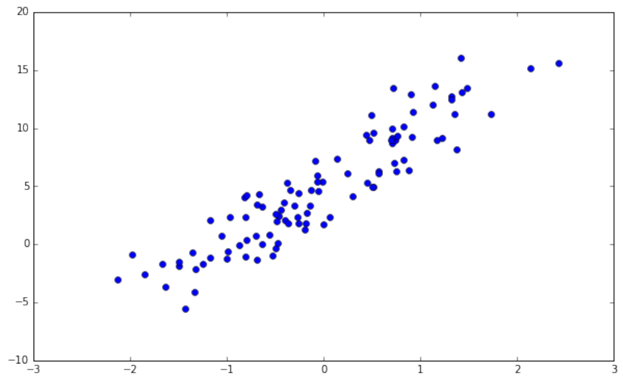
We use the package `matplotlib` for plotting, the package `numpy` for fast matrix calculations, the package `sklearn` to work with data and use different built-in methods.

Before we train our simple linear regression model we should get some data. For this example we generate data with 100 samples and 1 feature using method `make_regression` from the package `sklearn`.

```
In [2]: X_train,y_train = ds.make_regression(n_samples=100,
                                             n_features=1,
                                             n_informative=1,
                                             noise=20.0,
                                             bias=50,
                                             random_state=2016)
        y_train = y_train/10 # scale target to have nicer pictures
```

To visualize the data we use method `plot` from the `matplotlib` package.

```
In [3]: fig = plt.figure(figsize=(10,6))
        ax = plt.axes(xlim=(-3, 3), ylim=(-10, 20))
        ax.plot(X_train[:,0],y_train,'o')
        plt.show()
```



Now we build our first model using the method `LinearRegression` from the package `scikit-learn`. Notice that when you work with `scikit-learn` training different models is very similar. Usually, training contains the following steps:

1. Create a class for the model and define all parameters
2. Use the method `fit` to train the model on the training examples
3. Use the method `predict` or `predict_proba` to obtain predictions on the test examples.

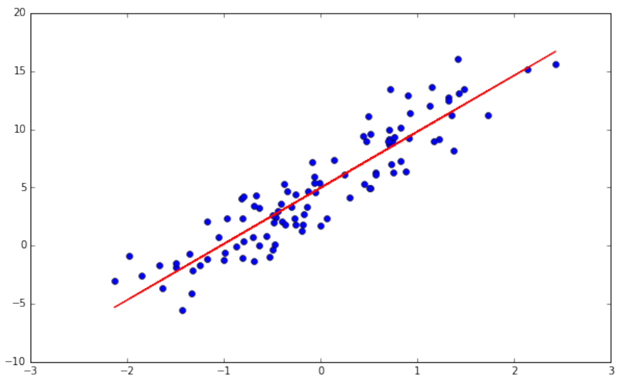
In our example we do not have any test examples, that is why we train the linear regression model and obtain the coefficients from our model.

```
In [4]: model = LinearRegression() # create an object of the LinearRegression class
        model.fit(X_train,y_train) # train the model on the training examples
        intercept = model.intercept_
        slope = model.coef_[0]
        best_fit = X_train[:,0] * slope + intercept
        print "Parameters thetas: " + str(round(model.intercept_,2)) +\
              ", " + ", ".join([str(round(x,2)) for x in model.coef_])
```

```
Parameters thetas: 4.98, 4.83
```

Finally, we visualize our data along with the regression line.

```
In [5]: fig = plt.figure(figsize=(10,6))
        ax = plt.axes(xlim=(-3, 3), ylim=(-10, 20))
        ax.plot(X_train[:,0],y_train,'o')
        line, = ax.plot([], [], lw=2)
        plt.plot(X_train[:,0], best_fit, 'k-', color = "r")
        plt.show()
```



3 Gradient descent

The idea of gradient descent is to start from the initial random value of θ (say $\theta = \vec{0}$). Then we keep changing θ to reduce $J(\theta)$. The new point could be obtained by choosing the direction of steepest descent (because the aim is to go down as quickly as possible). The procedure converges to the local minimum of $J(\theta)$. The problem is that if we start from different initial point, then we could find another minimum. If the function $J(\theta)$ is very complicated, then it could have a lot of minima (check the Python implementation section).

In more details: we update θ on each iteration by the following rule

$$\theta_i := \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta)$$

(notice, this is an operation of assignment).

We will find the partial derivative for one training example:

$$\frac{\partial}{\partial \theta_i} J(\theta) = \frac{\partial}{\partial \theta_i} \left(\frac{1}{2} (h_\theta(x) - y)^2 \right) = 2 \cdot \frac{1}{2} \cdot (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_i} (h_\theta(x) - y) =$$
$$= (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_i} (\theta_0 x_0 + \ldots + \theta_n x_n - y) = (h_\theta(x) - y) \cdot x_i$$

Then the update will be

$$\theta_i := \theta_i - \alpha (h_\theta(x) - y) \cdot x_i,$$

where α is a **learning rate**, which shows how large your step in the gradient descent. If α is too small, the algorithm takes long time to converge, if α is too big, the algorithm can be diverge.

Then the algorithm becomes very simple.

Algorithm 1 Batch Gradient Descent

```
1: repeat
2:     for $i = 0$ to $n$ do
3:         $\theta_i := \theta_i - \alpha \sum_{j=1}^{m} (h_\theta(x^{(j)}) - y^{(j)}) \cdot x_i^{(j)}$
4: until convergence
```

Notice again that the second part in the line 3 of the Algorithm 1 is just $\frac{\partial}{\partial \theta_i} J(\theta)$.

In our problem, $J(\theta)$ does not have a complicated form, it is just quadratic surface. It converges reasonably rapidly. The gradient is decreasing (the step becomes smaller and smaller).

The name of the algorithm Batch Gradient Descent came from the idea that we look at the whole training set every step. But when training set is very big (for example, $m > 1000000$), then we need to look at a huge amount of training samples each iteration. The alternative for this is called Stochastic Gradient Descent.

Algorithm 2 Stochastic Gradient Descent

```
1: repeat
2:     for $j = 1$ to $m$ do
3:         for $i = 0$ to $n$ do
4:             $\theta_i := \theta_i - \alpha(h_\theta(x^{(j)}) - y^{(j)}) \cdot x_i^{(j)}$
5: until convergence
```

In this algorithm we update weights based on the first training example only, after on the second training example only and so on. For large datasets the Stochastic Gradient Descent is much faster. But the problem that you do not walk to the minimum in the fastest way but you still approaching to it (check the Python implementation section).

3.1 Python implementation

For our previous example with one feature we add the bias term by adding the column of 1's to the design matrix:

```
In [6]: X_train_bias = np.c_[np.ones(X_train.shape[0]), X_train]
```

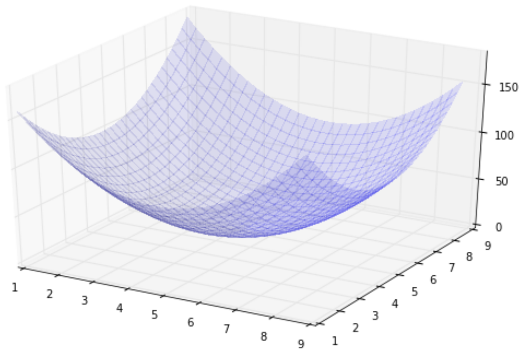
Next step will be to define the cost function $J(\theta_0, \theta_1)$ and visualize it as a function of two variables:

```
In [7]: def cost_function(x, y, theta):
            y = y.reshape((-1,1))
            theta = np.array(theta).reshape((2,-1))
            return np.sum((np.dot(x, theta) - y) ** 2, axis=0)/(2*10)

        fig = plt.figure(figsize=(10, 6))
        ax = fig.gca(projection='3d')
        plt.hold(True)
        a = np.arange(1, 9, 0.25)
        b = np.arange(1, 9, 0.25)
        a, b = np.meshgrid(a, b) # make a grid of values from a and b
        # we use some vectorization to speed up the calculations
        c = cost_function(X_train_bias, y_train,\
              np.c_[a.ravel(), b.ravel()].T).reshape(a.shape)

        surf = ax.plot_surface(a, b, c, rstride=1, cstride=1, alpha=0.1,
                               linewidth=0, antialiased=False)
        ax.set_zlim(-0.01, 180.01)

        plt.show()
```



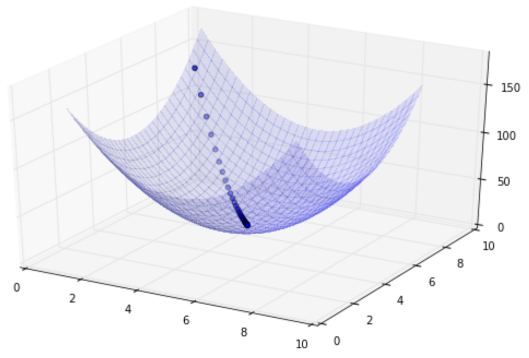
The following function is an implementation of Batch Gradient descent:

```
In [8]: def batch_gradient_descent(x, y, theta0, iters, alpha):
            theta = theta0     # initial values for theta's
            history = []       # array of theta's
            costs = []         # array of costs
            # main loop by number of iterations:
            for i in range(iters):
                history.append(theta)
                cost = cost_function(x, y, theta)[0]
                costs.append(cost)
                pred = np.dot(x, theta)
                error = pred - y
                gradient = x.T.dot(error)
                theta = theta - alpha * gradient  # update
            return history, costs

        history, costs = batch_gradient_descent(X_train_bias, y_train,
                                                theta0 = [1.2, 8],
                                                iters = 30,
                                                alpha = 0.001)
        fig = plt.figure(figsize=(10, 6))
        ax = fig.gca(projection='3d')
        plt.hold(True)
        surf = ax.plot_surface(a, b, c, rstride=1, cstride=1, alpha=0.1,
                               linewidth=0, antialiased=False)
        ax.set_zlim(-0.01, 180.01)

        t0 = np.array([x[0] for x in history])
        t1 = np.array([x[1] for x in history])
        ax.scatter(t0, t1, costs, color="k");

        plt.show()
```



As you can see, the algorithm found the fastest path to the minimum. Stochastic Gradient Descent does not find the shortest path, but in most cases it is much faster because it does not use all training examples to update the parameters:

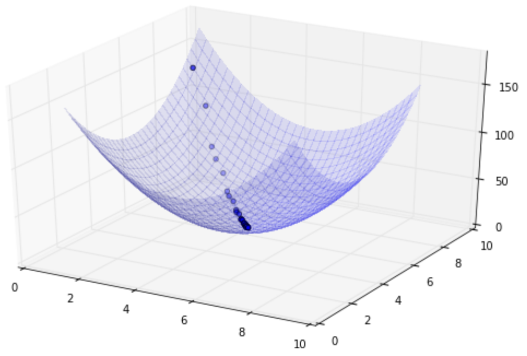
```
In [9]: def stochastic_gradient_descent(x, y, theta0, iters, alpha):
            theta = theta0     # initial values for theta's
            history = []       # array of theta's
            costs = []         # array of costs
            m = y.size         # number of training examples
            # main loop by number of iterations:
            for i in range(iters):
                for j in range(m):
                    pred = np.dot(x[j,:], theta)
                    error = pred - y[j]
                    gradient = error * x[j,:]
                    theta = theta - alpha * gradient  # update
                    if j % 40 == 0:
                        history.append(theta)
                        cost = cost_function(x, y, theta)[0]
                        costs.append(cost)
            return history, costs

        history, costs = stochastic_gradient_descent(X_train_bias, y_train,
                                                      theta0 = [1.2, 8],
                                                      iters = 10,
                                                      alpha = 0.005)

        fig = plt.figure(figsize=(10, 6))
        ax = fig.gca(projection='3d')
        plt.hold(True)
        surf = ax.plot_surface(a, b, c, rstride=1, cstride=1, alpha=0.1,
                               linewidth=0, antialiased=False)
        ax.set_zlim(-0.01, 180.01)

        t0 = np.array([x[0] for x in history])
        t1 = np.array([x[1] for x in history])
        ax.scatter(t0, t1, costs, color="k");

        plt.show()
```

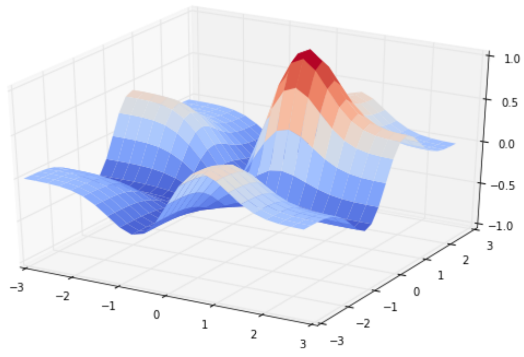


When we apply the Gradient Descent algorithm to the linear regression our straight line gradually approaching to the line which fits our data in the best way. We apply the above method `batch_gradient_descent` to fit the coefficients in the linear regression.

```
In [10]: alpha = 0.001             # learning rate
         iters = 100               # number of iterations
         theta = np.random.rand(2) # initial guess for theta's
         history, cost = batch_gradient_descent(X_train_bias,
                                                y_train,
                                                theta,
                                                iters,
                                                alpha)
         theta = history[-1]
         print "Parameters thetas after gradient descent: " +\
               ", ".join([str(round(x,2)) for x in theta])
]

Parameters thetas after gradient descent: 4.98, 4.82
```

Notice that the parameters from the manual implementation are very close to the parameters from `scikit-learn`. Finally, we show the function for which gradient descent can give different results for different initial values of θ.



4 Normal equations

There is another way to perform minimization. We will derive the solution using Linear Algebra terminology. But we need to take derivatives with respect to matrices. Given the function we need to find

$$\nabla_\theta J = \begin{bmatrix} \dfrac{\partial J}{\partial \theta_0} \\ \vdots \\ \dfrac{\partial J}{\partial \theta_n} \end{bmatrix} \in \mathbb{R}^{n+1}$$

Gradient descent iteration transforms to

$$\theta := \theta - \alpha \nabla_\theta J,$$

where left-hand and right-hand sides are $n+1$-dimensional vectors.

Definition. $f : \mathbb{R}^{m\times n} \to \mathbb{R}$ or $f(A), A \in \mathbb{R}^{m\times n}$. Then

$$\nabla_A f(A) = \begin{bmatrix} \dfrac{\partial f}{\partial A_{11}} & \dots & \dfrac{\partial f}{\partial A_{1n}} \\ \vdots & & \vdots \\ \dfrac{\partial f}{\partial A_{m1}} & \dots & \dfrac{\partial f}{\partial A_{mn}} \end{bmatrix}$$

Some facts from the Linear Algebra:

- $\operatorname{tr} AB = \operatorname{tr} BA$
- $\operatorname{tr} ABC = \operatorname{tr} CAB = \operatorname{tr} BCA$
- If $f(A) = \operatorname{tr} AB$, then $\nabla_A \operatorname{tr} AB = B^T$
- $\operatorname{tr} A = \operatorname{tr} A^T$

- If $a \in \mathbb{R}$ then $\operatorname{tr} a = a$
- $\nabla_A \operatorname{tr} ABA^TC = CAB + C^TAB^T$

Denote

$$X\theta = \begin{bmatrix} ---(x^{(1)})^T--- \\ ---(x^{(2)})^T--- \\ \dots \\ ---(x^{(m)})^T--- \end{bmatrix} \theta = \begin{bmatrix} (x^{(1)})^T\theta \\ (x^{(2)})^T\theta \\ \dots \\ (x^{(m)})^T\theta \end{bmatrix} = \begin{bmatrix} h_\theta(x^{(1)}) \\ h_\theta(x^{(2)}) \\ \dots \\ h_\theta(x^{(m)}) \end{bmatrix}$$

Remember that

$$y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix},$$

then

$$X\theta - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \dots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}.$$

Recall $z^Tz = \sum\limits_i z_i^2$. Then

$$\frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) = \frac{1}{2}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2 = J(\theta)$$

Now we should set gradient to zero:

$$\nabla_\theta J(\theta) = \vec{0}$$

It means

$$\nabla_\theta \left(\frac{1}{2}(X\theta - y)^T(X\theta - y)\right) = \frac{1}{2}\nabla_\theta \operatorname{tr}\left(\theta^TX^TX\theta - \theta^TX^Ty - y^TX\theta + y^Ty\right) =$$
$$= \frac{1}{2}\left(\nabla_\theta \operatorname{tr}\theta\theta^TX^TX - \nabla_\theta \operatorname{tr} y^TX\theta\right).$$

But

$$\nabla_\theta \operatorname{tr}\theta I\theta^TX^TX = X^TX\theta I + X^TX\theta I$$

and

$$\nabla_\theta \operatorname{tr} y^TX\theta = X^Ty,$$

then

$$\nabla_\theta J(\theta) = \frac{1}{2}\left[X^TX\theta + X^TX\theta - X^Ty - X^Ty\right] = X^TX\theta - X^Ty = 0.$$

The right hand side is called **normal equations**:

$$X^TX\theta = X^Ty$$

and finally

$$\theta = (X^TX)^{-1}X^Ty.$$