

PROJECT 2

Machine Learning

Implement the solutions of the least squares problem proposed in this document in Matlab and ipython notebooks.

- 1) Produce graphical representation of the results
- 2) Implement the Least-Mean-Squares (LMS) gradient descent method described in this document for the least squares. Notice that this is similar to the gradient method implemented in the Project 5.
- 3) Compare the above solutions with library module from optimize lib.
- 4) The comments in the notebook will explain each step of the code and should include the motivation and the theory for the least squares problem. Besides the related material from Houstis ebook you should consider the material from the slides I have load in the eclass.
The theoretical basis of the least squares problem and its solution should be part of the presentation and the degree of understanding you will demonstrate will count significantly in the grade for this project.

6) Implement the kmean algorithm in the link and the data test for the iris flower
http://en.wikipedia.org/wiki/Iris_flower_data_set

Study

5) Chapters 5, 8 in Houstis ebook

6) http://www.scipy.org/NumPy_for_Matlab_Users#head-381c9088b53dc22db3db569b05a362c7b02eb74b

7) <http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.leastsq.html#scipy.optimize.leastsq>

8) <http://structure.usc.edu/numarray/node69.html> (You need it to learn how to install the linear algebra package and to find routines for least srquares and pseudo inverse of a matrix)

9) Study the DataScience notebooks placed in the eclass documents

Linear Regression

Linear regression is the method for representing observed outputs as linear combinations of functions of the inputs. Since this method is often used to find linear relationships between variables, a common belief is that the "linear" in linear regression indicates that this method is only useful for finding simple linear relationships between variables; this is not true. The "linear" in linear regression refers to the mixing weights on the functions of the inputs, not on the form of the functions.

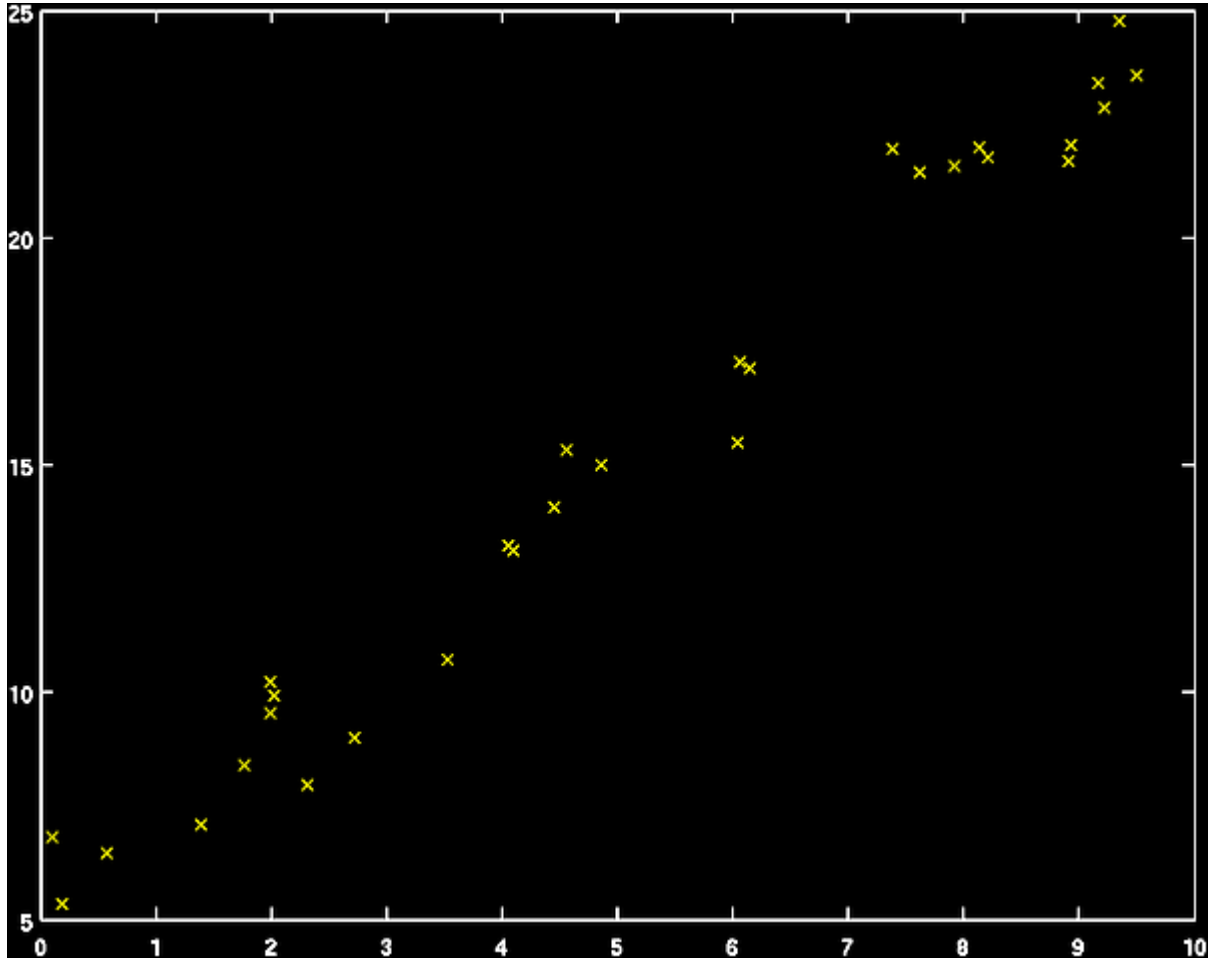
The simplest form of linear regression is the well-known case of fitting a trend line to data that appears to have a linear relationship. Consider the following data:

9.50129 23.57002

2.31139	7.95719
6.06843	17.26218
4.85982	15.00733
8.91299	21.67951
7.62097	21.43285
4.56468	15.31852
0.18504	5.33244
8.21407	21.75544
4.44703	14.06871
6.15432	17.12194
7.91937	21.56453
9.21813	22.84794
7.38207	21.94733
1.76266	8.38893
4.05706	13.22806
9.35470	24.77616
9.16904	23.39737
4.10270	13.10976
8.93650	22.04064
0.57891	6.45224
3.52868	10.72118

8.13166	21.97765
0.09861	6.82079
1.38891	7.08604
2.02765	9.91330
1.98722	10.22844
6.03792	15.48212
2.72188	9.00279
1.98814	9.54743

Plot of the above data



Visual inspection indicates that the relationship between the inputs (x-axis) and the outputs (y-axis) is linear. So, we want to find a function in the form of a line that fits the data:

$$\hat{y} = b + mx \quad (1)$$

The regression problem here is to find the constants m and b such that the value of $\hat{y}$ for a given x is, on average, close to the observed value of y .

We can frame this problem more generally by realizing that the formula for a line (equation 1) represents y as a 1st order polynomial of x . We can write this formally as:

$$\hat{y} = w_0\phi_0(x) + w_1\phi_1(x) = \sum_{i=0}^1 w_i\phi_i(x) \quad (2)$$

where

$$w_1 = m; \phi_1(x) = x^1 = x; w_0 = b; \phi_0(x) = x^0 = 1$$

In this form, $\hat{y}$ is clearly a linear combination of functions of x . (Note that the functions of x , e.g. $\phi_i(x)$, are called **basis functions**. The rather trivial basis function $\phi_0(x)=1$ is usually called the "bias" or "DC bias".) We can conveniently represent equation 2 for the whole dataset as a linear equation of matrices and vectors:

$$\begin{aligned} w_0\phi_0(x[1]) + w_1\phi_1(x[1]) &= y[1] \\ w_0\phi_0(x[2]) + w_1\phi_1(x[2]) &= y[2] \\ &\vdots \\ w_0\phi_0(x[n]) + w_1\phi_1(x[n]) &= y[n] \end{aligned}$$

which we can write as a matrix multiplication:

$$\begin{bmatrix} \phi_0(x[1]) & \phi_1(x[1]) \\ \phi_0(x[2]) & \phi_1(x[2]) \\ \vdots & \vdots \\ \phi_0(x[n]) & \phi_1(x[n]) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} y[1] \\ y[2] \\ \vdots \\ y[n] \end{bmatrix}$$

or more succinctly,

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

Note that the columns of $\mathbf{\Phi}$ are the basis functions applied to x (we leave off the functional indication for notational efficiency), and the vector $\mathbf{w}$ specifies how to linearly combine the basis functions of x to produce y . You may recognize this as a simple system of linear equations. However, there is a problem: the number of unknowns, the w 's, is 2, but the number of equations is the number of data points, n . With more equations than unknowns, this is an **overdetermined** system of equations. If this system of equations were not overdetermined, the solution would be simply:

$$\mathbf{w} = \mathbf{\Phi}^{-1} \mathbf{y} \tag{3}$$

However, because $\mathbf{\Phi}$ is not square, we can not invert it. Instead, we have to set up an optimization problem to find the best approximate solution for $\mathbf{w}$. To do this, we have to define what we mean by "best". For linear regression, the most common

definition of "best" is "best in a least squares sense", meaning that we want to minimize the sum of squares error:

$$E = \sum_{i=1}^n (\hat{y}[i] - y[i])^2$$

where

$$\hat{y}[i] = \sum_{j=0}^1 w_j \phi_j(x[i])$$

Using matrix notation,

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

We can find this solution using the standard approach of finding the stationary point where:

$$\nabla_{\mathbf{w}} E = 0$$

$$\begin{aligned} \nabla_{\mathbf{w}} [(\Phi \mathbf{w} - \mathbf{y})^T (\Phi \mathbf{w} - \mathbf{y})] &= 0 \\ \nabla_{\mathbf{w}} [\mathbf{w}^T \Phi^T \Phi \mathbf{w} - \mathbf{y}^T \Phi \mathbf{w} - \mathbf{w}^T \Phi^T \mathbf{y} + \mathbf{y}^T \mathbf{y}] &= 0 \\ 2\Phi^T \Phi \mathbf{w} - 2\Phi^T \mathbf{y} &= 0 \end{aligned}$$

which yields

$$\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y} \quad (4)$$

Note that equation 4 is very similar in form to equation 3, with the product $[\Phi(\mathbf{x})^T \Phi(\mathbf{x})]^{-1} \Phi(\mathbf{x})^T$ appearing in equation 4 where the inverse of the phi matrix, Φ^{-1} appears in equation 3. We could not actually invert Φ because it was not square (the system was overdetermined), but by solving the above optimization problem, we have found the best approximation of the inverse in the least-squares sense. This approximation is called the **pseudoinverse** or **Moore-Penrose pseudoinverse** and is notated as:

$$\Phi^\dagger = (\Phi^T \Phi)^{-1} \Phi^T$$

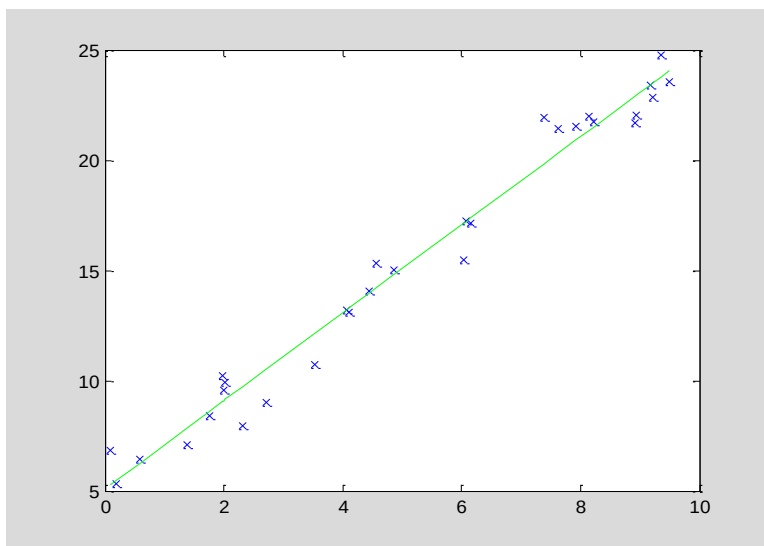
We can now write the solution to any linear regression problem as:

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

Returning to the example given at the top, we can perform the regression using matlab's **pinv** command:

```
load -ASCII 'C:\Users\elias\Dropbox\S3\DataMining\ex1_data.mat';
x=ex1_data(:,1);y=ex1_data(:,2);
phi=[x.^(0) x.^(1)];
w=pinv(phi)*y
plot(x,y,'x');
hold on;
[Y,I]=sort(x);
plot(x(I),phi(I,:)*w,'g');
```

```
w =
5.0988
1.9975
```



PINV Pseudoinverse (read chapter 8 Houstis ebook)

$X = \text{PINV}(A)$ produces a matrix X of the same dimensions as A' so that $A * X * A = A$, $X * A * X = X$ and $A * X$ and $X * A$ are Hermitian. The computation is based on $\text{SVD}(A)$ and any singular values less than a tolerance are treated as zero. The default tolerance is $\text{MAX}(\text{SIZE}(A)) * \text{NORM}(A) * \text{EPS}(\text{class}(A))$.

$\text{PINV}(A, \text{TOL})$ uses the tolerance TOL instead of the default.

Linear Regression and Data Mining Problems

1. The Approach to Data Mining Problems

The process of simple regression follows the same pattern of most data-mining and pattern recognition techniques. We now consider more closely the process we followed in the first example.

- First, we observed the data and decided on a representative model. In the example, we decided to use a simple linear model. The process of deciding on a particular model is called **model selection**. A model will have a set of parameters called **model parameters** which can be adjusted. In the example, the model parameters were the weights.
- After selecting a model, we defined a measure of error between our model's result and the truth. In the example, the error was the sum-of-squares error. This measure is called a **loss function**.
- After selecting a model and defining a loss function, we formulated an optimization problem to find the model parameters (e.g., the weights) that minimize the value of the loss function. Solving this optimization problem is called **model fitting**. In the example, we found the minimum analytically by solving for the stationary point of the loss function with respect to the weights. In general, data mining problems do not have simple analytic solutions, and finding efficient algorithms for model fitting is a fundamental pursuit.

In summary, the central approach to data mining problems is:

1. Define the model.
2. Define the loss function.
3. Decide how to minimize the loss function given the model.

2. Generalization and Evaluation

It is also important to consider our motivation for fitting a model to data. In the case of regression, we may just desire a simple functional explanation of noisy observations.

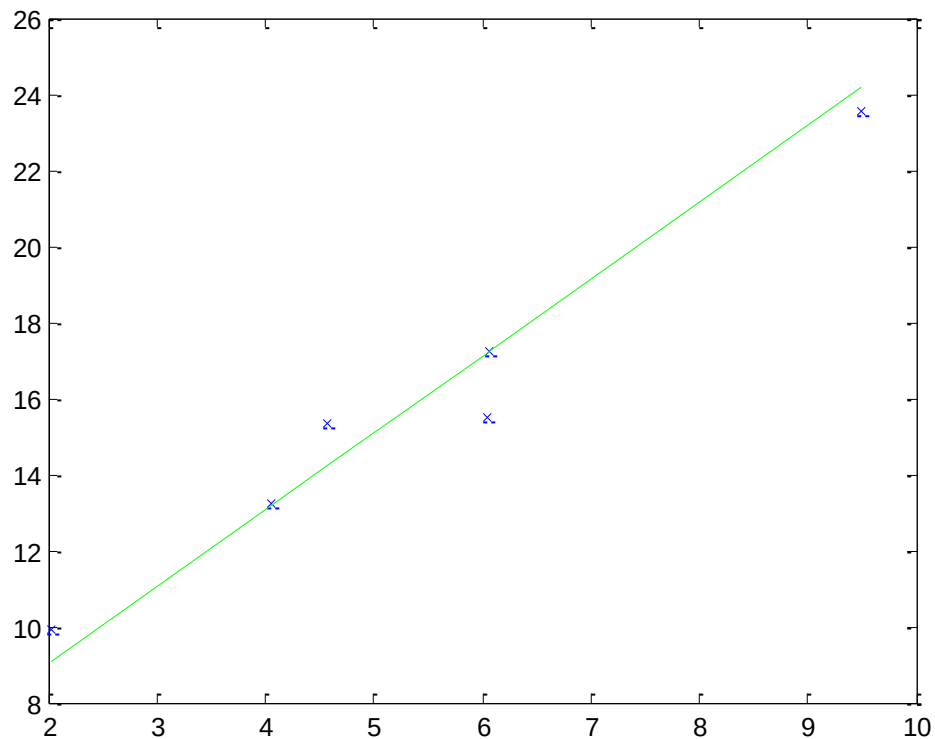
However, often, we intend to use the model for prediction of future outputs from inputs. A model's ability to accurately predict outputs for future inputs is called **generalization**.

In order to determine how well the model will generalize to future data generated by the same process, we can hold out some of the data that we used to fit the model and use it to test the model. The set that we hold out we call the **test set**, and the set we use to train the model we call the **training set**. A commonly used split is to use 80% of the data for the training set, and the remaining 20% for the test set.

```
load -ASCII 'C:\Users\elias\Dropbox\S3\DataMining\ex1_data.mat';
tmp = rand(1,size(ex1_data,1)); % assign a random number to each data point
[Y,I]=sort(tmp); % sort those random numbers
test_set_indices=I(1:floor(size(ex1_data)*0.2)); % assign the first 20% to
test
training_set_indices=setdiff(I,test_set_indices); % and the rest to train
training_set=ex1_data(training_set_indices,:);
test_set=ex1_data(test_set_indices,:);
x_test = test_set(:,1);
y_test = test_set(:,2);
x_train = training_set(:,1);
y_train = training_set(:,2);
phi_train=[x_train.^(0) x_train.^(1)]; % do the regression
w=pinv(phi_train)*y_train;

%Now we have learned the weights from the training data.
%We need to determine how well these learned weights work for
%the unseen test set points.

phi_test = [x_test.^(0) x_test.^(1)]; % form the matrix of basis functions
y_test_predicted = phi_test*w; % find the estimates of y_test
[Y,I]=sort(x_test);
plot(x_test(I),y_test(I),'x');
hold on;
plot(x_test(I),y_test_predicted(I),'g');
```



By inspection, the fit on the test points appears to be quite good. We can compute the mean-square-error (MSE) explicitly for the test set:

```
sum((y_test-y_test_predicted).^2)/length(y_test)
```

```
ans =
```

```
0.7587
```

How do we decide if this is a good score (note that your result may vary)? Let's compare to the MSE on the training set:

```
sum((y_train - phi_train*w).^2)/length(y_train)
```

```
ans =
```

```
0.8772
```

The MSE on the test set is nearly equivalent to the MSE on the training set, indicating that the weights learned on the training set do generalize well. We would thus expect that, assuming that whatever process generated our training observations generates future points, the parameters we learned from the training set will allow us to make good predictions of the targets (the unobserved values; the y 's) from the features (the observed values; the x 's).

Online Regression & Gradient Descent

The pseudoinverse is an exact solution of the squared error minimization problem. Of course, we can't do better than an exact solution; however, there are several situations where we can not compute the pseudoinverse.

Note that computing the pseudoinverse requires constructing the entire basis function matrix (Φ) at once. For large datasets, this matrix may not fit in memory, or may take an extremely long time to compute. Additionally, we may not have all of the training data available at once. In many real-world applications, e.g. stock-price forecasting, data becomes available sequentially. Computing the pseudoinverse will not work in such applications.

Recall the regression minimization problem:

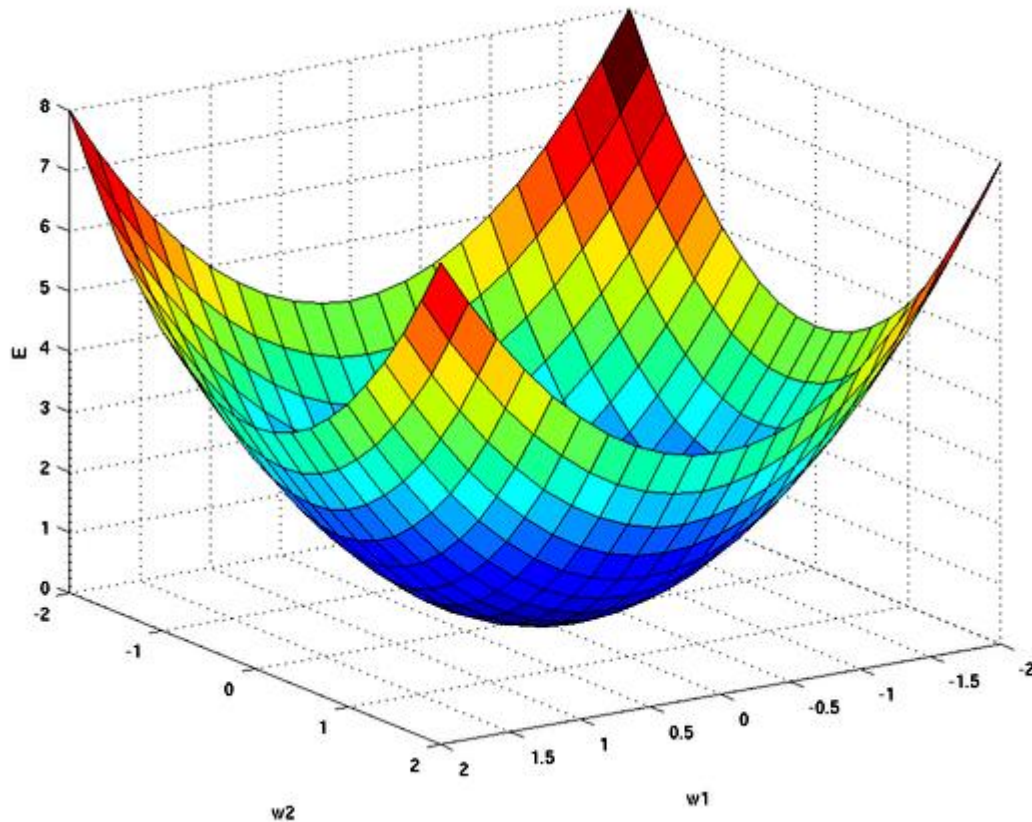
The error between our model of the data and the actual data is:

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

We want to find the $\mathbf{w}$ (vector of weights) that minimizes E (the total error). This is a simple quadratic minimization problem; the minimum will exist where the gradient (derivative) of E w.r.t. $\mathbf{w}$ is 0 - that is, at a point where any change to $\mathbf{w}$ results in an increase >0 E . In matrix notation, that is:

$$\nabla_{\mathbf{w}} E = 0$$

It is important to visualize what is happening here. The error, E takes a scalar value for every possible choice of values in the vector $\mathbf{w}$. If $\mathbf{w}$ is a vector of length 2, E has a value for every pair (w_1, w_2) . We can thus think of E as a surface over the plane: every location on the plane is indexed by the coordinates (w_1, w_2) , and E 's value at the point is the height of the surface above the plane:



It should be clear from this picture that at one point in the "weight space" (w_1, w_2) , E is minimized. The nice thing about quadratic functions like the squared-error is that *there is only one point* that minimizes the function. Another nice thing is that the gradient (derivative) everywhere else on the surface points toward the minimum point. That is, if you place a ball *anywhere* on the error surface and let it go, it will roll toward the minimum.

The pseudoinverse is equivalent to starting at some point in the weight space and rolling the ball down the surface at exactly the correct speed so that it stops at the minimum. Knowing the exact speed requires that we see all of the data at once; however, this is often impractical. If we can only see some of the data, e.g. one data point, we can still roll our ball. For that one data point, we can find the "downhill" direction, and roll the ball in that direction. With only one point, though, we can't be completely confident that the downhill direction we see - the **local** gradient - is the same direction that we would see if we could see all of the data - the **global** gradient. Since we aren't completely sure, we roll the ball cautiously, at a slow speed in the computed direction. We then walk to where the ball stops, look at

the next data point, and repeat the process. Eventually, we'll get to a point on the surface where there is no downhill direction to roll the ball. At this point, we will have reached the minimum.

This process of rolling the ball is called **gradient descent**. Gradient descent is an iterative algorithm that attempts to step down the gradient on each step, eventually stopping at the minimum. In order to make each step, we need to compute two things: the direction, and how far to step (i.e., roll the ball). We will now state gradient descent mathematically, and look at how to calculate the direction and step size. **Note that understanding gradient descent is critical for understanding data mining.**

Recall the MSE definition:

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

If we only see a single data point $(\mathbf{x}, y)$ (where $\mathbf{x}$ is a vector of size d), then the MSE will be:

$$E = \frac{1}{2} \left(y - \sum_{j=0}^d w_j \phi_j(x_j) \right)^2 = \frac{1}{2} (y - \mathbf{w}^T \phi(x))^2$$

Let us use the notation $\mathbf{w}[t]$ to indicate the weight vector at step t . Thus, we can say that for the set of weights at step t , the error on the observed data point is:

$$E_{\mathbf{w}[t]} = \frac{1}{2} (y - \mathbf{w}[t]^T \phi(x))^2$$

and the gradient for this current set of weights is (by the chain rule):

$$\nabla E_{\mathbf{w}[t]} = (y - \mathbf{w}[t]^T \phi(x)) (-\phi(x))$$

Observe that this result consists of two parts: the first term, $(y - \mathbf{w}[t]^T \phi(x))$ is a scalar that indicates how steep the gradient is at the current weights; the second term, $(-\phi(x))$ is a vector pointing in the direction of increasing E , that is, **up** the gradient. Since the steepness and direction are based on only one point and are thus only an approximation to the direction of the minimum, we take a scaled step *down* the gradient, using the scaling coefficient η (called the **learning rate**):

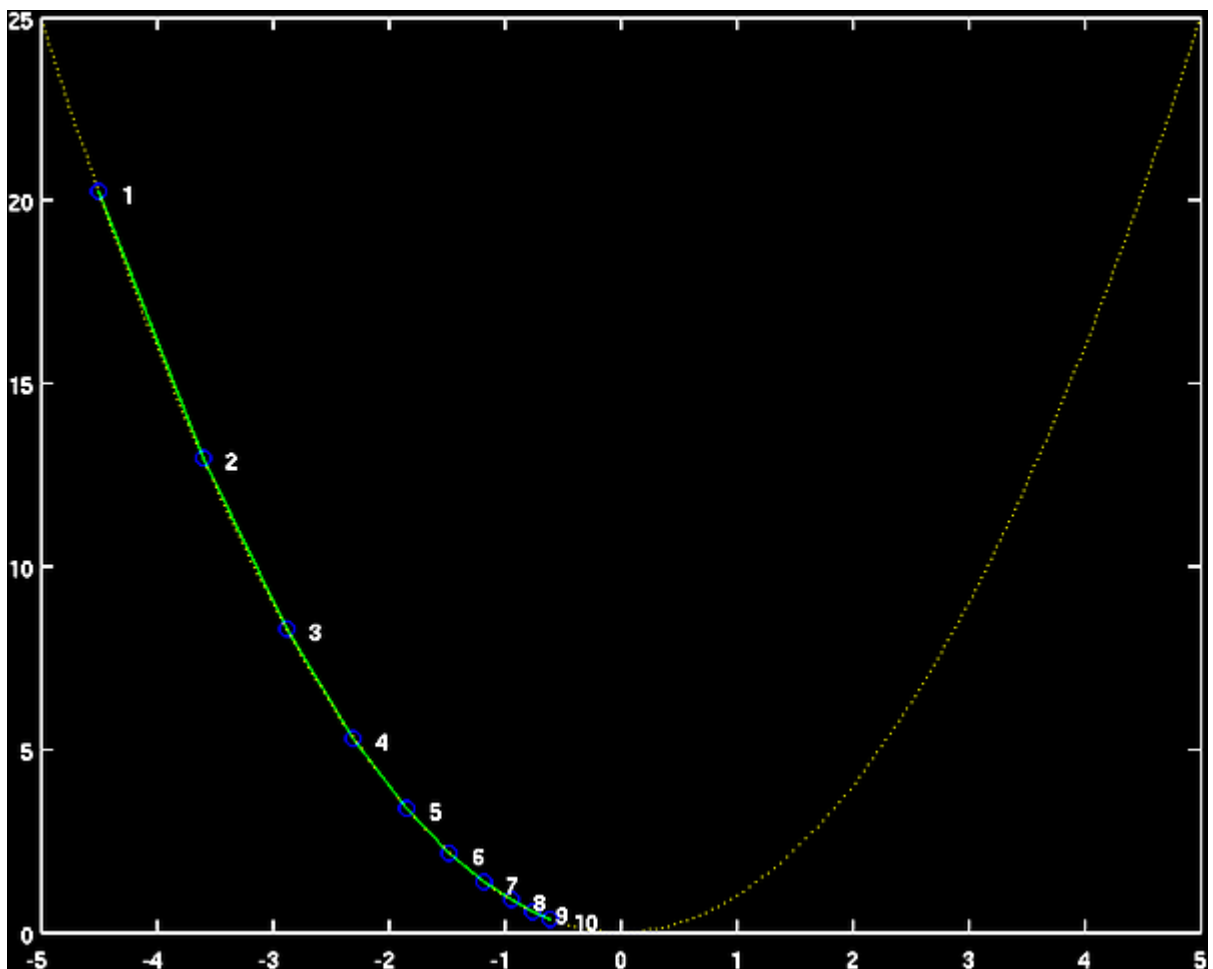
$$w[t + 1] = w[t] - \eta \nabla E_{w[t]}$$

which expands to

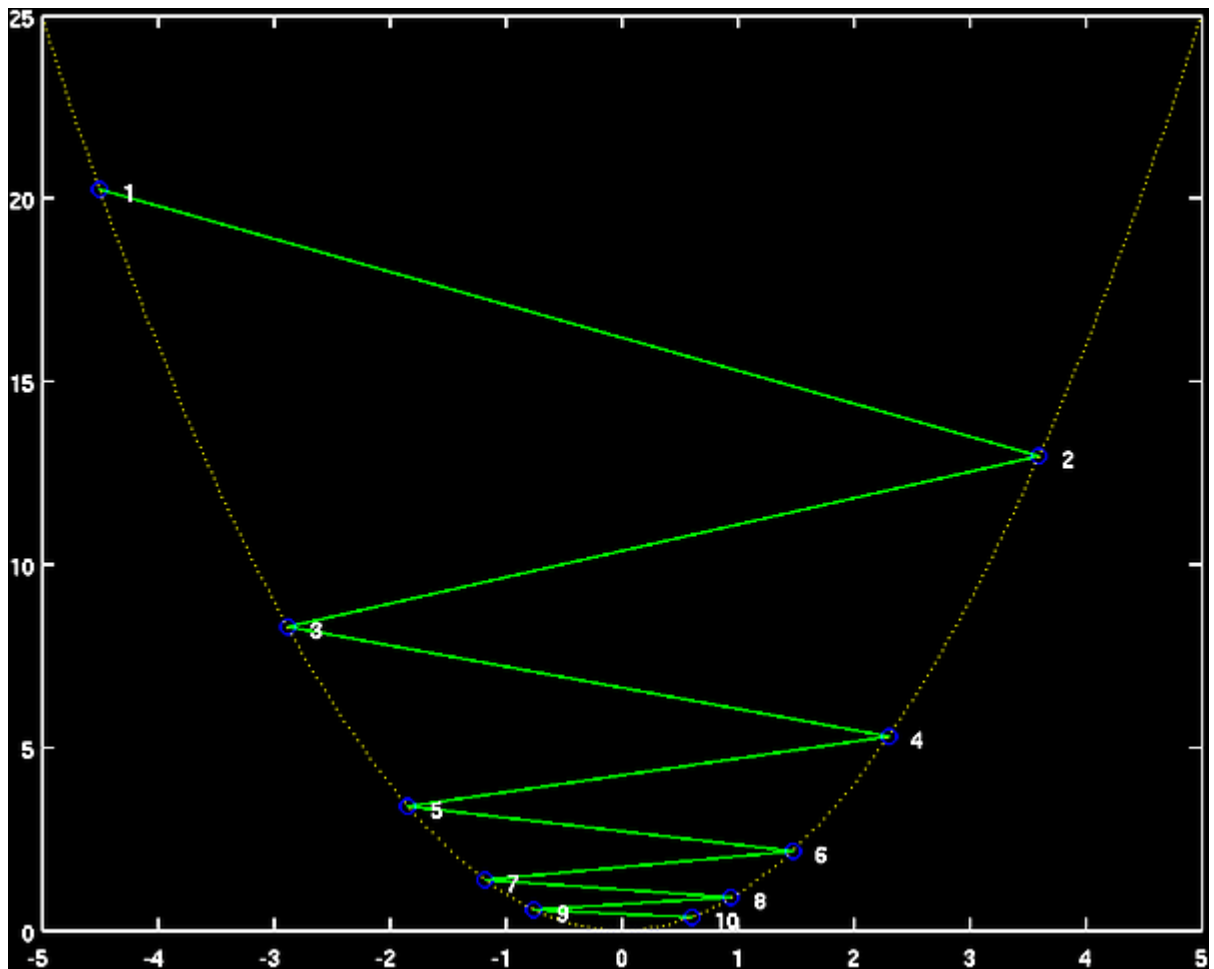
$$w[t + 1] = w[t] + \eta(y - \mathbf{w}[t]^T \phi(x)) \phi(x)$$

which is the formula for Least-Mean-Squares (LMS) gradient descent.

The gradient descent idea is straightforward, and closely resembles [Newton's Method](#) for finding the roots of a polynomial. The only odd thing is the learning rate η , which is somewhat arbitrary. η has to be less than 1; we are trying to be conservative in our leaps. If we select η to be too small, we will take tiny steps and take a long time to find the minimum:



If we select η to be too large, we will bounce back-and-forth over the minimum, and also converge slowly:



Evaluation: Cross Validation

In the previous example, we randomly split the data into two sets: a training set and a test set. We learned parameters from the data in the training set, applied those parameters to the data test set, and evaluated the quality of the predictions on the test set. This approach gives us an idea of how well our model will generalize to unseen data points.

There is, however, a potential flaw in this method. When we randomly partition the data into the training and test sets, we run some risk of creating a test set that is missing some characteristics that are present in the full data set. This could cause us to make an inaccurate estimate of our model's ability to generalize to unseen data.

The intuitive way to reduce this risk is to repeat the train/test split several times, each time calculating the error on the test set. However, even with several splits,

random partitioning does not guarantee that every point will appear in at least one test set.

Cross-validation provides a framework for creating several train/test splits and guaranteeing that each data point is in the test set at least once. The procedure is simple:

1. split the data into n equal-sized groups
2. for $i=1$ to n ,
 - (a) select group i to be the test set and all other $(n-1)$ groups to be the training set
 - (b) train the model on the training set and evaluate on the test set

Each iteration is called a **fold**. In general practice, setting $n=10$ ("10-fold cross validation") is accepted as giving a highly accurate estimate of the generalization error of a model. However, since 10 folds means retraining the models 10 times, 10-fold cross validation may be computationally expensive for large and/or complicated data sets. In such cases, 5-fold cross validation, or 10-fold cross validation run on a subset may be appropriate.

Returning to our simple linear regression example, let's implement a cross-validation framework for evaluating our model's generalizability. Pseudocode for this would be:

1. Split the available data into n equal-sized partitions.
2. for $i=1$ to n ,
 - (a) test_set =

To split the data into equal-sized folds, we can use matlab's uniform random number generator, **rand**. This function returns numbers from a uniform distribution between 0 and 1. For

```
function
split_assignments=cross_val_split(number_of_folds,number_of_data_points)

% function
split_assignments=cross_val_split(number_of_folds,number_of_data_points)
%
% returns an n x s cross-validation assignment matrix for
% n = number_of_data points
% s = number_of_splits
%
```

```

% initialize the split assignment matrix
split_assignments=zeros(number_of_data_points,number_of_folds);
% assign a random number to each data point
tmp = rand(1,number_of_data_points);
% sort the random numbers
[Y,I]=sort(tmp);
% set the starting point of the first group
group_start=1;

% iterate through the folds
for i=1:1:number_of_folds
    clear test_set_indices;
    % calculate the end of this group
    group_end = round(i*number_of_data_points/number_of_folds);
    test_set_indices = I(group_start:group_end);
    % make the start of the next group 1 past the
    % end of this group
    group_start=group_end+1;
    split_assignments(test_set_indices,i)=1;
end

```

Now, using the *cross_val_split* function, we write a script to easily run a ten-fold cross validation (save the following script into a file called "regression_ten_fold_cv.m" in your matlab directory):

```

load ex1_data;
split_assignments = cross_val_split(10,size(ex1_data,1));
for i=1:1:10
    training_set = ex1_data(find(split_assignments(:,i)==0),:);
    test_set = ex1_data(find(split_assignments(:,i)==1),:);
    % determine weights from the training set
    phi_train=[training_set(:,1).^(0) training_set(:,1).^(1)];
    w=pinv(phi_train)*training_set(:,2);
    phi_test=[test_set(:,1).^(0) test_set(:,1).^(1)];
    % apply learned weights to the test set and compute MSE
    MSE(i)=sum((test_set(:,2)-phi_test*w).^2)/size(test_set,1);
end

```

Now run the script:

```
regression_ten_fold_cv;
```

```
mean(MSE)
```

```
ans =
```

```
1.0922
```

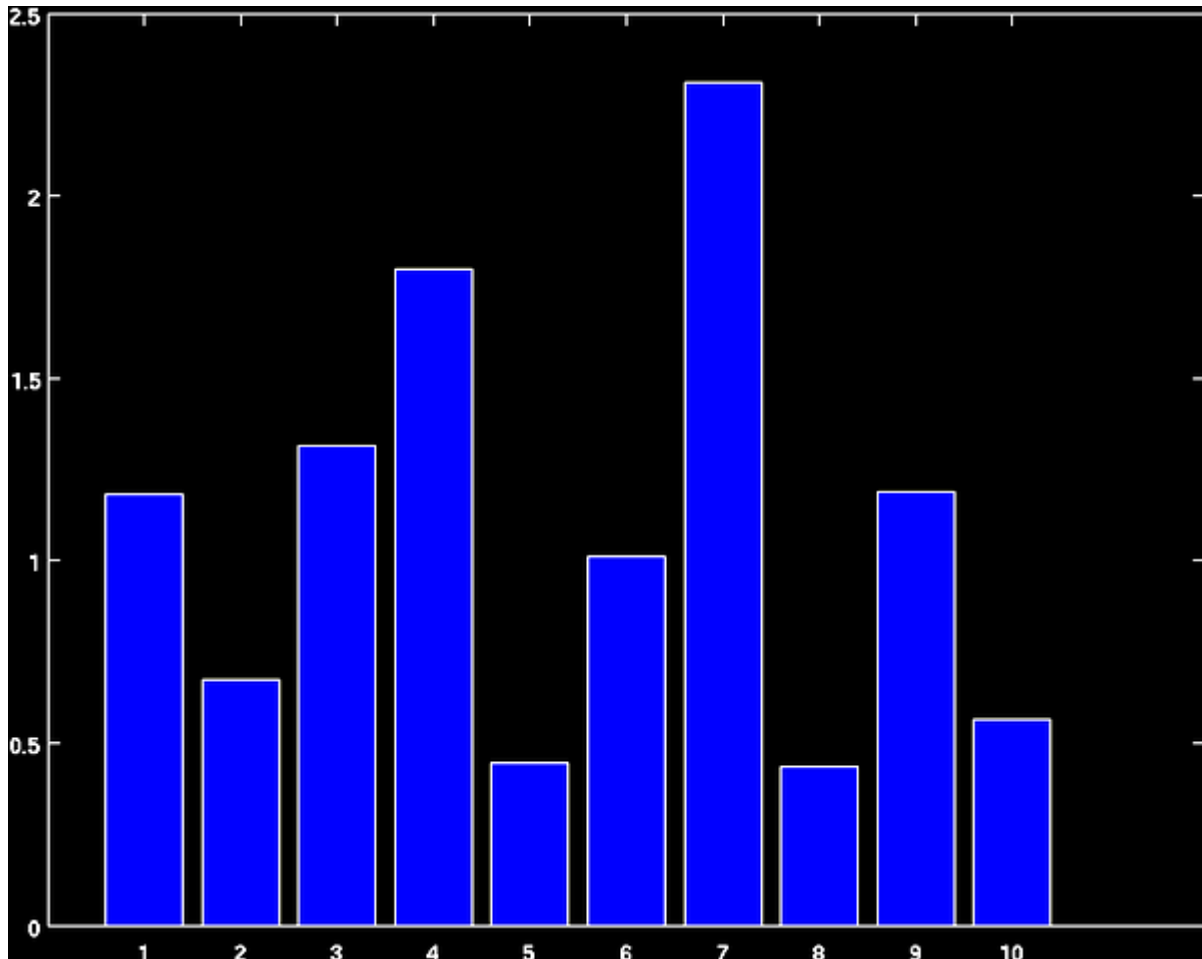
```
std(MSE)
```

```
ans =
```

0.6109

We also can visualize the MSE's as a bar chart:

```
bar(MSE) ;
```



Note that for this data set, the variance among folds is fairly high. This is because the data set is small - only 30 points. Each fold is evaluated on a test set of size 3, therefore individual outlying points have a large influence on the fold's MSE. Folds' scores generally become more consistent as the number of points in the data set increases.

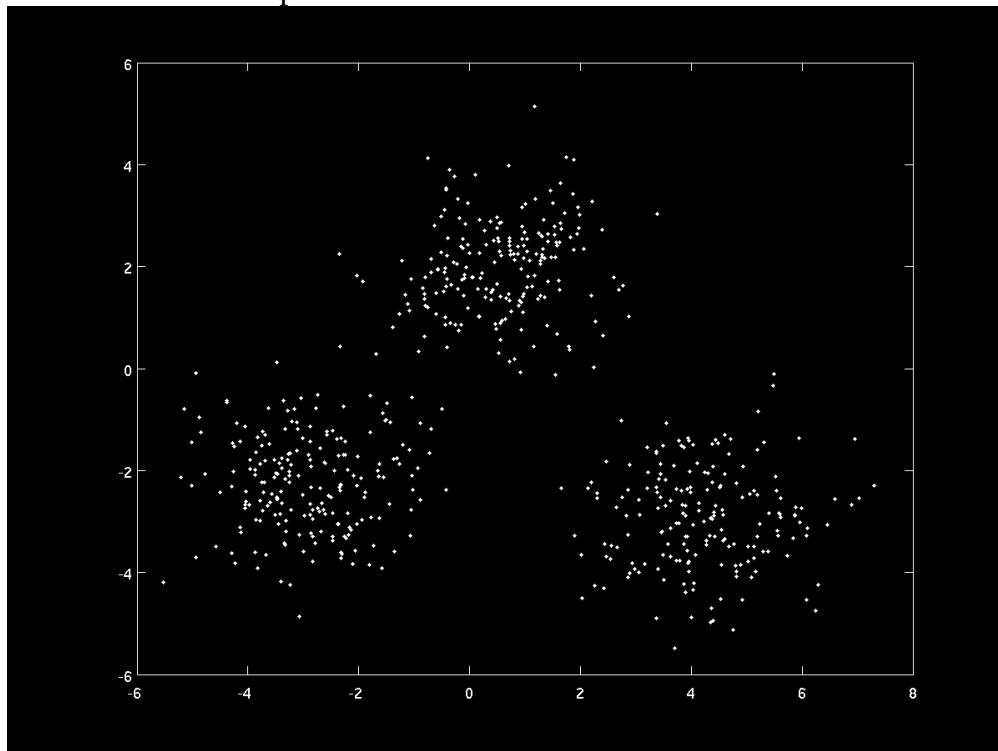
***k*-means clustering**

k-means clustering is a very simple form of clustering that uses a type of algorithm called **iterative relocation**. *k*-means is also a special case of [Expectation Maximization](#) which will be covered later.

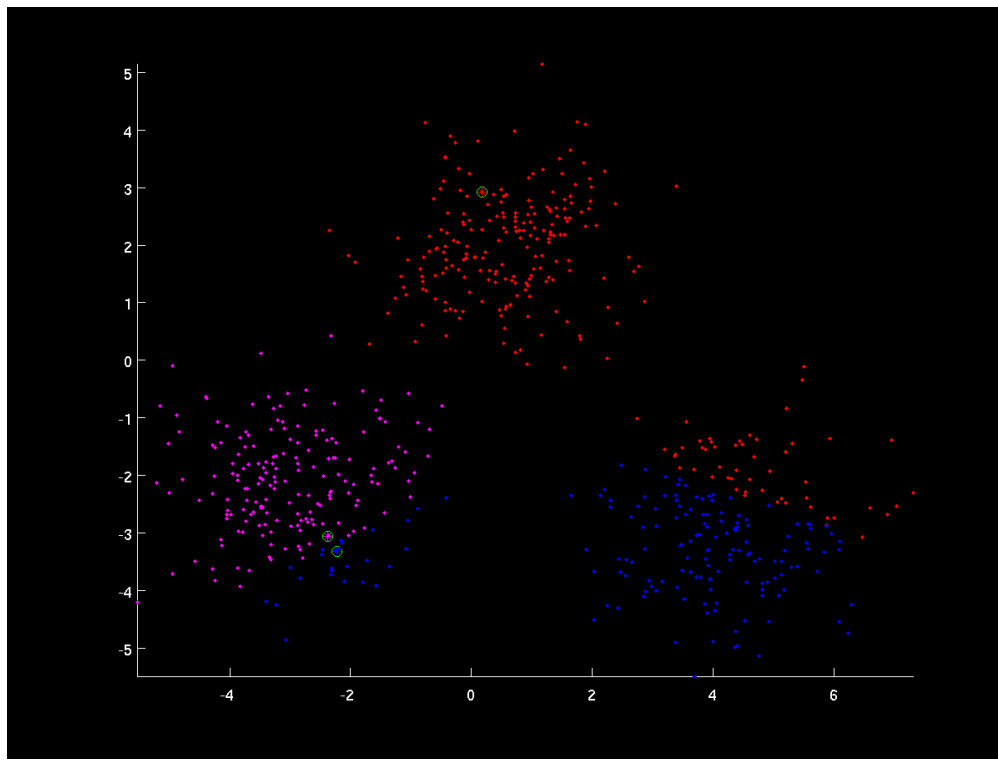
The *k*-means algorithm is very simple:

1. Choose the number of clusters k .
2. Randomly select k center points (often points in the dataset).
3. Assign each point in the dataset to its nearest center point.
4. Move the center point to the mean of all points assigned to it.
5. Repeat 3-4 until the centers stop moving.

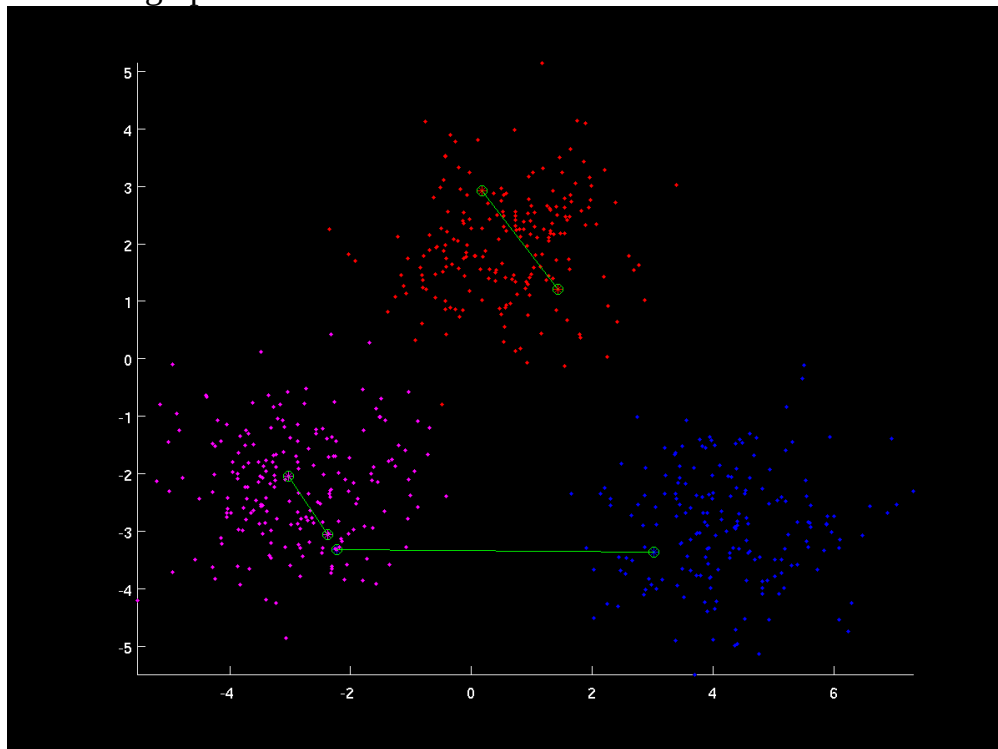
Let's look at a visual example. Consider this data set:



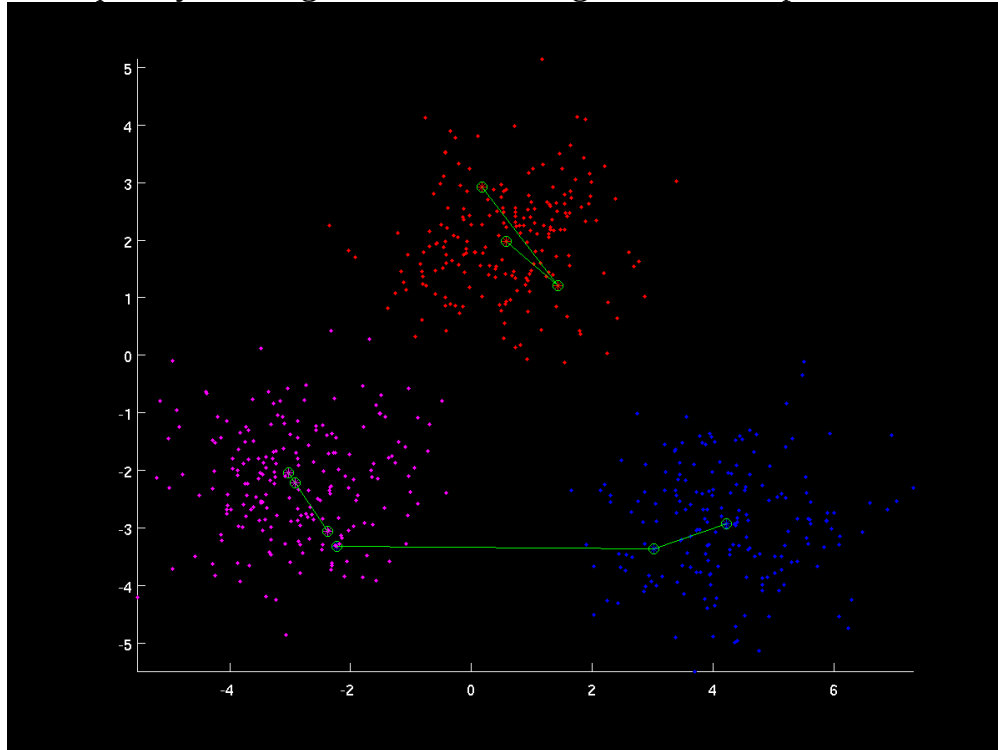
There are fairly clearly three clusters here. We randomly choose $k=3$ points from our dataset (green circles) to be centers and assign all points closest to each center the "label" of that center:



We now move the center points to the true center (mean) of its constellation of points, and reassign points to their closest center:



This process quickly converges to the clustering we would expect:



See this process as a movie [here](#)

***k*-means clustering**

Let's look at a simple implementation of *k*-means in matlab. Recall the algorithm:

1. Choose the number of clusters k .
2. Randomly select k center points (often points in the dataset).
3. Assign each point in the dataset to its nearest center point.
4. Move the center point to the mean of all points assigned to it.
5. Repeat 3-4 until the centers stop moving.

```
function labels=kmeans(x,k)

% function labels=kmeans(x,k)
% input: x, a set of vectors
%       k, the number of centers
% output: labels, the cluster membership for each point in x

% initialization
% choose points randomly from dataset to be initial centers
selector=rand(1,size(x,1)); % assign each point in x a random score
```

```

[Y,I]=sort(selector); % sort the random scores
centers(1:k,:)=x(I(1:k),:); % select the k top scoring points in x

% measure distance from each point to each center
for i=1:1:k    d(i,:)=sqrt(sum(( repmat(centers(i,:),size(x,1),1)-
x).^2,2));
end

% assign each point the label of its closest center
[Y,labels]=min(d);
labels_old=zeros(1,size(x,1));

% begin iterative reassignment algorithm
while(sum(abs(labels_old-labels))~=0)

    % reposition means
    for i=1:1:k
        centers(i,:)=mean(x(find(labels==i),:));
    end

    % measure distance from each point to each center
    for i=1:1:k
        d(i,:)=sqrt(sum(( repmat(centers(i,:),size(x,1),1)-x).^2,2));
    end

    % assign each point the label of its closest center
    labels_old=labels;
    [Y,labels]=min(d);
end

```

