# Linear regression

|  |  |  |
| --- | --- | --- |
| Hypothesis |  | |
| Cost function |  | |
| Training examples |  | |
| Features of each training example |  | |
| Parameters |  | |
| **Feature scaling and mean normalization:**  Speed up gradient descent by having each of input values in roughly the same range. | | |
| * Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. * Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero.     where  is the average of all the values for feature (i),  is the rage of values(max-min), or the standard deviation. | | |
| **Gradient descent:**  Gradient descent can converge to a local minimum, even with the learning rage  fixed. | | |
| simultaneously update    where  , is learning rate.  If  is too small, gradient descent can be slow.  If  is too large, gradient descent can overshoot minimum, it may fail to converge, or even diverge. | | |
| **Normal equation** | | |
| where  , n+1 by 1 vector  , m by n+1 matrix,  , m by 1 vector  If there’re redundant features (linearly dependent) or to many features (e.g. m≤n), then may be non-invertible. | | |
| **Gradient descent** | | **Normal Equation** |
| need to choose | | no need to choose |
| need many iterations | | don’t need to iterate |
| O(kn2) | | O(n3), need to compute |
| works well even when n is large | | slow if n is very large |
|  | | |
|  | | |

## Vectorial implementation

|  |
| --- |
| Hypothesis:  Cost function:  Gradient descent:    where  where  , n+1 by 1 vector  , m by 1 vector |

# Logistic regression

|  |  |
| --- | --- |
| Hypothesis | , sigmoid function or logistic function    where      , estimated the probability that y=1 on input x (given x, parameterized by θ). |
| Cost function | where      when y=1, if h→1, then cost→0, if h→0, then cost→∞  when y=0, if h→1, then cost→∞, if h→0, then cost→0 |
| Training examples | , note that |
| Features of each training example |  |
| Parameters |  |
| **Gradient descent** | |
| simultaneously update    where  , is learning rate.  note that | |

# Overfitting and regularization

|  |
| --- |
| **Overfitting:**  If there are too many features, the learned hypothesis may fit the training set very well, but fail to generalize to new examples (prediction on new examples). |
| Addressing overfitting options:  1. reduce number of features  -- manually select which features to keep  -- model selection algorithm  2. regularization  -- keep all the features, but reduce magnitude/values of parameters  -- works well when we have a lot of features, each of which contributes a bit to predicting y |
| **Regularization:**  is the regularization term, note j starts from 1  λ is the regularization parameter and , it controls the trade of between the goal of fitting the training set well and the goal of keeping the parameters small, therefore keeping the hypothesis relatively simple to avoid overfitting. |
| **for linear regression problems:**     * gradient descent:        * normal equation:     where  , n+1 by n+1 matrix  If m < n,  is non-invertible, when add the term ,  becomes invertible. |
| **for logistic regression problems:**    where       * gradient descent: |

# Neural network

|  |  |
| --- | --- |
| training set |  |
| “activation” of unit I in layer j |  |
| matrix of weights controlling function mapping from layer j to layer j+1  **If network has sj units in layer j and sj+1 units in layer j+1, then  will be of dimension sj+1 × (sj+1)** |  |
| weight controlling function mapping from ith unit in layer l-1 to jth unit in layer l |  |
| inputs of layer j |  |
| outputs of layer j | where g can be applied element-wise to z(j) |
| **Cost function**  (suppose there are K units in the output layer, which means the network is a classifier of K classes) | |
| where  is the ith output  L is the total number of layers in the network  sl is the number of units(not counting bias unit) in layer l  K is the number of output units/classes | |
| **Back propagation algorithm** | |
| * set * for i=1 to m   1. set  2. perform forward propagation to compute  3. compute  4. compute  where  g is the activation function, g’=g(1-g)  is the input of layer l,  5. compute  or   * compute the partial derivative terms:     where | |
| **Implementation Notes** | |
| * unrolling parameters * gradient checking * random initialization (symmetry breaking), initialize  to a random value in * pick a network architecture   number of input units: dimension of features  number of output units: number of classes  reasonable default: 1 hidden layer, or if >1 hidden layer, have same number of hidden units in every layer (usually the more the better) | |
| **Training a neural network** | |
| 1. randomly initialize weights  2. implement forward propagation to get  for any  3. implement code to compute cost function  4. implement back propagation to compute partial derivatives  for i=1:m {  perform forward propagation and back propagation using example ,  get activations  and  for l=2,…,L,  compute  }  compute , where  5. use gradient checking to compare  computed using back propagation vs. using numerical estimate of gradient of . Then disable gradient checking code.  6. use gradient descent or advanced optimization method with back propagation to try to minimize  as a function of parameters | |

## Vectorial implementation (a three layers network example)

|  |
| --- |
| * **Neural network architecture**   3 layer network, input layer has n units, hidden layer has h units, output layer has k units     * **Training examples (design matrix)**     (m training examples, each example has n features)   * **Desired outputs**     (k classes, if  belongs to the 5th class, then the corresponding  should by a K dimensional vector with )   * **Weights matrix and random initialization**   ,  is the weights matrix between input layer and hidden layer  is the weights matrix between hidden layer and output layer  initialize each  to a random value in   * **Inputs and Outputs of the hidden layer**     is the input value of jth unit, given ith training example    is the output value of jth unit, given ith training example, computed by the sigmoid function implementing element-wise on   * **Inputs and outputs of the output layer**   add one column (ones) to , make    is the input value of jth unit, given ith training example    is the output value (prediction value) of jth unit, given ith training example, computed by the sigmoid function implementing element-wise on   * **Cost function**              * **Back propagation algorithm (with regularization term)**     is the error of jth unit of the output layer, given ith training example    where (remove the first column of )  is the error of jth unit of the hidden layer, given ith training example  where  ,  , |

# Advice for applying machine learning

## Evaluating a learning algorithm

|  |
| --- |
| **Evaluating a hypothesis** |
| Given a dataset of training examples, we can split up the data into two sets: a **training set** and a **test set**. Typically, the training set consists of 70% of data and the test set is the remaining 30%.   * learn  and minimize  using the training set * compute the test error  1. for linear regression, 2. for classification,   where  is the **misclassification error** |
| **Model selection and Train/validation/test sets** |
| Given many models with different degrees, we can use a systematic approach to identify the ‘best function’. In order to choose the model of our hypothesis, we can test each degree of polynomial and look at the error result. One way to break down our dataset into the three sets is:   * Training set: 60% * Cross validation set: 20% * Test set: 20%   We can now calculate three separate error values for the three different sets using the following method:   * Optimize the parameters in  using the training set for each polynomial degree * Find the polynomial degree d with the least error using the cross validation set * Estimate the generalization error using the test set with , (d=theta from polynomial with lower error)   This way, the degree of the polynomial d has not been trained using the test set. |

## Bias vs. variance

|  |
| --- |
|  |
| **Diagnosing bias vs variance** |
| The relationship between the degree of the polynomial d and the underfitting or overfitting of our hypothesis:   * High bias (underfitting):  and  will be high, and * High variance (overfitting):  will be low and |
| **Regularization and Bias/variance** |
| The relationship of  to the training set and the variance set is:   * Low :  is low and  is high (high variance/overfitting) * Intermediate :  and  are low, and * Large :  and  will be high (high bias/underfitting)     In order to choose the model and the regularization term , we need to:   * Create a list of lambdas * Create a set of models with different degrees or any other variants * Iterate through the lambdas and for each lambda go through all the models to learn * Compute the cross validation error using the learned  (with ) on the * Select the best combo that produces the lowest error on the cross validation set * Using the best  and , compute  to see if it has a good generalization of the problem |
| **Learning curves** |
| **Experiencing high bias: if a learning algorithm is suffering high bias, getting more training data will not help much.**   * Low training set size:  to be low and  to be high * Large training set size:  and  to be high with     **Experiencing high variance: if a learning algorithm is suffering high variance, getting more training data is likely to help.**   * Low training set size:  will be low and  will be high * Large training set size:  increases and  continuous decreases without leveling off. Also,  but the difference remains significant |
| **Deciding what to do next revisited** |
| * Getting more training examples: fixes high variance * Trying smaller sets of features: fixes high variance * Adding features: fixes high bias * Adding polynomial features: fixes high bias * Decreasing lambda: fixes high bias * Increasing lambda: fixes high variance |

## Machine learning system design

|  |
| --- |
| **Error analysis** |
| The recommended approach to solving machine learning problems is to:   * start with a simple algorithm, implement it quickly, and test it on cross validation data * plot learning curve to decide if more data, more features, etc. are likely to help * manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors are made   It is important to get error results as a single, numerical value, otherwise it is difficult to assess the performance of algorithm. |
| **Error metric** |
| For binary classification problems:   * , the higher the better * , the higher the better |
| **Trade off precision and recall** |
| , high F score means high recall and high precision  F score can be used to choose the threshold of a classifier. |
| **Large data rationale** |
| Assume feature x has sufficient information to predict y accurately.  Useful test:   * Given the input x, can a human expert confidently predict y? * Can we actually get a large training set and train the learning algorithm of a lot of parameters in the training set? |

# Support vector machine

## Optimization objective

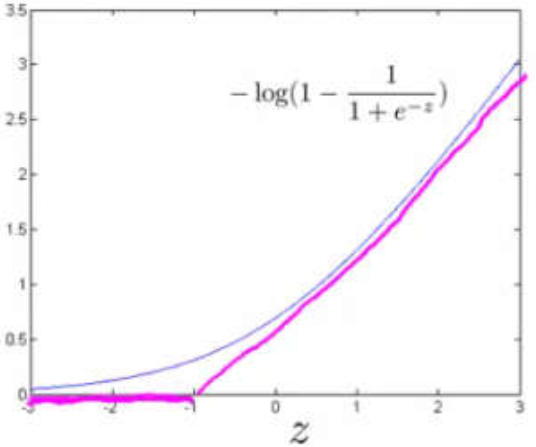
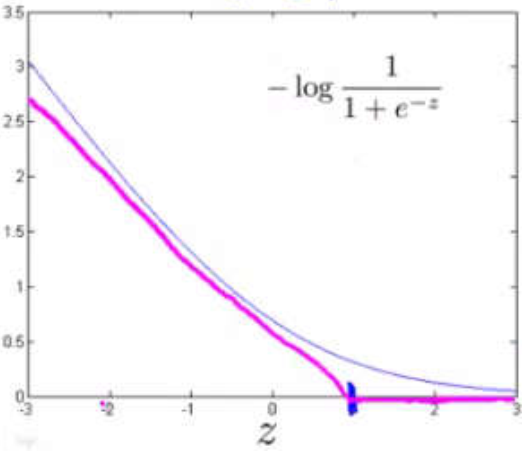
In logistic regression, we use the following rules:



The cost function (regularized) logistic regression:



* Modify the term  so that when , it outputs 0, and when , we use a straight decreasing line instead of the sigmoid curve. The new function is .
* Modify the term  so that when , it outputs 0, and when , we use a straight increasing line instead of the sigmoid curve. The new function is .



Thus, the cost function can be transformed to:



If we wish to regularize more, we decrease C, and if we wish to regularize less, we increase C.

Note that the hypothesis of the support vector machine is:



## Large margin Classification



Given the cost function of the SVM, we know that

* If y=1, we want 
* If y=0, we want 

Now, we set the constant  to a very large value, then we must choose  to ensure that, then minimization of the cost function is:

where  is the length of projection of the vector  onto the vector 

Note that , where  is the angel between  and .

Actually,  is the separating plane/decision boundary that we want to separate the negative and positive examples, and  is the normal vector of the separating plane.  is the angel between  and the normal vector of the separating plane, then  is the angle between  and the separating plane.

Given , in order to ensure the optimization objective to hold true, we need the absolute value of  to be as large as possible, which means the angel  needs to be as small as possible, and  needs to be as small as possible. As a result, the margin (distance of the decision boundary to the nearest example) needs to be as large as possible.

Therefore, the SVM will separate the negative and positive examples by a large margin. If , our decision boundary will intersect (0,0). If , the SVM will still find a large margin for the decision boundary. Note that the large margin is only achieved when C is very large, and if we have outlier examples that we don’t want to affect the decision boundary, we can reduce C (increasing and decreasing C is similar to respectively decreasing and increasing ).

## Gaussian Kernel

Kernels allow us to make complex and non-linear classifiers using SVM.

Given x, compute new feature depending on proximity to landmarks :

, where the similarity function is **Gaussian Kernel**.

* If  is close to , then 
* If  is far from , then 

One way to get the landmarks is to put them in the exact same locations as all the training examples, this will give us m landmarks with one landmark per training example. Given example , we can compute  of all features for example  (we may set  to correspond with ):



Now, we can use the SVM minimization algorithm to get  but with  substituted in for :



Using kernels to generate  is not exclusive to SVM and may also be applied to logistic regression. However, because of computational optimizations on SVM, kernels combined with SVM is much faster than with other algorithms, so kernels are almost always found combined only with SVM.

## Choosing SVM parameters

Choosing :

* If  is large, then we get higher variance and lower bias
* If  is small, then we get lower variance and higher bias

Choosing  for the Gaussian Kernel function:

* If  is large, the features  vary more smoothly, causing higher bias and lower variance
* If  is small, the features  vary less smoothly, causing lower bias and higher variance

Given a landmark , and two training examples with different classifications, and , the output value of Gaussian Kernel function represents the distance between example  and landmark .

If  is large, then  and , which means the two examples are very close to the landmark, and so the predictions of our hypothesis for these two examples will be very similar, that’s not what we want. Thus, a large  will cause higher bias and lower variance

If  is small, then  and , which means the two examples are very far from the landmark, and so the predictions of our hypothesis for these two examples will have a huge difference, causing our hypothesis more easier to separate the two examples. Thus, a small  will cause lower bias and higher variance.

## Using SVM

There are lots of good SVM libraries already written. In practical application, you should use one of these libraries rather than rewrite the functions.

In practical application, the choices we do need to make are:

* Choice of parameter 
* Choice of kernel (similarity function)
* No kernel (“linear kernel”), it gives standard linear classifier
* Gaussian Kernel needs to choose 

**Note**: do perform feature scaling before using the Gaussian Kernel.

**Note**: not all similarity functions are valid kernels. They must satisfy “Mercer’s Theorem” which guarantees that the SVM package’s optimizations run correctly and do not diverge.

**Note**: we can train  and  using the training and cross-validation datasets.

|  |
| --- |
| Logistic regression vs. SVM |
| **If n is large (relative to m), then use logistic regression, or SVM without a kernel.** (We don’t have enough examples to need a complicated polynomial hypothesis) |
| **If n is small and m is intermediate, then use SVM with a Gaussian Kernel.** (We have enough examples that we may need a complex non-linear hypothesis) |
| **If n is small and m is large, then manually create more features, then use logistic regression or SVM without a kernel.** (We want to increase our features so that logistic regression becomes applicable) |
| **Note**: neural network is likely to work well for any of these situations, but may be slower to train. |

# K-means (clustering algorithm)

## Introduction

|  |
| --- |
| Input:   * , the number of clusters * , the training set (unlabeled data) , where |
| K-means algorithm:   * Randomly initialize  cluster centroids, * Repeat {   **cluster assignment**: for i=1 to m, c(i):=index (from 1 to K) of cluster centroid closest to x(i)  **move centroid**: for k=1 to K, μ(i):=average of points assigned to cluster k  } |
| Note:   * If we have a cluster centroid with 0 points assigned to it, we can randomly re-initialize that centroid to a new point. We can also simply eliminate that cluster group. * After a number of iterations the algorithm will converge, where new iterations do not affect the clusters. * Some datasets have no real inner separation or natural structure. K-means can still evenly segment our data into K subsets. |

## Optimization objective



* In the cluster assignment step, our goal is to minimize J with  ( fixed)
* In the move centroid step, our goal is to minimize J with 

**Note: it is not possible for the cost function to sometimes increase, it should always descend.**

## Random initialization

Have K<m to make sure the number of clusters is less than the number of training examples.

Randomly pick K training examples, but be sure the selected examples are unique.

Set  equal to these K examples.

Note that K-means can stuck in local optima. To decrease the chance of this happening, we can run the algorithm on many different random initializations. In cases where K<10 it is strongly recommended to run a loop of random initializations.

|  |
| --- |
| for i=1 to 100,  randomly initialize K-means  run K-means to get c and μ  compute the cost function J(c, μ)  pick the clustering that gave us the lowest cost |

## Choosing the number of clusters

The elbow method: plot the cost J and the number of clusters K. The cost function should reduce as we increase the number of clusters, and then flatten out. Choose K at the point where the cost function starts to flatten out.

However, fairly often, the curve is very gradual, so there’s no clear elbow.

**Note: J will always decrease as K is increased. The one exception is if K-means gets stuck at a bad local optima.**

Another way to choose K is to observe how well K-means performs on a downstream purpose. In other words, we choose K that proves to be most useful for some goal we’re trying to achieve from using these clusters.

# Principal Component Analysis (PCA)

## PCA formulation

|  |
| --- |
| **Problem formulation**  Given two features, x1 and x2, we want to find a single line that effectively describes both features at once. We then map our old features onto this new line to get a new single feature. The same can be done with three features, where we map them to a plane.  The goal of PCA is to reduce the average of all the distances of every feature to the projection line. This is the projection error.  Reduce from n-dimension to k-dimension: find k vectors u(1),u(2),…,u(k) onto which to project the data so as to minimize the projection error. |
| **PCA is not linear regression**  In linear regression, we are minimizing the squared error from every point to our predict line. These are vertical distances.  In PCA, we are minimizing the shortest distance, or shortest orthogonal distances, to our data points.  In linear regression, we are taking our examples in x and applying the parameters in Θ to predict y.  In PCA, we are taking a number of features x1,x2,…,xn, and finding a closest common dataset among them. We aren’t trying to predict any result and we aren’t applying any weights to the features. |

## PCA algorithm

|  |
| --- |
| **Data reprocessing:**   * given training set, x(1),x(2),…,x(m) * compute * replace each  with * if different features on different scales, scale features to have comparable range of values   Above, we first subtract the mean of each feature from the original feature.  Then, we scale all the features . |
| **Compute “covariance matrix”:**  , the vectorial implementation is  where x(i) is a n by 1 vetor, X is a m by n vector, ∑ is a n by n vector |
| **Compute “eigenvectors” of covariance matrix :**  Octave command: [U,S,V]=svd(Sigma), where svd() is the ‘singular value decomposition’, a built-in function.  Note that U is a n by n matrix. |
| **Take the first k columns of the U matrix and compute z :**  Use Octave command Ureduce=U[:,1:k] to assign the first k columns of U to Ureduce, a n by k matrix.  Then, we compute z with:  , the vectorial implementation is  where z(i) is a k by 1 vector, Z is a m by k matrix, X is a m by n matrix, Ureduce is a n by k matrix |

## Reconstruction from compressed representation

, the vectorial implementation is 

where x(i) is a n by 1 vector, z(i) is a k by 1 vector, Ureduce is a n by k matrix, X is a m by k matrix.

## Choosing the number of principal components

|  |
| --- |
| One way to choose k is by using the following formula:   * given the average squared projection error: * also given the total variation in the data: * choose k to be the smallest value such that:   In other words, the squared projection error divided by the total variation should be less that 1%, so that 99% of the variance is retained. |
| **Algorithm for choosing k:**   * try PCA with k=1,2,… * compute Ureduce, z, x * check the formula given above that 99% of the variance is retained. If not, go to step one and increase k |
| **In Octave:**  Use the built-in function svd() to get matrix S: [U,S,V]=svd(Sigma), where Sigma is the covariance matrix. Then, we can check for 99% of retained variance using the S matrix as follows: |

## Advice for applying PCA

The most common use of PCA is to speed up supervised learning.

Given a training set with a large number of features, we can use PCA to reduce the number of features in each example of the training set.

**Note that we should define the PCA reduction from x(i) to z(i) only on the training set and not on the cross-validation or test sets. We can apply the mapping z(i) to the cross-validation and test sets after it is defined on the training set.**

|  |
| --- |
| **Appropriate use of PCA:**  compressions, reduce space of data, speed up algorithm, visualization of data (choose k=2 or 3) |
| **Bad use of PCA:**  trying to prevent overfitting. We might think that reducing the features with PCA would be an effective way to address overfitting. It might work, but is not recommended because it does not consider the values of our results y. Using the regularization will be at least as effective. |
| **Note:**  Don’t assume we need to do PCA. Try the machine learning algorithm without PCA first. Then use PCA if we find that we really need it. |

# Anomaly detection

## Density estimation algorithm

Given a training dataset , where each example is a vector .

Choose features  that might be indicative of anomalous examples.

Fit parameters , where , .

Given a new example , compute :



If , then x is an anomalous example.

## Developing and evaluating an anomaly detection system

|  |
| --- |
| To evaluate our learning algorithm, we take some labeled data, categorized into anomalous and non-anomalous examples (y=0 if normal, y=1 if anomalous).  Among that data, take a large proportion of good, non-anomalous data for the training set on which to train p(x).  Then, take a smaller proportion of mixed anomalous and non-anomalous examples (usually have many more non-anomalous examples) for the cross-validation and test sets.  For instance, we can split the data 60/20/20 training/CV/test and then split the anomalous examples 50/50 between CV and test sets. |
| **Algorithm evaluation:**  Fit model p(x) on training set.  **On a cross-validation/test example x, predict:**   * if p(x)< ε, then y=1 (anomaly) * if p(x)≥ε, then y=0 (normal)   **Possible evaluation metrics:**   * true positive, false positive, true negative, false negative * compute precision and recall * compute F1 score   **Note that we use the cross-validation set to choose parameter ε.** |

## Anomaly detection vs. supervised learning

|  |
| --- |
| **Use anomaly detection when:**   * We have a very small number of anomalous examples (y=1) and a large number of normal examples (y=0) * We have many different types of anomalies and it is hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we’ve seen so far. |
| **Using supervised learning when:**   * We have a large number of both positive and negative examples. In other words, the training set is more evenly divided into classes. * We have enough positive examples for the algorithm to get a sense of what new positives examples look like. The future positive examples are likely similar to the ones in the training set. |

## Choosing what feature to use

The features will greatly affect how well our anomaly detection algorithm works.

We can check that our features are Gaussian by plotting a histogram of our data and checking for the bell-shaped curve. Some transforms we can try on an example feature x that does not have the bell-shaped curve are:

* log(x)
* log(x+1)
* log(x+c) for some constant
* x1/2
* x1/3

We can play with each of these to try and achieve the Gaussian shape in our data.

There is an error analysis procedure for anomaly detection that is very similar to the one in supervised learning.

Our goal is for p(x) to be large for normal examples and small for anomalous examples.

One common problem is when p(x) is similar for both types of examples. In this case, we need to examine the anomalous examples that are giving high probability in detail and try to figure our new features that will better distinguish the data.

In general, choose features that might take on unusually large or small values in the event of an anomaly.

## Multivariate gaussian distribution

The multivariate gaussian distribution is an extension of anomaly detection and may (or may not) catch more anomalies.

Instead of modeling p(x1),p(x2),…separately, we’ll model p(x) all in one go:

, where  and 

Note that  is covariance matrix of x.

The importance effect is that we can model oblong gaussian contours, allowing us to better fit data that might not fit into the normal circular contours.

**Varying ∑ changes the shape, width, and orientation of the contours. Changing μ will move the center of the distribution.**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

## Anomaly detection using the multivariate gaussian distribution

When doing anomaly detection with multivariate gaussian distribution, we compute μ and ∑ normally. We then compute p(x) using the new formula in the previous section and flag an anomaly if p(x)< ε.

* **The original model for p(x) corresponds to a multivariate gaussian where the contours of p(x; μ, ∑) are axis-aligne.**
* **The multivariate gaussian model can automatically can capture correlations between different features of x.**

However, the original model maintains some advantages: it is computationally cheaper (no matrix to invert, which is costly for large number of features) and it performs well even with small training set size (in multivariate gaussian model, it should be greater than the number of features for ∑ to be invertible).

# Application: recommender system

## Problem formulation

Suppose we are trying to recommend movies to customers.

* =number of users
* =number of movies
* =1 if user j has rated movie i
* =rating given by user j to movie i (defined only if =1)

## Content based recommendations

Suppose we have two features, x1 and x2 which represents how much romance or how much action a movie may have (on a scale of 0-1).

One approach is that we could do linear regression for every single user. For each user j, learn a parameter  (including ). Predict user j as rating movie i with , where  is the parameter vector for user j,  is the feature vector for movie i.

To learn , we compute the following:

, eliminate the constant 

**To get the parameters for all users, we compute the following: (given feature vectors for movies, solve parameter vectors for users, then use them to predict)**

, eliminate the constant 

## Collaborative filtering algorithm

It can be very difficult to find features such as “amount of romance” or “amount of action” in a movie. To figure this out, we can use feature finders.

We can let the users tell us how much they like the different genres, providing their parameter vector immediately for us.

To infer the features from given parameters, we use the squared error function with regularization over all the users: (given parameter vectors for users, solve feature vectors for movies)

, eliminate the constant 

(Note: we can randomly guess the values for theta to guess the features repeatedly)

**To speed things up, we can simultaneously minimize our features and our parameters:**

****

**(because the algorithm can learn itself, so the bias unit where x0=1 have been removed, therefore)**

* **Initialize  to small random values, this serves to break symmetry and ensures that the algorithm learns features  that are different from each other.**
* **minimize  using gradient descent or other advanced optimization algorithm: **
* **for a user with parameter theta and a movie with (learned) features x, predict a rating .**

## Vectorization: low rank matrix factorization

Given matrix X (each row containing features of a particular movie) and θ (each row containing the weights for those features for a given user), then **the full matrix Y of all predicted ratings of all movies by all users is: Y=XθT** (X is a m by ? matrix, θ is a n by ? matrix, ? is the number of features, m is the number of movies, n is the number of users)

**Predicting how similar two movies i and j are can be done using the distance between their respective features x. Specifically, we are looking for a small value of ||x(i)-x(j)||.**

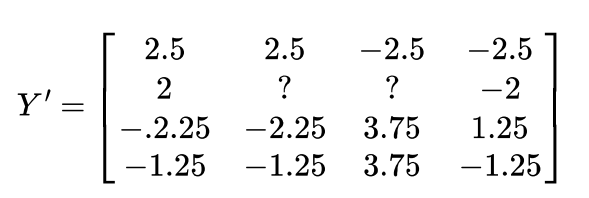
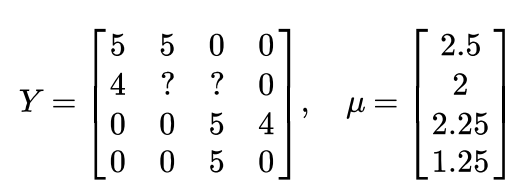
## Implementation detail: mean normalization

If the ranking system for movies is used, then new users (who have watched no movies), will be assigned new movies incorrectly. Specifically, they will be assigned θ with all components equal to zero due to the minimization of the regularization term. That is, we assume that the new user will rank all movies 0, which does not seem intuitively correct.

We rectify this problem by normalizing the data relative to the mean. First, we use a matrix Y to store the data from previous ratings, where the ith row of Y is the ratings for the ith movie, and the jth column corresponds to the ratings for the jth user.

We can define a vector: , such that  which is effectively the mean of previous of the previous ratings for the ith movie (where only movies that have been watched by users are counted).

**We can now normalize the data by subtracting μ, the mean rating, from the actual ratings for each user (column in Y).** As an example, given Y and μ, the resulting matrix Y’ is:



**We modify the prediction to include the mean normalization term:. For a new user, the initial predicted values will be equal to the μ term instead of simply being initialized to zero, which is more accurate.**

# Large scale machine learning

## Stochastic gradient descent

Stochastic gradient descent is an alternative to batch gradient descent and is more efficient and scalable to large datasets.

Stochastic gradient descent is written out in a different but similar way:



The only difference in the above function is the elimination of the m constant within 1/2.



The algorithm is as follows:

* randomly “shuffle” the dataset
* 

This algorithm will **only try to fit one training example at a time**. This way we can make progress in gradient descent without having to scan all m training examples first. **Stochastic gradient descent will be unlikely to converge at the global minimum and will instead wander around it randomly, but usually yields a result that is close enough. Stochastic gradient descent will usually take 1-10 passes through our dataset to get near the global minimum.**

## Mini-batch gradient descent

Mini-batch gradient descent can sometimes be even faster that stochastic gradient descent. Instead of using all m examples as in batch gradient descent, and instead of using only one example as in stochastic gradient descent, we will use some in-between number of examples b. Typical values for b range from 2-100 or so.

For instance, with b=10 and m=1000:

**Repeat{**

**for i=1,11,21,31,…,991{**

****

**}**

**}**

We are simply summing over ten examples at a time. The advantage of computing more than one example at a time is that we can use vectorized implementations over the b examples.

## Stochastic gradient descent convergence

How do we choose the learning rate α for stochastic gradient descent? Also, how do we debug stochastic gradient descent to make sure it is getting as close as possible to the global minimum?

One strategy is to plot the average cost of the hypothesis applied to every 1000 or so training examples. We can compute and save these costs during the gradient descent iterations.

**With a smaller learning rate, it is possible that we may get a slightly better solution with stochastic gradient descent.** That is because stochastic gradient descent will oscillate and jump around the global minimum, and it will make smaller random jumps with a smaller learning rate.

With a very small number of examples for the average, the line will be too noisy and it will be difficult to find the trend.

**One strategy for trying to actually converge at the global minimum is to slowly decrease α over time. For example, α=const1/(iterationNumber+const2).**

However, this is not often done because people don’t want to have to fiddle with even more parameters.

## Online learning

With a continuous stream of users to a website, we can run an endless loop that gets (x,y), where we collect some user actions for the features in x to predict some behavior y.

**We can update θ for each individual (x,y) pair as we collect them. This way, we can adapt to new pools of users, since we are continuously updating data.**

## Map reduce and data parallelism

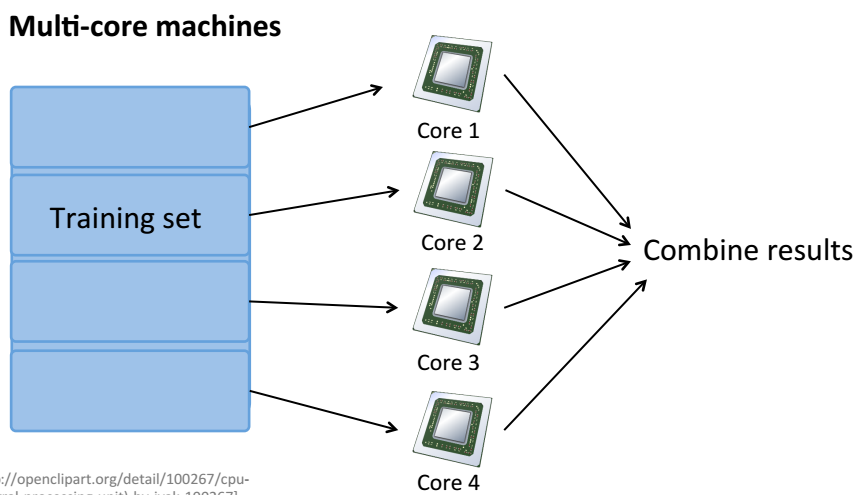
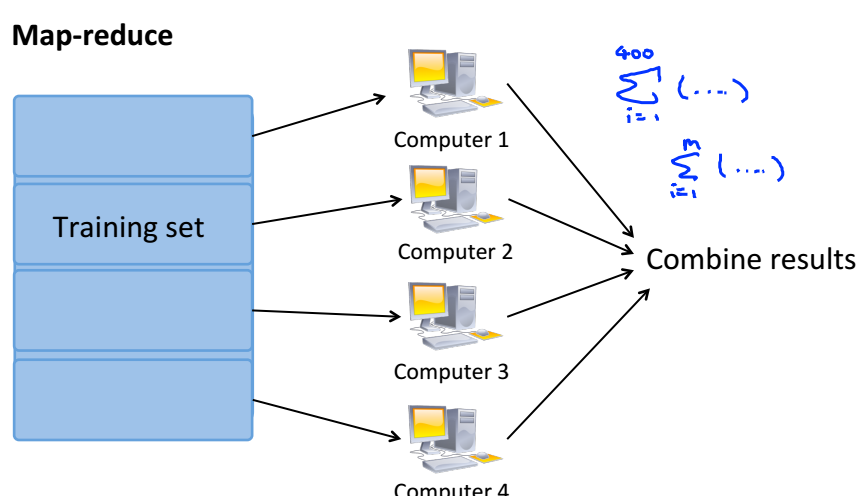
We can divide up batch gradient descent and dispatch the cost function for a subset of the data to many different machines so that we can train our algorithm in parallel.

We can split our training set into z subsets corresponding to the number of machines we have. On each of those machines calculate , where we’ve split the data starting at p and ending at q.

Map reduce will take all these dispatches jobs and “reduce” them by calculating:



This is simply taking computed cost from all the machines, calculating their average, multiplying by the learning rate, and updating theta.



**Our learning algorithm is MapReduceable if it can be expressed as computing sums of functions over the training set.** Linear regression and logistic regression are easily parallelizable.

For neural networks, we can compute forward propagation and back propagation on subsets of our data on many machines. Those machines can report their derivatives back to a “master” server that will combine them.