Machine Learning

Think you can help finish or improve this document?
Send me a request for Edit permissions!

Week 1
- I. Introduction
  - Types of machine learning
- II. Linear Regression with One Variable
  - Notation
  - Possible h ("hypothesis") functions
  - Cost function
  - Learning rate
  - Gradient descent for two-parameter linear regression (repeat until convergence)
  - "Batch" gradient descent
- III. Linear Algebra Revision (optional)

Week 2
- IV. Linear Regression with Multiple Variables
  - Large number of feature
  - Multi-parameter linear regression in vector notation
  - Picking features
  - Cost function for multiple features
  - Gradient descent for the cost function of multi-parameter linear regression
  - Feature scaling
  - Mean normalization
  - How to pick learning rate α
  - Polynomial regression
  - "Normal equation": Using matrix multiplication to solve for θ that gives min J(θ)
  - Gradient descent vs Solving for θ ("normal equation")
  - Using matrices to process multiple training cases at once
- V. Octave Tutorial

Week 3
- VI. Logistic Regression
  - Classification problems
  - Two-class (or binary-class) classification
  - Logistic regression
  - Decision boundary
  - Cost-function for logistic regression
  - Gradient descent for logistic regression
  - Advanced optimization algorithms and concepts
  - Multi-class classification
- VII. Regularization - the problem of overfitting
  - Underfitting vs. Overfitting

- Addressing overfitting
- Regularization
- How to pick λ (lambda)
- Linear gradient descent with regularization
- Normal equation with regularization
- Logistic gradient descent with regularization

Week 4

- VIII. Neural Networks: Representation
  - Neural networks
  - Neurons modelled as logistic units
  - Neural network
  - Notation
  - Calculating the hypothesis for a sample neural network
  - Vectorized forward propagation

Week 5

- IX. Neural Networks: Learning
  - Cost function for multi-class classification neural network
  - Forward propagation
  - Minimizing cost function for neural networks: back-propagation
  - Back-propagation for multiple training samples
  - Back-propagation intuition...
  - Use of advanced minimum cost optimization algorithms
  - Numerical gradient checking
  - Initial values of Θ
  - Network architecture
  - Steps in training a neural network

Week 6

- X. Advice for Applying Machine Learning
  - What to do when you get unacceptably large errors after learning
  - Machine learning diagnostic
  - Evaluating the hypothesis function
  - Calculating misclassification error rate
  - Cross-validation - evaluating alternative hypothesis function models
  - Distinguish high bias from high variance (underfitting vs. overfitting)
  - Choosing the regularization parameter λ
  - Too small training set? Learning curves
  - Selecting model for neural networks
- XI. Machine Learning System Design
  - Improving a machine learning system - what to prioritize
  - Recommended approach for building a new machine learning system
  - Error analysis
  - Error measure for skewed classes
  - Prediction metrics: precision and recall
  - Prediction metrics: average and F1 score

Week 7

- X. Support Vector Machines

Week 8
XIII. Clustering
Types of unsupervised learning
Notation
K-means clustering algorithm
K-means cost function (distortion function)
Practical considerations for K-means
XIV. Dimensionality Reduction
Data compression (data dimensionality reduction)
Principal component analysis (PCA)
How to choose k for PCA
Decompressing (reconstructing) PCR-compressed data
More about PCA
Bad use of PCA: to prevent overfitting
Recommendation on applying PCA
Week 9
XV. Anomaly Detection
Examples of anomaly detection
How anomaly detection works
Fraud detection
Gaussian (Normal) distribution
Density estimation
Anomaly detection algorithm
Training the anomaly detection algorithm
Evaluating the anomaly detection algorithm
Anomaly detection algorithm vs. supervised learning algorithm
Messaging features
Error analysis for anomaly detection
Multivariate Gaussian distribution
XVI. Recommender Systems
Week 10
XVII. Large Scale Machine Learning
XVIII. Application Example: Photo OCR
Useful resources

Week 1

I. Introduction

Types of machine learning

- Supervised learning (the "right answer" is provided as input, in the "training set")
  - Regression problem (expected output is real-value)
  - Classification problem (answer is a class, such as yes or no)
- Unsupervised learning

II. Linear Regression with One Variable

Notation

m: Number of training examples
x's: "input" variables (features)
y's: "output" (targets) variables
(x, y): one training example
$(x^{(i)}, y^{(i)})$: i^{th} training example
h(): function (t) found by the learning algorithm
θ: (theta) the "parameters" used in h() together with the features x
$h_\theta()$: not just generally the function h(), but specifically parameterized with θ
J(θ): the cost function of $h_\theta()$
n: number of features (inputs)
$x^{(i)}$: inputs (features) of the i^{th} training example
$x_j^{(i)}$: the value of feature j in the i^{th} training example
λ: (lambda) regularization parameter
:= means assignment in algorithm, rather than mathematical equality

Possible h ("hypothesis") functions

- Linear regression with one variable (a.k.a. univariate linear regression):
  - $h_\theta(x) = \theta_0 + \theta_1 x$ Shorthand: h(x) Where θ_0 and θ_1 are "parameters"
- Linear regression with multiple variables (a.k.a. multivariate linear regression):
  - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \ldots \theta_n x_n$ (for n features)
- Polynomial regression
  - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 (x_1)^2 + \theta_3 (x_1)^3$ (e.g. by just making up new features that are the square and cube of an existing feature)

Cost function

The cost function J() evaluates how close h(x) match y given the parameters finding the parameters θ used by h(). For linear regression (here with *one* feature x)
pick θ_0 and θ_1 so that $h_\theta(x)$ is close to y for our training examples (x,y)
i.e. minimize the "**sum of square errors**" cost function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

By taking the *square* of the error (i.e. $h_\theta(x) - y$), we avoid having too small results from h_θ cancelling out too large results (as $-1^2 == 1$), thus yielding a truer "cost" of the errors.

N.B. $\frac{1}{2m}$ "makes some of the math easier" (see explanation why).

"Squared error" cost function: a reasonable choice for cost function. The most common one for regression problems.

Gradient descent

Iterative algorithm for finding a local minimum for the cost function. Works for linear regression with any number of parameters, but also other kinds of "hypotheses" functions. Scales better for cases with large number of features than *solving* for the optimal min J()

$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ (for j=0 and j=1, repeat until convergence)

where

θ_j is the "parameter" in h_θ that is used for feature j

α is the learning rate

$\frac{\partial}{\partial \theta_j}$ is the partial derivative (slope) at the current point θ_j

$J(\theta_0, \theta_1)$ is the cost function (in this case with two parameters, for cases with only one feature)

N.B. update θ_0 and θ_1 *simultaneously! (i.e. as one atomic operation)*

Learning rate

The size α of each step when iterating to find a solution. N.B. no need to vary α between iterations. Gradient descent will naturally take smaller and smaller steps the closer we get to a solution.

Gradient descent for two-parameter linear regression (repeat until convergence)

for j = 0 and j = 1

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} \left(\frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \right)$$

simplifies to

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

"Batch" gradient descent

Just means each iteration of the gradient is applied to all the training examples (in a "batch").

III. Linear Algebra Revision (optional)

[...]

Week 2

IV. Linear Regression with Multiple Variables

Large number of feature

For problems involving many "features" (i.e.x_1,x_2,x_3 ... x_n) linear algebra vector notation is more efficient.

Multi-parameter linear regression in vector notation

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \ldots \theta_n x_n$$

for convenience of notation, and to allow use of vector multiplication, define a 0^{th} feature $x_0 = 1$, thus we can write

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \ldots \theta_n x_n$$

Now, defining a ((n+1) × 1) vector x containing all the features and a ((n+1) × 1) vector θ containing all the parameters for the hypothesis function h_θ, we can efficiently multiply the two (yielding a scalar result) if we first transpose (rotate) the θ into θ^T

$h_\theta(x) = \theta^T x$ in Octave: **theta' * x**

Picking features

Use your domain insights and intuitions to pick features. E.g. deriving a combined feature might help. There are also automatic algorithms for picking features.

Cost function for multiple features

For n+1 features (where $x_0 = 1$) the combined cost function over m training samples will be

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

which really means (note that i starts from 1 and j starts from 0)

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} ((\sum_{j=0}^{n} \theta_j x_j^{(i)}) - y^{(i)})^2$$

Gradient descent for the cost function of multi-parameter linear regression

with the number of features n >= 1 and $x_0 = 1$, one iteration j of the gradient descent is

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

in Octave: **theta = theta - alpha * (1/m) * sum((theta' * x - y) * x)**

thus (atomically updating θ_j for j = 0, ..., n)

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)}$$

...

$$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_n^{(i)}$$

Practical considerations for gradient descent

Feature scaling

Make sure the values in each group of properties x_n are on the same order (i.e. scale some groups if necessary), or the gradient descent will take a long time to converge (because the contour plot is too elongated). Typically, make all groups contain values $-1 \leq x_i \leq 1$

Mean normalization

To make sure the values for feature x_i range between -1 and +1, replace x_i with $x_i - \mu_i$ where μ_i is the mean of all values in the training set for feature x_i *(excluding x_0 as it is always 1)*

So together, $x_i := \frac{x_i - \mu_i}{s_i}$ where μ_i is the mean of all values for the property in the training set and s_i is the range of values (i.e. max(x) - min(x)) for the property i.

How to pick learning rate α

The number of iterations before linear descent converges can vary a lot (anything between 30 and 3 million is normal).

To make sure the linear descent works, plot the running-minimum of the cost function J(θ) and make sure it decrease for each iteration.

Automatic convergence test; e.g. declare convergence if J(θ) change by less than 10^{-3} in one iteration. However looking at the plot is usually better.

If J(θ) is increasing rather than decreasing (or oscillating) then the usual reason is that α is too big.

Too small α will result is too slow change in J(θ).

Trying to determine α heuristically, try steps of ≈ x3, so 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, ...

Polynomial regression

If a simple straight line does not fit the training data well, then polynomial regression can be used.

Just define some new feature that are the square and cube of an existing feature.
E.g. $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 (x_1)^2 + \theta_3 (x_1)^3$

Note that feature scaling is extra important when using squared and cubed features.

In addition to squaring and cubing a features, also consider adding the root of the feature.

"Normal equation": Using matrix multiplication to solve for θ that gives min J(θ)
For n features and m training examples, create a (m × n+1) matrix X (the "design matrix") from the features x in the training set, where the first column is for feature x_0 that is always 1, and a (m × 1) vector y containing the y values of the training set. Then this will give the parameters θ that minimize the cost function J for h_θ:

$$\theta = (X^T X)^{-1} X^T y$$ in Octave: **pinv(X' * X) * X' * y**

N.B. unlike when using gradient descent for finding θ that gives min J(θ), feature scaling is not necessary when solving for θ as shown above.

Gradient descent vs Solving for θ ("normal equation")
Gradient descent
Pros: 1) Works well even for large number of features,
Cons: 1) Need to choose learning rate, 2) Needs many iterations (may be slow)
Solving for θ
Pros: 1) No need to choose learning rate, 2) No need to iterate
Cons: 1) Need to compute inverse of n×n matrix, which is done in $O(n^3)$ time, so slow for very large number of features (> 1,000-10,000 features), 2) X^TX might be non-invertible (rarely)

Using matrices to process multiple training cases at once
By replacing the vector-based operations in the formulas mentioned above with matrix-based operations, all the training cases can be considered at once (instead of iterating over the set one case at the time). This allows making use of Octave's much faster low-level matrix multiplication implementation.

To calculate h for multiple training cases, make X a (m × n+1) matrix populated with the *transposed* feature vectors $x^{(i)}$ one per row, and make θ a (n+1 × 1) vector (as before). The result h_θ shall be a (m × 1) vector. Now, because of the non-commutative nature of matrix multiplication, the $h_\theta(x) = \theta^T x$ formula *won't work for X as is*. To yield a (m × 1) vector we need to multiply the (m × n+1) matrix X with the (n+1 × 1) vector θ in that order. I.e. the matrix version of the $h_\theta()$ functions is

$$h\theta(X) = X\theta$$ in Octave: **X * theta**

To calculate J(θ) for multiple training cases, use the same version of h above. Note that to take the power of $(h_\theta - y)$ when h_θ is a matrix we need expand it and transpose one of the terms, i.e. $(X\theta - y)^2 \Rightarrow (X\theta - y)^T(X\theta - y)$, and when using matrixes to hold multiple training cases the summation goes away!

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T (X\theta - y)$$

in Octave:
J = (1/(2*m)) * (X*theta - y)' * (X*theta - y)

To perform one iteration in gradient descent that considers all training cases at once,

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

needs to be changed into

$$\theta_j := \theta_j - \alpha \frac{1}{m} X^T (h_\theta(X) - y)$$

Note that with a matrix holding all features for all training sets, again the need to sum goes away!

in Octave:

d = (1/m) * X' * (X*theta - y); % sum i = 0 to m goes away!
theta = theta - alpha * d

V. Octave Tutorial

Comments
%

Assignment
a = 4
a = 4; % Suppress the output

Display
Print the value of a: **disp(a)**
Format string: **sprintf('6 decimals: %0.6f', a)**
Type formatting of subsequent outputs: **format short** or **format long**
Histogram: **hist(x, 50)** % matrix x in 50 bins
List variables in current scope: **who**
Detailed list of variables in current scope: **whos**

Plot
Plotting the graph for a function y1: **t = [0:0.01:0.98]; y1 = sin(2*pi*4*t); plot(t, y1);**
Incrementally add to the current graph: **hold on**
Label the axes: **xlabel('time'); ylabel('value')**
Add a legend: **legend('sin', 'cos')**
Save plot to file: **print -dpng 'myPlot.png'**
Close the figure window: **close**
Switch between figures: **figure(1); figure(2)**
Divide plot into 1x2 grid, access first element: **subplot(1,2,1)**
Change scale of axes: **axis([0.5 1 -1 1])**

Matrices and vectors
1 x 3 matrix: **[1 2 3]**
3 x 3 matrix: **[1 2 3; 4 5 6; 7 8 9]**
3 x 1 matrix: **[1; 2; 3]**
row matrix with values from 1 to 2 in increments of 0.1: **[1:0.1:2]**

2 x 3 matrix of all 0: **zeros(2,3)**

2 x 3 matrix of all 1: **ones(2,3)**
2 x 3 matrix of all 2: **ones(2,3) * 2**
3 x 3 matrix of all 1: **ones(3)**

Identity matrices
3 x 3 identity matrix: **eye(3)**

Random
3 x 4 matrix with all random numbers (between 0 and 1): **randn(3, 4)**

Magic squares
n x n matrix where sums of all columns, rows, diagonals are the same: **magic(5)**

Transpose
transpose of X = [1 2 3]: **X'** results in [1; 2; 3]

Flip Up-Down
flipud(A)

Size
Size of 2 x 3 matrix A: **size(A)** results in a vector [2 3]
Number of rows in 2 x 3 matrix A: **size(A, 1)** results in 2
Number of columns in 2 x 3 matrix A: **size(A, 2)** results in 3
The longest dimension of 2 x 3 matrix A: **length(A)** results in 3

Importing data
Display current path: **pwd**
Change path: **cd**
Load data from file feature.dat: **load('features.dat')**
Clear variable A: **clear A**
Assign items 10 through 20 of vector v to vector a: **a = v(10:20)**
Save variable v to binary format file: **save hello.mat v**
Loading variable v back from file: **load hello.mat** % automatically assigned to v!
Save to text file: **save hello.txt v -ascii**

Indexing and assignment
Get element $A_{2,3}$: **A(2,3)**
Get 2_{nd} row vector: **A(2,:)**
Get 2_{nd} column vector: **A(:,2)**
Get 1_{st} and 3_{rd} row: **A([1 3], :)**
Assign vector to 2_{nd} column: **A(:,2) = [10; 11; 12]**
Append another column vector to a matrix: **A = [A, [1; 2; 3]]**
Put all elements in matrix A into a single column: **A(:)**
Concatenate matrix B to the right of A into C: **C = [A B]**
Concatenate matrix B below A into C: **C = [A; B]**

Mathematical operations
Multiply matrix A with matrix B: **A * B**
Multiply matrix A *element-wise* with matrix B: **A .* B**
Square every element in matrix A: **A .^ 2**
Element-wise inverse of A: **1 ./ A**
Absolute values of A: **abs(A)**
Max value of elements in v: **max(v)**
Check which elements pass the condition: **v < 3** results in e.g. [1 0 1 1]
Return the values of the elements that pass the condition: **find(v < 3)**
Indices of all elements that pass the condition: **[i, j] = find(A < 3)**
Sum of all elements of A: **sum(A)**
Per-column sum of all elements of A: **sum(A,1)**
Per-row sum of all elements of A: **sum(A,2)**
Product of all elements of A: **prod(A)**
Round elements of A: **round(A)**
Round elements of A down: **floor(A)**
Round elements of A up: **ceil(A)**
Per-column maximum of matrix A: **max(A, [], 1)**
Per-row maximum of matrix A: **max(A, [], 2)**
Max of all the values in A: **max(max(A))** or **max(A(:))**

Flow control
For-loop from 1 to 10: **for i = 1:10, v(i) = 2^i; end;**
While-loop: **while i <= 5, v(i) = 100; i = i + 1; end;**
If-else statement: **if something==true, do(); elseif other ~= false, do2(); else do3(); end;**

Functions
Create a file with the extension .m, using the name of the function as the name of the file. State the function signature (including named return value!) on the first line in the file (prefixed by “function”), then leave a blank like, and implement the function below it.

Call **addpath(**...**)** with the path to the directory containing your function files.

Week 3

VI. Logistic Regression

Classification problems
These are problems to which the answer is one of a number of classes, e.g

- Online transactions: Fraudulent / Legit
- Tumor: Malignant / Benign
- Email classification: Work / Friends / Family / Hobby groups

Two-class (or binary-class) classification

Formally $y \in \{0, 1\}$ where 0 is the "Negative class", and 1 is the "Positive class"

The *linear regression* technique as used earlier might work reasonably well for some training sets, but suffers badly in many cases (e.g. when the "spread" of the groups are not equal). It may also output answers not even in the 0..1 range, which is at least inappropriate.

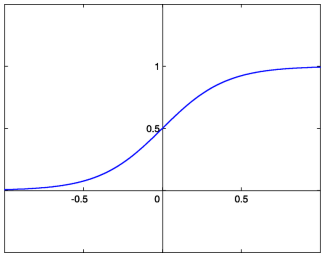
Logistic regression

This is a better method for classification than pure linear regression. Note that it is a classification technique, despite being called logistic *"regression"*, but also that it is a modification of linear regression (i.e. filtered by g(), see below) rather than a completely different technique.

For logistic regression, the hypothesis function is

$h_\theta(x) = g(\theta^T x)$ where $g(z) = \frac{1}{1 + e^{-z}}$ thus $h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$

and g() is called the "Sigmoid" or "logistic" function (two interchangeable names for the same thing).



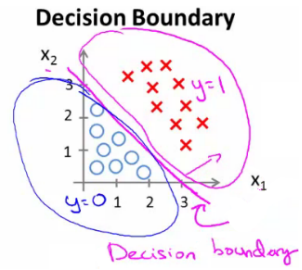
g(z) in Octave where z can be a real value, vector, or matrix:

```
g = zeros(size(z));
for i = 1:size(z, 2), g(:,i) = 1 ./ (1+exp(-z(:,i))); end;
```

Note that since y must be either 0 or 1, the i output of $h_\theta(x)$ should be interpreted as the probability that y is 1. Formally $h_\theta(x) = P(y = 1 \mid x; \theta)$ i.e. "the probability that y = 1, given x, parameterized by θ".

Decision boundary

Clamping the output of $h_\theta(x) = g(\theta^T x)$ hard to y, such that **y = 1 if h ≥ 0.5** and **y = 0 if h < 0.5** creates a *decision boundary*, such that drawn in a graph, all y = 0 should fall on one side of the boundary and all y = 1 should fall on the other. In terms of $\theta^T x$ (rather than $g(\theta^T x)$), **$y = 1$ if $\theta^T x \geq 0$**.



Note that the decision boundary is a property of the hypothesis function (provided its parameters), not of the training set (though the training set can be used to fit the parameters to achieve the correct boundary).

Non-linear decision boundaries

As for the polynomial regression seen earlier, adding additional higher-order parameters (such as $(x_1)^2$ and $(x_2)^2$) we can achieve non-linear and complex decision boundaries for logistic regression too.

Cost-function for logistic regression

The "squared error" cost function used for linear regression is not suitable for logistic regression, because the non-linear nature of g() will very likely produce a non-convex plot of the cost function, i.e. with many local minima.

A much better cost function for logistic regression is this

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost\left(h_\theta(x^{(i)}), y^{(i)}\right)$$

where **Cost(h_θ(x),y) = -log(h_θ(x)) if y = 1** *but* **-log(1 - h_θ(x)) if y = 0**
This is a convex function that has the desirable properties that its cost is zero when y = 1 and $h_\theta(x) = 1$, but as $h_\theta(x) \rightarrow 0$ the cost $\rightarrow \infty$. This means that if $h_\theta(x)$ is wrong, this cost function will penalize the learning algorithm by a very large amount.

Because y is always either 0 or 1 for two-class logistic regression, we can write the Cost function without the conditional statements, like this:

$$Cost(h\theta(x), y) = -y\ log(h\theta(x)) - (1 - y)\ log(1 - h\theta(x))$$

Note that one of the terms on either side of the central minus will always be zero, because y is either 1 or 0.

Thus in full (moving the minus out front):

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost\left(h_\theta(x^{(i)}), y^{(i)}\right) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\ log\left(h\theta\left(x^{(i)}\right)\right) + (1 - y^{(i)})\ log\left(1 - h\theta\left(x^{(i)}\right)\right)\right]$$

in Octave:
h = sigmoid(X*theta);
J = (1/m)*sum(-y'*log(h) - (1-y)'*log(1-h));

Gradient descent for logistic regression
After solving the partial derivative with respect to the new cost function, the formula for logistic gradient descent looks just like the one for linear gradient descent (minus the ½ which was only used before to simplify the derivative), i.e.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

(N.B. 1/m missing from lecture video in error)

but ***remember*** that the $h_\theta(x^{(i)})$ has changed from $h_\theta(x) = \theta^T x$ to $h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$

Apply and iterate the gradient descent algorithm for logistic regression the same way as for linear regression. Applying feature scaling (also in the same way) may help gradient descent run faster.

in Octave: **grad = (1./m) * X' * (sigmoid(X * theta) - y)**

Advanced optimization algorithms and concepts
Gradient is a fairly simple algorithm, but somewhat slow and for large problem sets more advanced and efficient algorithms can be used, such as Conjugate gradient, BFGS, and L-BFGS. These are much more complex, but also faster, and with no need to manually pick α. These "have a clever inner loop" called *line-search algorithm*. N.B. they can be used just fine without being fully understood; just use a library implementation (e.g. see fminunc()).

Multi-class classification
Classification of data into more-than-two groups (not be confused with multi-label

classification). Examples include classification of emails into Work / Friends / Family / Hobby groups.

There is no genuine multi-class classification algorithm. Instead, for n classes we can split the problem into n separate “one-vs-all” (or one-vs-rest) binary logistic regression problems, and afterwards assign the class for which the binary logistic regressions return the highest probability.

VII. Regularization - the problem of overfitting

Underfitting vs. Overfitting
With too few features we risk underfitting, or “high bias” (e.g. fitting a straight line through a second-degree curve), but with too many we risk overfitting, or “high variance” (e.g. fitting a high-order function perfectly to the training data but fail to be generic enough to fit any new data).

Addressing overfitting
Either the number for features can be manually reduced, or an automatic “model selection algorithm” can be used, or we can try regularization.

Regularization
Means keeping all features, but reducing the magnitude of the parameters θ, thus reducing each features’ contribution to the hypothesis function. Works well when we have a lot of features that each contribute a little to the prediction of y, but the weakness is that we don’t know which features actually correlate with y, so we just have to treat all the same.

To regularize e.g. the “square error” cost function we add a regularization term, with a regularization parameter λ (i.e. lambda), like this

$$J(\theta) = \frac{1}{2m}\left[\left(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2\right) + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

Lambda regulates the trade-off between the goal of fitting the data well and the goal of keeping the parameters small to avoid overfitting.

Note that the sum in the regularization term starts from 1. By convention we don’t regularize θ_0, but in practice it doesn’t make much difference if we do or not. See video VII.2 at 5:40.

Too big lambda will result in underfitting.

How to pick λ (lambda)
We’ll explore some automatic ways of picking lambda later, but for now just try out a couple of numbers manually. Lambda 0 means no regularization. Too large number will result in underfitting. Note that the number is fixed for the duration of running gradient descent. The $\left(1 - \alpha\frac{\lambda}{m}\right)$ term in gradient descent (see below) should be a little less than one, i.e. 0.99-0.95 or so.

Linear gradient descent with regularization
Working out the partial derivative with respect to the new cost function we get (note that θ_0 must be handled separately so not to penalize it too):
Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

}

Normal equation with regularization

$\theta = \left(X^T X + \lambda M\right)^{-1} X^T y$ where M is a (n+1 × n+1) identity matrix but with a zero at 0,0 and lambda is a scalar we pick manually (for now).

Logistic gradient descent with regularization
Adding the normalization term to the logistic cost function we get

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^{m} y^{(i)} \log\left(h\theta\left(x^{(i)}\right)\right) + (1 - y^{(i)}) \log\left(1 - h\theta\left(x^{(i)}\right)\right) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} {\theta_j}^2$$

in Octave:
h = sigmoid(X*theta):
regularizationTerm = (lambda/(2*m)) * sum(theta(2:end).^2);
J = (1/m)*sum(-y'*log(h) - (1-y)'*log(1-h)) + regularizationTerm;

and solving the partial derivative with respect to this, we get
Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right]$$

}

Week 4

VIII. Neural Networks: Representation

Neural networks
Artificial neural networks is an old technique inspired by the way the brain works. It fell out of favour in the late 90ies, but is now again considered the state of the art technique for many machine learning problems (mainly because hardware has gotten fast enough for large networks). It's main advantage is in learning complex non-linear hypothesis.

In problem sets with a very complex decision boundary, adding all the polynomial "features" required to express the correct hypothesis quickly grow very large. E.g. for a problem with

n=100 basic features, adding all the second-order polynomial features (i.e. all products of two features) n grows to about 5,000, and adding all the third-order polynomial features too (i.e. all combinations of three features) n grows to about 170,000. With a neural network we can do with only the basic features.

Another example is problems with very large inputs, like image or sound data, where the inputs may be a several thousand discrete values. Adding any extra polynomial features are out of the question (because it would result in many millions of features).

Neurons modelled as logistic units
A single neuron can be modelled as a "logistic unit", where as before the input x is a vector of values that get weighted by a vector θ, and the output is passed through the sigmoid function g(). As before, we add a x_0 feature that is always 1. This is called the "bias" input. It makes the x and θ vectors the same size (so they can be multiplied together), but it also allows the correct output when all the features are zero.

The output "hypothesis" of the artificial neuron is either 0 or 1, and for a logistic unit it is also known as the Sigmoid (logistic) activation function.

Neural network
A neural network is a group of logistic units, connected in layers. Layer one is called the "input layer" and the last layer the "output layer". Any layers in between are called "hidden layers" (numbered from 2 and up). Each logistic unit in a layer have the same inputs, which are all the outputs from the previous layer, plus the bias input. The network has one neuron in the output layer for each possible classification class. The more hidden layers there are, the more complex (i.e. non-linear) relationships the network can learn.

<u>Notation</u>
x_n: the n_{th} feature in layer one (the input layer), where n_0 is the bias input (that is always 1)
$a_i^{(j)}$: "activation" of unit i in layer j
$\Theta^{(j)}$: the weights controlling function mapping from layer j to layer j+1. Capital theta as this is a matrix.
$h_\Theta(x)$: the function yielding the output vector (i.e. our hypothesis function parameterized by the Θ matrix).
g(): the Sigmoid function yielding a value between 0 and 1. Round it to get the binary activation value.
L: total number of layers in the network
s_l: number of units in layer l (lowercase L)
s_L: number of units in the output layer
K: also the number of output units (the number of classification classes)

Calculating the hypothesis for a sample neural network
A neural network with three input features in layer 1, three logistic units in a hidden layer 2, and one logistic unit in the output layer 3, the activations are calculated like this (note that the activation of the only logistic unit in layer 3, i.e. $a_1^{(3)}$ is our "hypothesis" $h_\Theta(x)$):

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0) + \Theta_{11}^{(1)}x_1) + \Theta_{12}^{(1)}x_2) + \Theta_{13}^{(1)}x_3)\right)$$
$$a_2^{(2)} = g\left(\Theta_{20}^{(1)}x_0) + \Theta_{21}^{(1)}x_1) + \Theta_{22}^{(1)}x_2) + \Theta_{23}^{(1)}x_3)\right)$$
$$a_3^{(2)} = g\left(\Theta_{30}^{(1)}x_0) + \Theta_{31}^{(1)}x_1) + \Theta_{32}^{(1)}x_2) + \Theta_{33}^{(1)}x_3)\right)$$
$$h_\Theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(1)}a_0^{(2)}) + \Theta_{11}^{(1)}a_1^{(2)}) + \Theta_{12}^{(1)}a_2^{(2)}) + \Theta_{13}^{(1)}a_3^{(2)})\right)$$

Note that a_0 is the bias input, so it is always 1, and a_1 is the first logistic unit.

The $\Theta^{(1)}$ matrix (mapping from the input layer to the second layer) for the above network will be $\Theta \in \mathbb{R}^{3\times4}$, i.e. a 3×4 matrix. Defined generally, the dimensions will be (number of units in the next layer) × (number of units in this layer, plus one).

Vectorized forward propagation

...

Week 5

IX. Neural Networks: Learning

Cost function for multi-class classification neural network

The cost function we use for neural networks is a generalization of the regularized logistic regression cost function, where instead of just 1 output unit we have K number of them:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log\left(h_\Theta\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right) log\left(1 - h_\Theta\left(x^{(i)}\right)\right)_k\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2$$

where the output of J(Θ) is a K×1 vector, and $(J(\Theta))_i$ is the i^{th} output. Note that the regularization term sums over the (j,i) real values in the Θ matrix for the l network layer, but as before skipping the bias values, i.e. where i = 0.

Forward propagation

The prediction resulting from the neural network is computed by "forward propagation".
Example of forward propagation in a 4 layer neural network:

For layer 1 (input layer):	$a^{(1)} = x$	then	add $a_0^{(1)} = 1$		
For layer 2:	$z^{(2)} = \Theta^{(1)}a^{(1)}$	then	$a^{(2)} = g(z^{(2)})$	then	add $a_0^{(2)} = 1$
For layer 3:	$z^{(3)} = \Theta^{(2)}a^{(2)}$	then	$a^{(3)} = g(z^{(3)})$	then	add $a_0^{(3)} = 1$
For layer 4 (output layer):	$z^{(4)} = \Theta^{(3)}a^{(3)}$	then	$a^{(4)} = h_\Theta(x) = g(z^{(4)})$		

Minimizing cost function for neural networks: back-propagation

To perform gradient descent, for each unit (or node) j in layer l we need to compute the "error" $\delta_j^{(l)}$ in activation of that node.

Start with the last layer (e.g. layer 4) and compute the error in the activation of the nodes in

this (output) layer. I.e.
$\delta_j^{(4)} = a_j^{(4)} - y_j$ where $a_j^{(4)}$ is the activation we got by forward propagation, and y_j is the actual answer to sample j in the training set. Or vectorized $\delta^{(4)} = a^{(4)} - y$

Then proceed backwards (i.e. perform back-propagation) through the layers, calculating the errors for each layer, i.e. for layer 3 (note the element-wise matrix multiplication indicated by $.*$):
$\delta^{(3)} = \left(\Theta^{(3)}\right)^T \delta^{(4)} .* g'\left(z^{(3)}\right)$ where $g'\left(z^{(3)}\right)$ is the partial derivative: $a^{(3)} .* (1 - a^{(3)})$
and then for layer 2:
$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'\left(z^{(2)}\right)$ where $g'\left(z^{(2)}\right)$ is the partial derivative: $a^{(2)} .* (1 - a^{(2)})$
Note that the 1 in the partial derivative is an array of all 1's, and that there is no $\delta^{(1)}$ as this is the input layer and has no error associated with it. The partial derivative ignores the regularization, but this will be corrected for later.

Back-propagation for multiple training samples
Repeat for i = 1 to m {
$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
}

Same thing vectorized: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}\left(a^{(l)}\right)^T$

Then calculate
$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if j is = 0
$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if j is > 1

where $D_{ij}^{(l)} = \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Back-propagation intuition...
9-3 ...

No need to calculate error for bias units...

Use of advanced minimum cost optimization algorithms
For neural networks there is one theta matrix per layer so to use fminunc() we need to unroll the matrices into a single vector. Note that the size of each theta matrix depends on the number of activation units in its corresponding layer, so each matrix may be of a different size.

thetaVec = [theta1(:) ; theta2(:) ; theta3(:)]
DVec = [D1(:) ; D2(:) ; D3(:)]

To go the other way (assuming $s_1 = 10$, $s_2 = 10$, $s_3 = 1$):
Theta1 = reshape(thetaVec(1:110), 10, 11)
Theta1 = reshape(thetaVec(111:220), 10, 11)
Theta1 = reshape(thetaVec(221:321), 1, 11)

Numerical gradient checking
This can be used to check that the implementation of the forward- and back-propagation is correct. Specifically, it will numerically approximate the derivative of the cost function (i.e. the slope of the cost function at a point), so that we can check that the derivative yielded in e.g. the back-propagation implementation is roughly the same value.

The "two-sided difference" estimate formula is
$\frac{d}{d\theta}J(\theta) = \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$ where 2ϵ is the is the small distance (e.g. let ϵ be between 0.01 and 0.0001) between two points on the curve at the centerpoint of which we want to find the approximate derivative (the slope of the curve).

In Octave: **gradApprox = (J(theta - EPSILON) - J(theta - EPSILON)) / (2*EPSILON)**
where EPSILON = 0.0001

For multi-parameter theta vectors
for i = 1:n,
thetaPlus = theta;
thetaPlus(i) = thetaPlus(i) + EPSILON;
thetaMinus = theta;
thetaMinus(i) = thetaMinus(i) - EPSILON;
gradApprox(i) = (J(thetaPlus) - J(thetaMinus)) / (2*EPSILON);
end;

Then check that gradApprox is roughly the same value as DVec, but remember to turn off gradient checking once you've confirmed the correctness of the implementation, or performance will suffer greatly!

Initial values of Θ
For neural networks, initially Θ must not all be the same (e.g. all zeros) because if so, all the logistic units in one level will compute the same value. Instead use random initialization, to achieve "symmetry breaking".

Initialize theta to be a random number close to zero, between -EPSILON and EPSILON (N.B. this is not the same EPSILON as used in numerical gradient checking!)
E.g.
Theta1 = rand(10, 11) * (2*INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(1, 11) * (2*INIT_EPSILON) - INIT_EPSILON;

Network architecture
The number of input units, output units, hidden layers, and units per hidden layer is called the network's architecture.

- The number of input units is defined by the number of features in the problem.
- The number of output units is defined by the number of classification classes of the problem.
- Usually, one hidden layer is enough, but if there are more than one then they usually have the same number of activation units.
- The more activation units in a hidden layer the better, but too many may cause performance problems. Between 1 to 4 times as many units per hidden layer as there is features is reasonable.

Steps in training a neural network
Use a for loop in your first implementation
...

Our cost function used for neural networks is non-convex, so it is susceptible to local minimas, but practically this is usually not a problem.

Week 6

X. Advice for Applying Machine Learning

What to do when you get unacceptably large errors after learning
The options include:
Get more training examples - may help in cases of high variance
Try smaller set of features - may help in cases of high variance
Get additional features - may help in cases of high bias
Adding polynomial features - may help in cases of high bias
Decrease λ - may help in cases of high bias
Increase λ - may help in cases of high variance

Being able to tell whether a hypothesis function suffers from high bias or high variance is important, as collecting more training data is time consuming and may be a waste of time.

Machine learning diagnostic
A diagnostic test is a test that can be run to gain insight into what works or doesn't with a learning algorithm, and what the best way of fixing it might be. The test can take some time to implement, but it often saves time in the long run, by helping you avoid wasting time on things that make no difference.

Evaluating the hypothesis function
Since the hypothesis function arrived at by taining may be overfitted, we cannot evaluate it using the same examples used for training. Rather, one would typically do a (random) 70/30 split of the available data and use the larger portion for training and the smaller for evaluation testing. I.e. judging the accuracy of the hypothesis based on the errors it makes *on the training* set is a *very poor* metric.

Calculating misclassification error rate

The *misclassification error rate* can be calculated like this (in pseudo-code):

foreach test: totalErrors += (round(h(x_{test})) != y_{test})
errorRate = totalErrors / numberOfTests

Cross-validation - evaluating alternative hypothesis function models

When determine which of a number of models to use for the hypothesis function (e.g. varying number of polynomial features) to go with, one would make the pick based on cases from the test set (rather than the training set). But the resulting function will then be biased towards that test set. Thus in these cases we need to further spit the test data into a *cross-validation set*, and a test set, where the cross-validation set is used to pick between alternative hypothesis functions. A 60/20/20 split is typical.

Distinguish high bias from high variance (underfitting vs. overfitting)

Comparing the classification error for the training set and the test set, if the training set error and the test set error are both large, then the model probably suffer from underfitting. On the other hand if the training set error is small but the test set error is large, then the model probably suffer from overfitting.

Choosing the regularization parameter λ

Automatically vary λ from 0 to 10 in multiples of 2, i.e. 0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24. Then use the cross-validation set to pick the best λ and use the test set to evaluate the resulting hypothesis function.

Too small training set? Learning curves

By artificially decreasing the training set size in steps, and for each step plotting the error rate for the *training set* and the error rate for the *cross-validation set* we get two curves called the "learning curves". For small number of training examples, the *training set* error rate will be close to zero (because it's easy to fit to the few samples) and it will grow as we train with more samples. For small number of training examples, the *cross-validation set* error will be large (as the hypothesis function is too specific to the training set) and it will reduce as we train with more samples.

If the model used for the hypothesis function has **high bias** (i.e. underfitting due to too few polynomial features) the *cross-validation set* curve will start high and quickly plateau at a high level, and the *training set* curve will start low and quickly plateau too, and run close to the cross-validation set curve. In this case adding more training example *will not help*.

If the model has **high variance** (i.e. overfitting due to too many polynomial features) the *cross-validation set* curve will start high and slowly reduce but still at a high level, and the *training set* curve will slowly grow but stay low. I.e. in this case there will be a significant gap between the two curves. In this case adding more training *is likely to help*.

Selecting model for neural networks

For neural networks a single hidden layer is a reasonable default, but adding more hidden

layers and more units in each layer usually only improve predictiveness. If the network suffers from overfitting due to being too large then regularization is usually better than making the network smaller, but too large network will suffer from slow performance.

XI. Machine Learning System Design

Improving a machine learning system - what to prioritize
To improve the performance of a machine learning system, try to brainstorm a list of possible improvements and then analyse which will give the best bang-for-the-buck. Avoid going first for large-effort schemes (such as building a large training set) if there is not very good indication it will be most beneficial.

Recommended approach for building a new machine learning system
1) Start with a simple algorithm that you can implement quickly. Train it and then test it with your cross-validation data. 2) Plot learning curves to help decide how to improve the system step-by-step. 3) Perform error analysis: manually examine the examples that the system made errors on, and draw conclusions from this on how to improve the system.

Error analysis
Defining a single real number measure of the systems accuracy (e.g. its *misclassification error rate*) makes evaluating different modifications or improvements much easier.

Error measure for skewed classes
If the size of the classes are very skewed (e.g. 99.5% of data is one class and only 0.5% another class) then misclassification error rate is a very poor accuracy measure. A much better evaluation metric is “precision/recall”, where the accuracy for a skewed data set is only good if both precision *and* recall are high.

However, varying the probability threshold for when predictions are considered positive will always result in a trade-off between precision and recall (higher precision and lower recall, or lower precision and higher recall).
Prediction metrics: precision and recall
Assuming y = 1 for the rarer class, we can classify the predictions as follows:

	Predicted class 1	0
Actual class 1	True Positive	False Negative
0	False Positive	True Negative

Precision: How big portion of classifications did we get right? E.g. of all patients where we predicted y = 1, what fraction actually has cancer?

- Precision = (true positives) / (true positives + false positives)

Recall: How big portion of the "rare" cases where y = 1 did we catch? Catching too many is always better than too few. E.g. of all patients that actually has cancer, what fraction did we correctly predict as having cancer?

- Recall = (true positives) / (true positives + false negatives)

Prediction metrics: average and F_1 score

Using precision and recall may be better than misclassification error rate, but this gives us two metrics rather than one, which might make it harder to tell at a glance if the prediction is better or worse.

To get a single metric, the two numbers can be averaged, but this may result in the "best" prediction function having an unreasonable small precision or recall.

Average:

- $Average = \frac{true\ positives + true\ negatives}{total\ examples}$

A better metric is the "F_1 score", which ensures both precision and recall are reasonably large.

F_1 score:

- $F_1\ score = 2 \times \frac{precision \times recall}{precision + recall}$

Week 7

X. Support Vector Machines

Another commonly used and powerful supervised learning algorithm (besides logistic regression and neural networks) is Support Vector Machines.

The *support vector machine* is also sometimes called a "*large margin classifier*" because when defining the decision boundary between two classes it tries to maximize the margin between each class and the boundary.

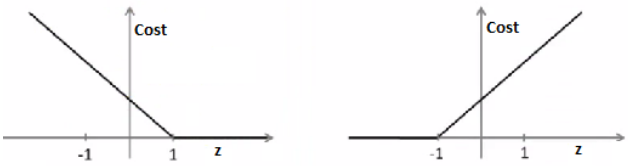
The "support vectors" are what defines the margins on either side of the decision boundary, between the boundary and the data points.

SVMs are essentially the same thing as logistic classification, but with a more efficient cost function (not based on a sigmoid function) and with slightly different syntax, mainly for historical reasons.

$$J(\theta) = C \sum_{i=1}^{m} \left[y^{(i)} cost_1\left(\theta^T x^{(i)}\right) + \left(1 - y^{(i)}\right) cost_0\left(\theta^T x^{(i)}\right) \right] + \frac{1}{2} \sum_{i=1}^{n} \theta_j^2$$

where C is really the same thing as $\frac{1}{\lambda}$ (but the inverse) and we've dropped the $\frac{1}{m}$ for no particular reason. $cost_0()$ and $cost_1()$ are cost functions for when y is 0 and y is 1 that looks

like this:



If $y = 1$, we want $\theta^T x \geq 1$ (not just ≥ 0)
If $y = 0$, we want $\theta^T x \leq -1$ (not just < 0)

The hypothesis $h_\theta(x)$ for a SVM is 1 if $\theta^T x \geq 0$ and 0 otherwise.

SVM's are great for quick hyperplane boundaries to separate different variables. Combining it with PCA can make it very robust.

.....

http://www.work.caltech.edu/~boswell/IntroToSVM.pdf
http://www.cs.cornell.edu/people/tj/publications/tsochantaridis_etal_04a.pdf

http://www.cs.cmu.edu/~ggordon/SVMs/new-svms-and-kernels.pdf

Week 8

XIII. Clustering

Types of unsupervised learning

- Clustering - a way to automatically find structure in data. Good for e.g. market segmentation, social network analysis, organize computer clusters, astronomical data analysis

Notation

$c^{(i)}$: index of clusters (1, 2, ... K) with which example $x^{(i)}$ is currently associated
μ_K: cluster centroid k
K: number of cluster centroids
$\mu_c^{(i)}$:

K-means clustering algorithm

By far the most popular clustering algorithm. It works my first randomly "placing" two (or more (typically up to 10), depending on how many groups we want to split the data into) *cluster centroids* (μ_1, μ_2, ... μ_K) into the data, then repeating these two steps:

1. associate each data point ($c^{(i)}$, where i goes from 1 to m) with the index of the cluster centroid that is "closest", then
2. move each centroid to the spot that is the average (mean) of all the data points now associated with it.

If a cluster centroid does not get associated with any data point then eliminate it (or

randomly re-place it).

For data-sets with non-separated clusters, running the K-means algorithm may result in useful groups of similarly distributed sizes, or it may not.

K-means cost function (distortion function)
The cost function for the K-means algorithm is also known as the *distortion function*, and is the sum of the squared-distance between each point in the training set and the cluster centroid to which it has been associated (i.e. the closest one). Formally:

$$J\left(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K\right) = \frac{1}{m} \sum_{i=1}^{m} || x^{(i)} - \mu_{c^{(i)}} ||^2$$

Practical considerations for K-means
To initialize the centroids, pick K number of data points at random from the training set and place the centroids there.

Running K-means this was often result in local optima (non-optimal clustering) so usually one needs to run K-means many times (e.g. 50 to 1000) before picking the resulting clustering with the smallest distortion.

There is no good way to automatically picking a value for K (the number of clusters to fit the data to), because often there is no clear-cut answer. Thus usually K is picked by hand, perhaps after visualizing the data to estimate the number of groups. An alternative is to run K-means many times initialized with different number of cluster centroids and then plot the minimum cost. The plot may show a clear “elbow”, indicating a good number of clusters. Or, just consider the number of clusters you want, for your “downstream” purpose.

XIV. Dimensionality Reduction

Data compression (data dimensionality reduction)
To reduce the number for features considered, we can try to project the data down on a smaller dimensional plane, e.g. project features in 3D down to a 2D plane. This works in higher dimensions too.

Principal component analysis (PCA)
PCA can be used to identify a lower-dimensional plane onto which the data can be projected, thereby reducing the number of features while minimizing the effect on the predictions for y. Thus the goal of PCA is to minimize the reprojection error (i.e. the difference between the higher- and lower-dimensional data before and after projection).

N.B. despite some cosmetic similarities between PCA and linear regression, they are two completely different things. However, just as for linear regression, the data needs to be preprocessed in a similar way, by mean normalization (and optionally feature scaling).

To reduce data from n-dimensional to k-dimensional first we need to compute the “covariance matrix” Σ (sigma):

$$\Sigma = \frac{1}{m}\sum_{i=1}^{n}\left(x^{(i)}\right)\left(x^{(i)}\right)^T$$

and then compute the "eigenvectors" of the matrix Σ, which in Octave is done like this:
[U, S, V] = svd(Sigma); (svd stands for Singular Value Decomposition)
where Sigma is a n×n matrix. U is the matrix we want. It is also a n×n matrix, and the first k columns can be used to calculate the feature vectors we want. Thus U_{reduce} is a n×k version of the original U, and the new k-dimensional feature vector z is $z^{(i)} = U_{reduce}^T \cdot x^{(i)}$

How to choose k for PCA
Typically k is chosen so that the average squared prediction error (i.e. the squared difference between the feature vectors in n-dimensions and in k-dimensions) is less than 1%, thus retaining 99% variance. I.e.

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\| x^{(i)} - x_{approx}^{(i)} \|^2}{\frac{1}{m}\sum_{i=1}^{m}\| x^{(i)} \|^2} \leq 0.01$$

An easier way to compute the above is to use the S matrix returned from the svd() call. S is a n×n matrix where only the diagonal is non-zero, and the retained variance can be calculated like this:

$$retained\ variance = \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}}$$

i.e. the sum of the k-first element in of the diagonal divided by the sum of all the elements of the diagonal.

Decompressing (reconstructing) PCR-compressed data
To compress from n- to k- dimensional features we did
$z^{(i)} = U_{reduce}^T \cdot x^{(i)}$
To decompress from z back to x_{approx} we just do
$x_{approx}^{(i)} = U_{reduce} \cdot z^{(i)}$

More about PCA
A main use for PCA is to speed up the training of a machine learning algorithm running on a very high-dimensional training set. I.e. PCA can be used to reduce e.g. a 10.000 dimensional feature vector to a 1.000 dimensional one, which will make the learning algorithm run much faster.

Note that to learn the U_{reduce} parameter we must only apply PCA to the training set, or we can't evaluate the effectiveness and correctness of the PCA result.

PCA can often be used to compress the feature vector by 5 or 10 times without loosing too much of the variance.

Bad use of PCA: to prevent overfitting
PCA might work to prevent overfitting (by reducing the number of features) but is not a good way to address overfitting! Use regularization instead.

Recommendation on applying PCA
Do not include PCA by default in your project plan. Apply your learning algorithm on your full training set first, and only introduce PCA if you have a good reason to do so (e.g. performance or data visualization).

Week 9

XV. Anomaly Detection

Examples of anomaly detection
Can be used for example to detect products that need further testing during manufacturing before shipping, or for detecting attempts of fraud, or detecting computers in a data center that are behaving in an unusual way, signaling that they might be about to crash or is being hacked.

How anomaly detection works
To perform anomaly detection, a model needs to be built that match examples of "normal" samples. The model will then flag samples poorly fitting the model as "anomalies".

Fraud detection
Features used e.g. to detect fraudulent users of a web site might include how often the user logs in; how many pages the user visits or number of transactions; number of posts the user makes on a forum; or the user's typing speed. Users that are found to be suspicious can be sent for further review, or given extra identification challenges.

Gaussian (Normal) distribution
The bell-shaped normal distribution function is parameterized by μ (the mean "mu", i.e. the center of the bell on the x-axis) and σ^2 (the variance "sigma squared", where σ is the "standard deviation", i.,e. the width from the mean to the "one normal deviation"). Thus if x is a Gaussian distribution with mean μ and variance σ^2 then $x \sim N(\mu, \sigma^2)$ where the tilde ~ is read "distributed as" and the N should be a curly "script" N (N for Normal) denoting the Gaussian normal-distribution function. Note that a larger σ result in a wider and lower bell shape, while a larger σ result in a narrower and taller bell (the area is always the same so narrowness results in tallness).

Given a sample set, we can estimate μ and σ like this $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ and $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x^{(1)} - \mu\right)^2$

Density estimation
To estimate the probability that x fits the model that describes "normal" cases (known as *density estimation)*, we calculate p(x) like this:

$p(x) = \prod_{j=1}^{n} p\left(x_j;\ \mu_j,\ \sigma_j^2\right)$ where $\prod$ means the product (rather than the sum) of all $x_{1,\dots,}x_n$ and each parameterized by their own μ and σ^2.

Anomaly detection algorithm

1. Choose features x_i you think might be indicative of anomalous examples.
2. Fir parameters $\mu_1, ..., \mu_n, \sigma_1^2, ..., \sigma_n^2$

   $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$

   $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} \left(x^{(1)} - \mu\right)^2$
3. Given a new example x, compute $p(x)$:

   $p(x) = \prod_{j=1}^{n} p\left(x_j; \mu_j, \sigma_j^2\right) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$

   Anomaly if $p(x) < \varepsilon$

Training the anomaly detection algorithm
In general use only the non-anomalous (i.e. where y = 0) examples as training data, but it is okay if a few anomalous examples are (unknowingly) included too, as long as the vast majority are non-anomalous. The y parameter is not used in the training, making the anomaly detection algorithm "unsupervised".

As an example, assuming 10.000 non-anomalous and 20 anomalous examples, split the former into 60/20/20 parts and use as training, cross-validation, and test data, and split the latter into 50/50 parts and include them in the cross-validation and test data sets only.

Use the cross-validation set to select a value for ε that maximise the algorithm's accuracy.

Evaluating the anomaly detection algorithm
Possible evaluation metrics include

- True positive, false positive, false negative, false positive
- Precision/Recall
- F_1-score

Anomaly detection algorithm vs. supervised learning algorithm
An unsupervised anomaly detection algorithm typically works *better* than a supervised learning algorithm if

- the number of anomalous examples are very few (or non existent) compare to the non-anomalous examples.
- if the anomalous are of different types (i.e. hard to cover by any one supervised learning algorithm).
- future anomalies may look nothing like any of the anomalous examples we have seen so far.

Messaging features
If the histogram plot of a feature does not "look Gaussian" (bell-shaped), then try changing the x into e.g. log(x), or sqrt(x), or log(x+c) where c is a constant that you can play with to make it look Gaussian, or x to the power of some fraction. Anything is fine as long as the feature histogram plot looks Gaussian.

Error analysis for anomaly detection

If the algorithm fail to flag some anomalous examples in the test data set, then look at each of those examples and try to come up with some new feature that distinguish it from the non-anomalous examples.

Given a set of features, try combining them as ratios of each other to make new features that capture certain anomalous cases.

Multivariate Gaussian distribution
The model used so far result in a Gaussian distribution where its contour plot is always axis-aligned (it may only be elongated horizontally or vertically, never diagonally). This limits the anomalies that can be detected. A more general model use multivariate Gaussian distribution, where the real-value σ is replaced by a $n \times n$ matrix Σ, and using non-zero values for the non-diagonal of the matrix results in a diagonally elongated contour plot.

XVI. Recommender Systems

Week 10

XVII. Large Scale Machine Learning

XVIII. Application Example: Photo OCR

Useful resources

List of 40+ Machine Learning APIs

Free books:

- The Elements of Statistical Learning: Data Mining, Inference, and Prediction (Second Edition), by Trevor Hastie, Robert Tibshirani, Jerome Friedman. February 2009
- A First Encounter with Machine Learning, by Max Welling, Donald Bren. November 4, 2011
- Information Theory, Inference, and Learning Algorithms, by David J.C. MacKay. Version 7.2 (fourth printing) March 28, 2005
- Bayesian Reasoning and Machine Learning, by David Barber. DRAFT March 29, 2013
- Pattern Recognition and Machine Learning, by Christopher M. Bishop. 2006
- Introduction to Machine Learning (Second Edition). by Ethem Alpaydın. 2010
- Pattern Classification and Machine Learning, by Matthias Seeger. April 8, 2013

Teaching slides etc:

- Alex Holehouse's Machine Learning 2011 lecture notes (the ultimate guide!)
- https://dl.dropboxusercontent.com/u/1477925/Machine%20Learning.pdf
- Statistical Learning. By Federica Giummol
- Stanford CS 229, Machine Learning Course Materials

Other

- http://graphlab.org/

- Everything You Wanted to Know About Machine Learning, But Were Too Afraid To Ask
- A Few Useful Things to Know about Machine Learning