Machine Learning Notes - Coursera Stanford Course

August 31, 2016

1 Week 1

1.1 Introduction

Supervised Learning: We are told the correct answer, and we base predictions off of those. Example: predicting house prices.

Unsupervised Learning: We are not told what each data point represents. The algorithm tries to find structure in the data. (Think clustering algorithms). Example: auto-clustering of news into sections regarding certain topics.

Classification Problems: predict discrete-valued outputs.
Regression Problems: predict continuous ie. real-valued outputs.

1.2 Linear Regression - Model and Cost Function

Model representation:

1. m = number of training examples
2. x = input variables
3. y = output variables

1.2.1 Univariate Linear Regression Hypothesis

$$h_\theta = \theta_0 + \theta_1 x$$

Note that this is a function of input variable x.

1.2.2 Cost Function

Cost function ie. squared error function:

$$J(\theta_1, \theta_2) = \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Note that this is a function of the parameters θ_1, θ_2. The goal is to minimize the cost function by tuning parameters θ_1, θ_2. We have in the denominator a $2m$ term to make the math cleaner, but having J without the 2 would yield the same result, as we are simply performing minimization. This cost function is the most common for regression problems.

1.2.3 Cost Function Intuition

We can use contour plots and surface plots in order to determine which values of θ_1 and θ_2 will cause a minimum of $J(\theta_1, \theta_2)$.

1.3 Gradient Descent

Algorithm, given $J(\theta_0, \theta_1)$:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a (local) minimum

Mathematically, we repeat the following until convergence:

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

For linear regression:

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

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

"Batch" Gradient Descent - each step of the gradient descent uses all the training examples. Not the only form.

where α is the learning rate, (the size of the step taken).

Updates must occur simultaneously (do not use latest value of θ_0 to find θ_1).

1.3.1 Intuition

1. small α makes process slow
2. large α makes gradient descent overshoot the minimum - it may fail to converge or diverge
3. the learning rate does not need to be changed during gradient descent (smaller steps taken automatically by smaller derivative)

2 Week 2

2.1 Multivariate Linear Regression

2.1.1 Multiple Features

- n = number of features
- $x^{(i)}$ = input (features) of i^{th} training example
- $x_j^{(i)}$ = value of feature j in the i^{th} training example

$$h_\theta = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n = \theta^T x$$

For convenience, $x_0 = 1$, such that x can be treated as a vector of size n + 1. The parameters are stored in a vector θ which is also of that size.

2.1.2 Gradient Descent for Multiple Variables ($n \geq 1$)

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

2.1.3 Feature Scaling

Get every feature into approximately a $-1 \leq x_i \leq 1$ range; makes gradient descent converge faster (makes contour plots less skewed)
Mean normalization: replace x_i with $x_i - \mu_i$ to make features have approximately zero mean (do not apply to x_0)

$$x_j = \frac{x_j - \mu}{\sigma}$$

Instead of using σ, can use range instead.

2.1.4 Learning Rate

There is a learning rate at which $J(\theta)$ will monotonically decrease (ie decrease for each iteration). Debug algorithm by plotting $J(\theta)$ vs. number of iterations. Symptoms of a learning rate that is too large are: divergence, $J(\theta)$ that decreases and then increases repeatedly (looks like $-|sin(x)| + c$).

2.1.5 Polynomial Regression

Nonlinear polynomial regression can be done via substitution of a x^n term to a x term. Feature scaling becomes increasingly important. The range used in feature scaling must change accordingly (eg. range from 1-100 for an x term becomes 1-10,000 for an x^2 term).

2.2 Computing Parameters Analytically - Normal Equation

Using calculus, we take partial derivative of J with respect to the parameters θ allows for minimization of J with respect to the parameters.

$$\theta = (X^T X)^{-1} X^T y$$

X is the 'design matrix' and is of size $m \times (n+1)$. The first column of X is a vector of 1's. Each column thereafter contains the j^{th} feature of a given data point x^i. In other words, each row is just a 1 followed by the x-values associated with a given data point. The column vector y contains the output values, and is $m \times 1$ in size.

Feature scaling is not necessary for the normal equation method.

2.2.1 Gradient Descent vs Normal Equation

Gradient descent requires iterations and a learning rate, and works well even when n (the number of features) is large. The normal equation method becomes slow when n is very large, as $X^T X$ is an $n \times n$ matrix whose inversion is slow. For $n \geq 10,000$, opt for gradient descent.

2.2.2 Normal Equation Non-invertibility

Two causes:

1. Redundant features (linearly dependent features)
2. Too many features ie. $m \leq n$ such that too little data is available to fit all those features (results in non-invertible/singular matrix)
3. (solution is to delete some features or to use regularization)

2.3 Vectorization of Implementation

User-defined iterating routines are likely slower than library calls which are vectorized implementations. Example: performing dot product as matrix multiplication is faster than performing it via a summing for-loop. A vectorized implementation of gradient descent:

$$\theta := \theta - \alpha\delta$$

where θ and δ are size $n+1$ column vectors, and

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

where $x^{(i)}$ is a size $n+1$ column vector and $(h_\theta(x^{(i)}) - y^{(i)})$ is a real number.

3 Week 3

3.1 Classification

$y\epsilon\{0,1\}$, where 0 is the negative class, and 1 is the positive class.
Logistic Regression: classification algorithm that enforces $0 \leq h_\theta(x) \leq 1$

3.1.1 Hypothesis Representation

$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}} = P(y=1|x;\theta)$
It is based on the Sigmoid/logistic function: $g(z) = \frac{1}{1+e^{-z}}$ The output of the hypothesis is the probability that y = 1 given an input x that is parameterized by θ.

3.1.2 Decision Boundary

Given a boundary (like predict 1 when $h(x)$ is greater than 0.5 and otherwise 0), it is easy to solve for the boundary value of $\theta^T x$. From there, it is easy to determine the values of x_i that result in a given outcome in terms of an inequality.

3.2 Logistic Regression Model

3.2.1 Cost Function

Since $h(x)$ is now a nonlinear function, using the cost function as previously defined would result in several local minimia, therefore it is unlikely for gradient descent to achieve the global minimum, which is only guaranteed to occur for convex functions. The following cost function gives us a convex cost function that is local minimum free, and is derived from the principle of maximum likelihood estimation (stats concept):

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

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases} = -ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$$

3.2.2 Gradient Descent

Again, use gradient descent to minimize cost function, and the update rule is identical to linear regression once partial derivatives of J are taken.

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

However, the definition of $h_\theta(x)$ is now based on the sigmoid function rather than a polynomial. Use vectorized implementation using θ as a vector of the parameters and update them simultaneously:

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

Can also use feature scaling make logistic regression run faster.

3.3 Advanced Optimization

Optimization algorithms to minimize $J(\theta)$:

1. Gradient Descent
2. Conjugate Gradient
3. BFGS

4. L-BFGS

The latter algorithms require no manual picking of α and often arrive at a solution much faster than gradient descent would. However, they are more complex. Use the fminunc (unconstrained minimization) function in MATLAB. You must write a function that computes the cost function given some θ vector and computes a vector known as the 'gradient' which stores the partial derivatives of J with respect to the parameters θ_i.

3.4 Multi-class Classification (any number of outputs)

One-vs-all (one-vs-rest) classification: given k classes, train k logistic regression classifiers where all data not in the k^{th} class is considered as being in the negative class. $h_\theta^{(i)}(x) = P(y = i|x;\theta)$ for each class $(i = 1, 2..k)$. To classify a unknown input x, pick the class i that maximizes $h_\theta^{(i)}(x)$. In other words, pick the classifier which thinks most enthusiastically that the new data point fits into that class.

3.5 The Problem of Over-fitting

Fitting data that appears to follow a square root function with a straight line is said to produce an output that is 'underfit' and has 'high bias'. The algorithm has a preconception that the data should be linear.
Fitting the same data with a quartic polynomial may result in all training examples being passed through well, but the algorithm has 'overfit' the data and has 'high variance'. We don't have enough data to constrain this high order polynomial.
Overfitting: the algorithm makes accurate predictions for examples in the training set (the cost function approaches 0), but it does not generalize well to make accurate predictions on new, previously unseen examples due to too many features and not enough data.

3.5.1 Addressing Overfitting

1. Reduce number of features (either manually or through model selection algorithm)

2. Regularization

3.5.2 Regularization

Penalize parameters $\theta_1 ... \theta_n$ to make them very small - modify cost function as follows:

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

where λ is the regularization parameter.

The result is a simpler hypothesis that is less prone to overfitting, as minimizing this function results in much smaller θ values. The left half satisfies the goal of fitting the training set, and the right half satisfies the goal of keeping parameters small. If λ is too large, then under-fitting will occur as essentially θ_0 will remain in the hypothesis, with very small contributions from the other parameters.

3.5.3 Regularized Linear Regression

For gradient descent, repeat:

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

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \text{ for } (j = 0, 1, 2, 3, ..., n)$$

For the normal equation:

$$\theta = (X^T X + \lambda I')^{-1} X^T y$$

where I' is the identity matrix with size $(n+1) \times (n+1)$ where the top left entry is 0. Using regularization with λ>0 makes $X^T X$ always invertible.

3.5.4 Regularized Logistic Regression

For gradient descent, repeat (same as linear regression) :

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

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \text{ for } (j = 0, 1, 2, 3, ..., n)$$

The partial derivatives are now:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j$$

4 Week 4

4.1 Neural Networks

4.1.1 Model Representation

Neuron model - logistic unit: based on sigmoid (logistic) activation function. "Parameters" θ can also be known as "weights". We often omit the x_0 (bias unit) because its value is always 1.
Neuron network: neurons strung together.

Layers:

1. Layer 1: input layer - x value
2. Layer 2: hidden layer - values you don't observe in the training set (could be more than jut one layer; there can also be a bias unit here)
3. Layer 3: output layer - y value

Terminology:

1. $a_i^{(j)}$: "activation" of unit i in layer j, where "activation" means the output value
2. $\theta^{(j)}$: matrix of weights controlling function mapping from layer j to layer $j+1$.
3. L = total number of layers in network
4. K = number of output units (for multi-class classification problems this is the number of classes)

If a network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$. These units do not include the bias unit.

Forward propagation (vectorized implementation): You're basically just using logistic regression to get an output from the last hidden layer to the output layer, but instead of using the inputs you're using the activations. The activations (in the hidden layers) are learned as a function of the input. You're no longer constrained to just using the features provided, and can have complex nonlinear hypotheses.

4.2 Applications

Digital logic can be computed given that the sigmoid function reaches 0.99 at 4.6 and 0.01 at -4.6. These can be rounded to binary values of 1 and 0. You can write mathematical functions that then just output values greater than these tolerances to make pseudo digital logic. XNOR is computed by ORing the AND of two inputs with the NAND of two inputs. This requires one hidden layer.

4.2.1 Multiclass Classification

Example: Handwriting detection
Extension of one-vs-all method. In the training set $x^{(m)}, y^{(m)}$, represent $y^{(m)}$ as a column vector of size equivalent to the number of classes. The values in the vector are 0 for all classes except the class to which the data point belongs. Likewise, the output $h_\theta(x)$ is a vector of similar size and characteristics. However, the values will be approximately 0 or 1, not necessarily exact due to the asymptotic nature of the sigmoid function.

5 Week 5

5.1 Neural Network Cost Function

Expands on regularized logistic regression cost function.

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

The hypothesis $h_\theta(x) \epsilon \mathbb{R}^K$ and $(h_\theta(x))_i = i^{th} output$.

We do not sum over the θ terms where $i = 0$ because those multiply into the bias units - we don't want to regularize them.

5.1.1 Gradient Computation

In order to minimize the cost function using an algorithm like gradient descent, we need to compute the gradient of J. Given one training example (x, y), we start with forward propagation. A vectorized implementation where there are 2 hidden layers (total 4 layers):

$$a^{(1)} = x$$
$$z^{(2)} = \theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)})$$
$$\text{add } a_0^{(2)} \text{ bias unit}$$
$$z^{(3)} = \theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)})$$
$$\text{add } a_0^{(3)} \text{ bias unit}$$
$$z^{(4)} = \theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\theta(x) = g(z^{(4)})$$

Now we do back propagation:

Define $\delta_j^{(l)} =$ = "error" of node j in layer l (based on its activation). There is no $\delta^{(1)}$ because the input layer has no error associated with it. Depending on the implementation, the δ values corresponding to the bias units ($\delta_0^{(l)}$) can be meaningless as we generally don't change the value of +1 assigned to the bias units. Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i)$

$$\delta^{(4)} = a^{(4)} - y$$

where each value is a vector of size equal to the number of outputs.

$$\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)}. \times g'(z^{(3)})$$
$$==$$
$$\delta^{(3)} = a^{(3)}. \times (1 - a^{(3)})$$

$$\delta^{(3)} = (\theta^{(2)})^T \delta^{(3)}. \times g'(z^{(2)})$$
$$==$$
$$\delta^{(2)} = a^{(2)}. \times (1 - a^{(2)})$$

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = a_j^{(l)} \delta_i^{(l+1)}$$

if we set $\lambda = 0$.

Formally, given training set $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$:

1. Set $\triangle_{ij}^{(l)} = 0$ for all l, i, j.
2. For i = 1 to m:
   (a) Set $a^{(1)} = x^{(i)}$
   (b) Perform forward prop to compute $a^{(l)} for l = 2, 3, ...L$
   (c) Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
   (d) Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$
   (e) $\triangle_{ij}^{(l)} = \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} == \triangle^{(l)} = \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
3. $D_{ij}^{(l)} = \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \theta_{ij}^{(l)}$ if $j \neq 0$
4. $D_{ij}^{(l)} = \frac{1}{m} \triangle_{ij}^{(l)}$ if $j = 0$

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)}$$

5.1.2 Intuition on Back Propagation

https://www.coursera.org/learn/machine-learning/lecture/du981/backpropagation-intuition

5.2 Back Propagation in Practice

Using advanced optimization algorithms like fminunc in MATLAB requires use of initial guess and gradient in the form of size $n + 1$ vectors. With a neural network, we now deal with θ and gradient D matrices that must be unrolled into vectors.

Learning algorithm (given L = 4):

1. Have initial parameters $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$
2. Unroll to get initialTheta to pass to fminunc(@costFunction, initialTheta, options) using [Theta1(:); Theta2(:), Theta3(:)]
3. function [jval, gradientVec] = costFunction(thetaVec)
   (a) From thetaVec, get $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$ using reshape command
   (b) Use forward/backward propagation to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\theta)$
   (c) Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\theta)$ to get gradientVec

5.2.1 Gradient Checking

Even with subtle bugs, neural networks can seem like they're working in that the cost function continues to be minimized. But in reality, they do not necessarily find the minimum. Gradient checking is a means of detecting such bugs.

Use a central difference to numerically approximate the gradient, using a step of 10^{-4}.

Given parameter unrolled vector θ:

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, ...\theta_n + \epsilon) - J(\theta_1, \theta_2, ...\theta_n - \epsilon)}{2\epsilon}$$

Code:

```
for i = 1:n
        thetaPlus = theta;
        thetaPlus(i) = thetaPlus(i) + EPSILON;
        thetaMinus = theta;
        thetaMinus(i) = thetaMinus(i) - EPSILON;
        gradientApproximation(i) = (J(thetaPlus)       J(thetaMinus)
end
```

Recall that back propagation generates DVec (which are the same partial derivatives of the cost function with respect to our params). Compare our numerically computed derivatives with the DVec value. If these two ways give similar answers (up to a few decimals), our implementation of back prop is likely correct. When actually using back prop for training the classifiers, turn off gradient checking as it is computationally expensive.

5.2.2 Initialization of θ Parameters

Initializing $\theta_{ij}^{(l)}$ to be 0 does not work like in regression. After each update, the parameters corresponding to the inputs going into each of the hidden units are equal to each other even through iteration of gradient descent. This limits what kind of functions the neural network can compute. This problem is called symmetric weighting.

Use random initialization to allow for symmetry breaking. Initialize each $\theta_{ij}^{(l)}$ to a (different) random value in $[-\epsilon, \epsilon]$ where ϵ is some value.

5.3 Putting it all Together

Training a NN:

1. Pick a network architecture (connectivity pattern)
2. Number of input units = dimension of features $x^{(i)}$
3. Number of output units = number of classes
4. Reasonable Default: use 1 hidden layer (most common), or if you have more than one hidden layer, have the same number of hidden units in every layer (usually the more the better but more expensive) and comparable to the number of inputs or some multiple of it?
5. Randomly initialize weights to small values near 0
6. Implement forward propagation to get $h_\theta(x^{(i)}$ for any $x^{(i)}$
7. Implement code to compute cost function $J(\theta)$
8. Implement back prop to compute partial derivatives $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$
9. Use a for loop, iterating of training examples, performing forward prop and backward prop on each training example $(x^{(i)}, y^{(i)}$ to get $a^{(l)}$ and $\delta^{(l)}$ terms for $l = 2, ..., L$. In the for loop, compute $\triangle^{(l)} = \triangle^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
10. Outside the for loop, compute partial derivatives of J, taking into account regularization term λ.
11. Use gradient checking to compare gradients computed using back prop vs. a numerical estimate using central differences of J
12. Disable gradient checking
13. Use gradient descent or advanced optimization method with back prop (which computes the partial derivatives) to try to minimize J as a function of parameters θ. Note that J is non-convex in this case, so you're not guaranteed to get a global optimum. Local optima tend to be good enough though.

Idealized output y is a column vector with a 1 in one row and zeros in all the rest.

6 Week 6

If the hypothesis is not very good at making predictions on new data, there are some options:

1. Getting more training examples (fixes high variance problems)
2. Trying a smaller set of features (choosing them more carefully to prevent overfitting) (fixes high variance)
3. Getting additional features (fixes high bias)
4. Try adding polynomial features (fixes high bias)
5. Decreasing λ for regularization (fixes high bias)
6. Increasing λ for regularization (fixes high variance)

Don't waste time using gut feelings to pick a learning algorithm. Use a machine learning diagnostic: a test you can run to gain insight into what is or isn't working with a learning algorithm, and gain guidance as to how to best improve its performance.

6.1 Evaluating a Learning Algorithm

6.1.1 Evaluating Hypothesis

You can plot in 2D to check for overfitting, but you can't do this with more features. Solution: split your training data into two sets: the training set, and the test set. The training set should contain a random 70% of your training data, and the test set should contain the balance 30%. Learn the parameters θ from the training set (ie. run minimization on $J(\theta)$, then compute the test set error. For linear regression, this means we compute (note no regularization):

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

For logistic regression, we compute:

$$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y_{test}^{(i)} log h_\theta(x_{test}^{(i)}) + (1 - y_{test}^{(i)}) log h_\theta(x_{test}^{(i)})$$

Or, we can use a number that's easier to understand - the 0/1 misclassification error which is just the ratio of how many are classified correctly vs incorrectly.

$$err(h_\theta(x), y) = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0.5 \text{ and } y = 0, \text{ or if } h_\theta(x) < 0.5 \text{ and } y = 1 \\ 0 & \text{otherwise} \end{cases}$$

Test error $= \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\theta(x_{test}^{(i)}), y_{test}^{(i)})$

6.1.2 Model Selection - How to Choose the Degree of Polynomial to Fit

Take another parameter, the degree of the polynomial, d, (in addition to the θ parameters) that we must determine. Take each model, fit it to the training set, and compute the test set error, and pick the one with the lowest test set error. But how well does this model generalize? We picked the value of d that gave us the best possible performance on the test set is likely to be an overly optimistic estimate of generalization error. Because we fit the parameter d based on the test set, it is no longer fair to evaluate the hypothesis on this test set. (You can't report the generalization error accurately by using the test set.)
Solution: use 60% of data as training set, 20% as cross-validation set, and 20% as the test set. Compute $J(\theta)$ for each of the sets. Minimize $J(\theta)$ w.r.t. θ for each degree of model. Compute the cross-validation set error, and pick the hypothesis with the lowest cross-validation error. Now, estimate the generalization error using the test set error.

6.2 Bias vs. Variance

6.2.1 Diagnosing Bias vs Variance

Low degree polynomials result in high training error and high cross validation error ($J_{cv}(\theta) \approx J_{train}(\theta)$). This is an underfitting/high bias problem.
High degree polynomials result in low training error and high cross validation error ($J_{cv}(\theta) >> J_{train}(\theta)$). This is a overfitting/high variance problem.
Good fits will have intermediate training error and intermediate cross-validation error.

6.2.2 Auto-selecting λ

Have some range of λ you want to fit, and increment in multiples of 2 (0, 0.01, 0.02, 0.04...). For each, minimize the cost function $J(\theta)$ (which includes the regularization term) to solve for the parameter vector θ. In this case, we

define separately $J_{train}, J_{cv}, J_{test}$ to not include the regularization term. Use the cross-validation set fit with the different parameter vectors, and compute the cross-validation error. Pick the one with the smallest cross-validation error. Report the test set error as J_{test}) as a measure of how well the model will generalize.
Consider J_{train} and J_{cv} (the ones without regularization). Low λ results in overfitting (high variance) and thus low J_{train} and high J_{cv}. Large λ results underfitting (high bias) and thus high J_{train} and high J_{cv}. Intermediate values of λ result in intermediate J_{train} and small J_{cv}. Plotting these allows for confidence in picking λ.

6.2.3 Learning Curves

High bias problems (low degree polynomial): as you increase the training set size, J_{cv} decreases up to a certain point but plateau's relatively quickly. J_{train} will increase up to a certain point and plateau. Both are high values and are approximately the same. Getting more training data will not, by itself, help much.
High variance problems (high degree polynomial with small λ): as you increase the training set size, the training set error increases as it becomes harder to fit lots of data points. Cross-validation error decreases as you increase the training set size. There is a large gap because the training error and cross-validation error. The two curves converge to one another, so getting more training data is likely to help - cross validation error will continue to go down, which is good.

6.2.4 Relating to Neural Networks

Small networks have fewer parameters - they are cheaper but prone to underfitting. Large networks have more parameters (either through more hidden layers or more units in a given hidden layer) - they are more expensive and are more computationally expensive. Use regularization to deal with overfitting. To pick the number of layers, you can use training, cross-validation, and test sets, and see which performs best on the cross-validation set.

6.3 Spam Classifier

There are many ways to develop what features to use. One way is to take most frequently occurring n words (10k-50k) in the training set, and those become the features.

6.3.1 Error Analysis

Recommended approach to developing an ML algorithm:

1. Start with a simple algorithm that you can implement quickly (like 24 hours). Implement it and test it on your cross-validation data.
2. Plot learning curves to decide if more data, more features, etc. are likely to help
3. Error analysis: manually example the examples (in the cross-validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

It is useful to have a numerical evaluation method that gives a single real-value to benchmark versions of the algorithm against others. Use the cross-validation error for this.

6.4 Handling Skewed Data

Skewed classes: when the ratio of positive to negative examples is very close to one of the extremes. For example, having 1% test error on cancer prediction isn't so great when only 0.5% of examples actually have cancer. Just always predicting that no one has cancer would actually get you better 'accuracy' by the error metric, although this would be a bad algorithm for prediction. Therefore we need a new error metric.

6.4.1 Precision/recall

$$\text{Accuracy} = \frac{\text{True positives + True negatives}}{\text{Total examples}}$$
$$\text{Precision} = \frac{\text{True positives}}{\text{True positives + False positives == Predicted positives}}$$
$$\text{Recall} = \frac{\text{True positives}}{\text{True positives + False negatives == Actual positives}}$$

Higher values are better for both. 'Positive' is defined in terms of presence of the rarer class that we want to detect. These metrics handle skewed classes.

6.4.2 Trading Off Precision and Recall

For most classifiers, there is a tradeoff between precision and recall. In logistic regression, we predict 1 if $h_\theta(x) \geq$ threshold and predict 0 otherwise. Higher thresholds cause higher precision (fewer false positives) and lower recall. Lower thresholds cause higher recall and lower precision. The tradeoff can be plotted.

We want a single value to represent if the algorithm is good or bad. Standard way to combine them is the F_1 score $= 2\frac{PR}{P+R}$. Measure P and R on the cross-validation set and choose the value of the threshold that maximizes $2\frac{PR}{P+R}$.

6.5 Using Large Data Sets

Large data rationale: use a linear algorithm with many parameters (e.g. logistic regression/linear regression with many features, or a NN with many hidden units). These are low bias algorithms, meaning $J_{train}(\theta)$ will be small. Using a very large training set with these algorithms means we will be unlikely to overfit, and hence exhibit low variance. Consequently, $J_{train}(\theta) \approx J_{test}(\theta)$ and hence $J_{test}(\theta)$ will be small. Therefore we will have a low bias and low variance algorithm.
More data will not help when the features x do not contain enough information to predict y accurately, regardless of if we have a large number of hidden units or have simple algorithm. A useful test is: given the input x, can a human expert confidently predict y based solely on the features x?

7 Week 7

7.1 Large Margin Classification - Support Vector Machines

7.1.1 Optimization Objective

SVMs are an alternative view of logistic regression. Examining the cost terms when $y = 0$ (when $\theta^T x >> 0$) and $y = 1$ (when $\theta^T x << 0$), we get $z = -log\frac{1}{1+e^{-z}}$ and $z = -log\frac{1}{1+e^{-z}}$, $z = -log(1 - \frac{1}{1+e^{-z}})$, respectively. Define $cost_1(z)$ and $cost_0(z)$ as the cost functions when $y = 1$ and $y = 0$ respectively for an SVM, which are the asymptotic versions of the aforementioned cost components for logistic regression. $cost_1(z) = 1$ for $z > 1$. $cost_0(z) = 1$ for $z < -1$.
Logistic regression had the form $A+\lambda B$; SVMs have the form $CA+B$ where if $C = \lambda^{-1}$, then the two optimization objectives should give you the same values for θ. In other words, the equations aren't identical, but yield the same result in minimization.

$$\text{minimize } \theta \text{ in } C\sum_{i=1}^{m}[y^{(i)}\text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)})\text{cost}_0(\theta^T x^{(i)})] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

SVM hypothesis does not output probabilities like in logistic regression.

$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

7.1.2 Large Margin Intuition

If $y = 1$, we want $\theta^T x \geq 1$ (not just ≥ 0 like in logistic regression). If $y = 0$, we want $\theta^T x \leq -1$ (not just ≤ 0 like in logistic regression). SVM doesn't just want to "barely get the example right." Adds a safety margin into SVM.

SVMs try to separate the data (ie. draw a decision boundary) with as large a margin from the training examples as possible when C is very large. If C is very large, then the SVM will not reject outliers. If C is not too large, it will have reasonable outlier rejection.

Math behind SVMs as a large margin classifier: https://www.coursera.org/learn/machine-learning/lecture/3eNnh/mathematics-behind-large-margin-classification

7.2 Kernels

7.2.1 Motivation

We want to fit non-linear decision boundaries, but it's not evident that introducing polynomial terms it the best choice as previously used.

7.2.2 Landmarks

Given x, compute new features depending on the proximity to 'landmarks' $l^{(1)}, ..., l^{(m)}$.

7.2.3 Kernel Function

Our features are now defined by f_i (formerly by x_i).

$$f_1 = similarity(x, l^{(1)}) = exp(-\frac{||x - l^{(1)}||^2}{2\sigma^2}) = exp(-\frac{\sum_{j=1}^{n}(x_j - l_j^{(1)})^2}{2\sigma^2})$$

The similarity function is known as a Gaussian Kernel. There are other Kernel functions. Also expressed as $k(x, l^{(1)})$. When $x \approx l^{(1)}$, then $f_1 \approx 1$ If x is far from $l^{(1)}$ then $f_1 \approx 0$.

7.2.4 How to Choose Landmarks

Choose landmarks as exactly the same location as training examples. So $l^{(m)} = x^{(m)}$. We generate a feature vector $f = [f_0, f_1, ..., f_m]$ where $f_0 = 1$. Given a training example, we can compute the vector f - one of the terms must be equal to 1 as there will be a landmark equivalent to that training example. Therefore, each training example can be represented as a feature vector.

7.2.5 SVM with Kernels

SVMs and Kernels go well together due to advanced computational optimization techniques - kernels tend to not be computationally efficient for other things like logistic regression.
Hypothesis: Given x, compute features $f \epsilon R^{m+1}$. Predict $y = 1$ if $\theta^T f \geq 0$. Note that $\theta \epsilon R^{m+1}$.
Training:

$$(\text{minimize w.r.t } \theta) C \sum_{i=1}^{m} y^{(i)} cost_1(\theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

Since $n = m$ since we have as many features (due to selection of landmarks) as we have training examples, the second summation can be rewritten. Also we continue to omit θ_0 in regularization hence why we start from $j = 1$.

7.2.6 SVM Parameters

C plays a role similar to λ^{-1}. Large C causes lower bias, higher variance (tells SVM to classify all examples correctly). Smaller C causes higher bias, lower variance.
C is a positive value that controls the penalty for misclassified training examples. Large σ^2 means features f_i vary more smoothly (not as sharp of a peak, but still the same peak value of 1). This means there is higher bias and lower variance. Small σ^2 means features less smoothly, meaning similarity function moves more abruptly. Higher bias, lower variance.

7.3 Using an SVM

Use an SVM software package like liblinear or libsvm to solve for θ. We need to specify: C and kernel (similarity function).

Choices:

1. No kernel (linear kernel). Predict $y = 1$ when $\theta^T x \geq 0$. Use when n is large (lots of features) and m is small, using linear decision boundary to avoid overfitting. Very similar to logistic regression.

2. Gaussian kernel. We need to choose a σ^2 which varies bias/variance. Use when n is small (like 2D training set) and/or m is large (lots of training examples).

Do perform feature scaling before using the Gaussian Kernel. This is important because we are taking squares and the distance between an input x and the landmark l can start to become dominated by the distance between just one of the features.

7.3.1 Other Kernels

Polynomial kernel: $(x^T l + \text{constant})^{\text{degree}}$. Tends to be not that great, used more when x and l are always non-negative.
Others: String kernel (for input that's string-based), chi-square kernel, histogram intersection kernel. Not all similarity functions make valid kernels - we must satisfy Mercer's Theorem to make sure the SVMs packages' optimizations run correctly and do not diverge.
Choose parameters and kernel type based on whatever performs best on the cross-validation data.

7.3.2 Multi-class Classification

Many SVM packages already have built-in multi-class classification functionality. Otherwise, use one-vs.-all method (Train K SVMs, one to distinguish $y = i$ from the rest, for $i = 1, 2, .., K$), get $\theta^{(1)}, ..., \theta^{(K)}$. Pick class i with largest $(\theta^{(i)})^T x$.

7.3.3 Logistic Regression vs SVM

When n = number of features ($x \epsilon R^{n+1}$), m = number of training examples.
If $n \geq m$ use logistic regression, or SVM with "linear kernel". This is because if you have a lot of features with a smaller training set, a linear function will probably do fine, and you don't have much data to fit a nonlinear function.
When n is small and m is intermediate ($n = 1 - 1000, m = 10 - 10000$), use SVM with Gaussian Kernel.
If n is small and m is large ($n = 1 - 1000, m = 50000+$), create/add more features, then use logistic regression or SVM without a Kernel. Having a massive training set size causes Gaussian Kernel use to be slow.

7.3.4 Neural Network vs SVM

Neural network likely to work well for all the above settings, but may be slower to train. SVM optimizations are convex problems, so you'll always get global optimum. Neural networks can give local optima.

8 Week 8

8.1 Clustering - K-means Algorithm

Most popular, most widely used. Inputs: K (number of clusters), training set $\{x^{(1)}, ..., x^{(m)}\}$. Use convention that training examples are n-dimensional vector, $x^{(i)} \epsilon R^n$ (drop $x_0 = 1$ convention). Algorithm: randomly initialize K cluster centroids $\mu_1, .., \mu_k \epsilon R^m$. Repeat:

```
for i = 1 to m
        # cluster assignment step
        c^(i) := an index (from 1 to K) corresponding to cluster
         centroid closest to x^(i). Take each example and find the
         closest cluster centroid. Find index k that minimizes
         ||x^{(i)} - Mu_k||^2.
for k = 1 to K
        # move position of centroid
        Mu_k = average (mean) of points assigned to cluster k
        # this means Mu an n-dimensional vector
```

If there is a cluster centroid with 0 points assigned to it, eliminate the cluster centroid, or randomly re-initialize it.

8.1.1 K-means for non-separated clusters

Works on non-separated clusters - market segmentation might be an example.

8.1.2 Optimization Objective

Terminology/variables

1. $c^{(i)}$ index of cluster 1,2,...,K to which example $x^{(i)}$ is currently assigned
2. μ_k = cluster centroid k ($\mu_k \epsilon R^n$)
3. $\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned

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

The summation is for the distance between the example $x^{(i)}$ and the location of the cluster centroid to which that example has been assigned. The objective is to minimize J with respect to parameters c, μ. Also called the 'distortion' function. The 'cluster assignment' step minimizes J with respect to c. The 'move centroid' step minimizes J with respect to μ.

8.1.3 Random Initialization

Should have $K < m$, intuitively. Randomly pick K training examples. Set $\mu_1, ..., \mu_K$ equal to these K examples.
K=means can end up at local optima depending on the initialization. Normally run K-means 50-1000 times. Algorithm (for i = 1 to 100):

1. Randomly initialize K-means
2. Run K-means. Get c, μ.
3. Compute cost function (distortion) J

Pick clustering that gave lowest cost $J(c^{(i)}, ..., c^{(m)}, \mu_1, ..., \mu_K)$. If $K = 2$to10 you're likely go see benefits from this method. But if K is in the hundreds, it's more likely that the first solution will give you a pretty decent optimum. Doing more random initializations may help a bit, but probably not huge.

8.1.4 Choosing Number of Clusters

No good automatic solution - look at visualizations/output of clustering algorithm.
Elbow method: Plot J against K (number of clusters). Cost will decrease rapidly, and then suddenly start decreasing less rapidly (forming an elbow). But location of 'elbow' isn't always clear.
Alternatively, evaluate K-means based on a metric for how well it performs for your later down-stream purpose.

8.2 Dimensionality Reduction

If you have highly correlated features (ie. measurement in centimetres and measurement in inches), you may want to reduce the dimension from 2D to 1D. Fit a line through the data with respect to the two features - project

the examples onto the line. Now you only need 1 number to specify the position of the point on the line. This is an approximation of the training set. Allows learning algorithms to run more quickly and reduces memory requirement.
You can also reduce 3D to 2D by projecting the data onto a plane. Each example can now be represented using 2 real numbers (or a 2-dimensional vector z).
Dimensionality reduction also allows us to visualize data more easily. Given n features find some way to determine two features that summarize the data then plot in 2D.

8.2.1 Principal Component Analysis (PCA)

Tries to find a lower-dimensional surface on which to project data such that average squared projection error is a minimum. First perform mean normalization and feature scaling. (For 2D to 1D, find a vector). Formally, to 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. Project data onto the linear subspace spanned by the set of k vectors.
PCA is not linear regression - PCA tries to minimize the orthogonal distance to the lower-dimensional surface, whereas linear regression tries to minimize the 'vertical' direction (the y direction) error. In PCA, all features are treated symmetrically (there is no y to prefer).

8.2.2 PCA Algorithm

Data preprocessing: mean normalization (compute the mean of each feature, replace each $x_j^{(i)}$ with $x_j - \mu_j$ to get 0 mean). If different features are on different scales, scale features to have comparable range of values.
Algorithm:

1. Compute covariance matrix $\sum = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})(x^{(i)})^T$. Vectorized: Sigma = 1/m * X' * X

2. Compute the eigenvectors of matrix $\sum$ with [U,S,V] = svd(Sigma) in MATLAB. Note that 'svd' is singular value decomposition. $x^{(i)}$ is an $n \times 1$ vector, making $(x^{(i)})(x^{(i)})^T$ $n \times n$.

3. U is $n \times n$. The first k columns of the U matrix are the vectors you want. This is a $n \times k$ matrix, which we call U_{reduce}. Ureduce = U(:, 1:k).

4. Let $z = U_{reduce} \times x$, which is a k-dimensional vector. ($z^{(i)} = U_{reduce} \times x^{(i)}$) or z = Ureduce' * x. Similar to k-means, apply to vectors $x \epsilon R^n$ - not done with $x_0 = 1$ convention.

We can often reduce 5-10x without losing much variance in the data.

8.2.3 Reconstruction from Compressed Representation

Since $z = U_{reduce}^T x$, then $X_{approx} = U_{reduce} z$ where X_{approx} is $n \times 1$.

8.2.4 Choosing Number of Principal Components k

Compactly, *svd* returns an S matrix, which is diagonal. Pick the smallest value of k such that

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m} S_{ii}} \geq 0.99$$

. This quantity is more explicitly equivalent to the average squared projection error $\frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - x_{approx}^{(i)}||^2$ divided by the total variation in the data $\frac{1}{m} \sum_{i=1}^{m} ||x^{(i)}||^2$ being less than 0.01. This means we have 99% of variance retained. We usually want between 95 and 99% variance retained. A crude algorithm would be to increase k iteratively, compute U_{reduce}, z, and x, and keep going until the aforementioned long-form division is less than or equal to 0.01, but this is very slow.

8.2.5 Advice for Applying PCA

Supervised learning speedup: given $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$, extract the inputs into an unlabeled dataset $x^{(1)}, ..., x^{(m)}$ which is of a high order. Run PCA on the unlabelled data set to make it a lower order input $z^{(1)}, ..., z^{(m)}$. The new training set is now $(z^{(1)}, y^{(1)}), ..., (z^{(m)}, y^{(m)})$. Any new examples coming in must first be mapped to z and then passed into the hypothesis. Note: mapping $x^{(i)}$ to $z^{(i)}$ should be defined by running PCA only on the training set. This mapping can be applied to the examples $x_{cv}^{(i)}$ and $x_{test}^{(i)}$ in the cross-validation and test sets.
Bad use of PCA: to prevent overfitting as a result of a smaller number of features. This might okay, but you should use regularization instead. PCA doesn't consider the y values when it throws some info away.
Before implementing PCA, first try running whatever you want to do with the original/raw data $x^{(i)}$. Only if that doesn't do what you want (slow,

too much memory), then implement PCA and consider using compressed representation $z^{(i)}$.

8.3 Anomaly Detection

Given dataset that is assumed to be non-anomalous: $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$. Is x_{test} anomalous? Build probabalistic model $p(x)$ using data set and use some ε threshold.
Applications: fraud detection (could be supervised learning if you have a lot of examples of kinds of fraud), manufacturing QA, monitoring computers in a data center

8.3.1 Gaussian Distribution

Area under the curve is constant. Smaller σ means taller curve. We say: $x\tilde{N}(\mu, \sigma^2)$. Tilde means 'distributed as'.

8.3.2 Density Estimation

Given training set $x^{(1)}, ..., x^{(m)}$ where each example $x \epsilon R^n$. Algorithm:

1. Choose features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, ...\mu_n, \sigma_1^2, ..., \sigma_n^2$
3. $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$, $\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$. μ is a vector of size n.
4. Given a new example x, compute $p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp(\frac{-(x_j-\mu_j)^2}{2\sigma_j^2})$
5. Anomaly if $p(x) < \varepsilon$

8.3.3 Developing and Evaluating an Anomaly Detection System

Assume we have some labeled data of anomalous and non-anomalous examples ($y = 0$ if normal, $y = 1$ if anomalous). Assume that the training set $x^{(1)}, ..., x^{(m)}$ contains normal examples (non-anomalous) but it's okay if a few anomalies slip into the training set. Cross-validation set: $(x_{cv}^{(1)}, y_{cv}^{(1)}), ..., (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$. Test set: $(x_{test}^{(1)}, y_{test}^{(1)}), ..., (x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$. Include 50% of your anomalous examples in the cross-validation set and 50% of your anomalous examples in the test set. Use the old 60-20-20 breakup

for the non-anomalous examples between the sets. We usually have 2-50 anomalous examples for thousands of normal examples.

Fit model $p(x)$ on training set. On a cross-validation/test example, x, predict $y = 1$ if $p(x) < \varepsilon$ (anomaly) and predict $y = 0$ otherwise (normal). We have a skewed data set inherently (we're dealing with anomalies), so some evaluation metrics should include precision/recall, F1 score, true/false positives/negatives. You can also use the cross-validation set to choose the parameter ε that maximizes F1 score. Continue to evaluate algorithm in terms of ε, which features to include, etc. based on cross-validation set. Then take final model and evaluate it on the test set.

8.3.4 Anomaly Detection vs Supervised Learning

Since we're labelling our data as anomalous and normal, why not use supervised learning (logistic regression, neural network)? Use anomaly detection if: we have a very small number of positive examples (y=1) in the range of 0-20 and a large number of negative (y=0) examples (used to fit $p(x)$). Usually you have such a small set of these, that you save them for the cross-validation set and the test set. Supervised learning is normally used when you have a large number of positive and negative examples.
Anomaly detection algorithms are used for many "types" of anomalies - future anomalies may look nothing like any of the anomalous examples we've seen so far. That's why we model based on the negative examples. Supervised learning is used when you have enough positive examples for the algorithm to get a sense of what positive examples are like - future positive examples are likely to be similar to ones in the training set.
For the spam email problem, although there are a lot of types of spam, we normally treat it as a supervised learning because we have enough data on all those 'anomalies'.

8.3.5 Choosing what Features to Use for Anomaly Detection

Non-gaussian features: Plot the data with histogram - even if data is non-gaussian with respect to a feature, the algorithm can work okay. You can apply various transformations (like $log(x + c), x^c$) to the data to get it to look gaussian.
Error analysis: The most common problem is that p(x) is comparable for both normal and anomalous examples, where in reality it should be large for normal examples and small for anomalous examples. Look at the anomalies that algorithm is failing to detect, and find unusual property of anomaly

to find features that may indicate the anomaly. Choose features that might take on unusually large or small value sin the event of an anomaly (use things like ratios of existing features if that makes sense, ex. (CPU load)/(network traffic) as opposed to just looking at those features independently.

8.3.6 Multivariate Gaussian Distribution

Motivation: Allows you to capture when you expect two different features to be positively/negatively correlated (contours not necessarily axis-aligned) $x \epsilon R^n$. We model $p(x)$ all in one shot as opposed to modelling $p(x_1), p(x_2), ...$ separately.

$$p(x; \mu, \sum) = \frac{1}{(2\pi)^{n/2} |\sum|^{1/2}} exp(-\frac{1}{2}(x-\mu)^T inv(\sum)(x-\mu))$$

, where $\sum$ is the $n \times n$ covariance matrix and $\mu \epsilon R^n$. Varying diagonal elements: larger values mean wider, shorter bell in the given axis. Varying off diagonal elements: changes correlation (note it can be positive or negative).

8.3.7 Fitting Parameters μ and $\sum$

Parameter fitting: given training set $x^{(1)}, ..., x^{(m)}$, where $x \epsilon R^n$

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

and

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

8.3.8 Relationship to Original Model

Original model (which is multiplication of $p(x, \mu, \sigma^2)$) is a special case of multivariate where there are zeros on the non-diagonal elements. (Note: diagonals are the σ^2 values). The original model has the contours always axis-aligned.

8.3.9 Anomaly Detection Algorithm with Multivariate Gaussian

1. Fit model $p(x)$ by setting $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ and $\sum = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)(x^{(i)} - \mu)^T$.
2. Given a new example x, compute: $p(x) = \frac{1}{(2\pi)^{n/2}|\sum|^{1/2}} exp(-\frac{1}{2}(x - \mu)^T inv(\sum)(x - \mu))$
3. Flag anomaly if $p(x) < \varepsilon$

8.3.10 When to use Multivariate vs Original Model

Original model: used more often, but requires that you manually create features to capture anomalies where x_1, x_2 take unusual combinations of values (ie. they're correlated via some non-linear combination). Computationally cheaper. Scales better to large number of features, n.
Multivariate Gaussian: Automatically captures correlations between features. Computationally more expensive as $\sum \epsilon R^{n \times n}$ and we're taking its inverse. We must also have $m > n$ or else $\sum$ is non-invertible. In reality, you should have m like 10x bigger than n. $\sum$ may also be non-invertible if you have redundant features or features that are linearly dependent.

8.4 Recommendation Systems

8.4.1 Problem formulation

Ex. predicting movie ratings - users rates movies using zero to five stars. Predict what they would have rated movies they haven't rated (ie. watched) in order to build a recommender system.

Terminology:

1. n_u = number of users
2. n_m = number of movies
3. $r(i, j) = 1$ if a user j has rated movie i (0 otherwise)
4. $y^{i,j}$ = rating by user j on movie i (if defined)
5. $\theta^{(j)}$ = parameter vector for user j
6. $x^{(i)}$ = feature vector for movie i
7. $m^{(j)}$ = number of movies rated by user j

8.4.2 Content-based Recommendations

Create a feature vector, x, for each movie (remember to include the $x_0 = 1$). For each user j, learn a parameter $\theta^{(j)} \epsilon R^3$. Predict user j as rating movie i with $(\theta^{(j)})^T x^{(i)}$ stars.

To learn $\theta^{(j)}$): minimize with respect to $\theta^{(j)}$ the

$$\frac{1}{2} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$\theta^{(j)}$ is an $n+1$ dimensional vector - don't regularize over θ_0. To learn $\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}$ (for all users), minimize over $\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n_u)}$:

$$J(\theta^{(i)}, ..., \theta^{(n_u)}) = \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

Gradient descent algorithm:

$$\theta_k^{(j)} = \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} \text{ for k} = 0$$

$$\theta_k^{(j)} = \theta_k^{(j)} - \alpha (\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)}) \text{ for k} \neq 0$$

8.5 Collaborative Learning

Algorithm learns what features to use by itself; every user collaborates to get better movie ratings for everyone. Let's say that the users told us their θ vector (their preference for movie genre). You can then try to find what the reasonable values for the features might be. We want to pick feature value such that the predicted value of how the user rates a movie is not far from the actual value in the squared error sense.

8.5.1 Optimization algorithm

Given $\theta^{(1)}, ..., \theta^{(n_u)}$, to learn $x^{(i)}$ we must minimize with respect to $x^{(i)}$:

$$\frac{1}{2} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(j,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Given $\theta^{(1)}, ..., \theta^{(n_u)}$, to learn $x^{(i)}, ..., x^{(n_m)}$ we must minimize with respect to $x^{(1)}, ..., x^{(n_m)}$:

$$\frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(j,j)})^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2$$

Basic collaborative filtering algorithm: given x we can estimate θ and given θ we can estimate x. Start by guessing θ and find x and then iterate to get a better guess of θ, etc.

8.5.2 Collaborative Filtering Algorithm

More efficient algorithm that doesn't alternative between θ and x - can solve simultaneously. Notice that the first double summation in both the θ estimation and the x estimation are basically doing the same summation just in different orders. We can combine the optimization objectives. In this case, we do away with x_0 and hence θ_0 so both are vectors of size n, as opposed to formerly, $n+1$. There is no need to hard code a feature equal to 1 because we're now learning all the features.

$$J(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}) = \frac{1}{2}\sum_{i,j:r(i.j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

Algorithm:

1. Initialize $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}$ to small random values (to allow for symmetry breaking so that the features that are learned are different from each other)

2. Minimize $J(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm). For every $j = 1, ..., n_u, i = 1, ..., n_m$:

$$x_k^{(i)} = x_k^{(i)} - \alpha(\sum_{j:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})\theta_k^{(j)} + \lambda x_k^{(i)})$$

$$\theta_k^{(i)} = \theta_k^{(i)} - \alpha(\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} + \lambda \theta_k^{(j)})$$

3. For a user with parameters θ and a movie with features x, predict a star rating of $\theta^T x$.

8.6 Low Rank Matrix Factorization

8.6.1 Vectorization: Low Rank Matrix Factorization

Low Rank Matrix Factorization is just another name for collaborative filtering. If we have matrix X which has as its rows the transpose of $x^{(1)}$ all the way to $x^{(n_m)}$ and a matrix θ which has as its rows the transpose of $\theta^{(1)}$ to $\theta^{(n_u)}$, then the predicted ratings are $X\theta^T$, which is a low rank matrix.

8.6.2 Finding Related Movies

After creating features, it's hard to have human-understandable form for them. To determine if movie j is related to movie i, you need to find it where you have a small value of $||x^{(i)} - x^{(j)}||$ (they're similar).

8.6.3 Implementation Detail - Mean Normalization

Can make algorithm work a little better. Normalize each row of matrix that stores ratings for various movies by various users (users on y axis). Subtract the row's mean from each element in that row. Using this mean-normalized data, then learn $\theta^{(j)}, x^{(i)}$. For a user j on a movie i, predict $(\theta^{(j)})^T(x^{(i)})+\mu_i$. This way, if a user has not rated anything, you'll have a non-zero θ vector for them which is now actually equivalent to just the average rating for that movie.

9 Week 10

9.1 Learning With Large Datasets

Before coding, do sanity check that a smaller data set would not suffice. Look for a high variance learning curve.

9.2 Gradient Descent with Large Datasets

Gradient Descent (more specifically, the 'batch' version) becomes more computationally expensive when dealing with large datasets as we sum over m examples.

9.2.1 Stochastic Gradient Descent

For linear regression: $cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$.

$$J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

Algorithm:

1. Randomly shuffle dataset (speeds up convergence)
2. Repeat: for i = 1,..,m
   (a) $\theta_j = \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$ (for j = 0,...,n)

The term multiplied by α is the partial derivative with respect to θ_j of the cost function in terms of θ and $(x^{(i)}, y^{(i)})$. In each step in the inner loop, do a little gradient descent step with respect to just the one training example to try to fit it better. Rather than passing through all training examples before modifying the parameters and making progress. In stochastic, you can just look at a single training example and start making progress already. Each iteration may generally move the parameters in the direction of the global minimum, but not always (takes a more stochastic path). Doesn't get to the global minimum - keeps wandering around it in a small region. You normally do the outer loop between 1 to 10 times.

9.2.2 Mini-Batch Gradient Descent

Can sometimes work faster than stochastic. Batch gradient descent used m examples in each iteration. Stochastic gradient descent uses 1 example in each iteration. Mini-batch gradient descent uses b examples in each iteration. Typically pick b between 2 and 100.Say $b = 10, m = 1000$.

1. Repeat:
   (a) for $i = 1, 11, ...991$
      i. $\theta_j = \theta_j - \alpha\frac{1}{10}\sum_{k=i}^{i+9}(h_\theta(x^{(k)}) - y^{(k)})x_j^{(k)}$ (for every $j = 0, ..., n$)

Out-performs stochastic gradient descent if you have a good vectorized implementation. (You can partially parallelize the gradient computations). Disadvantage is that you have to select b.

9.2.3 Stochastic Gradient Descent Convergence

During learning, compute $cost(\theta, (x^{(i)}, y^{(i)}))$ before updating θ using $(x^{(i)}, y^{(i)})$. Every 1000 (arbitrary) iterations, plot $cost(\theta, (x^{(i)}, y^{(i)}))$ averaged over the last 1000 processed by the algorithm. Graph may be noisy. Averaging over more examples can make smoother curve to see overall trend better but feedback is less granular obviously.
It is possible to get a better result with a smaller learning rate because the parameters oscillate in a smaller fashion around the global minimum. If cost increases, use smaller value of α. α is typically held constant, but you can slowly decrease it over time if you want θ to converge. $\alpha = \frac{const1}{iterationNumber1+const2}$.

9.3 Online Learning

Don't use a fixed training set, just learn continuously from the web traffic you're getting then throw away that data point. Example: dynamically adjusting price of product based on if a user happens to purchase the product. You can get the (x, y) corresponding to the given user, use it to update θ using the gradient descent update rule (with no summation since we only have the one example), and then throw it out. This algorithm adapts to changes in user preferences. Learn $p(y = 1|x; \theta)$, $\theta_j = \theta_j - \alpha(h_\theta(x) - y)x_j$ for (j = 0,...,n)
Example: predicted click through rate (CTR) ie. if you have a search results page, you want to determine, based on the features of the search results, what the probability is that the user will click through and then learn which search results are best to display.
Other examples: displaying special offers, customized selection of news articles, product recommendations.

9.4 Map Reduce and Data Parallelism

Many learning algorithms can be expressed as computing sums of functions over the training set. Split up chunks of the sums, send them to different computers/cores, then sum the results. For neural networks, use map-reduce method to compute forward propagation and backward propagation on 1/10 of the data to compute the derivative with respect to that 1/10 of the data.

10 Week 11

10.1 Photo OCR

Machine learning pipeline: system with many stages/components, several of which use machine learning.
Photo OCR machine learning pipeline: Given image, 1. Text detection. 2. Character Segmentation. 3. Character classification.

10.2 Sliding Windows

Example: pedestrian detection supervised learning. Each pedestrian probably has a 'fixed' aspect ratio - about 82 by 36 pixels lets say. Train classifier with lots of positive and negative examples. Then given a picture, move our sliding window of that constant rectangle size by a step-size/stride and try to see if there is a pedestrian in that window. Then do the same thing with a larger image patch, but scale it down into the size recognized by the classifier.
For text detection using sliding windows, train classifier with pictures of characters vs. pictures with no characters. You can then build a map of all the regions of probability where it thinks it found a character. Run an 'expansion' step on the high probability regions so that contiguous characters' regions now overlap into one big region.
Then, we need a 1D sliding window for character segmentation (constant window size and only 1D sliding because we're only looking at a box that contains a line of text). We train a classifier with positive examples being images with a split down the middle of two characters and negative examples being full characters. We can then slide the window and find the locations where we should split the image up into characters.
Lastly, we perform character classification.

10.3 Getting Lots of Data and Artificial Data

Most reliable way to get high performance learning system is to take low bias algorithm and train it on a massive data set. In the example of character recognition, you can download various fonts and paste them on random backgrounds and add some rotation and scaling to get lots of synthetic data. You can also synthesize data by introducing distortions in real data (like warping the characters). In the application of speech recognition, we could add various realistic background noise. Usually it does not help to add purely random/meaningless noise to the data.

Make sure you have a low bias classifier before expending the effort (ie. plot the learning curves). Keep increasing the number of features/hidden units in neural network until you have a low bias classifier.
Always ask: "how much work would it be to get 10x as much data as we currently have". The answer could be artificial data synthesis or to collect/label it yourself and you should mathematically determine how long it would take. Lastly, you could crowd source (get people to label) - Amazon Mechanical Turk.

10.4 Ceiling Analysis: What Part of Pipeline to Work on Next

You try to determine the upside potential of improving a given component. Create a table with Component and Accuracy columns. Compute the accuracy of the overall system as is. Now, go component by component, and for each test set example, just provide the correct output (manually), thereby simulating what would happen if that component had 100% accuracy. Compute the accuracy of the entire system. Keeping the 100% accuracy on the previous component, now provide the correct output manually for the next component, and compute the accuracy of the overall system. Find the step transition that has the biggest delta in accuracy from one step to the next. The difference between the current system state accuracy and the accuracy where the first component has perfect output is the potential gain from improving that first component. The difference between the accuracy where the first component has perfect output and where the second component has perfect output gives the potential gain from improving the second component.