Coursera Machine Learning Notes

Eddie Shim

June 8, 2017

1 Intro

- **Machine Learning**:
  A subfield of computer science that gives computers ability to learn without being explicitly programmed. Explores the study and construction of algorithms that can learn from and make predictions on data.

- **Supervised Learning**:
  Dataset fed to algorithm includes the dependent variables (answers to problem at hand is part of dataset)
  Eg: regression, classification (predicting housing prices given its characteristics, predicting tumor malignancy);
  (linear regression, logistic regression, neural networks, support vector machines)

- **Unsupervised Learning**:
  Dataset fed to algorithm does not include dependent variables
  Eg: clustering (market segmentation, social networking analysis, facial recognition);
  (k-means, PCA, anomaly detection)

2 Regression

2.1 Linear Regression

For a univariate linear regression, take a dataset of two variables and optimize the linear equation $y = \theta_1 x + \theta_0$. Use the least squares method $\sum_{i=1}^{m}(f(x)-y)^2$, where $(f(x)-y)^2$ represents the residual squared. To minimize residual squared, find $\vec{\theta}$ which makes the derivative of the cost function equal to 0.

- **Cost function for linear regression** (where $\vec{\theta} \in \mathbb{R}^{n+1}$, where n is the number of independent variables):

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

$$h_\theta(x) = \theta_0 + \sum_{i=1}^{n}\theta_i x_i$$

  where the framework h_θ represents a "hypothesis", or the output prediction of the model

- **Gradient descent** (Note: must update all θ_j simultaneously):

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

  ex: for 2 variable case:

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

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

- **Feature scaling**: normalizing all independent variables to an appropriate range. Purpose is the speed up gradient descent.

$$x_i = \frac{x_i - \mu}{\text{range of } x}$$

- Note: if α is too large, gradient descent may skip the global minimum point. However, if α is too little, it may take too long to converge.
- **Normal equation** an alternative to gradient descent:

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

Gradient Descent	**Normal Equation**
- need to choose an α	- no need for α
- needs many iterations	- no need to iterate, one calculation
- works well even when n is large	- need to compute $(X^T X)^{-1}$ which runs in $O(n^3)$ runtime
	- slow if n is very large

Table 1: Pros and Cons

- **Vectorization**: to speed up loops.
  e.g. Transform the following code from:

```
for i = 1:3
   for j = 1:m
      theta(i) := theta(i) - alpha * (1/m)*(h_theta(x(j))-y(j))*x(i);
   end
end
```

into:

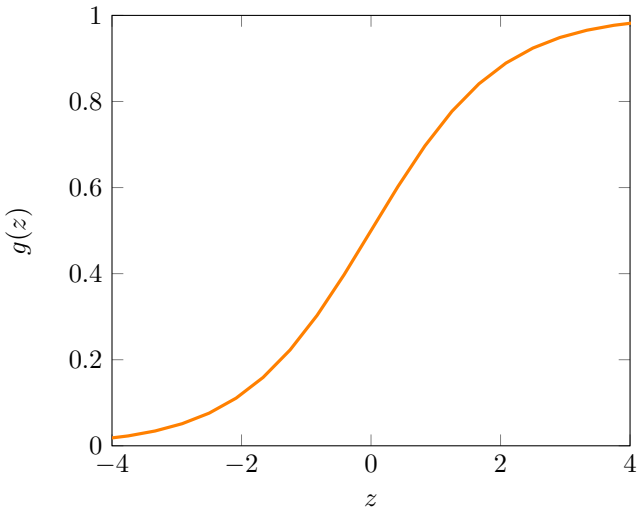
```
theta = theta - alpha * delta;
```

where $\theta \in \mathbb{R}^{n+1},\ \alpha \in \mathbb{R},\ \delta \in \mathbb{R}^{n+1}$

2.2 Logistic Regression

- A classification algorithm (not really regression, which predicts continuous variable given continuous variable)

$$h_\theta(x) = g(\theta^T x),$$
$$\text{where } g(z) = \frac{1}{1+e^{-z}} \quad \text{(sigmoid function)}$$



- Nice properties:
  - $0 \leq h_\theta(x) \leq 1$, good for properties of a probability
  - At $z = 0$, $g(z) = 0.5$
  - Converges to $g(z) = 1$ quickly as g increases, and vice versa for 0
  - We can use these properties to output the probability that an input exists in one of two binary states (1 or 0)
- Terminology: the probability of the output of results equaling 1:

$$h_\theta(x) = p(y = 1|\ x; \theta)$$

- Predict $y = 1$ if $\theta^T x \geq 0 \Leftrightarrow$ if $h_\theta(x) = g(\theta^T x) \geq 0.5$
- Cost Function:

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

where cost $(h_\theta(x), y) =$

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

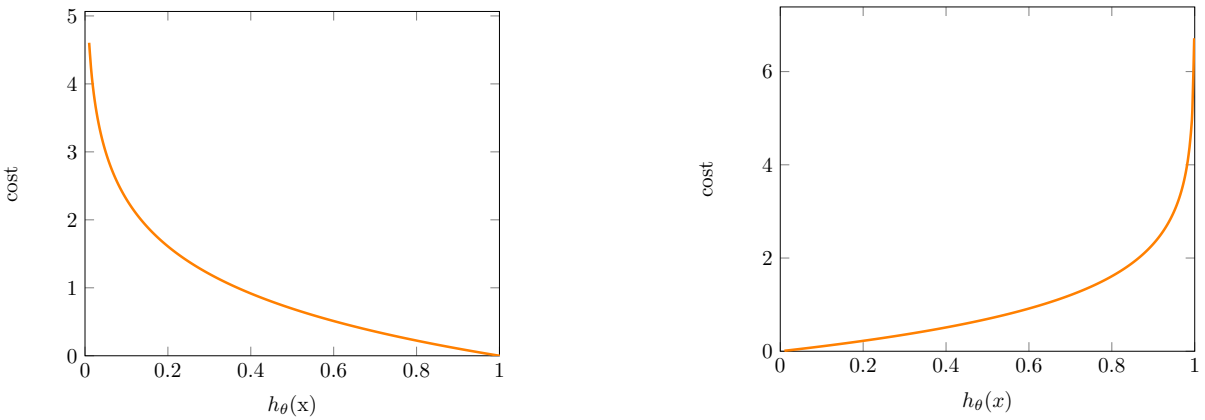


Figure 1: Left: $-log(h_\theta(x))$; Right: $-log(1-h_\theta(x))$

- We can create a more succinct function instead of using a piecewise function.
  Cost function for logistic regression:

$$\text{cost}(h_\theta(x), y) = -y * log(h_\theta(x)) - (1-y) * log(1-h_\theta(x))$$
$$J(\vec{\theta}) = -\frac{1}{m}\Big(\sum_{i=1}^{m} y^{(i)} * log(h_\theta(x^{(i)})) + (1-y^{(i)}) * log(1-h_\theta(x^{(i)}))\Big)$$

2.3 Optimization Algorithms

- Given θ, we want to compute 1) $J(\theta)$ and $\frac{\delta}{\delta\theta_j}J(\theta)$ (for $j = 0, 1, ..., n$) Examples of algorithms we can use to compute the above to motivations include gradient descent, conjugate gradient, BFGS, L-BFGS

2.4 Regularization

- Motivation: to combat overfitting the model, keep certain θ_i parameters small by taxing $J(\theta)$

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

  where λ is the regularization parameter

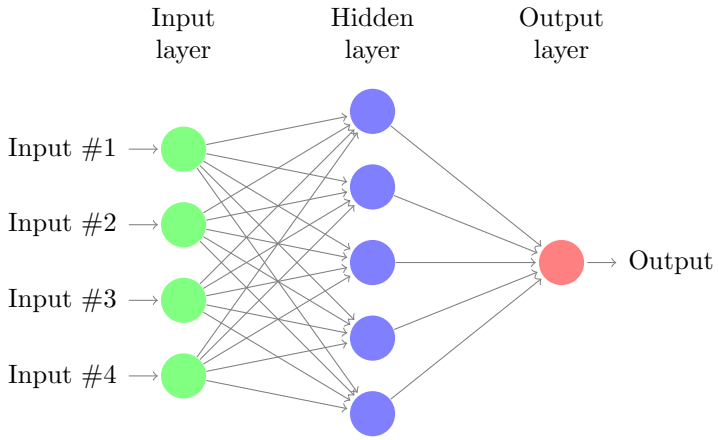
- Normal equation with regularization:

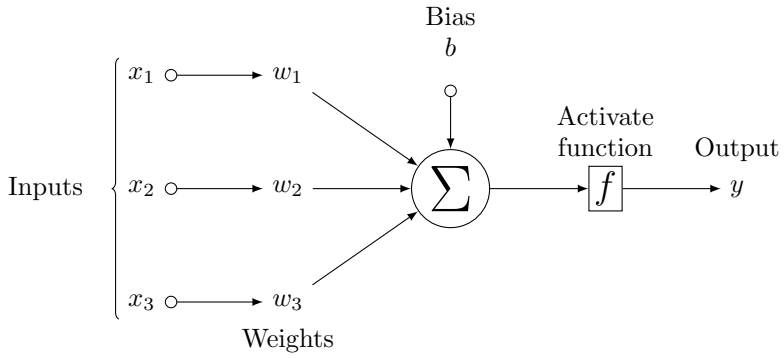
$$\theta = (X^T X + \lambda I)^{-1} X^T y$$

- Regularized gradient descent:

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

3 Neural Networks





- $a_i^{(j)}$ refers to the i^{th} node within the j^{th} layer
- The bias unit is simply =1 and represents a constant within the tuning of θ
- The network is updated per iteration as such:

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

- If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\theta^{(j)}$ will be of dimension $s_{j+1} * (s_j + 1)$
  (ie: $\theta^j \in \mathbb{R}^{S_{j+1} \times (S_j+1)}$)
  (ex: $\theta^{(1)} \in \mathbb{R}^{3\times 4}$ in above example)
- To simplify notation, we set the input of the function g as $\vec{z}$. Thus for example, $a_1^{(2)} = g(z_1^{(2)})$, where $z_1^{(2)} = \left(\theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3\right)$

  Thus, we can succinctly state:

$$
\begin{aligned}
z_i^{(j+1)} &= \theta^{(j)}a^{(j)} \\
a_i^{(j)} &= g(z_i^{(j)}
\end{aligned}
$$

- **Cost function for Neural Network:**

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

3.1 Backwards Propagation

- **Motivation:** To calculate the derivate of the cost function. Compare our forwards propagation calibration with the given data y_i. We compute a $\delta_j^{(l)}$ for every node except our initial layer o $\vec{x}$, which represents the raw data.

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}}\text{cost}(i)$$
$$\text{cost}(i) = y^{(i)} log h_\theta(x^{(i)}) + (1-y^{(i)}) log h_\theta(x^{(i)})$$

- Example: In a neural network wwith 1 input layer, 1 hidden layer, and 1 output node:
$\delta_1^{(3)} = y^{(i)} - a_1^{(3)}$
$\delta_2^{(2)} = \theta_{12}^{(2)}\delta_1^{(3)} + \theta_{22}^{(2)}\delta_2^{(3)}$

- **General procedure:**

1. Given a training set of $\{(x^{(1)}, y^{(1)}, ..., (x^{(m)}, y^{(m)})\}$
2. Set $\Delta_{ij}^l = 0$ (for all l, i, j)
3. For $i = 1$ to m
   - Set $a^{(1)} = x^{(i)}$
   - Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, ..., L$
   - Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
   - Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$
   - $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}$
4. $\Delta_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\theta_{ij}^{(l)}$ if $j \neq 0$
5. $\Delta_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)}$ if $j = 0$

3.2 Gradient Checking

- Motivation: an alternate way to calculate the derivative of the cost function, in order to make sure backpropagation is working properly ($\Delta \approx$ Gradient Approximation)

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

- Gradient checking is computationally expensive. Backpropagation algorithm is better to use.

3.3 Neural Network Overview

- **General procedure:**
  1. Randomly initialize weights $\theta \in [-\epsilon, \epsilon]$
  2. Implement forward propagation to get $h_\theta(x^{(i)})$ for any $x^{(i)}$
  3. Implement code to compute cost function
  4. Implement backpropagation to compute partial derivatives $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$
  5. Use gradient checking to compare $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$ computed using backpropagation versus using numerical estimate of gradient of $J(\theta)$. Then disable gradient checking code (one-off check)
  6. Use gradient descent or other optimization method with backpropagation to try and minimize $J(\theta)$ as a function of parameters θ. *If $J(\theta)$ is non-convex, then gradient descent may get stuck in a local minima.

4 Validation/Training

- Motivation: prevent over/underfitting the model to the data
- General ways to trouble shoot errors:
  - Get more training examples (fixes high variance)
  - Try smaller sets of features (fixes high variance)
  - Try additional features (fixes high bias)
  - Try polynomial features (fixes high bias)
  - Decrease λ (fixes high bias)
  - Increase λ (fixes high variance)
- **Misclassification error (0/1) algorithm:**

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

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

4.1 Model Selection

- **General procedure:**
  1. **Training:** Calculate optimal θ for each polynomial degree using training set (60 %)
  2. **Cross-Validation:** Using θ from Step 1, calculate $J(\theta^{(d)}$ for each degree d and choose optimal d using the cross-validation set (20 %)
  3. **Test:** Estimate Test Error using the test set (20 %)
- Plot error over degree of polynomial d for both the training error and CV error in order to visualize fit
  - Underfit = high bias ($J_{train}(\theta)$ will be high, $J_{CV}(\theta) \approx J_{train}(\theta)$)

- Overfit = high variance ($J_{train}(\theta)$ will be low, $J_{CV}(\theta) >> J_{train}(\theta)$)

- Data Size Impact (Learning Curves)
  - If you have high bias (underfit), then increasing datasize will not decrease error an impactful amount
  - I fyou have high variance (overfit), then more training data helps, as error from cross-validation and training sets converge

5 Large Margin Classification

5.1 Support Vector Machines (SVM)

- **Motivation:** A classification algorithm similar to logistic regression, but instead of outputting a probability of a 0 or 1 state, it directly outputs 0 or 1. Classifies by drawing linear decision boundaries.

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

- **Cost function for SVM:**

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

  where c is a weighting constant, similar to the purpose of λ
  - Large C: (overfitting) lower bias, high variance (small λ)
  - Small C: (underfitting) higher bias, low variance (large λ)

- **Nice properties of SVM:**
  - If $y = 1$, we want $\theta^T x \geq 1$ (not like logistic regression, where $\theta^T x \geq 0$)
  - If $y = 0$, we want $\theta^T x \leq -1$ (not like logistic regression, where $\theta^T x < 0$)

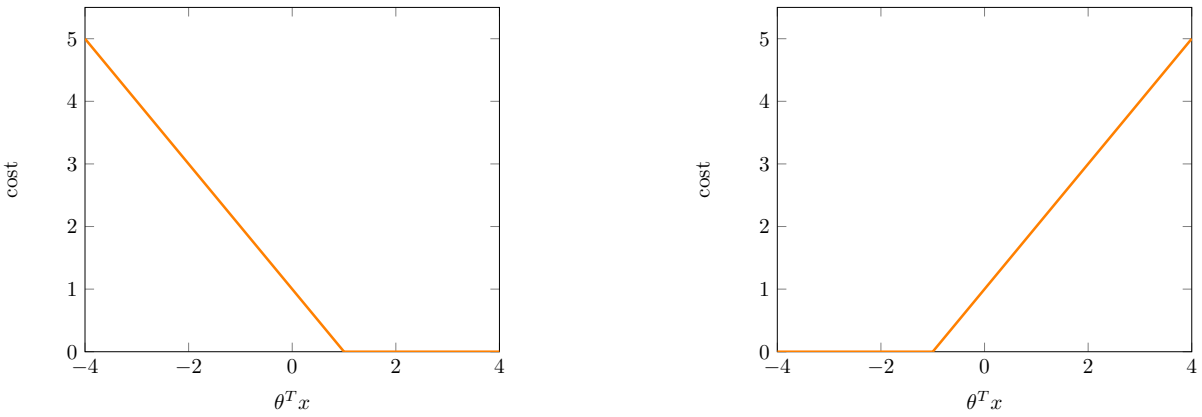


Figure 2: Left:$cost_1(z)$; Right: $cost_0(z)$

5.2 Kernels

- **Motivation:** A classification algorithm which draws non-linear decision boundaries (features can be polynomials). Kernels are calculated using similarity functions.
- Example (Gaussian kernel):

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

- Must choose tuning of σ^2.
  - Smaller σ^2: (overfitting) lower bias, higher variance (narrow Gaussian curve)
  - Larger σ^2: (underfitting) higher bias, lower variance (smoother, flatter Gaussian curve)
- Intuition: calculate the data point x's distance from landmark l (represented by coordinates). Using kernel function f_i, determine if x is close/similar enough to l and classify x in binary as ≈ 0 or 1.
- Example: Say you have three landmark points, l_1, l_2, l_3. To classify data point x as 1 or 0, we plut x into f_1, f_2, f_3 to check if x is similar to any locations. Then plug into $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta 3 f_3 \geq 0$ to classify x as 1 or 0.
- General procedure for using SVM with kernels:
  1. Given training set $\{(x^{(1)}, y^{(1)}, .., (x^{(m)}, y^{(m)})\}$, set landmarks $l^{(1)} = x^{(1)}, ..., l^{(m)} = x^{(m)}$
  2. **(Choice of kernel (sim function)/ choice of $\boldsymbol{\sigma^2}$)** After choosing which similarity function you will use, calculate feature vector $f^{(i)} = [f_0^{(i)}, f_1^{(i)}, ..., f_m^{(i)}]'$ for every $x^{(i)}$, where for example $f_1^{(i)} = \text{sim}(x^{(i)}, l^{(1)})$
  3. **(Choice of C, calibration of $\boldsymbol{\theta}$)** Calibrate weights $\theta \in \Re^{m+1}$ using SVM's cost function, but replacing $\theta^T x^{(i)}$ with $\theta^T f^{(i)}$
  4. For each feature vector from $i = 1, ..., m$, calculate if $\theta^T f^{(i)} \geq 0$,
     where $\theta^T f^{(i)} = \theta_0 f_0 + \theta_1 f_1 + ... + \theta_m f_m$
- Logistic regression versus SVM:
  - Let n = number of features ($x \in \Re^{n+1}$), m = number of training examples.
  - If n is large (relative to m), use logistic regression, or SVM without a kernel ("linear kernel") (e.g. $n \geq m, n = 10,000, m = 10...1000$)
  - If n is small, m is intermediate, use SVM with Gaussian kernel (e.g. $n = 1...1000, m = 10...10,000$)
  - If n is small, m is large, create/add more features, then use logistic regression or SVM without a kernel (e.g. $n = 1...1000, m = 50,000+$)

6 Clustering

6.1 K-means algorithm

- **Motivation:** Allows for categorization of datapoints $\{x^{(1)}, ..., x^{(m)}\}$ (unsupervised learning) to an arbitrary cluster group in $k = 1, ..., K$
- Essentially an optimization problem trying to minimize distance
- **General procedure:**

1. Given a dataset of $\{x^{(1)}, ..., x^{(m)}\}$, we want to allocate K number of **cluster centroids** such that we minimize the total distance between the datasets and cluster centroids.
2. Randomly allocate the cluster centroids $\{\mu_1, \mu_2, ..., \mu_k\}$
3. (Cluster Assignment) For every x from 1 to m, calculate which cluster centroid k is the closest (notated as $c^{(i)}$ = index from $1:k$ corresponding to $x^{(i)}$) where $\boldsymbol{c^{(i)} = \min_k ||x^{(i)} - \mu_k||^2}$
4. (Cluster Movement) Calculate the average of the points assigned to each cluster centroid (example: if datapoints $x^{(1)}, x^{(5)}$ are assigned to cluster centroid $k = 3$, then update $\mu_3 = \frac{1}{2}[x^{(1)} + x^{(5)}]$)
5. Move each cluster centroid to the previous calculation
6. Iterate until convergence

```
while (convergence condition){
for i = 1 to m
c^{(i)} := index (from 1 to K) of cluster centroid closest to x^{(i)}
for k = 1 to K
u_k := average(mean) of points assigned to cluster k
}
```

- **Choosing number of clusters K (Step 1)**
  - No universal/unanimous systematic method yet. Normally people choose by hand by observing visual or function output
  - (Elbow method): plot cost function J over K, (exponentially decreasing function like $\frac{1}{x}$) and pick optimal K (not really rigorous as described by video)
- **Random initialization (Step 2)**
  - Pick k distinct random integers $i_1, ..., i_k$ from $\{1, ..., m\}$. Then set $\mu_1 = x^{(i_1)}, ..., \mu_k = x^{(i_k)}$
  - Code example:

```
for i = 1:100{
Randomly initialize K-means
Run K-means. Get c^{(1)},...,c^{(m)}, mu_1,...,mu_K
Compute cost function (distortion): J(c^{(1)},...,c^{(m)}, mu_1,...,mu_K)
}
Pick clustering that gave lowest cost J(c^{(1)},...,c^{(m)}, mu_1,...,mu_K)
```

- **Optimization objective (Distortion function) (Steps 3 & 4):**

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

7 Dimensionality Reduction

7.1 Principal Component Analysis (PCA)

- **Motivation:** To reduce dataset of n features (unsupervised learning) from $\{x^{(1)}, ..., x^{(n)}\} \rightarrow z^{(1)}, ..., z^{(k)}\}$, where $x^{(i)} \in \Re^n$ and $z^{(i)} \in \Re^k$ and $k \leq n$. Allows for data compression (speed and efficiency) and for data visualization.
- **Summary of PCA:** Find a lower dimensional surface (represented by $\{u^{(1)}, ..., u^{(k)}\}$) onto which to project the data, so as to minimize the squared projection error. Features with high correlation can be represented in lower dimensions while maintaining original data variance.

- *Not the same as linear regression, which minimizes distance between line of best fit and data (vertical). In PCA, you minimize orthogonal distances from the line of best fit and data (angled). We want to minimize the following equation which represents the distance between the data and the principal component:

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

- **General procedure:**

  1. Take data and mean normalize it (calculate $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$ and replace each $x_j^{(i)}$ with $x_j - \mu_j$ as the data). Optimally, feature scale the data as well.
  2. Compute covariance matrix:

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

  3. Compute eigenvectors of matrix Σ:

     `[U,S,V] = svd(Sigma)`

     Where svd is the singular value decomposition algorithm and $U = [u^{(1)}, u^{(2)}, ..., u^{(n)}] \in \Re^{n \times n}$
  4. Take the first k columns of U, defined as matrix $U_{reduced} = [u^{(1)}, u^{(2)}, ..., u^{(k)}]$. Calculate our desired result $z = U_{reduced}^T x$, where $z \in \Re^k$.

- **(Choosing k/ testing quality of PCA)** We want to preserve variance in the original data, up to (1-α) %. Thus we choose k to be the smallest value such that

$$\frac{\text{average squared projection error}}{\text{total variation in the data}} = \frac{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x_{approx}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2} = 1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \leq \alpha$$

  Where $SS_{ii} \in \Re^{n \times n}$ is a diagonal matrix given by a singular value decomposition. Of the two formulas which represent variance retention, the second is more computationally efficient in finding the optimal k.

8 Anomaly Detection

- **Motivation:** Allows for identification of outliers/anomalies. Used for unsupervised learning, but aspects of it still draws from supervised learning concepts. (ex: QA testing on aircraft engines, financial fraud detection, monitoring computers in data center)

- **General procedure (Gaussian distribution example):**

  1. Choose features x_i which might be indicative of anomalous examples
  2. Fit parameters $\mu_1, ..., \mu_n, \sigma_1^2, ..., \sigma_n^2$, where:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

3. Given new example x, compute $p(x)$, where:

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

4. Define x as an anomaly if $p(x) < \epsilon$ (set $y = 1$), and x as normal if $p(x) \geq \epsilon$ (set $y = 0$)

- Anomaly detection vs. supervised learning:

Anomaly Detection	**Supervised Learning (e.g. logistic regression)**
- Very small number of positive examples ($y = 1$). 0-20 is common. Large number of negative examples ($y = 0$)	- Large number of positive and negative examples
- Many different types of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far	- Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set
- e.g.: fraud detection, manufacturing, monitoring machines in data center	- e.g.: email spam classification, weather prediciton, cancer classification

- **Multivariate Gaussian Distribution:** Allows for skewing of normal curve in order to capture correlated data (e.g.: changing top-down views of 2-d graphs from circles into ellipses by chaging off diagonals of Σ)

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right)$$

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

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

where $\mu \in \Re^n$ and $\Sigma \in \Re^{n \times n}$ (covariance matrix)

- Cons of using multivariate Gaussian: computationally more expensive, must have $m > n$ in order for Σ to be invertible, can manually capture correlations in univariate Gaussian by engineering extra features (e.g.: $x_3 = \frac{x_1}{x_2}$))

9 Recommender Systems

- Examples: Netflix's movie recommender, Amazon's product recommender

9.1 Collaborative Filtering

- Essentially a linear regression problem, using regularized gradient descent to find optimal $\vec{x}$ and $\vec{\theta}$
- **Problem formulation:**
  - $\theta^{(j)}$ = parameter vector for user j (calibrated weights for a user's category preferences)
  - $x^{(i)}$ = feature vector for movie i (calibrated weights for categorization of movie)

- $r(i,j) = 1$ if user j has rated movie i (0 otherwise)
- $y^{(i,j)}$ = rating by user j on movie i (if defined)
- n_m = number of movies
- n_u = number of users

General procedure:

1. Initialize $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_m)}$ to small random values (similar to neural network to ensure gradient descent starts at different values)
2. Minimize $J(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_m)}$ using gradient descent (or another optimization algorithm). For every $j = 1, ..., n_u, i = 1, ..., n_m$:

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

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

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

10 Dealing with Large Data

10.1 Stochastic Gradient Descent

- In batch gradient descent, can be computationally expensive to calculate θ_j:

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

because as the size of the dataset m grows, the computation must loop m times in order to take one step.

- **General procedure for stochastic gradient descent:**

1. Randomly shuffle dataset
2. Run gradient descent on loop which updates per data $x^{(i)}$ instead of entire dataset of size m at once:

```
repeat{
for i:=1,...,m{
\theta_j:= \theta - \alpha(h_theta(x^{(i)}) - y^{(i)})x_j^{(i)}
(for every j = 0,...,n)
}
}
```

3. Plot $J(\theta)$ over number of iterations to check how well θ is being tuned (cost is decreasing over time and converging)
4. Tune learning rate α as needed. Decrease α in order to help θ converge, if the cost seems divergent

- The difference: batch gradient descent requires m calculations to travel one step. Stochastic gradient descent takes one step per iteration, taking m steps by the time it loops through $1 : m$. Usually requires anywhere from one to ten times m of calculations to converge.

- Cost function is not necessarily monotonically decreasing, can bounce around

10.2 Mini-Batch Gradient Descent

- Can be faster than stochastic gradient descent. Instead of looking updating per one example of data, mini-batch descent udpates per b examples of data
- **General procedure** (example when $b = 10$):
  1. Set batch size b (e.g. say $b = 10$, on a dataset of $m = 1000$)
  2. Run gradient descent on loop which updates per b batches of data $x^{(i)}$ for $i = 1, 11, 21, 31, ..., 991$

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)} - y^{(k)}))x_j^{(k)}$$