Machine Learning (Stanford)

Tanner Prestegard

Course taken from 10/5/2015 - 12/27/2015

Introduction

Supervised learning

- Say you have data which gives the price and square footage of several houses.
- You want to predict the price given some square footage as an input.
- You can fit any type of function to the data: linear, quadratic, etc.
- Supervised learning: using a dataset where the "right" answers are known.
- Regression problem: predict a continuous valued output.
- Classification problem: predict for a problem with a discrete output.
  - Also want to estimate the probability associated with the prediction.
- Most interesting machine learning algorithms can deal with an infinite number of features!
  - Requires a support vector machine - we will talk about this later in the course.

Unsupervised learning

- We are given data where we don't know the "right answer." The actual data is not classified as true or false.
- The question is: here is the dataset, can you find some structure in the data?
- Example: clustering algorithm. Used in Google news to group similar stories together.

    - In this method, the algorithm assigns group labels to different clusters.

- Other examples: social network analysis, organization of computer clusters, market segmentation, astronomical data analysis.
- Example: cocktail party problem - useful for separating highly correlated audio tracks into independent tracks.
    - Cocktail party problem algorithm: `[W,s,v] = svd((repmat(sum(x.*x,1),size(x,1),1).*x)*` (singular value decomposition)

Linear regression with one variable

Model and cost function

- Regression problem: predict a (continuous output).
- General notation for this course:
    - m: number of examples in the training dataset.
    - x: input variables/features.
    - y: output variable/target.
    - *(x,y)*: denotes a single training example.
- General flow:
    - Training set -> learning algorithm -> hypothesis function.
    - The hypothesis function takes input (x) and predicts the outcome (y). It maps from x's to y's.
- How do we represent the hypothesis h?
    - $h_\theta\left(x\right) = \theta_0 + \theta_1 x$ for univariate linear regression, or linear regression with one variable.
    - The θ's are the parameters of the model.
- How do we choose the parameters of the model?
    - Find θ_0 and θ_1 such that we minimize the sum of the squared errors: $\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$.
    - Factor of $\frac{1}{2m}$ doesn't affect parameter values and will make later math a bit easier.
    - This also defined as the cost function $J\left(\theta_0, \theta_1\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$.

        - There are other cost functions that may work as well, but the squared error cost function is the most commonly used one for linear regression.
    - Can be useful to plot cost function in terms of the parameters θ_0 and θ_1.

- If we assume an intercept of 0 ($\theta_0 = 0$), the value of θ_1 that minimizes the cost function is $\theta_1 = \frac{\sum_i x^{(i)} y^{(i)}}{\sum_i x^{2(i)}}$
- For two parameters,
    - $\theta_1 = \frac{\sum_{i=1}^m x^{(i)} y^{(i)} - \bar{x}\bar{y}}{\sum_{i=1}^m x^{(i)2} - \bar{x}^2}$
    - $\theta_0 = \bar{y} - \theta_1 \bar{x}$

Gradient descent

- Have some cost function $J(\theta_0, \theta_1)$ that we want to minimize (find $\min\limits_{\theta_0, \theta_1} J(\theta_0, \theta_1)$
    - Outline:
        - 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 minimum.
    - Definition of gradient descent algorithm: `repeat until convergence` `{` $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ `}.`
        - :=: assignment operator.
        - α: learning rate. Controls step size.
    - When actually programming this, make sure you change the parameters at the same time and then update them at the same time! Otherwise you may incorrectly use new values of the parameters to calculate the cost function.
- Don't need to adjust α over time because the derivative term will get smaller as it approaches a local minimum.
- Applying gradient descent to our simple cost function:
    - $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^m \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)}\right)^2$.
    - For $j = 0$: $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m \left(h\left(x^{(i)}\right) - y^{(i)}\right)$
    - For $j = 1$: $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m \left(h\left(x^{(i)}\right) - y^{(i)}\right) x^{(i)}$
- Batch gradient descent: used in machine learning, each step of gradient descent uses all of the training examples.

Linear regression with multiple variables

- Also called multivariate linear regression.
- We now have multiple features (or predictors) that we want to use to develop a hypothesis function and make a prediction.
- Notation:
  - n: number of features.
  - $x^{(i)}$: input (features) of ith training example. This is a vector of features in the ith observation.
  - $x_j^{(i)}$: value of feature j in ith training example.
- Hypothesis function now has a new form:
  - $h_\theta\left(x\right) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$
  - For convenience of notation, we can define $x_0^{(i)} = 1$ so that $h_\theta\left(x\right) = \sum_{i=1}^{n} \theta_i x_i$
  - We can also use linear algebra:
    - $x = [x_0, x_1, ..., x_n] \in \mathbb{R}_{n+1}$ (row vector)
    - $\theta = [\theta_0, \theta_1, ..., \theta_n] \in \mathbb{R}_{n+1}$ (row vector)
    - Then $h_\theta\left(x\right) = \theta x^T$.

Gradient descent with multiple variables

- Write set of parameters as θ and cost function as $J\left(\theta\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$.
- Gradient descent: `repeat {` $\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J\left(\theta\right)$ `}` (simultaneously update for each $j = 0, 1, ..., n$)
  - $\theta_j := \theta_j - \alpha\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$
- Tips for gradient descent:
  - Feature scaling - make sure that different features taken on similar ranges of values.
    - Ideally, all features will be in the range $-1 \leq x \leq 1$.
  - Mean normalization: normalize a feature to have a mean of zero. When doing this, you just replace a feature x_i with $x_i - \mu_i$.
    - Do not apply this to $x_0 = 1$.
    - Can also normalize by standard deviation: $x_i \rightarrow \frac{x_i - \mu_i}{s_i}$.
- Making sure that gradient descent is working properly:

- Plot $\min_{\theta} J(\theta)$ vs. number of iterations. It should decrease after every iteration for sufficiently small α (can be shown mathematically).
- If α is too large, you may continue overshooting the minimum and never be able to actually reach it.
- If α is too small, convergence will happen, but very slowly.

- Automatic convergence tests: algorithms that tell you whether gradient descent has converged:
  - Example: declare convergence if $J(\theta)$ decreases by less than some value (like 10^{-3}) in one iteration.

Polynomial regression

- Can create new features from existing ones.
  - Example: area = frontage * depth.
- Polynomial model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ for a quadratic function.
  - Important to use feature scaling since when you square or cube features, the range becomes much larger.

Normal equation

- Method to solve for θ analytically.
- How to go about it:
  - $\frac{\partial}{\partial \theta_j} J(\theta) = ... = 0$, then solve for every θ_j.
- Linear algebra method:
  - Put all features into a matrix: $X = \begin{matrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{matrix}$ (including a column of ones for θ_0).
  - Put all y_js into a vector: $Y = \begin{matrix} 460 \\ 232 \\ 315 \\ 178 \end{matrix}$.
  - Then, $\theta = \left(X^T X\right)^{-1} X^T Y$.
  - In Matlab/Octave: `pinv(X'*X)*X'*Y`
  - Feature scaling is not needed for the normal equation method.

- Normal equation vs. gradient descent
  - Pros:
    - Normal equation: no need to choose α, no need to iterate.
    - Gradient descent: works well even when the number of features is large.
  - Cons:
    - Normal equation: slow if number of features is very large, need to compute $\left(X^T X\right)^{-1}$
    - Gradient descent: need to choose α, needs many iterations.
- Normal equation non-invertibility: what if $\left(X^T X\right)$ is non-invertible? Also known as "singular" or "degenerate."
  - Can occur when there are redundant features (linearly dependent) or if there are too many features.
    - Possible solution: delete some features or use regularization.
  - If you use `pinv` in Matlab/Octave instead of `inv`, it should handle singular cases for you.

Logistic Regression

Classification and Representation

- Classification: putting things into discrete categories. Binary classification is when there are only two categories.
- Linear regression is not usually a very good solution for classification problems. Examples where we can clearly use some sort of threshold for classification make this clear.
- Logistic regression has the property that the predictions of the hypothesis function are always between 0 and 1: $0 \leq h_\theta(x) \leq 1$.
- For linear regression, $h_\theta(x) = \theta^T x$.
- For logistic regression, $h_\theta(x) = g\left(\theta^T x\right)$, where $g(z)$ is the sigmoid or logistic function, $g(z) = \frac{1}{1+e^{-z}}$
  - Thus, $h_\theta(x) = \frac{1}{1+\exp(-\theta^T x)}$.
  - As before, we need to fit the parameters θ to our data.
- Interpretation of hypothesis output:
  - $h_\theta(x)$: estimated probability that $y = 1$ (i.e., positive result) on input x; i.e. $h_\theta(x) = p(y = 1|x;\theta)$.

- Decision boundary:
  - Suppose we predict that $y = 1$ when $h_\theta(x) \geq 0.5$ and $y = 0$ when $h_\theta < 0.5$.
  - We see that $g(z)$ is ≥ 0.5 when $z \geq 0$, thus $h_\theta(x) \geq 0.5$ whenever $\theta^T x \geq 0$.
  - This essentially defines a threshold for making predictions with our model.
  - The decision boundary is a property of the hypothesis and its parameters, not a property of the dataset.
- Non-linear decision boundaries:
  - Can use higher-order polynomials terms in logistic regression.
  - Example: $h_\theta(x) = g\left(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 + \theta_6 x_1^3 + ...\right)$

Logistic Regression Model

- Cost function:
  - Linear regression: $J(\theta) = \frac{1}{m}\sum_{i=1}^m \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = \sum_{i=1}^m \text{cost}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$
  - Logistic regression: $J(\theta) = \frac{1}{m}\sum_{i=1}^m \text{cost}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$
    - What cost function here should we use here?
    - We can't use the same thing that we used for linear regression because it will be a non-convex function - thus, gradient descent will not necessarily converge to the global minimum.
    - Instead, we use $\text{cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$
    - If $y = 1$: if $h_\theta(x) = 1$, the cost is 0, but as $h_\theta(x) \to 0$, the cost goes to ∞.
    - If $y = 0$: if $h_\theta(x) = 0$, the cost is 0, but as $h_\theta(x) \to 1$, the cost goes to ∞.
- Simplified way of writing the cost function:
  - $\text{cost}(h_\theta(x), y) = -y\log(h_\theta(x)) - (1 - y)\log(1 - h_\theta(x))$
  - Note: y is always equal to 0 or 1.
  - So the full cost function is $J(\theta) = -\frac{1}{m}\sum_{i=1}^m y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)$.
- Gradient descent: we do the same thing as before, we just have a new cost function.
  - $\theta_j := \theta_j - \alpha\sum_{i=1}^m \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$

  - This is the same as for linear regression, although our hypothesis function has changed.
  - Make sure to simultaneously update all θ_j!

- Advanced optimization:
  - We have some cost function $J(\theta)$ and we want the set of parametersθ that gives the minimum of $J(\theta)$.
  - Given θ, we have code that can compute $J(\theta)$ and $\frac{\partial}{\partial\theta_j}J(\theta)$.
  - Optimization algorithms:
    * Gradient descent
    * Conjugate gradient
    * BFGS
    * L-BFGS
  - Advantages: no need to manually pick the learning rate α, often faster than gradient descent.
  - Disadvantages: more complex, can cause implementations to be error prone.
  - Can use `fminunc` in Matlab/Octave.
    * Define a function which returns the cost and gradient.
    * Example:

```
% function
function [jVal, gradient] = costFunction(theta);

jVal = (theta(1)-5)^2 + (theta(2)-5)^2;
gradient(1) = 2*(theta(1)-5);
gradient(2) = 2*(theta(2)-5);

% Setup
options = optimset('GradObj','on','MaxIter','100');
initialTheta = zeros(2,1);

% Call fminunc
[optTheta, functionVal, exitFlag] = ...
        fminunc(@costFunction, initialTheta, options);
```

Multi-class classification

- Examples of this type of problem:
  - Email tagging into different folders.
  - Medical diagnosis.

  - Weather category.

- One-vs-all classification:
  - Example: a training dataset with three classes.
  - First: is it in class 1, or class 2/3?
  - Second: is it in class 2, or class 1/3?
  - Third: is it in class 3, or class 1/2?
  - Use standard logistic regression for each step.
  - Each gives a hypothesis: $h_\theta^{(i)}(x) = P(y = i|x;\theta)$, $(i = 1, 2, 3)$.
  - The hypothesis tells us the probability that y is in class i.
  - To make a prediction on a new input x, pick the class i that has the maximum probability.

Regularization and overfitting

- Overfitting: similar number of parameters to data points. This allows the model to fit the training dataset very well, however, it will not make accurate predictions for new examples. Also called "high variance."
  - Occurs when you have a lot of features and not a lot of training data.
  - Underfitting may occur when our model has too few parameters to properly model the data. In this situation, the model has "high bias."
- Options for dealing with overfitting:
  - Reduce number of features: can manually select which features to keep, or use a model selection algorithm to determine this (will be covered later in the course).
  - Regularization: keep all of the features, but reduce the magnitude/-values of the parameters θ_j. Works well when we have a lot of features, each of which contributes a little bit to predicting y.

Regularization

- Suppose we penalize and make θ_3, θ_4 very small. We do this by adjusting our cost function by adding in terms like $1000\theta_3^2 + 1000\theta_4^2$.
  - This will give values of θ_3 and θ_4 which are close to zero when the cost function is small.
- General idea behind regularization:
  - Small values for parameters $\theta_0, \theta_1, ..., \theta_n$.

    * Simpler hypothesis, less prone to overfitting.

- Example: housing price prediction.
  - Features: $x_1, x_2, ..., x_{100}$
  - Parameters: $\theta_0, \theta_1, \theta_2, ..., \theta_{100}$
  - We don't know which parameters to try to shrink *a priori*.
  - Take our cost function $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$ and add a new term to shrink *all* of our parameters except θ_0.
    * $J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right]$
    * The first part in brackets captures the goal of fitting the data well.
    * The second part captures the goal of keeping the parameters θ small.
    * λ is the regularization parameter and controls the trade-off between these two goals.
    * If λ is too large, we w
    * on't even fit the training dataset well. All of the parameters θ will go to zero except θ_0, so we will be fitting a flat line to our data.

Regularized linear regression

- Recall that we don't want to shrink θ_0.
- Now, our gradient descent algorithm looks like:
  - $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_0^{(i)}$
  - $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} + \frac{\lambda}{m}\theta_j$ for $j = 1, ..., n$.
  - Can re-write this as $\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$ for $j = 1, ..., n$.
    * The term $1 - \alpha \frac{\lambda}{m}$ is usually slightly less than 1 (assuming that your learning rate is small). This term shrinks your θ_j, then the other term is just the original gradient descent term.
- Normal equation
  - Before, we had $\theta = \left(X^T X\right)^{-1} X^T Y$.
  - With our new, regularized cost function, you can derive the minimum of the cost function by taking derivatives with respect to each parameter, as before.

  - Now, we get $\theta = \left(X^T X + \lambda Z\right)^{-1} X^T Y$, where Z is the identity matrix, except the top left entry is 0 instead of 1 (because we don't regularize θ_0).

- Non-invertibility
  - Suppose we have fewer examples than features ($m \leq n$).
  - $\theta = \left(X^T X\right)^{-1} X^T Y$ will give a strange answer since $X^T X$ is non-invertible/singular (make sure to use `pinv` instead of `inv`).
  - However, if $\lambda > 0$, we can prove that the matrix $X^T X + \lambda Z$ is invertible!

Regularized logistic regression

- To modify our cost function to use regularization, we add a term $\frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$.
- The full cost function is $J\left(\theta\right) = -\left[\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$
- Now, our gradient descent algorithm looks like:
  - $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_0^{(i)}$
  - $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} + \frac{\lambda}{m} \theta_j$ for $j = 1, ..., n$.
  - Can re-write this as $\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$ for $j = 1, ..., n$.
  - Recall that although this looks the same as for regularized linear regression, the hypothesis function is different for logistic regression.
- Advanced optimization methods:

```
% Define a function to compute the cost.
function [jVal, gradient] = costFunction(theta)
        jVal = %code to compute J(theta);
        gradient(1) = %code to compute d/dtheta_0 J(theta);
        gradient(2) = %code to compute d/dtheta_1 J(theta);
        ...
        gradient(n+1) = %code to compute d/dtheta_n J(theta);

% then, we pass this function to fminunc as before.
```

Neural networks: representation

Motivation

- Used for solving complex, non-linear hypotheses.

- For many predictors, including even quadratic terms can require a huge number of terms (grows roughly as n^2).
  - One option is to include only a subset of higher-order terms, but it will probably underfit the data.
- Neural networks: algorithms that try to mimic the brain.
- Recent resurgence: state-of-the-art technique for many computational applications.
- Computationally more expensive.

Neural networks

- Developed as simulating networks of neurons in the brain.
- Neuron structure:
  - Dendrites: “input wires,” receive inputs from other locations.
  - Axon: “output wire,” used to send signals to other neurons.
- Neuron model: logistic unit.
  - We use a model of what a neuron does: it takes some number of inputs $(x_1, x_2, ..., x_n)$, does some computation, and sends an output $h_\theta\left(x\right) = \frac{1}{1+e^{-\theta^T x}}$.
    - This is a sigmoid (logistic) activation function.
    - The parameters θ are sometimes referred to as “weights.”
  - x_0 is sometimes referred to as the “bias unit.”
- Neural network: divide into layers.
  - Layer 1: “input layer,” made up of input features.
  - Layer 2: some set of neurons, called “hidden layers.” Can be several hidden layers.
  - Final layer: “output layer,” outputs final value computed by a hypothesis.
- Notation:
  - $a_i^{(j)}$: “activation” of unit i in layer j.
  - $\Theta^{(j)}$: matrix of weights controlling function mapping from layer j to layer $j+1$.
  - If a network has s_j units in layer j and s_{j+1}units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

- Example: $a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right)$
- Can write $z_1^{(2)} = \Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3$; i.e. a row of Θ.

• Vectorized implementation:
  - $z^{(2)} = \Theta^{(1)}x = \Theta^{(1)}a^{(1)}$
  - $a^{(2)} = g\left(z^{(2)}\right)$ (here $z^{(2)}$ is a column vector of $z_1^{(2)}, ..., z_3^{(2)}$. Note that $g(z)$ is applied element-wise.
  - Add $a_0^{(2)} = 1$.
  - $z^{(3)} = \Theta^{(2)}a^{(2)}$
  - $h_\Theta(x) = a^{(3)} = g\left(z^{(3)}\right)$.

• We can think of the way a neural network works as though it is "learning its own features."
  - The function mapping $a^{(1)}$ to $a^{(2)}$ is what changes the input features to new features which go into the hidden layers.

• You can have neural networks with other types of layouts.
  - The layout is referred to as the "architechture."

Applications

- Can set up a neural network which acts as a logical filter (AND, OR, XOR, NOT, etc.).
- Interesting because sometimes these functions (XOR, for example) require non-linear decision boundaries.
- Multi-class classification:
  - Build a neural network with as many outputs as there are classes.
  - Then $h_\Theta(x)$ is a vector, and we take the largest entry to be the predicted class.

Neural networks: learning

Cost function and backpropagation

- L: number of layers in the network.
- s_l: number of units (not counting bias unit) in layer l.
- Binary classification:

  - Labels y are either 0 or 1.
  - One output unit ($K = 1$).

- Multi-class classification:
  - K classes, $y \in R^K$.
  - K output units.

- Cost function:
  - Logistic regression: $J(\theta) = \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$
  - Neural network: $h_\Theta(x) \in R^K$ (K-dimensional vector), $\left(h_\Theta(x)\right)_i =$ ith output.
    - $J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\Theta\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right)\log\left(1 - \left(h_\Theta\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2$

- Gradient computation
  - One training example (x, y).
    - Forward propagation:
      - · $a^{(1)} = x$.
      - · $a^{(2)} = g\left(z^{(2)}\right) = g\left(\Theta^{(1)}a^{(1)}\right)$ (add $a_0^{(2)}$)
      - · $a^{(3)} = g\left(z^{(3)}\right) = g\left(\Theta^{(2)}a^{(2)}\right)$ (add $a_0^{(3)}$)
      - · $a^{(4)} = h_\Theta(x) = g\left(z^{(4)}\right) = g\left(\Theta^{(3)}a^{(3)}\right)$
    - Then, we use backpropagation:
      - · Intuition: $\delta_j^{(l)} =$ "error" of node j in layer l.
      - · For each output unit (layer $L = 4$), $\delta_j^{(4)} = a_j^{(4)} - y_j$. (difference between the hypothesis and the actual classification)
      - · Vectorized version: $\delta^{(4)} = a^{(4)} - y$, each of these is a vector with number of elements equal to the number of output units K.
      - · $\delta^{(3)} = \left(\Theta^{(3)}\right)^T \delta^{(4)} .* g'\left(z^{(3)}\right)$
      - · $\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'\left(z^{(2)}\right)$
      - · No $\delta^{(1)}$ for this example.
      - · Notes: $g'\left(z^{(i)}\right) = a^{(i)} .* \left(1 - a^{(i)}\right) = g\left(z^{(i)}\right) .* \left(1 - g\left(z^{(i)}\right)\right)$, and $.*$ indicates element-wise multiplication.
      - · It is possible to prove that the partial derivative terms we want are given by $\frac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)}\delta_i^{(l+1)}$ (without regularization).

- Backpropagation algorithm:
  - Training set $\left\{\left(x^{(1)}, y^{(1)}\right), ..., \left(x^{(m)}, y^{(m)}\right)\right\}$
  - Set $\Delta_{ij}^{(l)} = 0$ for all l, i, j.
  - For $i = 1 : m$
    * 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 $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$.
    * $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
  - Vectorized form: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \left(a^{(l)}\right)^T$
  - After the for loop, we compute:
    * $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$, if $j \neq 0$.
    * $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$, if $j = 0$.
  - Can show that $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$, so we can use this in gradient descent or other optimization algorithms.

- What is backpropagation doing?
  - $\delta_j^{(l)}$: "error" of cost for $a_j^{(l)}$ (unit j in layer l). I.e.,
  - $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$ for $j \geq 0$, where $\text{cost}(i) = y^{(i)} \log h_\Theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log h_\Theta\left(x^{(i)}\right)$.
  - These are a measure of how much we would like to change the neural network's weights so as to change the intermediate values of the computation, and thus change the final outputs, and the cost.

Backpropagation in practice

- Unrolling parameters from matrices into vectors:

```
% Let's say you defined a function for computing the cost function
% Here jVal is a number, and theta and grad are vectors in R^{n+1}
function [jVal, grad] = costFunction(theta)
% You will pass it to fminunc as:
optTheta = fminunc(@costFunction, initialTheta, options)
% For a neural network, your parameters Theta are matrices.
% The gradients D are also matrices.
```

- Example: a neural net with 10 inputs, a hidden layer of 10 nodes, and 1 output. I.e., $s_1 = 10$, $s_2 = 10$, $s_3 = 1$.

  - $\Theta^{(1)} \in R^{10\times 11}$, $\Theta^{(2)} \in R^{10\times 11}$, $\Theta^{(3)} \in R^{1\times 11}$
  - $D^{(1)} \in R^{10\times 11}$, $D^{(2)} \in R^{10\times 11}$, $D^{(3)} \in R^{1\times 11}$
  - To unroll them, you can do: `thetaVec = [Theta1(:); Theta2(:); Theta3(:)];`, and the same for D.
  - To go back to the matrix representation, you can do `Theta1 = reshape(thetaVec(1:110),10,1` `Theta2 = reshape(thetaVec(111:220),10,11); Theta3 = reshape(thetaVec(221:231),1`

- Example: have initial parameters $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$
  - Unroll to get `initialTheta` to pass to `fminunc(@costFunction, initialTheta, options)`
  - Then we change our cost function:
    * `function [jVal, gradientVec] = costFunction(thetaVec)`
    * From `thetaVec`, get $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$.
    * Use forward propagation and backpropagation to compute $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ and $J(\Theta)$.
    * Unroll $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ to get `gradientVec`.

- Downside of backpropagation - easy to have many subtle bugs in your implementation.
  - However, there is something called "gradient checking" which allows us to catch almost all of these bugs.

- Gradient checking
  - Numerical estimate of gradients: $\frac{\partial}{\partial\theta}J(\theta) \approx \frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$. Usually, we take ϵ to be very small, something like 10^{-4}.
  - Implementation in Matlab: `gradApprox = (J(theta + eps) - J(theta - eps))/(2*eps);`

- Gradient checking with a parameter vector $\theta \in R^n$ (θ is the unrolled version of $\Theta^{(1)}, \Theta^{(2)}$, etc.).
  - $\theta = [\theta_1, \theta_2, ..., \theta_n]$
  - $\frac{\partial}{\partial\theta_i}J(\theta) \approx \frac{J(\theta_1,...,\theta_i+\epsilon,...,\theta_n)-J(\theta_1,...,\theta_i-\epsilon,...,\theta_n)}{2\epsilon}$
  - These equations give a way to numerically approximate the gradient with respect to any of your parameters.
  - Implementation in Matlab:

```
for i=1:n
        thetaPlus = theta;
        thetaPlus(i) = thetaPlus(i) + epsilon;
        thetaMinus = theta;
        thetaMinus(i) = thetaMinus(i) - epsilon;
        gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon);
```

  - Implemenation:
    - Implement backpropagation to compute `DVec`.
    - Implement numerical gradient checking to compute `gradApprox`.
    - Make sure they give similar values for a few test cases.
    - Turn off gradient checking; just use backpropagation for future runs.
  - Important: make sure to disable your gradient checking code before training your classifier, otherwise your code will be very slow.

- Random initialization: we need to pick some initial value for Θ.
  - For gradient descent, we previously set `initialTheta = zeros(n,1)`.
    - This doesn't work for training a neural network; it will result in all nodes being the same and all deltas being the same. All of the partial derivatives will be equal to each other, as well.
  - So, for a neural network, we need to use a random choice for `initialTheta` between $[-\epsilon, \epsilon]$.
    - `Theta1 = rand(10,11)*(2*initEps) - initEps;`
    - `Theta2 = rand(1,11)*(2*initEps) - initEps;`

- Putting it all together:
  - First, pick some network architecture, or connectivity pattern between neurons.
    - Number of input units should be the number of features.
    - Number of output units should be the number of possible classifications.
    - Reasonable default: one hidden layer, or if more than one hidden layer, have the same number of units in each hidden layer (usually the more, the better, although more units requires more computational power).
  - Next, we need to train a neural network.
    - First, randomly initialize the weights.
    - Then, implement forward propagation to get $h_\Theta\left(x^{(i)}\right)$ for any $x^{(i)}$.
    - Implement code to compute the cost function $J(\Theta)$.
    - Implement backpropagation to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$.
      - Usually requires a for loop.
  - Use gradient checking to compare the partial derivatives computed using backpropagation to those computed numerically for a few examples.

- Disable gradient checking code if everything seems OK.
- Use gradient descent or advanced optimization methods with back-propagation to try to minimize $J(\Theta)$ as a function of the parameters Θ.
  - Note: $J(\Theta)$ is not necessarily convex, so it is possible for the algorithms to get stuck in local optima.

Advice for applying machine learning

Evaluating a learning algorithm

- How do you decide what the most promising avenues to explore are?
  - Suppose you have implemented regularized linear regression to predict housing prices.
  - However, when you test your hypothesis on a new set of houses, you find that it makes unacceptably large errors in its predictions.
  - What should you try next?
    - Get more training examples, although this doesn't always help.
    - Spend time trying to carefully select a smaller set of features.
    - Try getting additional features.
    - Try adding polynomial features.
    - Try increasing or decreasing the regularization parameter λ.
- Machine learning diagnostics: tests you can run to gain insight into what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance.
  - Diagnostics can take time to implement, but doing so can be a very good use of your time.
- Evaluating your hypothesis:
  - Split your data into two portions: training and testing datasets.
    - Typical split is about 70% training, 30% testing.
    - m_{test}: number of test examples.
    - $\left(x_{test}^{(i)}, y_{test}^{(i)}\right)$: ith example from test dataset.
    - Should usually divide dataset by randomly choosing samples since there may sometimes be some order to the samples.
  - Procedure:
    - Learn parameters θ from training data by minimizing training error $J(\theta)$.

    * Compute test set error $J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} \left(h_\theta \left(x_{test}^{(i)} \right) - y_{test}^{(i)} \right)^2$ (this example is for linear regression).
    * Alternate definition of test set error: 0/1 misclassification error.
      · $err(h_\theta(x), y) = \begin{cases} 1, h_\theta(x) \geq 0.5, y = 0 | h_\theta(x) < 0.5, y = 1 \\ 0, \text{otherwise} \end{cases}$
      · Test error: $\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err\left(h_\theta \left(x^{(i)} \right), y^{(i)} \right)$

- When parameters are fit using a training set, the error of the parameters as measured on that data (training set error) is likely to be lower than when you measure the error on another, independent dataset (generalization error).

- Model selection

  - Say you have a choice of models with different degrees of polynomials,.
  - It's as if you have another parameter, d, which is the highest degree of the polynomial used in the model.
  - You could fit each model to your training data and get a set of parameters for each model $\theta^{(d)}$.
  - Then you can look at the test set and compute the cost for each model: $J_{test}\left(\theta^{(d)}\right)$.
  - Then, rind the model with the lowest cost. For this example, assume it's the model with parameters $\theta^{(5)}$.
  - Question: how well does the model generalize? Could report test set error $J_{test}\left(\theta^{(5)}\right)$.
  - Problem: $J_{test}\left(\theta^{(5)}\right)$ is likely to be optimistic because our extra parameter d is fit to our test data set.
  - Thus, we now need to split our dataset into three pieces:
    * Training dataset ($\approx 60\%$).
    * Cross-validation (CV) dataset, sometimes also called validation dataset ($\approx 20\%$). m_{cv} is the number of CV examples.
    * Test dataset ($\approx 20\%$).
  - Can calculate the CV and test errors just as we have for the training dataset.
  - For the best model selection strategy, calculate $J_{cv}\left(\theta^{(d)}\right)$. Then you can safely use $J_{test}\left(\theta^{(d)}\right)$ for the best model to determine the generalization error for that model.

Bias and variance

- High bias is equivalent to underfitting the data.
- High variance is equivalent to overfitting the data.
- Idea for diagnosing bias/variance:
  - Plot training error vs. degree of polynomial d.
  - Plot cross-validation error vs. degree of polynomial d.
  - Training error will probably decrease with d.
  - Cross-validation error will most likely reach a minimum at a certain value of d, then start increasing again.
  - High bias: high training error and high CV error. $J_{train}(\theta) \approx J_{cv}(\theta)$.
  - High variance: low training error, high CV error.
- Regularization can help prevent overfitting, but how does it affect bias and variance?
  - Suppose we are fitting a high-order polynomial.
  - How do we choose the regularization parameter λ to minimize bias and variance?
    - Try a range of λ values.
    - For each λ value, we find the set of parameters θ which minimizes $J(\theta)$.
    - Then, apply this set of parameters to the cross-validation dataset and pick whichever λ ends up giving the lowest $J_{cv}(\theta)$.
    - Then, apply this to the test set.
    - This is model selection applied to the regularization parameter λ.
- Learning curves
  - Plotting $J_{train}(\theta)$ and $J_{cv}(\theta)$ versus m (training set size).
  - Lets you see how your algorithm changes with more training examples.
  - Average training error will grow with m (easy to fit only a few data points, harder to fit many). Will usually start to plateau at a certain point.
  - Cross-validation error will tend to decrease as m increases.
  - High bias (underfitting) case:
    - Cross-validation error will initially decrease with m, but will plateau relatively quickly.

    * Training error will start small and increase with m, and will eventually plateau and end up very close to the cross-validation error.
    * If a learning algorithm has high bias, getting more training data will not help very much on its own.
  - High variance (overfitting) case:
    * Training error will start small and increase very slowly with m.
    * Cross-validation error will start high and decrease very slowly with m.
    * There will be a big gap between the training and cross-validation error.
    * If a learning algorithm has high variance, getting more training data is likely to help.

- Debugging a learning algorithm
  - Getting more training examples will help with high variance cases.
  - Trying a smaller set of features will also help with high variance cases.
  - Adding features will help with high bias problems.
  - Adding polynomial features will also help with high bias problems.
  - Decreasing λ will help fix high bias problems.
  - Increasing λ will help fix high variance problems.

- Neural networks and overfitting.
  - "Small" neural network: fewer parameters, computationally cheaper, more prone to underfitting.
  - "Large" neural network: more parameters, computationally more expensive, more prone to overfitting. Use regularization to address overfitting.

Machine learning system design

Building a spam classifier

- Supervised learning: how do we want to define x, the set of features of the e-mail?
  - Could choose $\approx$100 words indicative of spam/not spam. (e.g., deal, buy, discount, your name, etc.)
  - Given an e-mail, parse it and record whether each word occurs or not.

  - Another option is to look through a training set of spam e-mail and pick out the most frequently occurring words automatically, rather than choosing the features manually.

- How to spend your time to make your algorithm have low error?
  - Collect lots of data.
  - Develop sophisticated features based on e-mail routing information from e-mail header.
  - Develop sophisticated features for message body.
    - Should "deal" and "Dealer" be treated as the same word?
    - Features about punctuation (spam uses more exclamation points).
  - Detect misspellings (m0rtgage, w4tches, etc.).

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

- Example:
  - $m_{CV} = 500$ examples in CV data set.
  - Algorithm misclassifies 100 e-mails.
  - Manually examine the 100 errors and categorize them baesd on:
    - What type of e-mail it is.
    - What cues (features) you think would have helped the algorithm classify them correctly.

- The importance of numerical evaluation
  - Should discount/discounts/discounted/discounting be treated as the same word?
  - In natural language processing, can use "stemming" software (e.g. "porter stemmer").
  - Difficult example: universe/university.
  - Error analysis may not be helpful for deciding whether stemming would be useful. Best approach is to just try it and see if it works (i.e., is the cross-validation error lower with or without stemming?).
  - Need numerical evaluation (e.g., cross-validation error) of algorithm's performance with/without stemming.
  - Distinguish between upper/lower case.

Handling skewed data

- Example: cancer classification.
  - Train logistic regression model $h_\theta(x)$; $y = 1$ means that the patient has cancer.
  - Find that we get 1% error on test set (99% correct diagnoses).
  - But if only 0.50% of patients have cancer, our result is not particularly good.
  - This is an example of skewed classes, where we have a lot more of one class than the other.
  - In this case, classification accuracy is not very useful.
- Precision/recall evaluation metric:
  - Take $y = 1$ in presence of rare class that we want to detect.
  - True positive: predicted class is 1 and actual class is 1.
  - True negative: predicted class is 0 and actual class is 0.
  - False positive: predicted class is 1 and actual class is 0.
  - False negative: predicted class is 0 and actual class is 1.
  - Precision: of all patients where we predicted $y = 1$, what fraction actually has cancer?
    - Equal to number of true positives divided by the total number of predicted positives: $N_{TP}/(N_{TP} + N_{FP})$.
  - Recall: of all patients that actually have cancer, what fraction did we correctly detect as having cancer?
    - Equal to number of true positives divided by the number of actual positives: $N_{TP}/(N_{TP} + N_{FN})$.
    - Also called **sensitivity**.
  - This may be more useful in general for skewed classes than classification accuracy would be.
- Example: suppose we want to predict $y = 1$ (cancer) only if very confident.
  - Could increase our prediction threshold:
    - Predict 1 if $h_\theta(x) \geq 0.7$.
    - Predict 0 if $h_\theta(x) < 0.7$.
  - This will increase our precision - more of our predictions will be true.
  - It will also lower the recall because we predict fewer people will have cancer.

- Example: suppose we want to avoid missing too many cases of cancer (avoid false negatives).
  - Could decrease our prediction threshold:
    - Predict 1 if $h_\theta(x) \geq 0.3$.
    - Predict 0 if $h_\theta(x) < 0.3$
  - This will give higher recall - we will flag almost all patients who do have cancer.
  - But the precision will be lower - a higher fraction of the patients who we predict to have cancer will not actually have cancer.
- Main point: you can vary the value of your threshold and plot a curve which shows a precision vs. recall threshold.
- Question: is there a way to choose this threshold optimally?
  - F_1 score: tells us how to compare/weight precision and recall. (also just called F score)
  - $F_1 = 2\frac{PR}{P+R}$. Like taking an average, but gives the lower score more weight.

Data for machine learning

- Designing a high-accuracy learning system
  - Example: classify between confusable words (to, two, too; then, than).
    - Algorithms: Perceptron (logistic regression), Winnow, memory-based, naive Bayes.
  - Test all algorithms and vary training set size - as training set size increases, at a certain point, most algorithms end up getting very similar results.
  - Saying: "It's not who has the best algorithm that wins, it's who has the most data."
- Large data rationale
  - Assume features $x \in R^{n+1}$ has sufficient information to predict y accurately.
  - Example: for breakfast I ate _____ eggs. (algorithm should put in *two*)
  - Counterexample: predict housing price from only size (square feet) and no other features.
  - Useful test: given the input x, can a human expert confidently predict y?

      · Can help determine if we have enough information for the learning algorithm to do a good job.
    * Use a learning algorithm with many parameters (e.g., logistic/-linear regression with many features or a neural network with many hidden units).
      · This should give low bias and $J_{train}(\theta)$ should be small.
    * Use a very large training set (unlikely to overfit).
      · If $J_{train}(\theta) \approx J_{test}(\theta)$, then $J_{test}(\theta)$ will most likely be small.

Support vector machines

Large margin classification

- Alternative view of logistic regression:
  - Think about what we want logistic regression to do.
    * If $y = 1$, we want $h_\theta(x) \approx 1$, $\theta^T x \gg 0$.
    * If $y = 0$, we want $h_\theta(x) \approx 0$, $\theta^T x \ll 0$.
  - Cost of example: $-(y \log h_\theta(x) + (1-y)\log(1-h_\theta(x)))$
    * If $y = 1$, the cost is $-\log(h_\theta(x)) = -\log\left(\frac{1}{1+\exp(-\theta^T x)}\right)$; as $\theta^T x$ gets large, the cost becomes small.
  - To make a support vector machine, we use a "simpler" cost function: instead we use two lines.
    * Cost = 0 for $z = \theta^T x > 1$ (for $y = 1$).
    * Cost = linear function which follows curve for $z < 1$ (for $y = 1$).
    * Do the same thing for the case where $y = 0$.
    * This will give us a computational advantage.
- Support vector machine cost function: $J(\theta) = C\sum_{i=1}^{m} y^{(i)}\text{cost}_1\left(\theta^T x^{(i)}\right) + \left(1-y^{(i)}\right)\text{cost}_0\left(\theta^T x^{(i)}\right) + \frac{1}{2}\sum_{i=0}^{n}\theta_j^2$
  - We ignore the factor of m that was in previous cost functions because it's just a normalization factor.
  - We also pull out $\lambda$from the regularization term. $C = \frac{1}{\lambda}$ is now the coefficient of the first term.
- We could have set our threshold at $\theta^T x \geq 0$ for $y = 1$ (and similarly for $y = 0$), but putting at $\theta^T x \geq 1$ means a stronger prediction.
- Consider a case where we set C to be a very large value, like 10^5.

- When minimizing the cost function, it will try to set the first term to be 0.
- This requires we find $\theta^T x^{(i)} \geq 0$ for $y^{(i)} = 1$ and $\theta^T x^{(i)} \leq -1$ for $y^{(i)} = 0$.
- End result is a very interesting decision boundary. For a linearly separable case, the decision boundary will be close to what appears optimal by eye.
  - This "distance" between the two classes is the *margin*, the SVM tries to separate the two classes with as large of a margin as possible.

- Large margin classifiers can be sensitive to outliers.
  - Can use an intermediate value for C instead of a very large one in order to reduce the effect of outliers.

- Vector stuff
  - Magnitude: $|u| = \sqrt{u_1^2 + u_2^2}$
  - Inner product: $u \cdot v = \sum_i u_i v_i = u^T v = v^T u = p\,|u|$. This is equal to the length of the projection of v onto u (p) multiplied by the magnitude of u.

- SVM decision boundary
  - $\min \frac{1}{2} \sum_{j=1}^n \theta_j^2$, such that $\theta^T x^{(i)} \geq 1$ if $y^{(i)} = 1$ and $\theta^T x^{(i)} \leq -1$ if $y^{(i)} = 0$.
  - Can think of $\sum_{j=1}^n \theta_j^2 = \left(\sqrt{\sum_{j=1}^n \theta_j^2}\right)^2$, which is like the magnitude of the parameter vector theta:$|\theta|^2$.
    - Note: we make a simplification and take θ_0 to be 0 (this means that the decision boundary will pass through the origin).
  - $\theta^T x = p^{(i)} \cdot |\theta|$

Kernels

- Question: is there a different or better way to choose features?
- Example: given x, compute new features based on proximity to landmarks l^1, l^2, l^3 (points randomly chosen to be "important" or "landmarks").
  - Given x: $f_1 = \textit{similarity}\left(x, l^1\right) = \exp\left(-\frac{|x-l^1|^2}{2\sigma^2}\right)$, similarly for f_2, f_3.

- Here, the formula for similarity is called the *kernel*. In this case, we use a Gaussian kernel. Can also be written as $k\left(x, l^{(i)}\right)$.

- Kernels and similarity
  - $f_1 = similarity\left(x, l^1\right) = \exp\left(-\frac{\left|x-l^1\right|^2}{2\sigma^2}\right)$.
  - If $x \approx l^1$, then $f_1 \approx 1$.
  - If x is far from l^1, then $f_1 \approx 0$.
  - This feature f_1 measures how close x is to the first landmark, l^1.

- σ is a parameter of the kernel.
  - Larger σ means a more spread out Gaussian distribution.

- How do we choose the landmarks?
  - Start by choosing the positions of actual samples as landmarks (1 landmark per sample).
  - This says your features will measure how close an example is to something from the training set.
  - For each new example, there will be m features, one for each landmark.

- Hypothesis: given x, compute features $f \in R^{m+1}$, predict $y = 1$ if $\theta^T f \geq 0$. ($\theta \in R^{m+1}$)

- Training: minimize cost function $\min C \sum_{i=1}^{m} y^{(i)} cost_1\left(\theta^T f^{(i)}\right) + \left(1 - y^{(i)}\right) cost_0\left(\theta^T f^{(i)}\right) + \frac{1}{2}\sum_{j=1}^{n} \theta_j^2$ (here, $n = m$, since the number of features is equal to the number of samples).
  - Can write last term as $\sum_{j=1}^{n} \theta_j^2 = \theta^T\theta$ (ignoring θ_0).
  - Most SVMs replace this with $\theta^T M \theta$, where M is a matrix generally chosen based on your kernel. This is a mathematical detail that allows the algorithm to run much more efficiently.

- Can apply kernel method for things like logistic regression, but it doesn't work quite as well. SVMs use some computational tricks that logistic regression doesn't.

- Don't recommend to write software to minimize the SVM cost function, use stuff that has already been developed.

- SVM parameters:
  - $C = 1/\lambda$

    * Large C: lower bias, higher variance.
    * Small C: higher bias, lower variance.
  - σ^2
    * Large σ^2: features f_i vary more smoothly. Higher bias, lower variance.

Using an SVM

- Use SVM software package (liblinear, libsvm, etc.) to solve for parameters θ.
- Need to specify:
  - Choice of parameter C.
  - Choice of kernel or similarity function.
    * No kernel (linear kernel): predict $y = 1$ if $\theta^T x \geq 0$.
      · Useful if number of features is large and number of examples is small.
    * Gaussian kernel: $f_i = \exp\left(-\frac{|x-l^{(i)}|^2}{2\sigma^2}\right)$, where $l^{(i)} = x^{(i)}$.
      · Need to choose σ^2.
      · Useful when number of features is small and/or number of examples is large.
      · Note: need to perform feature scaling before using the Gaussian kernel if your features are on very different scales, this is because σ^2 is defined the same way for all features.
    * Polynomial kernel: $k(x,l) = \left(x^T l + C\right)^d$, C is a constant and d is the degree of the polynomial. Usually worse than a Gaussian kernel but can be useful in some situations.
    * Others: string kernel, chi-square kernel, histogram intersection kernel, etc.
    * Not all similarity functions make valid kernels. They technically need to satisfy Mercer's theorem to make sure that SVM packages' optimizations run correctly and do not diverge.
  - You may need to implement the kernel or similarity function on your own.
- Multi-class classification
  - Many SVM packages already have this functionality build in.
  - Otherwise, use the one-vs-all method. Train K SVMs, one to distinguish $y = i$ from the rest for $i = 1, 2, ..., K$, get $\theta^{(1)}, \theta^{(2)}, ..., \theta^{(K)}$, and pick the class i with the largest $\left(\theta^{(i)}\right)^T x$.

- Logistic regression vs. SVMs
  - n: number of features.
  - m: number of training examples.
  - If n is large relative to m, use logistic regression, or an SVM with a linear kernel.
  - If n is small (1 - 1000 or so) and m is "intermediate" (10 - 10000 or so), use an SVM with a Gaussian kernel.
  - If n is small and m is large (50000+), create/add more features, then use logistic regression or an SVM with a linear kernel.
  - A neural network will likely work well for most of these cases, but may be slower to train.

Unsupervised learning

Clustering

- Unsupervised learning - our data does not have any labels associated with it!
  - Training set: $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$
- We want the algorithm to find some structure in our data, often by clustering.
- Uses: market segmentation, social network analysis, organization of computing clusters, astronomical data analysis.
- K-means clustering: most widely used clustering algorithm.
  - An iterative algorithm:
    * First: assign data to a cluster based on cluster centroid locations.
    * Second: move cluster centroids to average position of data points in each cluster.
    * Repeat until the cluster centroids stop moving.
- K-means inputs:
  - K: number of clusters you want to find in the data (will talk more later about how to choose K).
  - Training set $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$, where $x^{(i)} \in R^n$ (drop $x_0 = 1$ convention).
- K-means algorithm:

  - Randomly initialize K cluster centroids $\mu_1, \mu_2, ..., \mu_k \in R^n$.
  - Repeat:

```
for i=1:m
        ci = index (from 1 to K) of cluster centroid closest to xi
for k=1:K
        mu_k = average of points assigned to cluster k
```

  - If a cluster centroid has no points assigned to it, usually you just want to eliminate it - then you'll have $K-1$ clusters. Otherwise, if you really need K clusters, you can randomly re-initialize that cluster centroid.

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

- K-means optimization objective:
  - $J\left(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K\right) = \frac{1}{m}\sum_{i=1}^{m}\left|x^{(i)} - \mu_{c^{(i)}}\right|^2$
  - I.e., we want to minimize the total squared distances between data points and their corresponding cluster centroid.
  - We minimize the cost function by moving the cluster centroids.
  - The cost function should never increase with more iterations.
  - Note: the cost function is also called the *distortion* function for k-means clustering.

- Random initialization of cluster centroids
  - Number of cluster centroids K should be less than number of training examples m.
  - Can end up with different solutions depending on your initialization; sometimes the algorithm can get caught in local optima.
  - One method for initialization: randomly pick K training examples, set $\mu_1, ..., \mu_K$ equal to these training examples.
  - Can also repeat k-means clustering many times with many different random initializations, then pick the case with the overall lowest cost.
    * If you're using a small number of clusters, $2 \lessapprox K \lessapprox 10$, many initializations can help make sure you find a better optimum.

        * For $K > 10$, your first initialization will probably be pretty good already, more initializations will help some, but not very much.

- Choosing the number of clusters
    - Not always clear how many clusters that actually are in the data.
    - Elbow method:
        * Run k-means with several different values of K and plot the cost function J vs. K.
        * You may see curve that looks like a bent arm. Try to choose the value of K that is at the "elbow."
        * Not used that often - you usually don't see a nice, clear elbow; it's usually a smooth curve.
    - Another method: evaluate k-means based on a metric for how well it performs for its actual, downstream purpose.
        * Example: picking T-shirt sizes (S, M, L vs. XS, S, M, L, XL). Think about how number of sizes might impact sales and manufacturing costs.

Dimensionality reduction

Motivation

- Can be useful when you have some features which are partly or fully correlated.
- Want to define a new feature which encapsulates one or more other features.
- Example: reduce data from 2D to 1D:
    - Originally have two examples $x^{(1)} \in R^2$ and $x^{(2)} \in R^2$.
    - If the two features are highly correlated, dimensionality reduction may let us convert to $z^{(1)} \in R$ and $z^{(2)} \in R$.
- Dimensionality reduction may allow our algorithms to run more quickly - this is one of the main reasons we pursue it.
- Dimensionality reduction also helps us to visualize the data better.
    - Hard to plot and visualize 50-dimensional data, but 2D or 3D data is easier.

Principal component analysis (PCA)

- Tries to find a lower-dimensional surface to project the data onto such that it minimizes the orthogonal projection error (least-squares error).
- Should perform feature scaling before doing PCA.
- Problem formulation for reducing from 2D to 1D: find a direction (a vector $u^{(1)} \in R$) onto which to project the data so as to minimize the projection error.
  - Can generalize this to going from n dimensions to k dimensions.
- How does PCA relate to linear regression?
  - They are not the same!
  - Linear regression: minimize vertical distance between points and regression line. Trying to use x_1 to predict x_2.
  - PCA: minimize *orthogonal* distance between points and projection.
- Implementation:
  - Perform feature scaling and mean normalization: $\mu_j = \sum_{i=1}^{m} x_j^{(i)}$; replace each $x_j^{(i)}$ with $x_j - \mu_j$. Then scale by standard deviation.
  - Example: reduce data from n dimensions to k dimensions.
  - Compute covariance matrix: $\Sigma = \frac{1}{m} \sum_{i=1}^{n} \left(x^{(i)}\right) \left(x^{(i)}\right)^T$
  - Compute eigenvectors of matrix Σ: `[U,S,V] = svd(Sigma);` (singular value decomposition). Can also use `eig(Sigma)`.
    * Σ is $n \times n$.
    * U is $n \times n$; each column is a vector corresponding to a principal component.
      · Take the first k columns of U. Call this U_{reduce}.
    * $z = U_{reduce}^T \cdot x$.
- Reconstruction from compressed representation: given a point z in the compressed space, can we map it back to the full representation x?
  - $x_{approx} = U_{reduce} \cdot z$
  - Variance retained: $\sum_{i=1}^{k} S_{ii} / \sum_{i=1}^{n} S_{ii} \leq 1$.
- Choosing the number of principal components to use:
  - PCA tries to minimize the average squared projection error: $\frac{1}{m} \sum_{i=1}^{m} \left| x^{(i)} - x_{approx}^{(i)} \right|^2$.
  - Total variation in the data: $\frac{1}{m} \sum_{i=1}^{m} \left| x^{(i)} \right|^2$.

- Typically, choose k so that the average squared projection error divided by the total variation is ≤ 0.01. "99% of variance is retained."
- Can use second output of `svd()`: this is a diagonal matrix; the quantity we want to calculate is $1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \leq 0.01$ OR $\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \geq 0.99$.

- Advice for applying PCA to speed up supervised learning:
  - Example: 100 by 100 images.
  - Extract inputs: unlabeled dataset $x^{(1)}, ..., x^{(m)} \in R^{10000}$
  - Use PCA to get $z^{(1)}, ..., z^{(m)} \in R^{1000}$.
  - Pair zs with ys.
  - Note: the mapping from x to z should be defined by running PCA on the training set. Then you can use the same mapping for the cross-validation and test sets.

- Bad use of PCA: to prevent overfitting.
  - The idea: if we use PCA to go to $k < n$ features, it should be less likely to overfit.
  - Sometimes, this might work OK, but regularization is a much better way to address overfitting.

- PCA is sometimes used where it shouldn't be.
  - Example: design of ML system.
    - Get training set.
    - Run PCA to reduce training set in dimension.
    - Train logistic regression.
    - Test on test set: map test set to lower dimension, use hypothesis function to test algorithm.
  - But why not just try it without PCA?
  - Should first try it without PCA; if it doesn't do what you want, then implement PCA and see what the results are.

Anomaly detection

Density estimation

- What is anomaly detection?
  - Trying to decide if a new example fits in with the other examples.
  - Is the new example x_{test} anomalous?

- Can define a model $p(x)$ which gives the probability that an example is OK.
  - We also define a threshold ϵ, where if $p(x) \geq \epsilon$, we mark the example as OK.

- Uses:
  - Fraud detection:
    - $x^{(i)}$: features of user i's activity.
    - Model $p(x)$ from data.
    - Identify unusual users by checking which have $p(x) < \epsilon$.
  - Manufacturing example: monitoring computers in a data center.
    - $x^{(i)}$: features of machine i. Examples: memory use, number of disk accesses, CPU load, network traffic, etc.
    - Model $p(x)$ from data and identify unusual machines.

- Gaussian distribution: $p\left(x;\mu,\sigma^2\right) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$.
  - Can estimate for a data set:
    - $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
    - $\sigma^2 = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)}-\mu\right)^2$
      - In machine learning, people tend to use $1/m$ instead of $1/(m-1)$ (not sure why....).

- Developing an anomaly detection algorithm
  - Training set with m examples: $\left\{x^{(1)}, ..., x^{(m)}\right\}$, where each example is $x \in R^n$.
  - Assume each feature x_i is Gaussian distributed with mean μ_i and variance σ_i^2 - this corresponds to assuming the features are all independent.
    - Should we do PCA to make this assumption more robust?
  - Model $p(x) = p\left(x_1;\mu_1,\sigma_1^2\right)p\left(x_2;\mu_2,\sigma_2^2\right)...p\left(x_n;\mu_n,\sigma_n^2\right) = \prod_{j=1}^{n} p\left(x_j;\mu_j,\sigma_j^2\right)$.

- Procedure:
  - Choose features that you think might be indicative of anomalous examples.
  - Fit parameters μ_j and σ_j^2.
  - Given a new example, compute $p(x)$.
  - Mark as anomalous if $p(x) < \epsilon$.

- Reasonable choice for ϵ: something like 0.02?

Building an anomaly detection system

- When developing a learning algorithm, making decisions is much easier if we have a way of evaluating the algorithm.
  - Assume we have some labeled data with anomalous and non-anomalous examples.
  - Training set: large number of examples, no anomalous cases.
  - Cross-validation and test sets should include anomalous examples.
- Algorithm evaluation
  - Fit model $p(x)$ on training set.
  - Make predictions on a test set.
  - Possible evaluation metrics:
    - True positive, false positive, false negative, true negative.
    - Precision/recall.
    - F_1 score.
    - Classification accuracy is not good because there are a small number of anomalies; an algorithm that always predicts $y = 0$ will do well.
  - Can also use cross-validation set to choose parameter ϵ.
- Anomaly detection vs. supervised learning
  - Anomaly detection
    - Very small number of positive examples ($y = 1$). 0-20 positive examples is common.
    - Large number of negative ($y = 0$) examples.
    - Many different types of anomalies, hard for any algorithm to learn from positive examples what the anomalies might look like; future anomalies may look very different from examples seen so far.
    - Examples: fraud detection, manufacturing, monitoring machines in a data center.
  - Supervised learning
    - Large number of positive and negative examples.
    - 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.
    - Examples: e-mail spam classification, weather prediction, cancer classification.

- Choosing features
  - First, plot histogram of the features, check for significant non-Gaussianity. Usually OK if it's somewhat close to Gaussian.
    - Can also test if the log of a feature is Gaussian; if so, replace that feature with its log.
    - Other transforms: $x \to x^{\alpha}$, usually $\alpha < 1$.
- Want $p(x)$ to be large for normal examples and small for anomalous examples.
  - Common problem: $p(x)$ is comparable (say, both large) for normal and anomalous examples.

Multivariate Gaussian distribution

- In some cases, we need to account for correlations between variables - our assumption of independence may not be valid.
- Then, we can't model $p(x)$ assuming the variables are independent. We need to do it all in one go.
  - Parameters: $\mu \in R^n$, $\Sigma \in R^{n \times n}$ (covariance matrix).
  - $p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} \det(\Sigma)} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$
- Anomaly detection with the multivariate Gaussian
  - First, fit model $p(x)$by setting:
    - $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
    - $\Sigma = \frac{1}{m} \sum_{i=1}^{m} \left(x^{(i)} - \mu\right)\left(x^{(i)} - \mu\right)^T$
  - Given a new example x, compute $p(x)$ and flag as anomalous if $p(x) < \epsilon$.
- Our original model (assuming independent features) corresponds to a multivariate Gaussian where the contours of the multivariate PDF are "axis-aligned."
  - This is equivalent to saying that the covariance matrix is diagonal.
- When should we use the independent model and when should we use the multivariate model?
  - Independent:
    - Manually create features to capture anomalies where x_1, x_2 take unusual combinations of values.

    * Computationally cheaper, scales better to large n.
    * Works OK even if m (training set size) is small.
  - Multivariate:
    * Automatically captures correlations between features.
    * Computationally more expensive because we need to invert the $n \times n$ matrix Σ.
    * Must have $m > n$, or else Σ is not invertible. (preferable to have $m \gg n$)

- If you fit a multivariate model and find that Σ is singular or non-invertible:
  - Check if $m > n$.
  - You may have highly redundant (linearly dependent) features ($x_3 = Cx_4$, $x_7 = x_8 + x_9$, etc.).

Recommender systems

- When you buy a product online, most websites automatically recommend other products that you may like. Recommender systems look at patterns of activity between different users and different products to produce these recommendations.

Predicting movie ratings

- User rates movies using 0 to 5 stars.
-

Movie	**Alice (1)**	**Bob (2)**	**Carol (3)**	**Dave (4)**
Love at last	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	4	?

- Notation:
  - n_u: number of users (4 in this example).
  - n_m: number of movies (5 in this example).
  - $r(i, j)$: 1 if user j has rated movie i.
  - $y^{(i,j)}$: rating given by user j to movie i (defined only if $r(i, j) = 1$).
  - $\theta^{(j)}$: parameter vector for user j.
  - $x^{(i)}$: feature vector for movie i.
  - $m^{(j)}$: number of movies rated by user j.

- Goal: based on a user's rating for some movies, predict how they would rate other movies.
- What if we had features for each movies?
  - In this example, we would want features which measure the degree of romance and the degree of action in each movie.
  -

Movie	x_1 (romance)	x_2 (action)
Love at last	0.9	0
Romance forever	1.0	0.01
Cute puppies of love	0.99	0
Nonstop car chases	0.1	1.0
Swords vs. karate	0	0.9

- For each user j, learn a parameter $\theta^{(j)} \in R^3$. Predict user j as rating movie i with $\left(\theta^{(j)}\right)^T x^{(i)}$.
- Problem formulation:
  - For user j and movie i, predicted rating: $\left(\theta^{(j)}\right)^T \left(x^{(i)}\right)$
  - To learn $\theta^{(j)}$ (parameter for user j): $\min_{\theta^{(j)}} \frac{1}{2m^{(j)}} \sum_{i:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T \left(x^{(i)}\right) - y^{(i,j)}\right)^2 + \frac{\lambda}{2m^{(j)}} \sum_{k=1}^{n} \left(\theta_k^{(j)}\right)^2$
  - To learn θ for all users: $J\left(\theta^{(1)}, ..., \theta^{(n_u)}\right) \min_{\theta^{(1)},...,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T \left(x^{(i)}\right) - y^{(i,j)}\right.$ $\frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^{(j)}\right)^2$
  - Gradient descent update:
    * $\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)}\right) x_k^{(i)}$ for $k = 0$.
    * $\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)}\right) x_k^{(i)} + \lambda \theta_k^{(j)}\right)$ for $k \neq 0$.
      · The term multiplying α is equal to $\frac{\partial}{\partial \theta_k^{(j)}} J\left(\theta^{(1)}, ..., \theta^{(n_u)}\right)$.

Collaborative filtering

- Sometimes, it might be hard to get features. Example: previous study with movie ratings - lots of work to get people to watch movies and rate the amount of romance and action. What if we let the people rating the movies define their own θ which describes how much they like each type of movie?
  - I.e., someone who likes romantic movies and hates action movies would have a parameter vector $\theta^{(i)} = [0, 5, 0]$.

- Optimization algorithm to learn the features:
  - Given $\theta^{(1)}, ..., \theta^{(n_u)}$, to learn $x^{(i)}$: $\min_{x^{(i)}} \frac{1}{2} \sum_{j:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} \left(x_k^{(i)} \right)^2$.
  - Given $\theta^{(1)}, ..., \theta^{(n_u)}$, to learn $x^{(1)}, ..., x^{(n_m)}$: $\min_{x^{(1)},...,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j}$
    $\frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)} \right)^2$.
- Also, if we know the features $x^{(1)}, ..., x^{(n_m)}$, can estimate the parameter $\theta^{(1)}, ..., \theta^{(n_u)}$ (this is what we did before).
- Can also iterate: $\theta \to x \to \theta \to x$, and so on. Will allow you to estimate parameters better.
- Full algorithm description:
  - Objective: given $x^{(1)}, ...x^{(n_m)}$, estimate $\theta^{(1)}, ..., \theta^{(n_u)}$ (or vice versa).
  - Actually, we can do both at the same time!
  - Minimizing $x^{(1)}, ...x^{(n_m)}$ and $\theta^{(1)}, ..., \theta^{(n_u)}$ simultaneously:
    * $J\left(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}\right) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^{(j)} \right)^2$
    * Here we minimize J over all features $x^{(1)}, ..., x^{(n_m)}$ and parameters $\theta^{(1)}, ..., \theta^{(n_u)}$.
    * Note that we don't minimize over x_0 because we are now learning all of the features
- Procedure:
  - Initialize $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}$ to small, random values.
  - Minimize $J\left(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}\right)$ using gradient descent or an advanced optimization algorithm.
    * $x_k^{(i)} := x_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)} \right) \theta_k^{(j)} + \lambda x_k^{(i)} \right)$
    * $\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)} \right) x_k^{(i)} + \lambda \theta_k^{(j)} \right)$
    * Term in parentheses is equal to $\frac{\partial}{\partial x_k^{(i)}} J\left(...\right)$.
  - For a user with parameters θ and a movie with learned features x, predict a star rating of $\theta^T x$.

Low rank matrix factorization

- Put all ratings from all users in a matrix.
- Using movies example: $n_u = 5$, $n_m = 4$.
  - $Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}$
  - Predicted ratings: $Y_{ij} = \left(\theta^{(j)}\right)^T x^{(i)}$. We can calculate this element-by-element, or we can vectorize it!
  - Define a vector of feature vectors $X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ . \\ . \\ \left(x^{(n_m)}\right)^T \end{bmatrix}$ and a vector of parameter vectors $\Theta = \begin{bmatrix} \left(\Theta^{(1)}\right)^T \\ \left(\Theta^{(2)}\right)^T \\ . \\ . \\ \left(\Theta^{(n_u)}\right)^T \end{bmatrix}$; then we can calculate $Y = X\Theta^T$.
  - This is called low rank matrix factorization.
- Can use learned features to find related movies.
  - For each product i, we learn a feature vector $x^{(i)} \in R^n$.
  - How to find movies j related to movie i? (can be used to make recommendations)
    * Find a movie j such that the distance between the two feature vectors is small.
    * $\left|x^{(i)} - x^{(j)}\right| =$ small.
- What if we have a user who hasn't rated any movies?
  - First term in cost function plays no role since $r\,(i, j)$ is 0 for all movies.
  - The only term that affects θ for the new user is the regularization term, so we'll get a parameter vector of all zeros.
  - Thus, we'll predict ratings of 0 for all movies for the new user - this doesn't seem very useful!

- To fix this, we use mean normalization:
  - Subtract the mean rating for each movie so that each movie has an average rating of zero.
  - Adjust your Y matrix to account for this and use this matrix to do collaborative filtering.
  - For user j on movie i, predict $\left(\theta^{(j)}\right)^T \left(x^{(i)}\right) + \mu_i$.
  - This results in the predicted ratings for the new user being the mean ratings for each movie.

Large scale machine learning

Gradient descent with large datasets

- Example: a training set size: $m = 1e8$.
  - A single step of gradient descent requires a sum over all of these examples!
  - Question: why not just test on a subset $m = 1000$ before investing the effort into training on the full dataset?
  - Can test if more data is useful by plotting $J_{CV}(\theta)$ and $J_{train}(\theta)$ vs. m and see how they compare as m increases.
- Linear regression with gradient descent:
  - $h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$
  - $J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$
  - Gradient descent: repeat $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$ for every $j = 0, ..., n$.
    - If m is large, this step takes a long time!
    - This is also called *batch* gradient descent.
- Stochastic gradient descent:
  - $\text{cost}\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right) = \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$
    - Measures how well the cost function does on a single example.
  - $J_{train}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{cost}\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$
    - This is the average of the cost function over all training examples.
  - Procedure:
    - Randomly shuffle the dataset.
    - Repeat for $i = 1, ..., m$:

        - $(\theta_j := \theta_j - \alpha \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$, for $j = 0, ..., n)$
    - How many times to repeat this whole thing? Usually 1 to 10 passes is most common, but it depends on the size of your training set.
    - Note: the term multiplying α is equal to $\frac{\partial}{\partial \theta_j} \text{cost}\left(\theta\left(x^{(i)}, y^{(i)}\right)\right)$.
  - Main difference between this and batch gradient descent is that we take a small step after looking at each example rather than having to sum over all examples before taking a step.
  - However, we end up taking a less "direct" route to the minimum of the cost function.

- Mini-batch gradient descent
  - In between batch and stochastic gradient descent.
  - We use b examples in each iteration, where b is the mini-batch size.
    - b is typically chosen to be something like 10, may be between 2 and 100.
  - Procedure:
    - Get b randomly chosen examples from the training set.
    - Run batch gradient descent once on these examples.
    - Repeat for each
  - Written out for $b = 10$, $m = 1000$:
    - Repeat for $i = 1, 11, 21, 31, ..., 991$:
      - $\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=1}^{i+9} \left(h_\theta\left(x^{(k)}\right) - y^{(k)}\right) x_j^{(k)}$ (for every $j = 0, ..., n$)
    - Usually want to use 1-10 passes.
  - Why is it useful?
    - Only outperforms stochastic gradient descent if we have a good vectorized implementation for handling the b examples on each iteration.
  - Disadvantage: new parameter b which you have to study.

- Checking for convergence:
  - Batch gradient descent
    - Plot $J_{train}(\theta)$ as a function of the number of iterations of gradient descent.
    - This should decrease as the number of iterations increases.
  - Stochastic gradient descent

        * During learning, compute cost $\left(\theta, \left(x^{(i)}, y^{(i)}\right)\right)$ before updating θ using $\left(x^{(i)}, y^{(i)}\right)$.
        * Every 1000 iterations or so, average the cost function over the last 1000 examples and plot it.
        * If the algorithm appears to be diverging, use a smaller α.
        * If the cost function seems to be flat vs. the number of iterations, you may need to change something - learning rate, features, etc.
        * Can slowly decrease α over time if we want θ to converge.
            · Ex: $\alpha = \frac{const1}{iterationNumber+const2}$.

Online learning

- Online learning algorithm: problems where we have a continuous stream of data coming in and we want an algorithm to learn from that.
- Example: shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service ($y = 1$) and sometimes not ($y = 0$).
    - Features x capture properties of the user, the origin and destination, and the asking price.
    - We want to learn $p\left(y = 1|x;\theta\right)$ to optimize the price.
    - Procedure:
        * Repeat forever:
            · Get (x, y) corresponding to user.
            · Update parameters θ using just this example (x, y).
            · $\theta_j := \theta_j - \alpha\left(h_\theta\left(x\right) - y\right)x_j$ (for $j = 0, ..., n$) (this is for logistic regression)
    - After using each data point, we throw it out and never use it again.
    - This type of algorithm can adapt to changing user preferences.
- Another application - product search.
    - User searches for "Android phone 1080p camera".
    - Have 100 phones in store, will return 10 results.
    - Want to have a learning algorithm figure out which 10 phones we should show to the user based on their search query.
    - x: features of phone, how many words in user query match name of phone, how many words in user search query match phone description, etc.
    - $y = 1$ if user clicks on link, $y = 0$ otherwise.

  - Learn $p\left(y=1|x;\theta\right)$ and use this to show the user the 10 phones that they are most likely to click on.

- Other examples: choosing special offers to show the user, customized selection of news articles, product recommendation, etc.

MapReduce and data parallelism

- Example with $m = 400$ data points:
  - Batch gradient descent: $\theta_j := \theta_j - \alpha\frac{1}{400}\sum_{i=1}^{400}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_j^{(i)}$
  - Machine 1: Use $\left(x^{(1)},y^{(1)}\right),...,\left(x^{(100)},y^{(100)}\right)$ and compute the summation for the first 100 examples.
    - $temp_j^{(1)} = \sum_{i=1}^{100}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_j^{(i)}$
  - Machine 2: same thing with examples 101 to 200.
  - Machine 3: same thing with examples 201 to 300.
  - Machine 4: same thing with examples 301 to 400.
  - After all of these computation are done, we send them to a master server, which combines the results and updates θ_j.
    - $\theta_j := \theta_j - \alpha\frac{1}{400}\left(temp_j^{(1)}+temp_j^{(2)}+temp_j^{(3)}+temp_j^{(4)}\right)$ (for $j = 0,...,n$).

- Many learning algorithms can be expressed as computing sums of functions over the training set.

- Example: advanced optimization with logistic regression.
  - We need:
    - $J_{train}\left(\theta\right) = -1\frac{1}{m}\sum_{i=1}^{m}y^{(i)}\log h_\theta\left(x^{(i)}\right)-\left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)$
    - $\frac{\partial}{\partial\theta_j}J_{train}\left(\theta\right) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)x_j^{(i)}$

- You can use MapReduce on computers with multiple cores as well!
  - No network latency for communication between computers.

Photo optical character recognition (OCR)

- Recognizing and reading text in digital images.

- Very useful to modularize the full procedure!

- Photo OCR pipeline overview:
  - Text detection: define a rectangle around the text.

  - Character segmentation: separate each character.
  - Character classification: classify each character as a letter.
  - May need a "correction" step. Example: "c1eaning" should be corrected to "cleaning".

- Sliding window: define a window size (82 by 50 pixels, for example) and slide the window around in your image, performing logistic regression to determine if what you are looking for (pedestrians, text, etc.) is contained in each window.
  - Use some step size of something like 5-10 pixels to slide the window after each iteration.
  - Can use larger window size to select parts of the image, then resize those to 82 by 50 pixels before passing to your logistic regression classifier.
- Text detection:
  - Want to define rectangles which contain text. It's tricky because different text blocks may have different heights, widths, and aspect ratios.
  - Get training examples of positive and negative cases.
  - Run a small window on the image (of the size that will detect single letters).
  - Then apply an expansion classifier: if a pixel is within X pixels of a pixel that is expected to have text, color that pixel white as well.
    - This helps us to define our bounding rectangles.
    - We define bounding rectangles around groups of white pixels which have reasonable aspect ratios.
- 1D sliding window for character segmentation
  - Positive examples should be focused on spaces between two characters.
  - Negative examples should have blanks and single letters.
  - We slide a window in one dimension through the text rectangle and let the learning algorithm determine where there are gaps.

Artificial data synthesis

- Using a small dataset, doing random operations to generate new data.
- Can take data you already have and distort it. Example: take a character and apply distortions or blurs to make the algorithm more robust.

- Can also generate completely new data yourself.
- Photo OCR: take random fonts and paste a few characters on a random background, applying distortions. Then include these new examples in your training data set.
- Usually, it doesn't help to add purely random/meaningless noise to your data. The distortion introduced should be representative of the type of distortions you might expect to see in your test data set.
- Discussion on getting more data:
  - Make sure you have a low bias classifier before expending the effort (plot learning curves, keep increasing the number of features or hidden units (for a neural network) until you have a low bias classifier).
  - Consider: how much work would it be to get 10 times as much data as we currently have?
    - Artificial data synthesis.
    - Collect/label it yourself.
    - "Crowdsourcing", e.g. Amazon Mechanical Turk

Ceiling analysis: which part of the pipeline to work on next?

- Ceiling analysis: estimating the error due to each component of your pipeline. What part of the pipeline should you spend the most time trying to improve?
- Example: image -> text detection -> character segmentation -> character recognition.
  - Overall system has accuracy 72%.
  - Manually label text in images to simulate 100% text detection accuracy.
  - Then check overall accuracy: 89%
  - Now manually segment character to simulate 100% character segmentation accuracy.
  - Then check overall accuracy: 90%.
  - Now manually perform character recognition to simulate 100% character recognition accuracy.
  - Then check overall accuracy: 100% (obviously).
  - This implies that improving the text detection could increase the overall accuracy by up to 17%, so this is the weakest link in your pipeline.

- Important to help you identify which are the most crucial parts of the pipeline to improve and where you should spend your time. Don't trust your gut feelings on this!