Machine Learning

Mueen Nawaz

January 3, 2015

Contents

1 Linear Regression

1.1 Definitions

- **Supervised Learning**: Informally it means you gave the algorithm a data set with the "correct" answers specified. There's also **unsupervised** learning as well as **reinforcement** learning.
- **Regression**: Predict continuous valued output.
- Notation:
  - m = Number of training examples.
  - x = Input—also called the input **feature**
  - y = Output—also called the **target**
  - $(x^{(i)}, y^{(i)})$ is the i th **training example**.
  - The full set of training examples is called the **training set**.
  - $\mathcal{X}$ is the space of inputs, $\mathcal{Y}$ the space of outputs.
  - $p(y|x;\theta)$ vs $p(y|x,\theta)$. In the former, θ is not a random variable, it is just a parameter. In the latter, θ *is* a random variable, and we're saying "the probability of y given x *and* θ". The two cases are differentiated with the use of a comma versus a semicolon.
- h is the **hypothesis**: It takes x as the input and computes y. Formally, $h : \mathcal{X} \mapsto \mathcal{Y}$
- **Linear regression** with one variable: $h(x) = \theta_0 + \theta_1 x$. Also called **univariate** linear regression.

- If you have multiple input variables (denoted by $x_j^{(i)}$ where the subscript denotes another feature of the input), then $h(x) = \sum_{i=0}^{n} \theta_i x_i = \Theta^T \mathbf{x}$. Note that $x_0 = 1$ always—it was introduced for notational convenience and is called the **intercept term**.
- θ_i's are called the **parameters**, or **weights**.
- Want to minimize $\min_{\theta_0,\theta_1} \frac{1}{2m} \sum_{i=1}^{m} \left(h(x^{(i)}) - y^{(i)}\right)^2$. The leading fraction is just to simplify the math later on.
- More general notation: $\min_{\theta_0,\theta_1} J(\theta_0, \theta_1)$ where J is the **cost function**.
  - In our case J is the squared error function.

1.2 Gradient Descent

- **Gradient Descent** algorithm: $\theta_j^{n+1} = \theta_j^n - \alpha \frac{\partial}{\partial \theta_j} J$ or $\Theta^{n+1} = \Theta^n - \alpha \nabla J$
  - α is the **learning rate**. It is always positive.
  - It controls the step size in the direction of maximum change. Too small and the convergence is slow. Too large and it will overshoot and possible diverge.
  - This "rule", when applied to the linear cost function, is also called the **LMS** (least mean squares) update, or the **Widrow-Hoff** learning rule. Note that the update is proportional to the error.
- For linear regression, the cost function is always convex. You're guaranteed to hit a global minimum (if you hit one). Why?
- **Batch** gradient descent: Each iteration uses all the training examples.
- There is also the **stochastic** gradient descent. Here we update Θ every time we come across an input $x^{(i)}$.
  - The difference between stochastic and batch is that the latter updates all the θ_j's with all the $x^{(i)}$'s (i.e. each time each θ_j is updated, we have to calculate over all inputs—then we calculate the new cost function). The stochastic approach iterates over the inputs. When it encounters an input, it updates all the parameters, and then calculates the cost function. Then it moves on to the next input.
  - Stochastic gradient can get closer to the minimum much more quickly but is not guaranteed to hit a minimum—it may oscillate around it. Why? This can be remedied by decreasing α as it approaches the minimum.
- For the simple case of linear regression, you can always use least squares method which is non-iterative. However, I think Andrew Ng claims that for large data sets, gradient descent is quicker?

1.2.1 Feature Scaling & Normalizing

- If the θ_j's are of widely different scales (e.g. one of them is over a 1000, and the other is between 0 and 1), you'll have convergence issues (or slow convergence).
- To get around this, scale all your θ_j's so that they are all around unity.
- It is also desirable to offset your features such that the mean is roughly 0. This way after scaling your features will (roughly) be between -1 and 1. Why is this important?
- More generally, let $x'_j = \frac{x_j - \mu}{s}$ where s is the range. One could also use the standard deviation.
- Don't get too dogmatic about these transformations.
- If you're solving directly using the normal equations (explained later), then scaling is unnecessary as no iteration is involved. {I strongly suspect scaling can be an issue even for direct approaches!}

1.2.2 Convergence Heuristics

- Plot $J(\Theta)$ vs the number of iterations (can do this while you're iterating—output $J(\Theta)$ to a file).
- Ideally we'd like to decrease fast and monotonically.
- One could use some automatic criterion for convergence (e.g. percent change in J). However, Andrew has not had good experience with it and often just looks at the plot and kills the iteration after he feels confident it has converged.
- If your J shoots up sharply with the number of iterations, reduce your α.
- Similarly if it oscillates, use a lower α.
- There should be some α_0 below which you always get a decrease. However, it's not obvious what that is and too low an α will lead to slow convergence.
- Andrew starts from a low $\alpha = 0.001$, and tries it. He increases it each time by a factor of 3 until he hits a sweet spot.

1.3 Alternatives to Gradient Descent

- Alternatives:
  - Conjugate Gradient
  - BFGS
  - L-BFGS

- Advantages to the above:
  - No need to pick α
  - Usually faster
- Disadvantages:
  - More complex
  - Don't write your own implementations.

1.4 Gradient Checking

- Many optimization algorithms require a gradient. If you supply one (and not an approximation), also write up an approximation routine.
- Start a run with gradient checking, and verify that it is approximately close to your claimed gradient.
- Then kill the run, disable the checking, and run the algorithm.

1.5 Matrix Derivatives

- Let $f : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$. Define the derivative of f with respect to matrix A as:

$$\nabla_A f(A) = \begin{pmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{pmatrix} \tag{1}$$

- $\operatorname{tr}(AB) = \operatorname{tr}(BA)$ assuming AB is square. This extends to a product of three or more matrices—cyclic rotation.
- $|A|$ is the determinant of A.
- $\nabla_A \operatorname{tr}(AB) = B^T$
- $\nabla_{A^T} f(A) = (\nabla_A f(A))^T$
- $\nabla_A \operatorname{tr}(ABA^TC) = CAB + C^TAB^T$ Show!
- $\nabla_A |A| = |A| \left(A^{-1}\right)^T$ assuming $|A|$ exists. {Show! To prove it, write A^{-1} in terms of its adjoint.}

1.6 Least Squares Revisited

- Given a training set, define the **design matrix** X to be the $m \times n$ matrix that contains the training examples' input values in its rows:

$$X = \begin{pmatrix} (x^{(1)})^T \\ \vdots \\ (x^{(m)})^T \end{pmatrix} \tag{2}$$

  This of course assumes n features.

- Let $\mathbf{y}$ be the m dimensional vector with the outputs.

- As $h_\theta(x^{(i)}) = \left(x^{(i)}\right)^T \Theta$, then $X\Theta$ has as its elements $h_\theta(x^{(i)})$ and $X\Theta - \mathbf{y}$ is the vector of errors.

- It is straightforward to show that $J(\Theta) = \frac{1}{2}(X\Theta - \mathbf{y})^T(X\Theta - \mathbf{y})$. Note that $z^T z$ is the sum of the square of the elements.

- Minimize J by calculating $\nabla_\Theta J(\Theta)$ and setting it to 0. The result is the **normal equation**: $X^T X\theta = X^T\mathbf{y}$. The solution is $\Theta = (X^T X)^{-1}X^T\mathbf{y}$.

- $X^T X$ can be singular, but that is rare. If it happens, use the pseudoinverse. If this happens:

  - Check that you don't have duplicate training examples. This can happen if you have linear dependence (e.g. same inputs but in different units).
  - Check that you don't have more features than the size of your data set. If you do, either delete some or use regularization (covered later).

- If you're solving using the normal equation (without iterating), then feature scaling is unnecessary. I have my doubts...

- The normal method doesn't scale well with respect to n. {Andrew's argument for this is the cost of inverting a matrix. However, one should not have to explicitly invert the matrix to solve a linear system! Is his argument valid if one simply does a simple solution without inversion?}

  - Andrew's heuristic. $n = 1000$ is OK. $n = 10^4$ is borderline. More than that and he'd use gradient descent.

1.7 Probabilistic Interpretaion

- We can assume $y^{(i)} = \Theta^T\mathbf{x}^{(i)} + \epsilon^{(i)}$ where $\epsilon^{(i)}$ is the error term.

- Assume all the error terms (across all i's) are independent and identically distributed Gaussians, and that the mean of the error is 0.

- It can be shown that the maximum likelihood function for the Θ that is the one that is the solution to the normal equations.
  - $L(\Theta; X, \mathbf{y}) = p(\mathbf{y}|X; \Theta)$
  - $L(\theta) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}; \theta)$. The product is because it is assumed each training example is independent of the others.
- This solution is independent of the σ of our distributions. Of course, it is assumed that they are identical.

1.8 Locally Weighted Linear Regression

- In the usual linear regression, we're trying to minimize $\sum_i \left(y^{(i)} - \Theta^T \mathbf{x}^{(i)}\right)^2$. A modification is to minimize $\sum_i w^{(i)} \left(y^{(i)} - \Theta^T \mathbf{x}^{(i)}\right)^2$. The **weights** are non-negative.
- Intuitively, the weights give some features more preference over others.
- A standard choice for the weights is $w^{(i)} = \exp\left(-\frac{\left(x^{(i)} - x\right)^2}{2\tau^2}\right)$.
  - Note that this makes the weight a function of x.
  - One benefit of this is that if x is close to one of the inputs in the training set, the weight is close to 1. This makes Θ ensure it's a good fit to the training sample.
  - τ controls how quickly the weight declines away from a training point. It is called the **bandwidth** parameter.
  - If x is actually $\mathbf{x}$, then the numerator is $(\mathbf{x}^{(i)} - \mathbf{x})^T(\mathbf{x}^{(i)} - \mathbf{x})$. Sometimes $(\mathbf{x}^{(i)} - \mathbf{x})^T\Sigma^{-1}(\mathbf{x}^{(i)} - \mathbf{x})$ is used for some choice of Σ
  - Similarities with the Gaussian (or probability in general) are superficial. Don't look for a deeper connection. One can even use weighting functions that integrate to infinity.
- This algorithm is an example of a **non-parametric** one. A **parametric** algorithm uses the training data to get the parameters, and once trained, we can discard the training data. In this algorithm the training data is part of the algorithm (at least if it depends on x).
- Every time you make a prediction, you need to retrain everything. Why?

1.9 Random Tips

- Make sure your hypothesis function's form makes sense.
  - Don't pick a quadratic hypothesis if when you extrapolate you get nonsensical values (e.g. a large enough input causes a decrease when you know from reality that never happens). Pick a cubic instead. {Although frankly most cubics will have a drop (two extrema).}

- Can convert a polynomial of one feature into a linear function of many features. Just let $x_2 = x^2$ and so on.

2 Classification

Classification is another task ML is used for. It is used for discrete outputs—usually of finite size and few in number.

- **Binary classification** is where the output takes only two values. Let it be 0 and 1.
- 0 is called the **negative class** and 1 the **positive class**. Sometimes they are denoted by – and +.
- Given $x^{(i)}$ the $y^{(i)}$ is called the **label** for the training example.
- The **decision boundary** is the "line" (or plane or surface) that separates the two classes. It is also called the **separating hyperplane**.
  - The boundary is a property of our hypothesis, not of the data set.

2.1 Logistic Regression

- Of course, one could try using a linear regression algorithm for a classification problem as it is simply a special case, but it's easy to construct problems where linear regression has poor performance.
  - An example is where an outlier in the training set will significantly alter your hypothesis (compared to when you didn't have an outlier).
- Try
$$h_\Theta(x) = g(\Theta^T \mathbf{x}) = \frac{1}{1 + e^{-\Theta^T \mathbf{x}}} \tag{3}$$
- $g(z)$ is called the **logistic** or **sigmoid** function.
- Insert plot here.
- $g(z)$ tends to 0 as $z \to -\infty$ and tends to 1 as $z \to \infty$. It's monotonic and smoothly goes to 1 (bounded between 0 and 1). This is important as it is a binary classification problem.
- Note that for this form of the hypothesis, the decision boundary is given by $z = \Theta^T \mathbf{x} = 0$. Examining this can give you insights.
- Can also use other functions that smoothly go from 0 to 1.
- Note that $g'(z) = g(z)(1 - g(z))$
- Using maximum likelihood we can compute the update rule. Let $P(y = 1|x; \theta) = h_\theta(x)$ and $P(y = 0|x; \theta) = 1 - h_\theta(x)$ for a given training example.

  - This can be written as $p(y|x;\theta) = (h_\theta(x))^y(1-h_\theta(x))^{1-y}$.
  - Then we want to *maximize* $\sum_{i=1}^m (y^{(i)} \log h(x^{(i)}) + (1-y^{(i)}) \log(1 - h(x^{(i)})))$
  - You can make your cost function to be the negative of the above. Divide by m for some kind of normalization.
  - This cost function makes sense if you consider its values near 1 and 0 (when correct or when wrong)
  - Had we used the same cost function as for linear regression, our cost function would be non-convex.

- Using maximum likelihood we get the update rule: $\theta_j^{n+1} = \theta_j^n + \alpha \sum_{i=1}^m \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$ This is the same as for linear regression—but our hypothesis is no longer linear.
- Note that this approach is essentially assuming a Bernoulli distribution.
- Note that this method does not output 0 or 1—it can output anything in between.

2.2 The Perceptron Learning Algorithm

- What if we want it to output either 0 or 1? Make $g(z)$ be the step function and you get the **perceptron learning algorithm** if we use the same update rule.
  - Does this update rule follow from maximum likelihood?

2.3 Another way to maximize the maximum likelihood function

- Generalization of Newton's Method to multiple dimensions: $\Theta^{n+1} = \Theta^n - H^{-1}\nabla_\Theta l(\Theta)$
  - H is the **Hessian** matrix. $H_{ij} = \frac{\partial^2}{\partial\theta_i \partial\theta_j}$
- Faster convergence than batch gradient descent.
- Each iteration is more expensive due to inverting a matrix. Not that much of a downside if the matrix is not large.
- **Fisher scoring** is when this method is used to the maximum likelihood function.

2.4 Classifying into k Categories

on-vs-all:

- Take one category, and do a logistic regression on it vs everything else.
- Do the same for all the other categories.
- Then when you have a data point, try all k categories and see which one fits it best.

3 Generalized Linear Models

3.1 The Exponential Family

- $p(y;\eta) = b(y)\exp\left(\eta^T T(y) - a(\eta)\right)$
  - η is called the **natural** or **canoncial** parameter.
  - $T(y)$ is the **sufficient statistic**. It will often simply be y.
  - $a(\eta)$ is the **log partition** function. $\exp(-a(\eta))$ is more or less a normalization constant s.t. the integral of p over the whole space is 1.
- Fixing T, a, b gives a *family* of distributions parametrized by η.
- Bernoulli and Gaussian distributions are just special cases.
  - Bernoulli: $b(y) = 1$, $\eta = \log(\phi/(1-\phi))$, $T(y) = y$, $a(\eta) = -\log(1-\phi)$
  - Gaussian: We're free to use any σ (it did not impact the linear regression) so for convenience we set it to 1. $\eta = \mu$, $T(y) = y$, $a(\eta) = \eta^2/2$, $b(y) = \left(1/\sqrt{2\pi}\right)\exp\left(-y^2/2\right)$
- Can also derive the multinomial, Poisson, gamma, exponential, beta, Dirichlet distributions from this.

3.2 Constructing GLMs

- GLM means **generalized linear models**
- 3 assumptions:
  1. $y|x;\theta$ is a member of some exponential family.
  2. Given x, the goal is to predict $T(y)$. This is normally y. In other words we'd like $h(x) = E[y|x]$. This is satisfied for both the logistic and the linear regression.
  3. η and $\mathbf{x}$ are linearly related: $\eta = \Theta^T\mathbf{x}$
- The third assumption is a design choice of ours.

3.2.1 Ordinary Least Squares

- This is a special case of the GLM family.
- Assume y is continuous, and the distribution of y given x follows a Gaussian where μ may be a function of x.
- From the previous section's "equivalence" of the exponential family and the Gaussian, we have $h_\theta(x) = \Theta^T \mathbf{x}$

3.2.2 Logistic Regression

- Assuming a Bernoulli distribution gives us $h_\theta(x) = 1/(1 + \exp(-\Theta^T \mathbf{x}))$
- $g(\eta) = E[T(y); \eta]$ is called the **canonical response function**.
- Its inverse is called the **canonical link function**.
- Thus for the Gaussian family g is just the identity function $g(z) = z$ and for the Bernoulli it is the logistic function.

3.2.3 Softmax Regression

- Assume we need to classify into k categories. We use a multinomial distribution.
- Let the output be from $\{1, 2, \cdots, k\}$
- Define $k-1$ parameters ϕ_i, which is the probability of getting i as the outcome. We have $k-1$ of them as we have the constraint that $\sum_{i=1}^{k} \phi_i = 1$
- Let:

$$T(i) = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \tag{4}$$

  where the i th row is 1 and $T(y) \in \mathbb{R}^{k-1}$
- For convenience, $T(k)$ is the vector of 0's.
- Notation. Let $1\{\cdot\}$ be the indicator function whose value is 1 if the argument is true and 0 if false.
  - Thus $(T(y))_i = 1\{y = i\}$
- This is shown to be the exponential function/distribution parametrized as:
  - η is a $k-1$ sized vector whose i th element is $\log(\phi_i/\phi_k)$

  - $a(\eta) = -\log(\phi_k)$
  - $b(y) = 1$
- The link function is given by $\eta_i = \log \frac{\phi_i}{\phi_k}$ for $i = 1, \ldots, k$ (η_k trivially is 0). Why?
- Inverting it to get g, the response function, $\phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}}$.
  - The denominator is merely a normalization constant. Its product with ϕ_k gives 1.
  - This function is called the **softmax** function.
- Apply the 3rd assumption of a GLM.
  - Let $\theta_0 = 0$ for convenience. Note that $\theta_i \in \mathbb{R}^{n+1}$, with $i = 1, \ldots, k$. Why?
  - This model is called the **softmax regression**
- The hypothesis outputs the estimated probability that $p(y = i|x; \theta)$ for all i.
- $l(\theta) = \sum_{i=1}^{m} \log \prod_{l=1}^{k} \left(\frac{e^{\theta_l^T x^{(i)}}}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \right)^{1\{y^{(i)}=l\}}$ is the log-likelihood function. Maximize θ using some technique (Newton's, etc). Verify if I ever use it.

4 The Problem of Overfitting

- When you have a poor fit (doesn't match many points well), it is called **underfitting** or **high bias**. In a sense you're saying that the hypothesis function is biased despite the evidence to the contrary.
- When you have a fit that matches the training data really well but performs poorly otherwise, you are **overfitting**—also known as **high variance**.
- Overfitting is essentially "memorizing" your training data. You don't really learn much about the model!

4.1 Addressing overfitting

1. Reduce number of features:
   - Manually select features.
   - Model selection algorithms (it'll choose for you what features to keep).

- This entails potentially losing valuable information.
- On the flip side, too many features requires a larger data set?

2. Regularization:
   - Keep all the features but reduce magnitudes of θ_j.
   - Works well when you have a lot of features, each of which contributes a bit to predicting y.

4.2 Regularization

- Choose small values for θ_i.
- You get a simpler hypothesis. Why?
- Get a "smoother" function. Why?
- Less prone to overfitting. Why?
- To force smaller θ_i, modify the cost function:
  - $J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{i=1}^{n}\theta_i^2\right]$
  - Note that the sum over the features starts from $i = 1$. This is by convention: θ_0 is not shrunk. In practice, it makes little difference.
  - λ is called the **regularization parameter** and the new sum is called the **regularization term**.
- λ controls the tradeoff between fitting the values and keeping the hypothesis simple.
- {Too large a λ will result in an almost straight horizontal line?}

4.3 Regularized Linear Regression

- The new update rule for gradient descent becomes:
  - $\theta_j^{k+1} = \theta_j^k\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$ for $j \neq 0$.
  - For $j = 0$, the update rule is the same as before. Verify!
- The normal equation solution is:
  - $\Theta = (X^TX + \lambda K)^{-1}X^T\mathbf{y}$ where K is like the identity matrix but with its first element as 0. Verify. {Put K explicitly!}
  - If $\lambda > 0$, then the matrix you are inverting will not be singular! Not proven

4.4 Regularized Logistic Regression

- The update rule is the same as for gradient descent, but with the logistic hypothesis.

5 Generative Learning Algorithms

- Thus far the content has been about modeling $p(y|x;\theta)$. For example in classification, we give it a training set, and it tries to come up with a boundary between the two categories. When trying to classify a new item, it will check which end of the boundary the new element is.
- A different approach is to build a model of each category (independent of other categories?) Then when a new item comes along, we see which model fits it better.
- **Discriminative** learning algorithms are ones that try to learn $p(y|x)$ directly.
- **Generative** algorithms try to model $p(x|y)$ and $p(y)$.
  - What is the probability of getting this item assuming it is of a certain category?
  - $p(y)$ is called the **class prior**.
  - Then use Baye's Rule to get the $p(y|x) = \frac{p(x|y)p(y)}{p(x)}$. Note that $p(x) = p(x|y=0)p(y=0) + p(x|y=1)p(y=1)$.
  - Of course, this makes sense only if x cannot be in multiple categories!
  - As $p(x)$ is a fixed number, we don't really need it. Just find the y that maximizes the numerator.

5.1 The Multivariate Normal Distribution

- $\mu \in \mathbb{R}^n$
- **Covariance matrix**: $\Sigma \in \mathbb{R}^{n\times n}$. $\Sigma \geq 0$. It is symmetric and positive semi-definite. {Verify the mean.}
- $p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$
  - $|A|$ is the determinant of A.
- **Covariance** of a random variable Z (vector valued) is defined by $\mathrm{Cov}(Z) = E[(Z-E[Z])(Z-E[Z])^T] = E[ZZ^T] - (E[Z])(E[Z])^T$. {Show the second equality from the first.}
  - It is a generalization of the variance.
- **Standard normal distribution**: $\mu = 1, \Sigma = I$

5.2 Gaussian Discriminant Analysis

5.2.1 The Gaussian Discriminant Analysis Model

- If the input features x are continuous valued random variables, then we can use the **Gaussian Discriminant Analysis** (GDA) model:
  - y is Bernoulli(ϕ). So $p(y) = \phi^y(1-\phi)^{1-y}$
  - $x|y=0$ is $\mathcal{N}(\mu_0, \Sigma)$
  - $x|y=1$ is $\mathcal{N}(\mu_1, \Sigma)$
  - Note that Σ is the same in both cases! {This is a choice?}
- By maximizing the log-likelihood function, we get: (verify!)
  - $\phi = \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)}=1\}$
  - $\mu_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=0\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}}$
  - $\mu_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=1\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}}$
  - $\Sigma = \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)} - \mu_{y^{(i)}}\right)\left(x^{(i)} - \mu_{y^{(i)}}\right)^T$

5.2.2 GDA and Logistic Regression

- If you look at $p(y=1|\mathbf{x}; \phi, \mu_0, \mu_1, \Sigma)$ as a function of x, you'll get $p(y=1|\mathbf{x}; \phi, \Sigma, \mu_0, \mu_1) = \frac{1}{1+\exp(-\Theta^T\mathbf{x})}$
  - Verify!
  - Θ is some function of $\phi, \Sigma, \mu_0, \mu_1$
- Note that this matches what we had earlier.
- GDA and logistic regression will give different boundaries on the same data set.
  - This is because while GDA reduces to the logistic regression in the case of a multivariate Gaussian with common Σ, the converse is not true: $p(y|x)$ being logistic does not imply $p(x|y)$ is multivariate Gaussian.
- Thus GDA makes stronger assumptions (e.g. $p(x|y)$ is Gaussian or near it).
- GDA does better when these assumptions are valid.
- If the assumptions are correct exactly (i.e. really a Gaussian), then GDA is **asymptotically efficient**. In the limit of large training sets, no algorithm will beat the GDA.

- Logistic regression is more robust—it is less sensitive to errors in your assumptions about the model.
  - Hence it is used more frequently.
  - Similar discussion on Naive Bayes
- On the flip side, because the assumptions GDA makes are stronger, you need fewer training examples. The model is exploiting the extra information it has (i.e. the assumptions).

5.3 Naive Bayes

- Assume x_i's are discrete.

5.3.1 Spam Filtering

- Example of email spam classification. Given an email, let $\mathbf{x}$ be the vector such that $x_i = 1$ if and only if the i th word in the dictionary appears in the email.
- The set of all known words is the **vocabulary**.
- So $\mathbf{x}$ is a huge, mostly sparse vector.
- In practice they don't use a dictionary, but use the union of the words that appear in the training set. This has the benefit of including words not in the dictionary.
- Sometimes some high frequency words ("and", "the") are excluded as they rarely help in determining the spam status. These are called **stop words**.
- The dictionary is large, and if we use a multinomial distribution (Why would we do that?) we would have a 2^N vector space where N is the number of words in the dictionary. Need to reduce the space.
- An assumption is made that the x_i's are conditionally independent *given* y. This means that $p(x_i|y) = p(x_i|y, x_j)$.
  - This does **not** mean that x_i and x_j are independent!
  - This is called the **Naive Bayes (NB) assumption** and the algorithm is called the **Naive Bayes Classifier**.
- Then $p(x_1, \ldots, x_N|y) = \prod_{i=1}^{n} p(x_i|y)$
- Parametrization: Let $\phi_{i|y=1}$ be the probability: $p(x_i = 1|y = 1)$—the probability that given the message is spam, the i th word was present. Let $\phi_y = p(y = 1)$.
- Using the maximum likelihood method, we get (Show!):

- $\phi_{j|y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)}=1 \wedge y^{(i)}=1\}}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}}$
- $\phi_{j|y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)}=1 \wedge y^{(i)}=0\}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}}$
- $\phi_y = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=1\}}{m}$
- The interpretation of the above is obvious.

- Then to determine if a new message is spam:
  - $p(y=1|x) = \frac{p(x|y=1)p(y=1)}{p(x)}$
  - Note: $p(x|y=1) = \prod_{i=1}^{n} p(x_i|y=1)$ and $p(x) = p(x|y=1)p(y=1) + p(x|y=0)p(y=0)$.

- Can generalize this to cases where the value of x_i is one of k choices (we had binary above). Instead of using a Bernoulli for $p(x_i|y)$, we use a multinomial.
  - If our data set were continuous, just discretize it make it a multinomial.

- Discretized Naive Bayes often works well when GDA doesn't (i.e. the continuous distribution does not follow a multivariate normal—just discretize and apply NB).

- The Naive Bayes as presented in this section is called the **multi-variate Bernoulli event model**.

5.3.2 Laplace Smoothing

- The problem with the spam filtering approach just described is that if you now receive an email with a new word, you'll have $\phi_{j|y=1} = 0 = \phi_{j|y=0}$ where j represents our new word.

- To fix this, change the estimate to $\phi_{j|y=1} = \frac{1+\sum_{i=1}^{m} 1\{x_j^{(i)}=1 \wedge y^{(i)}=1\}}{k+\sum_{i=1}^{m} 1\{y^{(i)}=1\}}$
  - k is the number of allowed values (2 for spam).

- In general, we have $\phi_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)}=j\}}{m}$ when the observations are independent and j is one of the k allowed values. The **Laplace smoothing** is where you add 1 to the numerator and k to the denominator.
  - Note that this still sums to 1! Verify
  - Under certain conditions, Laplace smoothing gives the optimal estimator. What does this even mean?

5.3.3 Event Models for Text Classification

- Another way to do spam filtering is called the **multinomial event model**.
- Let x_i represent the i-th word in the email. The possible values for x_i are $\{1, 2, \ldots, |V|\}$ where $|V|$ is the size of the vocabulary.
  - So an email of n words is represented by a vector of size n.
  - n is no longer fixed across emails.
- Define $\phi_y = p(y), \phi_{i|y=1} = p(x_j = i|y = 1), \phi_{i|y=0} = p(x_j = i|y = 0)$.
  - Note that we assume $p(x_j|y)$ is the same for all values of j—the likelihood of a word being present is invariant to its position in the text.
- Training set: $x^{(i)}$ is now the vector whose elements are $x_k^{(i)}$.
- Using likelihood techniques, we have (Verify!):
  - $\phi_{k|y=1} = \frac{1+\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)}=k \wedge y^{(i)}=1\}}{|V|+\sum_{i=1}^{m} 1\{y^{(i)}=1\}n_i}$
  - $\phi_{k|y=0} = \frac{1+\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\{x_j^{(i)}=k \wedge y^{(i)}=0\}}{|V|+\sum_{i=1}^{m} 1\{y^{(i)}=0\}n_i}$
  - Note that I've applied Laplace smoothing.
- Andrew claims the multinomial event model works better than the multivariate Bernoulli event model.

6 Support Vector Machines

Andrew claims many consider it the best supervised learning algorithm.

6.1 Notation

- For convenience, let the negative class be $y = -1$ instead of $y = 0$.
- Instead of using Θ, we'll parametrize using $\mathbf{w}$ and b s.t. $h_{\mathbf{w},b}(x) = g(\mathbf{w}^T\mathbf{x} + b)$
- Note that w is a vector and b is a scalar.
- We no longer need $x_0 = 1$—b fills the intercept term role now.
- No longer endow a probability interpretation to this ({Why not?}). We say that $g(z) = 1$ if $z \geq 0$ and -1 otherwise (not a continuous function)

6.2 Functional and Geometric Margins

- Given a training example, the **functional margin** of (w, b) with respect to the training example is given by $\hat{\gamma}^{(i)} = y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b)$.
  - If the functional margin is positive, we predicted correctly.
  - For the functional margin to be large, we need a high $\mathbf{w}^T\mathbf{x}^{(i)} + b$ (positive for positive class, and negative for negative class).
- A large functional margin implies a correct and confident prediction.
- Note that we can artificially make the functional margin large by scaling $\mathbf{w}$ and b. So we may want to enforce a normalization constraint.
- The functional margin is the distance from a training point to the seperating hyperplane. {I don't think so—true for the geometric margin!}
- Can also define a function margin with respect to the training set S: $\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^{(i)}$
  - This just means it is the "worst case" functional margin.
- Note that $\mathbf{w}$ will always be orthogonal to the separating hyperplane.
- Let the **geometric margin** be $\gamma^{(i)} = y^{(i)} \left(\left(\frac{\mathbf{w}}{||\mathbf{w}||} \right)^T \mathbf{x}^{(i)} + \frac{b}{||\mathbf{w}||} \right)$
  - When $y^{(i)} = 1$, then this represents the distance of $\mathbf{x}^{(i)}$ to the decision boundary.
  - $\gamma^{(i)} = \hat{\gamma}^{(i)}$ when $||\mathbf{w}|| = 1$ (normalized).
  - Hence the geometric margin is invariant to scaling.
  - This property is utilized by allowing us to scale $\mathbf{w}$ any way it is convenient for us to without impacting the outcome. For example, we can freely make the constraint: $|w_1 + b| + |w_2| = 4$
- Can also define a geometric margin with respect to the training set S: $\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}$ like we did for the functional margin.

6.3 The Optimal Margin Classifier

- The goal is to maximize the geometric margin.
- Assume we have a linearly separable data set. This means that a hyperplane exists that can separate all the training points.
  - The optimal margin classifier will fail for data that is not linearly separable.
- We want to find $\gamma, \mathbf{w}, b$ such that for all $i = 1, \ldots, m$, $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq \gamma$, and also keep $||w|| = 1$.

  - We want to find the largest possible γ.
  - In words, this means that each training example has a geometric/functional margin greater than γ, and we want to maximize that γ.
  - The problem is that $||w|| = 1$ is a non-convex constraint.
  - Not a general problem you can plug into an optimization routine.

- We could try replacing γ with $\hat{\gamma}$ and maximizing $\frac{\hat{\gamma}}{||w||}$, but our objective function is now non-convex.
- To get around this, utilize the freedom to scale the geometric margin.
- Constrain $\hat{\gamma} = 1$. This finally will lead to optimizing $\min_{\gamma,\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2$ such that $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 \; \forall i$
  - This has a convex quadratic objective, with linear constraints.
  - The solution is called the **optimal marginal classifier**.
  - Can use a generic quadratic programming (QP) optimizer.

6.4 Lagrange Duality

- Optimizing using Lagrange multipliers works when the constraints involve equalities. We need something that can handle inequalities.
- **Primal** optimization problem: $\min_w f(w)$ such that $g_i(w) \leq 0, i = 1, \ldots, k$ and $h_i(w) = 0, \; i = 1, \ldots, l$.
- Define the **generalized Lagrangian**: $\mathcal{L}(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w)$
  - α_i, β_i are the Lagrange multipliers.
- Consider the quantity $\theta_{\mathcal{P}}(w) = \max_{\alpha,\beta:\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta)$.
  - If a w violates the primal constraints (e.g. $h_i(w) \neq 0$ for some i), then $\theta_{\mathcal{P}}(w) = \infty$
  - If any w satisfies the constraints, then $\theta_{\mathcal{P}} = f(w)$
- Try to get $\min_w \theta_{\mathcal{P}}(w) = \min_w \max_{\alpha,\beta:\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta)$
  - This is identical to the original primal problem.
  - The optimal value of the objective is denoted by p^*. It is called the **value** of the primal problem.
- The **dual** optimization problem:
  - Define $\theta_{\mathcal{D}}(\alpha, \beta) = \min_w \mathcal{L}(w, \alpha, \beta)$

- Want to get $\max_{\alpha,\beta:\alpha_i \geq 0} \theta_{\mathcal{D}}(\alpha, \beta)$
- This is similar to the primal problem except the order of the min and the max's have been reversed.
- Denote the result with d^*—it is called the **value** as well.

- Note that $d^* \leq p^*$
  - This follows from the general principle that the "max min" of a function is never greater than the "min max".

- Under certain conditions we'll have $d^* = p^*$. So we can solve the dual problem instead. The (sufficient) conditions for this to occur are:
  - Let f and the g_i's be convex.
    - If f has a Hessian, it is convex iff the Hessian is positive semidefinite. Verify.
    - All linear and affine functions are convex. Verify.
  - Let the h_i's be affine.
    - Affine means that there exists a_i, b_i such that $h_i(\mathbf{w}) = a_i^T \mathbf{w} + b_i$
    - Affine really means linear but allowing for an intercept term.
  - Assume g_i are strictly feasible.
    - Feasible means that there exists some w so that $g_i(w) < 0 \ \forall i$.
  - Then there exists w^*, α^*, β^* such that w^* is the solution to the primal problem, α^*, β^* are the solution to the dual problem, and $p^* = d^* = \mathcal{L}(w^*, \alpha^*, \beta^*)$. {Stated without proof.}
  - The **Karush-Kuhn-Tucker (KKT) conditions** are satisfied: ({Stated without proof})
    - $\frac{\partial}{\partial w_i} \mathcal{L}(w^*, \alpha^*, \beta^*) = 0, \ i = 1, \ldots, n$
    - $\frac{\partial}{\partial \beta_i} \mathcal{L}(w^*, \alpha^*, \beta^*) = 0, \ i = 1, \ldots, l$
    - $\alpha_i^* g_i(w^*) = 0, \ i = 1, \ldots, k$ (The KKT **dual complementarity condition**)
    - $g_i(w^*) \leq 0, \ i = 1, \ldots, k$
    - $\alpha^* \geq 0, \ i = 1, \ldots, k$
  - If any w^*, α^*, β^* satisfy the KKT conditions, then they are a solution to the primal and dual problems.
  - The dual complementarity condition shows that if $\alpha_i \geq 0$, then $g_i(w^*) = 0$.

6.5 Optimal Margin Classifiers

- The primal problem for the optimal margin classifier has a constraint that can be written as $g_i(\mathbf{w}) = 1 - y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \leq 0$
- From KKT dual complementarity, we'll have $\alpha_i > 0$ only when $g_i(\mathbf{w}) = 0$
  - There will be only a few such points. They are called the **support vectors**.
  - Note that these support vectors have a functional margin of 1. They're the ones closest to the separating hyperplane.
- Setting it up as a Lagrange multiplier problem, can look at the dual form. All conditions required for $p^* = d^*$ are met, and so the KKT conditions hold. So solve the dual problem instead.
- The optimal value for $\mathbf{w}^*$ is $\sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)}$
- We also will have the constraint that $\sum_{i=1}^m \alpha_i y^{(i)} = 0$ (in the optimal case).
- The optimal value for the intercept term is:
  - $b^* = -\frac{1}{2}\left(\max_{i:y^{(i)}=-1} \mathbf{w}^{*T}\mathbf{x}^{(i)} + \min_{i:y^{(i)}=1} \mathbf{w}^{*T}\mathbf{x}^{(i)}\right)$ Verify!
  - This should be easy to derive. $\mathbf{w}^*$ dictates the normal vector of your plane. b is just the intercept. Sweep it to find the maximum.
- To make a prediction at a new input point $\mathbf{x}$:
  - Can use $\mathbf{w}^T\mathbf{x} + b$ and pick 1 or -1 accordingly.
  - Look at $b + \sum_{i=1}^m \alpha_i y^{(i)} < \mathbf{x}^{(i)}, \mathbf{x} >$ (and see if it is positive or negative).
    - Note that this is straightforward as only the support vectors have non-zero α_i's
- Is any of this valid if the data is /not/ linearly separable?

6.6 Kernels

- It had been mentioned in linear regression that given a feature x, we can make new features from it (e.g. x^2).
- New terminology: Let the "base" feature x be called an **attribute**, and all features derived from it (including x, if used) are the **features**.
- Let ϕ denote the **feature mapping** (e.g. $\phi(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix}$)
- Given a ϕ, the corresponding **kernel** is $K(x, z) = \phi(x)^T\phi(z)$.

  - This is an extension of $< x, z >$.

- Not all kernels $K(x, z)$ can be written in the above form. I think we need it to be writable in that form if we want to use them for SVM's.

- Computing the kernel can be very fast if you *don't* compute the individual $\phi(x), \phi(z)$ (e.g. $O(n)$ vs $O(n^2)$).

  - {Or so Andrew claims. I see it in his examples, but I don't know if it's a general truism}.

- **Gaussian kernel**: $K(x, z) = \exp\left(-\frac{||x-z||^2}{2\sigma^2}\right)$

  - It is a valid kernel for an SVM (i.e. it can be written as $\phi(x)^T\phi(z)$).

- Assume K is "valid". Then we denote a $k \times k$ matrix K such that $K_{ij} = K(x^{(i)}, x^{(j)})$ It is called the **Kernel matrix**.

- For a valid kernel, K is a symmetric, positive semi-definite matrix.

  - It is also a sufficiency criterion for validity. Not shown.

- **Mercer's Theorem** (for $\mathbb{R}^n$): Let $K : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$. Then for K to be valid, it is necessary and sufficient that for all $\{x^{(1)}, \ldots, x^{(m)}\}$ where m is finite, the corresponding kernel matrix is symmetric positive semi-definite. {I think the phrasing is "for all" and not "there exists".}

  - Here m is any number—it has no bearing to the training set.

- Kernels can be used for things other than SVM's. In general, if you have any machine learning algorithm that can be written in terms of $< x, z >$. then you can replace it with $K(x, z)$ and get a lower dimensional problem. Generally referred to as the **kernel trick**.

  - So you can use this trick even for linear regression, logistic regression, perceptron, etc.
  - Alternatively, think of it as a "free" way of going to higher dimensions. Your problem may not be linearly separable in your initial space, but it may be so in the higher dimensional one.
  - Although you often won't know in advance if it will be separable in the higher dimensional space.

6.7 Regularization and the non-separable Case

- As a general rule of thumb, the higher the number of dimensions (e.g. mapped using ϕ), the more likely the data will be separable. (It makes sense!). But you shouldn't assume it's the case a priori.

- A separating hyperplane is susceptible to outliers. In some cases you'd want to get those outliers wrong.

- l_1 **regularization**: $\min_{\gamma,\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^m \xi_i$ such that $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i \ \forall i$ and $\xi_i \geq 0 \ \forall i$
  - Note that if $\xi > 1$, then for that point, your classification will be incorrect (you need it to be positive for correctness).
  - Hence this algorithm is allowing for misclassifications, and I suppose this is where nonseparability is not a factor.
- This makes one less sensitive to outliers. Why?
- Can now have functional margins less than 1.
- Writing the Lagrangian and going through the same procedure as before, we get the same form for $\mathbf{w}$ in terms of α_i as before and the same equation with inner products for making predictions to new data. I did not rederive.
  - The only difference is $0 \leq \alpha_i \leq C$ Not verified.
  - b^* is now different.
- Dual-complementarity conditions are now (Not verified):
  - $\alpha_i = 0 \implies y^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) \geq 1$
  - $\alpha_i = C \implies y^{(i)}\left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) \leq 1$
  - $0 < \alpha_i < C \implies \left(\mathbf{w}^T\mathbf{x}^{(i)} + b\right) = 1$

6.8 The Sequential Minimal Optimization (SMO) Algorithm

6.8.1 Coordinate Ascent

- Let W be a function of n variables α_i. We want to find the α that maximizes it.
- In each iteration, find the $\hat{\alpha}_i$ that maximizes $W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots)$ and update α_i with it. Do this in sequence for all the α's (you can choose an ordering—and it probably matters). Repeat until you converge.
- It will take more iterations than Newton's method. However, the inner loop tends to be quite cheap—it's a lot easier to maximize with respect to one variable. So it need not be that much slower than Newton's method.

6.8.2 The Sequential Minimal Optimization (SMO) Algorithm

- To reiterate, here's the problem we want to solve: $\max_\alpha W(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j < \mathbf{x}^{(i)}, \mathbf{x}^{(j)} >$ such that $0 \leq \alpha_i \leq C, i = 1, \ldots, m$ and $\sum_{i=1}^m \alpha_i y^{(i)} = 0$
- Can't directly use coordinate ascent because one of the constraints would be violated. Fixing all but one α fixes that one as well.

- So we need to update two α's at a time.
- The SMO algorithm:
  1. Select α_i, α_j to update next.
  2. Reoptimize $W(\alpha)$ with respect to α_i, α_j holding all others constant.
  3. Repeat until convergence.
- Test for convergence can be done by seeing if KKT conditions are satisfied to within some tolerance.
  - Typically the tolerance is 0.001 to 0.01.
- The update can be done efficiently.
- {Read his notes and add the details here—may need to refer to Platt's paper!}

6.9 SVM & Kernel Method Examples

1. Say you want to recognize a digit—you scan it into an $n \times n$ grid. Then using a kernel like $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y})^d$ or the Gaussian kernel, you can do as well as the best neural networks.
   - This occurs in a $\mathbb{R}^{n^2}$ space, but kernel methods reduce the complexity.
   - You don't need to know which pixel is adjacent to which one.
2. A typical protein sequence can be represented by BADEMDIMENGNIQ... etc. Each letter represents an amino acid. The sequence length is variable. Need to categorize a protein into one of a number of classes.
   - Represent $\phi(\mathbf{x})$ by looking at all possible 4-letter sequences counting how many times each of them appears in the given sequence.
   - So you end up with a vector of dimension 26^4.
   - There are clever ways using dynamic programming for taking the scalar product, so the algorithm is fast.

7 Support Vector Machines—Coursera Treatment

- Let the classes be 0,1 as we originally had them.
- The cost function is now:
  - $C\sum_{i=1}^{m}\left[y^{(i)}cost_1(\Theta^T\mathbf{x}^{(i)}) + (1-y^{(i)})cost_0(\Theta^T\mathbf{x}^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$
  - By convention we don't have the m term—it's a scaling factor anyway.

  - C is essentially $1/\lambda$—another convention.
  - The cost functions are essentially 0 when the argument is greater than 1 (for positive class) and increase linearly as you decrease the argument. For the negative class it is just the reflection about the x-axis—the "corner" point is now at -1.
    * Draw cost functions!
  - The cost functions are a crude approximation of the one you have for logistic regression.
  - If $y = 1$, we want $\Theta^T \mathbf{x} \geq 1$ (and not just 0).
  - If $y = 0$, we want $\Theta^T \mathbf{x} \leq -1$ (and not just 0).
- Hypothesis: If $\Theta^T \mathbf{x} \geq 0$, predict 1. Else predict 0. It is now very binary (not a probability).
- So note that even if you predicted correctly, you may still incur a cost.
- {Not clear how this formalism ties to the one in the previous chapter. Here we're allowing Θ to vary in length—this is essentially not constraining $\mathbf{w}$ to be 1.}
- Assume we have a separating hyperplane. We're minimizing $||\Theta||$, which will in effect move/rotate the separating hyperplane to an ideal point.
  - This works because if you have a poor hyperplane, then $\Theta^T \mathbf{x}^{(i)}$ may be small for a given point, and you'd need a large $||\Theta||$ to compensate for it to make $\Theta^T \mathbf{x}^{(i)} \geq 1$ (for positive class) so that you have no contribution to the cost function by the first sum.

7.1 Kernels

- Define $\mathbf{l}^{(i)} = \mathbf{x}^{(i)}$ for $i = 1, \ldots, m$.
  - Note that using all m training inputs is merely one way to do it.
- Given $\mathbf{x}$, Define $f_i = K(\mathbf{x}, \mathbf{l}^{(i)})$ where K is a kernel as discussed in the previous chapter. It could be the Gaussian kernel, for example.
- Consider this a mapping of $\mathbf{x}^{(i)} \mapsto \mathbf{f}^{(i)}$ where $\mathbf{f}$ is the vector of all f_j associated with $\mathbf{x}^{(i)}$. Its dimension is m or $m + 1$.
  - Clearly $f_i^{(i)} = 1$ for the Gaussian kernel.
- Can also define $f_0^{(i)} = 1$
- To make a hypothesis, evaluate $\Theta^T \mathbf{f}$ and see if it's greater than 0.
- So we've transformed the "base" features $\mathbf{x}$ into m new features.
- To train, minimize the cost function as before, but use $\Theta^T \mathbf{f}^{(i)}$

- In many/most implementations, for regularization, instead of minimizing $||\Theta||^2$, they minimize $\Theta^T M \Theta$ where M is a matrix tied to your choice of kernel.
  - This is done for computational efficiency to offset the fact that your Θ is large (m entries).
- Andrew suggests not writing code to minimize the cost function but instead using an off the shelf SVM code.
  - Examples are liblinear and libsvm.
- To pick σ:
  - A large σ causes f_i to vary smoothly. Leads to a higher bias and lower variance.

7.2 Choice of Kernel

- Linear kernel:
  - The same as no kernel. Use this if your n is large and m is small.
- Gaussian kernel:
  - Use if n is small and m is large.
  - Perform feature scaling *before* using the Gaussian kernel.
- Polynomial kernel: $K(\mathbf{x}, \mathbf{l}) = (\mathbf{x}^T \mathbf{l} + a)^d$
  - Usually used when all x_i, l_i are non-negative.
- Other kernels:
  - String kernel
  - Chi-square kernel
  - Histogram Intersection kernel.
- Always try linear/Gaussian kernel first!
- If you implement your own kernel, you *must* satisfy Mercer's Theorem. Most SVM algorithms will rely on this.

7.3 Logistic Regression vs SVM

- If $n \geq m$, use logistic regression or a linear kernel.
- If $n = 1 - 10000, m = 10 - 10000$, use a Gaussian kernel.
- If $n = 1 - 1000, m \geq 50000$, then create/add more features and then use logistic regression or a linear kernel. Gaussian kernel may be very slow to run for large m.
- Neural network likely to work well, but slower to train.

8 Neural Networks

- Say you have n features. The way to form a neural network is to pass all n features into k sigmoid functions, each of which outputs an $a_i = g(\Theta_i^T \mathbf{x})$ where Θ_i is a vector of dimension n and is the set of parameters that correspond to the i-th sigmoid function. These a_i in turn are passed to another sigmoid function which outputs the final hypothesis $h_\Theta(\mathbf{x}) = \mathbf{a}^T\Theta$, where Θ depends on all $k+1$ Θ_i 's
- To train it, use the objective function $J(\Theta) = \frac{1}{2}\sum_{i=1}^{m}\left(h_\Theta(\mathbf{x}^{(i)}) - y^{(i)}\right)^2$
- Can apply gradient descent. **Back propagation** is the formal term used, but it's just gradient descent.
- Almost always a *non*-convex optimization problem. Can get stuck in a local optimum.
- Considered the most effective learning algorithm before SVM's (and that claim is still contested).
- $x_0 = 1$ is called the **bias unit**. Not always drawn in the diagrams.
- **Activation function** is usually just another name for the sigmoid function.
- **Weights** are just another way of saying "parameters".
- The first layer is called the **input layer**. The final layer is the **output layer**. Intermediate layers are called **hidden layers**.
- $a_i^{(j)}$: Activation of unit i in layer j.
- $\Theta^{(j)}$: Matrix of weights controlling function mapping from layer j to layer $j+1$.
  - $\Theta_{mn}^{(j)}$ is the n-th weight of the m-th activation unit in layer $j+1$.
- Have a diagram of a neural network!
- Let $z_i^{(j+1)} = \sum_{k=0}^{m}\Theta_{ik}^{(j)}x_k$
- Then $a_i^{(j)} = g(z_i^{(j)})$
- In vector notation, this becomes: $\mathbf{z}^{(j+1)} = \Theta^{(j)}\mathbf{a}^{(j)}$ where by convention we have $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{a}^{(j)} = g(\mathbf{z}^{(j)})$ where g is the sigmoid function and this is merely shorthand notation for saying we apply g to all the elements of $\mathbf{z}$
- This is **forward propagation**.

- One benefit of neural networks over linear regression is that the latter explodes combinatorially if you want all combinations of your inputs (e.g. you have k base features and you want all 4–fold products like $x_1x_3x_4x_6$ and $x_2^3x_5$)

8.1 Multiclass Classification

- You now have multiple outputs, with your output being a vector (e.g. [1 0 0 0] for class A, [0 1 0 0] for class B, etc). Thus your training set label will also be a vector.

8.2 Cost Function

- Let L be the number of layers in the network.
- Let s_l be the number of units in layer l, excluding the bias unit.
- Let k be the number of output units (s_L).
- Cost function for a neural network:

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

  - $h_\Theta(x) \in \mathbb{R}^K$ where K is the number of outputs.
  - Note: No sum over the bias units for regularization. It doesn't really make a big difference, though.
  - I'm being sloppy about vector notation.
- The cost function is typically non-convex—in practice it's not a serious problem, though.

8.3 Backpropagation Algorithm

- Notation: Let $\delta_j^{(l)}$ be the "error" of node j in layer l.
- For the output node, $\delta_j^{(L)} = a_j^{(L)} - y_j$. Can write it in vector notation.
  - I'm being sloppy about vector notation
- Then for the next to last layer, $\delta^{(L-1)} = (\left(\Theta^{(L-1)}\right)^T \delta^{(L)})^T g'(\mathbf{z}^{(L-1)})$
  - $g'(z^{(k)}) = (\mathbf{a}^{(k)})^T(\mathbf{1} - \mathbf{a}^{(k)})$ Verify!
- No $\delta^{(1)}$ term.
- Can show that if you ignore regularization:

- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$ Show it!

- Algorithm:
  - Set $\Delta_{ij}^{(l)} = 0$ for all i, j, l. These will be a proxy for the partial derivatives (gradient needed for optimization).
  - For $i = 1$ to m (number of training examples):
    - Set $\mathbf{a}^{(1)} = \mathbf{x}^{(i)}$
    - Perform forward propagation to compute $\mathbf{a}^{(l)}$ for $l = 2, \ldots, L$
    - Using $y^{(i)}$, compute $\delta^{(L)} = \mathbf{a}^{(L)} - y^{(i)}$
    - Compute $\delta^{(L-1)}, \ldots, \delta^{(2)}$
    - Set $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ In matrix notation this is $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (\mathbf{a}^{(l)})^T$ Verify!
  - Compute $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$ and $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$. Recall that $j = 0$ is a bias term.
    - I think it should be λ/m!

- Can show that $D_{ij}^{(l)}$ is the partial derivative of the cost function with respect to $\Theta_{ij}^{(l)}$ Show!

- So now you have the cost function and the derivative. Can use conjugate gradient, etc.

8.3.1 Implementation Details

- As backpropagation has a lot of minute details you can screw up, use gradient checking.

- Don't use a vector of 0's as your initial guess.

- Instead initialize $\Theta_{ij}^{(l)}$ to a random value between $-\epsilon$ and ϵ. This is just to ensure symmetry breaking.
  - One strategy for picking ϵ is to relate it to the number of units in the network.
  - $\epsilon = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}$
  - $L_{in} = s_l$ and $L_{out} = s_{l+1}$ for $\Theta^{(l)}$

8.4 Putting All The Pieces Together

- Pick an architecture.
  - By default, choose 1 hidden layer. You can try cross-validation to figure out how many layers to use.

  - If you have more than 1 hidden layer, have the same number of hidden units per layer.
  - More units is better. Anywhere from the number of inputs to 2-4×

- Randomly initialize weights
- Implement forward propagation to calculate hypothesis.
- Compute cost function.
- Implement backward propagation.
- Use gradient checking to confirm your gradients.
- Use gradient descent or similar algorithm to minimize the cost function.
  - A larger neural network will result in overfitting and you'll have to regularize. However, this usually works out better than a smaller neural network.

9 Learning Theory

- **Generalization error** of a hypothesis is its expected error on examples not necessarily in the training set.
- **Bias** of a model is the expected generalization error even when fit to a very large or infinite training set. I suppose this means it is the error from the model, as opposed to from your data.
- Definition of variance?
- **Union Bound**: $P(\bigcup_i A_i) \leq \sum_i P(A_i)$ as long as the union is of a countable number of sets.
- **Hoeffding Inequality**: Let $Z_1, \ldots, Z_m$ be independent and identically distributed random variables from a Bernoulli distribution ($P(Z_i = 1) = \phi$). Let $\hat{\phi} = (1/m)\sum_{i=1}^{m} Z_i$ be the mean of these random variables. Let $\gamma > 0$. Then $P(|\phi - \hat{\phi}| > \gamma) \leq 2\exp(-2\gamma^2 m)$ Not proven.
  - Also called the **Chernoff bound**.
  - Informally, it is saying that if you estimate ϕ by taking the average of a number of observations, then your estimate will be good as long as the number of observations is large.

9.1 Definitions

- Restrict ourselves to binary classification, but the results will apply to regression, etc.
- Let the training set S be of size m and the training examples $(x^{(i)}, y^{(i)})$ be drawn from an independent and identically distributed distribution $\mathcal{D}$.
- The **training error** of a hypothesis h is $\hat{\epsilon}(h) = \frac{1}{m} \sum_{i=1}^{m} 1\{h(x^{(i)}) \neq y^{(i)}\}$
  - This is just the fraction of cases we misclassify.
  - Also known as the **empirical risk** or the **empirical error**.
  - Note that it depends on the training set $\mathcal{S}$.
- Define the **generalization error** as $\epsilon(h) = P_{(x,y)\sim D}(h(x) \neq y)$
  - It is the probability of misclassifying a new sample (not necessarily from the training set).
  - Note that it is assumed that this is coming from the same set $\mathcal{D}$. This is one of the **PAC** (probably approximately correct) assumptions,
- One way to minimize the training error is $\hat{\theta} = \arg\min_\theta \hat{\epsilon}(h_\theta)$ for the linear classification case.
  - This is called the **empirical risk minimization** (ERM) and the resulting hypothesis is $\hat{h} = h_{\hat{\theta}}$.
  - ERM is one of the most basic learning algorithms.
  - This is a non-convex optimization problem. It is NP hard.
  - The logistic regression can be thought of as an approximation to ERM to make it a convex optimization.
- The **hypothesis class** $\mathcal{H}$ *used by a learning algorithm* is the set of all classifiers considered by it.
- ERM is a minimization over $\mathcal{H}$—the learning algorithm picks the hypothesis: $\hat{h} = \arg\min_{h\in\mathcal{H}} \hat{\epsilon}(h)$.

9.2 Finite $\mathcal{H}$

- Let $\mathcal{H}$ be a set of k hypotheses (for linear *classification*!).
- ERM will select one of these hypotheses with respect to a training sample.
- Can show that the probability that there does *not* exist a hypothesis in the training set such that $P(|\epsilon(h_i) - \hat{\epsilon}(h_i)| > \gamma)$ (i.e. we are always within γ) is greater than or equal to $1 - 2k\exp(-2\gamma^2 m)$.
  - This is called the **uniform convergence** result.

- If k is large (which it normally is), then this isn't a very useful result.

- An algorithm's **sample complexity** is the m needed to get a certain level of performance.
- Let h^* be the best possible hypothesis in $\mathcal{H}$. Then $\epsilon(\hat{h}) \leq \epsilon(h^*) + 2\gamma$.
- Let $|\mathcal{H}| = k$ and fix m, δ. Then with probability at least $1 - \delta$, we have $\epsilon(\hat{h}) \leq (\min_{h \in \mathcal{H}} \epsilon(h)) + 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}$
- Let $|\mathcal{H}| = k$ and fix δ, γ. Then for $\epsilon(\hat{h}) \leq \min_{h \in \mathcal{H}} \epsilon(h) + 2\gamma$ to hold with probability at least $1 - \delta$, it suffices that $m \geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta}$

9.3 Infinite $\mathcal{H}$

- All results below are for ERM for linear classification—may not be that correct for other algorithms.
- Given a set S of d points $\mathbf{x}$ (not necessarily the training set), we say that $\mathcal{H}$ **shatters** S if $\mathcal{H}$ can realize any labeling on S.
  - In other words no matter what labels you give to the x's, there will be some hypothesis that guesses them correctly (a different hypothesis for each method of labeling).
- Let $\mathcal{H}$ be the set of linear binary classifiers.
  - It can shatter any set of 2 points.
  - But not all sets of 3 points (it cannot handle 3 collinear points).
  - It cannot shatter even a single set of 4 points.
- Given $\mathcal{H}$, its **Vapnik-Chervonenkis dimension** $VC(\mathcal{H})$ is the size of the largest set it can shatter.
  - Its value could be ∞.
  - This does *not* mean it can shatter any set of that size—just that there exists at least one set of that size that it can shatter.
- For linear classifiers in n dimensions, $VC(\mathcal{H}) = n + 1$. Not proven. Yaser proves it for the perceptron algorithm.
- Given $\mathcal{H}$, let $d = VC(\mathcal{H})$. Then with probability at least $1 - \delta$, we have for all $h \in \mathcal{H}$, $|\epsilon(h) - \hat{\epsilon}(h)| \leq O\left(\sqrt{\frac{d}{m} \log \frac{m}{d} + \frac{1}{m} \log \frac{1}{\delta}}\right)$ Not shown!
  - Thus, with probability at least $1-\delta$, $\epsilon(\hat{h}) \leq \epsilon(h^*) + O\left(\sqrt{\frac{d}{m} \log \frac{m}{d} + \frac{1}{m} \log \frac{1}{\delta}}\right)$ Haven't verified!

- If you do the math, this means that you need $m = O(d)$ {Did not verify.}
  - It is also a lower bound. So it should be $m = \Theta(d)$?
- For most ordinary problems, d will be roughly similar to the number of parameters in your model. So as a heuristic, if you increase the number of parameters, then you correspondingly should increase (roughly linearly) the number of training examples you use.

10 Model Selection

The main idea is: How can we decide which algorithm will work best for our problem? This could be linear regression vs neural networks vs SVM, or even hypotheses within each of these models (how many features, etc).

- Assume we have a finite set of models $\mathcal{M}$.

10.1 Cross Validation

- Assume you have a training set S.

10.1.1 Bad Method

- Poor algorithm:
  - Train each M_i on S to get h_i.
  - Pick the hypothesis with the smallest training error.
- It's poor because you can always overfit to get a small training error.

10.1.2 Hold-Out Cross Validation

- Algorithm:
  1. Randomly split S into S_{train} (usually 70% of the data) and S_{cv}. cv stands for cross validation.
  2. Train each M_i on S_{train} to get h_i.
  3. Choose the hypothesis with the smallest $\hat{\epsilon}_{S_{cv}}(h_i)$
- Usually the hold out cross validation set has between 1/4 and 1/3 of the data.
- One could use step 3 to select the *model* M_i, and then retrain over the whole S.
  - Often a good idea unless your algorithm is sensitive to small perturbations in the data.
- Downside: You "waste" 30% of the data.

- If you want to report the *generalization error* (the expected error on untested samples), do *not* report the error on the cross validation set as the generalization error. Do it on a "test" set that has not been used for either training or cross validation.
  - Some times people use 60% for training, 20% for cross validation, and 20% to report the generalization error.

10.1.3 k-fold Cross Validation

- Algorithm:
  1. Randomly split S into k disjoint subsets of m/k training examples.
  2. For each M_i:
     - For each $j = 1, \ldots, k$, train M_i on all the subsets *except* S_j to get h_{ij}.
     - Test h_{ij} on S_j to get $\hat{\epsilon}_{S_j}(h_{ij})$
     - Take the average of $\hat{\epsilon}_{S_j}(h_{ij})$—averaged over j.
  3. Pick the M_i with the lowest averaged error, and then retrain M_i over S.
- $k = 10$ is typically used.
- Generally this method is more expensive than hold-out cross validation.

10.1.4 Leave-One-Out Cross Validation

- At extremes people take $k = m$ for the above.
- This is essentially training on all but one example in S, and using that one as the test.
- This method is called **leave-one-out cross validation**.

10.1.5 Comments

Can use these techniques to evaluate a *single* model/algorithm. {I don't understand how!}

10.2 Feature Selection

- You may have a large n (possibly even larger than m). You want to pare it down to the main features.
- Use heuristics to drop the number

10.2.1 Forward Search

- Algorithm:
  1. Initialize $\mathcal{F} = \emptyset$.
  2. Repeat:
     - For $i = 1, \ldots, n$, if $i \notin \mathcal{F}$, let $\mathcal{F}_i = \mathcal{F} \bigcup \{i\}$, and use some cross validation technique to evaluate $\mathcal{F}_i$
     - Pick the $\mathcal{F}_j$ that worked best in the above step and set it to $\mathcal{F}$.
  3. Select the best feature set of all the ones evaluated.
- Have to decide when to stop the outer loop.
  - One method is to stop when $|\mathcal{F}|$ hits some size (which could be n).
- This algorithm is an instance of a **wrapper model feature selection**. They work well but are expensive.

10.2.2 Backward Search

- Start with the full set of features, and delete one at a time until you have the empty set.

10.2.3 Filter Feature Selection

- Computationally cheaper
- Assign a score $S(i)$ that tells you how informative each x_i is about the class labels y. Then pick the k features with the largest scores.
- Choose $S(i)$ to be the **mutual information** $MI(x_i, y)$ between x_i and y: $MI(x, y) = \sum_{x_i \in \{0,1\}} \sum_{y \in \{0,1\}} p(x_i, y) \log \frac{p(x_i, y)}{p(x_i)p(y)}$.
  - This equation assumes both x_i and y are binary. More generally the sum would be over the domain of each.
  - The probabilities in the equation can be estimated based on their distributions in the training set.
- The mutual information can be expresses as a Kullback-Leibler (KL) divergence: $MI(x, y) = KL(p(x_i, y)||p(x_i)p(y))$
  - This gives a measure of how different $p(x_i, y)$ and $p(x_i)p(y)$ are.
  - If they were independent random variables, then they'd be equal and the divergence would be 0.
  - In the independent case the score should be low—knowing x_i doesn't help us know y.
- Once you have ranked the features, how do you pick k? One way is to use cross validation to select the k.

10.3 Bayesian Statistics and Regularization

- **Frequentist** view: θ is a constant, but unknown quantity. It is not random. We use statistics to estimate it.
- **Bayesian** view: θ is a random variable. Specify a **prior distribution** $p(\theta)$ that expresses "prior beliefs".
- Given a training set S, compute the posterior distribution:
  - $p(\theta|S) = \frac{p(S|\theta)p(\theta)}{p(S)}$
  - Which is equal to $\frac{\left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)\right)p(\theta)}{\int_\theta \left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)p(\theta)\right)\, d\theta}$
- $p(y^{(i)}|x^{(i)},\theta)$ comes from your model (e.g. the one we used for logistic regression, etc).
- Note the switch from $p(y|x;\theta)$ to $p(y|x,\theta)$.
- Given a new test example x, compute the posterior distribution on y using the posterior one on θ: $p(y|x,S) = \int_\theta p(y|x,\theta)p(\theta|S)\, d\theta$
- To get the expected value of y given x, calculate $E[y|x,S] = \int_y yp(y|x,S)\, dy$
- Doing all those integrals is expensive—especially if θ has a lot of features. So we approximate $p(\theta|S)$.
- One approximation is to use the **maximum a posteriori** (MAP) estimate: $\theta_{MAP} = \arg\max_\theta \prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)p(\theta)$
  - This is *almost* the same as the maximum likelihood estimate for θ. Verify!
- In practice, a common choice for $p(\theta)$ is $\theta \sim \mathcal{N}(0, \tau^2 I)$
  - With this choice, θ_{MAP} will have smaller norm than what you get from maximum likelihood. Not shown! {So what does that mean?}
  - This will cause the estimate to be less susceptible to overfitting than maximum likelihood. Why?
  - Bayesian logistic regression good for text classification.

11 The Perceptron and Large Margin Classifiers

- **Batch Learning**: First given the training set, then evaluate h on separate test data.
- **Online Learning**: Algorithm has to make predictions while it's learning.
- For convenience, let the negative class be -1 in this section and we'll consider the perceptron algorithm.

- If, given $(\mathbf{x}, y)$, the hypothesis correctly classifies the input, no updates to the weights are needed.
- If it misclassifies, then we update $\Theta := \Theta + y\mathbf{x}$. {He claims this is the same as the earlier update rule with -1 instead of 0 and with α dropped (not useful in perceptron), but I can't reproduce that claim.}
- **Theorem**: Let's say you have a sequence of $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$, and that $||\mathbf{x}^{(i)}|| \leq D \ \forall i$. If there exists a unit-length vector $\mathbf{u}$ such that $y^{(i)} \cdot (\mathbf{u}^T \mathbf{x}^{(i)}) \geq \gamma$ for all examples (this implies $\mathbf{u}$ separates the data with a margin of at least γ), then the total number of mistakes the online perceptron algorithm makes is at most $(D/\gamma)^2$.

12 The k-means Clustering Algorithm

- This is an unsupervised learning algorithm for clustering a set of m points.
- Algorithm:
  1. Initialize **cluster centroids** $\mu_1, \ldots, \mu_k \in \mathbb{R}^n$ randomly—assuming we want k clusters.
     - One option is to pick k of the m points, and use those as your initial μ_i.
  2. Repeat until convergence:
     - For every i, set $c^{(i)} = \arg\min_j ||\mathbf{x}^{(i)} - \mu_j||^2$
       - This is basically finding the centroid closest to each point.
     - For each j, set $\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)}=j\}\mathbf{x}^{(i)}}{\sum_{i=1}^m 1\{c^{(i)}=j\}}$
       - This is moving the centroid to the mean of the points assigned to it.
  3. If at any point you have a μ_j that has no points assigned to it, can do one of two things:
     - Eliminate that cluster (so you'll have $k-1$ ones). This is more common.
     - Randomly assign it elsewhere.
- k is a parameter of the algorithm. It is the number of clusters we desire. μ_j will represent the centroid of a cluster.
- The algorithm is guaranteed to converge. Not shown.
- Let the **distortion function** be $J(c, \mu) = \sum_{i=1}^m ||\mathbf{x}^{(i)} - \mu_{c^{(i)}}||^2$
- k-means is coordinate descent on J. Step 1 will minimize $c(i)$ while holding μ_j fixed. Step 2 does the opposite.

- In principle, it is possible to have oscillations between different clusterings, but it is very unlikely.
- J is non-convex, so a global minimum is not guaranteed.
  - However, in practice, it tends to work quite well.
  - To get around this, you can run the algorithm many times (50—1000) with different initial values. Then pick the one with the lowest J.
  - If $K = 2 - 10$, then running it several times can really help. If K is large, you don't get much benefit and you might as well use your first guess.
- To choose the number of clusters, the *most common* approach is to pick one through visualization of the data. You can try other methods, but keep this in mind.
- One approach is to plot J vs K. Pick the "elbow" point beyond which J doesn't reduce much.
  - You often don't get a clear elbow.
- Sometimes your application domain will suggest a K.

13 Mixtures of Gaussians and the EM Algorithm

- **EM** is the **Expectation Maximization** algorithm.
- Let's say we assume that each $z^{(i)}$ (unknown to us) is a value from 1 to k (integer). We assume that $P(\mathbf{x}^{(i)}|z^{(i)} = j) \sim \mathcal{N}(\mu_j, \Sigma_j)$
- $z^{(i)}$ indicate which of the k Gaussians each $x^{(i)}$ came from.
- This is the **mixture of Gaussians** model.
- The $z^{(i)}$'s are **latent** random variables as they are hidden.
- If we try to solve this using log likelihood, then we don't get a closed form expression.
- Is k known?
- The EM algorithm:
  - Repeat until convergence:
    1. E-step: For each i, j, set $w_j^{(i)} = p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$
    2. M-step: Update the parameters:
       * $\phi_j = \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)}$
       * $\mu_j = \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}}$

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

- The E-step tries to guess $z^{(i)}$. The M-step updates the parameters based on the guess.
- Susceptible to local optima, so repeat several times with different initializations.

14 Dimensionality Reduction

14.1 Motivation

- One motivation is to compress the data to speed up the algorithm.
  - Can do this if it seems, for example, that data in 3-D seems to lie on a plane. Parametrize it then into two new features (instead of the original three).
  - This works well if some features have a strong correlation. Often the case with a lot of features and large data sets.
- Another motivation is to aid in visualization (e.g. to understand the system, diagnostics, etc).
- The new features need not have a meaning.

14.2 Principal Component Analysis

- First perform feature normalization and scaling.
- Problem statement. If you want to reduce from n-dimensions to k-dimensions, find $\mathbf{u}_1, \ldots, \mathbf{u}_k \in \mathbb{R}^n$ such that the projection error is minimized.
- PCA is **not** linear regression. In the latter you're not minimizing the projection—you're minimizing something else.

14.3 Principal Component Analysis Algorithm

- Compute the **covariance matrix**: $\Sigma = \frac{1}{m} \sum_{i=1}^{n} \mathbf{x}^{(i)} (\mathbf{x}^{(i)})^T$
  - Should the index for the summation be m or n?
- This matrix is symmetric positive semi-definite.
- Compute its eigenvectors.
  - Andrew prefers using SVD to do so—claims it is more numerically stable for these kinds of matrices than the usual eigenvector routines.

- Pick the first k eigenvectors from the SVD result.
  - {Why the first ones? Are the singular values sorted from highest to lowest?}
- Let $\mathbf{z} = [\mathbf{u}^{(1)} \dots \mathbf{u}^{(k)}]^T \mathbf{x}$ (matrix whose columns are the eigenvectors). This is the projection in the reduced space.
- This procedure minimizes the projection error: $\sum_{i=1}^{m} ||\mathbf{x}^{(i)} - \mathbf{z}^{(i)}||^2$ Not shown.
- To choose k, one heuristic is to pick the minimum k that satisfies:

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

  - The denominator is the variance. So in effect 99% of variance is "retained"
  - Can use 0.05 or 0.1 if 0.01 won't work. Don't go above 0.15.
  - The quantity above is equivalent to:

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

    * where S_{ii} are the singular values.
    * This is much less expensive to calculate!
    * Shouldn't the sum be to m instead of n?
- To get back $\mathbf{x}$ from $\mathbf{z}$, do $\mathbf{x}_{approx} = U^T \mathbf{z}$ where U is the truncated $k \times n$ matrix of eigenvectors you got from SVD.
  - This is only an approximation of the original $\mathbf{x}$ unless the correlation was 1.
- Run PCA *only* on the training set! So m is the number of entries in the training set—exclude the cross-validation set.
- For visualization, only $k = 2$ or 3 would be effective. It's mostly used for data compression.
- Do *not* try to use PCA to prevent overfitting!
- Only use PCA if you can't do without it (i.e. if memory or speed is a constraint).

15 Anomaly Detection

- Examples:
  - Detecting outliers.
  - Detecting fraud.
- You usually have a data set of non-anomalous examples.

15.1 Algorithm

- Assume $x_j \sim \mathcal{N}(\mu_j, \sigma_j)$
- Define $p(\mathbf{x}) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j)$.
  - This implicitly assumes the features are independent. This is rarely true but Andrew claims the algorithm is robust nevertheless.
  - This is called the **density estimation**.
- Algorithm:
  - Choose features x_i that you think may be indicative of anomalous behavior.
  - Fit parameters μ_j, σ_j:

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

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

  - Given a new example $\mathbf{x}$, compute $p(\mathbf{x})$
  - It is an anomaly if $p(\mathbf{x}) < \epsilon$ (ϵ is chosen independently).
- To understand the rationale, consider the extreme case where all features are far along the tail of the Gaussian. That's clearly anomalous, and p will be very low.

15.2 Evaluating the Algorithm:

- Assume we have labeled data of anomalous and non-anomalous examples.
- Make a training set out of it and unlabel them. Make all of them are non-anomalous (although it's OK if it contains a few anomalous ones). Perhaps use 60% of the labeled data for this.
- Create a test and cross-validation set—include anomalous examples among them. Perhaps 20% each. Put half of the anomalous examples in each.
- Fit model $p(\mathbf{x})$ on training set.
- On a cross validation example, predict anomalousness.
- Pick your evaluation metric carefully. It's likely your data is heavily skewed (few anomalies).
  - True positive, true negative, false positive, false negative.
  - Precision/recall
  - F_1 score.
- Can use the cross-validation set to choose ϵ.

15.3 Anomaly Detection vs Supervised Learning Algorithm

Prefer anomaly detection when:

- Very small number of positive samples (0—20) and large number of negative examples.
- If you have many different types of anomalies. Hard for any algorithm to learn from positive examples—new ones may look nothing like previously known ones.

15.4 Choosing What Features to Use

- Make a plot (e.g. histogram) to see if the data is Gaussian.
  - Although it often works OK if it's not Gaussian.
- If it's not Gaussian, try a transformation that makes it look Gaussian.
  - Try $\log(x + c)$, or x^c.
- A common problem is where both your normal and anomalous examples end up with a comparable p. That means you may need to add another feature. Examine the anomalous example to see if you can find anything that sets it apart from the others.
- Try to choose features that take very large or small values on an anomaly.

15.5 Using the Multivariate Gaussian Distribution

- It can happen that when you examine the data, you can clearly see anomalous examples, but the individual features don't seem anomalous when you look at each one independently of the others. The algorithm may fail for this.
- Model $p(\mathbf{x})$ as a multivariate Gaussian all in one go.
  - $\mu \in \mathbb{R}^n$
  - $\Sigma \in \mathbb{R}^{n \times n}$ {I think this is the covariance matrix.}
- To fit:
  - $\mu = \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}^{(i)}$
  - $\sigma = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}^{(i)} - \mu)(\mathbf{x}^{(i)} - \mu)^T$
- The original model using a Gaussian for each feature is a special case of the multivariate. So why use it?
  - Use the original if performance is a concern—it tends to scale better with n.

  - Use the original model if $m < n$. The multivariate model simply won't work in this case—Σ will be singular. As a rule of thumb, if $m < 10n$, use the original.

- Andrew claims you do not need to create new features to catch anomalies if you use the multivariate model.
- If you find that Σ is singular, then consider the possibility that your features are linearly dependent. Very rare.

16 Recommender Systems

As a motivating example, let's consider a movie recommendation system.

- Notation:
  - n_u is the number of users.
  - n_m is the number of movies.
  - $r(i,j) = 1$ is the user j has rated movie i. 0 otherwise.
  - $y^{(i,j)}$ is the rating by user j on movie i.
  - $\Theta^{(j)}$ is the parameter vector for user j.
  - $\mathbf{x}^{(i)}$ is the feature vector for movie i. As an example, $x_1 \simeq 1$ means it is heavily romance, $x_2 \simeq 0.5$ means it has some comedy, etc.
  - $m^{(j)}$ is the number of movies rated by user j.
- We have $x_0 = 1$ as usual.
- For each user j, learn parameters $\Theta^{(j)} \in \mathbb{R}^{n+1}$. Then predict user j's rating for movie i with $(\Theta^{(j)})^T \mathbf{x}^{(i)}$
- Learning function for user j:

$$\min_{\Theta^{(j)}} \frac{1}{2} \sum_{i:r(i,j)=1} \left((\Theta^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\Theta_k^{(j)})^2$$

- Learning function for all users:

$$\min_{\Theta^{(1)},\ldots,\Theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\Theta^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^{(j)} \right)^2$$

- This is pretty much the same as linear regression but without the division by m which was merely a scaling factor any way.
- Gradient Descent Update:

– $k = 0$:

$$\Theta_k^{(j)} := \Theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} \left((\Theta^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)} \right) x_k^{(i)}$$

– $k \neq 0$:

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

16.1 Collaborative Filtering

- In the previous section, we assumed we had a $\mathbf{x}^{(i)}$ for every movie. But how do we know what weights to give to a money with respect to romance, comedy, action, etc? It's not like you can get someone to watch all the movies and rank them.
- But if you're Netflix, you can get your customers to do it!
- New problem: Given $\Theta^{(j)}$ for many j's, (i.e. for many people), try to optimize to get $\mathbf{x}^{(i)}$:

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

- To learn it for all movies:

$$\min_{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} \left((\Theta^{(j)})^T \mathbf{x}^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)} \right)^2$$

- Compared to the previous section, this sounds like a chicken and egg problem. So what you do is make an initial guess of $\Theta^{(i)}$, get $\mathbf{x}$ and use that to revise your guess for $\Theta^{(i)}$. I think Andrew claims it usually converges.
- A more systematic way is to minimize:

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

  – Minimize over all the $\mathbf{x}^{(i)}, \Theta^{(j)}$ simultaneously.
  – Drop $x_0 = 1$ and θ_0—Since we're co-optimizing, the optimizer will handle both (in a sense).
  – The first sum is the same as the sum in the original two formulations (and those two are identical, if you think about it).

- Algorithm:
  - Initialize $x^{(i)}, \Theta^{(i)}$ to small random values.
  - Update step in gradient descent:

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

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

16.2 Find Related Items

- To find the 5 movies most similar to $\mathbf{x}^{(i)}$, find the five j's such that $||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||$ are the smallest.

16.3 Mean Normalization

- Let's say a new user joins and (s)he has not made *any* ratings. How can we predict/recommend the movie to him/her?
- If we use the above algorithm, it will simply predict 0 (the only term in the sum will be the regularization term).
- So an alternative approach is to normalize all your movie ratings around the mean—simply take the mean of all the ratings for a given movie, and subtract it from all the ratings to get a mean of zero.
- Train everything on this data. Make a prediction as normal but add the mean back in at the end.
- Now for that user, the recommended ratings will be the mean of all the other ratings.
- {Does this not mean that, since the mean changes each time a movie is rated, the training has to occur repeatedly?}
- {Actually, even without mean normalization, would we have to retrain often?}

17 Large Datasets

17.1 Stochastic Gradient

- See notes in earlier chapter.
- Algorithm:

1. Shuffle your data set.
2. For each $\mathbf{x}^{(i)}$, update Θ (i.e. loop over all your inputs).
3. Repeat $(1 - 10\times)$

- The goal was to avoid the summation over m in each iteration. But it seems like I'm doing that anyway! The claim is that this will have faster convergence.
- Approaches the minimum in a circuitous route (it may backtrack). It will likely hover near the minimum (i.e. not quite converge to it).

17.1.1 Convergence of Stochastic Gradient

- Plot the cost after every, say, 1000 iterations (*before* updating Θ).
- Normally α is held constant. If you want it to converge to a solution, slowly decrease α over "time". As an example: $\alpha = \frac{a}{b+k}$ where a, b are constants and k is the current iteration number.
  - This is not normally done.

17.2 Mini-Batch Gradient Descent

- Stochastic uses one example per iteration. Batch uses all examples per iteration. Mini-batch is in between—it uses b examples per iteration. It is called the **mini-batch** size.
- Typical range is $b = 2 - 100$. 10 is typical.
- $\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{k=i}^{i+b-1} \left(h_\Theta(\mathbf{x}^{(k)}) - y^{(k)}\right) x_j^{(k)}$
  - Then increase i by b.
- The benefit over stochastic gradient descent is that it allows for vectorization/parallelization.

17.3 Online Learning

Say you have a web site and a user comes to your site and you collect some information about him and, say, classify him in some way. How do you handle this?

Very simple:

1. Get $(\mathbf{x}, y)$ corresponding to the user.
2. Update Θ as you normally would.

Done. No need to store your data. Assuming you keep getting people visiting your site, you have potentially limitless data.