Summary Notes
EE425X - Machine Learning: A signal processing perspective
Namrata Vaswani

These notes are work in progress. For the latest version at any time, see the link https://www.dropbox.com/s/4nqcceyyjviebwm/Summary_Notes.pdf?dl=0

Acknowledgement

Much of these notes are based on material presented in this Stanford Machine Learning class http://cs229.stanford.edu/syllabus.html.

Course Plan

Week 1: Linear Algebra recap.
Week 2: Linear Regression
Week 3, 4: Practical issues, how to simulate data, why that is important, Monte Carlo estimation of expected values
Week 3, 4: Review of probability
Week 3, 4: Bias Variance Tradeoff in the context of Linear Regression.
Week 4, 5: Logistic Regression.
Week 6: Generative Learning algorithms - Gaussian Discriminant Analysis and “Spam Filter” (Discrete or Categorical input data), Naive Bayes assumption to simplify models.
Week 7: Review and begin Support Vector Machines (SVM)

Today, Friday, Monday: Complete the intuitive brief discussion of bias-variance tradeoff, intuitive explanation of Monte Carlo estimation, Probability Recap (EE 322 notes), Justification for using Squared Loss for learning model parameters, Bias-Var tradeoff detailed discussion.

1 Supervised Learning

Given observed data (or features of the observed data) or other “input” \( x \), the goal is to predict some function of \( x \) that is denoted by \( y \) (often called “output”). In what we have talked so far \( x \) is an \( n \times 1 \) vector and \( y \) is a scalar. For example, \( x \) can be the feature vector of key attributes of a house, while \( y \) can be its price. In this case both are real valued. Or \( y \) can be a binary decision about whether a buyer buys the house or not.

In learning, we first decide a modeling strategy to model the input-output relationship; then come up with an algorithm to “learn” parameters given the training data (which is a set of \( m \) input-output pairs in case of supervised learning). All of this is done so that the “learnt model” can be used to predict \( y \) (get \( \hat{y} \)) for a new query \( x \).

“Predict” is often also called “estimate” (if \( y \) is real-valued) and it is also called “detect” or “classify” (if \( y \) is binary/discrete-valued).
Learning algorithms can be supervised or unsupervised. In supervised learning, we are provided with “training data” that allows us to “learn” the parameters used by the model that our algorithm relies on. Goal is to predict \( y \) using observed data or features \( x \).

\( x \) is \( n \times 1 \), \( y \) is a scalar. We use \( \theta \) to denote the set of parameters used by our assumed model. The number of parameters (length of \( \theta \)) can be \( n \) or more or less.

In many settings, the assumed model that predicts \( y \) is denoted \( h_\theta(x) \). Since the model is never perfect, we assume that the “true” output \( y \) satisfies

\[
y = h_\theta(x) + e
\]

where \( e \) is the modeling error or noise. This is typically modeled as a random variable with a probability density function (PDF), typically zero mean Gaussian and independent and identically distributed (i.i.d.) in each new sample.

Training data consists of \( m \) input/output pairs \( \{x^{(i)}, y^{(i)}\}, i = 1, 2, \ldots, m \). The modeling error / noise \( e \)

We use these to “learn” \( \theta \). Once that is done, we can predict \( y \) from \( x \) using the above equation.

## 2 Supervised Learning: Linear Regression

In the setting we have talked about in class, \( x \) is a real-valued \( n \times 1 \) vector and \( y \) is a real-valued scalar. In more general settings \( y \) can also be a real-valued vector (this will not be discussed in our class).

### 2.1 Model

In linear regression, \( h_\theta(x) \) is a linear function of \( x \).

\[
h_\theta(x) = \theta^T x.
\]

(I sometimes may use ’ for transpose – MATLAB notation)

### 2.2 Learning \( \theta \): minimize squared loss

The most common approach to learn \( \theta \) is to assume a squared error loss and try to minimize it, i.e., find

\[
\arg \min_\theta J(\theta) := \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2.
\]

Define an \( m \times n \) input data matrix \( X \) with \( (x^{(i)})^T \) as its rows, and define an \( m \times 1 \) vector \( y \) with \( y^{(i)} \) as its columns. Then, \( J(\theta) \) can be expressed more compactly as

\[
J(\theta) := \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2 = \frac{1}{m} \|y - X\theta\|_2^2
\]

#### 2.2.1 Understanding squared loss: Maximum Likelihood Estimation under i.i.d. Gaussian model

The above can be motivated as Maximum Likelihood Estimation: maximize \( p(y; X, \theta) \) under the model \( y^{(i)} = h_\theta(x^{(i)}) + e^{(i)}, i = 1, 2, \ldots, m \) with \( w^{(i)} \) independent identically distributed (i.i.d.) standard Gaussian. Here the randomness is only in the noise \( e \).
2.2.2 Solutions for minimizing squared loss: called Least Squares (LS) Estimation

1. We can get a closed form solution by taking the derivative of $J(\theta)$ w.r.t. $\theta$ and setting it to zero. When $X$ is full rank $n$ (a necessary condition for this is $m \geq n$), this simplifies to

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

2. For an $m \times n$ matrix with $m \geq n$, $(X'X)^{-1}X'$ is the pseudo-inverse of $X$, denoted $X^\dagger$. Thus, we can also write the above solution as

$$\hat{\theta} = X^\dagger y$$

Extra details: The pseudo-inverse is computed by first computing the singular value decomposition (SVD) of the matrix $X$. Suppose the SVD of $X = U\Sigma V'$ where $U$ is an $m \times m$ unitary matrix. $V$ is an $n \times n$ unitary matrix and $\Sigma$ is diagonal with non-negative entries. Then $X^\dagger = V\Sigma^\dagger U^T$. For a rectangular diagonal matrix such as $\Sigma$, we get the pseudo-inverse by taking the reciprocal of each non-zero element on the diagonal, leaving the zeros in place, and then transposing the matrix.

If algorithms for computing matrix inverse or matrix pseudo-inverse were "exact" (and not iterative) then the above two approaches would return the exact same solution. But they are not. Thus, when $X$ is "well-conditioned", both approaches above return same solution, but not otherwise.

3. For large sized problems (where $n$ is large), using Gradient Descent (GD) is a better idea. Since problem is convex, GD should, in principle, converge to above solution starting from any initialization, and should converge pretty quickly. We explain this next.

4. Approximate but even faster solution: Stochastic GD (S-GD) can be used. Advantages: faster per iteration; needs lesser memory; and is useful to get a fully streaming algorithm. But no easy guarantees on whether it will converge and to what.

3 Gradient Descent (GD) and Stochastic GD

The GD approach is an iterative algorithm to find a local minimizer of a cost function. Which minimizer is found depends on how one initializes. It does not always converge and of course local minimizer may not be a global minimizer either. But if cost function is convex, it will converge to a global minimizer. Moreover if minimizer is also unique (cost is strictly convex), then the only correct solution can be found. An example of this is the squared loss for linear regression. It is convex always. It is strictly convex if $m \geq n$ and $X$ has full rank $n$.

The GD Algorithm to minimize any cost $J(\theta)$ is as follows.

1. Initialize $\hat{\theta}_0$ as the zero vector (or anything else).

2. Repeat the following for iterations $t > 0$ until “stopping criterion” is reached:

$$\hat{\theta}_t = \hat{\theta}_{t-1} - \mu \nabla_{\theta} J(\hat{\theta}_{t-1})$$

increment $t$ by one and repeat until stopping criterion reached:

A typical stopping criterion: stop when $||\hat{\theta}_t - \hat{\theta}_{t-1}||_2/||\hat{\theta}_{t-1}||_2 \leq \epsilon$ with $\epsilon$ a very small tolerance, e.g., set $\epsilon = 10^{-8}$.

3.1 Stochastic GD

$J(\theta)$ is typically a sum of $m$ terms; in fact it always is under our assumption of different training data points being i.i.d. As a result, its gradient is also a sum of $m$ terms.

If $m$ is large computing the full gradient at each iteration can be expensive. Also, sometimes not all data is available immediately

Stoch GD idea: sum over a subset of the $m$ gradients at each iteration. Pick this subset randomly or use other strategies.
4 Supervised Learning: Logistic Regression

In this case, $x$ is still a real-valued $n \times 1$ vector but now $y$ is a binary scalar.

This assumes that $\Pr(y = 1; x, \theta) = h_\theta(x)$ with

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

g(.) is called the sigmoid function, it takes values between zero and one for all values of $z$. Thus, it can be used to model a probability. Said another way,

$$p(y; x, \theta) = h_\theta(x)^y(1 - h_\theta(x))^{1-y}$$

The prediction is

$$\hat{y} = \arg \max_{y=0,1} p(y; x, \theta)$$

Thus

$$\hat{y} = 1 \text{ if } h_\theta(x) > 1 - h_\theta(x),$$

and $\hat{y} = 0$ otherwise.

4.1 Learning $\theta$: Maximum Likelihood Estimation

Again define $y$ and $X$ as before from training data

Use Maximum Likelihood Estimation again: assume i.i.d. training data points $y^{(i)}$ (recall that this was assumed also in linear regression – it was imposed by letting the $w^{(i)}$’s be i.i.d.).

Thus, we maximize

$$J(\theta) := p(y|X; \theta) = \prod_{i=1}^m p(y_i; x^{(i)}, \theta) = \prod_{i=1}^m h_\theta(x^{(i)})^{y^{(i)}}(1 - h_\theta(x^{(i)}))^{1-y^{(i)}}$$

Find $\theta$ by maximizing $J(\theta)$ by GD or S-GD. Neither has convergence guarantees in general since the problem is not convex.

Better always to maximize the log of a probability for numerical reasons.

?? copy gradient expression.

5 Supervised Learning: Generative Learning (a.k.a. Bayesian models/learning)

For logistic or linear regression we just assumed a probabilistic model on how $y$ is generated from $x$, with $x$ being deterministic.

In Generative Learning, we assume a “generative model”: we first put a prior probabilistic model on $y$, and then assume a probabilistic model on how $x$ was generated from $y$. We then compute the probability (or probability density function in case $y$ is real-valued) of $y$ taking a certain value given $x$ using Bayes rule. Mathematically, we assume that we are given

$$p(x|y; \theta), p(y)$$

and we use these to obtain the prediction as follows

$$\hat{y} = \arg \max_y p(y|x; \theta) := \arg \max_y \frac{p(x|y; \theta)p(y; \theta)}{p(x; \theta)} = \arg \max_y p(x|y; \theta)p(y; \theta)$$

This use of Bayes rule is called Maximum A Posteriori (MAP) detection or estimation in other literature. The overall approach is often called Bayesian modeling or physics-based modeling.
5.1 Learning $\theta$: Maximum Likelihood Estimation

Estimate $\theta$: define $y$, $X$ as before from training data. Also assume training data points are independent: \{\(x^{(i)}, y^{(i)}\)\} are mutually independent for different $i$. Define the cost function

$$J(\theta) := \Pr(y, X; \theta) = \prod_{i=1}^{m} p(y^{(i)}, x^{(i)}; \theta)$$

or, usually its logarithm, and maximize it over $\theta$.

5.2 Generative Learning: Gaussian Discriminant Analysis (GDA)

This is one type of generative model and learning algorithm for the setting $x$ real-valued $n \times 1$ vector and $y$ binary scalar. Thus $y$ can take two values 0 or 1. It assumes $x$ is Gaussian given $y$ and $y$ itself is Bernoulli, i.e.,

$$p(x|y; \theta) = \mathcal{N}(x; \mu_y, \Sigma_y), \ p(y) = \phi^y (1 - \phi)^{1-y}$$

Notice in this case $\theta = \{\mu_0, \mu_1, \Sigma_0, \Sigma_1, \phi\}$. Parameters are still learnt by MLE

$$\max_{\theta} J(\theta) := \Pr(y, X; \theta) = \prod_{i=1}^{m} p(y^{(i)}, x^{(i)}; \theta) \text{ s.t. } 0 \leq \phi \leq 1$$

Notice that, without extra assumptions, we have $2n + 2n^2 + 1$ parameters. Training data are each $n$ length vectors $x^{(i)}$, thus we can say we have $mn$ training data scalars. We need $mn$ significantly larger than $2n + 2n^2 + 1$ for training/learning to be accurate. We will need $m$ growing at least linearly with $n$ to be able to learn anything useful.

But the point of Bayesian (generative) modeling is that we should be able to use a smaller $m$ and still train well.

When enough training data is not available, we need to simplify our model so that there are fewer parameters. As explained later, this will increase model bias, but will reduce the variance in parameter estimation.

A common model simplification is to assume that the different entries of each $x^{(i)}$ are independent conditioned on the class label $y^{(i)}$. This is called the Naive Bayes assumption.

5.2.1 Gaussian Discriminant Analysis with Naive Bayes assumption

A common model simplification is to assume that the different entries of each $x^{(i)}$ are independent conditioned on the class label $y^{(i)}$. This is called the Naive Bayes assumption.

In the Gaussian case, this translates to assuming that $\Sigma_0$, $\Sigma_1$ are diagonal. With the diagonal assumption, we now have only $2n + 2n + 1$ parameters which is much more manageable. A second commonly used simplification is to assume the same covariance under both classes, i.e., that $\Sigma_0 = \Sigma_1 = \Sigma$ and $\Sigma$ is diagonal. With this assumption too, we have the following simpler model

$$\prod_{i=1}^{m} \left( \prod_{j=1}^{n} \mathcal{N}(x_{ij}; \mu_{y^i}, \sigma_{y^i}^2) \right) \cdot \phi^{y_i} (1 - \phi)^{1-y_i}$$

Under the above assumption, the Max Likelihood Estimates (MLE) of the model parameters are com-
puted as follows:

\[ \hat{\phi} = \frac{1}{m} \sum_{i=1}^{m} 1(y^{(i)} = 1) \]

\[ \hat{\mu}_0 = \frac{\sum_{i=1}^{m} 1(y^{(i)} = 0)x^{(i)}}{\sum_{i=1}^{m} 1(y^{(i)} = 0)} \]

\[ \hat{\mu}_1 = \frac{\sum_{i=1}^{m} 1(y^{(i)} = 1)x^{(i)}}{\sum_{i=1}^{m} 1(y^{(i)} = 1)} \]

\[ \hat{\sigma}^2_j = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})^2, \quad j = 1, 2, \ldots, n \]

while setting all non-diagonal entries of \( \hat{\Sigma} \) to be zero. Here \( 1 \) denotes the indicator function of the statement in paranthesis. Thus, \( 1(y^{(i)} = 0) \) equals one if \( y^{(i)} = 0 \) and it equals zero otherwise.

Notice that the above is equivalent to learning the parameters for each feature independently, it can also be rewritten as follows: for each \( j = 1, 2, \ldots, n \), compute

\[ (\hat{\mu}_0)_j = \frac{\sum_{i=1}^{m} 1(y^{(i)} = 0)(x^{(i)})_j}{\sum_{i=1}^{m} 1(y^{(i)} = 0)} \]

\[ (\hat{\mu}_1)_j = \frac{\sum_{i=1}^{m} 1(y^{(i)} = 1)(x^{(i)})_j}{\sum_{i=1}^{m} 1(y^{(i)} = 1)} \]

\[ \hat{\sigma}^2_j = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})^2, \quad j = 1, 2, \ldots, n \]

### 5.3 Generative Learning: Spam Filter – an example of Discrete or Categorical Features (inputs)

In applications such as spam email detector (or filter) design, one typically models \( x \) as a discrete-valued vector given \( y \).

\( y = 0 \) means the email is not-spam, \( y = 1 \) means it is spam.

#### 5.3.1 Simple Spam Filter: entries of \( x \) are binary

In the simplest version, \( x \) is an \( n \times 1 \) binary vector with \( n \) being equal to the size of the English dictionary. We say \( x_j = 1 \) if the \( j \)-th dictionary word is in the email and \( x_j = 0 \) otherwise. This means \( n \) is really large. Also it is not counting how many times a word occurred.

Consider first the simplest model where \( x \) is a binary vector. As before, we specify

\[ p(x|y; \theta), \quad p(y) \]

and we predict

\[ \hat{y} = \arg \max_y p(x|y; \theta)p(y; \theta) \]

Notice now that \( x \) can take a total of \( 2^n \) possible values. We also need to specify the prior \( p(y) \). Thus, in this most general case, the number of parameters equals \( 2^n + 1 \).

Training data are each \( n \) length vectors \( x^{(i)} \), thus we can say we have \( mn \) training data scalars. We need \( mn \) significantly larger than \( 2^n + 1 \) for training/learning to be accurate. Here we will need \( m \) to grow linearly with \( 2^{n-1} \): this can be very large and is not practical.
5.3.2 Naive Bayes assumption

In both the above examples and especially the second one, the required $m$ can be very large for accurate training/learning. Thus we add a further modeling assumption called “Naive Bayes” in ML literature. Others would call it “conditional independence” of different entries of a feature vector (the different $x_j$’s, $j = 1, 2, \ldots, n$) given $y$. Mathematically, we are assuming

$$p(x|y; \theta) = \prod_{j=1}^{n} p(x_j|y; \theta)$$

This may not be a very realistic assumption, but it significantly reduces the number of parameters required by the model.

In the Gaussian case, this implies that $\Sigma_0, \Sigma_1$ are \textbf{diagonal matrices}. Thus, the number of parameters becomes $2n + 2n + 1$ which is much more tractable. In the spam filter case, this means we have $n + 1$ parameters.

So now the number of training samples $m$ does not even need to grow with $n$.

5.3.3 Simple Spam filter with Naive Bayes

$x$ is a binary vector, $y$ is a scalar. With using Naive Bayes, in the Simple Spam Filter case, we can now define

$$\phi_{j,y} := p(x_j = 1|y), j = 1, 2, \ldots, n; \text{ and } \phi := p(y = 1)$$

With this we have just $n + 1$ parameters to learn instead of $2^n + 1$.

We can again learn the parameters by MLE:

$$\max_{\theta} J(\theta) := \prod_{i=1}^{m} p(y^{(i)}, x^{(i)}; \theta) = \prod_{i=1}^{m} p(x^{(i)}|y^{(i)}; \theta)p(y^{(i)}; \theta) = \prod_{i=1}^{m} \prod_{j=1}^{n} p(x_j^{(i)}|y^{(i)}; \theta)p(y^{(i)}; \theta) = \prod_{i=1}^{m} \phi(\prod_{j=1}^{n} \phi_{j,y})$$

s.t. constraints that each $\phi_{j,y}, \phi$ lie between zero and 1.

Can again get closed form simple expressions for the MLE:

$$\hat{\phi}_{j,0} = \text{count the number of training data points for which } y^{(i)} = 0 \text{ and } x_j^{(i)} = 1 \text{ and divide by the total number of training data points for which } y^{(i)} = 0.$$  

Similarly

$$\hat{\phi}_{j,1} = \text{count the number of training data points for which } y^{(i)} = 1 \text{ and } x_j^{(i)} = 1 \text{ and divide by the total number of training data points for which } y^{(i)} = 1.$$  

$$\hat{\phi} = \text{count the number of training data points for which } y^{(i)} = 1 \text{ and divide by } m$$

Also, instead of letting $n$ by the size of the English dictionary, we can let $n$ be the size of the vocabulary (set of all words in all training data).

5.3.4 Dealing with Categorical data

Instead of modeling $x_j$ as binary, one could model it as taking one of $K_j$ possible values and assume each entry is independent. Such type of data is called “Categorical”.

The model would then require us to learn $\phi_{j,y,k} := \Pr(x_j = k|y)$ for $k = 1, 2, \ldots K_j, y = 0, 1, j = 1, 2, \ldots, n$. 


Our data model then is as follows

\[ p(x|y) = \prod_{j=1}^{n} \prod_{k=1}^{K_j} \phi_{j,y,k}^{[x_j == k]} \]

where \([x == k]\) takes the value 1 if \(x = k\) and zero otherwise (MATLAB notation). Thus for MLE we need to maximize

\[ \prod_{i} \left( \prod_{j=1}^{n} \prod_{k=1}^{K_j} \phi_{j,y(i),k}^{[x_j(i) == k]} \phi_{y(i)}^{(1 - \phi)^{1-y(i)}} \right) \]

s.t. sum to one constraints on all the probabilities.

Then MLE estimates are given by

\[ \hat{\phi}_{j,0,k} \]

counts the number of times \(y(i) = 0\) and \(x_j(i) = k\) in the training data and divides this by the number of times \(y(i) = 0\) (number of points from class zero in the training data). Do this for every value of \(k = 1, 2, \ldots, K_j\) and for every feature \(x_j, j = 1, 2, \ldots, n\).

Do the same for class label 1.

### 5.3.5 Spam Filter: entries of \(x\) are discrete-valued

A better model than just checking if word \(j\) is present or absent is to let \(x_j\) be the number of times word \(j\) occurred in an email. The probability model that should be used in this case depends on what else is assumed.

One could either deal with the “frequency of each word” in which case we use a multinomial distribution. But then the occurrences of different words are not independent.

Or treat the counts as independent categorical data. Problem here is we will need to set a max bound of \(K\) or quantize further.

### 5.3.6 Other approaches to Spam filter design (all with Naive Bayes)

In practice many other tricks are used with this model or other better models are used. For example \(x_j\) can be set as the number of times the word \(j\) occurred. And instead of entire English dictionary, we define \(n\) as the “vocabulary” size which means we let \(V\) be the set of all words present in any of the training data emails. With this, the vector \(x\) has a multinomial distribution given \(y\).

### 5.4 Dealing with discrete-valued and real-valued data together in the Generative Learning (a.ka. Bayesian) framework

Often some of the features can be categorical and some of them can be real-valued. Once we impose the Naive Bayes assumption (conditioned on the class label, different features are independent), this is easy to deal with. For simplicity suppose that the first \(r\) features are real-valued and the rest \(n - r\) are categorical with \(k_j\) categories. Also assume the real-valued feature follow a Gaussian distribution with mean \(\mu_0\) or \(\mu_1\) (depending on the class label) and covariance matrix \(\Sigma\). By Naive Bayes, \(\Sigma\) is diagonal.

Then, joint likelihood becomes

\[ \prod_{i=1}^{m} \left( \prod_{j=1}^{r} \mathcal{N}(x_j^{(i)}; (\mu_{y(i)}), \sigma_j^2) \cdot \prod_{j=r+1}^{n} \prod_{k=1}^{K_j} \phi_{j,y(i),k}^{[x_j^{(i)} == k]} \cdot \phi_{y(i)}^{(1 - \phi)^{1-y(i)}} \right) \]

s.t. sum to one constraints on \(\phi\) and on all the \(\phi_{j,y,k}\)'s.
The MLE estimates are computed as follows.

\[ \hat{\phi} = \frac{1}{m} \sum_{i=1}^{m} 1(y^{(i)} = 1) \]

For \( j = 1, 2, \ldots, r, \)

\[ (\hat{\mu}_0)_j = \frac{\sum_{i=1}^{m} 1(y^{(i)} = 0)(x^{(i)})_j}{\sum_{i=1}^{m} 1(y^{(i)} = 0)} \]

\[ (\hat{\mu}_1)_j = \frac{\sum_{i=1}^{m} 1(y^{(i)} = 1)(x^{(i)})_j}{\sum_{i=1}^{m} 1(y^{(i)} = 1)} \]

\[ \hat{\sigma}^2_j = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})^2, \quad j = 1, 2, \ldots, n \]

For \( j = r + 1, r + 2, \ldots, n, \)

\[ \hat{\phi}_{j,0,k} = \]

counts the number of times \( y^{(i)} = 0 \) and \( x^{(i)}_j = k \) in the training data and divides this by the number of times \( y^{(i)} = 0 \) (number of points from class zero in the training data). Do this for every value of \( k = 1, 2, \ldots, K_j \) and for every feature \( x_j, j = 1, 2, \ldots, n. \)

### 6 Reliability of an output

Linear regression predicts a real-valued scalar where as everything we learnt after that predicts a class label (solves a classification problem). Given a query, we can always obtain a prediction. But the other important question to answer is: how reliable is the prediction we obtained. The answer to this question depends on

- the problem itself, e.g., in case of classification by GDA, if the two class means are very close, it is not easy to distinguish the classes. More precisely what matters is how close the class means along a given direction compared to the standard deviation along that direction. Practically this means the following: it easier to distinguish dog pictures from human pictures than from cat pictures.

  Similarly for a regression problem, the amount of modeling error \( e \) or its variance decides how good the prediction is.

- the number of training data points, and how well the training and test data match (this decides quality of learnt model). In most of what we learn, it is assumed that training and test data are genareted from the same distribution, but in real life this may not be true.

- the specific query: if the query image is of a fluffy cat that may look dog-like, then it is hard to reliably provide a correct classification.

- the last problem can be partly addressed by changing the learning algorithm (the assumed model on the data).

### 7 Supervised Learning: Support Vector Machines (SVMs)

See handout

To Do. add summary.

### 8 Supervised Learning: Decision Trees

See handout
9 Unsupervised Learning: PCA

In unsupervised learning, there is no training data to learn parameters from. PCA is an unsupervised learning technique that is used for dimension reduction. Given data vectors in $\mathbb{R}^n$, if they approximately lie in a lower dimensional subspace, how do we find that subspace?

9.1 Why PCA

Suppose that we are given $m$ data vectors $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ each in $\mathbb{R}^n$, and are stacked as the rows of a matrix

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

Notice each data vector is $n$-length. $n$ may be very large can, e.g., can be equal to the image size if one uses all the pixels as “features”. The question is can we reduce the dimension of the data without losing too much “information”? This would mean we need

- less storage needed to save the data;
- data analysis, e.g., classification will be faster; and
- if the original data is noisy, the data analysis task, e.g., classification, can also be more accurate in the reduced dimensional space. You may notice this for certain real datasets with PCA implemented carefully. For simulated data, to observe this, you will have to deliberately generate noisy simulated data.

The reason we can dimension reduce without losing too much information is because a lot of the clean versions of real data (“signals”) are exactly or approximately “structured”; here this means that they lie close to a lower dimensional subspace of $\mathbb{R}^n$.

The goal of PCA is to find this subspace and to project data into it.

9.1.1 Goal of PCA

Said another way, the goal is to find a rank $r$ matrix $L$ that approximates $X$; with $r$ being much smaller than $n$ (and also smaller than $m$). What does approximates mean? More precisely, we would like to find a rank $r$ matrix $L$ that minimizes $\|X - L\|$, i.e., it minimizes some “norm” of the residual error (error when $X$ is approximated by $L$). Two commonly used norms are the induced 2-norm and the Frobenius norm (also called the vector 2-norm). Both are minimized by the “r-SVD” solution.

9.2 PCA solution: “r-SVD” – assume the desired dimension $r$ is known

Let

$$X^{SV} = U_{\text{full}}\Sigma_{\text{full}}V_{\text{full}}^T = \sum_{i=1}^{\min(m,n)} \sigma_i u_i v_i^T$$

denote the full SVD (Singular Value Decomposition) of $X$.

Here $\Sigma$ is a diagonal matrix with nonzero entries arranged in non-increasing order (largest one being the first and so on) and $U_{\text{full}}$ and $V_{\text{full}}$ are unitary matrices.

Since $U_{\text{full}}, V_{\text{full}}$ are unitary, we can rewrite both of them as

$$U_{\text{full}} = [U, U_\perp], \ V_{\text{full}} = [V, V_\perp]$$
with $U, V$ containing the first $r$ columns. Also we can let $\Sigma$ denote the top $r \times r$ block of $\Sigma_{\text{full}}$. Then $r$-SVD is

$$\hat{L} = U\Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

**9.2.1 Another way to understand the $r$-SVD solution**

The above is giving the “optimal” $r$-rank matrix approximation of $X$ - notice that $\hat{L}$ is specified by only $(n + m)r$ scalars and hence storing $L$ is much more efficient than storing $X$.

If we look at a row of $L$ it is still an $n$-length vector and there are $m$ rows. So what is going on? An alternative (easier-to-understand) approach to understand the PCA solution is to consider the matrix

$$B := XV = \hat{L}V = U\Sigma$$

this is an $m \times r$ matrix, each row of it is an $r$-dimensional vector of PCA “subspace coefficients” while the $n \times r$ matrix $V$ (with orthonormal columns) specifies the PCA “subspace”.

**9.2.2 Eigenvector interpretation of the same solution**

Observe that

- $V$ is also the matrix of top $r$ eigenvectors of $X^T X$. This is another way to compute $V$.
- We can then compute $B = XV$ which is the matrix of subspace basis coefficients.
- Finally, $\hat{L} = BV^T = XVV^T$ is the rank $r$ approximation of $X$ given above.

Also $\sigma_i^2$ is the $i$-th largest eigenvalue of $X^T X$.

**9.3 Optimality**

The above solution, $\hat{L}$, is optimal in various senses.

- It solves
  $$\min_{L: \text{rank } r} \|X - L\|_2$$

- It also solves
  $$\min_{L: \text{rank } r} \|X - L\|_F$$

- Thirdly, assume that the data vectors are “centered” (are zero mean random variables). Then $V$ (matrix of top $r$ right singular vectors) solves the following too: find an $n \times r$ matrix with orthonormal entries such that the sum of the empirically computed “variances” of each entry of the PCA projections, $b^{(i)} := V^T x^{(i)}$, is the largest. To be precise, $V$ solves
  $$\max_{\tilde{V}: n \times r \text{ matrix with } \tilde{V}^T \tilde{V} = I} \|\tilde{V}^T (X^T X) \tilde{V}\|_F^2$$

**9.3.1 Proof for the induced 2-norm optimality of $r$-SVD**

Goal is to find a rank $r$ matrix $L$ that solves

$$\min_{L: \text{rank } r} \|X - L\|_2$$

By Weyl-type inequality for singular values, $\sigma_i(M) \leq \sigma_i(M_2) + \|M - M_2\|$ for any singular value $i$ and any two matrices $M, M_2$.

Thus, $\|X - L\|_2 \geq \sigma_i(X) - \sigma_i(L)$.
Since $L$ is rank $r$, it has only $r$ nonzero singular values. Thus $\sigma_{r+1}(L) = 0$. And so $\|X - L\|_2 \geq \sigma_{r+1}(X)$. Since this is true for any rank $r$ matrix $L$, it is also true for the minimizer, i.e.

\[
\left( \min_{L: \text{rank } r} \|X - L\|_2 \right) \geq \sigma_{r+1}(X)
\]

Now if we can find a specific matrix $L$ for which $\|X - L\|_2 = \sigma_{r+1}(X)$ that will be the minimizer (since the minimum value cannot be any smaller than this).

If we let $\hat{L} = \sum_{i=1}^r \sigma_i u_i v_i^T$ (r-SVD of $X$), then $X - L = \sum_{i=r+1}^{\min(m,n)} \sigma_i u_i v_i^T$ and so $\|X - L\|_2 = \sigma_{r+1}$; here $\sigma_i = \sigma_i(X)$.

9.3.2 Proof for the variance optimality of r-SVD: rank $r = 1$ case

Goal is to find a $V$ solves

\[
\max_{\tilde{V}: n \times r \text{ matrix with } \tilde{V}^T \tilde{V} = I} \|\tilde{V}^T (X^T X) \tilde{V}\|_F^2.
\]

Rank one case: $V$ is a $n \times 1$ vector. Then the goal is to find

\[
\max_{\tilde{v}: \tilde{v}^T \tilde{v} = 1} \tilde{v}^T (X^T X) \tilde{v}.
\]

This is a constrained optimization problem with equality constraint. We can use KKT conditions to find the maximizer. Or simpler still, use the variational definition of the top eigenvalue/eigenvector. Using the latter, it is immediate that the solution to the above is the top eigenvector (eigenvector with largest eigenvalue) of $(X^T X)$ and the minimum value is the top eigenvalue.

9.4 PCA de-correlates the data – what does it mean?

This means that the correlation between the $j$-th and the $k$-th columns of the PCA subspace coefficients’ matrix, $B$, is zero for any $j \neq k$. To be precise,

\[
\langle b_j, b_k \rangle = \sum_{i=1}^m b_j^{(i)} b_k^{(i)} = 0.
\]

The above follows easily by noticing that $b_j = u_j \sigma_j$ and the $u_j$’s (columns of $U$) are orthonormal.

9.5 PCA - practical issues

- Mean subtraction: before performing r-SVD on the data matrix, it is a good idea to subtract the mean of the data vectors. This ensures that the principal directions of variation correspond to the directions of large variance in the data. This means, first compute $\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$ and the centered data vectors $\tilde{x}^{(i)} = x^{(i)} - \mu$. Do r-SVD on $Y$ to find the PCA subspace basis matrix $V$.

- Deciding $r$: so far in all the writing we have assumed that the desired lower-dimension $r$ is given. There is no one correct way of deciding $r$ in practice. A common heuristic is the 90% or some other high-enough percent heuristic: retain all eigenvectors so the that variance in the reduced dimensional space is at least 90% of the total variance of the data. In other words, find the smallest value $r$ so that

\[
\sum_{i=1}^r \sigma_i^2 \geq 0.9 \sum_{i=1}^{\min(m,n)} \sigma_i^2.
\]

10 Unsupervised Learning: Clustering

To Do - add introduction and k-means clustering
10.1 k-means clustering

To Do

10.2 EM algorithm

Consider Maximum Likelihood Estimation over i.i.d. data samples \(x^{(i)}\) coming a distribution \(p(x; \theta)\). The goal is to maximize the likelihood or equivalently its log, i.e.,

\[
\max _{\theta} \ell(\theta) := \sum _{i=1} ^{m} \log p(x^{(i)} ; \theta)
\]

For simple cases like Gaussian, this is easy to do (can get a closed form expression). But in other problems it is not easy, the cost function is not convex for example. In certain such settings, the EM algorithm helps simplify. Consider the Gaussian mixture model with \(k\) components. Then

\[
p(x; \theta) = \sum _{j=1} ^{k} \mathcal{N}(x; \mu _j, \Sigma _j) \phi _j
\]

where \(\phi_j\)'s are the mixture weights (probability of \(x\) coming from the \(j\)-th class in the mixture) and thus \(\sum _{j=1} ^{k} \phi _j = 1\). We often refer to \(j\) as the class labels. Thus,

\[
\ell(\theta) = \sum _{i=1} ^{m} \log \sum _{j=1} ^{k} \phi_j \mathcal{N}(x^{(i)}; \mu _j, \Sigma _j) \text{ s.t. } \sum _{j} \phi _j = 1
\]

and the parameters \(\theta\) are

\[
\theta = \{\mu _j, \Sigma _j, \phi _j\}, j = 1, 2, \ldots k
\]

with \(\mu _j\) being \(n \times 1\), \(\phi _j\) is a scalar and \(\Sigma _j\) is \(n \times n\).

We will try to lower bound \(\ell(\theta)\) and then maximize the lower bound by Alternating-Minimization. To do this we will first multiply divide the above expression by \(q_i(j)\) which are such that \(q_i(j) \ge 0\) and \(\sum _{j=1} ^{k} q_i(j) = 1\). So one can think of them as some probability distribution over the class labels.

\[
\ell(\theta) = \sum _{i=1} ^{m} \log \sum _{j=1} ^{k} \phi_j \mathcal{N}(x^{(i)}; \mu _j, \Sigma _j)
\]

\[
= \sum _{i=1} ^{m} \log \sum _{j=1} ^{k} \phi_j \mathcal{N}(x^{(i)}; \mu _j, \Sigma _j) \frac{q_i(j)}{q_i(j)}
\]

\[
= \sum _{i=1} ^{m} \log \sum _{j=1} ^{k} q_i(j) \frac{\phi_j \mathcal{N}(x^{(i)}; \mu _j, \Sigma _j)}{q_i(j)}
\]

\[
\ge \sum _{i=1} ^{m} \sum _{j=1} ^{k} q_i(j) \log \frac{\phi_j \mathcal{N}(x^{(i)}; \mu _j, \Sigma _j)}{q_i(j)}
\]

with equality holding if and only if \(q_i(j) = c \phi_j \mathcal{N}(x^{(i)}; \mu _j, \Sigma _j)\). The last step used log-sum (Jensen)'s inequality. Since the \(q_i(j)\)'s sum to 1, we get that

\[
c = \frac{1}{\sum _{j=1} ^{k} \phi_j \mathcal{N}(x^{(i)}; \mu _j, \Sigma _j)}.
\]

In all of above we also needed to keep the constraint \(\sum _{j} \phi_j = 1\) but we skipped it for ease of writing.
By the above approach, we have increased the number of unknowns - now the unknowns include \( q_i(j) \)'s and \( \theta \). BUT, the maximization over \( \theta \) given \( q_i(j) \)'s is easy (closed form); and the maximization over \( q_i(j) \)'s given \( \theta \) fixed is also easy (given above). Thus, we can do Alternating-Maximization over \( q_i(j) \)'s and \( \theta \). This gives us the EM algorithm for Gaussian Mixture Models. This is summarized next.

Iterate the following two steps until “convergence” (some reasonable stopping criterion holds):

- **E-step** (maximization over \( q_i(j) \)'s): as explained above, this is obtained by
  \[
  q_i(j) = \frac{\phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}{\sum_{j=1}^{k} \phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}, \quad j = 1, 2, \ldots, k, i = 1, 2, \ldots, m.
  \]

- **M-step** (maximize the lower bound expression over \( \theta \) holding \( q_i(j) \) fixed at the above value:
  \[
  \max_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{k} q_i(j) \log \frac{\phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}{q_i(j)}
  \]

  Since \( q_i(j) \)'s are held fixed, this is equivalent to
  \[
  \max_{\theta} \sum_{j=1}^{k} \sum_{i=1}^{m} q_i(j) \left[ \log \phi_j + \log \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j) \right] \quad \text{s.t.} \quad \sum_{j} \phi_j = 1
  \]

  This is easy, it is almost like GDA because it can be separated out over subsets of variables:
  
  - Solve for \( \phi_j \)'s
    \[
    \max_{\phi_1, \phi_2, \ldots, \phi_k} \sum_{j=1}^{k} q_i(j) \log \phi_j \quad \text{s.t.} \quad \sum_{j} \phi_j = 1
    \]
    This is solved by \( \hat{\phi}_j = \frac{1}{m} \sum_{i=1}^{m} q_i(j) \). This \footnote{we actually need \( c = \frac{1}{\sum_{j=1}^{k} \sum_{i=1}^{m} q_i(j)} \); not hard to see that this simplifies to \( 1/m \).}
  
  - For each \( j = 1, 2, \ldots, k \),
    \[
    \max_{\mu_j, \Sigma_j} \sum_{j=1}^{k} \sum_{i=1}^{m} q_i(j) \log \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)
    \]
    This is solved by
    \[
    \hat{\mu}_j = c \sum_{i=1}^{m} q_i(j) x^{(i)}
    \]
    and
    \[
    \hat{\Sigma}_j = c \sum_{i=1}^{m} q_i(j) (x^{(i)} - \hat{\mu}_j)(x^{(i)} - \hat{\mu}_j)^T
    \]
    with \( c = \frac{1}{\sum_{j=1}^{k} \sum_{i=1}^{m} q_i(j)} \).

### 10.3 General EM

In the above writing, I have written the above out for the Gaussian Mixture Model first to make it easier to understand. More generally, you could introduce any ”missing data” variables \( y^{(i)} \) and use a process similar
to the above. If the labels are discrete-valued, then we can write things as
\[
\ell(\theta) = \sum_{i=1}^{m} \log \sum_{j=1}^{k} p(x^{(i)}, y^{(i)} = j; \theta) \quad \text{s.t.} \sum_{j=1}^{k} q_i(y^{(i)} = j) = 1
\]
\[
= \sum_{i=1}^{m} \log \sum_{j=1}^{k} p(x^{(i)}, y^{(i)} = j; \theta) \frac{q_i(y^{(i)} = j)}{q_i(y^{(i)} = j)} \quad \text{s.t.} \sum_{j=1}^{k} q_i(y^{(i)} = j) = 1
\]
\[
\geq \sum_{i=1}^{m} \sum_{j=1}^{k} q_i(y^{(i)} = j) \log \frac{p(x^{(i)}, y^{(i)} = j; \theta)}{q_i(y^{(i)} = j)} \quad \text{s.t.} \sum_{j=1}^{k} q_i(y^{(i)} = j) = 1
\]

We now use Alternating-Maximization to maximize the above lower bound over \(q_i(y^{(i)} = j)\)’s keeping \(\theta\) fixed and vice-versa.

We know that the RHS cannot larger than its upper bound, thus if we can find a value that helps achieve the upper bound we are done. The following is easy to see, if \(\sum_j g_j = 1\), then \(\sum_j g_j \log c = \log \sum_j g_j c\). Thus the lower bound is maximized when
\[
q_i(y^{(i)} = j) = cp(x^{(i)}, y^{(i)} = j; \theta)
\]
Since the above sums to one over \(j\), we need \(c = 1/p(x^{i}; \theta)\) and so
\[
q_i(y^{(i)} = j) = p(y^{(i)} = j|x^{(i)}; \theta)
\]
Thus the general EM algorithm proceeds as follows.

- E step: keeping \(\theta\) fixed at its previous value, compute
  \[
  q_i(y^{(i)} = j) = p(y^{(i)} = j|x^{(i)}; \theta), j = 1, 2, \ldots, k, i = 1, 2, \ldots, m
  \]

- M step: keeping \(q_i(y^{(i)} = j)\) fixed at above value, compute
  \[
  \arg \max_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{k} q_i(y^{(i)} = j) \log \frac{p(x^{(i)}, y^{(i)} = j; \theta)}{q_i(y^{(i)} = j)}
  = \arg \max_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{k} q_i(y^{(i)} = j) \log p(x^{(i)}, y^{(i)} = j; \theta)
  = \arg \max_{\theta} \sum_{j=1}^{k} \sum_{i=1}^{m} q_i(y^{(i)} = j) \log p(x^{(i)}, y^{(i)} = j; \theta)
  \]

This is often easier if the parameters \(\theta\) have a mixture-model type form and the maximization can be separated out.

In more general settings \(y^{(i)}\) may be real-valued (continuous r.v.’s). In these cases, the summation over \(j\) gets replaced by integration, but a lot of the essential approach remains the same. A log-sum inequality exists for integrals also.
11 Deep Learning / Neural Networks: basic idea and training

This is another Supervised Learning approach.

The simplest neural net is the Feed-forward Network also called the Multilayer Perceptron or MLP. Each neuron receives a weighted sum of the outputs of the neurons of the previous layer, and applies a nonlinear “activation function on this. Thus neuron $j$ in layer $k$ receives

$$z^k_j = \sum_{i=1}^{r_k} w^k_{ij} o^{k-1}_i$$

as input and outputs

$$o^k_j = g(z^k_j)$$

Here $g(z)$ is an element-wise nonlinearity. It could be the sigmoid function $1/(1 + e^{-z})$ or the Rectified Linear Unit (ReLU) function $\max(z, 0)$ or the tanh function.

Vectorizing the above, the NN can be expressed as

$$o^k = g_{vec}(z^k), \quad z^k = W^k o^{k-1}$$

or equivalently,

$$z^k = W^k g_{vec}(z^{k-1})$$

Here $g_{vec}(z)$ applies $g(.)$ to each entry of the vector $z$.

The first layer takes the input $x$ as input thus

$$z^1 = x$$

Suppose the NN has 10 layers. The final (output) layer has only one neuron which outputs

$$\hat{y} = g(z^{10}_1)$$

Thus, the NN is

$$\hat{y} = g(W^{10}_{vec}(W^9_{vec}(W^8_{vec}(\ldots g_{vec}(x)))))$$

11.1 Training: Back-propagation

Given training data $(x^{(i)}, y^{(i)})$, $i = 1, 2, \ldots, m$, we use gradient descent or stochastic / mini-batch GD to train. For all of these, the first task is to define the cost function $E(\hat{y}(x), y)$ and to compute its gradient w.r.t. to each weight in each layer, i.e., compute

$$\frac{\partial E}{\partial w^k_{ij}}$$

This computation requires careful application of chain rule of differentiation. This leads to the following algorithm: consider a 10 layer NN and let $r_k$ denote the number of neurons in layer $k$

- For a given input $x$, compute the outputs of all the layers and $\hat{y}$.
- Compute the following intermediate quantity:

$$\delta^k_i := \frac{\partial E}{\partial z^k_i}$$

using the following backward recursion (back-propagation)
compute
\[ \delta_{1}^{10} = \frac{\partial E}{\partial \hat{y}}(\hat{y}, y) \cdot g'(z_{1}^{10}) \]

here \( g'(z) = \frac{\partial g(z)}{\partial z} \)

- for each \( k = 9, 8, \ldots, 1 \), compute the following for each \( i = 1, 2, \ldots, r_k \):
\[ \delta_{i}^{k} = g'(z_{i}^{k}) \sum_{j=1}^{r_{k+1}} w_{ij}^{k+1} \delta_{j}^{k+1} \]

vectorized computation in MATLAB (or do similar in Python): \( \delta_{k}^{i} = g'(z_{k}^{i}) \cdot (W_{k}^{k+1} \delta_{k}^{k+1}) \)

- Compute
\[ \frac{\partial E}{\partial w_{ij}^{k}} = \delta_{j}^{k} \cdot \delta_{i}^{k-1} \]

this can be vectorized too.

The above gives us \( \frac{\partial E}{\partial w_{ij}^{k}}(\hat{y}(x), y) \) for one input-output pair \( x, y \). The gradient w.r.t. the cost function that uses all the training data is thus
\[ \frac{1}{m} \sum_{i=1}^{m} \frac{\partial E}{\partial w_{ij}^{k}}(\hat{y}(x^{(i)}), y^{(i)}) \]

11.2 Convolutional Neural Network (CNN or ConvNet): basic idea

To Do

11.3 Recurrent Neural Network (RNN): basic idea

To Do

11.4 Different NN architectures and when to use each

see the other handout NeuralNets-intro.pdf

12 Bias-Variance Tradeoff

12.1 What is it?

Consider a generative model: suppose that \( y, x \) satisfy
\[ y = f(x) + e, \quad \mathbb{E}[e] = 0, \mathbb{E}[e^2] = \sigma^2 \]

with \( e \) being zero mean “modeling error” / “noise” that is independent of \( x \), it is also independent for each data point \( y_i \).

We do not know \( f(.) \).

We try to “model” it as \( \hat{y} = \hat{f}(x) = h_{\hat{\theta}}(x) \), e.g., in linear regression, \( \hat{f}(x) = h_{\hat{\theta}}(x) = \hat{\theta}^T x \) with \( \hat{\theta} \) estimated by Maximum Likelihood estimation (MLE) as described earlier using training data
\[ (y^i, x^i), i = 1, 2, \ldots, m \]

In logistic regression, \( \hat{f}(x) = h_{\hat{\theta}}(x) = g(\hat{\theta}^T x) \) with \( g(.) \) being the sigmoid function. MLE uses “training data”
The question is how good is my learnt model (in terms of mean squared error on test data), i.e., for a test query $x$, what is $\mathbb{E}[(y - \hat{f}(x))^2]$ and what can we do to improve it?

We define Test-MSE as

$$\text{Test-MSE} := \mathbb{E}[(y - \hat{y})^2] = \mathbb{E}[(y - \hat{f}(x))^2] = \mathbb{E}[(f(x) + e - \hat{f}(x))^2]$$

over test data, i.e., $\mathbb{E}[:]=\mathbb{E}_{\text{test data}}[:].$

Since $e$ is test-data noise, it is independent of $\hat{f}(x) = h_\hat{\theta}(x)$ since $\hat{\theta}$ was estimated using training data. Also by assumption, it is independent of $f(x)$. Thus, we have

$$\text{Test-MSE} = \mathbb{E}[e^2] + \mathbb{E}[(f(x) - \hat{f}(x))^2]$$

$$= \mathbb{E}[e^2] + \mathbb{E}[f(x) - \hat{f}(x)]^2 + \text{Variance}[f(x) - \hat{f}(x)]$$

$$= \sigma^2 + \text{Bias}^2 + \text{Variance}$$

The first term depends on how noisy the data is. The second two terms depend on the “assumed model” and how its parameters are estimated.

### 12.2 Approximating Test-MSE in practice

While we can write things as above, it is not actually possible to compute the above decomposition for test data. All one can do is the following: for a given model, one can approximate Test-MSE using the following approach

- Split available training data into training and test data: in other words do not use all $m$ data points to train, split them so that $m = m_{\text{train}} + m_{\text{test}}$.
- Use the $m_{\text{train}}$ data points to train, i.e. to estimate $\theta$ for the assumed model
- Approximate Test-MSE by $\frac{1}{m_{\text{test}}} \sum_{j=1}^{m_{\text{test}}} (y_j - h_\hat{\theta}(x_j))^2$

The above is one way to do what is called “Cross-Validation”. Typically, one uses $m_{\text{test}} = 0.25m$ and $m_{\text{train}} = 0.75m$ or similar.

**Leave-one-out Cross-Validation:** Do above with $m_{\text{test}} = 1$, and $m_{\text{train}} = m - 1$, but repeat the procedure $m$ times to compute the average error (Test-MSE)

For some other approaches to do the above, see the handout on Cross-Validation.

### 12.3 How to reduce test data MSE

Try to reduce variance, while hopefully not increasing the bias too much:

- Manually reduce the number of features $n$ or use domain knowledge to do this
- Use PCA on the feature data-set available for training followed by using the reduced dimensional "features" for the regression (or other task). Pick the reduced dimension $r$ carefully – see PCA section.
- Regularization – add extra constraints on $\theta$, for example, model $\theta$ as being sparse: get the LASSO in case of linear regression: add $||\theta||_1$ into the cost function for finding $\theta$.
- Regularization – suppose prior knowledge is available that $\theta$ is close to a given vector $\theta_0$, then add the $||\theta - \theta_0||_2^2$ into the cost function for finding $\theta$.
- Naive Bayes assumption described earlier is another way to reduce $n$, and hence the variance in the parameter estimates.
In case of unsupervised learning: in PCA – reduce \( r \) reduces “variance” in the estimate of the principal subspace; in case of Gaussian Mixture Model, reducing \( k \) (number of classes) will reduce the variance, increasing it will reduce the bias. So the solution can be: keep increasing \( k \) until Test-MSE begins to decrease.

With each new intervention, compute the Test-MSE as explained above, see if it gets reduced or not. What is expected is this: if we increase \( n \) starting at \( n = 0 \), the bias will reduce significantly up to a certain value of \( n \) (and variance will not increase too much), so that Test-MSE will reduce. After a certain point, variance will start increasing significantly compared to the further reduction in bias. This is the point to stop.

12.4 Mathematical explanation: Bias-Variance Tradeoff for Linear Regression

Suppose that \( y \) truly satisfies the following model

\[
y = f(x) + e, \quad f(x) := \theta_{\text{full}}^T x_{\text{full}}, \quad \mathbb{E}[e] = 0, \quad \text{Var}[e] = \sigma^2
\]

where \( x_{\text{full}} \) is the full \( n_{\text{full}} \) length “feature” or data. Also, the noise \( e \) is i.i.d. across various samples.

For the sake of tractability (reducing variance), when “modeling” \( y \), we throw away some of the features to get an \( n \)-length “feature” vector \( x \); and refer to the dropped part of \( x_{\text{full}} \) as \( x_{\text{drop}} \). This is a \( n_{\text{drop}} \) length vector. Thus \( n_{\text{full}} = n + n_{\text{drop}} \).

\[
x_{\text{full}}^T = [x^T, x_{\text{drop}}]
\]

Thus, \( y \) can be rewritten as

\[
y = \theta^T x + \theta_{\text{drop}}^T x_{\text{drop}} + \mu + e.
\]

Thus, the ”linear regression” model for \( y \) is \( \theta^T x \). Assume as before we are given training data \( \{y^{(i)}, x^{(i)}\}, i = 1, 2, \ldots, m \), and define the matrix \( X \) and the vector \( y \) as before.

We use MLE under this model to get the MLE estimate

\[
\hat{\theta} := (X^T X)^{-1} X^T y.
\]

Thus, for a query \( x_{\text{full}} \), with extracted features \( x \), we predict

\[
\hat{y} = x^T \hat{\theta}
\]

The true output, \( y \), for the query satisfies

\[
y = \theta^T x + \theta_{\text{drop}}^T x_{\text{drop}} + e
\]

The bias is

\[
\text{Bias} := \mathbb{E}[\hat{y}] - \mathbb{E}[y]
\]

and the variance is

\[
\text{Var} := \mathbb{E}[(\hat{y} - \mathbb{E}[\hat{y}])^2]
\]

To compute these two, observe that \( \hat{\theta} \) satisfies

\[
\hat{\theta} := (X^T X)^{-1} X^T (X \theta + \mu + e), \quad \mu := X_{\text{drop}} \theta_{\text{drop}}
\]

where \( X_{\text{drop}} := [x^{(1)}_{\text{drop}}; x^{(2)}_{\text{drop}}; \ldots; x^{(m)}_{\text{drop}}] \) is an \( m \times n_{\text{drop}} \) matrix with the dropped parts of each training data vector as its rows. Thus,

\[
\mathbb{E}[\hat{\theta}] = \theta + (X^T X)^{-1} X^T X_{\text{drop}} \theta_{\text{drop}}
\]
and so

\[ \text{Bias} := \mathbb{E}[\hat{y}] - \mathbb{E}[y] = x^T(\theta + (X^T X)^{-1}X^T X_{\text{drop}}\theta_{\text{drop}}) - x^T\theta - x_{\text{drop}}^T \theta_{\text{drop}} = x^T(X^T X)^{-1}X^T X_{\text{drop}}\theta_{\text{drop}} - x_{\text{drop}}^T \theta_{\text{drop}} = [x^T(X^T X)^{-1}X^T X_{\text{drop}} - x_{\text{drop}}^T] \theta_{\text{drop}} \]

Variance: Since the noise \(e\) was zero mean and i.i.d., we have \(\mathbb{E}[ee^T] = \sigma^2 I\). Using this,

\[ \text{Var} := \mathbb{E}[(\hat{y} - \mathbb{E}[\hat{y}])^2] = x^T(X^T X)^{-1}X^T ee^T X(X^T X)^{-1}x = x^T(X^T X)^{-1}x\sigma^2 \]

The variance is upper bounded by

\[ \text{Var} \leq \frac{\|x\|^2}{\lambda_{\text{min}}((X^T X))} \sigma^2 \]

Thus the variance is smaller if the minimum eigenvalue of \((X^T X)\) is larger. Notice that \(X\) is an \(m \times n\) matrix, thus the minimum eigenvalue is proportional to \(m/n\), the larger this ratio, the larger will be the min eigenvalue. Thus for a fixed available amount of training data, \(m\), the Variance will be reduced if we use a smaller \(n\).

Consider the Bias. This is some complicated function of \(\theta_{\text{drop}}\). We can simplify it by making more assumptions, but that is not needed. What is clear is that this will become larger as \(n_{\text{drop}}\) is made larger (or equivalently, as \(n = n_{\text{full}} - n_{\text{drop}}\) is made smaller). Thus with reducing \(n\), Variance decreases but Bias increases.

### 13 Learning Theory

?? replace indicator function by sign function

Hypothesis refers to a hypothesized model on the input output data. So suppose we assume that \(y = 1(\theta^T x)\) then \(1(\theta^T x)\) is the hypothesis.

Hypothesis class \(\mathcal{H}\) is the set of all hypothesis from a certain class, e.g., set of all linear classifiers is

\[ \mathcal{H} = \{ h : h = 1(\theta^T x), \ \forall \ \theta \in \mathbb{R}^n \} \]

Empirical Risk (Training Error) is denoted by \(\hat{\varepsilon}(h)\). For a loss function \(\text{loss}(y, \hat{y})\), it is computed as

\[ \hat{\varepsilon}(h) = \frac{1}{m} \sum_{i=1}^{m} \text{loss}(y^i, h(x^i)) \]

Generalization Error is denoted by \(\varepsilon(h)\). It is the expected loss for a query sample

\[ \varepsilon(h) = \mathbb{E}[\text{loss}(y, h(x))] \]

Assumptions

- Training and Test data are generated from the same probability distribution
- All training samples as well as test sample are mutually independent. The two assumptions combined mean that all training and test data are i.i.d. (independent identically distributed).

---

2 To understand this with a concrete example, if the training data vectors were i.i.d. standard Gaussian, then with high probability, the minimum eigenvalue is lower bounded by \((\sqrt{m} - C\sqrt{n})^2\). Reference: Vershynin tutorial.
Under the above assumptions, by using a law of large numbers' result, it can usually be argued that $\hat{\epsilon}(h)$ converges to $\epsilon(h)$ in probability as $m$ goes to infinity. We will look at a simple zero-one loss function and actually do this.

Empirical Risk Minimization (ERM) means minimize the empirical risk over all hypotheses from a certain class, i.e., try to find

$$\hat{h} = \arg \min_{h \in \mathcal{H}} \hat{\epsilon}(h)$$

Ideally what we would like to find is the hypothesis that minimizes the generalization error

$$h^* = \arg \min_{h \in \mathcal{H}} \epsilon(h)$$

Thus the minimum generalization error is $\epsilon(h^*)$, i.e.,

$$\epsilon(h^*) = \min_{h \in \mathcal{H}} \epsilon(h)$$

We would like to use $h^*$ but we cannot compute it. We instead use $\hat{h}$ on the test data too. The question is how much worse is this? I.e., how much worse is $\epsilon(\hat{h})$ compared to $\epsilon(h^*) = \min_{h \in \mathcal{H}} \epsilon(h)$? We work this out for the zero-one loss

### 13.1 Misclassification (zero-one) loss

Let us specialize to the misclassification (zero-one) loss:

$$\text{loss}(y, \hat{y}) = 1(y \neq \hat{y}) = 1(y \neq h(x))$$

For this, the empirical risk for a given $h$ is

$$\hat{\epsilon}(h) = \frac{1}{m} \sum_{i=1}^{m} 1(y^i \neq h(x^i))$$

while

$$\epsilon(h) = \mathbb{E}[1(y \neq h(x))] = \Pr(y \neq h(x))$$

Define the r.v. $Z_i = 1(y^i \neq h(x^i))$. Clearly the $Z_i$ are Bernoulli with probability of one equal to $\epsilon(h)$. Using Hoeffding’s inequality, this means that, for a given hypothesis $h$,

$$\Pr(|\hat{\epsilon}(h) - \epsilon(h)| > \gamma) \leq 2e^{-2\gamma^2 m}$$

Suppose for a moment that the size of the hypothesis class is $k$, i.e.,

$$|\mathcal{H}| = k.$$  

Then, by Union Bound,

$$\Pr(|\hat{\epsilon}(h) - \epsilon(h)| > \gamma, \text{ for some } h \in \mathcal{H}) \leq 2ke^{-2\gamma^2 m}$$

or equivalently

$$\Pr(|\hat{\epsilon}(h) - \epsilon(h)| < \gamma, \text{ for ALL } h \in \mathcal{H}) \geq 1 - 2ke^{-2\gamma^2 m}$$

For this probability to be at least $1 - \delta$, we need to set $\gamma = 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}$. With this we can rewrite things as follows

With probability at least $1 - \delta$, for all $h \in \mathcal{H}$, with $k = |\mathcal{H}|$,

$$|\hat{\epsilon}(h) - \epsilon(h)| \leq \gamma := \sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}$$
Since above is true for all \( h \), it is true for \( \hat{h} \) and \( h^* \) too. So with the above probability,

\[
\varepsilon(\hat{h}) \leq \hat{\varepsilon}(\hat{h}) + \gamma \leq \hat{\varepsilon}(h^*) + \gamma \leq \varepsilon(h^*) + \gamma + \min_{\hat{h} \in \mathcal{H}} \varepsilon(h) + 2\gamma
\]

The first and third inequalities used the Hoeffding bound from above; the second used the fact that \( \hat{\varepsilon}(\hat{h}) \) is the minimum over all \( h \).

Thus, substituting for \( \gamma \), we can write the following theorem

**Theorem 13.1.** Consider a hypothesis class \( \mathcal{H} \) with \( |\mathcal{H}| = k \). And consider zero-one loss. Suppose that \( \hat{h} \) minimizes the empirical risk.

With probability \( \geq 1 - \delta \),

\[
\varepsilon(\hat{h}) \leq \left( \min_{h \in \mathcal{H}} \varepsilon(h) \right) + 2\sqrt{\frac{1}{2m} \log \left( \frac{2k}{\delta} \right)}
\]

This theorem works for a finite hypothesis class, but not for an infinite one, for example it does not work for the class of all linear classifiers. However since everything is done on a computer (say one on which each real number is represented by 64 bits), we can assume that there are really only a finite number of degrees of freedom. Consider a hypothesis class with \( d \) parameters all of which are real numbers. Then the total number of possible options for hypotheses is

\[
k = 2^{64} \cdot 2^{64} \cdots 2^{64} = 2^{64d}
\]

We have the following corollary

**Corollary 13.2.** Consider a hypothesis class \( \mathcal{H}_d \) with \( d \) real-valued parameters. Assume implementation on a 64-bit computer so that \( k = |\mathcal{H}| = 2^{64d} \). Consider zero-one loss. Suppose that \( \hat{h} \) minimizes the empirical risk. Also set \( \delta = 2^{-d} \). Then, we can claim the following

With probability \( \geq 1 - 2^{-d} \),

\[
\varepsilon(\hat{h}) \leq \left( \min_{h \in \mathcal{H}_d} \varepsilon(h) \right) + C \sqrt{\frac{65d + 1}{2m}}
\]

In most of this class we have assumed \( n \) parameters, i.e., \( d = n \).

Implications: if we increase \( d \), the size of the hypothesis class increases, this means, in the first term we are taking a minimum over a larger set. Thus the first term will reduce if \( d \) increases. But the second term clearly increases (for a fixed \( m \)).
HW 5 (extra credit): 1. Re-do the "learning" part of HW 1 (linear regression) – for learning we are trying to estimate an nx1 vector $\theta$ using m training data samples, $x^i, y^i$. Suppose that the training data comes in sequentially; and/or suppose we want to keep updating our model as more training data comes in.

Consider the following setting: we initially have $m_0$ training samples, from which we can get an estimate $\hat{\theta}^{m_0}$ by regular LS.

Now suppose after this, we get one $x^i, y^i$ pair at a time. This means the matrix $X$ is growing in terms of number of rows and so is the vector $y$. Use Recursive LS to get $\theta_i$. Start with $\hat{\theta}^{m_0}$ and $\Pi = (X_0^T X_0)^{-1}$ where $X_0$ is the $m_0 \times n$ matrix formed by the initial training samples.

2. In this homework itself or HW 2, re-visit the case where you do not have enough training samples – m is small. In this case use a prior with $\hat{\theta}_0 = 0$ and $\Pi = (1/\epsilon)I$, with $\epsilon = 0.00001$ or some small number.

Distill out some topics from my Estimation/Detection notes: MMSE estimation; Least Squares, Regularized LS, and Recursive LS; Kalman Filter; HMM. Teach the problem setting and algorithm; skip the proofs.

One/two lecture version of Sparse Recovery, Matrix Completion (Low-Rank Matrix Recovery), Robust PCA, Phase Retrieval