Summary Notes
EE425X - Machine Learning: A Signal Processing Perspective

Namrata Vaswani
Iowa State University

These notes are a work in progress.For the latest version at any time, see the link `https://www.dropbox.com/scl/fi/nd7buqtksc55lei8e1k6z/ML_algorithms.pdf?rlkey=8e3oczn35ybwulvb82ecu5fcm&dl=0`

Acknowledgement

Some of these notes are based on material presented in this Stanford Machine Learning class `http://cs229.stanford.edu/syllabus.html`. Rest of the notes are based on material I have learned over the years from various sources.

Contents

Final Exam - Spring 2024 EE 425

- I will only accept handwritten exam solutions
- Allowed to only use ML-algorithms.pdf file. Need to do it individually (no group work). No use of internet.
- Take home exam. We will allow any 24 hrs during finals' week.
- Sections
  - 2.1, 2.2 (Cross validation)
  - 3.1 (Gradient Descent, basic GD only)
  - 4.2 (Supervised learning probabilistic model)
  - 5 Linear Reg (entire section),
  - 6.1, 6.2 (two class Logistic Reg),
  - 7.1, 7.2, 7.3.1 (Bayesian learning, GDA, spam filter basic),
  - 8.1, 8.2, 8.4, 8.7 (SVM basic ideas and soft-margin SVM),
  - 10.1 (Lasso and Ridge regression),
  - 11.1, 11.2, 11.3 (PCA),
  - 12.1, 12.2 (Clustering problem and k-means clustering)
  - 12.3 (AltMin algorithm)
- Topics that are good to know but will not be on the exam
  - Sec 1 Data simulation
  - Sec 3.3 Optimization overview
  - Sec 6.3 Multi-class Log Reg
  - 16.1, 16.2, 16.3, 16.4 (Regularization - practical ideas).
  - Sec 13 Deep learning MLP and CNN - top level ideas.

Why above policy?

- Writing by hand helps you remember things and sometimes helps really understand things.
- You could also potentially gain the same understanding by yourself Latex'ing the solutions; but I can't prevent you from finding all this info online. Hence this policy.
- Internet has a lot of information, not all of it is correct.

Notation

In these notes $'$ and ^{T} and $^\top$ are all used to denote vector or matrix transpose. At a few other places, MATLAB notation is used too.

Math basics

Probability - See file 322-recap- and cs229-

Linear Algebra - See cs229-linalg

1 Data Simulation Basics

1.1 Why simulate – why test code on simulated data first?

Consider the house price prediction based on house features example. In the previous section, we said we use linear regression to model the data and predict the price. If I write code to learn theta and apply it to a real dataset directly and suppose it does not "work too well" (my code does not give very good predictions on test data). How do I know if (i) the linear regression model is wrong or (ii) my learning approach (normal equations or GD) is wrong, or (iii) there is a code bug (I have an extra factor of 2 at some place by mistake)?

A partial fix to the above problem to above it is to first simulate data using the linear regression model. So when I test my learning code on this data I know what is true θ I am looking for. Then I can try to fix (ii) and (iii) issues. Once these are fixed, then we try the same code on real data and then maybe compare with another model to check which one is better.

1.2 Monte Carlo Estimation of Expected values

Suppose the goal is to compute/approximate $\mathbb{E}[z]$ (expected value or average). Suppose also we know the probability distribution of z, $p(z)$, and we have a way to generate (simulate) independent realizations of z. Then we can approximate

$$\mathbb{E}[z] \approx \frac{1}{m}\sum_{i=1}^{m} z^{(i)}, \ z^{(i)} \overset{\text{i.i.d.}}{\sim} p(z)$$

We use this to approximate prediction error in ML among other things.

1.3 Understanding a multivariate Gaussian distribution

See these two files `http://cs229.stanford.edu/section/gaussians.pdf`
`http://cs229.stanford.edu/notes2020fall/notes2020fall/cs229-prob.pdf`

1.4 How to simulate

Task: Generate your own data to simulate the linear regression model $y = \boldsymbol{\theta}^T \boldsymbol{x} + e$.

Recall: $n = d + 1$, thus θ is a vector of length n. The $\boldsymbol{x}^{(i)}$s are of length d with 1 appended as first entry.

As an example, suppose $d = 5$, then $n = 6$. Can set $\theta = [10, 1, -1, -3, 4, 2]^\top$.

Generate m such independent training data vectors. Also generate another set of m_{test} independent data vectors for the testing step.

Let $\mathcal{N}(\mu, \boldsymbol{\Sigma})$ denote the Gaussian distribution with mean μ and covariance $\boldsymbol{\Sigma}$.

Do this as follows:
(1) Fix $\boldsymbol{\theta}$ once as an n length vector.
(2) Training data generate:
For $i = 1$ to m, generate $\boldsymbol{x}^{(i)} \sim \mathcal{N}(0, I_d)$, $e^{(i)} \sim \mathcal{N}(0, \sigma_e^2)$ and $y^{(i)} = \boldsymbol{\theta}^T[1, \boldsymbol{x}^{(i)}] + e^{(i)}$.
(3) Test data generate:
In a different Loop, repeat (2) for $i = 1$ to m_{test}. Use this data in the testing step and not for training.

No-for-loop version of (2):
$\boldsymbol{X} = [ones(m, 1), randn(m, d)]$, $\boldsymbol{e} = \sigma_e * randn(m, 1)$, $\boldsymbol{y} = \boldsymbol{X}\theta + \boldsymbol{e}$.

1.4.1 Notes

For generating the data, I have suggested using the Gaussian distribution just as an example. But you do not have to use the Gaussian. You could also use any other distribution, e.g., the uniform distribution.

On the other hand, for the error $e^{(i)}$, we have assumed that it is Gaussian in our model (that is why the squared loss is justified). So for generating error, you *do have to* use the Gaussian distribution.

1.5 Estimation error computations: how to know your code is correct

- Always compute normalized errors: if code works well error should be less than 1. When increase m (number of samples), while keeping everything else fixed, this error should reduce.
- If data were simulated, then one can compute (normalized) parameter estimation error $\frac{\|\theta-\hat{\theta}\|_2^2}{\|\theta\|_2^2}$
- In all cases, one can compute (normalized) prediction error / training loss: $\frac{(1/m)\sum_{i=1}^{m}(\boldsymbol{y}^{(i)}-\hat{\boldsymbol{y}}^{(i)})^2}{(1/m)\sum_{i=1}^{m}(\boldsymbol{y}^{(i)})^2}$
- Normalized Test MSE / Generalization Error computation:
  - Simulated data: Generate an independent set of test data
  - Real data: Two options
    - Hold-out Cross Validation: Split available data into disjoint training and test sets, use most data for training, rest for testing.
    - Or, use Leave-One-Out Cross-Validation, see Sec. 2
  - For each test data point, compute $\hat{\boldsymbol{y}}_{test} = h_{\hat{\theta}}(\boldsymbol{x}_{test})$
  - Compute normalized average squared error $\frac{\sum_{i=1}^{m_{test}}(\boldsymbol{y}_{test}^{(i)}-\hat{\boldsymbol{y}}_{test}^{(i)})^2}{\sum_{i=1}^{m_{test}}(\boldsymbol{y}_{test}^{(i)})^2}$
- For different problems, squared error can be replaced by other error measures.

2 Cross Validation for Supervised Learning: training and test datasets

In case of supervised learning, we need to define training and test datasets. These could be non-overlapping (easiest to code, analyze, fastest to run) or overlapping (needed if less total data).

2.1 Simple Commonly Used Approach: Hold-Out Cross Validation

Simplest way to define training and test datasets is to use a certain fraction of the data for training and the rest for testing. A common rule of thumb is 80-20 or 70-30. In this case training and test data are disjoint which makes it easier to impose statistical independence assumptions. Also this is easiest to code in and the code runs fastest.

2.2 Leave-one-out cross validation: any general task

Use of distinct subsets is easiest to code in but wastes a lot of data. The other extreme solution is to use leave-one-out cross-validation. This means we loop over all the data multiple times. In the i-th loop, we use all but the i-th data point for training and use the i-th one for testing. Compute error on this i-th one and store it. At the end of this loop, average the stored errors from each loop to compute the average generalization error / predictor error.

- for $i = 1$ to m_{tot}
  - at iteration i, define the training data matrix $\boldsymbol{X}$ and vector $\boldsymbol{y}$ as follows: use *all* data except the i-th

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

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}^{(1)} \\ \boldsymbol{y}^{(2)} \\ \vdots \\ \boldsymbol{y}^{(i-1)} \\ \boldsymbol{y}^{(i+1)} \\ \vdots \\ \boldsymbol{y}^{(m)} \end{bmatrix}$$

  - Learn the model parameters use $\boldsymbol{X}, \boldsymbol{y}$ as the training data
  - Use the i-th data points for computing the error $err(i)$.

end for

Compute

$$TestMSE = \frac{1}{m_{tot}} \sum_{i=1}^{m_{tot}} err(i), NormalizedTestMSE = \text{as needed for the problem}$$

2.2.1 Leave-one-out cross validation for Linear regression

Given m training data points $\boldsymbol{x}^{(i)}, \boldsymbol{y}^{i}i$, $i = 1, 2, \ldots, m$. Suppose the goal is to evaluate the validity of a linear regression model on this data. We will compute Normalized-Test-MSE as follows.

- for $i = 1$ to m_{tot}
  - at iteration i, define the training data matrix $\boldsymbol{X}$ and vector $\boldsymbol{y}$ as follows: use *all* data except the i-th. This is defined above.

$$\boldsymbol{X} = \begin{bmatrix} -(\boldsymbol{x}^{(1)})^T- \\ -(\boldsymbol{x}^{(2)})^T- \\ \vdots \\ -(\boldsymbol{x}^{(i-1)})^T- \\ -(\boldsymbol{x}^{(i+1)})^T- \\ \vdots \\ -(\boldsymbol{x}^{(m)})^T- \end{bmatrix}, \ \boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}^{(1)} \\ \boldsymbol{y}^{(2)} \\ \vdots \\ \boldsymbol{y}^{(i-1)} \\ \boldsymbol{y}^{(i+1)} \\ \vdots \\ \boldsymbol{y}^{(m)} \end{bmatrix}$$

  - Compute $\hat{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$
  - $\hat{\boldsymbol{y}}^{(i)} = \hat{\theta}^T\boldsymbol{x}^{(i)}$
  - Compute $err(i) = (\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)})^2$

end for

Compute

$$TestMSE = \frac{1}{m_{tot}} \sum_{i=1}^{m} err(i), \ NormalizedTestMSE = TestMSE/(\sum_{i=1}^{m_{tot}} (\boldsymbol{y}^{(i)})^2)$$

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 $\boldsymbol{X}$ has full rank n.

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

$$J(\theta) := \sum_{i=1}^{m} cost(\theta, \boldsymbol{x}^{(i)}, y^{(i)})$$

where *cost* is squared error (as in previous section) or can be something else (as in later sections on Logistic Regression).

```
1. Initialize $\hat{\theta}$ as the zero vector
   (or anything else: this choice is problem dependent, for linear regression zero vector is good)
2. For $t = 1$ to $Tmax$ do
   (use a large value for $Tmax$)
   (a) $\hat{\theta}_{old} \leftarrow \hat{\theta}$
   (b) $\hat{\theta} \leftarrow \hat{\theta} - \mu \nabla_\theta J(\hat{\theta})$
   (c) Exit loop if $\|\hat{\theta} - \hat{\theta}_{old}\|_2 / \|\hat{\theta}_{old}\|_2 \leq \epsilon$ with $\epsilon$ a very small tolerance, e.g., set $\epsilon = 10^{-6}$ (this value actually also depends on noise level)
   End for
```

3.1 Setting the step size μ

- When using $J(\theta)$ as above (it is the sum of m terms), then use $\mu = c/m$ with c being a number between 0 and 1.
- Rule of thumb: Reduce c it if the cost seems to not decrease at all or starts increasing. Increase it if the cost decreases but very slowly with iteration.
  - Detailed: start with $c = 0.5$, check if error goes down with iterations, if not it is too large. Use a smaller c say $c = 0.05$. Check again. If still does not go down, use an even smaller step size.
- As you reduce c, you will need MORE total number of iterations.
- Good coding practice: Always use a T_{max}: maximum number of iterations and an exit criterion.
- Extra info: *Variable step size: Can also vary μ with iterations, start with a larger value but reduce with iterations.*

3.2 Extra material: Stochastic GD (SGD) and Stochastic Mini-Batch GD **

$J(\theta)$ is typically an average 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 divided by m.

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

Stochastic GD idea: sum over a subset of the m gradients at each iteration. Pick this subset randomly or use other strategies.

Details: see videos mentioned next.

3.2.1 Main idea

Let

$$J(\theta) := \sum_{i=1}^{m} J_i(\theta; \boldsymbol{x}^{(i)}, y^{(i)})$$

- Initialize $\hat{\theta}$.
- Repeat
  - Randomly shuffle samples in the training set.
  - Loop through all points in the training set one at a time.
    - For $i = 1, 2, \ldots, m$, do:
$$\hat{\theta} \leftarrow \hat{\theta} - \eta \nabla J_i(\hat{\theta})$$

  until stopping criterion reached

3.2.2 Good videos on stochastic and mini-batch SGD

Basic GD video `https://www.youtube.com/watch?v=yFPLyDwVifc`

Stochastic mini-batch GD `https://www.youtube.com/watch?v=4qJaSmvhxi8`

3.3 Optimization overview

Consider differentiable cost functions.

Unconstrained opt: solve

$$\min_{\theta} f(\theta)$$

A necessary condition for $\hat{\theta}$ to be a minimizer is

$$\frac{\partial f}{\partial \theta}(\hat{\theta}) = \mathbf{0}$$

Constrained opt: solve

$$\min_{\theta} f(\theta), \ \ s.t. \ \boldsymbol{h}(\theta) = \mathbf{0}, \ \boldsymbol{g}(\theta) \preceq 0$$

Here $\boldsymbol{z} \preceq \mathbf{0}$ means $\boldsymbol{z}_i \leq 0$ for all entries of i of the vector $\boldsymbol{z}$.

Under certain "regularity conditions" (sometimes called constraint qualifications), a set of necessary conditions for $\hat{\theta}$ to be a local minimizer is as follows: these are called *KKT conditions.* These regularity conditions hold for *convex optimization problems.* They also hold for other problems.

Let $\boldsymbol{\mu}, \boldsymbol{\lambda}$ be Lagrange multipliers for the inequality and equality constraints respectively. Define the Lagrangian as

$$L(\theta, \boldsymbol{\lambda}, \boldsymbol{\mu}) := f(\theta) + \boldsymbol{\lambda}^\top \boldsymbol{h}(\theta) + \boldsymbol{\mu}^\top \boldsymbol{g}(\theta)$$

1. We need
$$\frac{\partial L}{\partial \theta}(\hat{\theta}) = \mathbf{0} \quad \Leftrightarrow \quad \frac{\partial f}{\partial \theta}(\hat{\theta}) + \boldsymbol{\lambda}^\top \frac{\partial \boldsymbol{h}}{\partial \theta}(\hat{\theta}) + \boldsymbol{\mu}^\top \frac{\partial \boldsymbol{g}}{\partial \theta}(\hat{\theta}) = \mathbf{0}$$
2. The problem constraints should hold – called Primal feasibility
$$\boldsymbol{h}(\hat{\theta}) = \mathbf{0}, \ \text{ and } \ \boldsymbol{g}(\hat{\theta}) \preceq 0$$
3. The Lagrange multipliers for the inequality constraints are non-negative – called Primal feasibility
$$\boldsymbol{\mu} \succeq 0$$

4. Complementary slackness holds: if an inequality constraint is *not* satisfied with equality, then the corresponding Lag multiplier is zero.

$$\boldsymbol{\mu}_i \boldsymbol{g}_i(\hat{\theta}) = 0 \ \forall i = 1, 2, \ldots, m$$

In some texts this condition is written in a more compact (but more confusing-looking) fashion as

$$\boldsymbol{\mu}^\top \boldsymbol{g}(\theta) = \mathbf{0}$$

This and previous one are equivalent because we are assuming $\boldsymbol{g}(\theta) \preceq 0$ and $\boldsymbol{\mu} \succeq 0$. The first implies $\boldsymbol{\mu}^\top \boldsymbol{g}(\theta) \leq 0$. Since $\boldsymbol{\mu}_i \geq 0$ the only way the sum will be zero is if each term in the sum is zero.

3.3.1 Convex opt problem

A problem is convex if the cost is convex and the feasibility set (set of θs in the constraint set) is convex. One can show that this is the same as requiring

1. the inequality constraint functions $\boldsymbol{g}(\theta)$ are convex functions, i.e. $\boldsymbol{g}_i(\theta)$ is convex function for each i
2. the cost function $f(\theta)$ is convex function
3. equality constraints $\boldsymbol{h}(\theta)$ are affine, i.e., $\boldsymbol{h}(\theta) = \boldsymbol{A}\theta + \boldsymbol{b}$ for some matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$.

3.3.2 Convex function

A function is convex if for any $0 \leq \alpha \leq 1$

$$f(\alpha\theta_1 + (1-\alpha)\theta_2) \leq \alpha f(\theta_1) + (1-\alpha) f(\theta_2)$$

3.3.3 Convex set

A set is convex if θ_1, θ_2 in the set implies that $\alpha\theta_1 + (1-\alpha)\theta_2$ is also in the set for any $0 \leq \alpha \leq 1$.

4 Supervised Learning, Unsupervised Learning, Model-based Learning: basics

Given observed data (or features of the observed data) or other "input" $\boldsymbol{x}$, the goal in ML is to predict/compute some function of $\boldsymbol{x}$ that is usually denoted by y (often called "output"). In most of this course, $\boldsymbol{x}$ is an $n \times 1$ vector and y is a scalar. For example, $\boldsymbol{x}$ 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. As a different example, $\boldsymbol{x}$ can be a vectorized image while y is the class label for the image (dog vs cat or dog vs cat vs human etc).

In supervised 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). All of this is done so that the "learnt model" can be used to predict y (get $\hat{y}$) for a new query $\boldsymbol{x}$. "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 $\boldsymbol{x}$.

$\boldsymbol{x}$ is $n \times 1$, y is a scalar. We use θ to denote the set of parameters used by our assumed model.

In many settings, the assumed model that predicts y is denoted $h_\theta(\boldsymbol{x})$. Since the model is never perfect, we assume that the "true" output y satisfies

$$y = h_\theta(\boldsymbol{x}) + e$$

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 $\{\boldsymbol{x}^{(i)}, y^{(i)}\}, i = 1, 2, \ldots, m$. The modeling error / noise e We use these to "learn" θ. Once that is done, we can predict y from $\boldsymbol{x}$ using the above equation.

4.1 Examples

Examples of supervised learning include: linear regression, classification - logistic regression or GDA or naive-bayes/spam-filter, SVMs, deep learning.

In case of unsupervised learning, we are just provided data and we need to "make sense" of it. This can mean different things. In case of PCA this means finding a lower dimensional subspace to represent the data. In case of clustering, this means partition the data into K disjoint "clusters"; K itself may be known or unknown.

In Model-based Learning, the observed data model is provided (usually the Physics of the problem defines this). Examples include least squares (LS) problems, Bayesian LS, Bayesian estimation, or structured data recovery problems such as sparse recovery (compressive sensing) or low-rank matrix recovery. Standard LS is usually the oversampled data setting. Bayesian LS or structure models are used in case of undersampled data - some data is missing, corrupted, or deliberately undersampled.

The model learning (regression coefficients' vector learning) step of linear regression is an instance of LS. If limited training data is available, and we model the regression vector as being sparse, this can become an instance of sparse recovery.

Unsupervised deep learning, which is a newish area, in which one uses a single training data (that is a sequence of vectors or images or image Fourier transforms) to both learn the deep model parameters and predict the output - falls in the second or third category. Used in accelerated dynamc MRI applications.

4.2 Supervised Learning pipeline: probabilistic models

All learning assumes a model. Supervised learning also assumes availability of training data. Supervised learning itself can be probabilistic or not (SVM is not for example). Goal of learning: given input (feature vector) $\boldsymbol{x}$, predict output y.

- Decide Model:
  We first decide a modeling strategy to model the input-output relationship: how $\boldsymbol{x}$ and y are related
  - in classical (non-Bayesian) learning, this is specified by $p(y; \boldsymbol{x}, \theta)$

$$LinReg: \; p(y; \boldsymbol{x}, \theta) = \mathcal{N}(y; \theta^\top \boldsymbol{x}, \sigma_e^2)$$

$$(Binary) LogReg: \; p(y; \boldsymbol{x}, \theta) = g(\theta^\top \boldsymbol{x})^y [1 - g(\theta^\top \boldsymbol{x})]^{1-y}$$

$$(MultiClass) LogReg:$$

  - in generative (Bayesian) learning, this is specified by $p(y; \theta), p(\boldsymbol{x}|y; \theta)$ which can then specify $p(y|\boldsymbol{x})$ (Bayes rule)

$$GDA: p(y; \theta) = \prod_{k=1}^{K-1} \phi_k^{[y==k]}, \; p(\boldsymbol{x}|y; \theta) = \mathcal{N}(\boldsymbol{x}; \mu_y, \Sigma)$$

- Learning:
  We use training data set $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}_{i=1}^m$ to learn the model parameters θ. Arrange these into matrix $\boldsymbol{X}$ and vector $\boldsymbol{y}$ Learning is done using Max Likelihood Estimation (MLE).

  - Classical case: MLE finds

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

  - Bayesian case: MLE finds

$$\arg\max_{\theta} p(\boldsymbol{y}, \boldsymbol{X}; \theta) := \prod_{i=1}^{m} p(\boldsymbol{y}^{(i)}, \boldsymbol{x}^{(i)}; \theta) = \prod_{i=1}^{m} p(\boldsymbol{y}^{(i)}; \theta) p(\boldsymbol{x}^{(i)} | \boldsymbol{y}^{(i)}; \theta)$$

- Prediction:
  Given a query $\boldsymbol{x}$, we use the learned model parameters, $\hat{\theta}$, and the model to predict $\hat{y}$
  - Classical case: Max Likelihood for finding y

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

  - Bayesian case: Max A Posteriori (MAP) rule for finding y

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

- IMPORTANT: finding the argument to maximize a function is the same as finding the argument that minimizes its negative logarithm. Always use $-\log(.)$ for products of PMFs or PDFs - simpler expressions for deriving closed-form solutions; and this is also needed to prevent overflow/underflow in computer code.

To add: PDF, PMF difference and similarity, Bayes rule; Examples of each kind - Lin Reg, Log Reg, GDA, Spam filter

4.3 Supervised Learning: not probabilistic models

- SVM:

5 Supervised Learning: Linear Regression

In the setting we have talked about in class, $\boldsymbol{x}$ is a real-valued $n \times 1$ vector and y is a real-valued scalar [1]. The parameter vector θ is also an $n \times 1$ vector

5.1 Model

In linear regression, $h_\theta(\boldsymbol{x})$ is a linear function of $\boldsymbol{x}$.

$$h_\theta(\boldsymbol{x}) = \theta^T \boldsymbol{x}.$$

(I sometimes may use $'$ for transpose – MATLAB notation)

[1]In more general settings y can also be a real-valued vector (this will not be discussed in our class).

5.2 Including a nonzero mean in the model

To include a non-zero mean in the model, one can replace both $\boldsymbol{x}$ and θ by $n+1$ length vectors as follows. Let

$$\tilde{\theta} = \begin{bmatrix} \theta_0 \\ \theta \end{bmatrix}$$

and

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix}$$

and we let

$$h_{\tilde{\theta}}(\tilde{\boldsymbol{x}}) = \tilde{\theta}^T \tilde{\boldsymbol{x}}.$$

With this model, we do everything explained above using $\tilde{\boldsymbol{x}}$ and $\tilde{\theta}$ to replace θ and $\boldsymbol{x}$ respectively.

Here and below, we let θ be an $n = d+1$ length vector and same for $\boldsymbol{x}^{(i)}$ (append 1 as its first entry).

5.3 Learning θ: minimize squared loss

Here and below, we let θ be an $n = d+1$ length vector and same for $\boldsymbol{x}^{(i)}$ (append 1 as its first entry).

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

$$\arg\min_{\theta} J(\theta) := \sum_{i=1}^{m} (y^{(i)} - \theta^\top \boldsymbol{x}^{(i)})^2.$$

Define an $m \times n$ input data matrix $\boldsymbol{X}$ with $[1, (\boldsymbol{x}^{(i)})^T]$ as its rows, i.e., let

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

and define an $m \times 1$ vector $\boldsymbol{y}$ with $y^{(i)}$ as its columns.

Then, $J(\theta)$ can be expressed more compactly as

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

5.4 Solutions for minimizing squared loss: also called Least Squares (LS) Estimation

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

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

   Details: $\nabla J(\theta) = 2\boldsymbol{X}^\top(\boldsymbol{y} - \boldsymbol{X}\theta)$. Set this equal to zero and solve for $\hat{\theta}$. Reason: the gradient is zero at every minimizer, maximizer or stationary point of a differentiable cost function. gradient = 0 is a necessary condition for a point to be a minimizer. in this case, since $J(.)$ is strongly convex it has exactly one stationary point which is THE minimizer. Thus setting $\nabla J(\theta) = 0$ helps us find THE minimizer.

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

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

Extra details: The pseudo-inverse can be computed more efficiently than in the formula above: this is computed by first computing the singular value decomposition (SVD) of the matrix $\boldsymbol{X}$. Suppose the SVD of $\boldsymbol{X} = U\Sigma V^\top$ where U is an $m \times m$ unitary matrix. V is an $n \times n$ unitary matrix and Σ is diagonal with non-negative entries. Then $\boldsymbol{X}^\dagger = V\Sigma^\dagger U^T$. For a rectangular diagonal matrix such as Σ, 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.

Do not use the pinv function of numpy though. That gives wrong output for $m = n$.

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 $\boldsymbol{X}$ is "well-conditioned", both approaches above should 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 in Sec. 3.

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.

5.5 Prediction step: using the learned θ

Once we have learned $\hat{\theta}$ using training data, given a new feature vector, $\boldsymbol{x}$, we can predict y as

$$\hat{y} = [1, \boldsymbol{x}^\top]^\top \hat{\theta}$$

5.6 Understanding squared loss: Maximum Likelihood Estimation under i.i.d. Gaussian model **

The above can be motivated as Maximum Likelihood Estimation under the following probabilistic model for the data $\{y, \boldsymbol{x}\}$

$$p(y; \boldsymbol{x}, \theta) = \frac{1}{\sqrt{2\pi}\sigma_e} \exp((y - \boldsymbol{x}^\top\theta)^2/2\sigma_e^2)$$

Thus given m training data points all of which are independent identically distributed (i.i.d.), we have the following model: $y^{(i)} = h_\theta(\boldsymbol{x}^{(i)}) + e^{(i)},\ i = 1, 2, \ldots, m$ with $e^{(i)}$ standard Gaussian. Here the randomness is only in the noise e. In other words

$$\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{X}\theta, \sigma_e^2\boldsymbol{I})$$

or that

$$p(\boldsymbol{y}; \boldsymbol{X}, \theta) = \frac{1}{(\sqrt{2\pi}\sigma_e)^m} \exp\left(-\frac{\|\boldsymbol{y} - \boldsymbol{X}\theta\|^2}{2\sigma_e^2}\right)$$

Maximum Likelihood Estimation (MLE) means: find the value of θ that maximizes $p(\boldsymbol{y}; \boldsymbol{X}, \theta)$ (above probability density function). This is equivalent to minimizing the negative logarithm of the above PDF. It can be seen that this is equivalent to minimizing $J(\theta)$ given above and repeated here

$$J(\theta) = \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2$$

5.7 Least Squares, Regularized LS, Recursive LS and more

See Sec. 10.

6 Supervised Learning: Two class and Multi-class Logistic Regression

In this case, $\boldsymbol{x}$ is still a real-valued $d \times 1$ vector but now y is a scalar. The goal is classification. For two-class case, y takes values either 0 or 1.

6.1 Two-class Model and its probabilistic interpretation

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

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

$g(.)$ 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,

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

The prediction is

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

Thus

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

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

6.1.1 Use of bias term in Logistic regression

Introduce the bias term exactly as we did in case of linear regression Make both θ and $\boldsymbol{x}$ $n = (d+1)$ length vectors; set the first entry of $\boldsymbol{x}$ equal to 1. The first entry of θ is then the bias term.

6.1.2 Learning θ: Maximum Likelihood Estimation

Again define $\boldsymbol{y}$ and $\boldsymbol{X}$ 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 $e^{(i)}$'s be i.i.d.).

Thus, we minimize the negative log likelihood,

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

We can simplify this a lot as follows.

$$\begin{aligned} J(\theta) &= -\sum_{i=1}^{m} \log \left(\left(\frac{1}{1 + \exp(\theta' \boldsymbol{x}^{(i)})} \right)^{y^{(i)}} \left(\frac{\exp(\theta' \boldsymbol{x}^{(i)})}{1 + \exp(\theta' \boldsymbol{x}^{(i)})} \right)^{1-y^{(i)}} \right) \\ &= -\sum_{i=1}^{m} \log \left(\left(\frac{1}{1 + \exp(\theta' \boldsymbol{x}^{(i)})} \right)^{y^{(i)}} \left(\frac{1}{1 + \exp(-\theta' \boldsymbol{x}^{(i)})} \right)^{1-y^{(i)}} \right) \end{aligned}$$

The only difference between the first and second terms inside log(.) is the sign of $\theta' \boldsymbol{x}^{(i)}$ in the denominator and the power it is raised to. One is raised to the power $y^{(i)}$, the other is raised to the power $(1 - y^{(i)})$. Here $y^{(i)}$ is either 0 or 1. So, $2y^{(i)} - 1$ is either -1 (when $y^{(i)} = 0$) or $+1$ (when $y^{(i)} = 1$). Thus, when $y^{(i)}$ takes only values 0 or 1,

$$\left(\frac{1}{1 + \exp(\theta' \boldsymbol{x}^{(i)})} \right)^{y^{(i)}} \left(\frac{1}{1 + \exp(-\theta' \boldsymbol{x}^{(i)})} \right)^{1-y^{(i)}} = \frac{1}{1 + \exp\left((\theta' \boldsymbol{x}^{(i)})(2y^{(i)} - 1) \right)}$$

The reason the last equality is true is because when $y^{(i)} = 1$, $2y^{(i)} - 1 = 1$, but when when $y^{(i)} = 0$, $2y^{(i)} - 1 = -1$.

We now simply get

$$J(\theta) = -\sum_{i=1}^{m} \log \left(\frac{1}{1 + \exp\left((\theta' \boldsymbol{x}^{(i)})(2y^{(i)} - 1)\right)} \right) = \sum_{i=1}^{m} \log \left(1 + \exp\left((\theta' \boldsymbol{x}^{(i)})(2y^{(i)} - 1)\right) \right)$$

We can find θ by minimizing $J(\theta)$ by GD. The gradient is specified in the summary section given below.

It is possible to show that $J(\theta)$ is convex. Argument: weighted sum convex functions is convex when the weights are positive, here the weights are just 1; $(\theta' \boldsymbol{x})(2y - 1)$ is an affine function of θ (and hence is both convex and concave; the logistic function $\log(1 + \exp(-z))$ can be shown be convex (see next line); composition of a convex function and an affine function is convex. To show $\log(1+\exp(-z))$ is convex, since it is twice differentiable, we can compute the second derivative and show that it is $-\frac{1}{1+\exp(-z))^2} < 0$ for any z, thus it is convex everywhere.

6.2 Two class Logistic Regression summary

Let θ be a $n = d+1$ length vector with the first entry corresponding to the bias term. Replace $\boldsymbol{x}$ by $[1, \boldsymbol{x}^\top]^\top$ (this it makes it a $n = d + 1$ length vector

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

$g(z) = \frac{1}{1+\exp(-z)} = \frac{\exp(z)}{\exp(z)+1}$ is called the sigmoid function, $0 \leq g(z) \leq 1$ for all z.

- Let θ be a $n = d + 1$ length vector with the first entry corresponding to the bias term. Replace $\boldsymbol{x}$ by $[1, \boldsymbol{x}^\top]^\top$ (this it makes it a $n = d + 1$ length vector
- Given training data $\boldsymbol{X}, \boldsymbol{y}$, use cost func as negative log likelihood. Assume independence of different data pairs.

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

- We can estimate θ by minimizing $J(\theta)$ by GD. After simplification the gradient has the following expression:

$$\nabla J(\theta) = \sum_{i=1}^{m} (g(\theta^\top \boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)}) \boldsymbol{x}^{(i)}$$

  - Use algorithm given in Sec. 3 with above gradient

- Classification: Given a query, $\boldsymbol{x}$, predict y as

$$\hat{y} = 1 \text{ if } \theta^\top \boldsymbol{x} > 0, \quad \hat{y} = 0 \text{ otherwise}$$

- Note: do not use the above model for simulating data on which to test the algorithm: for simulating data, better to use the GDA generative model.

6.3 Multi-class Logistic Regression**

Multi-class logistic regression (LR) is known by a variety of other names, including polytomous LR, Multinomial LR, softmax regression, multinomial logit (mlogit), the maximum entropy (MaxEnt) classifier, and the conditional maximum entropy model.

We now have $K > 2$ classes. Recall that, for $K = 2$, the model is

$$\Pr(y=1) = \frac{1}{1+\exp(-\theta^\top \boldsymbol{x})} = \frac{\exp(\theta^\top \boldsymbol{x})}{1+\exp(\theta^\top \boldsymbol{x})}$$

and

$$\Pr(y=0) = 1 - \frac{\exp(\theta^\top \boldsymbol{x})}{1+\exp(\theta^\top \boldsymbol{x})}$$

To extend this to K classes labeled $k = 0, 1, \dots, K-1$, instead of a single "regression vector" θ, we now use $K-1$ "regression" vectors $\beta_k, k = 1, 2, \dots K-1$ and use the model:

$$\Pr(y=k) = \frac{\exp(\beta_k^\top \boldsymbol{x})}{1+\sum_{k'=1}^{K-1}\exp(\beta_{k'}^\top \boldsymbol{x})}, \quad k = 1, 2, \dots, K-1$$

and

$$\Pr(y=0) = \frac{1}{1+\sum_{k'=1}^{K-1}\exp(\beta_{k'}^\top \boldsymbol{x})}$$

If we let $\beta_0 = \mathbf{0}$ the $y = 0$ case can be interpreted as

$$\Pr(y=0) = \frac{\exp(\beta_0^\top \boldsymbol{x})}{1+\sum_{k'=1}^{K-1}\exp(\beta_{k'}^\top \boldsymbol{x})}, \text{ with } \beta_0 = \mathbf{0}$$

With setting $\beta_0 = \mathbf{0}$, observe that $1 = \exp(\beta_0^\top \boldsymbol{x})$. Replace this in both numerator and denominator of above expressions. Then, the above can be simplified to

$$\Pr(y=k) = \frac{\exp(\beta_k^\top \boldsymbol{x})}{\sum_{k'=0}^{K-1}\exp(\beta_{k'}^\top \boldsymbol{x})}, \quad k = 1, 2, \dots, K-1$$

- Learning the parameters:
  - Need to estimate $\theta := \{\beta_1, \beta_2, \dots, \beta_{K-1}\}$.
  - Solution 1: define the joint log-likelihood function over all m training data points and maximize it (minimize its negative).
  - This solution often does not work too well (too many parameters and one may start becoming too large).
  - Solution 2: add regularization on the β_k's, e.g. add $\lambda \sum_{k=1}^{K-1} \|\beta_k\|^2$ to the cost function and then minimize it
  - Basic GD may not work easily in this complicated setting. To explore this case, see what Python or MATLAB's built in toolboxes do for your data.
- Classification: find the k for which $\Pr(y = k)$ is highest.
  Equivalently, taking log of the probabilities, using $\beta_0 = \mathbf{0}$ (zero vector), and ignoring the denominator, this translates to
  $$\hat{y} = \arg\max_{k=0,\dots,K-1} \beta_k^\top \boldsymbol{x}$$

6.4 One-vs-rest (OVR) or One-vs-all use of two-class logistic regression**

There are two ways to deal with multiple classes using Log Reg. Say we have K classes, $k = 0, 1, \ldots, K-1$.

1. Option 1: use what I have above
2. Option 2: use two class log reg in one-vs-rest or OVR mode. Idea of OVR:
   - First learn a θ_0 for separating classes into 0 and everything else (combine classes 1,2,3, ... K-1 into the "not-zero" class)
   - Then learn a θ_1 for separating classes into 1 and everything else (combine classes 0,2,3, ... K-1 into the "not-one" class)
   - Then learn a θ_2 for separating classes into 2 and everything else (combine classes 0,1,3, ... K-1 into the "not-two" class)
   - And so on.
   - Output of above set of steps will be K regression vectors $\theta_0, \theta_1, \ldots \theta_{K-1}$
   - Classification on a query $\boldsymbol{x}$:
   $$\hat{y} = \arg \max_{k=0,1,2,\ldots,K-1} \theta_k^\top \boldsymbol{x}$$
3. More details: see the video below
4. Also see `https://en.wikipedia.org/wiki/Multiclass_classification`

Video on One-vs-Rest (or One-vs-All) use of two-class Log Reg:
Lecture 6.7 of
`https://www.youtube.com/watch?v=PPLop4L2eGk&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN`

7 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 $\boldsymbol{x}$, with $\boldsymbol{x}$ 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 $\boldsymbol{x}$ 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 $\boldsymbol{x}$ using Bayes rule. Mathematically, we assume that we are given

$$p(\boldsymbol{x}|y;\theta), p(y)$$

and we use these to obtain the prediction as follows

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

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.

7.1 Learning θ: Maximum Likelihood Estimation

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

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

or, usually its logarithm, and maximize it over θ.

7.2 Generative Learning: Gaussian Discriminant Analysis (GDA)

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

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

Notice in this case $\theta = \{\mu_0,\mu_1,\boldsymbol{\Sigma}_0,\boldsymbol{\Sigma}_1,\phi\}$. Parameters are still learnt by MLE

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

Notice that, without extra assumptions, we have $2n+2n^2+1$ parameters. Training data are each n length vectors $\boldsymbol{x}^{(i)}$, 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 $\boldsymbol{x}^{(i)}$ are independent conditioned on the class label $y^{(i)}$. This is called the *Naive Bayes assumption.*

7.2.1 Gaussian Discriminant Analysis (GDA) with Naive Bayes assumption and equal covariances

A common model simplification is to assume that the different entries of each $\boldsymbol{x}^{(i)}$ 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 Σ_0, Σ_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 Σ is diagonal. With this assumption too, we have the following simpler model

$$\prod_{i=1}^m \left(\left(\prod_{j=1}^n \mathcal{N}(\boldsymbol{x}_j;(\mu_{y_i})_j,\sigma_j^2) \right) \cdot \phi^{y_i}(1-\phi)^{1-y_i} \right)$$

Under the above assumption, the Max Likelihood Estimates (MLE) of the model parameters, $\theta := \{\phi,\mu_0,\mu_1,\Sigma\}$ i.e., the value of the model parameters that solve

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

are computed as follows:

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^m \mathbf{1}(y^{(i)}=1)$$

$$\hat{\mu}_0 = \frac{\sum_{i=1}^m \mathbf{1}(y^{(i)}=0)\boldsymbol{x}^{(i)}}{\sum_{i=1}^m \mathbf{1}(y^{(i)}=0)}$$

$$\hat{\mu}_1 = \frac{\sum_{i=1}^m \mathbf{1}(y^{(i)}=1)\boldsymbol{x}^{(i)}}{\sum_{i=1}^m \mathbf{1}(y^{(i)}=1)}$$

$$\hat{\sigma_j^2} = \frac{1}{m}\sum_{i=1}^m (\boldsymbol{x}^{(i)} - \mu_{y^{(i)}})_j^2,\ j = 1,2,\ldots,n$$

while setting all non-diagonal entries of $\hat{\Sigma}$ to be zero. Here $\mathbf{1}$ denotes the *indicator function* of the statement in paranthesis. Thus, $\mathbf{1}(y^{(i)} = 0)$ 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 $j = 1, 2, \dots, n$, compute

$$(\hat{\mu}_0)_j = \frac{\sum_{i=1}^m \mathbf{1}(y^{(i)} = 0)(\boldsymbol{x}^{(i)})_j}{\sum_{i=1}^m \mathbf{1}(y^{(i)} = 0)}$$

$$(\hat{\mu}_1)_j = \frac{\sum_{i=1}^m \mathbf{1}(y^{(i)} = 1)(\boldsymbol{x}^{(i)})_j}{\sum_{i=1}^m \mathbf{1}(y^{(i)} = 1)}$$

$$\hat{\sigma_j^2} = \frac{1}{m}\sum_{i=1}^m (\boldsymbol{x}^{(i)} - \mu_{y^{(i)}})_j^2, \ j = 1, 2, \dots, n$$

In this Naive Bayes' setting, the selecting of $\hat{y}$ to solve

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

simplifies to the following test (just simplify the expressions)

$$\hat{y} = 1 \text{ if } \sum_{j=1}^n \frac{(\boldsymbol{x} - \hat{\mu}_1)_j^2}{\hat{\sigma}_j^2} - \sum_j \frac{(\boldsymbol{x} - \hat{\mu}_0)_j^2}{\hat{\sigma}_j^2} < \mathit{thresh}$$

with *thresh* depending on $\hat{\phi}$. If $\hat{\phi} \approx 0.5$ then $\mathit{thresh} = 0$.

7.2.2 Notes

1. In the Gaussian case, we get a simple closed form as above. In many cases of Bayesian modeling also, this is possible. In general when this is not possible, *do not* ever work directly with probabilities. *Always work with logarithms of the probabilities.* Otherwise you will run into numerical problems while coding. Remember: under the naive Bayes assumption, $p(\boldsymbol{x}|y;\hat{\theta}) = \prod_{j=1}^n p(\boldsymbol{x}_j|y;\hat{\theta})$ if $p(.)$ is a PMF, it is a product of n real numbers all less than one. Even if $p(.)$ is a PDF (which could be more than one), very often it is actually less than one only. Product n numbers less than one can become very small very soon.

2. When working with real data, e.g., images, there may be a certain region that is black in *all* the images. For these pixels the estimate of the variance will be zero meaning $\sigma_j^{-1} = \infty$ resulting in code bugs.

   (a) Fix 1: do not use these directions.

   (b) Fix 2: Better fix to deal with similar issues where in some directions variance is very small (may not be zero): use PCA to reduce the dimensionality of the data. For classification there is no need to use all n features.

   (c) Fix 3: add a small value to replace the zeros: the added value should be *much smaller* than any of the *important directions' variances.* This is easy in the zero/nonzero case but in practice ill-conditioning of Σ may make this hard to do. Use of PCA and a smaller dimension is thus a much better fix.

7.3 Generative Learning: Spam Filter – an example of Discrete-valued or Categorical Features

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

$y = 0$ means the email is not-spam, $y = 1$ means it is spam.

7.3.1 Simple Spam Filter: entries of x are binary

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

Consider first the simplest model where $\boldsymbol{x}$ is a binary vector. As before, we specify

$$p(\boldsymbol{x}|y;\theta),\ p(y)$$

and we predict

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

Notice now that $\boldsymbol{x}$ 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 $\boldsymbol{x}^{(i)}$, 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.

7.3.2 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 $\boldsymbol{x}_j$'s, $j = 1, 2, \ldots, n$) given y.* Mathematically, we are assuming

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

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 $\boldsymbol{\Sigma}_0$, $\boldsymbol{\Sigma}_1$ are **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.

7.3.3 Simple Spam filter with Naive Bayes

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

$$\psi_{j,y} := p(\boldsymbol{x}_j = 1|y), j = 1, 2, \ldots, n;\ \textit{and}\ \phi := p(y = 1)$$

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

We can again learn the parameters by MLE:

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

s.t. constraints that

$$0 \le \psi_{j,y} \le 1,\ 0 \le \phi \le 1$$

Can again get closed form simple expressions for the MLE:

$$\hat{\psi}_{j,0} =$$

count the number of training data points for which $y^{(i)} = 0$ and $\boldsymbol{x}_j^{(i)} = 1$ and divide by the total number of training data points for which $y^{(i)} = 0$.

Similarly

$$\hat{\psi}_{j,1} =$$

count the number of training data points for which $y^{(i)} = 1$ and $\boldsymbol{x}^{(i)} = 1$ and divide by the total number of training data points for which $y^{(i)} = 1$. and

$$\hat{\phi} =$$

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

Use of n to be dictionary size makes it extremely large causing the algorithm to be very slow. Also, instead of letting n be the size of the English dictionary, we can let n be the size of the vocabulary (set of all words in all training data). But this has the disadvantage that it does not tell you how to deal with an unseen word. Options include (i) ignore unseen words (can be problematic in case the unseen words are the only reason an email is obviously spam); or (ii) in the model, assume a small nonzero probability for an unseen word (this means: increase the vocabulary size by 1, this probability cannot be learned easily, you just have to make up a "reasonable" value for it).

7.3.4 Spam Filter: entries of z count the number of times a word occurs

The simplest spam filter explained above is not using word-counts (the number of times a word occurs), but just checking whether a word is present or not. A better model is to consider a different feature vector $\boldsymbol{z}$ with $\boldsymbol{z}_j$ being the number of times word j occurs in the vocabulary (or dictionary).

d: number of words in any email – to use this model, we can use blank-space as one "word".

n: number of words in dictionary

m: number of training emails

Treating each word occurrence as independent (this is the naive Bayes assumption, it is actually not true since certain pairs of words are much more likely occur together, but simplifies our modeling), and assuming as before that $\psi_{j,y} = \Pr(\text{word } j \text{ is present in the email}|y)$, the feature vector $\boldsymbol{z}$ would be modeled by what is called a "multinomial distribution" as follows. This assumes that there are a total of d words in the sample, i.e., that $\sum_j \boldsymbol{z}_j = d$ for all training emails. *To use this model, we can use blank-space as one "word".*

$$p(\boldsymbol{z}|y) = \binom{d}{\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_n} \prod_{j=1}^{n} \psi_{j,y}^{\boldsymbol{z}_j}$$

This is called the multinomial distribution with parameters $\psi_{j,y}$.

MLE The joint likelihood of m independent training samples can be expressed as follows:

$$J(\theta) = \prod_{i=1}^{m} p(y^{(i)}) p(\boldsymbol{z}^{(i)}|y^{(i)}) = \prod_{i=1}^{m} \phi^{y^{(i)}} (1-\phi)^{(1-y^{(i)})} \binom{d}{\boldsymbol{z}_1^{(i)}, \boldsymbol{z}_2^{(i)}, \ldots, \boldsymbol{z}_n^{(i)}} \prod_{j=1}^{n} \psi_{j,y^{(i)}}^{\boldsymbol{z}_j^{(i)}}$$

Learning parameters: we need to maximize the above over $\phi, \psi_{j,y}$ subject to the constraints that

$$0 \le \psi_{j,y} \le 1, \; 0 \le \phi \le 1$$

It can be shown that we get

$$\hat{\psi}_{j,0} = \frac{\sum_{i=1}^{m} \boldsymbol{z}_j^{(i)} \mathbb{1}(y^{(i)} = 0)}{\sum_{j=1}^{n} \sum_{i=1}^{m} \boldsymbol{z}_j^{(i)} \mathbb{1}(y^{(i)} = 0)}$$

In words this is the total number of times word j occurs in **all** emails that are not spam (for which $y^{(i)} = 0$) divided by the total number of words in **all** emails that are not spam. We can similarly compute

$$\hat{\psi}_{j,1}$$

7.3.5 Spam filter: model 3

A third possible model is as follows. Let j referred to the j-th word in the email and let $\boldsymbol{x}_j$ be the location of the j-th word in the dictionary. So then this becomes "categorical data". We explain below how to deal with categorical data. In this case the length of the feature vector can be different for different emails. It will be $n^{(i)}$ if the email has $n^{(i)}$ distinct words.

7.3.6 More ideas and related problems

A more advanced document model models co-occurrences of words.

Similar ideas are also used to classify blogs or webpages into various categories as well.

7.4 Modeling Categorical data **

In some problems, the different features may be "categorical" instead of binary, which means feature j takes one of K_j possible values. For example $\boldsymbol{x}_j$ could be color of the front door of a house in case of the house price example and we assume $K_j = 5$ possible colors for example. There is no ordering to which color is preferred, thus the integer labels are arbitrary; they do not have a numerical meaning.

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

Our data model then is as follows

$$p(\boldsymbol{x}|y) = \prod_{j=1}^{n} \prod_{k=1}^{K_j} \psi_{j,y,k}^{[\boldsymbol{x}_j == k]}$$

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

$$\prod_i \left(\prod_{j=1}^{n} \prod_{k=1}^{K_j} (\psi_{j,y^{(i)},k})^{[\boldsymbol{x}_j^{(i)} == k]} \phi^{y^{(i)}} (1-\phi)^{1-y^{(i)}} \right)$$

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

Then MLE estimates are given by

$$\hat{\psi}_{j,0,k} =$$

counts the number of times $y^{(i)} = 0$ and $\boldsymbol{x}_j^{(i)} = k$ 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 $\boldsymbol{x}_j$, $j = 1, 2, \ldots, n$.

Do the same for class label 1.

7.5 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 μ_1 (depending on the class label) and covariance matrix Σ. By Naive Bayes, Σ is diagonal.

Then, joint likelihood becomes

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

s.t. sum to one constraints on ϕ and on all the $\psi_{j,y,k}$'s.

The MLE estimates are computed as follows.

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{1}(y^{(i)} = 1)$$

For $j = 1, 2, \ldots, r$,

$$(\hat{\mu}_0)_j = \frac{\sum_{i=1}^{m} \mathbf{1}(y^{(i)} = 0)(\boldsymbol{x}^{(i)})_j}{\sum_{i=1}^{m} \mathbf{1}(y^{(i)} = 0)}$$
$$(\hat{\mu}_1)_j = \frac{\sum_{i=1}^{m} \mathbf{1}(y^{(i)} = 1)(\boldsymbol{x}^{(i)})_j}{\sum_{i=1}^{m} \mathbf{1}(y^{(i)} = 1)}$$
$$\hat{\sigma_j^2} = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{x}^{(i)} - \mu_{y^{(i)}})_j^2,\ j = 1, 2, \ldots, n$$

For $j = r+1, r+2, \ldots, n$,

$$\hat{\psi}_{j,0,k} =$$

counts the number of times $y^{(i)} = 0$ and $\boldsymbol{x}_j^{(i)} = k$ 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 $\boldsymbol{x}_j$, $j = 1, 2, \ldots, n$.

8 Supervised Learning: Linear Classifiers and Support Vector Machines (SVMs)

8.1 Linear Classifiers **

Both logistic regression and GDA with Naive Bayes and equal covariances result in a linear classifier, i.e., one can simplify the classification rule in both cases to get the following:

$$\hat{y} = 1 \text{ if } \boldsymbol{w}^T\boldsymbol{x} + b > 0$$

and equals 0 otherwise.

Proof for GDA:

GDA decision rule is: $\hat{y} = 1$ if

$$(\boldsymbol{x} - \mu_1)^T \Sigma_1^{-1}(\boldsymbol{x} - \mu_1) < (\boldsymbol{x} - \mu_0)^T \Sigma_0^{-1}(\boldsymbol{x} - \mu_0)$$

With $\Sigma_1 = \Sigma_0 = \Sigma$ and naive Bayes (Σ is diagonal), this simplifies to checking if

$$\sum_j \frac{(\boldsymbol{x} - \mu_1)_j^2}{\sigma_j^2} < \sum_j \frac{(\boldsymbol{x} - \mu_0)_j^2}{\sigma_j^2}$$

Simplifying the above, we equivalently need

$$\boldsymbol{w}^T\boldsymbol{x} + b > 0, \quad \text{with} \quad \boldsymbol{w}_j = \frac{(\mu_1 - \mu_0)_j}{\sigma_j^2}, \quad b = 0.5\sum_j \frac{(\mu_1)_j^2 - (\mu_0)_j^2}{\sigma_j^2}$$

In fact we can also get an expression for $\boldsymbol{w}$ and b even if we just have $\Sigma_1 = \Sigma_0 = \Sigma$

$$\boldsymbol{w} = \Sigma^{-1}(\mu_1 - \mu_0), \quad b = 0.5(\mu_1^T \Sigma^{-1}\mu_1 - \mu_0^T \Sigma^{-1}\mu_0).$$

8.2 Support Vector Machines (SVMs): Motivation and main idea

Both the classifiers we talked about so far are linear classifiers as explained above. Consider logistic regression. The classification decision would be much more reliable if $\theta^T \boldsymbol{x}$ were either much larger or much smaller than zero.

The idea of SVM is this: we do not assume any data model here. Instead we try to look for the "separating hyperplane", equivalently, a vector θ so that the "margin" from the decision boundary is maximized for *all* training data points. Visual explanation in class or see cs229-notes-3.

8.2.1 Notation change

Instead of a single vector θ with the first entry used for the bias term, in case of SVMs, we use a weight vector $\boldsymbol{w}$ which is the same length as the data and a scalar b. Also, instead of labeling the two classes as 0 and 1, we label them is -1 and $+1$ because this simplifies some of the writing.

Margin: the distance of a data point from the separating hyperplane. Margin for the i-th training data point is computed as

$$y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b)$$

8.2.2 Using SVM for classification

Suppose first that the optimal choice of $\boldsymbol{w}, b$ is available. Then we classify as follows

$$\hat{y} = sign(\boldsymbol{w}^T \boldsymbol{x} + b)$$

If the term > 0, then the class is $+1$ else the class is -1.

8.2.3 Goal

The goal in case of SVMs is to find $\boldsymbol{w}, \boldsymbol{b}$ that maximize the worst-case margin defined by

$$\min_{i=1,2,\ldots,m} y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b)$$

By multiplying $\boldsymbol{w}, b$ by a scalar we could keep increasing the margin, but that will not improve classification. Thus, we need to impose the constraint that $\|\boldsymbol{w}\|_2 = 1$ or equivalently, replace $\boldsymbol{w}$ by $\boldsymbol{w}/\|\boldsymbol{w}\|$.

8.3 Simplifications to obtain a convex optimization problem **

Writing the above, we need to solve

$$\max_{\boldsymbol{w},b} \gamma \text{ s.t. } \quad \frac{y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b)}{\|\boldsymbol{w}\|} \geq \gamma, \; i = 1, 2, \ldots, m$$

This is not a convex optimization problem yet. So we try to simplify further. This is the same as

$$\max_{\boldsymbol{w},b} \gamma \text{ s.t. } \quad y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b) \geq \gamma \|\boldsymbol{w}\|, \; i = 1, 2, \ldots, m$$

Let $\tilde{\gamma} = \gamma \|\boldsymbol{w}\|$, then we have

$$\max_{\boldsymbol{w},b} \frac{\tilde{\gamma}}{\|\boldsymbol{w}\|} \text{ subject to } y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b) \geq \tilde{\gamma}, \; i = 1, 2, \ldots, m$$

This is still not convex. Another point to notice is this: we could fix the value of the margin γ to 1, and nothing will change. This gives

$$\max_{\boldsymbol{w},b} \frac{1}{\|\boldsymbol{w}\|_2} \text{ subject to } y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b) \geq 1, \; i = 1, 2, \ldots, m$$

This is not convex either but now a simple reformulation gives us a convex problem. Maximizing $1/\|\boldsymbol{w}\|_2$ is equivalent to minimizing $\|\boldsymbol{w}\|_2$ which is the same as minimizing $\|\boldsymbol{w}\|_2^2$. This gives

$$\min_{\boldsymbol{w},b} \|\boldsymbol{w}\|_2^2 \text{ subject to } y^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b) \geq 1,\ i = 1, 2, \ldots, m$$

This is now a convex optimization problem since the cost is a convex function and the inequality constraints are linear. In particular it is what is called a Quadratic Program or QP.

8.4 Final primal problem

$$\min_{\boldsymbol{w},b} \|\boldsymbol{w}\|_2^2 \text{ subject to } y^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b) \geq 1,\ i = 1, 2, \ldots, m$$

This is now a convex optimization problem since the cost is a convex function and the inequality constraints are linear. In particular it is what is called a Quadratic Program or QP.

8.5 Simplification using duality: needed to develop Kernel SVMs

By using *Lagrange duality*, it is possible to show that we can also compute the optimal $\boldsymbol{w}, b$ as follows.

1. Solve the following "dual problem": optimize over the Langrange multipliers α_i

$$\max_{\alpha} \left(\sum_{i=1}^{m} \alpha_i - 0.5 \sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i\alpha_j y^{(i)}y^{(j)}\langle \boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\rangle \right) \text{ s.t. } \sum_{i=1}^{m} \alpha_i y^{(i)} = 0,\ \alpha_i \geq 0,\ i = 1, 2, \ldots, m$$

   or equivalently solve the following minimization problem: note change of sign

$$\min_{\alpha} \left(-\sum_{i=1}^{m} \alpha_i + 0.5 \sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i\alpha_j y^{(i)}y^{(j)}\langle \boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\rangle \right) \text{ s.t. } \sum_{i=1}^{m} \alpha_i y^{(i)} = 0,\ \alpha_i \geq 0,\ i = 1, 2, \ldots, m$$

   Get $\hat{\alpha}$ as output

   (here $\langle \boldsymbol{x}_1, \boldsymbol{x}_2\rangle = \boldsymbol{x}_1^T\boldsymbol{x}_2$ is the inner product)

2. Obtain

$$\hat{\boldsymbol{w}} = \sum_{i=1}^{m} \hat{\alpha}_i y^{(i)}\boldsymbol{x}^{(i)} \text{ (this never needs to be computed, see notes below)}$$

$$\hat{b} = -0.5\left(\max_{i:y^{(i)}=-1} \hat{\boldsymbol{w}}^T\boldsymbol{x}^{(i)} + \min_{i:y^{(i)}=1} \hat{\boldsymbol{w}}^T\boldsymbol{x}^{(i)} \right)$$

3. Classification step: this also only uses inner products: for a query feature $\boldsymbol{x}$, compute

$$\hat{y} = sign(\hat{\boldsymbol{w}}^T\boldsymbol{x} + \hat{b}) = sign(\sum_i \alpha_i y^{(i)}\langle \boldsymbol{x}^{(i)}, \boldsymbol{x}\rangle + \hat{b})$$

   where

$$\hat{b} = -0.5\left(\max_{i:y^{(i)}=-1} \sum_{j=1}^{m} \hat{\alpha}_j y^{(j)}\langle \boldsymbol{x}^{(j)}, \boldsymbol{x}^{(i)}\rangle + \min_{i:y^{(i)}=1} \sum_{j=1}^{m} \hat{\alpha}_j y^{(j)}\langle \boldsymbol{x}^{(j)}, \boldsymbol{x}^{(i)}\rangle \right)$$

Notice: In the above dual program

- α is a vector of length m: for classification usually $m \ll n$ suffices. Thus the dual opt problem is computationally much cheaper to solve

- The dependence on feature vectors is only through inner products: this is true for training (for obtaining $\hat{\alpha}$)
- This is also true for testing: the weight vector $\hat{\boldsymbol{w}}$ actually never needs to be computed. We only need to compute the $\hat{\alpha}$ vector: we can then use inner products between the query $\boldsymbol{x}$ and the training data features and the α vector to compute the output $\hat{y}$ for a given $\boldsymbol{x}$
- The above implies we can replace the regular Euclidean inner product by any other inner product and our solution form will not change (in terms of coding, it would imply you need to code in a new function. Why would we do that? One reason:
  - If the original data is not linearly separable, we try to map it to a higher dimensional space and hope that it is linearly separable in that space.
  - How to do this? One example is use the original feature vector and pairwise products of the features, e.g. if $n = 2$, then use $\phi(\boldsymbol{x}) = [x_1, x_2, x_1^2, x_2^2, x_1 x_2]^\top$.
  - With a few constants changed, the inner product between $\phi(\boldsymbol{x}_a)$, $\phi(\boldsymbol{x}_b)$ for the above $\phi(.)$ mapping can be computed as
    $$K(\boldsymbol{x}_a, \boldsymbol{x}_b) :=< \phi(\boldsymbol{x}_a), \phi(\boldsymbol{x}_b) >= (\boldsymbol{x}_a^\top \boldsymbol{x}_b + c)^2$$
    The above is called the "kernel product"
  - Notice computing the inner product using $(\boldsymbol{x}_a^\top \boldsymbol{x}_b + c)^2$ has time cost of order n, while computing the inner product directly for $\phi(\boldsymbol{x}_a), \phi(\boldsymbol{x}_b)$ has cost order $n_{higher} = n^2$.
  - There are many other options for higher dimensional mappings and the corresponding Kernel products
  - To add: more examples; Theorem for what can be a valid Kernel product, final conclusion

The above is useful in two settings: (i) if $n \gg m$, i.e. the original data lies in a much higher dimensional space compared to the available number of data points (this often happens in classification problems), then the above dual is much less expensive to solve. (ii) Notice that in the above problem, all dependence on the feature vectors $\boldsymbol{x}^{(i)}$ is through the inner product $\langle \boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)} \rangle$. This means we could consider higher dimensional feature vectors, i.e., convert $\boldsymbol{x}$ to $\phi(\boldsymbol{x})$ by considering features of the form $\boldsymbol{x}_j^2$, $\boldsymbol{x}_j \boldsymbol{x}_k$.

8.6 Kernel SVM: more details**

For many such high dimensional feature mappings, $\phi(\boldsymbol{x})$ is very expensive to compute. But just computing the inner product, defined as the "kernel product"

$$K_\phi(\boldsymbol{x}_1, \boldsymbol{x}_2) := \langle \phi(\boldsymbol{x}_1), \phi(\boldsymbol{x}_2) \rangle$$

is much less expensive. An example is $\phi(x)$ obtained as all pairwise products of entries of $\boldsymbol{x}$. For this, computing $\phi(\boldsymbol{x}^{(i)})$ takes order n^2 time, thus computing the new feature vector for all i takes order $n^2 m$ time. But computing one kernel product can be done as

$$K_\phi(\boldsymbol{x}_1, \boldsymbol{x}_2) = (\boldsymbol{x}_1^T \boldsymbol{x}_2)^2$$

This only takes order n time. There are $m(m+1)/2$ total products to compute. Thus, the time needed is of order nm^2. Since typically, $m \ll n$, this is much quicker.

It is also possible to define kernel products for cases where the actual feature mapping is infinite dimensional. Gaussian kernel is an example. This is defined by

$$K_\phi(\boldsymbol{x}_1, \boldsymbol{x}_2) = \exp\left(\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{2\sigma^2} \right)$$

Kernel product: is basically some measure of similarity between two data points. Any useful measure of similarity can be used to define a "kernel" (kernel product), one may not even need to specify the underlying feature mapping.

A very large number of kernels can be defined. The purpose is for datasets which are not linearly separable in the original feature space, it is possible they are in a higher dimensional space.

8.6.1 Training and Classification using Kernel SVM

1. Training: Solve

$$\min_{\alpha}\left(-\sum_{i=1}^{m}\alpha_i + 0.5\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)}y^{(j)}K_{\phi}(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})\right) \text{ s.t. } \sum_{i=1}^{m}\alpha_i y^{(i)} = 0,\ \alpha_i \geq 0,\ i = 1,2,\dots,m$$

   Get $\hat{\alpha}$ as output.

   (here $\langle \boldsymbol{x}_1, \boldsymbol{x}_2\rangle = \boldsymbol{x}_1^T\boldsymbol{x}_2$ is the inner product)

2. Classify:

$$\hat{y} = sign(\sum_i \alpha_i y^{(i)} K_{\phi}(\boldsymbol{x}^{(i)},\boldsymbol{x}) + \hat{b})$$

   where

$$\hat{b} = -0.5\left(\max_{i:y^{(i)}=-1}\sum_{j=1}^{m}\hat{\alpha}_j y^{(j)}K_{\phi}(\boldsymbol{x}^{(j)},\boldsymbol{x}^{(i)}) + \min_{i:y^{(i)}=1}\sum_{j=1}^{m}\hat{\alpha}_j y^{(j)}K_{\phi}(\boldsymbol{x}^{(j)},\boldsymbol{x}^{(i)})\right)$$

8.6.2 Other ML problems

This "kernel trick" can be used for many other learning algorithms as well. Anytime all computation depends on inner products between the features, this can be used.

8.7 Soft margin SVMs

See page 19 of ML-cs229-noted-3

The SVM presented so far assumes data is linearly separable. But often data is not. Then we use the following

$$\min_{\boldsymbol{w},b}\|\boldsymbol{w}\|_2^2 + C_1\sum_{i=1}^{m}\zeta^{(i)} \quad \text{s.t.} \quad y^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b) \geq 1 - \zeta^{(i)},\ \zeta^{(i)} \geq 0,\ i = 1,2,\dots,m,$$

The above is imposing the following: find $\boldsymbol{w}, b$ so that some training data are misclassified but not too many.

Dual problem for the above

$$\max_{\alpha}\left(\sum_{i=1}^{m}\alpha_i - 0.5\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y^{(i)}y^{(j)}\langle x^{(i)},\boldsymbol{x}^{(j)}\rangle\right) \text{ s.t. } \sum_{i=1}^{m}\alpha_i y^{(i)} = 0,\ \alpha_i \geq 0,\ \alpha_i \leq C_1,\ i = 1,2,\dots,m$$

9 Deviation: Introduction to Langrange duality to understand how to derive the dual program **

To be updated later (to make this course 425/525).

See ML-cs229-notes3

To do also: add a section on basic optimization ideas - just enough to teach MLE derivations; and a section on MLE where we derive all the parameter estimates.

10 Least Squares (LS): all kinds

10.1 Regularized LS

The opt problem that needs to be solved to learn the parameter vector θ for Lin Reg is called the "Least Squares" problem. This involves solving

$$\min_{\theta} \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2$$

The above is called *Ordinary Least Squares* or *Ordinary Linear Regression.* This problem also occurs in various other domains. Many examples in communications and in signal processing. Thus it is useful to understand its various modifications.

We may claim that no house feature is way too important, and thus, no entry of θ should be too large. In this case we solve

$$\min_{\theta} \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2 + \lambda\|\theta\|^2$$

The above is also called *Ridge Regression* or *ell-2 regularization*

ell-1 regularized LS: this ensures the vector θ is sparse (has many zero entries), larger λ makes it sparser. Use if only a subset of all the features in the feature vector matter but do not know which ones.

$$\min_{\theta} \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2 + \lambda\|\theta\|_1$$

The above is also called *Lasso* or *ell-1 regularization.*

10.2 Use of prior knowledge and Recursive LS

In many settings, we may have prior knowledge that θ should be close to θ_0. For example, we may have θ_0 from houses sold a year ago and their features. In such cases, we solve

$$\min_{\theta} \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2 + \lambda\|\theta - \theta_0\|^2$$

Recursive LS: consider house price example again and suppose we want to update our model each week. Instead of starting from scratch, can we just "update" the model: save time/computation?

Turns out we can using recursive LS: suppose we update with each new house sold, then we get this

$$\begin{aligned}
\hat{\theta}_0 &= \theta_0, P_0 = \lambda \boldsymbol{I},\\
K_t &= P_{t-1}\boldsymbol{x}^{(i)}(\lambda + \boldsymbol{x}^{(i)\top}P_{t-1}\boldsymbol{x}^{(i)})^{-1},\\
P_t &= (\boldsymbol{I} - K_t\boldsymbol{x}^{(i)\top})P_{t-1},\\
\hat{\theta}_t &= \hat{\theta}_{t-1} + K_t(\boldsymbol{y}^{(i)} - \boldsymbol{x}^{(i)\top}\hat{\theta}_{t-1}),
\end{aligned}$$

Suppose we update after a set of m_t new houses sold (so $\boldsymbol{x}^{(i)}$ is an $n \times m_t$ matrix and $\boldsymbol{y}^{(i)}$ is a $m_t \times 1$ vector), then we get this

$$\begin{aligned}
\hat{\theta}_0 &= \theta_0, P_0 = \boldsymbol{I},\\
K_t &= P_{t-1}\boldsymbol{x}^{(i)}(\lambda\boldsymbol{I} + \boldsymbol{x}^{(i)\top}P_{t-1}\boldsymbol{x}^{(i)})^{-1},\\
P_t &= (\boldsymbol{I} - K_t\boldsymbol{x}^{(i)\top})P_{t-1},\\
\hat{\theta}_t &= \hat{\theta}_{t-1} + K_t(\boldsymbol{y}^{(i)} - \boldsymbol{x}^{(i)\top}\hat{\theta}_{t-1}),
\end{aligned}$$

11 Unsupervised Learning: PCA

In unsupervised learning, there is no *labeled* training data to learn parameters from. So no "output" $y^{(i)}$ is available for "input" $\boldsymbol{x}^{(i)}$. PCA is an unsupervised learning technique that is used for dimension reduction. Given data vectors in $\Re^n$, if they approximately lie in a lower dimensional subspace, how do we find that subspace?

11.1 What is PCA

As before we assume that we are given m data vectors (usually called feature vectors) $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \cdots, \boldsymbol{x}^{(m)}$ each in $\Re^n$, and these are stacked as the rows of a matrix

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

Given a reduced dimension r, the goal is find an r dimensional representation of the data vectors so that maximum information (variance) is retained.

11.2 Why PCA

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". So we try to reduce the dimension to $r < n$. This is helpful in the following ways:

- Memory and Speed: Less storage needed to save the data. Analysis of the reduced-dimension dataset, e.g., regression or classification, will be faster;
- Noise reduction: if features are noisy, i.e., if $\boldsymbol{X}$ is such that $(1/m)\boldsymbol{X}\boldsymbol{X}^\top \approx \Sigma + \sigma_e^2 \boldsymbol{I}$.
- Removing feature vector entries (or directions) with zero or near zero variability in the training data
  - In some other cases, there may be zero variance along some directions and when the data covariance matrix is computed, it ends up being rank deficient. Hence it cannot be inverted, but inversion is needed for example, for GDA based classification. This is actually the perfect application where PCA resolve the problem and may in fact help improve classification accuracy.
  - This is also useful for noise reduction when test data does not follow the same distribution as training data - and test data has noise even along these directions.
- Finally: PCA helps to get subspace coefficients which are uncorrelated. This is needed in some applications and is convenient in others. For example, for GDA things means that the covariance matrix of the new data vectors is diagonal.
  - For this, we need not reduce the dimension though, we can obtain uncorrelated variables in n dimensions too by using the full SVD.
- PCA useful for linear regression only if (i) either the original set of features is very noisy with noise being i.i.d. across all directions; or (ii) if the features set is highly correlated (the resulting feature covariance is approx low rank). Even so, in most cases using $r = (m-1)$ will often be the best solution. The real question is: is $r = m/2$ almost as good?

11.2.1 Use of PCA

PCA is a dimension reduction technique.

It is one way to do feature selection if the assumption that "features having the most variation are the most important ones" is true. In addition, of course PCA helps de-correlate the features.

A typical setting where this is useful is when the raw feature vector is all image pixels. There are too many pixels and pixels are highly correlated.

PCA is not very useful if features are first variance normalized and then PCA is applied.

11.3 Practical information: How to implement PCA

11.3.1 Reduced dimension, r, is specified

INPUT: $m \times n$ feature vectors' matrix $\boldsymbol{X}$, chosen rank r.

1. Compute $\hat{\mu} = \frac{1}{m} \sum_i \boldsymbol{x}^{(i)}$
2. For $i = 1, 2, \ldots, m$, let $\boldsymbol{z}^{(i)} = \boldsymbol{x}^{(i)} - \hat{\mu}$ and let

$$\boldsymbol{Z} = \begin{bmatrix} -(\boldsymbol{z}^{(1)})^T- \\ -(\boldsymbol{z}^{(2)})^T- \\ \vdots \\ -(\boldsymbol{z}^{(m)})^T- \end{bmatrix} \tag{3}$$

   OR: compute directly

$$\boldsymbol{Z} = \boldsymbol{X} - \mathbf{1}_m(\mathbf{1}_m^T \boldsymbol{X}/m)$$

   where $\mathbf{1}_m = [1\ 1\ \ldots\ 1]^T$ is an m-length vector of ones.

3. Compute the singular value decomposition (SVD) of $\boldsymbol{Z}$ and set $\boldsymbol{V}$ equal to the r right singular vectors with the largest singular values (called "top r" right singular vectors), i.e., compute

$$\boldsymbol{Z} \overset{SVD}{=} \boldsymbol{U}_{full} \boldsymbol{S}_{full} \boldsymbol{V}_{full}^T$$

   where $\boldsymbol{U}_{full}, \boldsymbol{V}_{full}$ are unitary matrices and $\boldsymbol{S}_{full}$ is an $m \times n$ diagonal matrix with non-negative entries (singular values) arranged in decreasing order of magnitude.

   Set $\boldsymbol{V} = \boldsymbol{V}_{full}(:, 1:r)$ in MATLAB notation (set $\boldsymbol{V}$ equal to first r columns of $\boldsymbol{V}_{full}$).

   Thus $\boldsymbol{V}$ is the $n \times r$ matrix whose columns span the computed *principal subspace.*

   - We can also obtain $\boldsymbol{V}$ as the eigenvectors with the r largest eigenvalues (called top r eigenvectors) of $\boldsymbol{Z}^T\boldsymbol{Z}$.

4. Project the original data $\boldsymbol{X}$ to $range(\boldsymbol{V})$: compute $\boldsymbol{b}^{(i)} = \boldsymbol{V}^T \boldsymbol{x}^{(i)}$ for each i or equivalently, compute

$$\boldsymbol{B} = \boldsymbol{X}\boldsymbol{V}$$

   Thus $\boldsymbol{B}$ is $m \times r$. These are the new feature vectors in the reduced dimensional space.

OUTPUT: $m \times r$ matrix $\boldsymbol{B}$: reduced dim feature vectors; $n \times r$ matrix $\boldsymbol{V}$: principal subspace.

- If the rank r approximation of $\boldsymbol{X}$ is needed, this is obtained as the $m \times n$ matrix

$$\boldsymbol{L} = \boldsymbol{B}\boldsymbol{V}^T = \boldsymbol{X}\boldsymbol{V}\boldsymbol{V}^T$$

11.3.2 Deciding r

So far we have assumed that the desired lower-dimension r is given. However, there is no one correct way of deciding r in practice. Two common approaches:

1. A common heuristic is the 99% or some other high-enough percent heuristic: retain all eigenvectors so the that variance in the reduced dimensional space is at least 99% of the total variance of the data. In other words, find the smallest value r so that

$$\sum_{i=1}^{r} \sigma_i^2 \geq 0.99 \sum_{i=1}^{\min(m,n)} \sigma_i^2.$$

   There is nothing "special" about 99%, we could also use another percentage.

2. An alternative approach is to pick the r that is best for the final application for which PCA is being used as a pre-processing step. We use simple or leave-one-out cross-validation to compute the value of r that minimizes the test-MSE.

11.3.3 Details for the second approach

This approach is most useful if the feature vectors are noisy. It is typically not useful if output is a noisy function of feature vec's; in that case the best r will often be equal or very close to n. Of course, in practice, for a real dataset, this part is not clear (it's clear for simulated data).

Consider any classification application. Given $\boldsymbol{X}$, $\boldsymbol{y}$ and $\boldsymbol{X}_{test}, \boldsymbol{y}_{test}$: obtain by splitting all available data into 10% testing and 90% training). OR BETTER: use leave-one-out cross-validation (most efficient but more difficult to explain here).

1. Compute $\boldsymbol{Z} = \boldsymbol{X} - (\boldsymbol{1}_m \boldsymbol{1}_m^T \boldsymbol{X}/m)$
2. Compute SVD of $\boldsymbol{Z}$
$$\boldsymbol{Z} \overset{SVD}{=} \boldsymbol{U}_{full} \boldsymbol{S}_{full} \boldsymbol{V}_{full}^T$$
where $\boldsymbol{U}_{full}, \boldsymbol{V}_{full}$ are unitary matrices and $\boldsymbol{S}_{full}$ is an $m \times n$ diagonal matrix with non-negative entries (singular values) arranged in decreasing order of magnitude.
3. Loop over r from 1 to n
   (a) Set $\boldsymbol{V} = \boldsymbol{V}_{full}(:, 1 : r)$ in MATLAB notation (set $\boldsymbol{V}$ equal to first r columns of $\boldsymbol{V}_{full}$).
   (b) Compute $\boldsymbol{B} = \boldsymbol{X}\boldsymbol{V}$
   (c) Use $\boldsymbol{B}$ where you would have used $\boldsymbol{X}$: if linear regression then use it to get $\hat{\theta} = (\tilde{\boldsymbol{B}}^T \tilde{\boldsymbol{B}})^{-1} \tilde{\boldsymbol{B}}^\top \boldsymbol{y}$, if logistic reg, then use it in the GD algorithm to find $\hat{\theta}$.
   (d) Compute test error
      i. Compute $\boldsymbol{B}_{test} = \boldsymbol{X}_{test} \boldsymbol{V}$
      ii. Use $\boldsymbol{B}_{test}$ where you would have used $\boldsymbol{X}_{test}$:
         - if linear regression, then obtain $\hat{\boldsymbol{y}}_{test} = \boldsymbol{B}_{test} \hat{\theta}$ and compute $NormalizedTestMSE(r) = \|\boldsymbol{y}_{test} - \hat{\boldsymbol{y}}_{test}\|_2^2 / \|\boldsymbol{y}_{test}\|_2^2$
         - if logistic regression, then again obtain $\hat{\boldsymbol{y}}_{test}$ as explained in the Logistic Regression section, followed by computing the misclassification percentage
4. Pick r for which $NormalizedTestMSE(r)$ is the smallest.

11.3.4 Sample Py Code

Three ways to get the SVD. The last one directly gives you the top $k = 2$ singular vectors. In case of the others, some post processing is needed to get the top r singular vectors

```
import numpy as np
from scipy import linalg
from scipy.sparse import linalg as slinalg

x = np.array([[1,1,0,0,0],[0,0,1,1,0],[1,1,1,1,1]],dtype=np.float64)

npsvd = np.linalg.svd(x)
spsvd = linalg.svd(x)
sptop = slinalg.svds(x,k=2)
```

11.4 Theory – computational - 1

PCA solves

$$\min_{\boldsymbol{L}} \|\boldsymbol{X} - \boldsymbol{L}\|_F$$

Set $\boldsymbol{L} = \boldsymbol{B}\boldsymbol{V}^\top$ and simplify, then this becomes

$$\min_{\boldsymbol{B},\boldsymbol{V}:\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}} \|\boldsymbol{X} - \boldsymbol{B}\boldsymbol{V}^\top\|_F^2 = \min_{\boldsymbol{B},\boldsymbol{V}:\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}} \sum_i \|\boldsymbol{x}^{(i)} - \boldsymbol{V}\boldsymbol{b}^{(i)}\|^2$$

Now we can solve for $\boldsymbol{b}^{(i)}$ in closed form: $\boldsymbol{b}^{(i)} = \boldsymbol{V}^\top\boldsymbol{x}^{(i)}$ and substitute it in. Thus we get

$$\min_{\boldsymbol{B},\boldsymbol{V}:\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}} \|\boldsymbol{X} - \boldsymbol{B}\boldsymbol{V}^\top\|_F^2 = \min_{\boldsymbol{V}:\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}} \sum_i \|(\boldsymbol{I} - \boldsymbol{V}\boldsymbol{V}^\top)\boldsymbol{x}^{(i)}\|^2$$

Since

$$\|(\boldsymbol{I} - \boldsymbol{V}\boldsymbol{V}^\top)\boldsymbol{x}\|^2 = \|\boldsymbol{x}\|^2 + \|\boldsymbol{V}^\top\boldsymbol{x}\|^2 - 2\|\boldsymbol{V}^\top\boldsymbol{x}\|^2 = \|\boldsymbol{x}\|^2 - \|\boldsymbol{V}^\top\boldsymbol{x}\|^2$$

the min simplifies to

$$\max_{\boldsymbol{V}:\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}} \sum_i \|\boldsymbol{V}^\top\boldsymbol{x}^{(i)}\|^2$$

Since

$$\sum_i \|\boldsymbol{V}^\top\boldsymbol{x}^{(i)}\|^2 = \sum_i (\boldsymbol{x}^{(i)\top}\boldsymbol{V}\boldsymbol{V}^\top\boldsymbol{x}^{(i)}) = trace(\sum_i \boldsymbol{x}^{(i)\top}\boldsymbol{V}\boldsymbol{V}^\top\boldsymbol{x}^{(i)}) = trace(\boldsymbol{V}^\top(\sum_i \boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)\top})\boldsymbol{V})$$

thus this further simplifies to

$$\max_{\boldsymbol{V}:\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}} trace(\boldsymbol{V}^\top(\sum_i \boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)\top})\boldsymbol{V})$$

Let EVD of $\hat{\Sigma} = (\sum_i \boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)\top})/m = \boldsymbol{V}\Lambda\boldsymbol{V}^\top$ with $\boldsymbol{U}$ orthonormal Suppose $r = 2$. This then simplifies to

$$\max_{\boldsymbol{v}_2\perp\boldsymbol{v}_1,:\|\boldsymbol{v}_2\|=1} [\max_{\boldsymbol{v}_1:\|\boldsymbol{v}_1\|=1} \boldsymbol{v}_1^\top(\sum_i \boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)\top})\boldsymbol{v}_1 + \boldsymbol{v}_2^\top(\sum_i \boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)\top})\boldsymbol{v}_2]$$

The first term is maximized by the top eigenvector, $\boldsymbol{v}_1$ of $\hat{\Sigma} = (\sum_i \boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)\top})/m$ and its maximum value equals the top eigenvalue, λ_1. Thus,

$$\max_{\boldsymbol{V}:\boldsymbol{V}^\top\boldsymbol{V}=\boldsymbol{I}} trace(\boldsymbol{V}^\top\hat{\Sigma}\boldsymbol{V}) = \lambda_1 + \max_{\boldsymbol{v}_2\perp\boldsymbol{v}_1,:\|\boldsymbol{v}_2\|=1} \boldsymbol{v}_2^\top\boldsymbol{U}\Lambda\boldsymbol{U}^\top\boldsymbol{v}_2$$

Thus, this is solved by $\boldsymbol{v}_2 =$ the second eigenvector of $\hat{\Sigma}$. Proceeding similarly

$$\hat{\boldsymbol{V}} = \boldsymbol{V}_{1:r}$$

solves the maximimization.

Eigenvectors of $\hat{\Sigma}$ are right singular vectors of $\boldsymbol{X}$.

Then setting $\hat{\boldsymbol{B}} = \boldsymbol{X}\hat{\boldsymbol{V}}^\top$ proves that

$$\hat{\boldsymbol{L}} = \boldsymbol{X}\boldsymbol{V}_{1:r}\boldsymbol{V}_{1:r}^\top = \boldsymbol{U}_{1:r}\boldsymbol{S}_{r\times r}\boldsymbol{V}_{1:r}^\top$$

(r-SVD of $\boldsymbol{X}$) solves

$$\min_{\boldsymbol{L}} \|\boldsymbol{X} - \boldsymbol{L}\|_F$$

11.5 Theory – computational 2

PCA also solves

$$\min_{\boldsymbol{L} \text{ rank r}} \|\boldsymbol{X} - \boldsymbol{L}\|_2^2 = \min_{\boldsymbol{L} \text{ rank r}} \lambda_{\max}((\boldsymbol{X} - \boldsymbol{L})^T(\boldsymbol{X} - \boldsymbol{L}))$$

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,

$$\|\boldsymbol{X} - \boldsymbol{L}\|_2 \geq \sigma_i(\boldsymbol{X}) - \sigma_i(\boldsymbol{L})$$

Since $\boldsymbol{L}$ is rank r, it has only r nonzero singular values. Thus $\sigma_{r+1}(\boldsymbol{L}) = 0$. And so

$$\|\boldsymbol{X} - \boldsymbol{L}\|_2 \geq \sigma_{r+1}(\boldsymbol{X}).$$

Since this is true for any rank r matrix $\boldsymbol{L}$, it is also true for the minimizer, i.e.

$$\min_{\boldsymbol{L}:\text{rank } r} \|\boldsymbol{X} - \boldsymbol{L}\|_2 \geq \sigma_{r+1}(\boldsymbol{X})$$

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

Set

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

(r-SVD of $\boldsymbol{X}$), then $\boldsymbol{X} - \hat{\boldsymbol{L}} = \sum_{i=r+1}^{\min(m,n)} \sigma_i u_i v_i^T$ and so

$$\|\boldsymbol{X} - \hat{\boldsymbol{L}}\|_2 = \sigma_{r+1}$$

here $\sigma_i = \sigma_i(\boldsymbol{X})$.

11.6 Statistical optimality

First assume everything is zero mean.

1. PCA finds the subspace $\boldsymbol{V}$ ($\boldsymbol{V}$ is a matrix with orthonormal columsn that define the subspace) and the projected random vector $\boldsymbol{b}$ so that the expected value of the squared 2-norm of the reconstruction error $\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b}$ is minimized. Thus, it solves

$$\min_{\boldsymbol{b} \in \Re^r, \boldsymbol{V} \in \Re^{n \times r}: \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}} \mathbb{E}[\|\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b}\|_2^2]$$

If we minimize over $\boldsymbol{b}$ first as a function of $\boldsymbol{V}$, then this is a standard Least Squares problem whose solution is

$$\boldsymbol{b} = (\boldsymbol{V}^T\boldsymbol{V})^{-1}\boldsymbol{V}^T\boldsymbol{x} = \boldsymbol{V}^T\boldsymbol{x} \quad \text{since } \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I} \text{ in this case}$$

Thus, we need to solve

$$\min_{\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}} \mathbb{E}[\|\boldsymbol{x} - \boldsymbol{V}\boldsymbol{V}^T\boldsymbol{x}\|_2^2]$$

Since $\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}$, thus $\|\boldsymbol{V}\boldsymbol{V}^T\boldsymbol{x}\|_2^2 = \|\boldsymbol{V}^T\boldsymbol{x}\|_2^2$ and so $\|\boldsymbol{x} - \boldsymbol{V}\boldsymbol{V}^T\boldsymbol{x}\|_2^2 = \|\boldsymbol{x}\|^2 - \|\boldsymbol{V}^T\boldsymbol{x}\|^2$. With this, the above is also equivalent to

$$\max_{\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}} \mathbb{E}[\|\boldsymbol{V}^T\boldsymbol{x}\|^2]$$

Using property of trace, this is further equivalent to

$$\max_{\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}} trace(\boldsymbol{V}^T\mathbb{E}[\boldsymbol{x}\boldsymbol{x}^T]\boldsymbol{V})$$

This is another commonly use definition for PCA: PCA finds the r directions of largest variance of the data. Here $\mathbb{E}[\boldsymbol{x}\boldsymbol{x}^T]$ is the covariance matrix.

2. PCA can also be understood as minimizing the worst-case (largest) expected reconstruction error in any direction: for direction $\boldsymbol{w}$, this is $|\boldsymbol{w}^T(\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b})|$

$$\min_{\boldsymbol{b},\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V}=\boldsymbol{I}} \max_{\boldsymbol{w}:\|\boldsymbol{w}\|_2=1} \mathbb{E}[(\boldsymbol{w}^T(\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b}))^2]$$

   Using the properties of trace, this is equivalent to

$$\min_{\boldsymbol{b},\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V}=\boldsymbol{I}} \max_{\boldsymbol{w}:\|\boldsymbol{w}\|_2=1} \boldsymbol{w}^T\mathbb{E}[(\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b})(\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b})^T]\boldsymbol{w} = \min_{\boldsymbol{b},\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V}=\boldsymbol{I}} \lambda_{\max}(\mathbb{E}[(\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b})(\boldsymbol{x} - \boldsymbol{V}\boldsymbol{b})^T])$$

   The last equality follows using the variational definition of the maximum eigenvalue.

In the nonzero mean case, we do either of the above for $(\boldsymbol{x} - \mu)$.

11.7 PCA de-correlates the data: what does it mean **

This claim depends on how the principal subspace is defined. In (statistical) theory, we are finding $\boldsymbol{V}$ and $\boldsymbol{b}$ that solves $\min_{\boldsymbol{b},\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V}=\boldsymbol{I}} \mathbb{E}[\|(\boldsymbol{x} - \mu) - \boldsymbol{V}\boldsymbol{b} + \boldsymbol{V}\boldsymbol{V}^T\mu)\|_2^2]$, i.e., it minimizes the expected value of the reconstruction error. Since the minimizer over $\boldsymbol{b}$ is $\boldsymbol{b} = \boldsymbol{V}^T\boldsymbol{x}$, we are actually finding $\boldsymbol{V}$ that solves $\min_{\boldsymbol{V}:\boldsymbol{V}^T\boldsymbol{V}=\boldsymbol{I}} \mathbb{E}[\|(\boldsymbol{x} - \mu) - \boldsymbol{V}\boldsymbol{V}^T(\boldsymbol{x} - \mu)\|_2^2]$ and then setting $\boldsymbol{b} = \boldsymbol{V}^T\boldsymbol{x}$. If we can find this $\boldsymbol{V}$, then the computed lower dimensional r.v. $\boldsymbol{b}$ satisfies

$$\mathbb{E}[(\boldsymbol{b} - \mathbb{E}[\boldsymbol{b}])(\boldsymbol{b} - \mathbb{E}[\boldsymbol{b}])^T] \text{ is diagonal}$$

i.e.

$$\mathbb{E}[(\boldsymbol{b} - \mathbb{E}[\boldsymbol{b}])_j(\boldsymbol{b} - \mathbb{E}[\boldsymbol{b}])_k] = 0$$

for $j \neq k$.

Proof: assume everything is zero mean for ease of writing. We have $\boldsymbol{b} = \boldsymbol{V}'\boldsymbol{x}$ *so* $\boldsymbol{b}_j = \boldsymbol{v}_j'\boldsymbol{x}$*, thus,* $\mathbb{E}[\boldsymbol{b}_j\boldsymbol{b}_k] = \boldsymbol{v}_j'\mathbb{E}[\boldsymbol{x}\boldsymbol{x}']\boldsymbol{v}_k$*. As explained in previous section,* $\boldsymbol{V}$ *is the matrix of top* r *eigenvectors of* $\mathbb{E}[\boldsymbol{x}\boldsymbol{x}']$*, i.e., that* $\mathbb{E}[\boldsymbol{x}\boldsymbol{x}'] = \boldsymbol{V}\Sigma V' + \boldsymbol{V}_\perp\Sigma_\perp\boldsymbol{V}_\perp'$*. Hence,* $\mathbb{E}[\boldsymbol{b}_j\boldsymbol{b}_k] = \boldsymbol{v}_j'(\boldsymbol{V}\Sigma\boldsymbol{V}' + \boldsymbol{V}_\perp\Sigma_\perp\boldsymbol{V}_\perp')\boldsymbol{v}_k = \boldsymbol{v}_j'(\boldsymbol{V}\Sigma V')\boldsymbol{v}_k = 0$.

In practice, we cannot find above but only its data-based (empirical) approximation. Hence in practice, we are always finding $\boldsymbol{V}$ as the top r right singular vectors of $\boldsymbol{Z} = \boldsymbol{X} - \mathbf{1}_m\mathbf{1}_m^T\boldsymbol{X}/m$ (recall: where $\mathbf{1}_m = [1\ 1\ \ldots\ 1]^T$ is an m-length vector of ones). With this choice of $\boldsymbol{V}$, "uncorrelated" means the following:

$$\sum_{i=1}^{m}(\boldsymbol{b}^{(i)})_j(\boldsymbol{b}^{(i)})_k = 0$$

for $j \neq k$. In other words, the columns of the matrix $\boldsymbol{B}$ are mutually orthogonal.

Proof: same basic idea as above.

12 Unsupervised Learning: Clustering

Use ML-cs229- notes on Clustering to see the figures.

12.1 Problem

Given unlabeled data/features $\boldsymbol{x}^{(i)}$, $i = 1, 2, \ldots, m$, the goal is to partition the dataset into "cohesive" clusters (all points in same cluster are "close" while those in different clusters are "far"). Suppose we want to partition into k clusters. Then the goal can also be stated as: for each i, find the class label $y^{(i)}$ (this can take values from $\{1, 2, \ldots, k\}$).

12.2 k-means clustering

The goal is to find the class labels $y^{(i)}$ for each data point and the cluster centers so that the following cost is minimized

$$J(\mu_j, j = 1, 2, \dots, k, \boldsymbol{y}) = \sum_{i=1}^{m} \|\boldsymbol{x}^{(i)} - \mu_{y^{(i)}}\|_2^2$$

The original k-means clustering algorithm provides an Alternating-Minimization (AltMin) algorithm to minimize the above cost.

1. Initialize cluster centers $\hat{\mu}_1, \hat{\mu}_2, \dots, \hat{\mu}_k$. Can do random init.
2. Repeat
   (a) For each $i = 1, 2, \dots, m$, find the class labels
   $$\hat{y}^{(i)} = \arg \min_{j=1,2,\dots,k} \|\boldsymbol{x}^{(i)} - \hat{\mu}_j\|_2^2.$$
   (b) Update cluster centers: for each $j = 1, 2, \dots, k$, compute
   $$\hat{\mu}_j = \frac{\sum_{i=1}^{m} \mathbb{1}(\hat{y}^{(i)} == j)\boldsymbol{x}^{(i)}}{\sum_{i=1}^{m} \mathbb{1}(\hat{y}^{(i)} == j)}$$
   (the above is a solution to $\arg\min_{\mu_j, j=1,2,\dots,k} \sum_{i=1}^{m} \|\boldsymbol{x}^{(i)} - \mu_{yhat^{(i)}}\|_2^2$)

   until cluster center estimates do not change much, i.e., until $\max_j(\|\hat{\mu}_j^{(t+1)} - \hat{\mu}_j^{(t)}\| / \|\hat{\mu}_j^{(t)}\|) < threshold$ where $threshold = 0.001$ or some small fraction.
3. IMPROVED VERSION: repeat above algorithm N times with different random init's each time. For each repeat, compute the cost function value $J(\hat{\mu}_j, j = 1, 2, \dots, k, \hat{y})$. Keep the output of the repeat with the smallest cost.

IMPROVEMENT 2: one can replace the regular Euclidean distance to the cluster center by any other distance that is more relevant to the application. As an example, if it is known that different features are likely to have significantly different variances, one could init with $\hat{\Sigma}_j = \boldsymbol{I}$, replace $\arg\min_{j=1,2,\dots,k} \|\boldsymbol{x}^{(i)} - \hat{\mu}_j\|_2$ in step 2a by

$$\arg \min_{j=1,2,\dots,k} (\boldsymbol{x}^{(i)} - \hat{\mu}_j)\hat{\Sigma}_j^{-1}(\boldsymbol{x}^{(i)} - \hat{\mu}_j)$$

and in step 2b, update

$$\hat{\Sigma}_j = \frac{\sum_{i=1}^{m} \mathbb{1}(\hat{y}^{(i)} == j)(\boldsymbol{x}^{(i)} - \hat{\mu}_j)(\boldsymbol{x}^{(i)} - \hat{\mu}_j)^T}{\sum_{i=1}^{m} \mathbb{1}(\hat{y}^{(i)} == j)}$$

12.3 Alternating Minimization (AltMin) algorithm

AltMin is another approach to solve an optimization problem. It is a better one to use that Gradient Descent when the variable to be minimized over, $\boldsymbol{x}$, can be split into two subsets of variables $\boldsymbol{x} = [\boldsymbol{x}_1, \boldsymbol{x}_2]$ such that minimizing over $\boldsymbol{x}_1$ keeping $\boldsymbol{x}_2$ fixed and vice versa is either closed form or otherwise easy to do.

AltMin proceeds as follows

1. Randomly initialize $\boldsymbol{x}_1$ to $\hat{\boldsymbol{x}}_1$.
2. Repeat
   (a) Minimize over $\boldsymbol{x}_2$ keeping $\boldsymbol{x}_1$ fixed at $\hat{\boldsymbol{x}}_1$, i.e., compute
   $$\hat{\boldsymbol{x}}_2 = \arg\min_{\boldsymbol{x}_2} J(\hat{\boldsymbol{x}}_1, \boldsymbol{x}_2)$$

(b) Minimize over $\boldsymbol{x}_1$ keeping $\boldsymbol{x}_2$ fixed at $\hat{\boldsymbol{x}}_2$, i.e., compute

$$\hat{\boldsymbol{x}}_1 = \arg\min_{\boldsymbol{x}_1} J(\boldsymbol{x}_1, \hat{\boldsymbol{x}}_2)$$

until "convergence" i.e. estimates of $\hat{\boldsymbol{x}}_1$ do not change much from previous to current iteration.

Like Grad Desc, AltMin also only converges to the local minimum of the cost function. Thus to make it work better one can run it N times with different random init's and pick the solutions that result in the smallest cost function value.

12.4 Probabilistic Model (Generative Model) for Clustering: Gaussian Mixture Model (GMM)

The Gaussian Mixture Model or GMM is a common way to specify a clustering problem. $\boldsymbol{x}$ follows the GMM with k components means that

$$p(\boldsymbol{x};\theta) = \sum_{j=1}^{k} \mathcal{N}(\boldsymbol{x};\mu_j,\Sigma_j)\phi_j$$

where ϕ_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.

GMM: means that $\boldsymbol{x}$ is generated from class j with probability ϕ_j, and given that it is generated from class j, it follows a Gaussian distribution with mean μ_j and covariance Σ_j.

The model assumed by Gaussian discriminant analysis (GDA) covered earlier and in HW 2 is also GMM. See Sec 5.2. Except there we had labeled data (for each training data point, the class label was available), so it was easy to "learn" the model parameters. As a result, the learning of ϕ_j's was decoupled from the learning of the mean and covariances. As a result we in fact got closed form expressions for the MLE.

Here we do not have class labels and thus the learning problem is more difficult. We need to use Gradient Descent or some other iterative approach.

12.5 EM algorithm for MLE for Gaussian Mixture Model **

A popular approach for MLE for the GMM is the EM algorithm.

EM algorithm is another approach (besides AltMin and Grad Descent) to solve a Maximum Likelihood estimation (MLE) problem. Like AltMin it is useful for certain types of problems in which, by using some tricks, part of the problem can be made non-iterative (closed form) or can otherwise be simplified.

Consider Maximum Likelihood Estimation over i.i.d. data samples $x^{(i)}$ coming from a distribution $p(x;\theta)$. The goal is to maximum 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 ϕ_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,

$$\ell(\theta) = \sum_{i=1}^{m} \log \sum_{j=1}^{k} \phi_j \mathcal{N}(x;\mu_j,\Sigma_j) \ \text{ s.t. } \ \sum_j \phi_j = 1$$

and the parameters θ are

$$\theta = \{\mu_j, \Sigma_j, \phi_j\}, j = 1, 2, \ldots k$$

with μ_j being $n \times 1$, ϕ_j is a scalar and Σ_j is $n \times n$. Recall $\mathcal{N}(\boldsymbol{x}; \mu, \Sigma) := C \exp(-(\boldsymbol{x} - \mu)^T \Sigma^{-1} (\boldsymbol{x} - \mu) + \log(det(Sigma))$ where C is a constant (contains the $1/\sqrt{2\pi}$ etc terms).

$\ell(\theta)$ is a pretty messy expression to even compute the gradient of: notice it involves $\log \sum_j \phi_j \exp(....)$. And Grad Descent will not work well. Compare this with the expression we had for GDA: since y^i's were known, there was no log of sums of weighted exponentials. We instead just had sum of log of exponentials which simplified easily.

For the above problem, the following is an easier approach – somehow try to approximate the "complete data likelihood". Since class labels are not available, we call this "incomplete data". The idea proceeds as follows.

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 $q_i(j) \geq 0$ and $\sum_{j=1}^k q_i(j) = 1$. So one can think of them as some probability distribution over the class labels.

$$\begin{aligned}
\ell(\theta) &= \sum_{i=1}^m \log \sum_{j=1}^k \phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j) \\
&= \sum_{i=1}^m \log \sum_{j=1}^k \phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j) \frac{q_i(j)}{q_i(j)} \\
&= \sum_{i=1}^m \log \sum_{j=1}^k q_i(j) \frac{\phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}{q_i(j)} \\
&\geq \sum_{i=1}^m \sum_{j=1}^k q_i(j) \log \frac{\phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}{q_i(j)}
\end{aligned}$$

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.

Now we will use AltMin to MAXIMIZE the above LOWER BOUND ON $\ell(\theta)$.

By the above approach, we have increased the number of unknowns - now the unknowns include $q_i(j)$'s and θ. BUT, the maximization over θ given $q_i(j)$'s is easy (closed form, as we will see below); and the maximization over $q_i(j)$'s given θ fixed is also easy (the lower bound is maximized when the inequality holds with equality, i.e., for $q_i(j) = c\phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)$ with c specified above).

12.6 Final EM algorithm for Gaussian Mixture Models **

Thus, AltMin over $q_i(j)$'s and θ is our FINAL ALGORITHM. It is called "EM algorithm for Gaussian Mixture Models". This is summarized next.

- Initialize the values of θ. Recall $\theta = \{\phi_j, \mu_j, \Sigma_j\}, j = 1, 2, ..k$. Can be random or something else.
- Iterate the following two steps until "convergence" (some reasonable stopping criterion holds):
  - E-step (maximize lower bound on $\ell(\theta)$ over $q_i(j)$'s, holding θ fixed): this is obtained by

$$q_i(j) = \frac{\phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}{\sum_{j=1}^k \phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}, \ j = 1, 2, \ldots, k, i = 1, 2, \ldots, m.$$

  - M-step (maximize lower bound on $\ell(\theta)$ over θ 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

$$\max_\theta \ell_2(\theta) := \sum_{j=1}^{k}\sum_{i=1}^{m} q_i(j)[\log\phi_j + \log\mathcal{N}(x^{(i)};\mu_j,\Sigma_j)] \;\; \text{s.t.} \;\; \sum_j \phi_j = 1$$

    Notice we have converted log of weighted sums of exponentials to "weighted sum of log of exponentials", so the form of above expression is similar to that for GDA.

    This is easy, it is almost like GDA because it can be separated out over subsets of variables:

    * Solve for ϕ_j's

$$\max_{\phi_1,\phi_2,\ldots,\phi_k} \sum_{j=1}^{k} q_i(j)\log\phi_j \;\; \text{s.t.} \sum_j \phi_j = 1$$

      This is solved by $\hat{\phi}_j = \frac{1}{m}\sum_{i=1}^{m} q_i(j)$. This [2]
    * 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 $c = \frac{1}{\sum_{i=1}^{m} q_i(j)}$.

- Repeat with N different initializations. Pick the one for which we get the largest value of $\ell(\theta)$

The "E step" is the "Expectation step". Reason it is called this is the following. We have data that is incomplete. We do not know the values of the misisng entries, and so we do not know the complete data likelihood (in this case this corresponds to the likelihood func for GDA). But given estimates of the paramters from the previous iteration, we can compute what is called the posterior expectation of the complete data likelihood.

- The $q_i(j)$'s are called the posterior probabilities of $y^{(i)} = j$ given observed data, and
- $\ell_2(\theta)$ is called the posterior Expectation of the complete data likelihood.

Thus "E step" computes this posterior expectation given previous estimate of parameters, while "M step" maximizes this posterior expectation to find a new value of the parameters. Hence the same EM algorithm.

[2]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 $1/m$.

12.7 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

$$\begin{aligned}
\ell(\theta) &= \sum_{i=1}^{m} \log \sum_{j=1}^{k} p(x^{(i)}, y^{(i)} = j; \theta) \quad \text{s.t.} \sum_{j=1}^{k} q_i(y^{(i)} = j) = 1 \\
&= \sum_{i=1}^{m} \log \sum_{j=1}^{k} p(x^{(i)}, y^{(i)} = j; \theta) \frac{q_i(y^{(i)} = j)}{q_i(y^{(i)} = j)} \quad \text{s.t.} \sum_{j=1}^{k} q_i(y^{(i)} = j) = 1 \\
&= \sum_{i=1}^{m} \log \sum_{j=1}^{k} q_i(y^{(i)} = j) \frac{p(x^{(i)}, y^{(i)} = j; \theta)}{q_i(y^{(i)} = j)} \quad \text{s.t.} \sum_{j=1}^{k} q_i(y^{(i)} = j) = 1 \\
&\geq \sum_{i=1}^{m} \sum_{j=1}^{k} q_i(y^{(i)} = j) \log \frac{p(x^{(i)}, y^{(i)} = j; \theta)}{q_i(y^{(i)} = j)} \quad \text{s.t.} \sum_{j=1}^{k} q_i(y^{(i)} = j) = 1
\end{aligned}$$

We now use Alternating-Maximization to maximize the above lower bound over $q_i(y^{(i)} = j)$'s keeping θ 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, $\sum_j g_j \log c = \log c = \log \sum_j g_j c$. Thus the lower bound is maximized when

$$q_i(y^{(i)} = j) = c p(x^{(i)}, y^{(i)} = j; \theta)$$

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 θ 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

$$\begin{aligned}
&\arg\max_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{k} q_i(y^{(i)} = j) \log \frac{p(x^{(i)}, y^{(i)} = j; \theta)}{q_i(y^{(i)} = j)} \\
&= \arg\max_{\theta} \sum_{i=1}^{m} \sum_{j=1}^{k} q_i(y^{(i)} = j) \log p(x^{(i)}, y^{(i)} = j; \theta) \\
&= \arg\max_{\theta} \sum_{j=1}^{k} \sum_{i=1}^{m} q_i(y^{(i)} = j) \log p(x^{(i)}, y^{(i)} = j; \theta)
\end{aligned}$$

  This is often easier if the parameters θ 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.

13 Deep Learning / Deep Neural Networks: basic idea and training

13.1 Introduction

From Wikipedia `https://en.wikipedia.org/wiki/Deep_learning`

- An older definition: Deep learning (DL) is a class of ML algorithms that uses multiple layers to progressively extract higher-level features from the raw input.
  - For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human, such as digits, letters, or faces.
- Modern definition: Deep learning (DL) is the subset of ML methods based on artificial neural networks with representation learning.
  - The goal is to "automate" human learning processes that map a source (e.g., an image of dog) to a learned object label (dog) by attempting to model what human brain neurons do.
  - The adjective "deep" refers to the use of multiple layers in the network.
- Most common DL methods involve supervised learning. Some newer approaches are also semi-supervised or unsupervised. Pros and Cons:
  - Supervised DL: training is very compute intensive and memory intensive. Query/test data processing is quick (linear in layer depth and width)
  - Unsupervised DL: no training; but then query/test data processing is extremely slow. Also needs a lot of memory (even more than supervised case).
- Types of architectures:
  - Feed-forward: multilayer perceptron (MLP) also called fully-connected net; Convolutional neural networks (CNN); more
  - Feed-back: Recurrent NN
- History taken from Wikipedia `https://en.wikipedia.org/wiki/Deep_learning`
  - The first deep learning multilayer perceptron (MLP) trained by stochastic gradient descent (Robbins, Munro 1951) was published in 1967 by Shun'ichi Amari.
  - In 1970, Seppo Linnainmaa published the basic idea of the algorithm that is now called "backpropoagation". Basic idea is to use chain rule of differentiation to compute the gradient of the cost function (energy) w.r.t. the weights of all layers.
    - Details: In 1970, Seppo Linnainmaa published the reverse mode of automatic differentiation of discrete connected networks of nested differentiable functions. This became known as backpropagation. It is an efficient application of the chain rule derived by Gottfried Wilhelm Leibniz in 1673 to networks of differentiable nodes. More details: The terminology "back-propagating errors" was actually introduced in 1962 by Rosenblatt,[34][31] but he did not know how to implement this, although Henry J. Kelley had a continuous precursor of backpropagation[46] already in 1960 in the context of control theory.[31] In 1982, Paul Werbos applied backpropagation to MLPs in the way that has become standard.[47][48][31] In 1985, David E. Rumelhart et al. published an experimental analysis of the technique.[49]
  - Convolutional neural networks (CNNs) with convolutional layers and downsampling layers began with the Neocognitron introduced by Kunihiko Fukushima in 1980.[50] In 1969, he also introduced the ReLU (rectified linear unit) activation function.
  - The term Deep Learning was introduced to the ML community by Rina Dechter in 1986

- Deep learning architectures for convolutional neural networks (CNNs) with convolutional layers and downsampling layers began with the Neocognitron introduced by Kunihiko Fukushima in 1980.[50] In 1969, he also introduced the ReLU (rectified linear unit) activation function.[26][31]

13.2 Multi-layer Perceptron (MLP) or Fully Connected Feed-forward Network

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 o^{k-1} as input. It then computes

$$z_j^k = \sum_{i=1}^{r_k} w_{ij}^k o_i^{k-1}$$

and then outputs

$$o_j^k = g(z_j^k)$$

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

Vectorizing the above, it can be expressed as

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

or equivalently,

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

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

The first layer takes the input $\boldsymbol{x}$ as input thus

$$\boldsymbol{z}^1 = \boldsymbol{x}$$

and computes $g_{vec}(\boldsymbol{z})$. Suppose the NN has 10 layers. The final (output) layer has only one neuron which outputs

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

Thus, the NN is

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

where $\boldsymbol{W}^{10}$ is a row vector (instead of a matrix).

Energy (cost function) for training is

$$E(\boldsymbol{W}^1, \boldsymbol{W}^2, \dots \boldsymbol{W}^{10}) := \sum_{i=1}^{m} (\boldsymbol{y}^{(i)} - \hat{y}^{(i)})^2 = \sum_{i=1}^{m} (\boldsymbol{y}^{(i)} - g(\boldsymbol{W}^{10} g_{vec}(\boldsymbol{W}^9 g_{vec}(\boldsymbol{W}^8 \dots g_{vec}(\boldsymbol{x}^{(i)})))))^2$$

13.3 Training an MLP: Back-propagation **

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

$$\frac{\partial E}{\partial w_{ij}^k}$$

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 $\boldsymbol{x}$, compute the outputs of all the layers and $\hat{y}$.

- Compute the following intermediate quantity:

$$\delta_i^k := \frac{\partial E}{\partial z_i^k}$$

  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): $\boldsymbol{\delta}^k = g'(\boldsymbol{z}^k). * (\boldsymbol{W}^{k+1}\boldsymbol{\delta}^{k+1})$

- Compute

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k \cdot o_i^{k-1}$$

  this can be vectorized too.

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

$$\frac{1}{m} \sum_{i=1}^{m} \frac{\partial E}{\partial w_{ij}^k}(\hat{y}(\boldsymbol{x}^{(i)}), y^{(i)})$$

13.4 Understanding CNNs at a top level with visuals

See `https://towardsdatascience.com/beginners-guide-to-understanding-convolutional-neural-networks-a`

See Dr. Zhengdao Wang's CNN notes (cnn-notes-zhengdao) `https://www.dropbox.com/s/4avs5jg3r5qctqe/cnn_notes_zhengdao.pdf?dl=0` for visuals and a quick review of LTI, 1D and 2D conv.

13.5 Aside: understanding convolution

1D convolution:

2D convolution:

types of filters: low-pass, high-pass, band-pass,

low-pass filter, e.g., averaging filter: reduces noise but also blurs edges

high-pass filter, e.g., edge detector: nonzero response only at edges, zero response in parts of image that are constant (piecewise constant image)

high-pass in 2D : edges along different directions, can also be low-pass in one direction and high-pass in orthogonal direction

band-pass filters: created by applying high-pass filter on low-pass filter.

connection to wavelets.

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

In case of MLP, each layer is fully-connected (FC), this means that all (or most) weights of the weight matrix are nonzero.

In case of CNN, this is not true. CNN consists of

- a convolutional (CN) layer followed by
- a pooling layer (pool),
- this sequence repeated a few times,
- finally 2-3 fully-connected (FC) layers.

Why each layer

- CN layer: extract useful features from images; most good features are local. Also much fewer parameters (due to paramater sharing – space-invariant filter for entire image)
- Pool layer: down-sample the image after the filtering to get a new smaller sized "image" (when you pass an image through any band-pass filter, e.g.m low pass or high pass filter, the amount of information reduces, consequently next layer should be smaller sized).

  How to downsample? Down-sample by 2: "low pass filter" the image followed by keeping every other pixel. For "low pass filter": either take average of neighbors or take max of neighbors. Usually max-pooling is popular. Averaging is linear space-invariant while Max is nonlinear and space-invariant.
- FC layer: CN and pooling layers help extract the features. FC layers take these features and provides a final classification output.

13.6.1 Details of why CN + pooling instead of FC

- Local connectivity : useful when inputs are images
- Much fewer parameters (parameter sharing): $\ell_h^2 \cdot K'$, even with $\ell_h = 11, K' = 96$ (first layer of Alex Net), number of parameters is around 1000
  but if we vectorize even a 100×100 image and the next layer has more neurons than input layer (typically this is used), then there are more than 10000 parameters in an FC
- Heavily used in computer vision and image analyis applications.

13.7 Different types of layer

Let $\boldsymbol{x}$ be the input to the layer and $\boldsymbol{z}$ the output.

13.7.1 Fully connected (FC) layer:

$\boldsymbol{x}, \boldsymbol{z}$ are represented as vectors

- $\boldsymbol{x}$ is $n \times 1$, $\boldsymbol{z}$ is $n' \times 1$, n' can be $>, =, < n$
- parameters: $\boldsymbol{W}$
- number of parameters: $n \cdot n'$
- apply nonlinearity, then weights' matrix

$$\boldsymbol{z}_{out} = \boldsymbol{W} g_{vec}(\boldsymbol{z}_{in})$$

13.7.2 Convolutional (CN) layer:

$\boldsymbol{x}, \boldsymbol{z}$ are represented as 3D tensors

- $\boldsymbol{x}$ is $n_1 \times n_2 \times K$, $\boldsymbol{z}$ is $n_1' \times n_2' \times K'$ with $n_1' \approx n_1,\ n_2' \approx n_2$
- parameters: $h_k(u, v)$, $k = 1, 2, \ldots, K'$, $u = 0, 1, \ldots, \ell_h$, $v = 0, 1, \ldots, \ell_h$
- number of parameters: $\ell_h^2 \cdot K'$
- apply nonlinearity, convolution over the first 2 dimensions, sum over the third

$$\boldsymbol{z}(a, b, k) = \sum_w \left(\sum_u \sum_v h_k(u, v) g_{vec}(\ \boldsymbol{x}(a+u, b+v, w)\ ) \right)$$

- Details:
  - dealing with edges: zero pad, or reduce dimension of output
  - stride: above is stride = 1 (moving filter window by 1 pixel at a time) – default option
  - stride = 2, move filter window by 2 pixels
  - stride = ℓ_h: non-overlapping window

13.7.3 Pooling layer:

$\boldsymbol{x}, \boldsymbol{z}$ are represented as 3D tensors

- $\boldsymbol{x}, \boldsymbol{z}$ are represented as 3D tensors, $\boldsymbol{x}$ is $n_1 \times n_2 \times K$, $\boldsymbol{z}$ is $n_1' \times n_2' \times K$
- if using a $\ell \times \ell$ non-overlapping window for pooling, then $n_1' = n_1/\ell, n_2' = n_2/\ell$
- parameters: none
- number of parameters: zero
- applied on each $k = 1, 2, \ldots K$ "image" or "activation map" separately.
- helps reduce the output dimension (first 2 dimensions)
- max pooling over a $\ell \times \ell$ size window: keep the largest magnitude pixel value and throw the rest
- average pooling:
- stride = ℓ: non-overlapping window – default option

13.8 Other DL concepts

- Recurrent Neural Network (RNN)
- Generative Adversarial Networks GANs – used for DeepFake
- Auto-Encoder – use for denoising. Input and output is an image
- Unrolling
- Transformer

13.9 Recurrent Neural Network (RNN): basic idea

To Do

13.10 Different NN architectures and when to use each

see the other handout NeuralNets-intro.pdf

14 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).

15 Bias-Variance Tradeoff

15.1 What is it?

Consider a generative model: suppose that $y, \boldsymbol{x}$ satisfy

$$y = f(\boldsymbol{x}) + e, \ \mathbb{E}[e] = 0, \mathbb{E}[e^2] = \sigma^2$$

with e being zero mean "modeling error"/"noise" that is independent of $\boldsymbol{x}$, it is also independent for each data point y_i.

We do not know $f(.)$.

We try to "model" it as $\hat{y} = \hat{f}(x) = h_{\hat{\theta}}(\boldsymbol{x})$, e.g., in linear regression, $\hat{f}(x) = h_{\hat{\theta}}(\boldsymbol{x}) = \hat{\theta}^T \boldsymbol{x}$ with $\hat{\theta}$ estimated by Maximum Likelihood estimation (MLE) as described earlier using training data

$$(y^i, \boldsymbol{x}^i), i = 1, 2, \ldots, m$$

In logistic regression, $\hat{f}(x) = h_{\hat{\theta}}(\boldsymbol{x}) = g(\hat{\theta}^T \boldsymbol{x})$ 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 $\boldsymbol{x}$, what is $\mathbb{E}[(y - \hat{f}(\boldsymbol{x}))^2]$ and what can we do to improve it?

We define Test-MSE as

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

over test data, i.e., $\mathbb{E}[.] \equiv \mathbb{E}_{\text{test data}}[.]$. Usually $\boldsymbol{x}$ is treated as a constant, so then the expected value is over the distribution of the noise e.

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

$$\begin{aligned}\text{Test-MSE} &= \mathbb{E}[e^2] + \mathbb{E}[(f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}))^2] \\ &= \mathbb{E}[e^2] + (\mathbb{E}[f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})])^2 + Variance[f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})] \\ &= \sigma^2 + Bias^2 + Variance\end{aligned}$$

where $Variance(Z) := E[(Z - E[Z])^2]$. The first term, σ^2, depends on how noisy the data is. The second two terms depend on the "assumed model" and how well its parameters are estimated.

15.2 Bias-Variance Tradeoff for Linear Regression **

THERE IS A MISTAKE HERE: NEED TO FIX THESE NOTES to all for a constant "mean" term in linear regression.

Suppose that y truly satisfies the following model

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

where x_{full} is the full n_{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 $\boldsymbol{x}$; and refer to the dropped part of $\boldsymbol{x}_{\text{full}}$ as $\boldsymbol{x}_{\text{drop}}$. This is a n_{drop} length vector. Thus $n_{\text{full}} = n + n_{\text{drop}}$.

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

Thus, y can be rewritten as

$$y = \theta^T x + \underbrace{\theta_{\text{drop}}^T \boldsymbol{x}_{\text{drop}}}_{\mu} + e.$$

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

We use MLE under this model to get the MLE estimate

$$\hat{\theta} := (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}, \text{ with } \boldsymbol{y} := \boldsymbol{X}\theta + \boldsymbol{X}_{\text{drop}}\theta_{\text{drop}} + \boldsymbol{e}$$

where $\boldsymbol{X}_{\text{drop}} := [\boldsymbol{x}_{\text{drop}}^{(1)}; \boldsymbol{x}_{\text{drop}}^{(2)}; \dots, \boldsymbol{x}_{\text{drop}}^{(m)}]^T$ is an $m \times n_{\text{drop}}$ matrix with the dropped parts of each training data vector as its rows. Recall that $\boldsymbol{e}$ (training data noise vector) is independent of $\boldsymbol{X}$ (training data).

Thus, for a query $\boldsymbol{x}_{\text{full}}^{\text{tst}}$, with extracted features $\boldsymbol{x}^{\text{tst}}$, we predict

$$\hat{y}^{\text{tst}} = \boldsymbol{x}^{\text{tst}T} \hat{\theta}$$

The true output, y^{tst}, for the query satisfies

$$y^{\text{tst}} = \theta^T \boldsymbol{x}^{\text{tst}} + \theta_{\text{drop}}^T \boldsymbol{x}_{\text{drop}}^{\text{tst}} + e^{\text{tst}}$$

Observe that $\hat{\theta}$ satisfies

$$\hat{\theta} := (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T (\boldsymbol{X}\theta + \boldsymbol{X}_{\text{drop}}\theta_{\text{drop}} + \boldsymbol{e}) = \theta + (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{X}_{\text{drop}}\theta_{\text{drop}} + (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{e}$$

So,

$$\begin{aligned}\hat{y}^{\text{tst}} - \boldsymbol{y}^{\text{tst}} &= \boldsymbol{x}^{\text{tst}T} \hat{\theta} - \boldsymbol{x}^{\text{tst}T} \theta - {\boldsymbol{x}_{\text{drop}}^{\text{tst}}}^T \theta_{\text{drop}} - e^{\text{tst}} \\ &= \underbrace{\boldsymbol{x}^{\text{tst}T} \left((\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{X}_{\text{drop}}\theta_{\text{drop}} + (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{e} \right) - {\boldsymbol{x}_{\text{drop}}^{\text{tst}}}^T \theta_{\text{drop}}}_{Z} - e^{\text{tst}} \\ &= Z - e_{\text{tst}}\end{aligned}$$

Recall that $\mathbb{E}[.]$ is expected value over test-data noise e^{tst}. As before (since e^{tst} independent of everything in Z),

$$\begin{aligned} TestMSE := \mathbb{E}[(\hat{y}^{\text{tst}} - \boldsymbol{y}^{\text{tst}})^2] &= \mathbb{E}[(e^{\text{tst}})^2] + \mathbb{E}[Z^2] \\ &= \mathbb{E}[(e^{\text{tst}})^2] + Bias(Z)^2 + Variance(Z) \end{aligned}$$

with

$$\begin{aligned} Bias(Z) &:= \mathbb{E}[\boldsymbol{x}^{\text{tst}T}\left((\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{X}_{\text{drop}}\theta_{\text{drop}} + (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{e}\right) - \boldsymbol{x}^{\text{tst}}_{\text{drop}}{}^T\theta_{\text{drop}}] \\ &= \boldsymbol{x}^{\text{tst}T}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{X}_{\text{drop}}\theta_{\text{drop}} - \boldsymbol{x}^{\text{tst}}_{\text{drop}}{}^T\theta_{\text{drop}} \\ &= [\boldsymbol{x}^{\text{tst}T}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{X}_{\text{drop}} - \boldsymbol{x}^{\text{tst}}_{\text{drop}}{}^T]\theta_{\text{drop}} \end{aligned}$$

The second row follows because $\boldsymbol{e}$ is independent of $\boldsymbol{X}, \boldsymbol{X}_{\text{drop}}$ (training data noise is independent of training data features) and $\boldsymbol{e}$ is independent of $\boldsymbol{x}^{\text{tst}}$ and $\boldsymbol{e}$ is zero mean.

Since the training data noise vector $\boldsymbol{e}$ is zero mean and i.i.d., we have $\mathbb{E}[\boldsymbol{e}\boldsymbol{e}^T] = \sigma^2 I$. Using this,

$$\begin{aligned} Variance(Z) &:= \mathbb{E}[(Z - Bias(Z))^2] \\ &= \mathbb{E}[(\boldsymbol{x}^{\text{tst}T}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{e})^2] \\ &= \mathbb{E}[(\boldsymbol{x}^{\text{tst}T}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{e})(\boldsymbol{e}^T\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{x}^{\text{tst}})] \\ &= \boldsymbol{x}^{\text{tst}T}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\mathbb{E}[\boldsymbol{e}\boldsymbol{e}^T]\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{x}^{\text{tst}} \\ &= \boldsymbol{x}^{\text{tst}T}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{x}^{\text{tst}}\sigma^2 \end{aligned}$$

This is upper bounded by

$$Variance(Z) \leq \frac{\|\boldsymbol{x}^{\text{tst}}\|^2}{\lambda_{\min}(\boldsymbol{X}^T\boldsymbol{X})}\sigma^2$$

1. Variance: Thus the variance is smaller if the minimum eigenvalue of $(\boldsymbol{X}^T\boldsymbol{X})$ is larger.
   - Notice that $\boldsymbol{X}$ is an $m \times n$ matrix. Consider $\boldsymbol{X}^T\boldsymbol{X}$ which is $n \times n$. If $m < n$, its minimum eigenvalue is zero. For $m > n$, its minimum eigenvalue is proportional to m/n. The reason is, for a fixed m, by the interlacing theorem for eigenvalues, as you increase n, the smallest eigenvalue can only decrease [3].

   Thus for a fixed available amount of training data, m, the Variance will either be reduced, or stay the same, if we use a smaller n.
2. Bias: Consider the Bias. This is some complicated function of θ_{drop}. We can simplify it by making more assumptions, but that is not needed. What is clear is that Bias will become larger as n_{drop} is made larger (or equivalently, as $n = n_{\text{full}} - n_{\text{drop}}$ is made smaller). Thus, as we reduce n, we increase n_{drop}, and so we increase Bias.

Thus with reducing n, Variance decreases but Bias increases.

16 Regularization / Feature Selection / Picking reduced dimension r to minimize test-error / generalization error (optimize Bias-Variance Tradeoff)

As noted in previous section,

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

[3]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.

over test data, i.e., $\mathbb{E}[.] \equiv \mathbb{E}_{\text{test data}}[.]$. Usually $\boldsymbol{x}$ is treated as a constant, so then the expected value is over the distribution of the noise e.

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

$$\begin{aligned}\text{Test-MSE} &= \mathbb{E}[e^2] + \mathbb{E}[(f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x}))^2] \\ &= \mathbb{E}[e^2] + (\mathbb{E}[f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})])^2 + Variance[f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})] \\ &= \sigma^2 + Bias^2 + Variance\end{aligned}$$

where $Variance(Z) := E[(Z - E[Z])^2]$. The first term, σ^2, depends on how noisy the data is. The second two terms depend on the "assumed model" and how well its parameters are estimated.

16.1 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.

Simple approach: 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 $m = m_{train} + m_{test}$.
- Use the m_{train} data points to train, i.e. to estimate θ for the assumed model
- Approximate Test-MSE by $\frac{1}{m_{test}} \sum_{j=1}^{m_{test}} (y_j - h_{\hat{\theta}}(\boldsymbol{x}_j))^2$

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

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

16.2 How to reduce test error (generalization error) a.k.a. Regularization a.k.a. Feature Selection

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

1. Regularization on θ: stepwise regression; start with no features; at each iteration, select the one feature that is the "best" to add from remaining set. Define "best" using some heuristic (say t-test etc) or by model fitting using each new feature.

2. Regularization on θ: use domain knowledge such as assuming entries of θ are in decreasing order of magnitude. If this is known, then you just use the first r features. Vary r to pick which option reduces test data error the most.

3. Regularization on θ – model θ as being sparse (only a few features are important but we do not know which ones), this is a generalization of the assumption used in item 1. In this case, we add $\|\theta\|_1$ into the cost function for learning θ.

   In general, if the cost function is $J(\theta)$, then this approach says we replace it by $J(\theta) + \lambda\|\theta\|_1$

   In case of linear regression this becomes

   $$\min_{\theta} \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2 + \lambda\|\theta\|_1$$

   Do not try to write your own GD code for this. Use cvxopt or cvxpy etc.
   Vary λ as multiples of m, so $\lambda = \beta m$, with $\beta = 0.01, 0.1, 1, 10$ etc
   More information in Sec. 10.

4. Regularization on θ – use the assumption that all features are important and no feature is too important (no weight can be too large): then use the ℓ_2 regularization, i.e. add $\|\theta\|^2$ into the cost function for learning θ.

   In general, if the cost function is $J(\theta)$, then this approach says we replace it by $J(\theta) + \lambda\|\theta\|^2$

   In case of linear regression this becomes

$$\min_{\theta} \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2 + \lambda\|\theta\|^2$$

   This cost has closed form solution, given next.

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

   More information in Sec. 10.

5. Regularization on θ – suppose prior knowledge is available that θ is close to a given vector θ_0, then add the $\|\theta - \theta_0\|^2$ into the cost function for finding θ. See Sec. 10.2.

6. Regularization on input features' matrix $\boldsymbol{X}$ using PCA: PCA is a dimension reduction technique. It is used for feature selection under the assumption that "features having the most variation are the most important ones". Also of course PCA helps de-correlate the features. A typical setting where this is useful is when the raw feature vector is all image pixels. There are too many pixels and pixels are highly correlated. As explained earlier, PCA is a good pre-processing approach to regularization/feature-selection before doing classification (GDA or Log Reg or SVMs). *PCA does not do much if features are first "normalized" and then PCA is applied, see Sec. 16.3 and 16.4.*

7. In case of Clustering (say assuming Gaussian Mixture Model), reducing k (number of classes) will reduce the variance, increasing it will reduce the bias.

8. Naive Bayes assumption described earlier is another way to reduce the number of parameters. This is used in generative or Bayesian learning settings where the feature vector itself is modeled probabilistically.

With each new intervention, compute the Test-MSE as explained above, see if it gets reduced or not. In general as you keep reducing the number of features that are being used, variance reduces but bias increases. Initially when you begin reducing the number, the reduction in variance is significant but the increase in bias is negligible. So Test MSE will decrease. This will happen until the bias starts increasing (Test MSE starts increasing). Use the number of features that results in the smallest Test MSE.

16.3 Feature normalization

When ML features have vastly different ranges of values, e.g., in the house price example, one house feature can be number of bedrooms which takes values say between 1 to 5, while a second feature is the house areas in square feet, this could be a few 100 square feet. The two features have vastly different ranges.

If these were not normalized, then the matrix $\boldsymbol{X}$ would be ill-conditioned. This means that it will have a very high condition number. This, in turn, will imply inaccurate estimation if the closed form solution is used (due to numerical errors in inverting an ill conditioned $\boldsymbol{X}^\top\boldsymbol{X}$). If GD is used, the same can happen unless the step size is chosen to be very small. In this case, it will converge to the correct solution but will need too many iterations.

One solution is the normalize each feature: compute the j-th feature's mean μ_j and standard deviation σ_j; replace $\boldsymbol{x}_j$ by $(\boldsymbol{x}_j - \mu_j)/\sigma_j$.

16.4 Feature normalization vs PCA

Feature normalization is used in ML settings like the above where different features have different ranges because they are measured using very different measuring tools or sensors; and consequently, larger variance does not mean the feature is more important. In the house features example, number of bedrooms is as or important than house area.

PCA on the other hand is a "dimension reduction" technique that is used when all entries of the original data vector are acquired using the same type of sensor / measuring tool. For example, image pixels. All CCD sensors are the same type in a camera. In such settings, it may be a valid assumption to claim that pixels with larger variance are more relevant features. When this assumption is true, then the PCA can be used as a feature selection technique.

However PCA is not very useful if it is applied after feature normalization. Reason: feature norm is ensuring the new features all have unit variance. It can still be useful to de-correlate.

16.5 Use of prior knowledge and Recursive LS

In many settings, we may have prior knowledge that θ should be close to θ_0. For example, we may have θ_0 from houses sold a year ago and their features. In such cases, we solve

$$\min_\theta \|\boldsymbol{y} - \boldsymbol{X}\theta\|^2 + \lambda\|\theta - \theta_0\|^2$$

Recursive LS: consider house price example again and suppose we want to update our model each week. Instead of starting from scratch, can we just "update" the model: save time/computation?

Turns out we can using recursive LS: suppose we update with each new house sold, then we get this

$$\hat{\theta}_0 = \theta_0, P_0 = \lambda\boldsymbol{I},$$

$$\begin{aligned}
K_t &= P_{t-1}\boldsymbol{x}^{(i)}(\lambda + \boldsymbol{x}^{(i)\top}P_{t-1}\boldsymbol{x}^{(i)})^{-1}, \\
P_t &= (\boldsymbol{I} - K_t\boldsymbol{x}^{(i)\top})P_{t-1}, \\
\hat{\theta}_t &= \hat{\theta}_{t-1} + K_t(\boldsymbol{y}^{(i)} - \boldsymbol{x}^{(i)\top}\hat{\theta}_{t-1}),
\end{aligned}$$

Suppose we update after a set of m_t new houses sold (so $\boldsymbol{x}^{(i)}$ is an $n \times m_t$ matrix and $\boldsymbol{y}^{(i)}$ is a $m_t \times 1$ vector), then we get this

$$\hat{\theta}_0 = \theta_0, P_0 = \boldsymbol{I},$$

$$\begin{aligned}
K_t &= P_{t-1}\boldsymbol{x}^{(i)}(\lambda\boldsymbol{I} + \boldsymbol{x}^{(i)\top}P_{t-1}\boldsymbol{x}^{(i)})^{-1}, \\
P_t &= (\boldsymbol{I} - K_t\boldsymbol{x}^{(i)\top})P_{t-1}, \\
\hat{\theta}_t &= \hat{\theta}_{t-1} + K_t(\boldsymbol{y}^{(i)} - \boldsymbol{x}^{(i)\top}\hat{\theta}_{t-1}),
\end{aligned}$$

17 Learning Theory

based on Andrew Ng's cs229-notes-4

Hypothesis refers to a hypothesized model on the input output data. So suppose we assume that $y = sign(\theta^T\boldsymbol{x}))$ then $sign(\theta^T\boldsymbol{x}))$ 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

$$\mathcal{H} = \{h : h = sign(\theta^T\boldsymbol{x}),\ \forall\ \theta \in \Re^n\}$$

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

$$\hat{\varepsilon}(h) = \frac{1}{m}\sum_{i=1}^{m} loss(y^i, h(\boldsymbol{x}^i))$$

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

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

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).

Under the above assumptions, by using a law of large numbers' result, it can usually be argued that $\hat{\varepsilon}(h)$ converges to $\varepsilon(h)$ in probability as m goes to infinity. We will look at a simple zero-one loss function and actually work this out.

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

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

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

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

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

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

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 $\varepsilon(\hat{h})$ compared to $\varepsilon(h^*) = \min_{h\in\mathcal{H}} \varepsilon(h)$? We work this out for the zero-one loss.

17.1 Two probability results we use

Lemma 17.1 (Hoeffding inequality for Bernoulli r.v.'s)**.** *Let $Z_1, Z_2, \dots Z_m$ be iid Bernoulli random variables with parameter ϕ. Let $\hat{\phi} := \frac{1}{m}\sum_{i=1}^m Z_i$ be the empirical mean of these random variables. Then*

$$\Pr(|\hat{\phi} - \phi| > \gamma) \le 2e^{-2\gamma^2 m}$$

Lemma 17.2 (Union Bound)**.** *For any K events $A_1, A_2, \dots, A_K$*

$$\Pr(A_1 \cup A_2 \dots \cup A_K) \le \sum_{k=1}^K \Pr(A_k)$$

and thus

$$\Pr(A_1^c \cap A_2^c \dots \cap A_K^c) \ge 1 - \sum_{k=1}^K \Pr(A_k)$$

17.2 Misclassification (zero-one) loss

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

$$loss(y, \hat{y}) = \mathbb{1}(y \ne \hat{y}) = \mathbb{1}(y \ne h(\boldsymbol{x}))$$

For this, the empirical risk for a given h is

$$\hat{\varepsilon}(h) = \frac{1}{m}\sum_{i=1}^m \mathbb{1}(y^i \ne h(\boldsymbol{x}^i))$$

while

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

Define the r.v. $Z_i = \mathbb{1}(y^i \neq h(\boldsymbol{x}^i))$. Clearly the Z_i are Bernoulli with probability of one equal to $\varepsilon(h)$. We will use the Hoeffding inequality

Using Hoeffding inequality, this means that, for a given hypothesis h,

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

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

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

Then, by Union Bound,

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

or equivalently

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

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

$$|\hat{\varepsilon}(h) - \varepsilon(h)| \leq \gamma := \sqrt{\frac{1}{2m}\log\frac{2k}{\delta}}$$

Since above is true for all h, it is true for $\hat{h}$ and h^* too. So with the above probability,

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

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 γ, we can write the following theorem

Theorem 17.3. *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 $\geq 1-\delta$,

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

?? add corollary for lower bound on m.

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

$$k = \underbrace{2^{64} * 2^{64} \cdots * 2^{64}}_{d \text{ times}} = 2^{64d}$$

We have the following corollary

Corollary 17.4. *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 $\geq 1 - 2^{-d}$,

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

Remark 17.5. *In most of this class, we have assumed $d = n+1$ parameters. This is true for linear regression and logistic regression. For GDA, $d > n$.*

17.3 Tradeoffs

First note: we are discussing these tradeoffs using an "upper bound" on $\varepsilon(\hat{h})$. This upper bound may or may not be tight. The discussion is valid only when it is tight.

If we increase d (for a fixed m),

- the size of the hypotheses class on a 64-bit computer is $|\mathcal{H}_d| = 2^{64d}$ as argued above. Thus, increasing d, increases its size. This means, in the first term from the above corollary, $\min_{h \in \mathcal{H}_d} \varepsilon(h)$, we are taking a minimum over a larger set. Minimizing over a larger set implies the minimum value is the same or smaller (cannot be larger). Thus the first term can only *decrease* or *stay the same* if we increase d. This term does not depend on m at all.
- but the second term, $\sqrt{\frac{65d+1}{2m}}$ clearly increases linearly with d

Increasing d and hence the size of the hypotheses class is analogous to *reducing the bias*. Typically, up to a certain increase in d, the first term will decrease (bias will decrease). But when m is fixed, this also *increases the variance.*

In fact at the very least, we need $m > d$ to even just get the RHS to be smaller than one. Notice $\varepsilon(h)$ is a probability so has to be less than one,

17.4 Connections to what we have learnt

In case of the 0-1 loss, the empirical risk is computationally not possible to minimize because one will literally have to check all 2^{64d} possible hypotheses values and compute the empirical risk for each of them to find the minimize. It is what is called a "combinatorial optimization" problem and cannot be done in any reasonable time.

Also, as noted above, we are discussing these tradeoffs using an "upper bound" on $\varepsilon(\hat{h})$. This upper bound may or may not be tight. The discussion is valid only when it is tight.

For the above two reasons, the Tradeoffs' discussion does not apply to Generative Learning Models like Gaussian Discriminant Analysis where are placing many more assumptions on our data. For instance, in GDA, once we impose $\Sigma_0 = \Sigma_1 = \Sigma$ and Σ diagonal, we have basically decoupled the learning of each scalar parameter for each feature. In this case, as seen in your Home it is possible to get good parameter estimates and thus good classification even with $m < d$, in this case $d = 2n + n + 1$.

It "somewhat" applies to SVMs, though not directly since there we aren't trying to find a $\boldsymbol{w}, b$ to minimize empirical risk (training error), instead we are trying to find w, b to maximize the worst-case margin.

17.5 VC dimension

We will not talk about VC dimension (last two pages of Andew Ng's cs229 notes-4) in this course.