Lectures

1 Mistake Bounded Learning

1.1 Vocabulary

None yet

1.2 Introduction

Learner, Teacher, and Update Rule

Imagine a program that filters spam emails and has an iron-clad guarantee that it will only make 100 mistakes, even when your inbox has 30,000+ emails. This is called the mistake-bounded model of learning.

Learner

- Takes in samples (data points) and responds with its guess for the sample's classification.

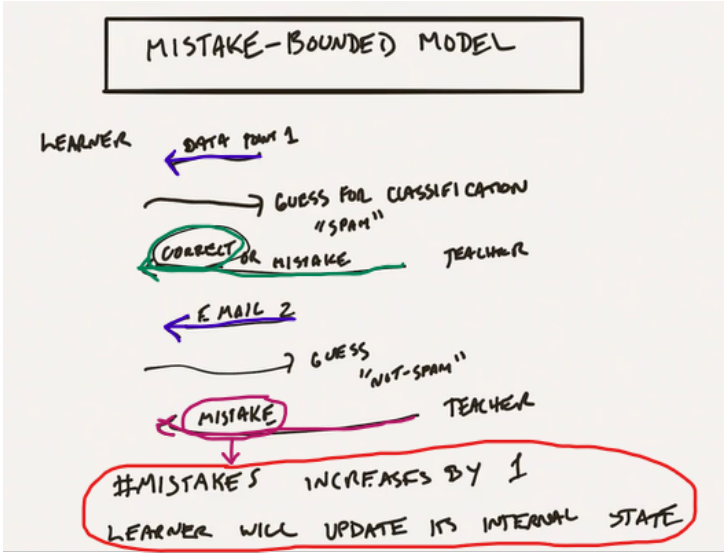
Teacher

- Responds to the classification guess with whether the guess was correct or incorrect.

Update Rule

- When the teacher tells the learner that it made a mistake, a counter for the number of mistakes increases by one. However, when the learner makes a mistake, it also learns from the mistake, updating its internal state.

We say a learner has mistake-bound t if for every sequence of challenges, the learner makes at most t mistakes.



Function Class Introduction

Let's introduce a function class called $\mathcal{C}$, which is the function class of all monotone disjunctions:

$$\mathcal{C} = \{\text{Monotone Disjunctions on } n \text{ variables}\}$$

And our domain, $\mathcal{D}$, consists of bit strings of length n:

$$\mathcal{D} = \{0,1\}^n$$

Some examples of functions in $\mathcal{C}$:

$$f(x) = x_1 \vee x_3$$
$$f(x) = x_1 \vee x_5 \vee x_7$$

1.3 Monotone Disjunctions

Mistake Bounded Model Example

Let us fix $f \in \mathcal{C}$, and the learner **does not** know f. The learner is trying to learn f, so it will guess 0 or 1, and the teacher will respond with correct if the guess equals $f(x)$, or will respond with a mistake otherwise.

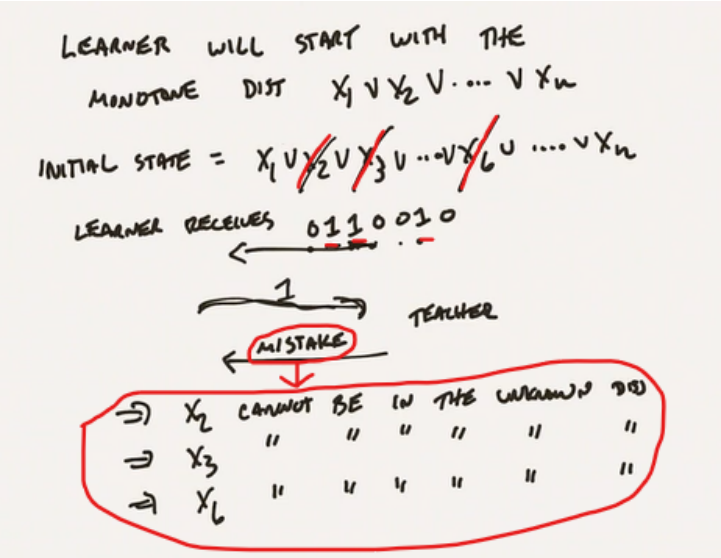
No matter if the guess was correct or not, the learner learns something after each challenge.

Question: How can we come up with a learner/algorithm that has a mistake bound of n?

In this case, we will start with some monotone disjunction as our initial state. Each time we guess incorrectly, we will update our monotone disjunction so it is consistent with what we've seen.

We will start with the learner using the monotone disjunction: $x_1 \vee x_2 \vee \cdots \vee x_n$ as its initial state. After each mistake, the learner will update its state to be consistent with the seen data.

With each mistake, we will be able to eliminate at least one literal from our monotone disjunction. There are at most n literals, which implies that the number of mistakes is at most n.



1.4 Disjunctions

Updating to a More Interesting Function Class

Let's update our function class $\mathcal{C}$ to be the function class of all disjunctions:

$$\mathcal{C} = \{\text{Disjunctions}\}$$

And our domain, $\mathcal{D}$, consists of bit strings of length n:

$$\mathcal{D} = \{0,1\}^n$$

We can now have negations in our disjunctions. Some examples of functions in the new $\mathcal{C}$:

$$f(x) = x_1 \vee \bar{x_2} \vee x_5 \vee \bar{x_7}$$

Question: How can we use the algorithm for monotone disjunctions to learn disjunctions?

We will perform something called "feature expansion," where we take a bit string of length n and map it to a new string of length $2n$:

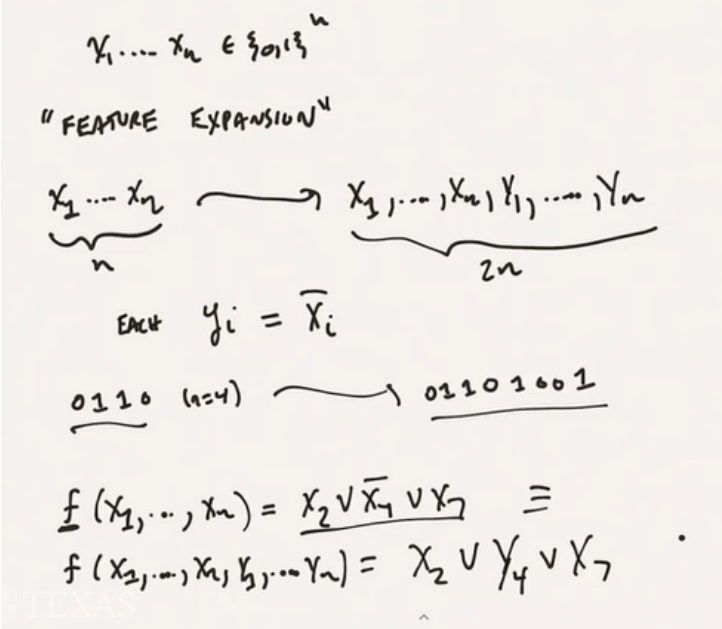
$$x_1, \ldots, x_n \mapsto x_1, \ldots, x_n, y_1, \ldots, y_n$$

Where each $y_i = \bar{x_i}$, meaning y_i equals the negation of x_i.

Thus, each $f(x_1, \ldots, x_n)$ can be rewritten as a new function $f(x_1, \ldots, x_n, y_1, \ldots, y_n)$ that behaves the same:

$$f(x_1, \ldots, x_n) = x_2 \vee \bar{x_4} \vee x_7$$

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = x_2 \vee y_4 \vee x_7$$



We now have a new algorithm for arbitrary disjunctions with mistake bound $\leq 2n$.

1.5 Personal Notes

On-Line Algorithms in Machine Learning

2 Decision Trees

2.1 Vocabulary

None yet

2.2 Introduction and Construction

A decision tree is a boolean function (outputs true or false). At each node in the decision tree, there is a literal. At the leaves, there is a fixed value, which is the output.

The size of the decision tree is the number of nodes in the tree. The depth (or height) of the tree is the length of the longest path from the root to a leaf.

Note that for an input going into a decision tree, the x is referred to as a "challenge", and the y a "label".

Topics:

- Heuristics for learning decision trees
- Theoretical properties

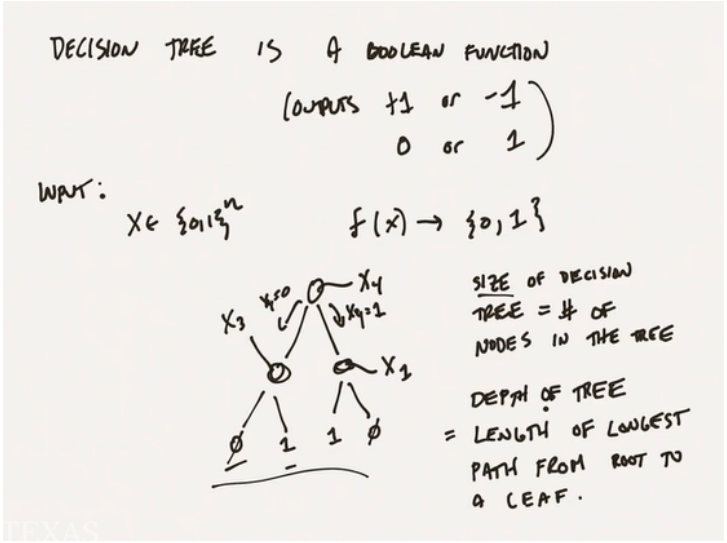
—

Example input: $X \in \{0,1\}^n$ (bit string of length n)

The decision tree is going to encode some function $f(x)$ into $\{0,1\}$ as follows:

- At each node, the tree decides which branch to take based on the value of the literal, until it reaches the leaf.

For example, the decision tree's depth = 2, and size = 3.



—

The machine learning problem:

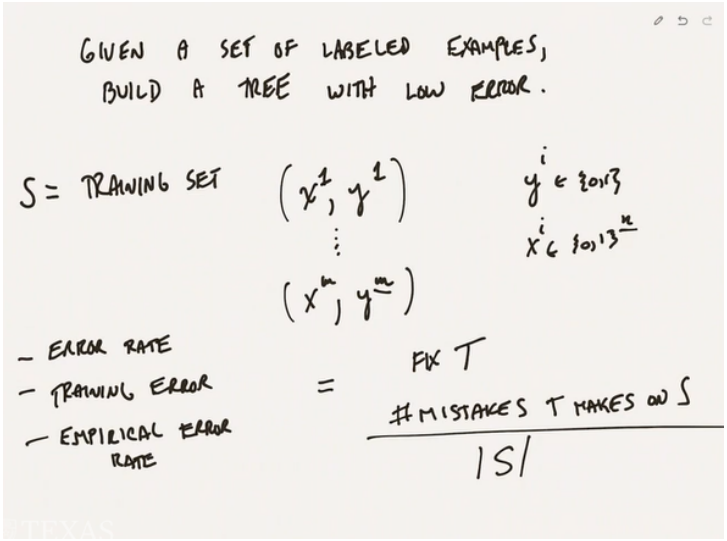
- Given a set of labeled examples, build a tree with low error.

—

Let $\mathcal{T}$ be a training set, where $\mathcal{T}$ is a collection of strings and 0, 1 labels.

- So c is a collection of X's and y's, where $X \in \{0,1\}^n$, and $y \in \{0,1\}$.

Error Rate/Training Error/Empirical Error Rate is defined as the number of mistakes that $\mathcal{T}$ makes on the training set, divided by the size of $\mathcal{T}$.

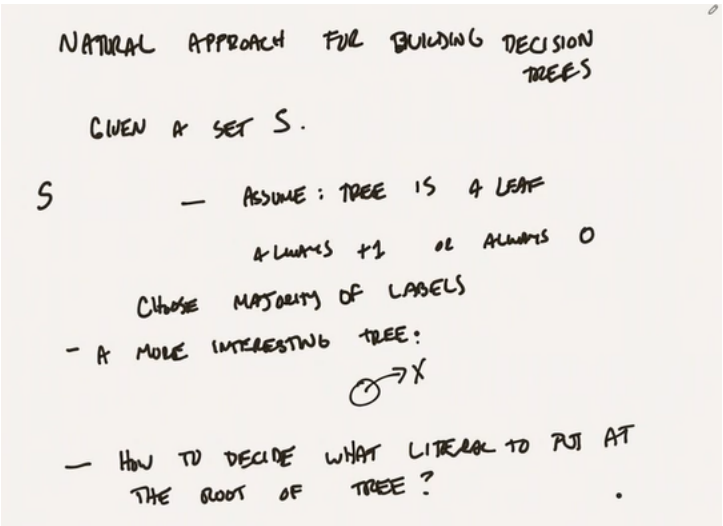


—

2.2.1 Natural Approach for Building Decision Trees

- Given a set $\mathcal{T}$:
  - **Tree 1:** Very simple, trivial tree
    - Tree is a leaf (we don't query any literals, always output 0 or 1).
    - How do we decide what to output?
      - Choose 1 or 0 depending on which label is more prevalent in the dataset.
  - **Tree 2:** More advanced tree
    - Tree has one node, the root.

* How do we decide which literal to put at the root?
  · You want a literal at the root that discriminates between zero and one labels.



—

How do we decide which literal to put at the root?
Define a potential function $\Phi(a)$:

$$\Phi(a) = \min(a, 1-a)$$

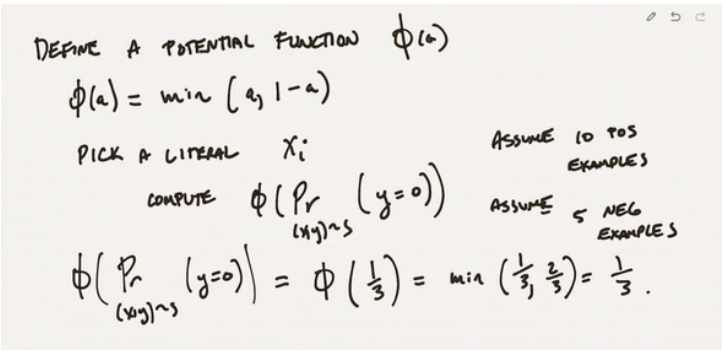
—

For the trivial decision tree, pick a literal x_i, then compute $\Phi(\Pr_{(x,y)\sim\mathcal{T}}(y=0))$.

- Assume: 10 positive examples.
- Assume: 5 negative examples.

Thus, $\Phi(\Pr_{(x,y)\sim\mathcal{T}}(y=0)) = \frac{1}{3}$.

$$\Phi(\Pr_{(x,y)\sim\mathcal{T}}(y=0))$$



—

Looking at the tree with one node, pick a literal x_1 as the root node.
For the first leaf:

- Condition on $x_1 = 0 \rightarrow$ output the majority value.

For the second leaf:

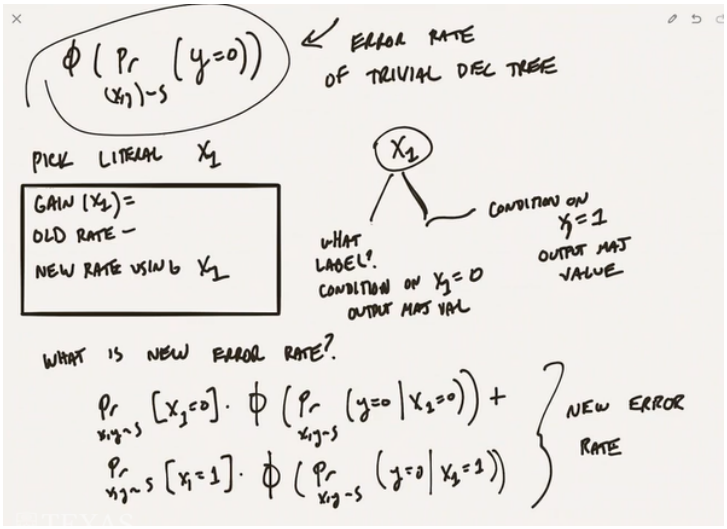
- Condition on $x_1 = 1 \rightarrow$ output the majority value.

The error rate is the weighted average of the errors of each leaf:

$$\Pr_{(x,y)\sim\mathcal{T}}[x_1=0] \cdot \Phi(\Pr_{(x,y)\sim\mathcal{T}}(y=0)|x_1=0) + \Pr_{(x,y)\sim\mathcal{T}}[x_1=1] \cdot \Phi(\Pr_{(x,y)\sim\mathcal{T}}(y=0)|x_1=1)$$

—

Gain(x_1) = Old Rate - New Rate using x_1
This is the gain in training error by moving from the trivial decision tree to the tree where we put x_1 at the root.



—

We can now compute the gain of each literal x_i and find which literal maximizes the gain, placing that literal at the root.

Once this is done, each branch uses a subset of the original set. For example, the left branch will use the training set $\mathcal{T}_{|x_1=0}$, and the right branch will use $\mathcal{T}_{|x_1=1}$.

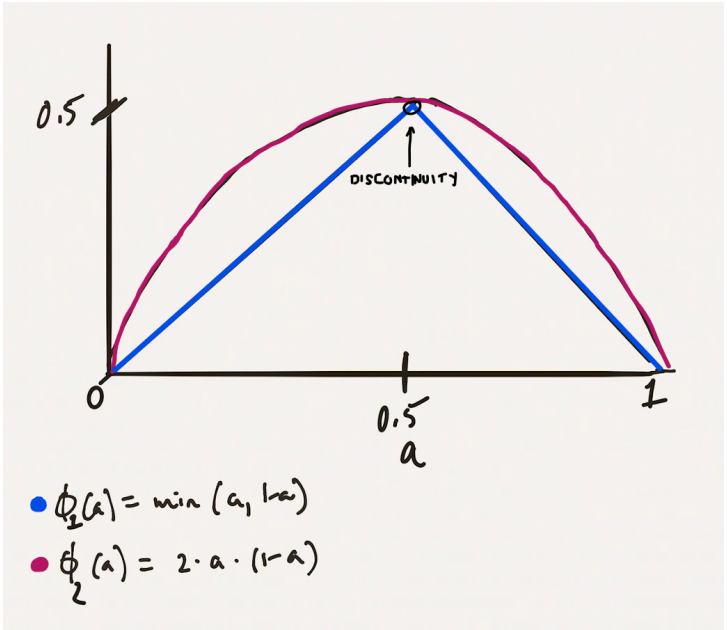
We repeat the process until the tree is complete. Whether this is computationally feasible depends on the size of the tree and complexity of the functions.

—

2.3 Potential Functions and Random Forests

2.3.1 Tree Structure

The structure of the tree is determined by the choice of potential function, ϕ. For example, we used $\phi(a) = \min(a, 1-a)$, corresponding to training error. Another common potential function is the Gini function $\phi = 2 \cdot a \cdot (1-a)$.



We can see that ϕ_2 is an upper bound on ϕ_1, and it is smooth. Because ϕ_2 is an upper bound:

$$\text{small values of } \phi_2 \implies \text{small values of } \phi_1$$

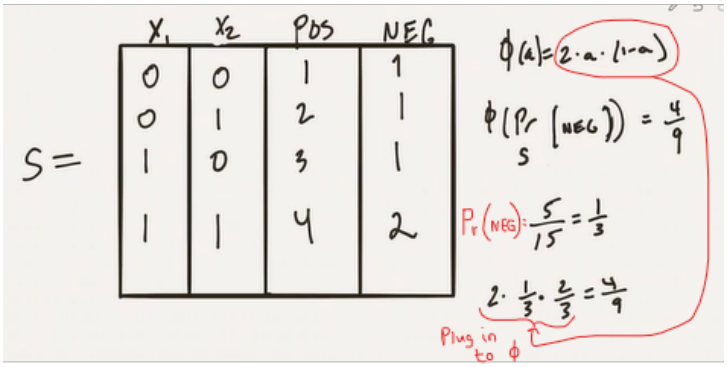
—

Example with Gini Index

Using the table S and the potential function $\phi(a) = 2 \cdot a \cdot (1-a)$:

x_1	x_2	Pos	Neg
0	0	1	1
0	1	2	1
1	0	3	1
1	1	4	2

What is $\phi_S(Pr(Neg))$? The value is $\frac{4}{9}$ for the trivial tree.



—

Now, should we pick x_1 or x_2 to be at the root?
For x_1:

$$\Pr(x_1 = 0) \cdot \left(2 \cdot \frac{2}{5} \cdot \frac{3}{5}\right) + \Pr(x_1 = 1) \cdot \left(2 \cdot \frac{3}{10} \cdot \frac{7}{10}\right) = \frac{11}{25}$$

For x_2:

$$\Pr(x_2 = 0) \cdot \left(2 \cdot \frac{1}{3} \cdot \frac{2}{3}\right) + \Pr(x_2 = 1) \cdot \left(2 \cdot \frac{3}{9} \cdot \frac{7}{9}\right) = \frac{4}{9}$$

—

We compute the gain of each x. For x_1:

$$\frac{4}{9} - \frac{11}{25} > 0$$

For x_2:

$$\frac{4}{9} - \frac{4}{9} = 0$$

Thus, x_1 has the greatest gain and is the best choice for the root node.

—

2.3.2 When to Stop

- Stop when the gain is extremely small for all literals.
- Pruning: Build a large tree and remove branches with minimal effect.

2.3.3 Random Forests

Random forests involve building many small decision trees and taking the majority vote of the resulting trees.
Algorithm:

- Take training set S.
- Randomly subsample from S to create S'.
- Randomly choose some features $\{x_1, \ldots, x_n\}$ of size k.
- Build a decision tree using S' and the k random features.

2.4 Personal Notes

Understanding Machine Learning: From Theory to Algorithms, Chapter 18

3 Generalization

3.1 Vocabulary

None yet

3.2 Lecture Notes

3.2.1 Introduction

Generalization, or predictive power of a classifier

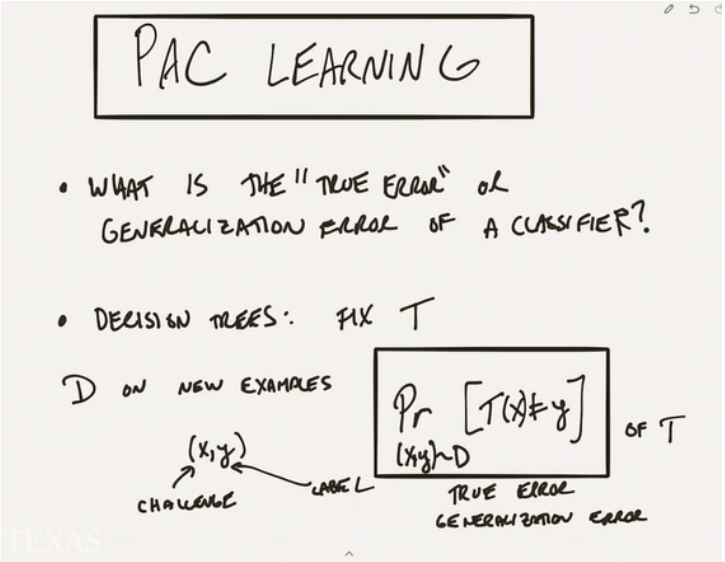
The goal is to understand how well a classifier performs when given unseen data. Estimating the generalization error and understanding when it will have good generalization leads to the PAC model of learning, a foundational model.

- What is the "true error" or generalization error of a classifier? - Consider decision trees: - Fix some tree T, created using the rules discussed earlier. - Assume we have a probability distribution D on new examples. - For a new challenge and label, we want to know:

$$Pr_{(x,y)\sim D}[T(x) \neq y]$$

[English: The probability that when we randomly draw a challenge from some unknown distribution D, $T(x)$ does not equal y]

This is the **true error** or generalization error of T. We hope this value is small.

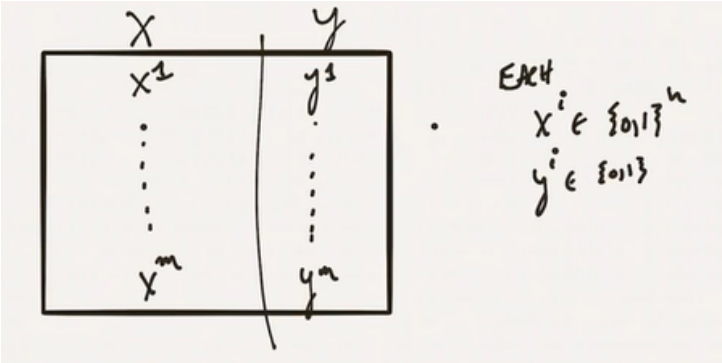


—

3.2.2 When Might This Quantity Not Be Small?

Consider a training set $\mathcal{T}$, with challenges $x^1, \ldots, x^m$ and labels $y^1, \ldots, y^m$ where $x^i \in \{0,1\}^n$ and $y^i \in \{0,1\}$. Assume all x^i are distinct.

A learner given $\mathcal{T}$ provides a classifier that memorizes the training set exactly. This results in poor generalization since no true learning occurred.



—

Now consider constructing a decision tree consistent with all points in $\mathcal{T}$.

Question: How well does this tree generalize? - It will likely generalize poorly since it is memorizing the training set. - In practice, we seek a balance between low training error and low generalization error.

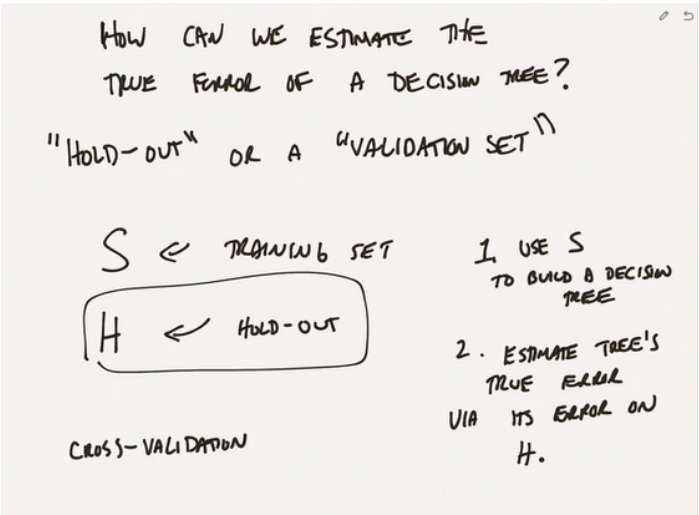
—

3.2.3 Estimating the True Error of a Decision Tree

A **hold-out** or **validation set** can be used to estimate true error. The procedure is as follows:

1. Use the training set $\mathcal{T}$ to build a decision tree. 2. Estimate the tree's true error via its error on the validation set $\mathcal{H}$.

By counting the number of mistakes the tree makes on $\mathcal{H}$, we compute its error rate on $\mathcal{H}$ as an estimate of the true error. As long as $\mathcal{H}$ is sufficiently large, the estimate will be close to the true error.



—

3.3 Model Complexity

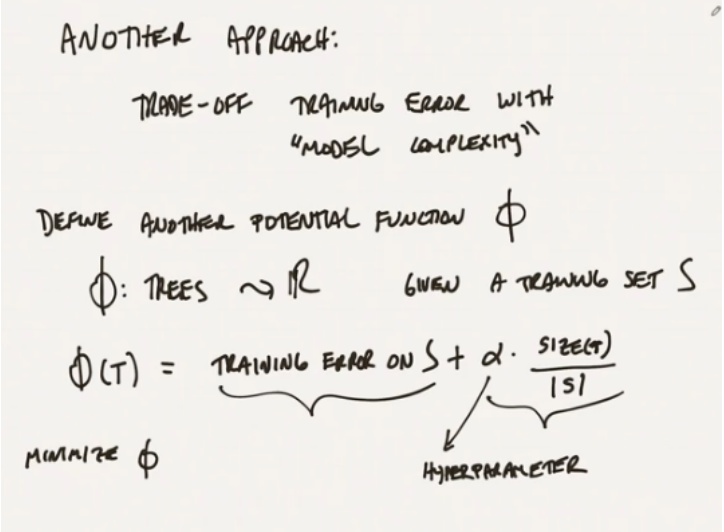
3.3.1 Trading Training Error for Model Complexity

Another method to estimate the true error is to balance training error with model complexity. We define a potential function ϕ as follows:

$$\phi(T) = \text{training error on } \mathcal{T} + \alpha \cdot \frac{\text{size}(T)}{|\mathcal{T}|}$$

[English: $\phi(T)$ outputs the training error on $\mathcal{T}$ plus a value $\alpha \cdot \text{size}(T)/|\mathcal{T}|$.]

The goal is to minimize $\phi(T)$, balancing training error with tree size. A smaller tree with reasonable training error tends to generalize better.



—

3.3.2 Minimum Description Length (MDL)

The **Minimum Description Length** (MDL) principle is another approach. Given a training set $\mathcal{T}$, the tree T can be encoded with bits:

$$\text{bits}(T) + \#\text{bits to encode remaining errors}$$

This approach emphasizes compression: a tree with fewer bits and fewer errors is preferred.

—

3.4 PAC Model of Learning

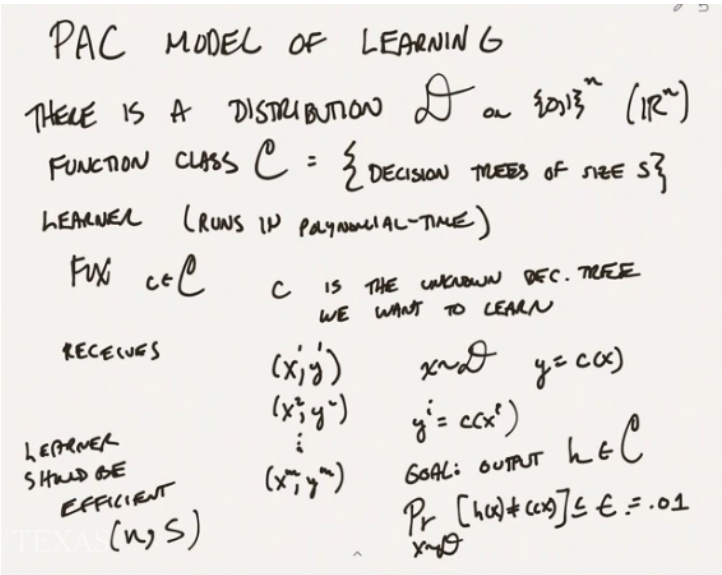
Consider a distribution D on $\{0,1\}^n$ (the domain) and a function class $\mathcal{C}$:

$$\mathcal{C} = \{\text{decision trees of size } S\}$$

The learner's goal is to output a hypothesis $h \in \mathcal{C}$ such that:

$$Pr_{x \sim D}[h(x) \neq c(x)] \leq \epsilon$$

[English: The probability that $h(x)$ does not equal $c(x)$ for x drawn from D is at most ϵ].
The learner should run in polynomial time, bounded by n and s.



—

Formalization:

With probability at least $1 - \delta$, the learner outputs h such that:

$$Pr_{x \sim D}[h(x) \neq c(x)] \leq \epsilon$$

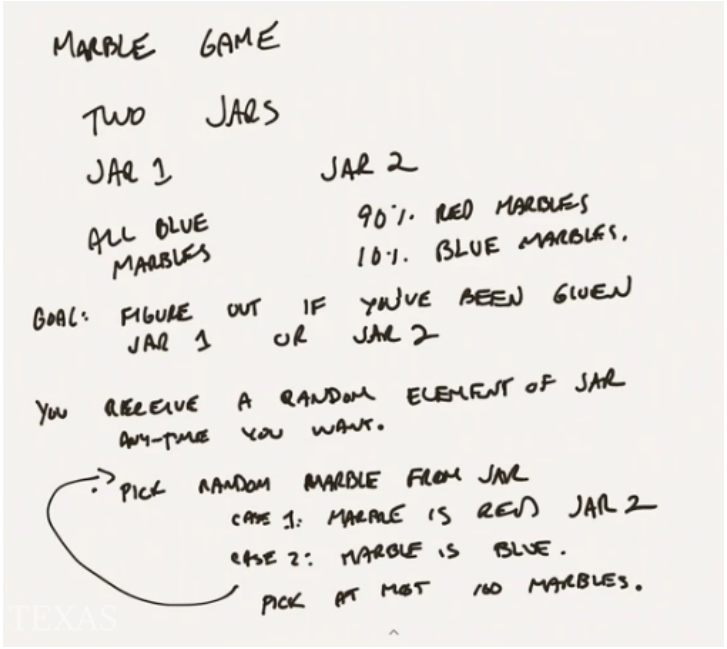
and the runtime is bounded by:

$$\text{runtime} = \text{polynomial}\left(\frac{1}{\epsilon}, \frac{1}{\delta}, n, s\right)$$

3.4.1 Illustration: Marble Game

Imagine two jars: - Jar 1: All blue marbles. - Jar 2: 90% red, 10% blue.

Draw marbles randomly. After a few draws, determine which jar you've been given. Failure probability decreases exponentially with more draws, analogous to the δ parameter in PAC learning.

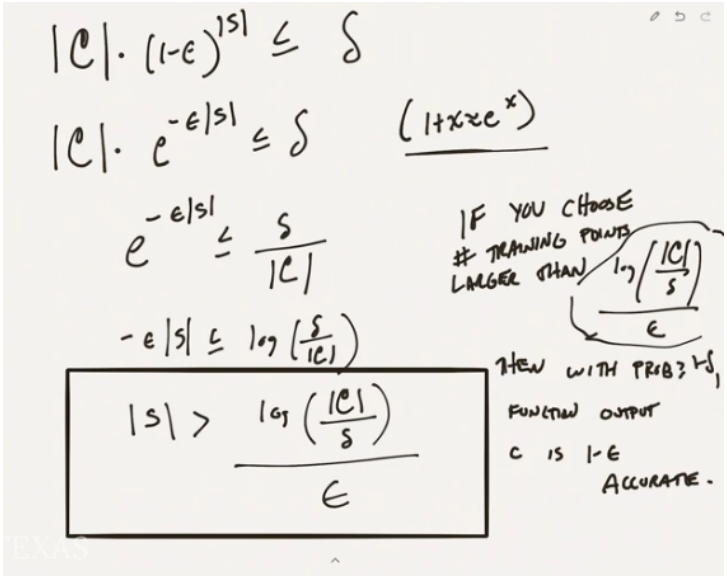


—

In PAC learning, the sample size $|S|$ must satisfy:

$$|S| \geq \frac{\log\left(\frac{\delta}{|C|}\right)}{\epsilon}$$

This ensures that with probability $\geq 1 - \delta$, the output function c is $1 - \epsilon$ accurate.



3.5 Personal Notes

Understanding Machine Learning: From Theory to Algorithms, Chapter 3

4 PAC Learning

4.1 Vocabulary

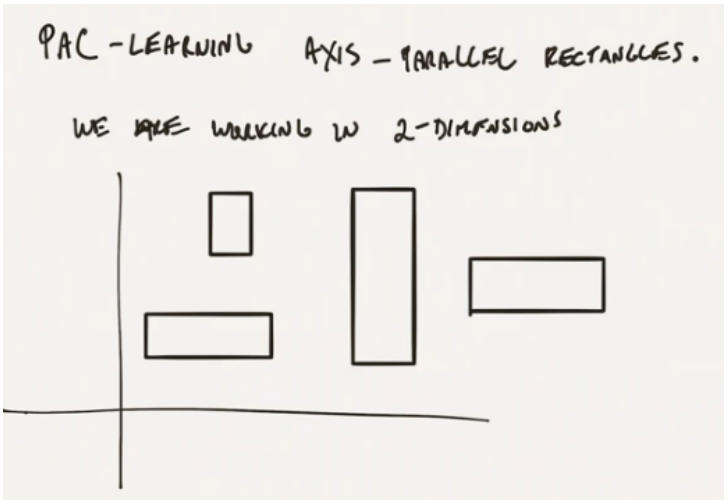
None yet

4.2 Lecture Notes

4.2.1 Infinite Function Class

PAC-learning axis-parallel rectangles

We are working in two dimensions, and axis-parallel rectangles implies the axes are parallel to the x and y axis.

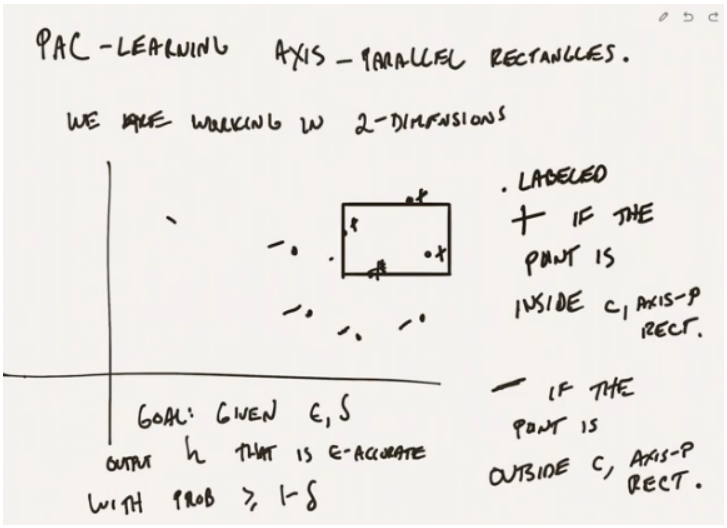


Learning problem: We are given points, where some are labeled positive (inside the unknown rectangle c) and others negative (outside c).

Goal: Given ϵ and δ, output hypothesis h that is ϵ-accurate with probability $\geq 1 - \delta$.

Recall in 1.3.2, we used algorithm A, which produced a consistent result because it depended on the size of the function class $|\mathcal{C}|$. Here, we have infinitely many axis-parallel rectangles.

The tightest fitting rectangle will solve this problem.

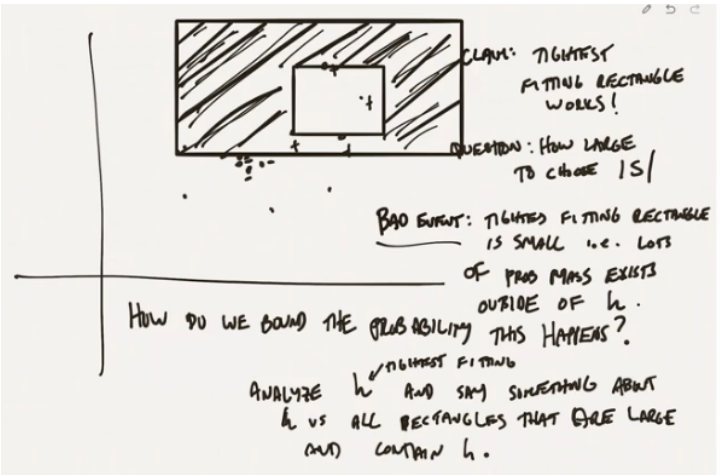


Claim: The tightest fitting rectangle works for this problem.

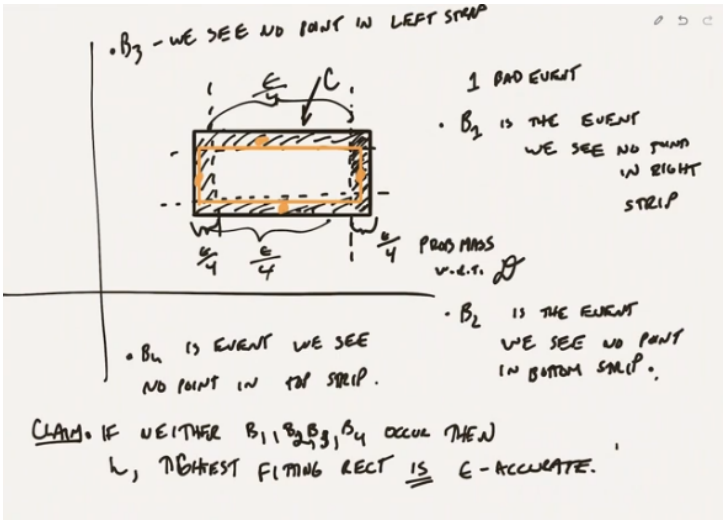
Question: How large should the training set $|\mathcal{S}|$ be?

Bad event: Positive points are clustered around a tiny rectangle, meaning significant probability mass lies outside h (the tightest rectangle). If the probability of landing in that mass is $\geq \epsilon$, we are in trouble.

How do we bound the probability of this happening?



If none of B_1, B_2, B_3, or B_4 occur (see image), then h (the tightest rectangle) is ϵ-accurate.



We still need to figure out how large $|\mathcal{S}|$ should be.
For m random samples, $Pr[B_1] \leq (1 - \frac{\epsilon}{4})^m$. By that logic:

$$Pr[B_1 \vee B_2 \vee B_3 \vee B_4] \leq 4(1 - \frac{\epsilon}{4})^m \leq \delta$$

Using the approximation $1 - x \approx e^{-x}$:

$$4(1 - \frac{\epsilon}{4})^m \leq \delta$$

$$(1 - \frac{\epsilon}{4})^m \leq \frac{\delta}{4}$$

$$e^{-\frac{\epsilon m}{4}} \leq \frac{\delta}{4}$$

$$-\frac{\epsilon m}{4} \leq \log\left(\frac{\delta}{4}\right)$$

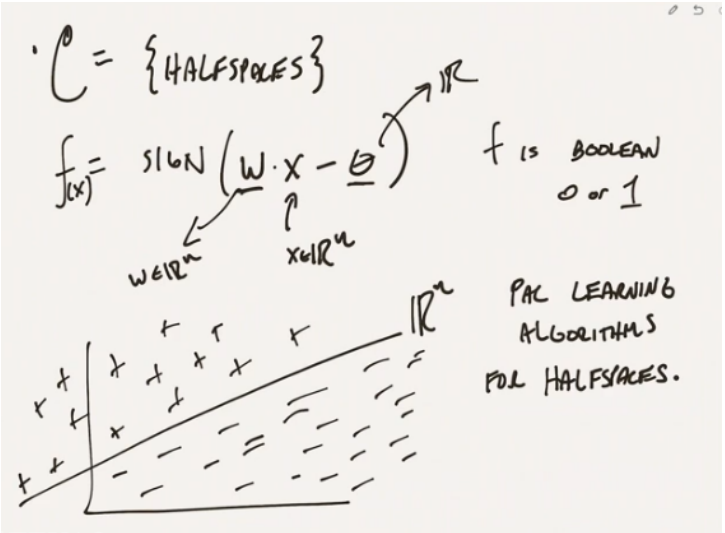
$$m \geq \frac{4\log\left(\frac{\delta}{4}\right)}{\epsilon}$$

Thus, as long as we choose at least $m \geq \frac{4\log\left(\frac{\delta}{4}\right)}{\epsilon}$, h will be ϵ-accurate with probability $\geq 1 - \delta$.
Correction:
At around 28:30 of 1.4.0, $m \geq \frac{4\log\left(\frac{\delta}{4}\right)}{\epsilon}$ should instead be $m \geq \frac{4\log\left(\frac{4}{\delta}\right)}{\epsilon}$. Note the inversion of the expression inside the logarithm.

4.2.2 Half Spaces

Another interesting class is $C = \{\text{half spaces}\}$. This is a function that takes x and outputs $f = \text{sign}(w \cdot x - \theta)$, where $w \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, and θ is a scalar ($\mathbb{R}$).

The output f is boolean (0 or 1), with 0 if the result is ≤ 0, and 1 if the result is > 0. Geometrically, this divides n-dimensional Euclidean space into two half spaces.



One approach for learning half spaces:

Recall that w and θ are unknown. The function to learn is:

$$f = \text{sign}\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$

Draws from D are given as $(x, f(x))$, where x is distributed according to D.

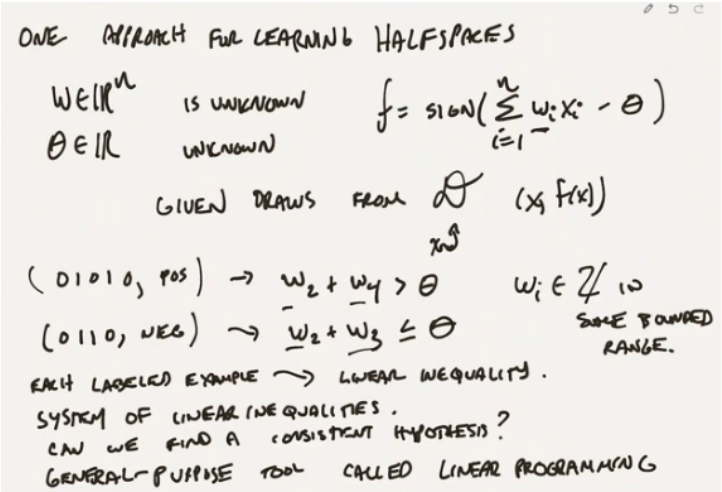
$$(01010, \text{pos}) \Rightarrow w_2 + w_4 > \theta$$
$$(0110, \text{neg}) \Rightarrow w_2 + w_3 \leq \theta$$

Each labeled example corresponds to a linear inequality, producing a system of inequalities.

Question: Can we find a consistent hypothesis?

Assuming $w_i \in \mathbb{Z}$ in some bounded range, we can apply consistency analysis. Given a system of inequalities, we can solve for w_i using **linear programming**.

Linear programming algorithms can solve general systems of inequalities in polynomial time.



4.3 Personal Notes

5 Cross-Validation

5.1 Vocabulary

None yet

5.2 Lecture Notes

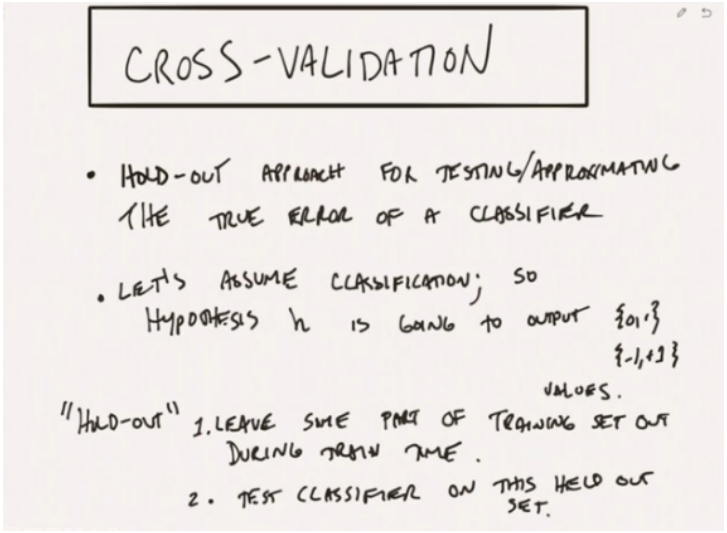
5.2.1 Introduction

We will be looking at how to calculate the true error of a classifier model.

Let us assume classification; the hypothesis h outputs boolean values (e.g., {0,1}, {-1,1}).

Hold-out approach (validation set) for testing/approximating the true error of a classifier:

- Leave some part of the training set out during training.
- When evaluating the true error of the classifier, test the classifier on this held-out set.
- The error on the held-out set approximates the true error for unseen data.



Markov's Inequality

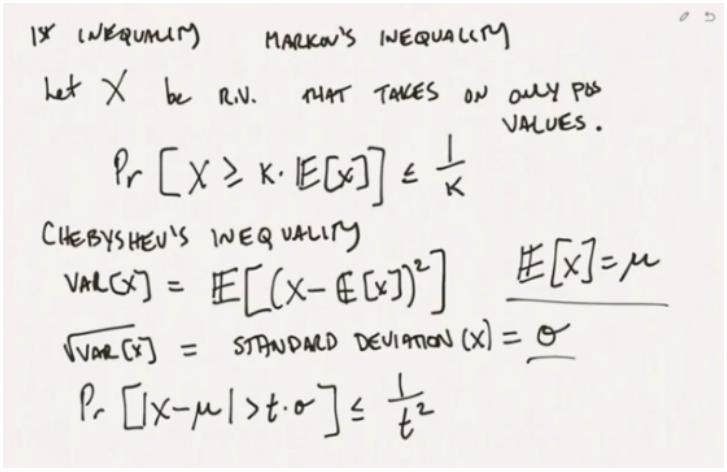
- Let x be a random variable that takes only positive values.
- $Pr[x \geq k \cdot \mathbf{E}[x]] \leq \frac{1}{k}$.
- *The probability that x is more than k times the expected value of x is at most $\frac{1}{k}$.*

Chebyshev's Inequality

- Let $\mathbf{E}[x] = \mu$.
- Variance of a random variable: $Var[x] = \mathbf{E}[(x - \mathbf{E}[x])^2]$, which measures how much x deviates from its expectation.
- Recall: $\sqrt{Var[x]} = \sigma$, the standard deviation.

If we have a random variable with mean μ and variance σ, the probability that x deviates from μ by more than t standard deviations is at most $\frac{1}{t^2}$:

$$Pr[|x - \mu| > t \cdot \sigma] \leq \frac{1}{t^2}$$



5.2.2 Chernoff Bound

Let us say we have random variables $x_1, x_2, \ldots, x_n$ where $x_i \in \{0, 1\}$, and $\mathbf{E}[x_i] = p$ (fixing p for simplicity).

Define:

$$S = \sum_{i=1}^{n} x_i$$

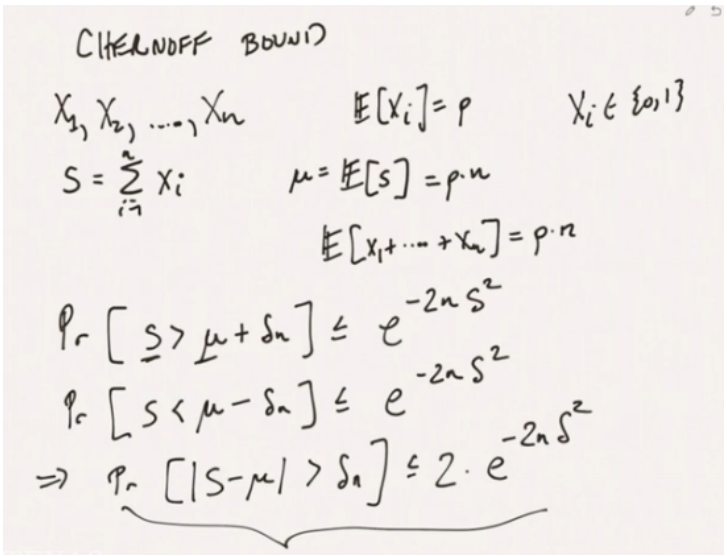
Also, let $\mathbf{E}[S] = \mu = p \cdot n$. The Chernoff Bound states:

$$Pr[S > \mu + \delta n] \leq e^{-2n\delta^2}$$

$$Pr[S < \mu - \delta n] \leq e^{-2n\delta^2}$$

$$\Rightarrow Pr[|S - \mu| > \delta n] \leq 2e^{-2n\delta^2}$$

The probability that S deviates from μ is exponentially small in n. Depending on the choice of δ, we obtain different bounds for these probabilities.



Applying the Chernoff Bound to estimate the true error of a classifier:

Let S be the hold-out set of size n, and fix h. There is some underlying distribution D from which we are generating points, and S is a sample drawn from D independently of the training set. Let:

$$Z = Pr_{x \sim D}[h(x) \neq c(x)]$$

where c is the unknown function we are trying to learn, and h is the classifier.

Define random variables x_i for the Chernoff Bound:

$$x_i = \begin{cases} 1 & \text{if } h \text{ is incorrect on the } i\text{-th element of } S, \\ 0 & \text{otherwise} \end{cases}$$

and

$$S = \sum_{i=1}^{n} x_i$$

with

$$\mathbf{E}[S] = n \cdot p$$

where p is the true error of h. The Chernoff Bound gives:

$$Pr[|S - n \cdot p| > \delta n] \leq 2e^{-2n\delta^2}$$

Let $\delta = 0.1$, so:

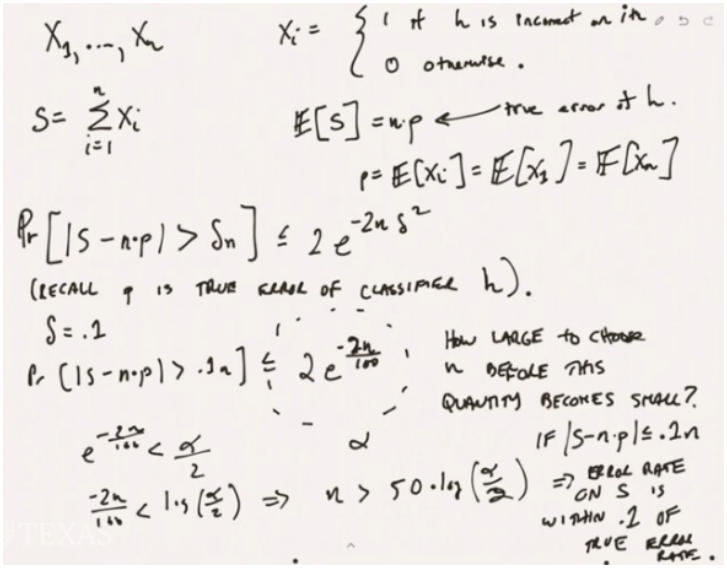
$$Pr[|S - n \cdot p| > 0.1n] \leq 2e^{-\frac{2n}{100}}$$

The quantity $2e^{-\frac{2n}{100}}$ is the confidence parameter. How large should n be to ensure this confidence parameter is smaller than α?

$$2e^{-\frac{2n}{100}} < \alpha$$
$$e^{-\frac{2n}{100}} < \frac{\alpha}{2}$$
$$\frac{-2n}{100} < \log\left(\frac{\alpha}{2}\right) \Rightarrow n > 50 \cdot \log\left(\frac{2}{\alpha}\right)$$

Thus, to ensure the probability of failure is less than α and that our estimate is within $0.1 \cdot n$, we need:

$$n > 50 \cdot \log\left(\frac{2}{\alpha}\right)$$



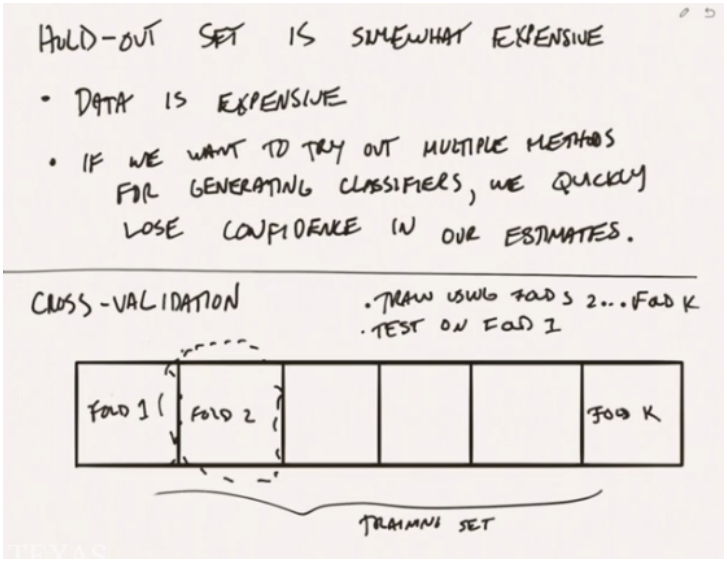
5.2.3 How It Works

The hold-out set approach is somewhat expensive:

- Data can be expensive to obtain, and we are not using it for training.
- Multiple evaluations on the hold-out set reduce confidence, as each reuse multiplies α.

How can we build and evaluate multiple classifiers while still understanding their true error? Cross-validation offers a practical solution.

Cross-validation is widely used (e.g., in `scikit-learn`). The idea is to divide the training set into *folds* and use them both to train the classifier and estimate the true error.

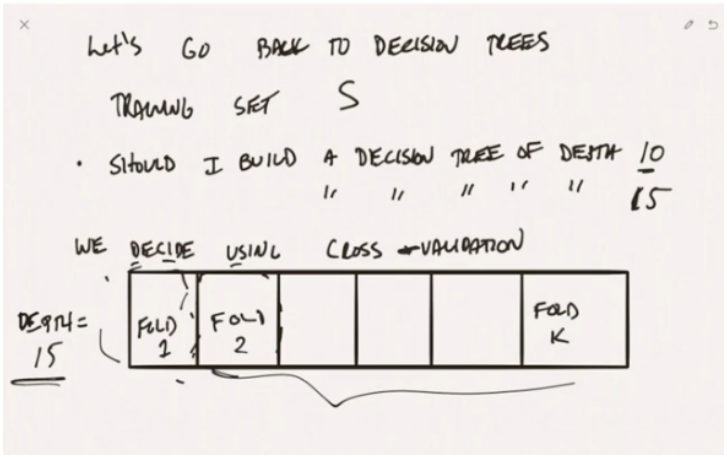


Using decision trees as an example:

- To decide between a decision tree of depth 10 or depth 15, apply cross-validation.
- Create folds in the training set, set depth = 10, leave out fold 1, and test on it. Repeat for all folds, then average their errors.
- Repeat for depth = 15.
- The model with the lower average error is preferred.

Question: What should k (the number of folds) be?

- Typically, k is set between 5 and 10.



5.3 Resources

[Understanding Machine Learning: From Theory to Algorithms, Chapter 3](https://www.cs.huji.ac.il/ shais/UnderstandingMachineLe

5.4 Personal Notes

None yet

6 Perceptron Learning

6.1 Vocabulary & Code

Euclidean Norm The L2 norm, or Euclidean norm, measures the length of a vector. It is calculated as:

$$||w^*||_2 = \sqrt{\sum_i (w_i^*)^2}$$

Importance:

- **Size:** Measures the vector's magnitude.
- **Optimization:** Helps in regularization to avoid large weights.
- **Distance:** Used for calculating distances between points or vectors.

Python Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the linear function
def linear_function(x1, x2):
    return 2 * x1 + 3 * x2 + 1

# Define the weight vector and bias
w = np.array([2, 3])
b = 1

# Define the decision boundary line
def decision_boundary(x1):
    return - (w[0] * x1 + b) / w[1]

# Create data points
points = np.array([
    [1, 1],
    [2, -1],
    [-1, 2],
    [1, -2],
    [0, -1]
])

# Compute function values
values = np.array([linear_function(x1, x2) for x1, x2 in points])

# Extract x1 and x2 for plotting
x1 = points[:, 0]
x2 = points[:, 1]

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the decision boundary
x1_vals = np.linspace(-3, 3, 400)
x2_vals = decision_boundary(x1_vals)
plt.plot(x1_vals, x2_vals, 'r--', label='Decision Boundary')

# Plot the data points
plt.scatter(x1, x2, c=values, cmap='magma', s=100, edgecolor='k', label='Data Points')
plt.colorbar(label='Function Value')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Data Points and Decision Boundary')
plt.legend()
plt.grid(True)
plt.show()
```

6.2 1.6.0 Introduction

The perceptron is a learning algorithm for half-spaces that has provable guarantees.

The function class we will be learning is:

$$C = \{\text{Halfspaces}\}$$

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$

Where θ is some scalar threshold, and the vector $\mathbf{w}$ is unknown. We assume $f(x) \in \{-1, 1\}$ and $x_i \in \mathbb{R}$.

The perceptron evaluates a linear function on some unknown weight vector $\mathbf{w}$, and then takes its sign.

Algorithm Overview

Initially, we guess the weight vector, which could be $w^0 = (0, ..., 0)$ or the unit vector.

The learner has $\mathbf{w}$ as its state, and the teacher presents a challenge, $x \in \mathbb{R}^n$. The learner responds with $\text{sign}(w \cdot x)$. If a mistake is made, the learner updates its state based on the following rule:

Update Rule:

$$w_{\text{new}} = w_{\text{old}} + y \cdot x$$

Where y is the label ($y \in \{-1, 1\}$).

6.3 1.6.1 Expanding on the Update Rule

To prove the algorithm works, we assume:

Assumptions:

1. $\exists w^*$, true unknown weight vector, $||w^*||_2 = 1$
2. Every x has norm 1, i.e., $||x||_2 = 1$
3. $\theta = 0$, threshold equals zero

The main assumption is the **Margin Assumption**, which posits that there exists a margin ρ, and all points are at least distance ρ from w^*.

Perceptron Convergence Theorem

The mistake bound of the perceptron algorithm is $\mathcal{O}\left(\frac{1}{\rho^2}\right)$.

Proof: Let t be the number of mistakes made at any point.

After t mistakes:

$$t \cdot \rho \leq w \cdot w^* \leq ||w|| \cdot ||w^*||$$

Given the norm constraints, this simplifies to:

$$t \leq \frac{1}{\rho^2}$$

6.4 1.6.2 Proving Claims 1 & 2

Claim 1: On every mistake, $w \cdot w^*$ increases by at least ρ.

Proof: The update rule is $w_{\text{new}} = w_{\text{old}} + y \cdot x$. We know from the margin assumption that $y \cdot x \cdot w^* \geq \rho$. Thus, the inner product increases by at least ρ.

Claim 2: $||w||^2$ increases after every mistake by at most 1.

Proof: Using the update rule:

$$||w_{\text{new}}||^2 = ||w_{\text{old}}||^2 + 2y\langle x, w_{\text{old}}\rangle + ||x||^2$$

Since $||x||_2 = 1$ and $2y\langle x, w_{\text{old}}\rangle$ is negative, we have $||w_{\text{new}}||^2 \leq ||w_{\text{old}}||^2 + 1$.

6.5 1.6.3 Polynomial Threshold Functions

Polynomial threshold functions (PTFs) are of the form:

$$f = \text{sign}(p(x))$$

where $p(x)$ is a multivariate polynomial of degree d. Learning PTFs of degree d is equivalent to learning halfspaces in n^d dimensions.

6.6 1.6.4 Kernel Functions

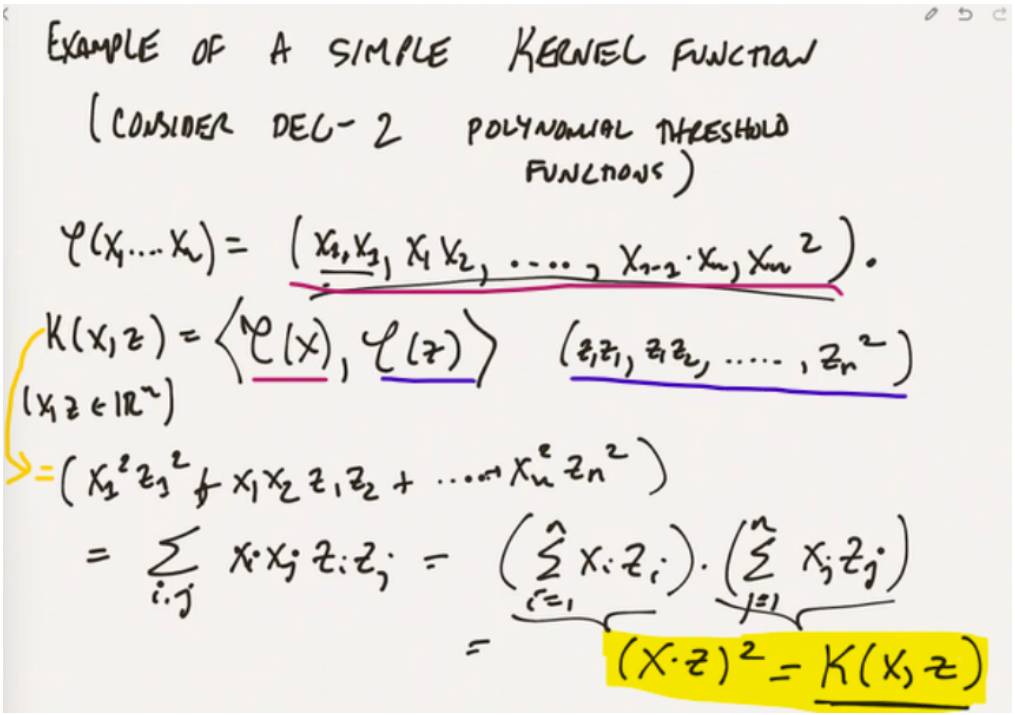
The **Kernel Trick** can be used to save on runtime. The kernel function $K(x^1, x^2)$ outputs the inner product of $\phi(x^1)$ and $\phi(x^2)$ in feature space. This reduces the computational complexity when dealing with high-dimensional spaces.

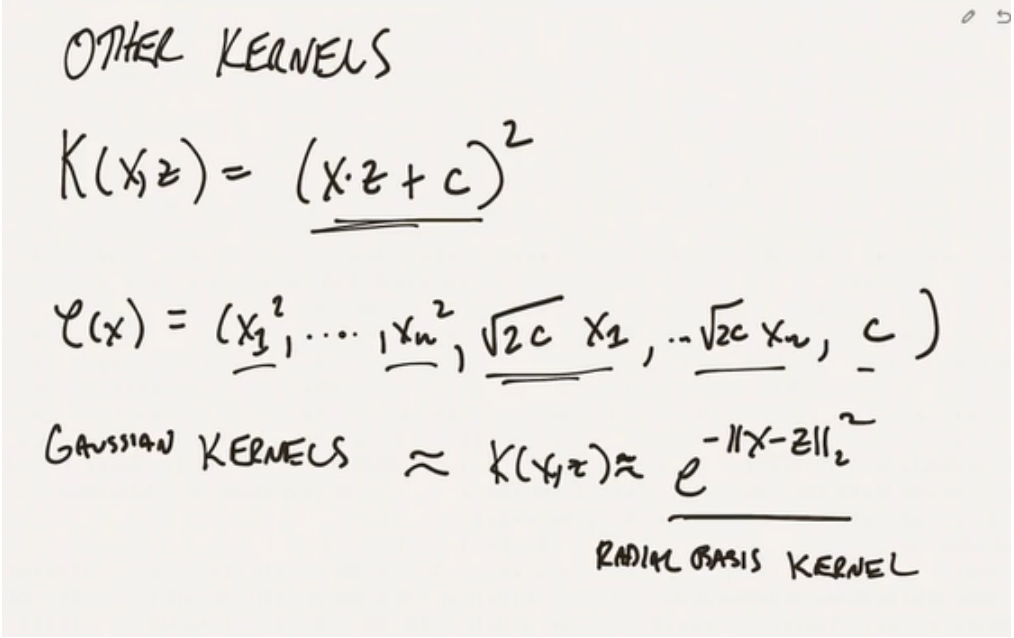
Kernel Perceptron: Given that:

$$w_{\text{new}} = \sum_{i=1}^{t} y^i \cdot \phi(x^i)$$

we compute $w_{\text{new}} \cdot \phi(x^{t+1})$ efficiently using the kernel function.

6.7 1.6.5 Example Kernel Functions





6.8 Reading

For further reference, see:

- Understanding Machine Learning: From Theory to Algorithms, Chapter 9.

7 Linear Regression

7.1 Vocabulary

- Second derivative test
- Variance
- Covariance
- Span
- Projection
- Transpose
- Normal equations
- Pseudo-inverse
- Likelihood
- Derivative rules
- PDF of the Gaussian

7.2 Lecture Notes

7.3 Introduction

7.3.1 Background

Linear regression is a core task in multiple fields, including statistics, computer science, and machine learning. It involves fitting a line to data.

Unlike classification, the labels in regression are real-valued, not just binary (0 or 1).

7.3.2 Deriving the Regression Function

Let X and Y be two random variables, where Y is the label to be predicted.

- Without X: The optimal guess for Y is $\mathbb{E}[Y]$.
- With X: The optimal prediction for Y becomes $\mathbb{E}[Y|X]$.

The function $\mathbb{E}[Y|X]$ is the **regression function** $f(X)$.
Linear Regression Question: Given X, what linear function of X should be used to predict Y?
The goal is to minimize the square loss:

$$\mathbb{E}_{(X,Y)\sim\mathbf{D}}[(Y-(\beta_0+\beta_1 X))^2]$$

7.4 Finding the Betas

7.4.1 Minimization Function

We are given a training set of size m: $(x^1,y^1),\ldots,(x^m,y^m)$. The goal is to minimize:

$$\min_{\beta_0,\beta_1}\frac{1}{m}\sum_{j=1}^{m}(y^j-(\beta_0+\beta_1 x^j))^2$$

7.4.2 How to Find β_0 and β_1

We compute the derivatives with respect to β_0 and β_1, then set them to 0. The loss function $\updownarrow$ is:

$$\updownarrow=\sum_{j=1}^{m}(y^j-(\beta_0+\beta_1 x^j))^2$$

The partial derivatives are:

$$\frac{\partial\updownarrow}{\partial\beta_0}=\frac{1}{m}\sum_{j=1}^{m}(y^j-\beta_0-\beta_1 x^j)(-2)$$

$$\frac{\partial\updownarrow}{\partial\beta_1}=\frac{1}{m}\sum_{j=1}^{m}(y^j-\beta_0-\beta_1 x^j)(-2x^j)$$

Solving for β_0 and β_1:

$$\beta_0=\bar{y}-\beta_1\bar{x}$$
$$\beta_1=\frac{\bar{xy}-\bar{x}\cdot\bar{y}}{\bar{x^2}-(\bar{x})^2}$$

7.4.3 Beta 1 and Variance of X

The denominator of β_1 resembles the variance of x, and the numerator resembles the covariance of x and y. Thus:

$$\beta_1=\frac{Cov(X,Y)}{Var(X)}$$

7.5 Regression with Multiple Variables

7.5.1 Defining the Problem

Instead of assuming X is scalar, we now assume X is an n-dimensional vector, while Y remains scalar:

$$X\in\mathbb{R}^n,\ y\in\mathbb{R}$$

Consider a matrix $X\in\mathbb{R}^{m\times n}$, with m rows (each corresponding to an n-dimensional data point) and n columns (features).
The goal is to find a vector $w\in\mathbb{R}^n$ that minimizes:

$$\min_w||Xw-y||_2^2$$

7.5.2 How to Find w

We project y onto the span of the columns of X. The optimal w is found by solving the normal equations:

$$X^T(y - Xw) = 0$$
$$X^T y = X^T X w$$
$$w = (X^T X)^{-1} X^T y$$

Normal Equations: $(X^T X)^{-1} X^T y = w$.

7.6 Maximum Likelihood

7.6.1 Assumption

We assume $y = \beta_0 + \beta_1 x + \epsilon$, where $\epsilon \sim N(0, \sigma^2)$ (Gaussian noise).

7.6.2 Likelihood Function

The likelihood function is the probability of observing the training set $(x^1, y^1), \ldots, (x^m, y^m)$ given β_0 and β_1:

$$L(\beta_0, \beta_1) = \prod_{i=1}^{m} P(y^i | x^i; \beta_0, \beta_1)$$

With the Gaussian PDF:

$$L(\beta_0, \beta_1) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y^i - (\beta_0 + \beta_1 x^i))^2}{2\sigma^2}}$$

7.6.3 Maximizing Likelihood

We maximize the log-likelihood instead:

$$\log L(\beta_0, \beta_1) = -\frac{m}{2} \log 2\pi - m \log \sigma - \frac{1}{2\sigma^2} \sum_{i=1}^{m} (y^i - (\beta_0 + \beta_1 x^i))^2$$

The last term corresponds to the least-squares estimate from linear regression.

7.6.4 Interpreting Coefficients

The coefficients have two interpretations:

- **Geometric:** Coefficients minimize squared distance to the labels.
- **Statistical:** Coefficients maximize the likelihood for a dataset generated with noise $\epsilon \sim N(0, \sigma^2)$.

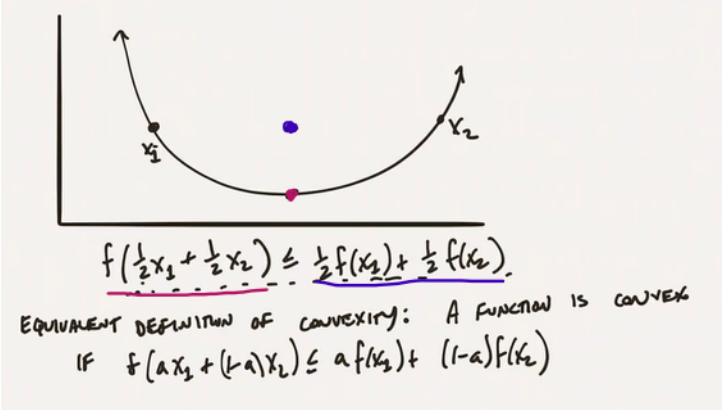
7.7 Personal Notes

Understanding Machine Learning: From Theory to Algorithms, Chapter 9

8 1.8 Gradient Descent

Vocabulary & Code

Convexity: A function is convex if the chord connecting any two points on the graph lies above the function.

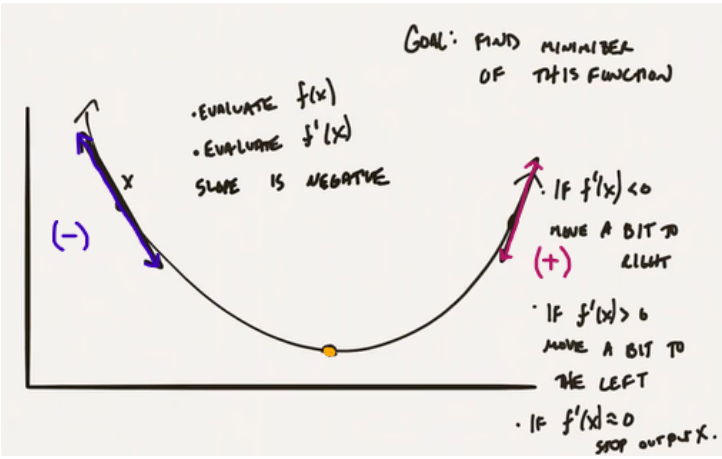


1.8.0 Introduction

The goal is to find the minimizer of a function. We can evaluate $f(x)$ at some point x, and also evaluate its derivative, $f'(x)$, to find the slope of the tangent line at that point.

Update Rule:

- If $f'(x) < 0$, move a bit to the right.
- If $f'(x) > 0$, move a bit to the left.
- If $f'(x) = 0$, or close to 0, stop and output x.



For more complex functions, applying the same update rule can help find the global minimum, assuming the function is convex. In convex functions, gradient descent will always reach the global minimum.

For linear functions in d dimensions, such as $f(x) = w^T x + b$, we can use a unit vector to determine the direction to move in, with the correct choice of unit vector being $\frac{-w}{||w||}$. Moving in this direction will decrease f by $||w||_2$.

1.8.1 Putting it Together

We can apply these concepts to more complex functions. The Taylor expansion of $f(x)$ around a point x is:

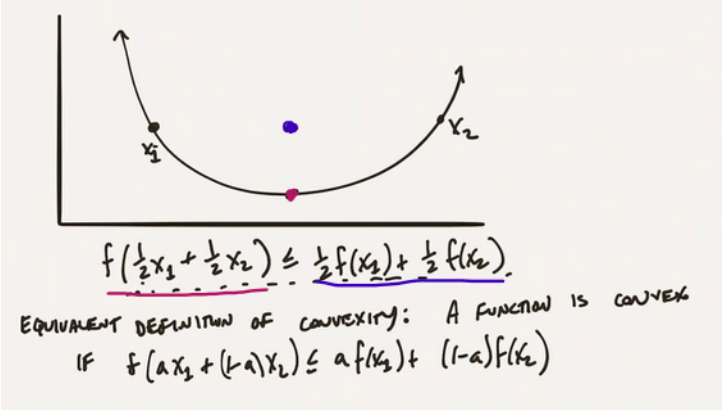
$$f(x+\epsilon) = f(x) + \epsilon f'(x) + \frac{\epsilon^2}{2!}f''(x) + \frac{\epsilon^3}{3!}f'''(x) + \dots$$

For small ϵ, the linear term dominates, implying that locally, $f(x)$ behaves like a linear function.

In higher dimensions, we use the gradient:

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_d}(x)\right)$$

For example, for $f = w^T x + b$, the gradient is $\nabla f = w$.



1.8.2 Defining and Applying

To minimize $f(w)$, we initially choose w randomly. If $||\nabla f(w)||_2 < \epsilon$, we stop and output w. Otherwise:

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla f(w_{\text{old}})$$

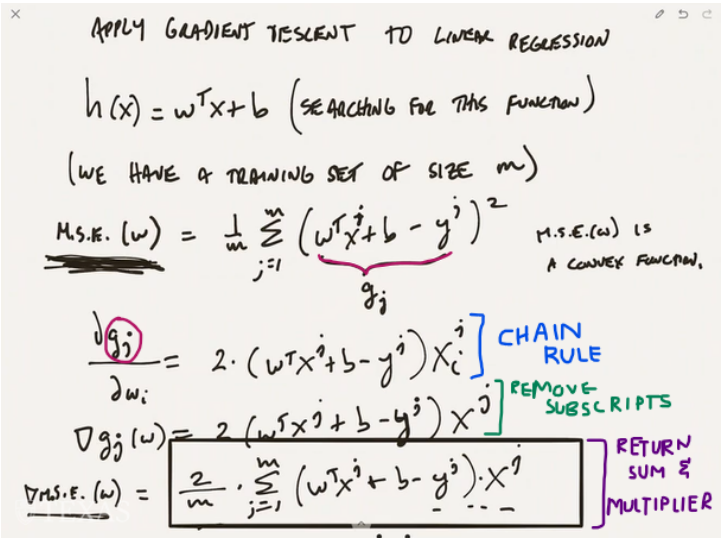
where η is the step size.

Applying Gradient Descent to Linear Regression: For linear regression, we aim to minimize the mean squared error (MSE):

$$MSE(w) = \frac{1}{m}\sum_{j=1}^{m}(w^T x^j + b - y^j)^2$$

We can apply gradient descent to minimize MSE with the update rule:

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla MSE(w)$$



Stochastic Gradient Descent (SGD): In SGD, we compute the gradient using a single point j at random:

$$w_{\text{new}} = w_{\text{old}} - 2\eta(w^T x^j + b - y^j)x^j$$

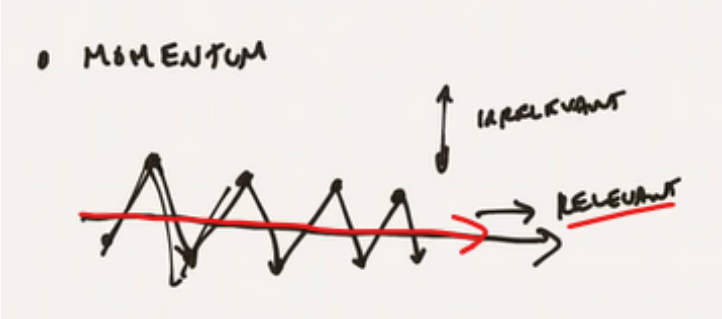
In expectation, this update rule approximates the full gradient. Batches can also be used to interpolate between full gradient descent and SGD.

1.8.3 Choosing Step Size

Choosing η, the step size, is more art than science. Cross-validation is commonly used to pick η. Techniques like **momentum** are also helpful:

$$V_i = \alpha V_{i-1} - \eta g_i$$

where g_i is the gradient at step i. The update rule becomes $w_{\text{new}} = w_{\text{old}} + V_i$.



Personal Notes

Understanding Machine Learning: From Theory to Algorithms, Chapter 14

Summary of the Gradient Descent Process:

1. If $||\nabla f(w)||_2 < \epsilon$, stop and output w.
2. Otherwise, update: $w_{\text{new}} = w_{\text{old}} - \eta \nabla f(w_{\text{old}})$.

Generalized Gradient Descent (GD) Update Rule:

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla L(w_{\text{old}})$$

Generalized Stochastic Gradient Descent (SGD) Update Rule:

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla L_i(w_{\text{old}})$$

where L_i is the loss function computed for a single data point i.

Key Differences Between GD and SGD:

- GD computes the full gradient, while SGD computes a noisy estimate using a random data point.
- GD converges smoothly, while SGD has more variability but can avoid local minima.

Summary of Update Rules:

- GD: $w_{\text{new}} = w_{\text{old}} - \eta \nabla L(w_{\text{old}})$
- SGD: $w_{\text{new}} = w_{\text{old}} - \eta \nabla L_i(w_{\text{old}})$

9 1.9 Boosting

Vocabulary & Code

1.9.0 Introduction

Recall two key parameters from PAC learning:

- δ: probability of failure
- ϵ: accuracy parameter

Requirement: For any choice of ϵ and δ, A should output, with probability $\geq 1 - \delta$, an ϵ-accurate classifier. The algorithm A is allowed to run in time $poly\left(\frac{1}{\epsilon}, \frac{1}{\delta}\right)$ and take a number of samples $poly\left(\frac{1}{\epsilon}, \frac{1}{\delta}\right)$.

—

Question: What if we have an algorithm, A, that with probability 5% outputs an ϵ-accurate classifier? How can we use A to obtain a standard PAC learner?

We want to increase that 5% probability of success to $1 - \delta$. By running A a large number of times, say t, we have:

$$Pr\left[A \text{ fails to output an } \epsilon \text{ accurate classifier}\right] \leq (0.95)^t$$

Choosing $t = O\left(\log \frac{1}{\delta}\right)$ makes this probability very small, allowing us to test each classifier over t trials to find a good classifier.

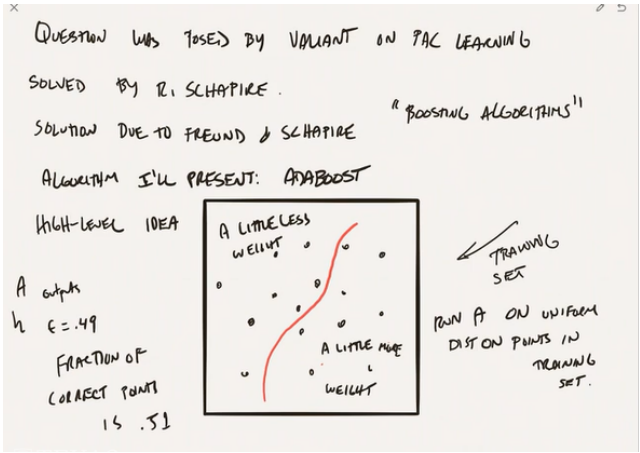
Summary: Running A t times results in the probability of failure being at most $(0.95)^t$, so by choosing $t = O\left(\log \frac{1}{\delta}\right)$, the probability of not outputting an accurate hypothesis over these t trials is smaller than δ. Therefore, with reasonable probability, one of the classifiers will be ϵ-accurate.

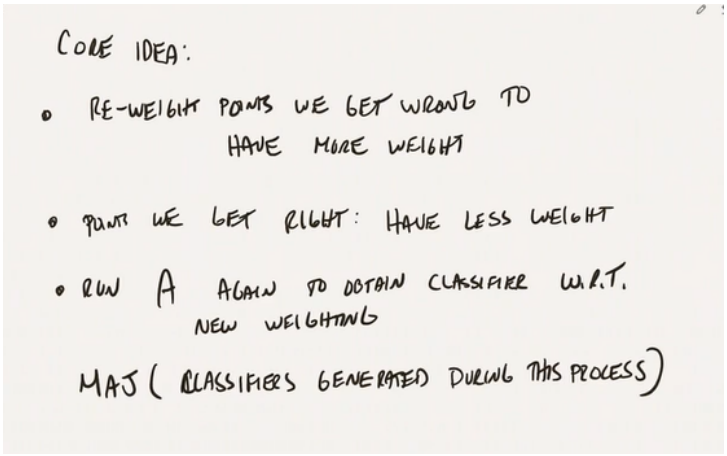
—

Trickier Question: What if $\epsilon = 0.49$?
Consider A with probability $\geq 1 - \delta$ outputs a classifier with $\epsilon = 0.49$. How can we improve the accuracy parameter?
Solution: Boosting algorithms.

—

Adaboost Overview:





1.9.1 Adaboost

Simplified Adaboost: Assume a training set of size m.

Initially, the first distribution D_0 is uniform, corresponding to $w_i = 1$ for all i. The distribution is obtained by dividing by W, the sum of the weights.

- E: error rate
- A: accuracy $= 1 - E$
- $\beta = \frac{E}{A}$

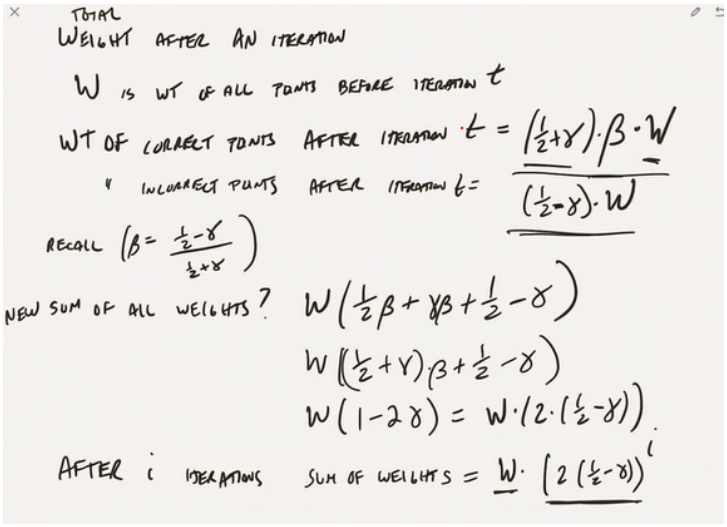
Concretely: $E = \frac{1}{2} - \gamma$, and $\beta = \frac{\frac{1}{2}-\gamma}{\frac{1}{2}+\gamma}$.
Weight Update: At iteration t, run A to obtain h_t. For each x_i:

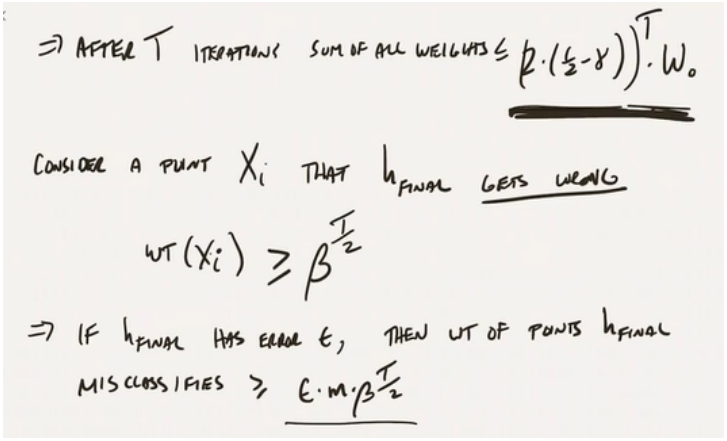
- If $h_t(x_i)$ is correct: $w_i^{new} = \beta w_i^{old}$
- If $h_t(x_i)$ is incorrect: $w_i^{new} = w_i^{old}$

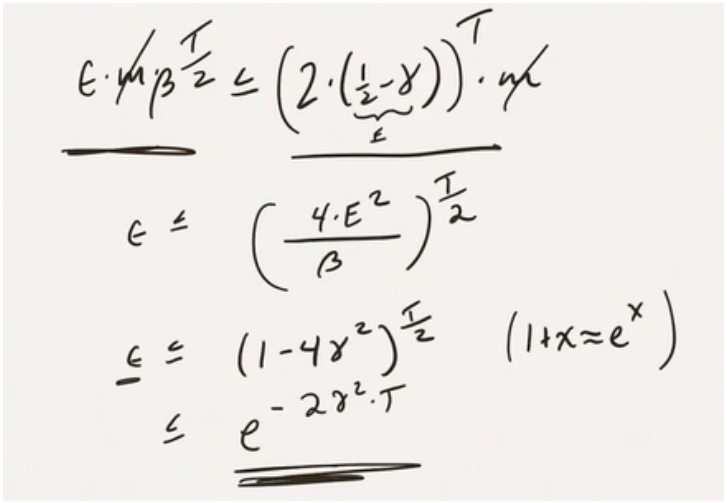
Repeat for T steps and output $\text{maj}(h_1, h_2, \ldots, h_T)$.

—

Claim: After T iterations, the error of the final classifier $h_{final} = \text{maj}(h_1, h_2, \ldots, h_T) \leq e^{-2T\gamma^2}$. Choosing $T \approx \frac{1}{\gamma^2} \log\left(\frac{1}{\epsilon}\right)$ results in h_{final} error $\leq \epsilon$.







1.9.2 Adaboost Modifications

Adaboost can be modified such that, if a very good classifier is found, it takes advantage of that, requiring fewer iterations.

IN ADABOOST

$h_t \in \{-1, +1\}$

$$\beta_t = \frac{E_t}{A_t}$$

OUTPUT: $\text{SIGN}\left(\sum_t \alpha_t h_t - \frac{1}{2}\right)$ $\quad \alpha_t = \dfrac{\log\left(\frac{1}{\beta_t}\right)}{\sum_t \log\left(\frac{1}{\beta_t}\right)}$

HOW DO WE GUARANTEE THAT H_{FINAL} GENERALIZES?

WE NEED TO MAKE SURE THAT MY # OF TRAINING POINTS IS SUFFICIENTLY LARGE.

IF γ (ACCURACY) IS INDEPENDENT FROM m, SIZE OF TRAINING SET THEN WE CAN CHOOSE m TO BE SUFFICIENTLY LARGE.

ADABOOST IS ACTUALLY A SPECIAL CASE OF AN ALGORITHM DUE TO FREUND AND SCHAPIRE CALLED "HEDGE."

HEDGE: "BEST EXPERTS" SETUP.

$c_1, \ldots, c_m$ AT EACH ITERATION t EXPERT c_i SUFFERS A LOSS $\ell_i^t \in [0,1]$

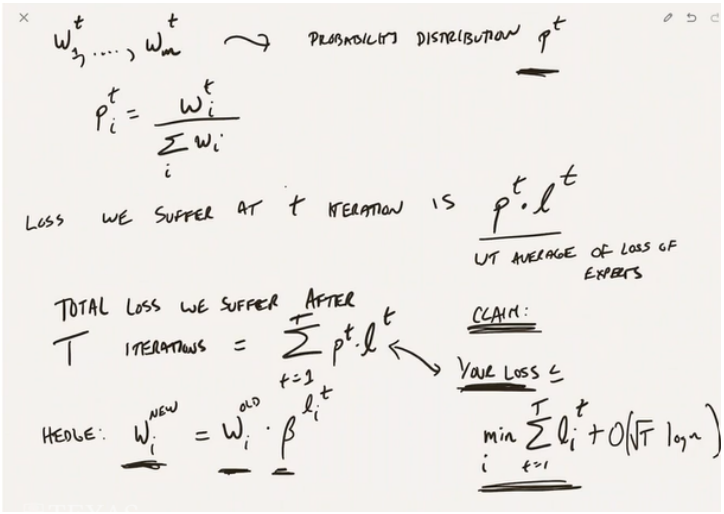
ℓ^t A VECTOR OF LOSSES SUFFERED BY ALL EXPERTS AT t^{th} ITERATION.

INTUITION: WE WANT TO HAVE A MIXED STRATEGY OF EXPERTS WEIGHTED AVG OF "

GOAL: SUM OF OUR LOSSES AFTER T ITERATIONS SHOULD BE "CLOSE" TO BEST EXPERT IN HINDSIGHT.

AT EACH iteration WE MAINTAIN A SET OF WEIGHTS

$w_1, \ldots, w_m$ WEIGHTED AVERAGE $= \dfrac{w_i}{\sum_i w_i} = p_i$



Personal Notes

Understanding Machine Learning: From Theory to Algorithms, Chapter 10

10 1.9 Boosting

Vocabulary & Code

1.9.0 Introduction

Recall two key parameters from PAC learning:

- δ: probability of failure
- ϵ: accuracy parameter

Requirement: For any choice of ϵ and δ, A should output, with probability $\geq 1-\delta$, an ϵ-accurate classifier. The algorithm A is allowed to run in time $poly\left(\frac{1}{\epsilon}, \frac{1}{\delta}\right)$ and take a number of samples $poly\left(\frac{1}{\epsilon}, \frac{1}{\delta}\right)$.

—

Question: What if we have an algorithm, A, that with probability 5% outputs an ϵ-accurate classifier? How can we use A to obtain a standard PAC learner?

We want to increase that 5% probability of success to $1-\delta$. By running A a large number of times, say t, we have:

$$Pr\left[A \text{ fails to output an } \epsilon \text{ accurate classifier}\right] \leq (0.95)^t$$

Choosing $t = O\left(\log \frac{1}{\delta}\right)$ makes this probability very small, allowing us to test each classifier over t trials to find a good classifier.

Summary: Running A t times results in the probability of failure being at most $(0.95)^t$, so by choosing $t = O\left(\log \frac{1}{\delta}\right)$, the probability of not outputting an accurate hypothesis over these t trials is smaller than δ. Therefore, with reasonable probability, one of the classifiers will be ϵ-accurate.

—

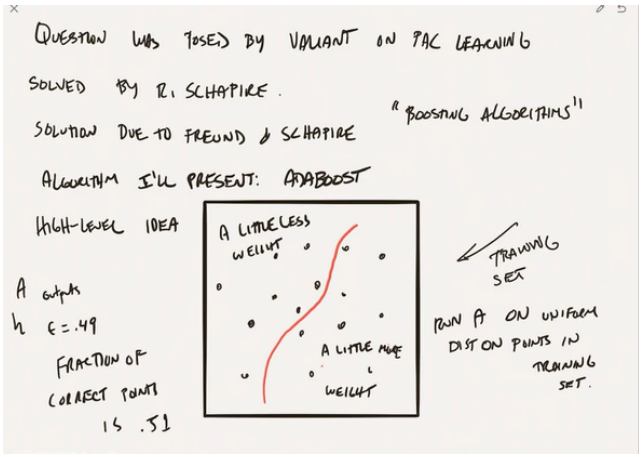
Trickier Question: What if $\epsilon = 0.49$?

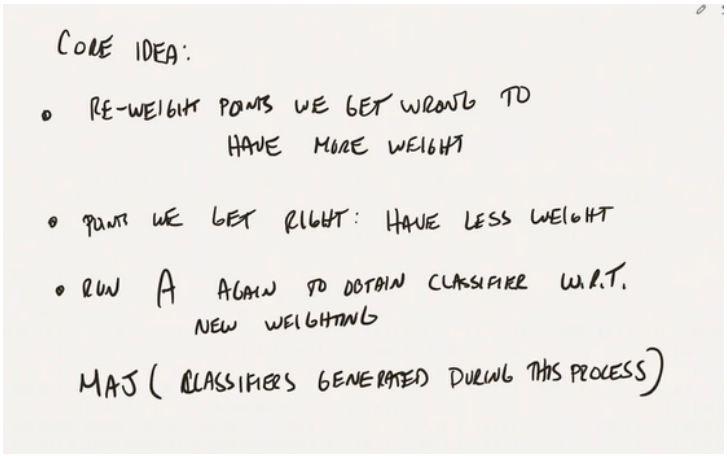
Consider A with probability $\geq 1-\delta$ outputs a classifier with $\epsilon = 0.49$. How can we improve the accuracy parameter?

Solution: Boosting algorithms.

—

Adaboost Overview:





1.9.1 Adaboost

Simplified Adaboost: Assume a training set of size m.

Initially, the first distribution D_0 is uniform, corresponding to $w_i = 1$ for all i. The distribution is obtained by dividing by W, the sum of the weights.

- E: error rate
- A: accuracy $= 1 - E$
- $\beta = \frac{E}{A}$

Concretely: $E = \frac{1}{2} - \gamma$, and $\beta = \frac{\frac{1}{2}-\gamma}{\frac{1}{2}+\gamma}$.

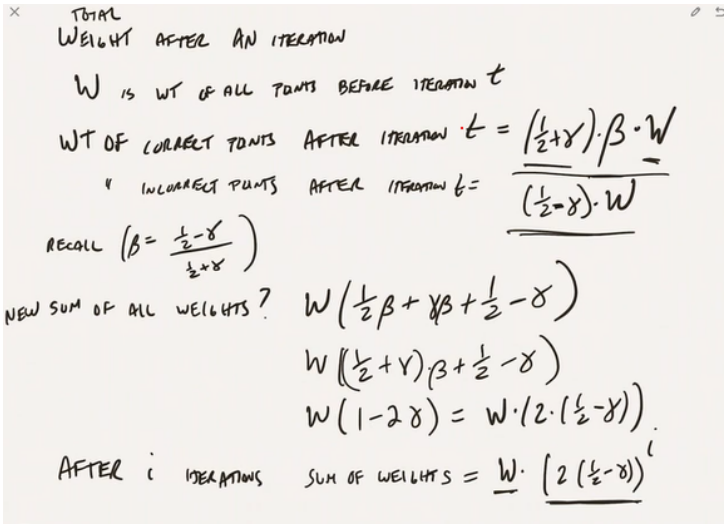
Weight Update: At iteration t, run A to obtain h_t. For each x_i:

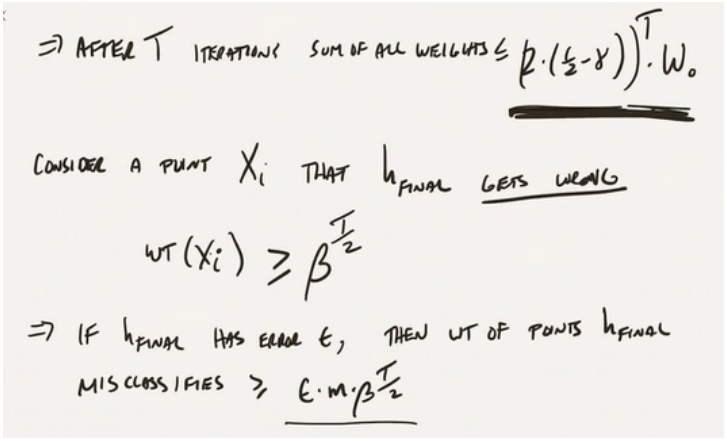
- If $h_t(x_i)$ is correct: $w_i^{new} = \beta w_i^{old}$
- If $h_t(x_i)$ is incorrect: $w_i^{new} = w_i^{old}$

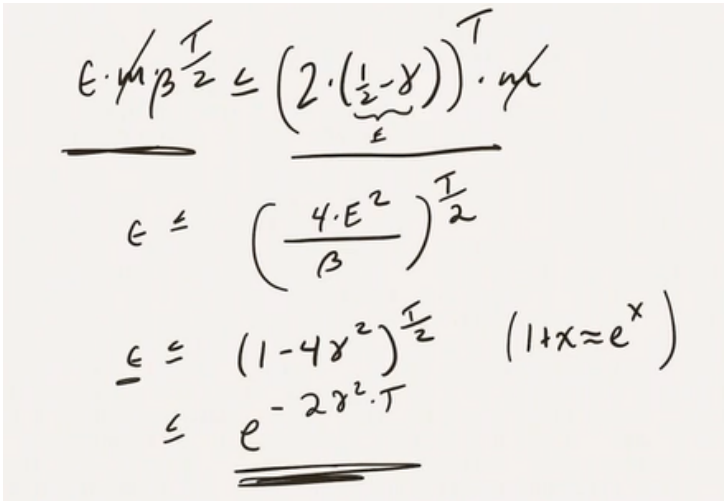
Repeat for T steps and output $\text{maj}(h_1, h_2, \ldots, h_T)$.

—

Claim: After T iterations, the error of the final classifier $h_{final} = \text{maj}(h_1, h_2, \ldots, h_T) \leq e^{-2T\gamma^2}$. Choosing $T \approx \frac{1}{\gamma^2} \log\left(\frac{1}{\epsilon}\right)$ results in h_{final} error $\leq \epsilon$.







1.9.2 Adaboost Modifications

Adaboost can be modified such that, if a very good classifier is found, it takes advantage of that, requiring fewer iterations.

IN ADABOOST

$h_t \in \{-1, +1\}$

$$\beta_t = \frac{E_t}{A_t}$$

OUTPUT: $\text{SIGN}\left(\sum_t \alpha_t h_t - \frac{1}{2}\right)$ $\quad \alpha_t = \dfrac{\log\left(\frac{1}{\beta_t}\right)}{\sum_t \log\left(\frac{1}{\beta_t}\right)}$

HOW DO WE GUARANTEE THAT H_{FINAL} GENERALIZES?

WE NEED TO MAKE SURE THAT MY # OF TRAINING POINTS IS SUFFICIENTLY LARGE.

IF γ (ACCURACY) IS INDEPENDENT FROM m, SIZE OF TRAINING SET THEN WE CAN CHOOSE m TO BE SUFFICIENTLY LARGE.

ADABOOST IS ACTUALLY A SPECIAL CASE OF AN ALGORITHM DUE TO FREUND AND SCHAPIRE CALLED "HEDGE."

HEDGE: "BEST EXPERTS" SETUP.

$c_1, \ldots, c_m$ AT EACH ITERATION t EXPERT c_i SUFFERS A LOSS $\ell_i^t \in [0,1]$

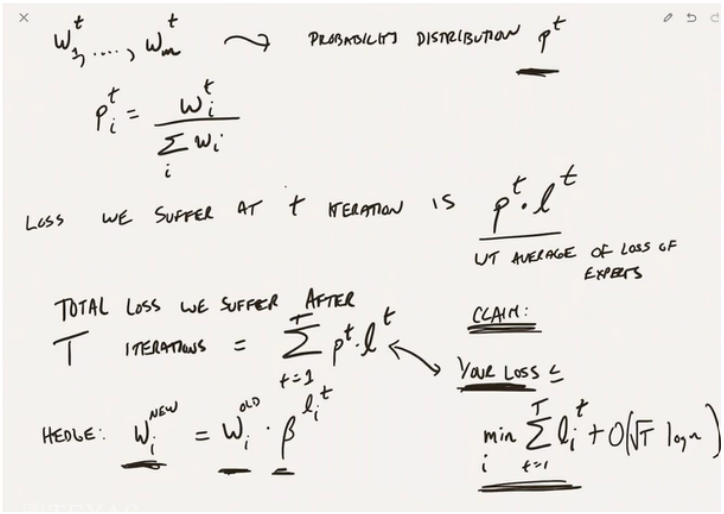
ℓ^t A VECTOR OF LOSSES SUFFERED BY ALL EXPERTS AT t^{th} ITERATION.

INTUITION: WE WANT TO HAVE A MIXED STRATEGY OF EXPERTS WEIGHTED AVG OF "

GOAL: SUM OF OUR LOSSES AFTER T ITERATIONS SHOULD BE "CLOSE" TO BEST EXPERT IN HINDSIGHT.

AT EACH iteration WE MAINTAIN A SET OF WEIGHTS

$w_1, \ldots, w_m$ $\quad$ WEIGHTED AVERAGE $= \dfrac{w_i}{\sum_i w_i} = p_i$



Personal Notes

Understanding Machine Learning: From Theory to Algorithms, Chapter 10

11 Logistic Regression

Vocabulary

Margin

$$yx^T\theta$$

Identifies how far the data point is from the decision boundary, considering the true label.

Confidence

$$h_\theta(x) = x^T\theta$$

Simply how confident the model is about the prediction, does not account for the true label.

Loss Function

$$\phi(y, \hat{y})$$

where y is the true label and $\hat{y}$ is the predicted value. A loss function quantifies the penalty for making incorrect predictions on a single example. It measures how far off a prediction is from the true label and is used to guide the learning process by providing feedback on individual predictions.

Risk Function

$$R(\theta) = \mathbf{E}[\phi(y, \hat{y})]$$

The risk function (or expected loss) is the average of the loss function over the entire distribution of data, representing the expected penalty for the model's predictions.

Likelihood Quantifies how well a model explains the data, while log-likelihood transforms this measure for easier manipulation and optimization. Both are fundamental in statistical modeling and inference, inversely correlated to risk.

1.10.0 Introduction

Loss Functions

- Classification:
  - Loss function: we predict $sign(w^T x)$, penalized if $sign(w^T x) \neq y^i$.
  - Penalty can be 0-1 loss:

$$\phi_{0-1}(z) = \begin{cases} 1 & \text{if } z \leq 0 \\ 0 & \text{if } z > 0 \end{cases}$$

- Linear Regression:
  - Loss function (square loss): $\min_w \frac{1}{m} \sum_{i=1}^{m} (w^T x^i - y^i)^2$

Optimization Problems

- Classification:

$$\min_w \frac{1}{m} \sum_{i=1}^{m} \phi_{0-1}(y^i w^T x^i)$$

—

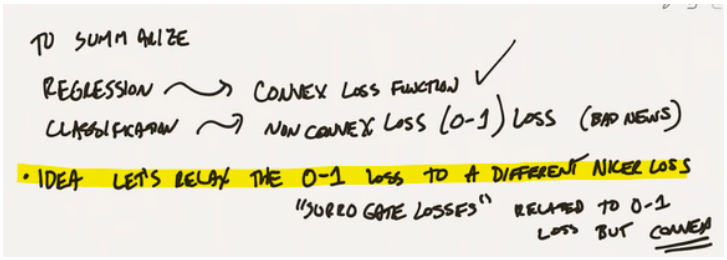
Question: When does the perceptron find a w with small loss?
Perceptron requires w such that $\forall x : yw^T x > \rho$, ensuring convergence with at most $\frac{1}{\rho^2}$ mistakes.

—

What if there is no margin? We aim to minimize mistakes on the training set:

$$\min_w \frac{1}{m} \sum_{i=1}^{m} \phi_{0-1}(y^i w^T x^i)$$

This optimization problem is NP-hard, unlike linear regression. Thus, we use a more relaxed loss function: logistic loss, which is convex and differentiable.



1.10.1 Losses

New Losses

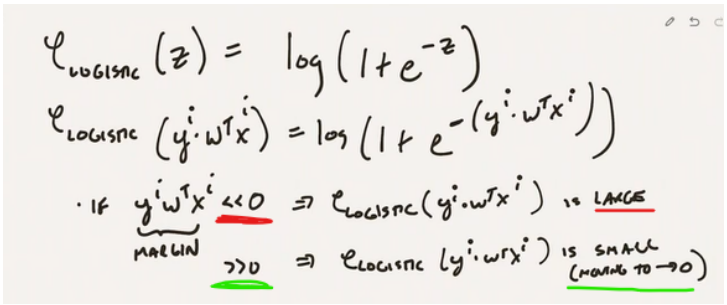
- $\phi_{logistic}$
- ϕ_{hinge}
- ϕ_{exp}

—

Logistic Loss Function

$$\phi_{logistic}(z) = \log(1 + e^{-z})$$

If $y^i w^T x^i$ (the margin) is $\ll 0$, $\phi_{logistic}(y^i w^T x^i)$ becomes large (incorrect guess). If the guess is correct, the loss is small.

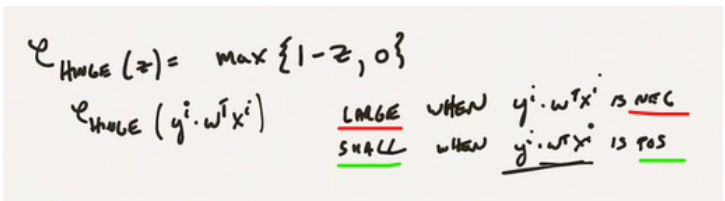


—

Hinge Loss Function

$$\phi_{hinge}(z) = \max\{1 - z, 0\}$$

If prediction is correct with margin 1, there is no loss. For incorrect predictions, there is a loss.



Exponential Loss Function

$$\phi_{exp}(z) = e^{-z}$$

Visualizing the Losses

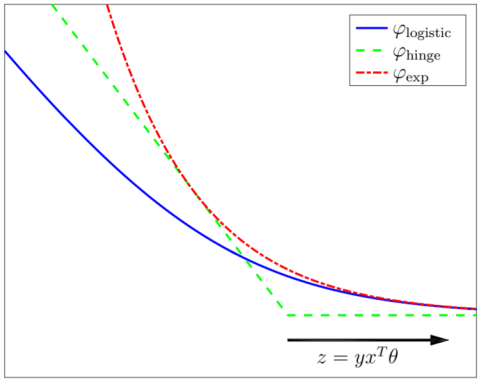


Figure 2: The three margin-based loss functions logistic loss, hinge loss, and exponential loss.

All losses are convex and differentiable. We will focus on logistic loss.

1.10.2 Logistic Loss Optimization

Optimization Problem:

$$L(w) = \frac{1}{m}\sum_{i=1}^{m} \log(1 + \exp(-y^i w^T x^i))$$

We want to find $\min_w L(w)$. Enter the sigmoid function:

$$g(z) = \frac{1}{1 + e^{-z}}$$

As $z \to \infty$, $g(z) \to 1$, and as $z \to -\infty$, $g(z) \to 0$.

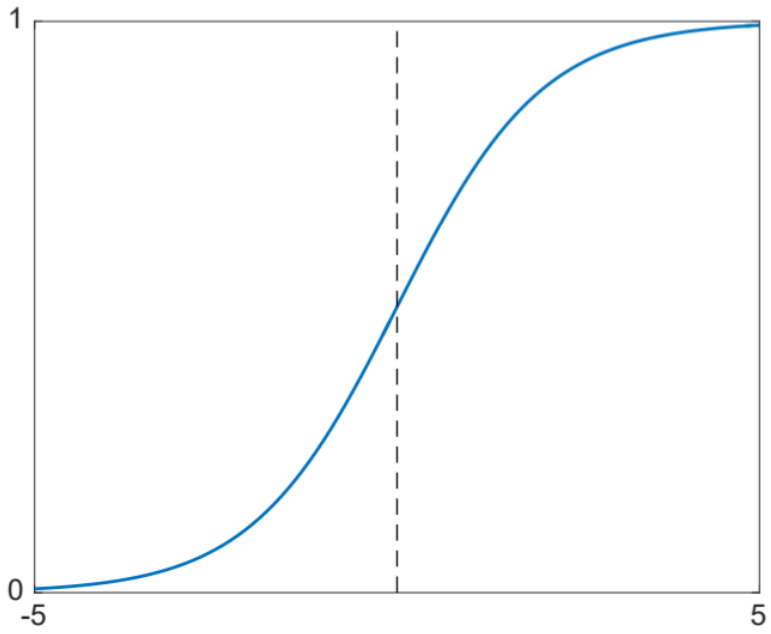


Figure 3: Sigmoid function

Properties of Sigmoid

$$g(z) + g(-z) = 1$$

There exists a w such that:

$$\mathbf{E}[Y|X] = g(Yw^T X)$$

Thus, $Pr[Y = 1|X] = g(w^T x)$, implying logistic regression models the probability of y given x.

1.10.3 Minimizing Logistic Loss

Goal: Minimizing $L(w)$

Since logistic loss is convex and differentiable, we can use gradient descent on logistic loss, which is logistic regression.

Gradient of Logistic Loss

$$\phi'_{logistic}(z) = \frac{-e^{-z}}{1 + e^{-z}} = \frac{1}{1 + e^{z}} = -g(-z)$$

The gradient for the entire dataset is:

$$\nabla L(w) = \frac{1}{m} \sum_{i=1}^{m} (y^i g(-y^i w^T x^i)) x^i$$

This allows us to apply gradient descent to find the maximum likelihood w.

—

Multinomial Logistic Regression

For multiple labels $y \in \{1, \ldots, k\}$:

$$Pr[y = j|x] \propto e^{{w^j}^T x}$$

The associated loss is cross-entropy loss, a generalization of logistic loss.

Softmax

$$(z_1, \ldots, z_k) \mapsto \left(\frac{e^{z_1}}{z}, \frac{e^{z_2}}{z}, \ldots, \frac{e^{z_k}}{z} \right)$$

Softmax maps real values to probabilities, normalizing the vector so values sum to 1.

Personal Notes

Loss Functions

12 Principal Component Analysis

Vocabulary

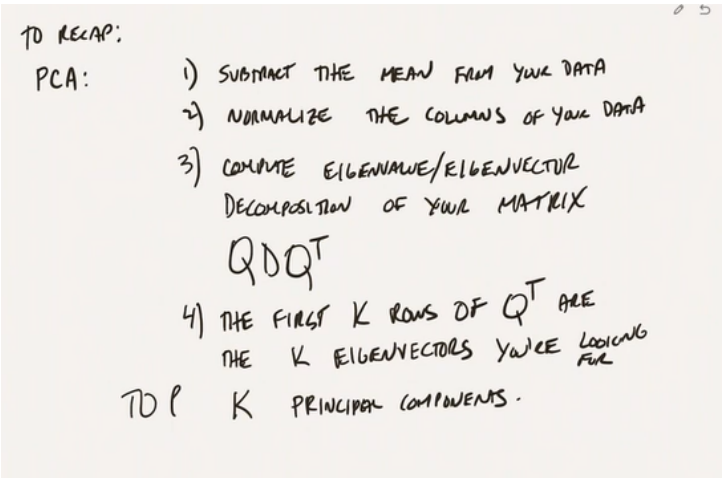
Covariance Matrix

A square matrix that provides the covariances between pairs of features in the dataset.

1.11.0 Introduction

Principal Component Analysis (PCA) is one of the most important techniques for dimensionality reduction. It transforms data into a new coordinate system by finding directions (called principal components) that capture the maximum variance in the data.

Preview of PCA Process:



Goal of PCA: The goal of PCA is to find vectors $v_1, \ldots, v_k$ such that:

$$\forall x \in S, \;\; x \approx \sum_{j=1}^{k} a_j v_j$$

This gives a new representation of x in terms of the principal components.

Data Preprocessing:

- Subtract the mean from each data point (center the data so mean is 0).
- Normalize by the standard deviation of each feature.
- This ensures each feature has mean 0 and standard deviation 1.

Initial Objective: Find v_1, a vector that minimizes the square-distance between the data points and the vector:

$$\min_{v,\, ||v||_2=1} \;\; \frac{1}{m} \sum_{j=1}^{m} (\textit{distance between } x^j \textit{ and } v)^2$$

The goal is to find a direction such that projecting all points onto this direction minimizes the sum of squared distances.

1.11.1 Explaining PCA

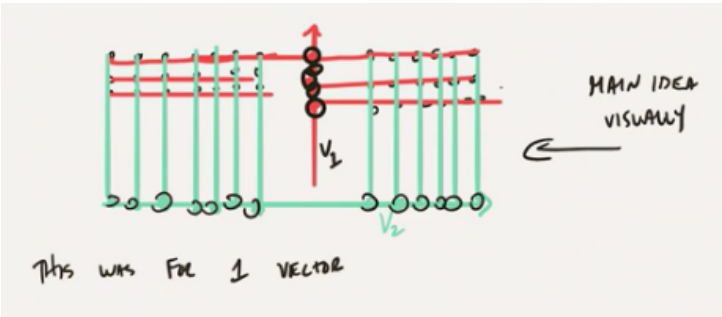
Objective Function:

$$\min_{v,\, ||v||_2=1} \;\; \frac{1}{m} \sum_{j=1}^{m} (\textit{distance between } x^j \textit{ and } v)^2$$

Using the Pythagorean theorem, we can restate the problem to maximize the variance in the direction v:

$$\max_{v,\, ||v||_2=1} \;\; \frac{1}{m} \sum_{j=1}^{m} \langle x^j, v \rangle^2$$

This corresponds to finding the direction of maximal variance in the dataset.



Multiple Components:
For k components, the optimization problem becomes:

$$\underset{S \text{ of dimension } k}{max \text{ subspaces}} \frac{1}{m}\sum_{j=1}^{m}(length \text{ of } x^j \text{ projected onto } S)^2$$

We can solve this problem using an orthonormal basis $v_1, \ldots, v_k$. The goal is to maximize the sum of squared projections onto these vectors.

PCA Objective:

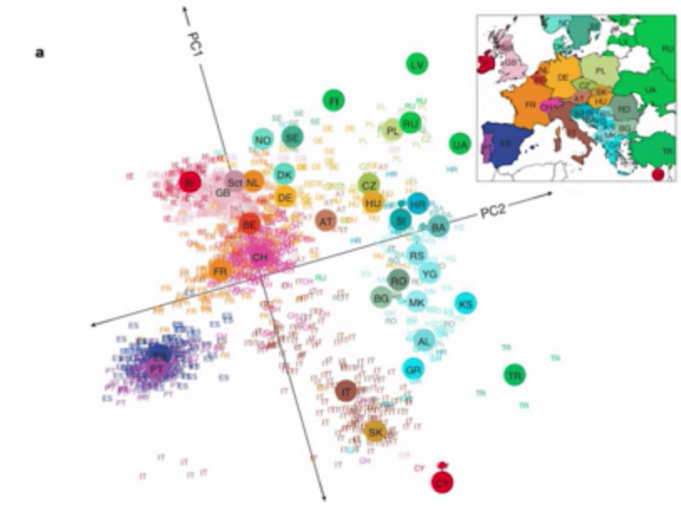
$$\max_{v_1,\ldots,v_k \text{ ; orthogonal}} \frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k}\langle x^j, v_i\rangle^2$$

The optimal solution is to project data points onto the k principal components that retain the most variance.

1.11.2 Applications

1. Understanding Genomes - "Genes Mirror Geography Within Europe"

- PCA was applied to genetic data from 1,400 Europeans, each represented by 200,000 genetic markers.
- After running PCA, the first two components were plotted, and the result closely mirrored a map of Europe.



2. Image Data Compression - "Eigenfaces"

- Each image is represented as a vector of 65,000 pixels.
- PCA was used to reduce the dimensionality to 100-150 features.
- Below is an original image next to its compressed version:

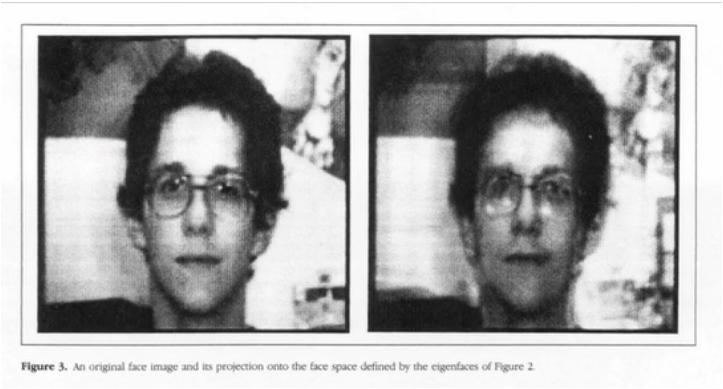


Figure 3. An original face image and its projection onto the face space defined by the eigenfaces of Figure 2.

1.11.3 The Big Question

How do we find the vectors v_1 through v_k?

Recall the optimization problem:

$$\max_{v_1,\ldots,v_k \text{ ; orthonormal}} \frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k}\langle x^j, v_i\rangle^2$$

Covariance Matrix X^TX: We need to compute X^TX, an $n \times n$ matrix. This is the sample covariance matrix of the dataset. The goal is to find the eigenvectors and eigenvalues of this matrix to compute the principal components.

Spectral Theorem:
For any symmetric matrix A, there exists an eigendecomposition $A = QDQ^T$, where Q is orthogonal and D is diagonal.

PCA Objective Restated: The PCA objective can be restated as:

$$\max_{v,\ ||v||_2=1} v^TAv$$

This is equivalent to finding the eigenvector corresponding to the largest eigenvalue of the covariance matrix A.

—

Breakdown of PCA:

- Step 1: Compute the covariance matrix $A = X^TX$.
- Step 2: Find the eigenvalues and eigenvectors of A.
- Step 3: The eigenvector corresponding to the largest eigenvalue is the first principal component.

1.11.4 Recap

Example Optimization Problem:

$$\max_{v,\ ||v||_2=1} v^TAv$$

where $A = X^TX$.

Rotation Matrices: A rotation matrix is used to rotate the coordinate axis. For example, a counterclockwise rotation matrix is:

$$\begin{pmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{pmatrix}$$

Covariance matrices can be written as rotations and scalings.

1.11.5 Spectral Theorem

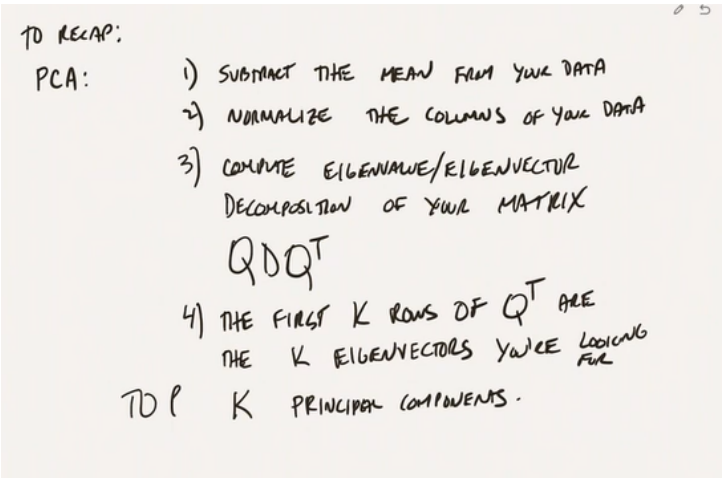
Spectral Theorem: Any symmetric matrix can be written as $A = QDQ^T$, where Q is orthogonal and D is diagonal. The entries of D are the eigenvalues of A.

Proof:

- Claim 1: For any vector v, $v^TAv \geq 0$.
- Claim 2: The eigenvalues of A cannot be negative.

1.11.6 Eigenvector Decomposition

PCA Recap: PCA involves finding the eigenvectors and eigenvalues of the covariance matrix. These eigenvectors form the principal components, and the eigenvalues indicate how much variance is captured by each component.



The next step involves using singular value decomposition (SVD) to compute the eigenvectors and eigenvalues.

Personal Notes

none yet

13 Singular Value Decomposition

Vocabulary

- **Outer product**: Multiplying a column vector by a row vector.
- **Inner product**: Multiplying a row vector by a column vector of the same dimension, resulting in a scalar value.
- **Orthogonal**: Vectors are orthogonal if their inner product is 0.
- **Principal axes**: The main directions along which the variance of data is maximized, identified via SVD or PCA.

Introduction

Singular Value Decomposition (SVD) is one of the most important methods for decomposing a matrix into simpler structures, revealing its underlying properties.

Netflix Challenge Problem

This is a matrix completion problem. Imagine Netflix wants to predict which users will like certain movies. We can represent users and movies in a matrix, where rows correspond to users, and columns to movies. The entries of this matrix are ratings given by users to movies. However, most of the matrix is incomplete because users haven't rated every movie.

The goal is to predict the missing entries in this matrix. With certain assumptions about structure, SVD helps solve this problem.

Example incomplete matrix:

$$\begin{pmatrix} 1 & ? & ? \\ ? & 2 & ? \\ ? & 6 & 9 \\ ? & ? & 3 \\ 4 & 4 & ? \end{pmatrix}$$

If we know that each row is a multiple of the other rows (indicating the matrix has rank 1), we can deduce the missing values. Solving this gives:

$$\begin{pmatrix} 1 & 1 & \frac{3}{2} \\ 2 & 2 & 3 \\ 6 & 6 & 9 \\ 2 & 2 & 3 \\ 4 & 4 & 6 \end{pmatrix}$$

Matrix Ranks:

- **Rank-0 matrix**: All entries are 0.
- **Rank-1 matrix**: All rows (or columns) are multiples of each other.

If A is a rank-1 matrix, we can write:

$$A = u \otimes v^T$$

where u is an $m \times 1$ vector, and v is an $n \times 1$ vector. Each element $A_{ij} = u_i v_j$.

Matrix Notation for Rank-1 Matrix:

$$A = u \otimes v^T = \begin{pmatrix} u_1v_1 & u_1v_2 & \dots & u_1v_n \\ u_2v_1 & u_2v_2 & \dots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \dots & u_mv_n \end{pmatrix}$$

Rank Matrices

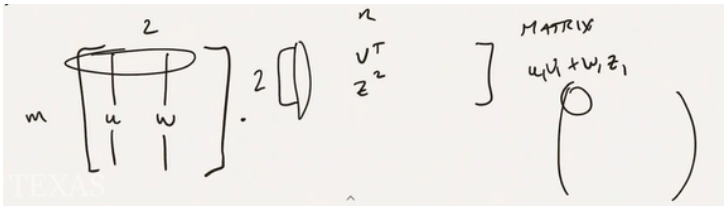
For a rank-2 matrix, we have:

$$A = uv^T + wz^T$$

This can be expressed as:

$$A = \begin{pmatrix} u_1v_1 + w_1z_1 & u_1v_2 + w_1z_2 & \dots & u_1v_n + w_1z_n \\ u_2v_1 + w_2z_1 & u_2v_2 + w_2z_2 & \dots & u_2v_n + w_2z_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 + w_mz_1 & u_mv_2 + w_mz_2 & \dots & u_mv_n + w_mz_n \end{pmatrix}$$

This corresponds to multiplying an $m \times 2$ matrix of columns u and w by a $2 \times n$ matrix of rows v^T and z^T.



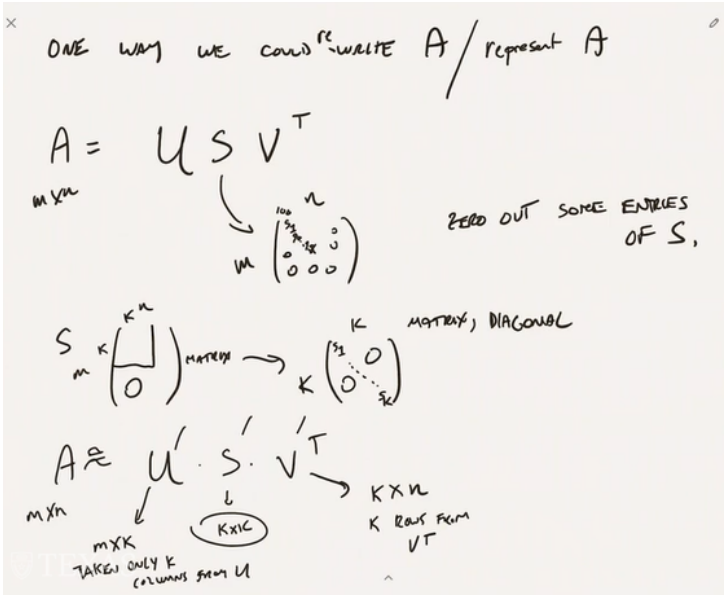
Defining the Singular Value Decomposition (SVD)

A matrix A can be factorized as:

$$A = USV^T$$

where:

- U is an $m \times m$ orthogonal matrix (columns are left singular vectors).
- V^T is an $n \times n$ orthogonal matrix (rows are right singular vectors).
- S is an $m \times n$ diagonal matrix with non-negative entries called the singular values.



The singular values are unique and can be ordered $s_1 \geq s_2 \geq \cdots \geq 0$. Algebraically, A can be written as:

$$A = \sum_{i=1}^{\min(m,n)} s_i u_i \otimes v_i^T$$

Computational Complexity

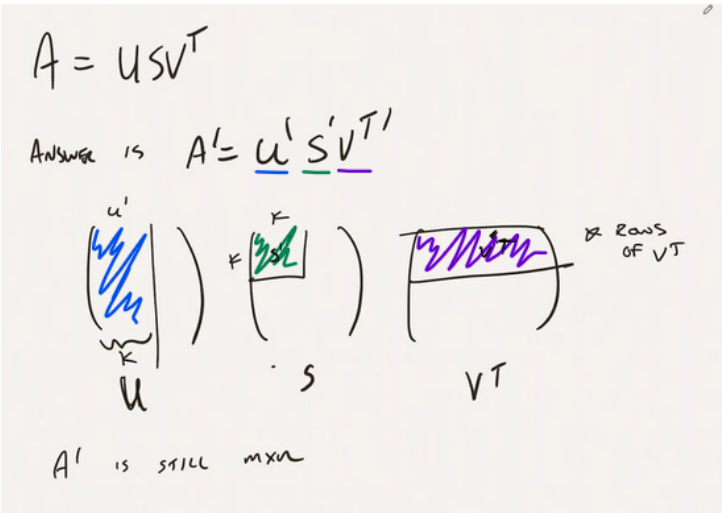
The time complexity of computing SVD is $O(m^2 n)$ or $O(n^2 m)$, whichever is smaller.

Frobenius Norm

The Frobenius norm of a matrix A is defined as:

$$||A||_F = \sqrt{\sum_{i,j} A_{ij}^2}$$

To approximate a matrix A by a rank-k matrix A', we compute the SVD of A and take the top k singular values and vectors.



Matrix Completion

To complete a matrix with missing entries, we can:

1. Make an initial guess for the missing entries.
2. Compute the best rank-k approximation.
3. Use SVD to output the completed matrix.

This approach works well when the matrix has low rank, meaning much of the data is structured.

Choosing k

The value of k is a hyperparameter that can be determined using heuristics. A common approach is to choose enough singular values so that the sum of the remaining singular values is less than 10% of the total.

Applying SVD

Using SVD for Linear Regression:

For a linear regression problem:

$$\min_{x \in \mathbb{R}^n} ||Ax - b||^2$$

When A is diagonal:

$$D = \begin{pmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & d_n \end{pmatrix}$$

The solution is:

$$x = D^{\dagger}b$$

where $D^{\dagger}$ is the pseudoinverse of D.

General Case: For a general matrix A, the solution is:

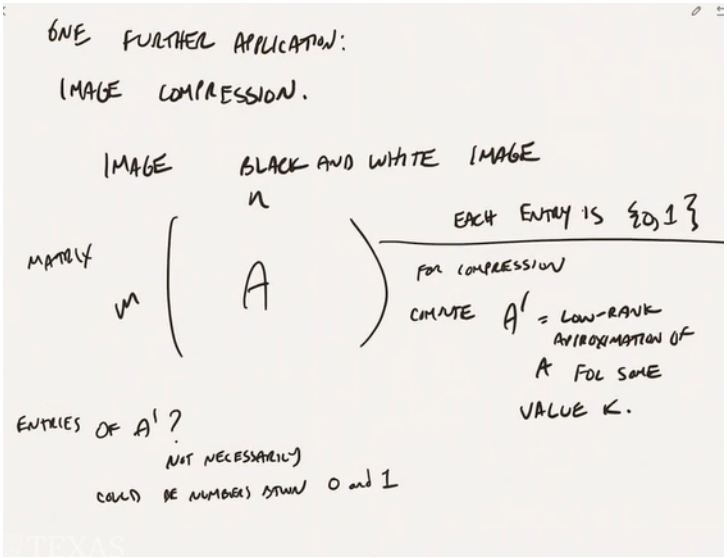
$$x = VS^{\dagger}U^{T}b$$

Using SVD for PCA:

Recall PCA involves the eigendecomposition of a covariance matrix. Using SVD:

$$X^{T}X = VS^{2}V^{T}$$

The right singular vectors of X are the principal components, and the singular values are the square roots of the eigenvalues of $X^{T}X$.



Personal Notes

[A singularly valuable decomposition: the SVD of a matrix](1.12-supplementary-reading-svd.pdf)
A matrix with rank-5 has at most 5 non-zero distinct eigenvalues, but these could repeat or be zero. It won't exceed 5.

14 Study Questions

Question 1

Question:
Devise the update rule for minimizing $f(x) = (\max(0, x)\frac{1}{2})^2$ using gradient descent. Roughly, when will gradient descent succeed or fail (based on where you start and step size)?

Answer:
The gradient descent update rule for a function is derived by:

1. Finding the loss function for a point x_i
2. Taking the derivative with respect to the model parameters to compute the gradient of the loss function, or $\nabla L(w)$
3. Plugging these values into the update rule to form:

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla L(w_{\text{old}})$$

Finding the Loss Function:
Since we are given $f(x) = (\max(0, x)\frac{1}{2})^2$, we assume this is the loss function to perform gradient descent on.

Taking the Derivatives:
Since the function contains a max() function, we will compute the gradient for when $\max(0, x) = 0$ and when $\max(0, x) = x$:

- For $f(x) = (0\frac{1}{2})^2$:

$$\frac{d}{dx}((0\frac{1}{2})^2) = 0$$

- For $f(x) = (x\frac{1}{2})^2$:

$$\frac{d}{dx}((x\frac{1}{2})^2) = 2(x\frac{1}{2}) \cdot 1 = 2(x\frac{1}{2})$$

Plugging it all in:

- When $\max(0, x) = 0$, the update rule is:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot 0$$

- When $\max(0, x) = x$, the update rule is:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot 2(x - \frac{1}{2})$$

When Will Gradient Descent Succeed or Fail:

- Gradient descent may fail when:
  - The function has local minima, which may prevent the algorithm from finding the global minimum.
  - The step size parameter is too large, causing oscillation or divergence.
- We check convexity via the second derivative:

$$\frac{d}{dx}(2(x - \frac{1}{2})) = 2$$

$$\frac{d}{dx}(2(0 - \frac{1}{2})) = 0$$

- The function is convex, so the algorithm can converge to a minimum. The minimum occurs when:

$$2(x - \frac{1}{2}) = 0$$

$$x = \frac{1}{2}$$

In conclusion, gradient descent will succeed if $x > 0$ with a reasonable step size. It will fail if $x \leq 0$ or if the step size is too large.

Question 2

Question:
Given a dataset A, under what circumstances will its projection onto its principal components equal its projection onto its right and left singular vectors?

Answer:
PCA is typically applied to a dataset after organizing it into a matrix A. There are two scenarios:

- **Points are rows:** Principal components are the right singular vectors, the eigenvectors of A^TA.
- **Points are columns:** Principal components are the left singular vectors, the eigenvectors of AA^T.

In both cases, if a point's projection onto the first k principal components equals the point itself, it implies the data is k-dimensional.

PCA computes the eigendecomposition of the covariance matrix A^TA as QQ^T, where Q is an orthogonal matrix, and contains the eigenvalues. On the other hand, SVD factorizes A as USV^T, where U and V^T are orthogonal, and S is a diagonal matrix of singular values. SVD generalizes PCA to non-square matrices.

Question 3

Question:
Let S be a set of documents and T a set of terms. Suppose C is a binary term-document incidence matrix. What do the entries of C^TC represent?

Answer:
C is a term-document incidence matrix, and C^TC will yield a document-document matrix:

- Entry (i, j) represents the number of shared terms between documents i and j.
- Diagonal entries (i, i) represent the total number of terms in document i.

Question 4

Question:
How did we derive the equations for simple linear regression in class?

Answer:
The goal is to find a vector $w \in \mathbb{R}^n$ that minimizes $||X \cdot w - y||_2^2$.
Given data points, the formal problem statement is:

$$\min_w ||Xw - y||_2^2$$

We solve for w using geometric intuition, where Xw is the orthogonal projection of y onto the span of X's columns. The projection condition gives:

$$X^T \cdot (y - Xw) = 0 \implies X^T y = X^T X w \implies w = (X^T X)^{-1} X^T y$$

Question 5

Question:
Why can you assume without loss of generality that a mistake-bounded learner only updates its state when it makes a mistake?

Answer:
Any mistake-bounded learner can be modified to only update when it makes a mistake without changing its mistake bound. If a learner updates when the label is correct, the order of examples can be rearranged so the learner only updates on the first T examples where mistakes occur.

Question 6

Question:
You are given a dataset with m points and a weak-learning algorithm with 60% accuracy. Each classifier output by the algorithm is encoded with two bits. How can you construct a classifier with less than m bits that is correct on every data point?

Answer:
The idea is to use boosting to reduce error. With AdaBoost, the training error after T rounds is at most $\exp(-2\gamma^2 T)$, where $\gamma = 0.1$. Thus, $T = \mathcal{O}(\log m)$ rounds are sufficient for training error $< \frac{1}{m}$. The final classifier, a majority vote of T classifiers (each encoded with 2 bits), requires $\mathcal{O}(\log m)$ bits.

Question 7

Question:
How can you compare Markov's inequality, Chebyshev's inequality, and the Chernoff bound?

Answer:

- **Markov's Inequality:** Applies to non-negative random variables.
- **Chebyshev's Inequality:** Applies when you have a variance bound.
- **Chernoff Bound:** Applies to sums of i.i.d. random variables, particularly indicator variables.

Chernoff is the strongest, while Markov is the weakest. Each inequality describes the probability of a random variable deviating from its mean under different assumptions.