8 Linear Regression
  8.1 Introduction .................................................. 19
  8.2 Single-Variable Case ........................................... 19
  8.3 Multiple-Variable Case ....................................... 20
  8.4 Maximum Likelihood ........................................... 20

9 Bias and Variance
  9.1 Introduction .................................................. 21
  9.2 Ridge Regression .............................................. 21
  9.3 The Bias-Variance Tradeoff ................................. 21

10 Gradient Descent
  10.1 Introduction .................................................. 23

11 Logistic Regression
  11.1 Introduction .................................................. 25
  11.2 0-1 Loss and Motivation for Logistic Regression ....... 25
  11.3 Some Other Losses ........................................... 25
  11.4 Logistic Regression .......................................... 25

12 Neural Networks
  12.1 Introduction .................................................. 27
  12.2 Backpropagation .............................................. 27

13 Boosting
  13.1 Introduction .................................................. 28
  13.2 AdaBoost Algorithm ......................................... 28
  13.3 Hedge Algorithm .............................................. 29

14 Confidence Intervals
  14.1 Introduction .................................................. 30
  14.2 Statistical Preliminaries .................................... 30

15 Random Projection
  15.1 Introduction .................................................. 32
  15.2 Hashing .......................................................... 32

16 Principal Component Analysis
  16.1 Introduction .................................................. 35
  16.2 PCA .............................................................. 35
  16.3 Calculating Principal Components ........................ 36

17 Principal Component Analysis, Part 2
  17.1 Recap ............................................................ 38
  17.2 Full PCA Procedure .......................................... 38
18 Singular Value Decomposition 41
  18.1 Introduction ................................................................. 41
  18.2 Matrix Completion .......................................................... 41
  18.3 Linear Regression ........................................................... 41
  18.4 PCA ................................................................................. 42

19 Latent Semantic Index 43
  19.1 Latent Semantic Indexing ...................................................... 43
  19.2 K-Means Clustering ............................................................ 44

20 Learning with Noise 45
  20.1 Introduction ................................................................. 45
  20.2 The Statistical Query Model ................................................... 45

21 Agnostic Learning 47
  21.1 Introduction ................................................................. 47
1. The Mistake-Bounded Model

1.1 Mistake-bounded Model

A learner receives an unlabeled data point (a challenge), predicts the class, and is then given the correct answer. When the learner makes a mistake, it changes its internal state in an attempt to improve its prediction accuracy, but it cannot control which data points it receives. More abstractly, from a family of functions $C$, the learner tries to figure out which $f \in C$ represents the labeling of the data.

**Definition 1.1.** A learner has mistake bound $t$ if for a sequence of challenges, it makes at most $t$ mistakes in class prediction.

**Example 1.2.** Let the function $f$ to learn belong to the class $C$ of monotone disjunctions of $n$ variables $x_1...x_n$. Come up with an algorithm with a mistake bound of at most $n$.

Start with an initial “guess” of $f = x_1 \lor x_2 \lor ... \lor x_n$. If a prediction is correct, don’t change the state. If it is wrong, then remove all variables present in the challenge from the guess of $f$. Since the true function has at least 0 elements in its disjunction and every mistake removes at least 1 from the guess, the mistake bound is $n$.

**Example 1.3.** How could you modify the above algorithm to learn general disjunctions (like monotone disjunctions but with negations) with mistake bound $O(n)$?

Conduct feature expansion. Create new variables $y_1...y_n$ corresponding to $\neg x_1...\neg x_n$ respectively, and use the algorithm from the previous example. You get a mistake bound of $2n = O(n)$.

1.2 Learning Decision Lists

A decision list is a structure used to express nested if-then-else relationships between boolean variables and are strictly more expressive than disjunctions. The proof of this is left as a homework problem. The following is an example:

\[
\begin{array}{c}
\begin{array}{c}
 x_1 \\
 1 \\
\end{array} \\
\begin{array}{c}
 x_2 \\
 0 \\
\end{array} \\
\begin{array}{c}
 x_3 \\
 1 \\
\end{array} \\
\end{array}
\]

In the above decision list, if $x_1 = 1$, then go down and choose the value below; otherwise, move right and recurse (if $x_2 = 1$ choose the value below, otherwise move right, etc.). The right-most 1 is a “catch-all” which is selected if every other variable is 0.

**Definition 1.4.** A basic block is a single node in a decision list.
A decision list of $n$ possible variables has $4n$ choices for basic blocks, since each block of a given variable has 2 possible literal values (negated or not) and could map to 0 or 1. Since \LaTeX drawing is painful, I will represent basic blocks as tuples, like $(-x_1, 1)$ and $(x_2, 0)$.

1.2.1 Learning Algorithm

When learning a decision list, the initial state is a list of $k$ bags, with the first one containing all $4n$ possible basic blocks and the rest containing nothing.

When the learner receives an input, it chooses some basic block from the first bag whose “down arrow” is taken (e.g. with 01010 input, $(-x_1, 0)$ and $(-x_1, 1)$ are valid choices). If none such blocks exist, it moves on to the next bag, and so on. The learner then outputs the value of the down arrow.

If the guess is correct, nothing is changed. If it is incorrect, the basic block is demoted to the next bag. Since there are $4n$ basic blocks and each can be demoted at most $k$ times, the mistake bound is $4nk$. A true basic block in the true decision list we’re trying to learn will never be demoted past its proper location. We can use induction to show that if bag $k$ is reached, then all the basic blocks in the previous bags are consistent with the provided input.

1.3 The Halving Algorithm

In this algorithm, you enumerate all the functions in the function class and output the majority vote. If a mistake is made, then all the functions that output the majority are discarded. Since at least half the remaining functions are discarded every time a mistake is made, the mistake bound is $O(\log |C|)$.

This algorithm is very inefficient for large function classes but provides a good theoretical mistake bound target for any mistake-bounded algorithm.
2. Mistake-bounded Model, Part 2

2.1 The Winnowing Algorithm

Last lecture, we discussed the Halving Algorithm for learning functions from a finite class $C$ with a mistake bound $O(\log |C|)$. This algorithm is usually not practical as the size of $C$ may be extremely large.

Now we describe an algorithm called the winnowing algorithm that can efficiently learn monotone disjunctions in $n$ variables with at most $k$ terms. It works as follows:

- Assign weights $w_i$ to each variable $x_i$. Initialize them all to 1.
- Read an input from the teacher. If the sum of the weights on the variables in the input $\geq n$, predict 1, else predict 0.
- If you’re correct then don’t change state. If you guessed 1 but the answer was 0, zero out the weights of all the variables in the input. If you guessed 0 but the answer was 1, double the weights of the variables in the input. These updates are type I and type II respectively.

**Proof.** We now prove that the algorithm has mistake bound $O(k \log n)$. The completely correct state is when each of the variables in the true disjunction has weight $n$ and every other weight is 0. To get every true variable to $n$, you need at most $k \log n$ type I updates.

Since for every type I update, the weight increases by at most $n$ and with every type II update, it decreases by at least $n$, and the sum of the weights is non-negative, the number of type II updates is $\leq$ the number of type I updates. Thus, the mistake bound is $O(k \log n)$.

2.2 The Best Experts Algorithm

This algorithm can be extended to the best experts algorithm. Say we have $k$ predictors $C_1, ..., C_k$.

- Initialize weights to 1.
- Get an input from the teacher, and output the weighted majority vote.
- If the majority vote is correct, do nothing. If it is wrong, cut the weights of the incorrect predictors in half.

If $OPT$ is the mistake bound of the best predictor, the mistake bound of the ensemble of experts is $O(OPT + \log k)$. 

3. Decision Trees

3.1 Introduction to Decision Trees

Definition 3.1. A decision tree is a boolean function \( f : \{0, 1\}^n \rightarrow \{0, 1\} \) in a tree format with nodes representing conditions (in our lecture today, boolean literals). Starting at the node, if the condition is true, recurse to the left, else recurse to the right. The leaves of the tree are truth values output if reached.

Definition 3.2. The error rate, training error, or empirical error rate of a tree \( T \) on training set \( S \) is
\[
\frac{\# \text{ mistakes } T \text{ makes on } S}{|S|}
\]

Definition 3.3. The gain of a model with respect to an old model is the reduction in error rate brought about by the new model (old error rate – new error rate).

Question 3.4. Given a labeled training set, how do we build a decision tree with low error rate?

We start with very naive solutions and progress to a viable solution.

Example 3.5. If our decision tree is just a leaf (a single output), the smartest strategy is to make its truth value the one that the majority label in set \( S \). Thus, if \( S = \{(1010, 1), (1000, 0), (0010, 0)\} \), the leaf should be set to 0.

Example 3.6. Now a more interesting case: If our tree has a single node that’s not a leaf, which variable do we assign it?

At a high level, we wish to pick a literal whose being true most correlates to an output of 1 and whose being false most correlates to the output being 0; in other words, we want to pick a literal whose truth value most accurately reflects that of true decision tree underlying the labeling of \( S \).

Let’s defer figuring out which literal to choose and assume that \( x_1 \) has been chosen for us. Intuitively, we make the left leaf equal to the majority output vote of data in the training set where \( x_1 = 0 \) and do the same on the right. How do we assess the error rate of this decision tree?

We define a potential function \( \phi(a) = \min(a, 1 - a) \), where \( a \) is a probability. For example, \( \phi(10/15) = \min(2/3, 1 - 2/3) = 1/3 \). Also, the error rate of the trivial decision tree from the previous example is \( \phi(\Pr_{(x,y)\sim S}(y = 0)) \). In this example, the tree with \( x_1 \) at the root has an error rate of \( \Pr(x_1 = 0)\Pr(y = 1|x_1 = 0) + \Pr(x_1 = 1)\Pr(y = 0|x_1 = 1) \), which can be rewritten in terms of the potential function as \( \Pr(x_1 = 0)\phi(\Pr(y = 0|x_1 = 0)) + \Pr(x_1 = 1)\phi(\Pr(y = 0|x_1 = 1)) \).

Now the algorithm should be apparent. Pick the root of the tree to be the literal that minimizes the provided error function. Recursively build the left and right subtrees using the subset of \( S \) in which \( x_1 = 0 \) and \( x_1 = 1 \) respectively. We discuss stopping criteria (when we should stop adding more subtrees) later on.
Notice that the final decision tree obtained depends on the error function we use to pick the root of each subtree. An alternative is the Gini index, $\phi(a) = 2a(1 - a)$. Figure 3.1 shows graphs of both functions over their domains. From the graphs it should be clear that using the Gini function would produce the same tree as the function we came up with. An advantage of the Gini function is that it’s differentiable over its entire domain, which is pretty useful in many machine learning applications.

**Exercise 3.7.** If we’re given sample data shown in Figure 3.2, what is the error rate of a trivial decision leaf? If we decide to make a 1-node tree, which literal should be placed at the root? Use the Gini function as $\phi$.

For the trivial decision tree, we get $\phi(Pr(Neg)) = 2(5/15)(1 - 5/15) = 4/9$. To pick the root of the less trivial tree, we choose the literal that minimizes error. Let’s calculate the error for each literal:

- **$x_1$:**

  \[
  error = Pr(x_1 = 0)\phi(Pr(y = 0|x_1 = 0)) + Pr(x_1 = 1)\phi(Pr(y = 0|x_1 = 1)) \\
  = \frac{5}{15} \cdot \phi\left(\frac{2}{5}\right) + \frac{10}{15} \cdot \phi\left(\frac{3}{10}\right) \\
  = \frac{1}{3} \cdot 2 \cdot \frac{2}{5} \cdot \frac{3}{5} + \frac{2}{3} \cdot 2 \cdot \frac{3}{10} \cdot \frac{7}{10} \\
  = \frac{4}{25} + \frac{7}{25} \\
  = \frac{11}{25}
  
  
  
- **$x_2$:** The same arduous calculations as above yield $error = \frac{4}{5}$.

Since the error for $x_2$ is slightly smaller, we would place it at the root and recurse on both sides, conditioning on the value of $x_2$ that each branch takes.
3.2 When do we stop?

The recursive method we derived for building trees is pretty intuitive and computationally simple but does not provide stopping conditions (a base case). There is no “best” stopping condition, and this is an active area of research, but we explore some simple methods.

One obvious choice is to stop when the gain from building another subtree is below a certain threshold. This ensures that we stop unnecessarily increasing the size of the tree if it doesn’t help much and helps reduce overfitting (covered in a future lecture). In a similar vein, we could just place a restriction on the total number of nodes in the tree.

Another popular technique is random forests. We first build many small trees using the following method:

1. Randomly subsample (usually with replacement) $S'$ from $S$.
2. Randomly subsample $k$ features from the literal set $\{x_1, ..., x_n\}$.
3. Build a decision tree using $S'$ restricted to using the $k$ features as nodes.

For each input, output the majority vote of the “forest”.

3.3 Back to the Mistake-bounded Model

Our goal is to now develop an algorithm to learn decision trees of size $S$ with mistake-bound and runtime of $n^{O(\log S)}$. The following theorem shows how it’s done.

**Theorem 3.8.** Decision trees of size $S$ can be learned by decision lists with $O(\log s)$ conjunctions inside each block.

If the above theorem were true, we could just conduct feature expansion to make a basic block out of every subset of literals of size $\leq O(\log s)$, of which there are $n^{O(\log s)}$ many. Running the same decision list learning algorithm we developed earlier would give a runtime and mistake-bound of $n^{O(\log s)}$. We still need to prove the theorem though, which is easiest when we build some framework first.

**Definition 3.9.** The rank of a decision tree $T$ is recursively as follows:

- If $T$ is a leaf, $\text{rank}(T) = 0$.
- If $\text{rank}(T.left) = \text{rank}(T.right)$, then $\text{rank}(T) = \text{rank}(T.left) + 1$.
- Otherwise, $\text{rank}(T) = \max(\text{rank}(T.left), \text{rank}(T.right))$.
Remark 3.10. The definition of height is similar to the one above, except you add 1 to the max in the last case (so the last 2 cases collapse to 1).

Lemma 3.11. A decision tree of size \( S \) has a rank of at most \( \log S \).

Proof: We induct on the size of the tree. If \( S = 1 \), then the tree is just a leaf and has rank 0 = \( \log 1 \). Assume that for all sizes \( s \leq \) some fixed \( k > 1 \), a tree of size \( s \) has rank \( \leq \log s \). We show that a tree of size \( k + 1 \) has rank \( \leq \log(k + 1) \).

Without loss of generality, assume that the left subtree \( T_1 \) has size \( \leq k/2 \). By the inductive hypothesis, \( \text{rank}(T_1) \leq \log k/2 = \log k - 1 \). If \( \text{rank}(T_1) = \text{rank}(T_2) \), then \( \text{rank}(T) = \text{rank}(T_1) + 1 \leq \log(k + 1) \). If they are not equal, then the rank of \( T \) is the max of the ranks of both subtrees, but since both subtrees have size less than \( k + 1 \), we get \( \text{rank}(T) = \max(\text{rank}(T_1), \text{rank}(T_2)) \leq \log \max(\text{size}(T_1), \text{size}(T_2)) < \log(k + 1) \). □

Lemma 3.12. If a decision tree has rank \( t \), it is computable by a decision list of conjunctions of at most \( t \) literals in each block.

Proof: Left to the reader. □
4. PAC Learning

Notation: $S$ is the training set.

4.1 The Problem of Generalization

A major concern in machine learning is how well a model generalizes from the training set from which it was built to test data after the fact.

- Pathological case: The learner memorizes the training set (0 training error) and outputs a default value for anything not in the training set. This is an example of very poor generalization (unless the training set is the entire set of data).

- Another bad example: Build a decision tree of size $|S|$. This is just memorization (every possible path found in training data).

4.2 Estimating Test Error

At a high level, there are 2 approaches to better capturing the generalization capabilities of a model: the use of a training set and the use of a potential function to tweak the training error rate.

The former is done by setting aside a fraction of the training data, fitting a model with the remaining data, and calculating the error rate on the “holdout” set. Since the holdout set was not used to fit the model, that error rate better illustrates how the model generalizes to data it has not seen before.

The latter is done by adding terms to the training error rate to penalize it for potential signs of overfitting. For example, we could define the error function of a tree as

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

This function penalizes larger trees for potentially overfitting the training data, even if they do reasonably well.

4.3 The PAC Model

**Definition 4.1.** Let $\mathcal{D}$ be the distribution of inputs, $\mathcal{C}$ be the class of functions to learn from, and $c$ be the particular function from $\mathcal{C}$ we are trying to learn. We want to output some function $h$ such that with probability $\geq 1 - \delta$, the function $h$ is such that $\Pr_{x \sim \mathcal{D}}[h(x) \neq c(x)] < \epsilon$, and the runtime of the learning algorithm should be polynomial with respect to $(1/\epsilon, 1/\delta, n, s)$. Here, $s$ is the “size” if the model with respect to $S$, and $n$ is the size of the input.

Let’s assume that we have some algorithm $A$ that, given some training set $S$, outputs an $h \in \mathcal{C}$ consistent with $S$. Once we have chosen a function class to learn from, $\delta$, and $\epsilon$, all that remains is to
determine a lower bound on $S$ to achieve the aforementioned probability conditions.

The “bad event” whose probability must be below $\delta$ is that $A$ outputs a function consistent with $S$ but with error $> \epsilon$. That probability is $\leq C(1 - \epsilon)^{|S|}$. With that in mind, we can derive a suitable lower bound on $S$:

$$C(1 - \epsilon)^{|S|} \leq \delta$$

$$e^{-\epsilon|S|} \leq \delta/C$$

$$|S| \geq \log \frac{C}{\delta/\epsilon}$$

In this particular equation, we require that the function class is finite.
5. PAC Learning, Part 2

5.1 PAC Algorithms for Familiar Problems

We start with a couple of examples of converting mistake-bounded models from previous lectures into PAC ones.

Example 5.1. Come up with a PAC learning algorithm for monotone disjunctions.

We can conduct the same elimination algorithm we came up with for the mistake-bounded model, but this time we do it on \( S \) labeled samples drawn from distribution \( \mathcal{D} \). \( S \) can be calculated using the equation derived last class.

Example 5.2. Come up with a PAC learning algorithm for decision lists.

First use \( S \) to find some basic block at the head that is consistent with the training set. Then remove all items from \( S \) that short-circuit at the head to form training set \( S' \). Then recurse on the tail of the list. The resulting decision list will be consistent with the training set. \( S \) should be chosen based on the equation derived earlier.

5.2 A Little Harder

We now examine a problem that presents a new challenge, an infinite function class. Since the sample size equation derived the last class requires that the function class be finite, we cannot directly apply it here; however, similar reasoning is used.

Example 5.3. Given a distribution \( \mathcal{D} \) that outputs some point in \( \mathbb{R}^2 \) from anywhere in an axis-parallel rectangle with equal probability, how do you PAC-learn such a rectangle?

Our strategy is simple: just draw \( |S| \) points from the distribution and output the rectangle that exactly bounds the points, namely that with corner points consisting of combinations of the min and max \( x \) and \( y \) values of points in \( S \). The hard part is figuring \( |S| \), how many points should be drawn.

To do this, we must choose a large enough \( |S| \) such that the probability of a “bad event” is less than \( \delta \).
6. Cross-Validation and VC Dimension

6.1 Probability Review

**Theorem 6.1 (Markov’s Inequality).** Let $X$ be a nonnegative random variable. Then

$$\Pr[ X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k}$$

**Definition 6.2.** If $X$ is a random variable, then its variance is $\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2]$. Its standard deviation is $\sigma = \sqrt{\text{Var}[X]}$.

**Theorem 6.3 (Chebyshev’s Inequality).** If $X$ is a random variable with $\mathbb{E}[X] = \mu$, then

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

**Theorem 6.4 (Chernoff Bound).** Let $X_1, \ldots, X_n$ be random variables with $\mathbb{E}[X_i] = p$. Let $S = \sum_{i=1}^n X_i$.

Then the following inequalities hold

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

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

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

6.2 Cross-Validation

**Example 6.5.** We can use the Chernoff bound to estimate the true error of a classifier on a holdout set. Let $S$ be a holdout sample of size $n$ drawn from distribution $D$. Our classifier outputs hypothesis $h$, and we’re trying to predict $z = \Pr_{x \sim D}[h(x) \neq c(x)]$.

Let $X_i$ be the random variable that equals 0 if $h$ is correct on $x_i$ and 1 otherwise. Let $S = \sum_{i=1}^n X_i$. $\mathbb{E}[S] = np$, where $p$ is the true error of the classifier. The Chernoff bound tells us that $\Pr[ |S - np| > \delta n ] \leq 2e^{-2n\delta^2}$. For a given $\delta$, we can choose an appropriate $n$ to get arbitrarily small (but still nonzero) error.

The above example shows that in practice, the holdout set method is pretty expensive. The best alternative is cross-validation. This is a less expensive method that is used in practice to estimate test error and tune the hyperparameters in a model. Here’s how $k$-fold cross-validation works:

1. Randomly split the training data into $k$ groups of roughly equal size.

2. Set 1 of the groups aside, train the model on the other $k-1$ and calculate the test error on the held-out data set.

3. Repeat the above process $k$ times, using a different hold-out set each time. Average the $k$ error estimates you get.
6.3 VC Dimension

So far, our inequality for the minimum number of samples required to get a classifier with test error < \varepsilon with probability > 1 - \delta is as follows

$$|S| \geq \frac{1}{\varepsilon} \log \frac{|\mathcal{C}|}{\delta}$$

The problem with this equation is that it assumes that the function class \mathcal{C} is finite. This is usually not the case. The concept of VC dimension alleviates that. For infinite function classes, we can replace \(|\mathcal{C}| with VC(\mathcal{C}) in the above equation.

**Definition 6.6.** Let \( S \) be a set of \( m \) unlabeled points drawn from distribution \( \mathcal{D} \). Fix the function class \( \mathcal{C} \). For some \( c \in \mathcal{C} \), denote the “realization” of that function be the labels it generates for set \( S \), namely \((c(s_1),...,c(s_m)) \in \{0,1\}^m\). We say that function class \( \mathcal{C} \) shatters \( S \) if across all functions in \( \mathcal{C} \), all \( 2^m \) possible realizations of \( S \) are achieved.

**Definition 6.7.** We say that \( VC(\mathcal{C}) \geq m \) if there exists some \( S \subset X \) with \(|S| = m \) that is shattered by \( \mathcal{C} \). We say that \( VC(\mathcal{C}) < m \) if no training set of size \( m \) is shattered by \( \mathcal{C} \). Thus, \( VC(\mathcal{C}) = m \) if \( m \) is the largest cardinality such that for some \(|S| = m \), \( \mathcal{C} \) shatters \( S \).

**Example 6.8.** Take \( \mathcal{C} = \{ \text{closed intervals in real line} \} \), \( X = \mathbb{R} \). For any possible labeling of 2 points, we can construct an interval that generates that label. Thus, \( VC(\mathcal{C}) \geq 2 \). However, for 3 points, if we label the middle point negative and the outer 2 positive, no interval can generate that labeling. Thus, \( VC(\mathcal{C}) < 3 \).
7. Perceptron

7.1 Learning Halfspaces

The perceptron algorithm is a method for learning halfspaces. As a review, a halfspace is a function of the form \( f(x) = \text{SIGN}(w \cdot x - \theta) \), where \( w, x \in \mathbb{R}^n \) and \( \theta \in \mathbb{R} \). Our goal is to learn the vector \( w \).

A less efficient (but still polynomial-time) algorithm is to express training data as a set of inequalities and use linear programming algorithms to solve for a valid \( w \). This is much more expensive than the perceptron algorithm.

7.2 The Perceptron Algorithm

The perceptron algorithm is a mistake-bounded one with the learner’s state being its current estimate of \( w \). The algorithm works as follows:

- Initialize \( w \) to some starting value, \( 0 \) or \( \left( \frac{1}{\sqrt{n}}, \ldots, \frac{1}{\sqrt{n}} \right) \) is fine.
- The teacher feeds the learner an input \( x \) and the learner responds with its prediction \( \text{SIGN}(w \cdot x) \).
- If the learner is incorrect, it updates its state: \( w_{\text{new}} = w_{\text{old}} + yx \), where \( y \) is the true label of \( x \).

The above algorithm works as long as the following assumptions are met:

- \( \exists w^* \) that fits the training data in the first place.
- \( \|w^*\|_2 = 1 \)
- \( \|x\|_2 = 1 \)
- \( \theta = 0 \)
- The above assumptions can easily be dealt with, but this is the biggest one. Assume that all inputs \( x \) are at least distance \( \rho \) from \( w^* \). Equivalently, \( |w^* \cdot x| > \rho \). It’s known as the margin assumption.

**Theorem 7.1** (Perceptron Convergence Theorem). The mistake bound of the perceptron algorithm is \( O\left(\frac{1}{\rho^2}\right) \).

**Claim 7.2.** With each update, the quantity \( w \cdot w^* \) increases by at least \( \rho \).

**Proof:**

Let \( w_o \) denote the old value of \( w \) before the update. We get \( w \cdot w^* = (w_o + y \cdot x) \cdot w^* \), where \( y \in \{-1, 1\} \) is the true label of the input. Simplifying this shows that \( w \cdot w^* \) increases by the quantity \( y \cdot x \cdot w^* \).

By hypothesis, \( |x \cdot w^*| > \rho \) and since \( y \) has the same sign as the inner product, we can drop
the absolute value and conclude that $x \cdot w^*$, the quantity by which $w \cdot w^*$ increases, is at least $\rho$.

\[ \square \]

**Claim 7.3.** With each update, the quantity $\|w\|^2$ increases by at most 1.

**Proof:**
In the same vein as the last proof, we write
\[
\|w\|^2 = (w_0 + y \cdot x) \cdot (w_0 + y \cdot x) = \|w_0\|^2 + \|x\|^2 + 2y \cdot x \cdot w_0
\]

By hypothesis, $\|x\|^2 = 1$ and since we made a mistake, $x \cdot w_0$ and $y$ have opposite signs, making the last term negative and proving our claim.

\[ \square \]

We now prove the original theorem using these claims.

**Proof:**
After $t$ mistake, the following inequality holds:
\[
t \rho \leq w \cdot w^* \leq \|w\| \|w^*\|
\]
which can be simplified as follows:
\[
t \rho \leq w \cdot w^* \leq \|w\| \|w^*\|
\]
\[
t \rho \leq w \cdot w^* \leq \sqrt{t} \times 1
\]
\[
t \rho \leq \sqrt{t}
\]
from which we conclude that $t \leq \frac{1}{\rho^2}$.

\[ \square \]

### 7.3 Addressing our Assumptions

Earlier we made some assumptions that greatly simplified the following proofs. Now we revisit these and how they’re addressed in practice.

The assumption that $\theta = 0$ is addressed by just adding another variable to our hyperplane model and setting it to 1. We can re-analyze the perceptron algorithm with $\|x\| = R > 1$ to come up with a mistake bound of $O(R^2/\rho^2)$. Thus, using larger norms only gives us quadratic slowdown with respect to the upper bound of that norm.

### 7.4 Extending to Polynomial Thresholds
We can easily extend our perceptron algorithm for learning halfspaces to learning polynomial threshold functions, namely \( \text{SIGN}(p(x)) \). The way this is done is mapping each term in the expanded polynomial to a feature in the final halfspace.

**Example 7.4.** We want to learn the threshold function for a 2-variable polynomial of degree 2. This polynomial would be of the form \( p(x, y) = a_1x^2 + a_2xy + a_3y^2 + a_4 \). We just express the terms as a feature vector \( x = (x^2, xy, y^2, 1) \) while trying to learn the weights \( w^* = (a_1, a_2, a_3, a_4) \).

The problem with learning polynomials this way is that for a polynomial of \( n \) variables and degree \( d \), there are \( n^d \) features, and coming up with an appropriate \( \rho \) becomes more difficult. However, we can use something called a *kernel perceptron* to dramatically decrease runtime.

### 7.5 Kernel Perceptron

Let’s denote \( \varphi(x) \in \mathbb{R}^{n^d} \) to be the image of \( x \) in the expanded feature space. We define a kernel function \( K(x^1, x^2) = \varphi(x^1) \cdot \varphi(x^2) \). Assume \( K \) can be computed efficiently. Our goal is to learn \( w^* \in \mathbb{R}^{n^d} \), where \( n \) is the number of variables in the polynomial and \( d \) is its degree.

We initialize \( w = 0 \). If the true \( w^* \neq 0 \), then we’re guaranteed to update reading the first input, in which case \( w_{\text{new}} = 0 + y\varphi(x^1) \). When the next input arrives, we need to calculate \( w_{\text{new}} \cdot x^2 \), which really equals \((y\varphi(x^1))x^2 = yK(\varphi(x^1), \varphi(x^2))\). Continuing with this logic, if we make mistakes on the first \( t \) inputs, our prediction of the halfspace after \( t \) mistake, \( w_{t+1} \), equals \( \sum_{i=1}^{t} y \cdot \varphi(x^i) \). Thus, \( w_{t+1} \cdot x^{t+1} = \sum_{i=1}^{t} K(\varphi(x^i), \varphi(x^{i+1})) \). The benefit of this reorganization is that the kernel function is assumed to be easily computable. Now we look at an example of a kernel function.

**Example 7.5.** Consider the class of degree-2 polynomials in \( n \) variables.
8. Linear Regression

8.1 Introduction

Let $X, Y$ be 2 random variables. Our goal is to predict the value of $Y$. In the case that we don’t get to see $X$, the best we can do is just guess $\mathbb{E}[Y]$, the average of the values we’ve seen so far. In regression however, we do observe $X$ and condition our prediction of $Y$ on the value of $X$ in a given tuple of input data. Thus, we output $\mathbb{E}[Y|X]$.

The main hurdles are that $X$ may be hard to observe or even be unknown. Additionally, we may not know the relationship between $X$ and $Y$. Linear regression makes the assumption that the relationship between $X$ and $Y$ is linear. This may or may not be a good assumption.

That being said, the linear regression problem can be stated as follows: given random variables $X, Y$, what linear function $f(X) = \beta_0 + \beta_1 X$ should be used to predict $Y$? More specifically, what linear function minimizes the expected value of our loss function $\mathbb{E}[(Y - (\beta_0 + \beta_1 X))^2]$.

8.2 Single-Variable Case

Given a labeled training set $S = \{(x^1, y^1), \ldots, (x^n, y^n)\}$, we want to choose $\beta_0, \beta_1$ to minimize the quantity

$$l = \frac{1}{m} \sum_{i=1}^{m} (y^i - (\beta_0 + \beta_1 x^i))^2$$

This can be done by taking the partial derivatives with respect to $\beta_0, \beta_1$ and setting them to 0. This results in the following equations

$$\frac{\partial l}{\partial \beta_0} = \frac{1}{m} \sum_{j=1}^{m} (y^j - \beta_0 - \beta_1 x^j)(-2) = 0$$

$$\frac{\partial l}{\partial \beta_1} = \frac{1}{m} \sum_{j=1}^{m} (y^j - \beta_0 - \beta_1 x^j)(-2x^j) = 0$$

Setting both of these equations to 0 and solving for the coefficients yields

$$\beta_1 = \frac{\overline{xy} - \overline{y}\overline{x}}{\overline{x^2} - (\overline{x})^2} = \frac{\text{Cov}(X,Y)}{\text{Var}(X)}$$

$$\beta_0 = \overline{y} - \beta_1 \overline{x}$$

Since the function is convex, the minimum we obtain is global.
8.3 Multiple-Variable Case

In this case we’re fitting a line to \(n\)-dimensional data. \(X\) is an \(m \times n\) matrix, with each row \(i\) representing an \(n\)-dimensional random variable \(X_i\). Also, \(y \in \mathbb{R}^m\) and represents the predicted classes for the \(m\) random variables. Our goal is to find a \(w \in \mathbb{R}^n\) that minimizes \(\|X \cdot w - y\|^2\).

The value of \(X \cdot w\) is in the column space of \(X\) and the value of \(X \cdot w - y\) that minimizes the squared error is that which is orthogonal to \(y\). This can be written as \(X^T(y - X \cdot x) = 0\). Solving for \(w\) yields \(w = (X^T X)^{-1}X^T y\). This is known as the normal equation. The runtime of this algorithm can be crudely described as \(O(n^3 + mn^2)\). Other methods, like stochastic gradient descent, actually yield faster approximations than explicitly computing the solution.

8.4 Maximum Likelihood

Let \(X\) be a random variable and assume that \(Y\) is of the form

\[ Y = \beta_0 + \beta_1 X + \epsilon \]

where \(\epsilon\) is a noise variable that is normally distributed with mean 0 and variance \(\sigma^2\). Assume that we have also drawn samples \((x^1, y^1), \ldots, (x^m, y^m)\). Clearly, as we vary the values of the parameters, the probability that we see the training set changes. More specifically, the probability of seeing a data point is the probability that the error term takes on the difference between the data \(y\) and the predicted \(y\) from the equation. This can be written for all training points as:

\[ \prod_{i=1}^{m} P(Y^i|X^i, \beta_0, \beta_1) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(y^i - (\beta_0 + \beta_1 x^i))^2}{2\sigma^2}} \]

Our goal is to choose the coefficients that maximize the likelihood, which is equivalent to maximizing log-likelihood:

\[ \text{LOG}(L(\beta_0, \beta_1)) = \log \prod_{i=1}^{m} P(Y^i|X^i, \beta_0, \beta_1) \]

\[ = \sum_{i=1}^{m} \log(P(Y^i|X^i, \beta_0, \beta_1)) \]

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

From this we realize that we have to minimize the same least-squares cost function we came up with earlier. This illustrates the similarities between 2 different ways of fitting the linear regression model:

- Geometric: fitting coefficients to minimize squared distances from training data
- Statistical: maximizing log-likelihood
9. Bias and Variance

9.1 Introduction

In this lecture, we explore the concept of \textit{regularization}, penalizing models with large parameter weights that could signify overfitting.

Let’s review the setting in which we did linear regression. We had a training set \( X = \{x^1, ..., x^m\} \) of data, a set of labels \( y = \{y^1, ..., y^m\} \), and sought to minimize the cost function \( \|X \cdot w - Y\|^2 \) over all coefficient vectors \( w \). Simple multivariable calculus yielded a solution of \( w = (X^T X)^{-1} (X^T Y) \).

9.2 Ridge Regression

\textbf{Definition 9.1.} In the \textit{ridge regression} problem, we minimize a similar loss function as ordinary linear regression but add an L2 penalty on the weights of the model. Thus, we solve the following:

\[
\arg \min_{w \in \mathbb{R}^n} \|Xw - Y\|^2 + \lambda \|w\|^2
\]

where \( \lambda > 0 \) controls the severity of the penalty and is a hyperparameter we choose. When \( \lambda = 0 \), ridge regression reduces to ordinary least-squares regression. The solution to the ridge regression problem is:

\[
w = (X^T X + \lambda I)^{-1} (X^T Y)
\]

The benefit of ridge regression is that it helps prevent overfitting by penalizing large coefficients as large coefficients generally indicate that the model is fitting noise in the dataset. This is not an exact science, so \( \lambda \) can only be determined experimentally.

We now explore the effects of regularization on linear regression with a noise variable. Say we want to learn a model of the form \( y^i = a^* x^i + \varepsilon^i \), where all the parameters are scalars and \( \varepsilon \) is a noise term with \( \mathbb{E}[\varepsilon] = 0 \), \( \text{Var} [\varepsilon] = \sigma^2 \). Say we have \( m \) points in our training set of the form \((x^i, y^i)\). In this context, the ridge regression problem is

\[
\arg \min_a \sum_{i=1}^m (y^i - ax^i)^2 + \lambda a^2
\]

For a fixed \( \lambda \), we can solve the above minimization problem:

\[
\hat{a}(\lambda) = \frac{\sum_{i=1}^m x^i y^i}{\lambda + \sum_{i=1}^m (x^i)^2}
\]

9.3 The Bias-Variance Tradeoff

Now our task is to find the optimal \( \lambda \). We have a source of randomness in our data, namely the error term, so the quantity that we must minimize is the \textit{expected} mean squared error as a function of \( \lambda \):

\[
MSE(\lambda) = \mathbb{E}[(\hat{a}(\lambda) - a^*)^2]
\]
Using some algebraic trickery, we can rewrite the above equation in a very intuitive and useful form.

For convenience, we write $\hat{a}(\lambda)$ as simply $\hat{a}$ and assume $\lambda$ is fixed throughout the computation.

\[
MSE = \mathbb{E}[(a^* - \hat{a})^2]
\]

\[
= \mathbb{E}[(a^* - \hat{a} + \mathbb{E}[\hat{a}] - \mathbb{E}[\hat{a}])^2]
\]

\[
= \mathbb{E}[(a^* - \mathbb{E}[\hat{a}])^2] + \mathbb{E}[(\mathbb{E}[\hat{a}] - \hat{a})^2] + 2\mathbb{E}[(\mathbb{E}[\hat{a}] - \hat{a})(\mathbb{E}[\hat{a}] - \mathbb{E}[\hat{a}])]
\]

\[
= (a^* - \mathbb{E}[\hat{a}])^2 + \text{Var}(\hat{a})
\]

The relationship we derived is called the bias-variance tradeoff. We will now see that if there is a nonzero error term, then the $\lambda$ that minimizes squared loss is positive. This actually produces a biased estimator of $a^*$ but the reduction in variance is sufficient to justify it. We begin by calculating bias. For clarity we define $\beta = \frac{1}{m} \sum_{i=1}^{m} x_i^2$.

\[
\hat{a} - a^* = \frac{\sum_{i=1}^{m} x_i y_i}{\lambda + \sum_{i=1}^{m} (x_i)^2} - a^*
\]

\[
= \frac{\sum_{i=1}^{m} x_i (a^* x_i + \varepsilon_i)}{\lambda + m \beta} - a^*
\]

\[
= -\frac{\lambda a^*}{\lambda + m \beta} + \sum_{i=1}^{m} \frac{x_i}{\lambda + m \beta} \varepsilon_i
\]

Taking the expected value of the above quantity eliminates the second term since the expected value of the error term is 0. Thus, we end up with:

\[
\text{bias} = -\frac{\lambda a^*}{\lambda + m \beta}
\]

Similarly, we calculate variance as:

\[
\text{Var}(\hat{a}) = \frac{m \beta \sigma^2}{(\lambda + m \beta)^2}
\]

The final calculation for MSE is:

\[
MSE(\lambda) = \frac{m \beta \sigma^2}{(\lambda + m \beta)^2} + \frac{\lambda^2 (a^* )^2}{(\lambda + m \beta)^2}
\]

From this equation we can make some useful observations. If there is no error term, then the variance term disappears and choosing $\lambda = 0$ gives us an MSE of 0. If there is some error term (usually the case), then as $\lambda$ increases, the reduction in variance brings MSE down to some minimum after which the increase in bias overtakes and error goes back up.
10. Gradient Descent

10.1 Introduction

Gradient descent is a simple algorithm that pervades machine learning and is used to fit both simple models, like linear regression, and complicated ones, like neural networks with millions of parameters. Let’s begin with a simple example.

Example 10.1. Consider the simple bowl-shaped function \( f = (x - 2)^2 + 1 \). Pretend that we don’t know the exact equation whose value we’re trying to minimize, but we have a way of estimating its derivative at a given point. An intuitive way of approximating the minimum is to start at a given point, calculate the derivative, and move in the direction opposite to the derivative to “descend” to the bottom of the “bowl”.

Definition 10.2. A real-valued function is convex if the line segment connecting any 2 points on its graph is contained on or above the graph. More formally, \( f : \mathbb{R}^n \to \mathbb{R} \), is convex if for \( \lambda \in [0, 1], x_1, x_2 \in \mathbb{R}^n, f(\lambda x_1 + (1 - \lambda) x_2) \leq \lambda f(x_1) + (1 - \lambda) f(x_2) \).

It is clear that the naive method in the example does not work on general functions, since they may not have local minima, and even if they do, they may not be convex and have multiple local minima, leading you to a suboptimal solution if you start at some random point.

Example 10.3. Suppose \( f : \mathbb{R}^n \to \mathbb{R} \), with \( f(x) = w^T x + b \). The direction in which we should move to minimize \( f \) is just opposite the slopes, so \(-\frac{w}{||w||}\).

So far we’ve only dealt with trivial, linear functions. However, even in general cases, functions can be locally approximated with linear functions (e.g. Taylor series). Moving in the exact opposite direction of the gradient maximizes the “descent” when locating the minimum.

Example 10.4. Suppose \( f(x) = x^T A x - b^T x \). Calculate \( \nabla f \).

This becomes easier when the function is rewritten as a summation: \( f(x) = \sum_{i=1}^{n} a_{ij}x_i x_j + \sum_{i=1}^{n} b_i x_i \). Differentiating this with respect to some component \( x_k \) yields \( \frac{\partial f}{\partial x_k} = \sum_{j=1}^{n} a_{kj} x_j + \sum_{i=1}^{n} a_{ik} x_i - b_k \). Thus, in matrix notation

\[ \nabla f = Ax + A^T x - b \]

We are now ready to define the gradient descent procedure:

Definition 10.5. Given a differentiable function \( f \), gradient descent approximately finds it minimum using the following procedure:

1. Initialize \( w \) to some element in the domain of \( f \).
2. If \( \| \nabla f(w) \|_2 < \varepsilon \), terminate and output \( w \).
3. Otherwise update \( w' = w - \eta \nabla f(w) \) and recalculate the gradient.

\( \eta \) should be chosen to be small since the gradient gives only a local linear approximation of \( f \).
Example 10.6. In linear regression, with a training set of size $m$ our cost function is

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

The gradient of this is

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

This can be used to conduct gradient descent and approximate the minimum of the cost function.

The main drawback to this method is that the cost function sums over all points in the training set; doing this for every gradient calculation is really expensive. A common alternative method is *stochastic gradient descent*, in which only a single point is used in the cost function. Over many calculations, this still approximates calculating the gradients with all the points.

Estimating $\eta$ is not a precise science, but here are a couple popular methods:

- Momentum method: Take a weighted moving average of the values used.
- Acceleration: more complicated, outside the scope of this class
11. Logistic Regression

11.1 Introduction

We now transition from regression to classification problems. Logistic regression (despite the name) is a simple classification model that works surprisingly well and outputs a measure of confidence that can be interpreted as a classification. In this lecture, we define the logistic regression problem and explore how it can be solved.

11.2 0-1 Loss and Motivation for Logistic Regression

We begin with the naive classification strategy of learning a halfspace parametrized by a vector $w$ and outputing $\text{SIGN}(w^T x^i)$ for a given input. The possible labels are of course $y^i \in \{-1, 1\}$.

**Definition 11.1.** We define the 0-1 loss function as follows:

$$\varphi_{0-1}(z) = 1 \text{ if } z \leq 0, 0 \text{ otherwise}$$

A reasonable cost function in this classification scenario is just $\varphi_{0-1}(y^i \cdot w^T x^i)$, that is, only apply a penalty if the prediction and actual classification differ in sign. Thus, even if the margin assumption of the perceptron algorithm is not met, minimizing 0-1 loss does produce the best-fitting halfspace to the training set. 0-1 loss, however, is not used in practice because unlike squared loss, it is neither convex nor differentiable. In fact, minimizing 0-1 loss is NP-hard, so we need a better cost function for this classification task. One idea is to relax the 0-1 loss to come up with a nicer function (known as a surrogate loss), and doing so gives us the logistic regression problem.

11.3 Some Other Losses

**Definition 11.2.** Logistic and hinge losses are defined as follows:

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

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

Notice that logistic loss is very small if the input is very positive (classifier output is large and same sign as label) and very large if the prediction and class differ in sign greatly. Even hinge loss, though more linear, still penalizes correct classifications with outputs close to the margin (e.g. between 1 and 0). These relaxations make both functions convex and logistic loss differentiable.

11.4 Logistic Regression

**Definition 11.3.** The sigmoid function is defined as

$$g(z) = \frac{1}{1 + e^{-z}}$$
Examining the single-variable sigmoid function shows that as the input grows, the function asymptotically approaches 1 and as the input becomes smaller, it approaches 0. Thus, the output of \( g(w^Tx) \) can be interpreted as the probability that \( x \) has a label of 1. This brings us to the formal definition of the logistic regression problem.

**Definition 11.4.** The *logistic regression* model is a model of classification in which we make the assumption that

\[
\Pr[Y = y^i|x^i, w] = g(y^i \cdot w^T x^i)
\]

Thus, for a training set of size \( m \), the likelihood function that we are trying to maximize over \( w \) is

\[
Likelihood(w) = \prod_{i=1}^{m} g(y^i \cdot w^T x^i)
\]

\[
\text{Log-likelihood}(w) = -\sum_{i=1}^{m} \varphi_{\text{logistic}}(y^i \cdot w^T x^i) =
\]

Logistic regression can be solved with gradient descent by maximizing log-likelihood. Basic calculus shows us that:

\[
\varphi'_{\text{logistic}}(z) = -\frac{e^{-z}}{1 + e^{-z}} = -g(-z)
\]

\[
\frac{\partial \varphi_{\text{logistic}}(y \cdot w^T x)}{\partial w_k} = -g(-y \cdot w^T x) \cdot yx_k
\]
12. Neural Networks

12.1 Introduction

Neural networks are a method to learn really complicated (nonlinear) functions with numerous parameters and have been responsible for many breakthroughs in computer vision, natural language processing, and machine learning in general. We will first review some concepts from logistic regression that were covered in the previous lecture and expand upon them when discussing neural nets.

Recall that in logistic regression, we are trying to find the $w$ that best fits the equation $\Pr[Y = 1|X] = \sigma(w \cdot x)$, where $\sigma$ is the sigmoid function. To do this, we seek to minimize the logistic loss function using gradient descent. Since the loss function is convex, this procedure is well-behaved, and we converge to the absolute minimum.

The natural question is, how do we extend this procedure to more complex models, say $\Pr[Y = 1|X] = f(w \cdot x)$? For arbitrary models, the computational complexity is not easy to compute, so in practice we just run gradient descent and cross our fingers. In general, at least some of the layers in a neural network are chosen to be nonlinear since if they were all linear, their composition would just be a linear function. The whole point of neural networks is to be able to learn really subtle and complicated relationships between variables to achieve amazing feats like detecting tumors in x-ray scans and programmatically determining the genre of a piece of text. Additionally, any continuous function can be approximated by a composition of nonlinear functions.

The general method of learning neural networks is to first fix its architecture (depth, activation functions, number of nodes/layer, etc.) and use gradient descent to learn the parameters. This is what we’ll be exploring this lecture. Another less common method is to use test out many different architectures and use genetic algorithms to hone in on a good one, a technique known as neuroevolution. We will now explore a general technique to compute gradient of functions that can be represented as layered graphs.

12.2 Backpropagation

This method depends heavily on the chain rule, so review that if you need to. Backpropagation is a technique used in gradient descent to compute the partial derivatives of really complicated functions with respect to the parameters variables. Once these are calculated, the descent procedure then updates their values accordingly. The best way to understand backpropagation is to look at it from the perspective of a single node.

Example 12.1.
13. Boosting

13.1 Introduction

Boosting is a “wrapper technique” that can be used to improve the accuracy of weak learners, and is very popular in practice.

Before we explore boosting algorithms, recall the PAC learning model. For any choice of $\varepsilon, \delta$, our algorithm $\mathcal{A}$ must output an $\varepsilon$-accurate classifier with probability at least $1 - \delta$. Additionally, $\mathcal{A}$ must run in time polynomial with respect to $1/\delta, 1/\varepsilon$.

Example 13.1. Say we have some algorithm $\mathcal{A}$ that outputs an $\varepsilon$-accurate classifier with 5% probability. How do we output an accurate classifier with high probability?

Make $\mathcal{A}$ output $t$ classifiers, and over those $t$ classifiers, the probability that all are bad is $\leq 0.95^t$, which can be taken to some arbitrarily small $\delta$ by making $t$ large. After obtaining $t$ classifiers, just draw points from the distribution and use that to choose one of the $t$ classifiers. The exact math is an exercise left to the reader.

Example 13.2. Now, we have an algorithm $\mathcal{A}$ that outputs a 0.49-accurate classifier with any $\delta$ probability. How do we output a classifier that is $\varepsilon$-accurate for any $\varepsilon$? In other words, how to we amplify the accuracy of the model we output, using $\mathcal{A}$ as a subroutine, despite the 0.49 accuracy limitation?

This is a harder problem and was posed by Valiant in his original PAC-learning paper. The solutions presented are known as boosting algorithms. Today we’ll examine a particularly famous one known as AdaBoost, presented by Freund and Schapire in 1995.

13.2 AdaBoost Algorithm

Let us first explore the algorithm at a high level. Say we have the algorithm $\mathcal{A}$ from the previous example that outputs a 0.49-accurate classifier with $\delta$ probability. So, on a given set of labeled points drawn from distribution $D$, the classifier that is output gets 51% of the points right and 49% of them wrong. Intuitively, the 49% that were missed are “harder” to get right, so those points are given more weight, and those that were correctly classified are given less weight. We then rerun $\mathcal{A}$ on the re-weighted dataset and obtain a new classifier. We repeat this process a few times and output a classifier that takes a majority vote of the different classifiers we produced with $\mathcal{A}$.

Remark 13.3. It is important to note that for AdaBoost to work, the classifiers that $\mathcal{A}$ outputs must be whatever $\varepsilon$-accurate regardless of the distribution from which they are trained on. This is a pretty strong assumption to make.

The following is a simplified version of the AdaBoost algorithm. Let $m$ denote the size of the training set, $D_0$ be the initial distribution of points (uniform, corresponding to all weights being 1), $E$ be the error rate, $A = 1 - E$ be accuracy, and $\beta = E/A$ be the update factor. Each step, the error of the output classifier is written as $\varepsilon = 1 - \gamma$ for some $\gamma$. $\beta$ is update as $\beta = \frac{1/2 - \varepsilon}{1/2 + \varepsilon}$. Here is how we update the weights:

- At iteration $t$, run $\mathcal{A}$ to obtain classifier $h_t$. 
• For each point $x_i$, if $h_t(x_i)$ is correct, then $w_i^{\text{new}} = w_i \cdot \beta$. If it is wrong, then don’t change the weight.

• Repeat for $T$ steps, and output the classifier that predicts $\text{MAJ}(h_1, \ldots, h_T)$.

**Claim 13.4.** After $T$ iterations of the algorithm, the error of $h_{\text{final}} = \text{MAJ}(h_1, \ldots, h_T)$ is at most $e^{-2T\gamma^2}$. Thus, choosing $T \geq 1/\gamma^2 \log(1/\varepsilon)$ makes the error of $h_{\text{final}}$ at most $\varepsilon$, which is the goal.

**Proof:**

Let $w$ denote the sum of the weights of all the points in the dataset before iteration $t$. The sum of the weights of the correctly-labeled points is $(1/2 + \gamma) \cdot \beta \cdot w$ and the sum of the weights of the incorrectly-labeled points is $(1/2 - \gamma) \cdot w$. Thus, the new sum of all the weights is

$$(1/2 + \gamma) \cdot (1/2 - \gamma) + (1/2 - \gamma) \cdot w = w \cdot 2(1/2 - \gamma).$$

After $i$ iterations, the sum of the weights is $w(2(1/2 - \gamma)^i)$. Thus, after $T$ iterations, the sum of the weights is $w(2(1/2 - \gamma)^T)$. Now, if $h_{\text{final}}$ labels point $x_i$ incorrectly, that means the points was labeled incorrectly in at least half of the iterations of the algorithm. Thus, $w(x_i) \geq \beta^T/2$. So, if the error of $h_{\text{final}}$ is $\varepsilon$, then the sum of the weights of the misclassified points is at least $\varepsilon \cdot m \cdot \beta^T/2$. Since we have an upper bound on the total weight of all the points and a lower bound on the weights of the incorrectly-classified points, we derive the inequality

$$\varepsilon \cdot m \cdot \beta^{T/2} \leq w(2(1/2 - \gamma)^T)$$

Do some math:

$$\varepsilon \cdot m \cdot \beta^{T/2} \leq w(2(1/2 - \gamma)^T) \cdot m$$

$$\varepsilon \leq \left( \frac{4E^2}{\beta} \right)^{T/2}$$

$$\varepsilon \leq (1 - 4\gamma^2)^{T/2}$$

$$\varepsilon \leq e^{-2\gamma^2 T} \quad \square$$

To make sure that the classifier output by AdaBoost generalizes, choose a sufficiently large training set.

### 13.3 Hedge Algorithm

This is a more general algorithm presented in the same paper that introduced AdaBoost, of which AdaBoost is in fact a specific application. This more general algorithm is called the *hedge algorithm*. It is reminiscent of the “best experts” setup: we have $k$ classifiers $c_1, \ldots, c_k$. 
14. Confidence Intervals

14.1 Introduction

Previously we discussed cross-validation as a useful tool for estimating generalization error but also mentioned that it does not really provide any statistical guarantees about its estimates. Today, we focus on calculating confidence intervals for our training error so that we have some probabilistic guarantees on the generalization error.

14.2 Statistical Preliminaries

Definition 14.1. A Bernoulli random variable \( X \) is one which takes on the value 1 with probability \( p \) and 0 with probability \( 1 - p \). \( \mathbb{E}[X] = p \) and \( \text{Var}[X] = p(1 - p) \).

Theorem 14.2 (Informal Central Limit Theorem). The sum of independent, identically-distributed (i.i.d) random variables behaves like a Gaussian distribution.

Notice that the outcomes of a classifier \( h \) on a dataset can be modeled with Bernoulli random variables, where \( Y_i = 1 \) if \( h(x_i) \) is classified incorrectly and 0 otherwise. Each \( Y_i \) is a Bernoulli random variable with \( p = \) true error rate of \( h \). This motivates an obvious procedure for estimating test error.

- Draw a random sample \( S \) of labeled data.
- Create random variables \( Y_i \) as above where \( \mathbb{E}[Y_i] = p = \) true error rate.
- Define our estimate of the true error rate, \( \hat{p} = 1/n \sum_{i=1}^{n} Y_i \).

Now the big question is, how large of an interval should I draw around \( \hat{p} \) so that I’m pretty confident (say 95%) that \( p \) lies in this interval? To do that, let’s first calculate \( \mathbb{E}[\hat{p}] \) and \( \text{Var}(\hat{p}) \):

- \( \mathbb{E}[\hat{p}] = \mathbb{E}[1/n \sum_{i=1}^{n} Y_i] = 1/n \cdot np = p \)
- \( \text{Var}(\hat{p}) = \text{Var}(1/n \sum_{i=1}^{n} Y_i) = 1/n^2 \cdot np(1-p) = p(1-p)/n. \)

Definition 14.3. The \textit{z-score} of a value \( x \) drawn from a distribution of mean \( \mu \) and standard deviation \( \sigma \) is

\[
z = \frac{x - \mu}{\sigma}
\]

Intuitively, it measures how many standard deviations to the right or left of the mean point \( x \) is.

We can then write the equation \( z = \frac{x - \hat{p}}{\sigma} \) and choose a particular \( z \) that corresponds to the level of confidence we want from a table of z-scores and then solve for an interval (shown in an example below). The main issue is that \( \sigma \) depends on \( p \), which is what we’re trying to solve for. We crudely use \( \hat{p} \) in this computation. In reality, another test (the t-test) exists to take this into account, but we ignore it in this lecture for simplicity.
Example 14.4. Suppose we draw a sample $S$ of size 1000 and that we computed $\hat{p} = 0.75$. Compute a confidence interval with confidence level 80%.

$$z = \frac{\hat{p} - p}{\sigma} \text{ cheating a bit by using } \hat{\sigma}$$

$$z = \frac{\hat{p} - p}{\sqrt{\hat{p}(1-\hat{p})/|S|}}$$

$$1.3 = \frac{0.75 - p}{0.013}$$

$$0.75 - p = 0.0169$$

From this we conclude that the confidence interval is $0.75 \pm 0.0169$. 

$31$
15. Random Projection

15.1 Introduction

One big problem in machine learning is that the data we have to analyze is very high-dimensional, that is, it contains numerous features (e.g. robot sensor data). Some algorithms do not scale very well as the number of dimensions in the data increases, so in this lecture we discuss ways to decrease the dimensionality of data.

Example 15.1. In the nearest-neighbor problem, we are given a set of points and asked to calculate the closest point to a given point in the set (Euclidean distance). The naive solution is to search through all the points, but that is very inefficient ($O(n)$) and unacceptable if we have a large set and need to make numerous queries.

A more sophisticated approach that has been developed recursively splits Euclidean space into 2 (by a halfspace) and stores this information in a structure known as a k-d tree. In a k-d tree, nodes are points and subtrees are subspaces that lie on one side or the other of an implicit hyperplane represented by the point with respect to some axis (i.e. if we’re splitting across the x-axis, points with an x-value smaller than the node would be on the left and the rest would be on the right). Using this data structure, the algorithm works as follows to find the nearest neighbor to point $p$:

- Traverse the k-d tree, maintaining the closes visited node to $p$ as you go down the tree.
- At each level, create a hypersphere around $p$ with radius equal to the distance from the current closest point. If that hypersphere intersects the hyperplane at the current level, the other subtree of the current node must also be searched.
- Recurse back up the tree and return the calculated minimum.

The number of splits increases exponentially as the number of dimensions of the data increases (the so-called curse of dimensionality). Thus, even the improved algorithm above does not perform very well in higher dimensions. We need a way to decrease the dimensionality of our data.

At a high level, the techniques we come up with allow us to reduce the dimensionality of the data at the cost of only coming up with an approximate solution to nearest-neighbors (that can be made arbitrarily accurate).

15.2 Hashing

Definition 15.2. Let $\mathcal{U}$ be a universe of points that we want to hash and $\mathcal{H}$ be a family of hash functions $h: \mathcal{U} \rightarrow [m]$. The family is universal if $\forall h \in \mathcal{H}, \forall x \neq y \in \mathcal{U}, h(x) - h(y) \mod m$ is uniformly distributed over $m$.

Example 15.3. If $h: \mathcal{U} \rightarrow [m]$ is a universal hash function, then we conclude that

$$\forall x \neq y \Pr[h(x) \equiv h(y) \mod 2] = 1/2$$

If we sample $k$ hash functions $h_1, ..., h_k$ from a universal family $\mathcal{H}$, then we conclude that

$$\forall x \neq y \Pr[h_1(x) \equiv h_1(y) \mod 2, ..., h_k(x) \equiv h_k(y) \mod 2] = 2^{-k}$$
Thus, for any arbitrarily small $\delta > 0$, we can choose $k = \log(1/\delta)$ to be sure that the compound hash function $h_1, ..., h_k$ preserves distinctness with at least $\delta$ probability.

To tackle the dimensionality problem, we apply a variant of the hashing concept. For some $x \in \mathbb{R}^n$, we draw a vector $r \in \mathbb{R}^n$ where each component is drawn from the standard normal distribution $\mathcal{N}(0, 1)$. We define the function $f_r(x) = \langle x, r \rangle$. Let’s now compute a few quantities:

$$
E_r[f_r(x) - f_r(y)] = E_r[(x - y, r)] = E[\sum_{i=1}^{n} (x_i - y_i)r_i] = \sum_{i=1}^{n} (x_i - y_i)E[r_i] = \sum_{i=1}^{n} (x_i - y_i) \cdot 0 = 0
$$

The last couple lines hold since the expected value of the standard normal distribution is 0. Similar calculations for the variance yield the following results:

$$
\text{Var}[f_r(x) - f_r(y)] = \|x - y\|^2
$$
$$
E[(f_r(x) - f_r(y))^2] = \|x - y\|^2
$$

The above results show that the quantity is $(f_r(x) - f_r(y))^2$ is an unbiased estimator of the Euclidean distance between the points $x$ and $y$. Sampling multiple times, specifically $k = O(\log m/\epsilon)$, and taking the averages of the calculations gives a Euclidean distance that is within $\epsilon$ of the true value with high probability. We have thus found a way to reduce the dimensionality of a point in $\mathbb{R}^n$ for use in nearest neighbors, namely $(\langle x, r_1 \rangle, ..., \langle x, r_k \rangle)$, where $k$ is significantly smaller than $n$ for large $n$. This technique can be generalized and is known as the Johnson-Lindenstrauss lemma, or JL lemma, which is outlined below. You could read the paper written in 1984 (with the approval of Big Brother, of course), but it’s typewritten and thus an eyesore.

Say we’re working in $\mathbb{R}^n$. Choose an appropriate $k = O(\log m/\epsilon)$ and create a $k \times n$ matrix $A$ with each entry being drawn from either $\mathcal{N}(0, 1)$ or $\{-1, 1\}$. We can reduce the dimensionality of some $x \in \mathbb{R}^n$ using the transformation $f_A(x) = \frac{1}{\sqrt{k}} A \cdot x$. The quantity $\|f_A(x) - f_A(y)\|^2$ approximates squared Euclidean distance between $x$ and $y$ within $\epsilon$ with high probability.

Now we discuss an interesting application of the JL transform to unsupervised learning

**Example 15.4.** The Gaussian mixture model is one in which we draw points from 2 Gaussian distributions $\mathcal{N}(\mu_1, \Sigma_1)$ and $\mathcal{N}(\mu_2, \Sigma_2)$ and with probabilities $p$ and $1 - p$ respectively. Given many draws from this distribution, we want to recover the parameters $p, \mu_1, \mu_2, \Sigma_1, \Sigma_2$.

Since points drawn cluster near the means of the 2 normal distributions, we use the JL transform to efficiently find the points with the most nearest neighbors and use those to estimate the means of the Gaussian distributions (Dasgupta ’99). The rest is beyond the scope of this class.
Example 15.5. We have documents and want to measure how similar they are. We want to reduce their dimensionality while preserving Jaccard similarity, which is defined as follows, taking \( \mathcal{U} \) to be the universe of words and \( A \) and \( B \) to be the sets of words in each of 2 documents:

\[
J = \frac{|A \cap B|}{|A \cup B|}
\]

For convenience, we define the function \( J : \mathcal{P}(\mathcal{U}) \times \mathcal{P}(\mathcal{U}) \rightarrow \mathbb{R} \) that takes 2 documents and outputs their Jaccard similarity.

The idea is to first assign each word in \( \mathcal{U} \) with a distinct random number from 1 to \( |\mathcal{U}| \) and then randomly permute these mappings using a permutation function \( \pi \). We define the function that takes in a document \( S \subseteq \mathcal{U} \) and outputs an integer as \( f(S) = \min_{x \in S} \pi(x) \). We claim that for any 2 documents \( A \) and \( B \), \( \Pr[f(A) = f(B)] = J(A, B) \). This is trivially true since the only possible words that could be mapped to the same number are those which exist in both documents and live in \( A \cap B \). Basic reasoning from here proves the claim.

We now create a random variable \( X \) that takes the value 1 if \( A, B \) are mapped to the same value and 0 if they are not. \( \mathbb{E}[X] = J(A, B) \). If we choose \( k \) permutations randomly (with \( k \) chosen intelligently as explained earlier) and use them to calculate the value of \( X \) \( k \) times, their average approximates the Jaccard similarity of the documents accurately and in a lower-dimensional space. This general technique of quickly estimating set similarity is known as *min-hashing*. 
16. Principal Component Analysis

16.1 Introduction

PCA is one of the most popular dimensionality reduction techniques used in practice and has a variety of advantages over the technique we looked at previously, random projection. The drawbacks of random projection are as follows:

- Each random vector chosen from the normal distribution has no special meaning with respect to the original data. PCA produces vectors that directly depend on the original data (no random draws).
- In practice, vector length \( \leq 100 \) for good results. With PCA, good results are achieved with much smaller results.

16.2 PCA

Before applying PCA to a dataset, we do some preprocessing:

- Subtract the mean from each point.
- Divide each feature of the point by the standard deviation of that feature.

At a high level, the goal of PCA is to find a set of unit vectors such that the original dataset can be expressed as a linear combination of these vectors. The first vector we choose is the one that minimizes the sum of the squared distances (divided by size of the training set) from all the vectors in the dataset. For a training set of size \( m \), this can be expressed concisely as

\[
\arg \min_{v, \|v\|=1} \frac{1}{m} \sum_{j=1}^{m} d(x^j, v)^2
\]

For a given training vector \( x \) and arbitrary unit vector \( v \), we see from the Pythagorean theorem that

\[
\langle x, v \rangle^2 + d(x, v)^2 = \|x\|^2
\]

Since \( \|x\| \) is a fixed quantity, minimizing the squared distance is equivalent to maximizing \( \langle x, v \rangle^2 \). More concisely, we want to find

\[
\arg \max_{v, \|v\|=1} \frac{1}{m} \sum_{j=1}^{m} \langle x^j, v \rangle^2
\]

This task is known as determining the direction of maximum variance, as projecting the data onto \( v \) preserves the general structure of the data. To generalize this task to \( k \) orthonormal basis vectors, we conduct the following minimization, known as the PCA objective:

\[
\arg \max_{v_1, \ldots, v_k \text{ orthonormal}} \frac{1}{m} \sum_{j=1}^{m} \sum_{i=1}^{k} \langle x^j, v^i \rangle^2
\]
Now, suppose we used some PCA method to obtain orthonormal basis vectors \( v_1, \ldots, v_k \). How do we determine the coefficients of these vectors when expressing some data point \( x \) as a linear combination of these vectors? Say the coefficients are unknown \( c_1, \ldots, c_k \), so we have

\[
x = c_1 v_1 + \ldots + c_k v_k
\]

Take the inner product of both sides with \( v_i \) for any \( i \in [k] \).

\[
\langle x, v_i \rangle = \langle c_1 v_1 + \ldots + c_k v_k, v_i \rangle \\
= \langle c_1 v_1, v_i \rangle + \ldots + \langle c_i v_i, v_i \rangle + \ldots + \langle c_k v_k, v_i \rangle \\
= c_i \cdot 1 \\
= c_i
\]

The eliminations above are possible since the basis is orthonormal. It is pretty obvious that each coefficient is simply the inner product of that basis vector with \( x \), giving

\[
x = \langle x, v_1 \rangle v_1 + \ldots + \langle x, v_k \rangle v_k
\]

16.3 Calculating Principal Components

16.3.1 Some Linear Algebra Preliminaries

Definition 16.1. A vector \( v \) by itself denotes a column vector and thus \( v^T \) represents a row vector. \( v^T v \) is an inner product and produces a scalar, and \( vv^T \) is an outer product and produces a matrix.

Definition 16.2. The outer product of 2 vectors \( u \) and \( v \) is the matrix \((uv^T)_{ij} = u_i v_j\).

Definition 16.3. An orthogonal matrix \( A \) is a matrix with orthonormal columns, so \( AA^T = A^T A = I \).

Proposition 16.4. All eigenvalues of symmetric matrices are non-negative.

Theorem 16.5 (Spectral Theorem). Every symmetric matrix \( A \) has an eigendecomposition \( A = QDQ^T \) where \( Q \) is orthogonal and \( D \) is a diagonal matrix containing the eigenvalues of \( A \).

16.3.2 Some Basic Calculations

We first start with the simple of task of a calculating a single principal component of a dataset with pretty strong assumptions. Next lecture we tackle the general case.

Let \( X \) denote an \( m \times n \) matrix of \( m \) data points in \( \mathbb{R}^n \), and let \( v \) be the first principal component of \( X \) that we’re trying to solve. Notice that multiplying out \( Xv \) produces a column vector of the form \( (Xv)_i = \langle X_i, v \rangle \). From this we see that

\[
\langle (Xv)^T, Xv \rangle = v^T (X^T X) v = \frac{1}{m} \sum_{i=1}^{m} \langle X_i, v \rangle^2
\]

The resulting expression is precisely what we want to maximize. Letting \( X^T X \) be some matrix \( A \), the task of finding \( v \) to maximize \( v^T Av \) is known as maximizing a quadratic form. Our problem has been simplified even further to the following task:

\[
\arg \max_{v, \|v\|=1} v^T Av, A = X^T X
\]

36
In the case where $A$ is a diagonal matrix with diagonal entries $d_1, ..., d_n$ sorted by decreasing value, the expression we are trying to maximize simplifies to $\sum_{i=1}^{n} v_i^2 d_i$. The solution to the maximization is then $v = (1, 0, 0, ..., 0)^T$ since that is the vector with norm 1 that puts the most weight on the largest diagonal entry $d_1$. The issue is now that $X^T X$ is not necessarily diagonal. However, by the spectral theorem it can be written in the form $X^T X = QDQ^T$, where $D$ is diagonal. In this case, we choose $v = Qe_1$ where $e_1 = (1, 0, ..., 0)^T$ to solve the maximization.
17. Principal Component Analysis, Part 2

17.1 Recap

Last lecture we simplified the principal component problem to solving the following maximization:

$$\arg \max_v v^T Av, A = X^T X$$

$$v, \|v\| = 1$$

We also discussed last week that $X^T X$ is a covariance matrix. It turns out that any covariance matrix can be represented as the composition of a diagonal operator with some rotations. The following example illustrates this.

**Example 17.1.** Consider the following matrix representing a linear transformation $\mathbb{R}^2 \rightarrow \mathbb{R}^2$:

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

The above matrix can be decomposed as follows:

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

The first matrix in the decomposition is a rotation $45^\circ$ counterclockwise, the middle is a horizontal stretch, and the right is a rotation $45^\circ$ clockwise.

17.2 Full PCA Procedure

We now present the full PCA procedure and then prove that it works. Say we have an $m \times n$ matrix $X$ representing the $m$ vectors in our training set whose principal components we want to find.

1. First normalize the data (subtract mean and divide each feature by the standard deviation of that feature).

2. Compute the covariance matrix $A = X^T X$. Then compute the eigendecomposition $A = QDQ^T$, where $Q, Q^T$ are orthogonal inverses and $D$ is diagonal.

3. The first $k$ rows of $Q^T$ are eigenvectors for the first $k$ eigenvalues of $A$ (diagonal entries $\lambda_i$ in $D$) and the top $k$ principal components of the dataset.

**Claim 17.2.** The $i^{th}$ row of $Q^T$ is an eigenvector of $A$.

**Proof:**

The $i^{th}$ row of $Q^T$ equals $Qe_i$, where $e_i$ is the $i^{th}$ standard basis vector of $\mathbb{R}^n$. Since $Q$ is orthogonal, its transpose is also its inverse so setting $v = Qe_i$ we see that

$$Av = QDQ^T(Qe_i) = QD(Q^TQ)e_i = QDe_i = Q(0, ..., \lambda_i, ..., 0) = (Q_1\lambda_i, ..., Q_n\lambda_i) = \lambda_i Q^T = \lambda_i v$$
Now the question is, how do we compute this eigendecomposition in the first place? One method is to compute the singular value decomposition (SVD), but that algorithm is computationally expensive. We instead develop a more efficient method known as the “power method” using geometric intuition.

The idea is that if one eigenvalue is much larger than the others, then multiplying a vector by \( A \) will move the vector disproportionately in the direction of the corresponding vector since an eigenvalue represents the scaling of a vector along a fixed line (the eigenspace). Applying \( A \) to a random vector repeatedly will eventually converge on to an eigenvector associated with the largest eigenvalue of \( A \).

More formally, we begin with a randomly chosen unit vector \( u_0 \) and keep applying \( A \) to it. Let \( u_i \) be the result of applying \( A \) to \( u_0 \) \( i \) times, so \( u_i = Au_{i-1} \). Keep applying \( A \) until \( u_i/\|u_i\| \approx u_{i-1}/\|u_{i-1}\| \) and then output \( u_i \) as the approximation of an eigenvector of the largest eigenvalue of \( A \). We now prove that this method works.

**Claim 17.3.** Let \( A = QDQ^T \) denote the eigendecomposition of \( A \). Then \( A^i = QD^iQ^T \).

**Proof:**

Just apply a simple case of induction. For \( i = 1 \) the statement is trivially true. Assume for some \( k > i \), \( A^k = QD^kQ^T \). Then \( A^{k+1} = A^kA = QD^k(Q^TQ)DQ^T = QD^kDQ^T = QD^{k+1}Q^T \). \( \square \)

**Theorem 17.4.** Let \( v_1, ..., v_n \) be eigenvectors associated with eigenvalues \( \lambda_1, ..., \lambda_n \) ordered by decreasing value. Borrowing the notation from the last claim,

\[
\langle u_t/\|u_t\|, v_1 \rangle \geq 1 - 2n \left( \frac{\lambda_2}{\lambda_1} \right)^t
\]

**Proof:**

We use the fact that if \( u_0 \) is chosen at random, when it is written as a linear combination of the eigenvectors \( u_0 = \sum_{i=1}^n c_i \), with high probability \( c_1 \geq 1/2\sqrt{n} \).
$$\langle u_t/\|u_t\|, v_1 \rangle = \frac{c_1 \lambda_1^t}{\sqrt{\sum_{i=1}^{n} (\lambda_i^t c_i)^2}}$$  \hspace{1cm} (17.2.1)

$$\geq \frac{c_1 \lambda_1^t}{\sqrt{\lambda_2^t c_1^2 + n \lambda_2^t}}$$  \hspace{1cm} (17.2.2)

$$\geq \frac{1}{1 + \frac{\lambda_2^t \sqrt{n}}{c_1 \lambda_1^t}}$$  \hspace{1cm} (17.2.3)

$$\geq \frac{1}{1 + \frac{\lambda_2^t \sqrt{n}}{\sqrt[n]{n} \lambda_1^t}}$$  \hspace{1cm} (17.2.4)

$$= \frac{1}{1 + 2n(\frac{\lambda_2^t}{\lambda_1^t})^t}$$  \hspace{1cm} (17.2.5)

$$\approx 1 - 2n \left(\frac{\lambda_2}{\lambda_1}\right)^t$$  \hspace{1cm} (17.2.6)

□

To calculate an eigenvector associated with the second largest eigenvalue $\lambda_2$, run the power method with a new dataset $X'$ in which each data row has its projection onto $v_1$ subtracted from it ($x := x - \langle x, v_1 \rangle v_1$).
18. Singular Value Decomposition

18.1 Introduction

We begin this lecture with a motivational problem, the Netflix challenge. We are given a massive matrix with rows representing individual Netflix users and columns representing movies. Each entry $A_{ij}$ represents the rating person $i$ gives movie $j$. Since not every user has seen and rated every movie, the matrix is sparse. Our task is to intelligently complete the matrix.

18.2 Matrix Completion

Definition 18.1. The singular value decomposition of an $m \times n$ matrix $A$ is its expression in the form $A = USV^T$, where $U$ and $V$ are square orthogonal matrices and $S$ is diagonal with entries decreasing in value down the diagonal. The columns of $U$ are called the left singular vectors and the rows of $V^T$ are the right singular vectors. The diagonal entries of $S$ are called singular values.

Definition 18.2. The Frobenius norm of a matrix is defined as

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

The SVD can be used to compress tabular data owing to the fact that many times the first few singular values are much larger than the rest. If we only keep the first $k$ singular values, we can express a “compressed” SVD where we reduce $S$ to a $k \times k$ matrix and only keep the first $k$ columns of $A$ and $k$ rows of $V^T$. In fact, it can be shown that the rank-$k$ approximation that minimizes the Frobenius norm of the difference from the original matrix is the “compressed” SVD keeping $k$ singular values.

Now we present a solution to the matrix completion problem.

- Fill the missing entries in the original with 0s or the average of the present entries in that row.
- Output rank-$k$ approximation using SVD.

18.3 Linear Regression

In the linear regression problem, we have an $m \times n$ matrix $A$ of $n$ data points with $m$ features and an output vector $b \in \mathbb{R}^m$. We seek to solve for the coefficients in the linear model that minimizes squared loss, namely

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

In the linear regression lecture, we assumed that $A$ was a square matrix when solving for $x$. Now we find a solution in the general case using the SVD $A = USV^T$.

$$\|Ax - b\|^2 = \|USV^T x - b\|^2$$

$$= \| (U^T U )SV^T x - U^T b \|^2$$

$$= \|SV^T x - U^T b \|^2$$

$$= \|Sy - U^T b \|^2 \text{ variable substitution } y = V^T x$$

41
The value of $y$ that minimizes the final expression is $y = S^\dagger U^T b$. Solving for $x$ yields $x = (V^T)^{-1}S^\dagger U^T b = VS^\dagger U^T b$ since $V$ is orthogonal.

18.4 PCA

In PCA, we find the principal components by calculating the eigendecomposition of $X^T X$ given a dataset $X$. Calculating the SVD of $X$ yields $USV^T$ and substituting this yields $X^T X = (USV^T)^T \cdot USV^T = VS(U^T U)SV^T = VS^2 V^T$. The singular values of $X$ are the square root of the eigenvalues of $X^T X$ and the right singular vectors of $X$ are the principal components.
19. Latent Semantic Index

19.1 Latent Semantic Indexing

Latent Semantic Indexing (LSI) is a popular technique in text data mining and information retrieval that consists of the following steps, at a high level:

- Start with a collection of documents.
- Choose a subset of the words (terms) and pick those out in each document.
- Create a term-document matrix, which can be used to search queries and do other stuff which we’ll get to.

Example 19.1. We are given the following 5 documents:

- \(d_1\): Romeo and Juliet
- \(d_2\): Juliet: O happy dagger!
- \(d_3\): Romeo died by dagger.
- \(d_4\): “Live free or die”. That’s the New Hampshire motto.
- \(d_5\): Did you know New Hampshire is in New England?

We are given the query: \(dies, dagger\) and would like to return the document that is most relevant.

Just by examining the documents, it is clear that the best match is \(d_3\) and the second best match is \(d_2\), even though \(d_4\) mentions death as well. We need to develop a technique that helps conceptually separate the different documents to return accurate query results.

Definition 19.2. A term-document matrix is a matrix \(A\) in which each row corresponds to a term and columns correspond to documents. \(A_{ij} = 1\) if word \(i\) is in document \(j\) and \(0\) otherwise.

Example 19.3. The term-document matrix from the previous example is as follows:

<table>
<thead>
<tr>
<th>Term</th>
<th>(d_1)</th>
<th>(d_2)</th>
<th>(d_3)</th>
<th>(d_4)</th>
<th>(d_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Romeo</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Juliet</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Happy</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dagger</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Live</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Die</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>New Hampshire</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We denote this matrix \(A\), \(A^T A\) to be the document-document matrix, and \(AA^T\) to be the term-term matrix. Note that \(A^T A_{ij}\) is the number of words documents \(i\) and \(j\) have in common, and \(AA^T_{ij}\) is the number of common documents of terms \(i\) and \(j\).
Recall that the SVD of matrix $A$ is $A = U \Sigma V^T$, and in our case, the diagonal entries of $\Sigma$ are 2.285, 2.02, 1.36, 1.12, and 0.797. We just keep the first 2 eigenvalues to make a $2 \times 2$ approximation $\Sigma'$, and come up with two 8-dimensional left-singular column vectors and two 5-dimensional right-singular row vectors.

We can now represent each term as $i$ a 2-dimensional vector taking the inner product of $A_i$ with each right-singular vector. Another way is to multiply the $i^{th}$ number in the first left-singular vector with the first singular value and do the same with the second left-singular vector and second singular value. We can similarly calculate compressed representations of each document vector (just swap the words “left” and “right” in the previous explanation).

We now have a compact representation of both documents and terms. We can represent the query as the average of the 2 words and calculate the *cosine similarity* between the query and each document (i.e. the angle between the vectors). The cosine similarity between 2 vectors is calculated as

$$\cos \theta = \frac{x \cdot y}{\|x\|\|y\|}$$

### 19.2 K-Means Clustering

In the k-means clustering problem, we seek to take a set of data points and partition them into some chosen number of clusters $k$. Each of these clusters has a center $\mu$, and we seek to minimize the sum of the squared distances from each point to its center. More formally, if $C : [n] \rightarrow \{1, \ldots, k\}$ is a particular clustering and $\mu_1, \ldots, \mu_k$ represents a particular assignment of centers, we seek to minimize the following cost function

$$\min_{C, \mu_1, \ldots, \mu_k} \sum_{i=1}^{k} \sum_{j, C(j) = i} \|\mu_i - x_j\|^2$$

Unfortunately, this problem is NP-hard to even approximate, but there are still good heuristics to estimate the clusterings and their centers. The following is one such strategy:

- Initialize $\mu_1, \ldots, \mu_k$ to random points.
- Assign each data point to nearest $\mu_i$. We already discussed ways to make this more efficient (e.g. random projection).
- Update each $\mu_i$ to the center of mass of all the points assigned to it.
- Repeat.

It can be shown that the above algorithm never increases the cost with each iteration. Other more sophisticated clustering techniques use SVDs (spectral clustering).
20. Learning with Noise

20.1 Introduction

So far, we have assumed that all the inputs that our classifiers are given are properly labeled without any mistakes. In this lecture, we explore the idea of learning noisy data, provide some examples of classifiers that can handle noise, and prove some of their interesting properties. Let’s begin with an example.

Example 20.1. Our goal is to learn \( C = \{ \text{parity functions} \} \), where parity functions are functions mapping \( \{0, 1\}^n \rightarrow \{0, 1\} \) and are computed by taking the XOR of some subset of bits (e.g. \( f(x) = x_1 \oplus x_2 \oplus x_3 \)). Obviously \( |C| = 2^n \).

One insight we can use is that the XOR function is the same as adding mod 2 (e.g. \( x_1 \oplus x_2 \oplus x_3 \) is the same as \( x_1 + x_2 + x_3 \mod 2 \)). Thus, we can come up with a function that is consistent with the training set by using Gaussian elimination. If we want to PAC-learn \( C \), we require about \( \log |C|/\varepsilon = O(n/\varepsilon) \) training examples.

Now let’s consider a harder problem, in which a fraction of the labels are flipped. This sort of noise is known as random classification noise, and the best known algorithm to learn parity functions that deal with this noise run in \( O(2^O(n/\log n)) \) time. Although parity functions don’t deal with this noise very well, many other function classes can.

20.2 The Statistical Query Model

Definition 20.2. In the statistical query model, the learner specifies a function \( q : X \times \{-1, 1\} \rightarrow \{-1, 1\} \) and a tolerance parameter \( \tau \) and receives in return \( \mathbb{E}_{x \sim D}[q(x, c(x))] \pm \tau \) from an oracle, where \( c \) is some unknown concept.

Intuitively, \( q \) is a statistical property of the data and \( \tau \) is the tolerance of this read. In the PAC-learning model, statistical queries can be simulated by evaluating \( q \) on draws from a labeled training set, with the number of draws determined by \( \tau \) and the Chernoff bound. Thus, the SQ model is strictly more restrictive than PAC.

Theorem 20.3. Any function class that can be learned with statistical queries without noise can be PAC-learned with random classification noise.

Example 20.4. Our goal is to learn PAC-learn monotone conjunctions using statistical queries. We need to first specify \( q \) and \( \tau \). We choose the following:

- \( q_i(x) = (x_i = 0) \land (c(x) = 1) \), defined for each \( x_i \)

- \( \tau = \frac{\varepsilon}{2n} \).

Proof:
If a particular literal is in the conjunction, then it is impossible for the expression to evaluate to 1 if the literal is 0. Thus, a literal in the true conjunction will never be removed.

The only mistake that can be made is if a literal not in the conjunction is falsely not removed.
21. Agnostic Learning

21.1 Introduction