Machine Learning

Introduction

The input to a learning algorithm is training data, representing experience, and the output is some expertise, which usually takes the form of another computer program that can perform some task. Memorization of training data is quite different from learning because memorization can't label unseen data and can't generalize, also known as inductive reasoning or inductive inference.

In training, the learner is requested to output a *prediction rule, also known as a* predictor*, a* hypothesis*, or a* classifier*. The learning paradigm–coming up with a predictor h that minimizes $L_S(h)$–is called the* Empirical Risk Minimization* or ERM for short. This is also known as **empirical error** and **emperical risk**.

Mistake-Bounded Model of Learning

MB Learning guarantees an ironclad contract that defines how many errors will occur over a set of samples. The model makes a prediction, and if correct, moves on; otherwise, if it predicts wrong, it learns. **It is said a learner has mistake-bound t if for every sequence of challenges, the learner makes at most t mistakes.** A **monotone dijunction** is is essentially an *ORing* of the variables and monotonic because there aren't any negations. For monotone disjunctions, you'll have at most n mistakes but for monotone disjunctions, you can have up to $2n$ disjunctions because you negate each variable and append it as the initial string. Each expansion is $y_i = \overline{x_i}$. The process of doubling the string initially is called **feature expansion**.

Initial monotone disjuction: x_1 or x_2 or x_n. Without monotone but generic disjunctions, we double input size and therefore the MB will be $2n$ since we can have $2n$ possible mistakes.

On-Line Algorithms

The areas of On-Line Algorithms and Machine Learning are both concerned with problems of making decisions about the present based on knowledge of the past.

Decision Trees

The size of the decision tree is the number of nodes in the tree. The depth of the tree is the length of the longest path from root to leaf. **The $\phi(x)$ function is the error rate.**

Decision Trees are a powerful way of classifying data that possess a lot of useful theoretical properties. A decision tree is a boolean function, and in this segment, we're only considering boolean trees and ignorning decision trees with more than two outputs.

Tree Input: $x \in \{0,1\}^n$ with the tree encoding a function $f(x) \to \{0,1\}$.

We navigate the tree by considering, for example, x_4 and traversing it down the tree depending on the binary value. If we reach a leaf node, we output the resulting boolean value. The size of the decision tree will be the number of nodes, and height of the tree will be the length of the longest path from the root to a leaf.

Building a Tree

Given a set of labeled examples, build a tree with low error. Suppose we have a training set $S = \{(x^1, y^1), ..., (x^m, y^m)\}$ where m is the number of traning samples and $y^i \in \{0,1\}$ and $x^i \in \{0,1\}^n$. The **error rate**, also known as **training error** or **empirical error rate**, is equal to $\frac{\text{\# of mistakes T makes on S}}{|S|}$, where T is the decision tree.

In the case of a single-leaf tree, it's prudent to pick the *mode* of S as the output. If we had a potential function $\phi(a) = min(a, 1-a)$, then with 5 positive and 10 negative examples, the error rate would be

$$\phi(Pr_{(x,y)\sim S}(y=0)) = \phi(\frac{1}{3}) = min(\frac{1}{3}, \frac{2}{3}) = \frac{1}{3}$$

. We would have the same error rate if we picked 10 positive and 5 negative examples.

Now, expand a trivial tree to a deeper tree. Whenever you pick a tree branch, say $x_1 = 0$, you exclude all $x_1 = 1$ points on that branch. After this decision is made, you condition on those x's from the training set. The gain function, the decrease in training error, of a chosen value, say x_1, is

$$\text{GAIN}(x_1) = \text{OLD RATE} - \text{NEW ERROR RATE USING} X_1$$

. The new error rate is

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

. We then pick the literal x_* that maximizes the gain function and repeat the process.

Pick the smallest computed ϕ value. This means that the the gain is maximized.

Structure of Tree

The structure of the tree is determined by the choice of ϕ, which corresponded to the training error earlier. Given $\phi(a) = 2 \cdot a \cdot (1-a)$ (Gini Function/Index), it's a dome that's an upper bound for $\phi(a) = min(a, 1-a)$. The difference is the smoother function gives us more useful theoretical properties.

Random Forest Termination Criterion

- easy option: stop when the gain is extremely small for all literals
- better optimized: build an enormous tree and prune trees from the bottom up that have the least overall effect on the decision tree, retain up to n nodes. This is called the random forest.
- random trees: build many small decision trees
  1. randomly subsample from training set S to S'
  2. randomly pick some features from $x_1, ..., x_k$ of size k
  3. build a decision tree using S' and the k random forests

Generalization

A common question what is the **true error** or **generalization error** of a classifier? As an example, a decision tree: for T, D, the probability

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

where x is the challenge and y is the label. Given a training sample S, you can build a decision tree whose size is $size \geq |S|$ that is consistent with all the points in S–this would be bad because you're memorizing your data. You not only have to worry about a low training error but are you getting a true low generalization errror/good predictive power?

One common way to measure the "true error of a classifier" is to use a hold-out or validation set. The problem is that this validation set should only be used once since it's easy to indirectly incoperate the hold-out into the training, which is called overfitting. We'll need a better technique such as cross-validation.

Generalization

Another approach is to trade-off training error with "model complexity". Define another potential function:

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

, where α is a hyperparameter. We'd like to minimize ϕ. T is the tree hypothesis.

Another common approach is the **Minimum Description Length Principle**(MDL), which involves encoding the number of bits involved. You can imagine the upper-bound $m \cdot (n+1)$, where m is the number of samples in the training set and n for the sample and 1 for the label. You could then encode S and the number of bits in $bits(T)$ for the generated tree plus the number of bits to encoded the 10% we got wrong, assuming our tree T is 90% correct on S. You can think of the T as compressing the data and the wrong examples can then be memorized with encoding.

Probably Approximately Correct(PAC) Learning

PAC works for classification. Suppose there is a distribution D on $\{0,1\}^n$. Define function class $C = \{\text{decision trees of size s}\}$ where a learner runs in polynomial(not exponential) time. Fix $c \in C$, where c is the unknown decision tree we want to learn. The learner draws a sample (x, y) according to prob. dist. of D and y is equal to $c(x)$. PAC model has to run in polynomial time in all of the relevant parameters. We cannot draw $|S|$ from exponentially many training points.

Bad Goal: we output $c \in C$ that is consistent with S but the true error of c is $> \epsilon$.

Goal: For any choice of δ, ϵ, A should otput with probability $\geq 1-\delta$ an ϵ-accurate classifer. A is allowed to run in time polynomial $polynomial(\frac{1}{\epsilon}, \frac{1}{\delta})$ and take many more samples in time of $polynomial(\frac{1}{\epsilon}, \frac{1}{\delta})$.

PAC Sampling

When sampling our training set, we denote the probability of getting a non-representative sample by δ, and call $(1-\delta)$ the **confidence parameter** of our prediction. δ can be seen as the probability of getting a non-representative sample. Since we can't guarantee perfect label prediction, we introduce the **accuracy parameter**, ϵ, to describe the quality of the prediction. This accuracy parameter determines how far the output classifier can be from the optimal one, and a confidence parameter indicating how likely the classifier is to meet that accuracy requirement.

Anything that exceeds ϵ is considered a failure of the learner. The goal is for the learner to output an h(decision tree) that's in C such that

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

With probability at least $1-\delta$, the learner should output a hypothesis h s.t. $Pr_{x\sim D}[h(x) \neq c(x)] \leq \epsilon$.

The run-time is polynomial$(\frac{1}{\epsilon}, \frac{1}{\delta}, n, s)$.

Here's a simple example for the existence of δ. Suppose you're given one of two jars: a) jar with 100 blue marbles, and b) jar with 90 red and 100 blue marbles. The probability of failure for one hundred draws, or δ, is $(0.1)^{100}$.

Similarly, fix c_1 and assume it has a true error rate greater than ϵ. What is the probability draws from the distribution that c_1(or any $c \in C$) is consistent with S? It's $(1-\epsilon)^{|S|}$, where S is the the training set. In general, for every c_i with error greater than ϵ, the probability is at most $(1-\epsilon)^{|S|}$.

Another question is what is the probability there exists a function $c \in C$ whose true error is greater than ϵ and is consistent with S? $|C| \cdot (1-\epsilon)^{|S|} \leq \delta$. Continuing further, using the union bound, you'll get

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

What this is saying is that if you the number of training points larger than the quantity above, then with probability $\geq 1-\delta$ the function output c is $1-\epsilon$ accurate. We used $1-x \approx e^{-x}, 1+x \approx e^{x}$ to prove this.

The reason why we compute the bad event is because if we have an error greater than ϵ, it's this computed probability; otherwise, if the algorithm doesn't fail because bad doesn't occur, then our learner can't find a function that's consistent AND has a large error. If the bad event doesn't happen, then A can't find a function that's consistent and has a large error because they don't exist.

Realizability Assumption There exists $h^\star \in H$ s.t. $L_{(D,f)}(h^\star) = 0$. Note that this assumption implies with probability 1 over random samples, S, where instances of S are sampled according to D and are labeled by f, we have $L_S(h^\star) = 0$.

Corollary 2.3 Let H be a **finite hypothesis class**. Let $\delta \in (0,1)$ and $\epsilon > 0$ and let m be an integer that satifies

$$m \geq \frac{\log(|H|/\delta)}{\epsilon}.$$

Then, for any labeling function, f, and for any distribution, D, for which the realizability assumption holds(that is, for some $h \in H, L_{(D,f)}(h) = 0$), with probability of at least $1-\delta$ over the the choice of an i.i.d sample S of size m, we have that for every ERM hypothesis, h_S, it holds that

$$L_{(D,f)}(h_s) \leq \epsilon.$$

Efficiency

The learner should should be efficient: (n, s) where s is the size of decision tree and n in time polynomial in n and s. This also means that number of samples or draws from the distribution the learner can request should be bounded by a polynomial in n and s. With a probability of at least $1-\delta$ ($PROBABLY$), the learner should output an h s.t. $Pr_{x\sim D}[h(x) \neq c(x)] \leq \epsilon$. The run-time should be polynomial of $\frac{1}{\epsilon}, \frac{1}{\delta}, n(where\{0,1\}^n), s(sizeofdecisiontree)$(TODO WHY).

Sample Complexity

The sample complexity of learning H discusses how many examples are required to guarantee a PAC solution. It also depends on the log size of H.

TODO write down defintion 3.1 pac learnability

Corollary 3.2

Every finite hypothesis class is PAC leranable with sample complexity

$$m_h(\epsilon, \delta) \leq ceiling(\frac{log(|H|/\delta)}{\epsilon}).$$

When can we learn a function class, and which function classes?

Given an algorithm A which converts training sets to decision trees, $A(S)$ outputs a tree T consistent with S, where S is a training set. The $size(T)$ is going to be at most s. The A always outputs a consistent hypothesis from c given any traning set, assuming one exists.

Given A, how can we learn $c \in C$? Simply draw sufficiently many training points.

Errors Revisited

We redefine the true error(or risk) of a prediction rule h to be

$$L_D(h) \stackrel{\text{def}}{=} P_{(x,y)\sim D}[h(x) \neq y] \stackrel{\text{def}}{=} D(\{(x,y) : h(x) \neq y\}).$$

The empirical risk(or training error) remains unchanged:

$$L_S(h) \stackrel{\text{def}}{=} \frac{|\{i \in [m] : h(x_i \neq y_i\}|}{m}.$$

PAC-Learning Axis-Parallel Rectangles

We're now dealing with infinite function classes with two dimensions. We can't exactly rely on what learned in the previous PAC section because there are infinitely many axes-parallel rectangles. The goal is given ϵ and δ, output h that is ϵ-accurate with probability $\geq 1-\delta$.

The tighest fitting rectangle should have m samples as follows:

$$m \geq \frac{4 \cdot log(4/\delta)}{\epsilon},$$

then the tightest fitting rectangle will be an epsilon-accurate hypothesis with probablity at least $1 - \delta$.

The intuitive explanation is that we construct a rectangle that has too much probablility mass outside of it. Meaning, the sample we drew didn't contain points outside of it. This can happen each if the strips with probablity $\epsilon/4$ strips has no points, so our training set failed to capture these points. Define b_1 for each of the strips which says points occurred in training sample but exist in distribution. Then the define $Pr(B_1)$ (out of 4) to be

$$Pr(B_1) \leq (1 - \epsilon/4)^m.$$

The probablity of failure is you draw outside of this error region for all m samples, is then **4 times the quantity above**.

The bad event is the tightest-fitting rectangle is too small, i.e. significant prob. mass exists outside of bounds of h. Pick a shaded region, say B_1, and B_1 is the probability we see no points in that strip on distribution D. If NEITHER of them occur, then it is ϵ-accurate hypothesis because of the probability union. First compute for one strip.

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

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

Half Spaces

Let's consider a class C of half-spaces for PAC learning. A half-space is

$$SIGN(w \cdot x - \theta),$$

where $x \in \mathbb{R}^n$, $w \in \mathbb{R}^n$, $\theta \in \mathbb{R}$ is an unknown scalar, f is boolean that outputs 0 or 1.

The function we're trying to learn is

$$f = SIGN(\Sigma_{i=1}^n w_i x_i - \theta).$$

Given draws of $(x, f(x))$ from D where x is distributed according to D. Suppose we get 0101 whose label is positive, then we know that $w_2 + w_4 > \theta$, and conversely, given a negative example with input 0110, then $w_2 + w_3 \leq \theta$. Each labeled example yields a linear inequality leading a system of linear inequalities. There's an infinite number of half-spaces here, since C can be infinite. If we assume that each w_i is an integer in some bounded range to have a bounded function class, then we can apply our consistency analysis.

We are looking for an algorithm with an uknown $\theta \in \mathbb{R}$ s.t.

$$h_\theta(x) = \begin{cases} -1 & \text{if } x \leq \theta \\ 1 & \text{otherwise} \end{cases}$$

Cross-Validation

CV is the hold-out approach for testing/approximating the true error of a classifier. You "leave out" some part of the training set and whatever fraction of mistakes your classifier makes here is the true error. We use the full training set and divide it into k folds. We then average the errors of all the k-folds.

Hold-out data is

1. labeled data is expensive to obtain
2. if we want to try out multiple methods for generating classifiers, we quickly lose confidnece in our estimates

Chernoff Bound

Suppose you have $X_1, X_2, .., X_n$ r.v.s where $X_i \in \{0, 1\}$, and suppose $\mathbb{E}[X_i] = p$. It's saying when you take the sum of a bunch of r.v.s, the sum deviates from μ is exponentially small in the quantity little n. **This bound still holds even when the probabilities differ, we just chose the same p here.** We can apply the Chernoff bound to the case of estimating the true error of a classifier. Then,

$$n = |S|$$

$$S = \Sigma_i^n X_i$$

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

We want to apply the Chernoff bound to the case of estimating the true error of a classifier with holdout set S, where $n = |S|$.We're interested in Z which is the r.v. that corresponds to the true error of the classifier.

We can let X_i be r.v. that equals 1 if h is incorrect on the ith element of S; 0 otherwise. **We define p to be the true error of the classifier h.** Pick a delta and decide what n is needed. $\mathbb{Z}$ corresponds to the true error of the classifier.

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

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

I added α as the bound of how incorrect you want to be(starting with (...) $< \alpha$, solve for n). Note: if $|S - n \cdot p| \leq \delta n$(the good event), the error rate on S(holdout) is within δ of the true error rate.

Note: the δ here is not the δ from PAC learning, it's defined by the α. The δ is how close you want to be between the training error versuses the true error of our classifier(e.g., 0.1). **I want the emperical error rate to be within δ of the true error on our classifier, and I want to be α confident.** We want to claim p is the true error of h. **Plug in the value for δ as the true error. Ignore the Pr[] side and solve only for the α and 2(...) part, solve for n.** It's saying the probability that your error rate on holdout set S doesn't deviate δ from the true error rate.

Markov's Inequality

Let X be a r.v. that only takes on **non-negative** values for some factor K. This is useful for upper bounds. This bound depends on the expected value of the r.v. and that X is non-negative.

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

$$Pr[X \geq K] \leq \frac{\mathbb{E}[X]}{K}$$

Chebyshev's Inequality

This tells us when something deviates by more than t standard deviations, where σ is the std. deviation and μ is the expectation of X. This bound assumes X has non-zero, finite variance.

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

Perceptron Learning

The class of half-spaces, designed for classification problems, namely R^d and $Y = -1, +1$.

$$HS_d = sign\text{đ}L_d = \{x \to sign(h_{w,b}(x)) : h_{w,b} \in L_d\}.$$

In the context of halfspaces, realizability is important because it means we can cleanly separate with a hyperplane all the positive examples from the negative examples.

Complicated LP Solver

A complicated algorithm using LP is to write m inequalities and then solve using LP solvers. The good news is they run in polynomial time.

Perceptron Algorithm

1. Choose an initial vector such as $w^0 = (0, ..., 0)$ or $w^0 = (\frac{1}{\sqrt{n}}, ...)$
2. A teacher provides an $x \in R^n$, and only if a mistake is made does the learner update its state.
3. **Assume a mistake was made.** In case 1 where x was truly a negative example, then $w_{new} = w_{old} - x$; otherwise, $w_{new} = w_{old} + x$ for the truly positive example. **An equivalent way to view the update rule is for every time you mistake to**

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

```
input: A training set (x_1, y_1), ..., (x_m, y_m)
initialize: w^(1) = (0, ..., 0)
  for t = 1, 2, ...
    if (∃i s.t. y_i⟨w^(t), x_i⟩ ≤ 0) then
      w^(t+1) = w^(t) + y_i x_i
    else
      output w^(t)
```

Assumptions

- 1st: Assume $\exists w^\star$, true unknown weight vector that labels all the points, assume norm $||w^\star|| = 1$. The point means no loss of generality. If I increase the magnitude of w, it wont impact the sign.
- 2nd: Assume every x is $||x|| = 1$. This is made using the assumption of $||x|| = 1$ but if we didn't make this assumption, then $||x|| = r$, the mistake bound would then be $O(\frac{r^2}{\rho^2})$.
- 3rd: $\theta = 0$. However, if we can add a constant 1 to x to have a bias term that will always be pulled from the weight vector.
- **Main Assumption**: there exists a margin ρ, all points are at least distance ρ from $w^\star$. All the challenges will be at least distance ρ. If you're a positive/negative example, you're going to be at least this distance from true halfspace. $w^\star$ is perpendicular to the half-space line.

Because all of our points have a distance of $\geq \rho$ from the true halfspace and are unit vectors, then $|\langle x, w^\star \rangle| \geq \rho$. This is called the **margin assumption**.

Peceptron Convergence Theorem

The mistake-bound of perceptron algorithm is $\leq O(\frac{1}{\rho^2})$. If all of the examples obey this margin assumption, then the learner will make at most the mistakes above.

Proof of MB on Perceptron

- Claim 1: on every mistake $w \cdot w^\star$ increases by at least ρ.

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

$$w_{new} \cdot w^\star = (w_{old} + y \cdot x) \cdot w^\star = w_{old} \cdot w^\star + y \cdot x \cdot w^\star$$

  However, we know that $|\langle x, w^\star \rangle| \geq \rho$, therefore the added quantity is always at least ρ and positive because we multiply by y.
- Claim 2: $||w||^2$ increases after every mistake by at most 1

$$||x|| \text{ is at least magnitude 1}$$

$$w_{new} = ||w_{old} + y \cdot x||^2 = ||w_{old}||^2 + 2 \cdot y\langle x, w_{old} \rangle (\text{ negative}) + ||x||^2 (\text{ at most 1})$$

Perceptron: Error Bound Proof

For t mistakes, we know that $t\rho$ is the minimum. Furthermore, if we had to account for the theta or a bias, we can add a new feature and call it x_{n+1}. Also, if $||x|| \neq 1$ but r, then the M.B. will be $O(\frac{R^2}{\rho^2})$. We also know that $||w|| \leq \sqrt{t}$.

$$t \cdot \rho \leq w \cdot w^\star \leq ||w|| \cdot ||w^\star||$$
$$t \cdot \rho \leq ||w|| \leq \sqrt{t}$$
$$t \cdot \rho \leq \sqrt{t}$$
$$t \leq \frac{1}{\rho^2}$$

Polynomial Threshold Functions

PTF where $p(x)$ is a multivariate polynomial of degree d.

$$f = sign(p(x))$$

We can use a feature map for exactly n^2 new variables for a degree two polynomial. A degree of two polynomial in n dimensions as being converted to a halfspace in n^2 dimensions. Learning PTFs of degree d is equivalent to learning halfspaces in n^d dimensions.

$$(x_1, ..., x_n) \to (x_1^2, x_1 \cdot x_2, ..., x_n^2)$$

$$(y_1, ..., y_{N=n^2})$$
$$f = sign\left(\Sigma_i^N w_i y_i\right)$$

Polynomial Threshold Functions Run-time

Running in n^d dimensions, just computing the feature map takes time n^d, where n is the length of the input. The margin would be expensive in this n^d dimension. We need a better approach(next section: kernel trick).

Kernel Perceptron

The function $|\langle x, w^\star \rangle| \geq \rho$. You can run the perceptron implicitly in this higher dimensional space via a kernel function.

$$K(x^1, x^2) = \langle \phi(x^1), \phi(x^2) \rangle$$

Assume $w_{old} = 0$ on the first iteration because we initialize to 0 by default($w = 0^{n^d}$).

$$w_{new} = w_{old} + y \cdot \phi(x)$$

After the first mistake, we need to evaluate x^2, so

$$w_{new} \cdot \phi(x^2) = \langle y \cdot \phi(x^1), \phi(x^2) \rangle = y \cdot K(x^1, x^2)$$

$$w_{t+1} = \Sigma_{i=1}^{t} y^i \cdot \phi(x^i) \in \mathbb{R}^{n^d}$$

Simple Kernel Function

Take an example of a deg-2 PTF.

$$\phi(x_1, ..., x_n) = (x_1 \cdot x_1, x_1 \cdot x_2, ..., x_n \cdot x_n)$$

Let's define $K(x, z) = \langle \phi(x), \phi(z) \rangle$. Expanding out the definition of the inner product

$$K(x, z) = (x_1^2 z_1^2 + x_1 \cdot x_2 \cdot z_1 \cdot z_2 + ...) = (\Sigma_{i=1}^{n} x_i \cdot z_i) \cdot (\Sigma_{j=1}^{n} x_j \cdot z_j) = (x \cdot z)^2 = K(x, z)$$

The k's now vary from $1, ..., n$ instead of $1, ..., n^d$.

Other Kernel Functions

$$K(x, z) = (x \cdot z + c)^2$$
$$\phi(x) = (x_1^2, ..., x_n^2, \sqrt{2c} \cdot x_1, .., \sqrt{2c} \cdot x^n, c)$$

Gaussian Kernels $\approx K(x, z) \approx e^{-||x-z||^2}$ (Radial Basis Kernel)

Linear Regression

Linear Regression is fitting a line to data. One fundamental difference is our labels will be real-valued, i.e., (x, y) where $y \in \mathbb{R}$. X and Y are random variables, and suppose we want to predict the label/value, and as usual, we get to see X.

Suppose we wanted to predict Y but we don't know X, where $(x, y) \sim D$, then the optimal guess is to use $\mathbb{E}[Y]$.

Our loss function will be the square-loss: $(\text{prediction} - Y)^2$.

We can also observe X and we want to predict Y. The optimal prediction will be $\mathbb{E}[Y|X] = f(x)$, where $f(x)$ is the regression function. **The obstacle is that $f(x)$ could be unknown or difficult to compute.**

LR says: given X, what linear function of X should we use to predict Y? We want to learn coefficients β_0 and β_1 to **minimize**

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

Simple Linear Regression

Where x and y are scalars with m samples,

$$(x^1, y^1), ..., (x^m, y^m)$$

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

We can take derivative w.r.t B_0, B_1 and set them equal to 0.

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

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

We eliminate some coefficients(e.g., -2) because they aren't important.

We can use $\overline{y}$, $\overline{x}$(averages), and β_1.

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

β_1 will not involve β_0.

$$0 = \overline{xy} - \beta_0\overline{x} - \beta_1\overline{x}^2$$
$$0 = \overline{xy} - (\overline{y} - \beta_1\overline{x})\overline{x} + \beta_1\overline{x}^2$$
$$\beta_1 = \frac{\overline{xy} - \overline{x}\cdot\overline{y}}{\overline{x^2} - (\overline{x})^2}$$

The denominator is the variance of x, not exact but close and the numerator is the covariance.

Linear Regression with Multiple Variables

$x \in \mathbb{R}^n, y \in \mathbb{R}$, where we're fitting a line to n-dimensional data with m rows and n columns by representing and $m \times n$ matrix, with $y \in \mathbb{R}^m$ labels.

Goal: find a vector $w \in \mathbb{R}^n$ s.t. $\min_w ||X \cdot w - y||_2^2$

We should take the orthogonal projection of y onto Xw. Another way of writing it is $X^T \cdot (y - Xw) = 0$.

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

Linear Regression: Maximum Likelihood

Assume simple linear regression case. Assume $y = \beta_0 + \beta_1 x + \epsilon$, where ϵ $N(0, \sigma^2)$ – this is a random gaussian noise variable. The likelihood function is the probability of seeing a training set given choices of β_0, β_1 of our parameters.

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

PDF of a Gaussian is $\frac{1}{\sqrt{w\pi\sigma^2}} \cdot e^{\frac{-(y^i - (\beta_0 + \beta_1 x^i))^2}{2\sigma^2}}$. Choose β_0 and β_1 to maximize this likelihood.

We can compute using the log likelihood of

$$\Sigma_{i=1}^m log(p(y^i | x^i, \beta_0, \beta_1))$$

Vector Linear Regression: Convexity

A function is **convex** if the chord connecting any two points of the graph lies above the function.

A mathematical way of describing convexity is

$$f(\frac{1}{2}x_1 + \frac{1}{2}x_2) \leq \frac{1}{2}f(x_1) + \frac{1}{2}f(x_2)$$

This is the definition of convexity where $x^\star$ is the global minimum and substitute it for x_2 below. We'll always find the global minimum if the function is convex.

$$f(a \cdot x_1 + (1-a) \cdot x_2) \leq a \cdot f(x_1) + (1-a) \cdot f(x_2)$$

Gradient Descent

Even for more complicated functions, assume they are locally linear(differentiable). Even in d dimensions it's assumed the function looks linear in a local neighborhood. When performing GD, we need to keep our step size η small to keep it "locally linear".

Gradients

It might be sometimes easier to compute gradients coordinate by coordinate.

$$f(x) = x^T A x - b^T x = \Sigma_{i=1}^n \Sigma_{j=1}^n a_{ij} x_i x_j - \Sigma_{i=1}^n b_i x_i$$

$$\frac{\partial f}{\partial x_k} = \Sigma_{j=1}^n a_{kj} x_j + \Sigma_{i=1}^n a_{ik} x_i - b_k$$

Split into two cases: $i = k$ and $i \neq k (j = k)$

Answer: $Ax + A^T x - b_k$ gradient at point x

GD: Minimizing f(w)

Initially, we choose w randomly. If $||\nabla f(w)|| < \epsilon$, stop and output w; otherwise

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

GD: Linear Regression

Mean Squared Error(MSE) for linear regression is a convex function, since parabolas are convex, and we're taking positive sums of these convex functions, if this is for linear regression.

Where m is the number of traning data,

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

Derivation:

$$\frac{\partial g_j}{\partial w_i} = 2 \cdot (w^T x^j + b - y^j)x_i^j$$
$$\nabla g_j(w) = 2 \cdot (w^T x^j + b - y^j)x^j$$

The runtime complexity is $O(m \cdot n)$, because each sample is n-dimensional.

Stochastic Gradient Descent

We previously summed over all points in the training set. For SGD, we pick an index j at random and compute the gradient w.r.t this point only. You can also use batches to interpolate between GD or pure SGD. Batches help reduce the variance of the r.v..

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

The reason why this works is because in expectation,

$$\mathbb{E}[w_{new}] = w_{old} - 2 \cdot \eta \cdot \frac{1}{m}\Sigma_{j=1}^{m}(w^T x^j + b - y^j)x^j$$

Step Size η

- Many techniques for adaptively choosing η
- This is more art than science. We can use cross-validation to pick η.
- momentum has a "velocity" variable

$$V_0 = 0$$
$$V_i = \alpha \cdot v_{i-1} - \eta g_i$$
$$w_{new} = w_{old} + V_i$$

Boosting

A weak learner is a learner that gets 51% accuracy on a training set. We construct an algorithm that uses many sub-algorithms to come up with many classifiers with 51% accuracy. Then, you'll re-adjust the training distribution to add more weight to the points we got wrong. And because we still have to be 51% accurate, it's going to do better on the points we got wrong before because they're more likely to be drawn. You then take the majority vote of those classifiers.

Re-weight the points we got wrong in the training set to have slightly more weight than last time; the points we got correct last time will have less weight. Then, take the majority of classifiers generated during this process.

Adaptive Boosting

INPUT: A training set of size m

Initially D_0 is the uniform dist. corresponding to $w_i = \forall_i$, where the distribution is obrtain by W(sum of all weights)

E=error rate

A=accuracy=1-E

$\beta = \frac{E}{A}$

`Concretely,` $E = \frac{1}{2} - \gamma$

`Thus`
$\beta = \frac{1/2-\gamma}{1/2+\gamma}$ `at every iteration`

`How to update weights: at iteration t, run A to obtain` h_t

`For each` x_i `s.t.` $h_t(x_i)$ `is correct:`
$w_i^{new} = \beta \cdot w_i^{old}$;
`otherwise if it incorrect,` $w_i^{new} = w_i^{old}$.

`Repeat for` t `steps and output the majority vote of MAJ(h_1,...,h_t).`

AdaBoost Error

After T iterations, $err(h_{final}) = MAJ(h_1, ..., h_t) \leq e^{-2T\gamma^2}$ means choose $T \approx \frac{1}{\gamma^2}\log(\frac{1}{\epsilon})$, then error of $h_{final} \leq \epsilon$.

AdaBoost Error Proof

Weight of **correct points** after iteration t, where W is the weight of the points before iteration T, is

$$(1/2 + \gamma) \cdot \beta \cdot W$$

Weight of **incorrect points** after iteration t, where W is the weight of the points before iteration T, is (use β definition for substitution)

$$(1/2 - \gamma) \cdot W$$

Thus, the total sum of all the weights is the sum of those two terms, which ends up being

$$W((1/2 + \gamma)\beta + 1/2 - \gamma)$$

$$W \cdot (2 \cdot (\frac{1}{2} - \gamma))$$

After i iterations, the sum of the weights is

$$W_o \cdot (2 \cdot (\frac{1}{2} - \gamma))^i$$

This implies after T iterations, the sum of all the weights (upper bound)

$$\leq (2 \cdot (1/2 - \gamma))^T \cdot W_0$$

Consider the point x_i that h_{final} gets wrong. It means in the majority of iterations, the point was labeled incorrectly. This means the majority of the hypothesis got it wrong. It means we didn't multiply it by β for at least $t/2$ iterations. The weight of this point is $weight(x_i) \geq \beta^{T/2}$.

If h_{final} has error rate ϵ, then weight of points h_{final} misclassifies is $\geq \epsilon \cdot m \cdot \beta^{T/2}$(lower bound).

Therefore, error is less than total:

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

$$...$$

$$\epsilon \leq (1 - 4\gamma^2)^{T/2} \text{ using } (1 + x \approx e^x)$$

$$\epsilon \leq e^{-2\gamma^2 T}$$

After T iterations, the error of final hypothesis is at most $\epsilon \leq e^{-2\gamma^2 T}$.

Adaboost Modifications

In the simplified version, we assumed accuracy was exactly $\frac{1}{2} + \gamma$. This change allows us to have a higher accuracy if we encounter it.

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

Output:

$$\text{sign}(\Sigma_t \alpha_t h_t - \frac{1}{2})$$

where

$$\alpha_t = \frac{\log(1/\beta_t)}{\Sigma log(1/\beta_t)}$$

How do we guarantee that h_{final} generalizes? Make sure that m is sufficiently large.

Nothing in the boosting procedure depended on m directly, as long as the accuracy, γ is independent of the size of the training set. If this is the case, then we have the freedom to choose m large enough to make sure after T iterations of boosting, the number of classifiers times the size of the classifier is going to be less than m.

If γ(accuracy) is independent from m, the size of the training set we can choose m to be sufficiently large.

Hedge

m experts from $C_1, ..., C_m$. At each iteration, expert C_i suffers a loss of $l_i^t \in [0, 1]$ at iteration t.

We maintain a set of weights $w_1, ..., w_m$, one weight for each expert, and the weighted average is

$$P_i = \frac{w_i}{\Sigma_i w_i}$$

At t^{th} iteration,

$$P_i^t = \frac{w_i^t}{\Sigma_i w_i}$$

The loss we suffer at iteration t is $p^t \cdot l^t$(dot product) is the weighted average of loss of the experts at iteration t.

Total loss we suffer after t iterations is $\Sigma_{t=1}^T p^t \cdot l^t$

Hedge says the new weight should be $w_i^{new} = w_i^{old} \cdot \beta^{l_i^t}$. If the loss suffered by the ith expert is 0, then the weight doesn't change.

Your loss is

$$\leq \min_i \Sigma_{t=1}^T l_i^t + O(\sqrt{T} log(m))$$

. This doesn't look enticing because of the O factor but if you divide everything by T, the T will grow faster than the $\sqrt{T}$, hence the average loss is converging to the loss of the very best expert.

Logistic Regression: Classification

This is classification by half-space. The loss function for classification for label $y^i \in 0, +1$ is going to be the

$$sign(w^t x)$$

for some vector w. We'll be penalized if $sign(w^t x^i) \neq y^i$. This means $y_i \cdot (w^T x^i)$ is negative; otherwise no penalty.

This corresponds to 0-1 loss function. **This is a non-convex loss function(bad)**.

$$l(z) = \begin{cases} 1 & z \leq 0 \\ 0 & z > 0 \end{cases}$$

Classification: Optimization Problem

$$min_w \frac{1}{m} \Sigma_{i=1}^m l_{0-1}(y^i \cdot w^T x^i)$$

Logistic Regression: Linear Regression

$$min_w \frac{1}{m} \Sigma_{i=1}^m (w^T x^i - y)^2$$

If $\mathbb{E}[Y|X] = w^T x$, then the optimal approach is to perform linear regression and output the w.

The least squares loss function for regression

Regression → convex loss function (good)

Classification → non-convex loss function(0-1 loss) (bad)

Let's relax the 0-1 loss to a different "surrogate loss" that's related to 0-1 loss but they'll be convex. You won't get the same solution as a 0-1 loss because it's NP-hard but you'll get solutions that are close.

Losses

A few loss functions that are convex:

- $y_{\text{logistic regression}}(z) = \log(1 + e^{-z})$
- $y_{\text{logistic regression}}(y^i \cdot w^t x^i) = \log(1 + e^{-(y^i \cdot w^t x^i)})$
- $y_{\text{hinge}} = max(1 - z, 0) = max(1 - (y^i \cdot w^t x^i), 0)$
- $y_{\text{exponential}} = e^{-z}$

Logistic Loss Optimization

$$L(w) = min_w \frac{1}{m} \Sigma_{i=1}^m log(1 + e^{-y^i \cdot w^T x^i})$$

We will use the sigmoid function and this fact(prove by multiplying first quantity by e^z)

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

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

Model for logistic regression:

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

What is the most likely w given the training set? **Likelihood**

$$\text{Likelihood}(w) = \Pi_{i=1}^m P(y = y^i | x^i, w) = \Pi_{i=1}^m g(y^i \cdot w^T x^i)$$

We want to maximize w,

$$\begin{aligned}\text{log-likelihood}(w) &= \Sigma_{i=1}^m log[g(y^i \cdot w^T x^i)] \\ &= -\Sigma_{i=1}^m log(1 + e^{-y^i \cdot w^T x^i})\end{aligned}$$

Minimizing Logistic Loss

The purpose of the log. loss function on w was to get a differentiable, convex function to run GD on to minimize the logistic loss value. Once we find w', we label future examples with +1 with probability $g(w'^T \cdot x)$, where g is the sigmoid function.

$$y_{\text{log reg}}(z) = \log(1 + e^{-z})$$

$$\frac{\partial y_{\text{log reg}}(z)}{\partial z} = \frac{-e^{-z}}{1 + e^{-z}} = -\frac{1}{1 + e^z} = -g(-z)$$

If we take an example on

$$\frac{\partial y_{\text{log reg}}(y \cdot w^T x)}{\partial w_k} = -g(-y \cdot w^T x) \cdot (y \cdot x_k)$$

This precisely tells us how to find the max likelihood w.

Derivation of log. reg. **with z**

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

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

Derivation of log. reg. **with** $y \cdot w^T x$. THis tells us how to find the max likelihood.

$$l_{\text{logistic regression}}(z) = \log(1 + e^{-y \cdot w^T x})$$

$$\frac{\partial l'_{\text{logistic regression}}(y \cdot w^T x)}{\partial w_k} = -g(-y \cdot w^T x) = -g(-y \cdot w^T x) \cdot y \cdot x_k$$

Multinomial Logistic Regression

We'll have $w_1, ..., w^{k-1}$ vectors is going to be

$$Pr[y = 1 | x] \propto e^{w^{1T} \cdot x}$$

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

We only have $k - 1$ vectors because the $Pr[y = k] = 1 - \Sigma_{i=1}^{k-1} Pr[y = i]$

The loss we should use is the cross-entropy loss, which is a generalization of logistic loss.

Cross-entropy loss

$$-\Sigma_{i=1}^k y_i log(p_i)$$

The softmax converts real-values into probabilities with $(\frac{e^{z_1}}{z}, \frac{e^{z_2}}{z}, ...)$.

PCA

Probably the most important technique in reducing dimensionality of your data. PCA objective is to find the eigendecomposition of a covariance matrix.

The goal is to find $v_1, ..., v_k$ orthogonals s.t. $||v||_2 = 1$ where

$$max(\frac{1}{m}) \Sigma_{j=1}^m \Sigma_{i=1}^k \langle x^j, v_i \rangle^2 \text{ is maximized}$$

This is referred to as the direction of the maximum variance, also the sample variance. The expectation is 0 in this case.

Where A corresponds to the covariance matrix ($X^T X$ or $\frac{1}{n} X^T X$). Alternatively, this is written as

$$max_{v, ||v||=1} V^T A V$$

PCA Preprocessing

1) Subtract the mean from your data
2) Normalize the columns by the standard deviation. For every feature i, you compute $\sqrt{\frac{1}{m}\Sigma_{j=1}^{m}(x_i^j)^2} = \sigma_i$, then divide the i th features by σ_i, where j ranges from to 0 to m.
3) Compute the eigenvalue/eigenvector decomposition of your matrix QDQ^T
4) First k rows of Q^T are the top k eigenvectors/principal components

To prove i th row of Q^T is an eigenvector of A,

$$\text{ith row of } Q^T = Q \cdot e_i$$

$$A \cdot Q \cdot e_i = QDQ^TQ \cdot e_i = QDe_i = \lambda_i \cdot Q \cdot e_i$$

PCA Derivation

Suppose we have a matrix X that encodes our training set S by $m \times n$. We'll then look at $\frac{1}{m}X^TX(n \times n)$, which is a sample covariance matrix.

Note: X^TX is a symmetric matrix.

All eigenvalues of symmetric matrices are ≥ 0. For a matrix A, v is an eigenvector if $A \cdot v = \lambda \cdot v$ where $\lambda \in \mathbb{R}$.

You can write a sample covariance matrix as a diagonal matrix times some rotation matrix.

Take a vector v, then \$we don't know A is diagonal but we know that $A = QDQ^T$, where A is almost diagonal.**

Where $e_1 = (1, 0, ..., 0)$, choose $v = Q \cdot e_1$ which maximizes the top eigenvector of A.

You can think of Q as rotational matrices that transform a matrix but don't change the magnitude of the vectors. You can write a covariance matrix as a diagonal matrix times some rotation matrices.

Spectral Therorem:

This theorem establishes the relationship between matrices of the form X^TX and their eigendecompositions.

Any symmetric matrix A has an eigendecomposition

$$A = Q \cdot D \cdot Q^T$$

where Q is an orthogonal matrix and D is a diagonal matrix(entries $\in \mathbb{R}$ are the eignvalues of A). Furthermore, if $A = X^TX$, then all eigenvalues are ≥ 0. Because covariance matrices are symmetric, then all of the eigenvalues are positive; however, eigenvalues could be negative if it's not a symmetric matrix.

Claim 1: for any V, $V^TAV \geq 0$. We're taking an inner product of it self, so it must be greater than 0. Recall that $A = X^TX$.

$$V^TAV = (XV)^T \cdot (XV)$$

Claim 2: Matrix A cannot have any negative values(proof by contradiction).

Let's assume that $\lambda_i < 0$, and we know that $A = QDQ^T$ (because symmetric matrix), let's consider $v = Q \cdot e_i$. Then, V^TAV is $e_i^TQ^TQDQ^TQe_i$. Q is orthogonal, so it goes to identity. This leaves us with

$$e_i^TDe_i < 0$$

We take the inner product with e_i because it picks out that one negative value. We know that it's less than 0 but that contradicts our claim.

Defining SVD

Every matrix A can be written as $A = U \cdot S \cdot V^T$, where U is an $m \times m$ orthogonal matrix, V is an $n \times n$ orthogonal matrix, and S is an $m \times n$ diagonal matrix. The rows of V^T are the right singular vectors, and columns of U are the left singular vectors. Entries of S are $s_1 \geq s_2 \geq ... \geq 0$. Singular values are unique but singular vectors are not unique. The computation time of SVD is $O(m^2n)$ or $O(n^2m)$.

$$A = \Sigma_{i=1}^{min(n,m)} = s_iu_i \cdot v_i^T \text{ (dot is the outer product)}$$

We can define a matrix to have rank k if $A = YZ^T$, where A is $m \times n$, Y is $m \times k$, and Z^T is $k \times n$. This is important because if we could factor A into $k(m + n)$ entries, we would be able to compress it. The result is still a $m \times n$ matrix. The goal is to find a matrix A' that has rank k and minimizes $||A - A'||_F$ over all rank-k matrices.

In the Netflix Challenge example mentioned in class, we assumed rank=1 where each row is a multiple of some other row. A rank-0 matrix is an all zeros matrix, a rank-1 matrix is all rows are multiples of each other or all columns are multiples of each other. A rank-2 matrix means A is the sum of two rank-1 matrices and A is not rank-1.

SVD: Matrix Completion

A common approach is to replace the question marks/unknowns with either 0, the average of all known values, or average value in that column or row. Then, we find the best rank k approximation to the matrix after filling the question marks in. This outputs the best rank k approximation.

SVD: Choosing K

A common heuristic for choosing k is to take enough singular values so that the sum of the remaining values is $\leq \frac{1}{10}$ for the taken values.

For the linear regression case, if $A = D$(where A is what we're trying to solve for), then let $x_1 = \frac{b_1}{d_1}$ since it will best align with $||Ax - b||^2$. Repeat this for all n dimensions. We can let the pseudo-inverse of D to be $D^\dagger$. The solution is $x = D^\dagger \cdot b$ in the easy case when A is diagonal. Each diagonal entry is $\frac{1}{d_1}, \frac{1}{d_2}, ...$ in $D^\dagger$ in the easy case.

In the general case, for linear regression, knowing that if U is an orthogonal matrix, $||Ux|| = ||x||$.

$$||Ax - b||^2 \equiv min_x||USV^Tx - b||^2$$

Multiplying by U^t and letting $(y = V^Tx) \equiv (Vy = x)$,

$$\equiv min_x||USy - b||^2$$

Knowing $y = S^\dagger \cdot U^Tb$,

$$y = S^\dagger \cdot U^Tb$$

$$x = VS^\dagger U^Tb \text{ (solution to linear least squares)}$$

We don't need to worry that the matrix A is invertible or not.

SVD: Application for PCA

We had a covariance matrix X^TX.

$$X^TX = (USV)^T \cdot (USV^T)$$
$$\equiv V^TS(U^TU)SV^T$$
$$\equiv V^TS^2V^T$$

V and V^T are both orthogonal matrices. We got the eigendecomposition for free. The right singular vectors of X(rows of V^T) are the principal components (top eigenvectors of X^TX). The singular values are the square root of the eigenvalues of X^TX. In other words, we square the singular values to get the eigenvalues.

Maximum Likelihood Estimation

The general idea is to estimate parameters of probabilistic models. We want to maximize θ where

$$\theta = \text{argmax}_\theta L(\theta) = \text{argmax}_\theta p(D|\theta)$$

We can also use $log(...)$ of the function as well because the log is a monotonically-increasing function. If the θ maximized the original function, it should also maximize the log of that function. The small l is the log-likelihood function. Then take the gradient and set it to 0.

$$l(\theta) = \log\left(L(\theta)\right)$$

Parameter Estimation by Maximum Likelihood Estimation

Independent and identially distributed following an unknown distribituion $p_\star$ from a parametric family of distributions, estimate θ.

$$\{p(\cdot|\theta) : \theta \in \Theta\}$$

Likelihood function

$$L(\theta) = P(x_1, ..., x_n|\theta) = \Pi_{i=1}^n p(x_i|\theta)$$

Log-likelihood function

$$l(\theta) = log(L(\theta)) = P(x_1, ..., x_n|\theta) = log(\Pi_{i=1}^n p(x_i|\theta))$$

$$= \Sigma_{i=1}^n log(p(x_i|\theta))$$

Maximum Likelihood Estimation:

$$\hat{\theta} = \text{argmax}_\theta l(\theta)$$

We can often solve using closed form solutions but generally, we can solve using numerical methods.

MLE: Bernoulli

Bernoulli reformulation for $x = 0(\theta), 1(1-\theta)$:

$$Pr(x) = \frac{e^{xw}}{1+e^{w}}$$

MLE: Gaussian Distribution

$N(\mu, \sigma^2)$, where μ is the center/mean and the width of the peak is the σ(std. deviation) and variance is σ^2. If we take the integral, the area is 1. An example is the stock market where you know the prices and with MLE you can estimate the mean and variance.

$$p(x|\theta) = \frac{1}{(2\pi)^{1/2}\sigma}\exp(-\frac{1}{2\sigma^2}(x-\mu)^2), \theta = \mu, \sigma$$

$$l(\theta) = \max_{\mu,\sigma} \frac{-n}{2} log(2\pi) - nlog(\sigma) - \frac{1}{2\sigma^2}\Sigma_{i=1}^{n}(x_i-\mu)^2$$
$$\hat{\mu} = \text{argmin}_{\mu}(\Sigma_{i=1}^{n}(x_i-\mu)^2)$$

We get the empirical mean

$$\frac{\partial l(\theta)}{\partial \mu} = \frac{1}{n}\Sigma_{i=1}^{n} x_i$$

We get the empirical variance

$$\frac{\partial l(\theta)}{\partial \sigma} = \hat{\sigma}^2 = \frac{1}{n}\Sigma_{i=1}^{n}(x_i-\hat{\mu})^2$$

MLE: Uniform Distribution TODO

Where $1/\theta$ for some $x \in [0, \theta]$ and 0 otherwise. We can't use log-likelihood directly because it's sometimes 0. We'll use the product directly(Π).

$$P(x|\theta) = \frac{1}{\theta}\mathbb{1}(x \in [0, \theta])$$

MLE: Regression TODO

MLE: Theoretical Properties

- $\text{bias}(\hat{\theta}) = \mathbb{E}_{\theta^\star}[\hat{\theta}(x_1, ..., x_n)] - \theta^\star$ - bias measures the difference between the mean and true value
- $\text{var}(\hat{\theta}) = \mathbb{E}_{\theta^\star}[(\hat{\theta}(x_1, ..., x_n) - \mathbb{E}_{\theta^\star}(\hat{\theta}(x_1, ..., x_n)))^2]$ - variance measures fluctuation around the mean
- $\text{MSE}(\hat{\theta}) = \mathbb{E}_{\theta^\star}[(\hat{\theta}(x_1, ..., x_n) - \theta_\star)^2]$
- If $\text{bias}(\hat{\theta}) = 0$, then it's considered an **unbiased estimator**
- Constitent estimators imply asymptotic unbiasedness
- If you are unbiased, you cannot imply consistency
- $\text{MSE}(\hat{\theta}) \to 0$ as n approches infinity, then it's consistent.
- "Asymptotic Unbiased" - bias goes to 0 as n goes to infinity
- Consitent Estimator implies asymptotic unbiasedness but doesn't imply direct unbiasedness

Bias-Variance Decomposition

$$\text{MSE}(\hat{\theta}) = (\text{BIAS}(\hat{\theta}))^2 + \text{VAR}(\hat{\theta})$$

KL Divergence

It's a way of measuring the difference between two distributions. MLE can be viewed as minimizing this notion of difference between data distrubtion and model distribution. Note that $KL(q||p) \neq KL(p||q)$. Maximizing the log-likehood function is equivalent to minimizing the KL divergence between the data distribution and the model.

$$KL(q||p) = \mathbb{E}_q[\log q(x) - \log p(x)]$$

If it's discrete, then it's

$$\mathbb{E}_q[\frac{p(x)}{q(x)}] = \Sigma_x q(x)\Big(\log \frac{q(x)}{p(x)}\Big)$$

Otherwise it's continuous and it's

$$\int q(x)\Big(\log \frac{q(x)}{p(x)}\Big)dx$$

Properties of KL:

$$KL(q||p) \geq 0 \text{ for any } q \text{ and } p$$
$$KL(q||p) = 0 \text{ if and only if } q = p$$

Jensen's Inequality for convex functions:

$$\mathbb{E}_q[f(x)] \geq f(\mathbb{E}_q(x))$$

Bayesian Inference

Bayesian Inference is another technique for estimating parameters but have the advantage of incorporating prior information as well as qualifying uncertainty.

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} \propto P(D|\theta)P(\theta)$$

$$P(\theta|D) \text{ (posterior)}$$

$$P(D|\theta) \text{ (likelihood)}$$

$$P(\theta) \text{ (prior)}$$

$$P(D) = \int P(D|\theta)P(\theta)d\theta$$

Math Review

TOOD log rules here / stats probs rules like variance and so forth , derivatives of logs TODO linearity of expectation

$$-log(x) = log(\frac{1}{x})$$

$$1 - x \approx e^{-x}$$

$$1 + x \approx e^{x}$$

$$ln(x) \approx x - 1$$

$$|u \cdot v| \leq ||u||\ ||v||$$

$$covariance(x, y) = \mathbb{E}[x \cdot y] - \mathbb{E}[x] \cdot \mathbb{E}[y]$$

TODO WRITE VARIANCE and SAMPLE FORMULA $$ The two-norm is the square root of the squares.

Gradient for d-dimensions,

$$\nabla f(x) = (\frac{\delta f}{\delta x_1}x, ..., \frac{\delta f}{\delta x_d}x)$$

l^t is a vector of losses suffered by all experts at the t'th iteration. The goal is for the sum of our losses after t iterations should be "close" to the best expert in hindsight.

Linear Algebra

- Frobenious Norm: $\sqrt{\Sigma_{i,j}A_{i,j}^2}$
- Where A is an orthogonal matrix, then
  - A^{-1} is an orthogonal matrix
  - A^T is an orthogonal matrix
  - $A^TA = AA^T = I$
  - $A^{-1} = A^T$
- $proj_v a = \frac{a \cdot v}{||v||^2}v$
- If A and B are symmetric matrices of the same size, then $AB = BA$.

Statistics

Overall Accuracy

$$P_D[f(x) = y] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(f(x_i) = y_i) = \frac{TP + TN}{TP + FP + TN + FN}$$

Gaussian

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2}$$

Probability

Union bound: for any two sets A,B and a distribution D we have

$$D(A \cup B) \leq D(A) + D(B).$$

$$\text{VAR}[X] = \mathbb{E}[(X - \mathbb{E}[\mathbb{X}])^2] = \mathbb{E}[\mathbb{X}^2] - \mathbb{E}[X]^2$$

$$\sqrt{\text{VAR}[X]} = \text{standard deviation}(X) = \sigma$$

$$\mu = \mathbb{E}[X]$$

Sample Variance:

$$S^2 = \frac{\Sigma(x_i - \overline{x})^2}{n-1} =$$

We know that this is at most norm of w times norm of w star.

We know the square of the norm of w is at most t. Therefore, the norm of w is at most $\sqrt{t}$.