1 Machine Learning

Machine learning is a branch of data analysis that focuses on techniques to improve the ability of program to complete a task (e.g. by changing the weights of various parameters) without direct input from the programmer. Instead, data is used alongside some sort of metric which measures how "good" the machine is performing the task to learn from both successes and failures. I learned this material as part of the class CSE 446 that I took at University of Washington. We used the textbook *Machine Learning: A Probablistic Perspective* by Kevin Murphy, but I also read some supplementary material from the free online book *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman.

1.1 Maximum Likelihood Estimation

Maximum likelihood estimation essentially just assumes the data we have observed comes from some true probabilistic distribution of a set of random variables. If we can make some reasonable assumptions about what these distributions might be, we can choose the parameters of the distribution that maximize the probability of seeing the data we have. Suppose we have a series of samples from our probability distribution $f(x;\theta)$ which has parameter(s) given by θ that are unknown. Each sample will generate a random variable from the true probability distribution which has parameter(s) $\theta = \theta^*$, generating a sequence of random variables $X_1, X_2, ..., X_n$ as our data. **Assuming these samples are independent**, the X_i are iid random variables, then the probability we get this particular sequence is:

$$P(X_1 \cap X_2 \cap ... \cap X_n) = L_n(\theta) = \prod_{i=1}^{n} f(X_i;\theta)$$

We call this function of the distribution parameters $L_n(\theta)$ the **likelihood function**. The goal then is to find the parameter(s) θ that maximizes this function. In some cases, it may be easier to work with $log(L_n(\theta))$ as the log separates products into sums. This property may be useful in the next step. Once we have found $L_n(\theta)$ we can use multi-variable calculus to determine where the maximum of $L_n(\theta)$ occurs. For each parameter θ_i set the partial derivative to zero:

$$\frac{\delta L_n(\theta)}{\delta \theta_1} = 0$$
$$\frac{\delta L_n(\theta)}{\delta \theta_2} = 0$$
$$...$$
$$\frac{\delta L_n(\theta)}{\delta \theta_m} = 0$$

Solving this system yields a set of critical points, one of which is a maximum ($L_n(\theta)$ is a probability so it is bounded in the interval $[0, 1]$ and thus must have a max/min). Choose these parameter(s) $\hat{\theta}$ to be our "best guess" of the parameters of the system. Using Hoeffding's Inequality (see probability chapter) we can put some bounds on how close this approximation truly is.

Ideally, $E[\hat{\theta}] = \theta$ in which case we call $\hat{\theta}$ an **unbiased** estimator of the true parameters. If this is not the case, we call $\hat{\theta}$ a biased estimator of the true parameters as it favors some value over

the true values. In the later case, adjustments to the formula for $\hat{\theta}$ may be made to remove this bias.

Example
Suppose we are trying to determine the true probability θ^* a coin flips heads. We collect a sample of n tosses and k of them were heads. Let $X_i \in \{H, T\}$ represent the outcome of each toss. Clearly, the tosses are independent of each other so $X_1, ..., X_n$ is a sequence of iid random variables. The probability of getting this sequence (not just getting k heads but the particular order of flips) given a probability the coin lands heads of θ is:

$$L_n(\theta) = \theta^k(1-\theta)^{n-k}$$

$$log(L_n(\theta)) = klog(\theta) + (n-k)log(1-\theta)$$

Thus (in this case there is only a single parameter):

$$\frac{\delta log(L_n(\theta))}{\delta\theta} = \frac{k}{\theta} - \frac{n-k}{1-\theta} = 0 \rightarrow \theta = \frac{k}{n}$$

1.2 Linear Regression

Given a model, some function $f(x; \beta_i)$ that takes inputs $x = [x_1, x_2, ..., x_n]$ that is parameterized by $[\beta_0, \beta_1, ..., \beta_m] \in \mathbb{R}$, the goal of regression is to find the parameters of the model that best explain the given data. Often, this is done so by minimizing some sort of error metric, the difference between the true output y_i associated to input x_i and the predicted output $f(x_i)$. However, this choice of error statistic can have distinct consequences (see Data Analysis: Regression subsection for more information). A common measure of error is the SSE (aka RSS) which is the sum of the squared errors:

$$SSE = \sum_{i=1}^{n}(y_i - f(x_i))^2$$

GOAL: given n data points $\{(x_i, y_i)\}_{i=1}^n$ with features $x_i \in \mathbb{R}^d$ and response $y_i \in \mathbb{R}$ find the best weights $w \in \mathbb{R}^d$ for predicting y from x using the linear model.

$$y_{pred} = w^T x \tag{1}$$

To find the best w we have to have some way of specifying how "good" a model is. Using the data we have, we generate a model that has some desirable properties. Often, we choose to minimize the sum of the squared errors over the training dataset T:

$$RSS(w) = \sum_{(x_i,y_i)\in T}(y_i - w^T x_i)^2 = ||y - Xw||^2 \tag{2}$$

$$X = \begin{bmatrix} -x_1- \\ -x_2- \\ ... \\ -x_n- \end{bmatrix} \in \mathbb{R}^{n\times d} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_n \end{bmatrix} \in \mathbb{R}^{n\times 1}$$

To minimize the RSS, we can look use the gradient to look where local minimizers occur ($\nabla_w RSS(w) = 0$).

$$\nabla_w RSS(w) = -2X^T(y - Xw) \tag{3}$$

$$\nabla_w RSS(w) = 0 \rightarrow \boxed{X^T X w = X^T y} \tag{4}$$

What if we want an intercept? Replace $w^T x$ with $w^T x + b$. The solution is simple if all our features have zero mean: $\frac{1}{n}\sum_{i=1}^{n} x_i = 0$. Following directly from the partial derivative with respect to b we see that:

$$b = \frac{1}{n}\sum_{i=1}^{n} y_i$$

If our features are not zero mean, simply demean them (i.e. subtract the mean in each column from the entire column). If we do not want an intercept, we see we can just demean the response variable y as well. Including this intercept, our model becomes:

$$y_{pred} = w^T x + b \tag{5}$$

Note the "linear" part of linear regression only implies our model is a linear sum of a bunch of features. We can use feature maps to generate nonlinearities within this model. For example, if we had two variables a, b that we wanted to use as a predictor for y we could define our features to be:

$$x_i = [a, b, a^2 + b^2, cos(a), sin(b)]$$

1.2.1 Why use sum of squared errors?

Suppose the our model is subject to zero-mean normally distributed noise. Many kinds of noise fall into this category so this is typically a fair assumption. Thus, the true model is some function of the input $f(x)$ but the data measurements we receive are:

$$f_N(x;\beta) = f(x;\beta) + \mathcal{N}(0, \sigma^2)$$
$$y_i = w^T x_i + error_i$$

for some weights w. If our goal is to approximate the true parameters β then we can use maximum likelihood estimation (MLE) from last chapter. Given a bunch of points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ we can find the probability our function gives y_i given the parameter σ^2 and the input x_i. Since the noise makes up the difference between y_i and $w^T x_i$, this is equivalent to asking about the (relative) probability that the noise takes on a value of $y_i - w^T x_i$. Thus we can use the gaussian probability density function:

$$P(y \mid x, \sigma^2) \sim L_n(\sigma^2) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - w^T x_i)^2}{2\sigma^2}}$$

$$log(L_n(\sigma^2)) = \sum_{i=1}^{n} log\left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - w^T x_i)^2}{2\sigma^2}}\right)$$

$$= nlog\left(\frac{1}{\sigma\sqrt{2\pi}}\right) + \sum_{i=1}^{n}\left(-\frac{(y_i - w^T x_i)^2}{2\sigma^2}\right)$$

As the left term and the denominator of the right term are constant, then maximizing the likelihood is the same as maximizing $\sum_i -(y_i - w^T x_i)^2$, or equivalently minimizing $\sum_i (y_i - w^T x_i)^2$, our SSE measurement!

1.3 Bias-Variance Trade-off

Once we generate a model, we have to have some way of assessing how good it is. Is the model accurate? Does it generalize outside the training data? Is it interpretable? To understand these questions, first lets consider the data. Often, there are two main types of data: **training** and **testing**. This data is collected from some overall true distribution $\mathcal{D}$.

$$data = \{(x_i, y_i)\}_{i=1}^{n} \sim \mathcal{D} \tag{6}$$

The training data is the set of datapoints that you fit your model with. Once the model is fit, the test data is used to see how well the model performs. As the model has not seen the testing data during the tuning of its parameters, the testing data serves to measure how generalizable your model truly is. Let T represent the training dataset and S the testing dataset. Whatever our model framework is, we will use T to find the best parameters and generate a model $\hat{f}_T(x)$. Given some features x, it makes a prediction on the response y. Note that the model is dependent on the training data set used: different T will generate different $\hat{f}$. Once this model is generated, we can assess its performance based on some sort of loss metric (i.e. RSS). Define:

$$\epsilon(f) = E_{(x,y)\sim\mathcal{D}}[loss((x,y) \mid f)] \to E_{(x,y)\sim\mathcal{D}}[(f(x) - y)^2] \tag{7}$$

This is the **true error** of the fixed model f. Based on the loss function (i.e. RSS is chosen above) we could determine the expected error a model f will make. If we knew all possible datapoints we could get and their corresponding likelihoods (i.e. if we knew $\mathcal{D}$) we could get an accurate measure of the performance of the model. Of course we do not have all the data, only some samples T, S, so we can only approximate this true error. Define the **training error** and **testing error** on a fixed model f to be respectively:

$$\hat{\epsilon}_{train}(f) = \frac{1}{N_{train}} \sum_{(x,y)\in T} loss((x,y) \mid f) \to \frac{1}{N_{train}} \sum_{(x,y)\in T} (f(x) - y)^2 \tag{8}$$

$$\hat{\epsilon}_{test}(f) = \frac{1}{N_{test}} \sum_{(x,y)\in S} loss((x,y) \mid f) \to \frac{1}{N_{test}} \sum_{(x,y)\in S} (f(x) - y)^2 \tag{9}$$

We showed in HW1 that these are **unbiased estimators** of the true error $\epsilon(f)$. What do we mean by this? Well, first note that $\hat{\epsilon}_{train}(f)$ and $\hat{\epsilon}_{test}(f)$ are random variables: they depend on whatever the random sample, the training/testing dataset, from $\mathcal{D}$ happens to be. As such, they have an expectation. We call them unbiased estimators for the true error as $E[\hat{\epsilon}_{train}(f)] = E[\hat{\epsilon}_{test}(f)] = \epsilon(f)$ rather than some other quantity. However, we are usually not concerned with the error over some random model f but rather the over the model we generate from our training data $\hat{f}$. The parameters of $\hat{f}$ are chosen such that $\hat{f}$ is the model that has the lowest loss over all other possible models. Here in lies the problem: $\hat{f}$ has a strict dependence on the training dataset.

While $\hat{f}$ is fixed with respect to the testing dataset as the training and testing data are disjoint, and hence $E[\hat{\epsilon}_{test}(\hat{f})] = \epsilon(\hat{f})$ still, our training error is no longer an unbiased estimator. In fact, the training error will tend to be significantly lower than the true error due to the fact that $\hat{f}$ is chosen to minimize the training error.

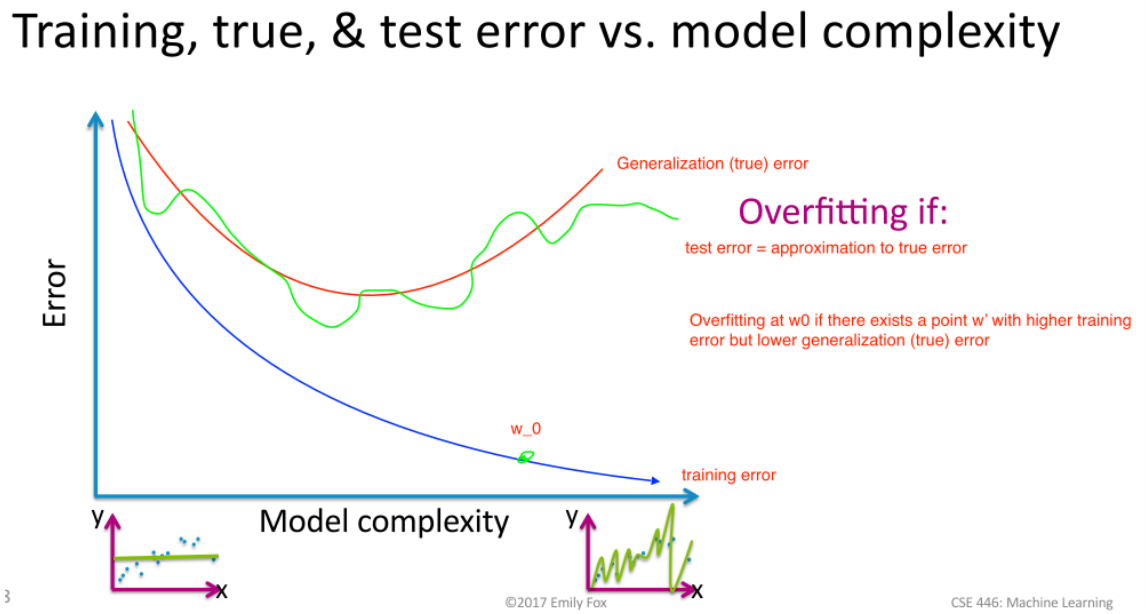

Figure 1: Here, we can see that the test error is still an unbiased estimator of the true error. However, the training error consistently decreases with increasing model complexity. With more parameters to tune, the training error can always be decreased. However, at some point overfitting begins to take its toll and the model loses generalizability. This is why we see the test/true error begin to rise again with increasing model complexity

NOTE: do not let your testing data influence the training of your model in any way. You should never be working with it before it comes time to test the model. Doing so will make the test error no longer an unbiased estimator and it too might continuously decrease with model complexity, masking signs of overfitting.

Keep in mind that to be a good estimator of the true error, the testing dataset must be sufficiently small. Similarly, to get a good estimate of $\hat{w}$ the training dataset must be large enough. Be sure to properly balance these two requirements.

1.3.1 3 sources of error

Irreducible Error

Suppose there is some underlying true distribution that our data comes from, but our data measurements are noisy samples from this distribution. Specifically:

$$y_i = f_{true}(x_i) + \epsilon$$

No matter how good we made our model, even if we were able to discover f_{true} somehow (our best possible guess), we could never predict this random noise. As a result, there will always be errors in our predictions resulting from this. For this reason, this kind of error is called **irreducible error**. Formally, we can define this error as:

$$\sigma^2(x) = E_{y|x}[\epsilon^2] = E[(y - f_{true}(x))^2]$$

The irreducible error can vary with x if ϵ itself varies with x. This is why we define it separately at each x value.

Bias

Bias is essentially a measure of how well the model class we choose (i.e. a linear fit, quadratic fit, etc) can capture the true distribution f_{true} (on average). If we choose too simple of a model (i.e. a linear fit for data that has a quadratic trend) then no matter how well we choose the parameters of the model there are always going to be these systematic errors in our prediction. If we choose a complex enough model, f_{true} is contained in the model class and thus if we chose the best possible parameters, we would perfectly fit the true distribution. Thus, **bias is inversely related to model complexity**. Formally, bias is defined as:

$$Bias(x) = f_{true}(x) - E_{train}[\hat{f}_{train}(x)]$$

Note that the bias is defined for each x individually. It is a measure of the difference between the true model and the expected model we generate from whatever random training set we get.

Variance

Variance describes how variable the model generated is with respect to the training dataset. It is the variance in the random variable $\hat{f}(x)$ which is random with respect to the randomly chosen training dataset. If I change the training dataset, does the model change significantly or not?

Variance has a positive relationship with model complexity. Increase the model complexity and the variance tends to increase. Formally, the variance at x is defined as:

$$var(\hat{f}(x)) = E\left[\left(\hat{f}(x) - E[\hat{f}(x)]\right)^2\right]$$

All three of these contribute to errors in the prediction. Thus, the average prediction error at x is:

$$E_{train,y|x}[(y - \hat{f}(x))^2] = \sigma^2 + (bias(\hat{f}(x)))^2 + var(\hat{f}(x))$$

Note that the average prediction error is random with respect to both the y value (i.e. the random noise ϵ) and the randomly chosen training set.

$$MSE[\hat{f}_{train}(x)] = E_{train}[(f_{true}(x) - \hat{f}(x))^2] = bias^2 + var$$

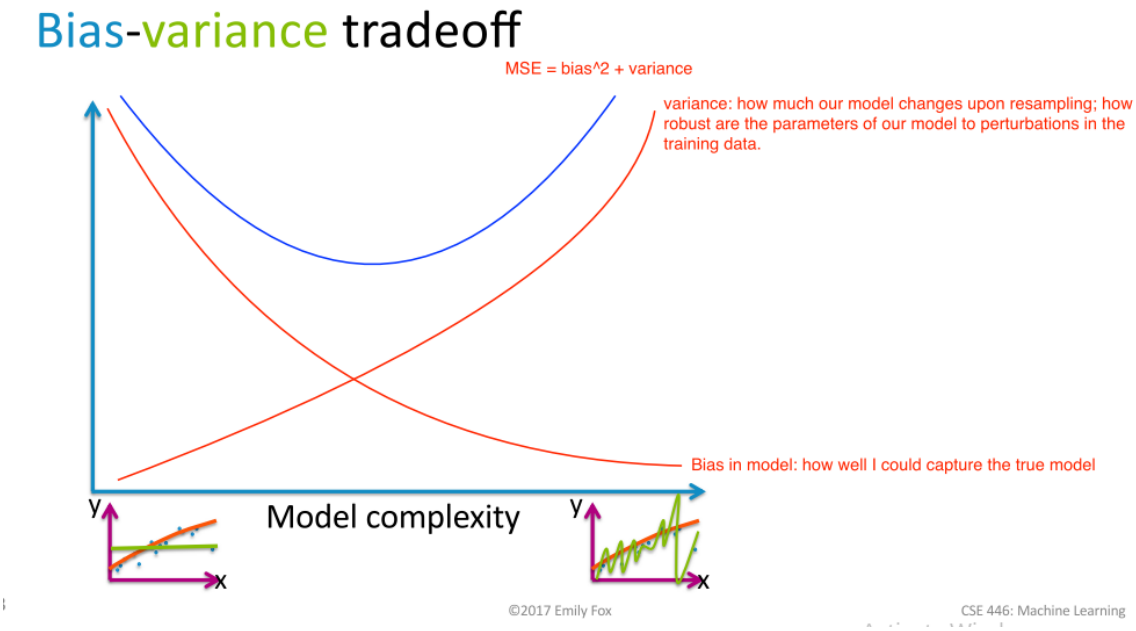

Figure 2: Bias Variance tradeoff

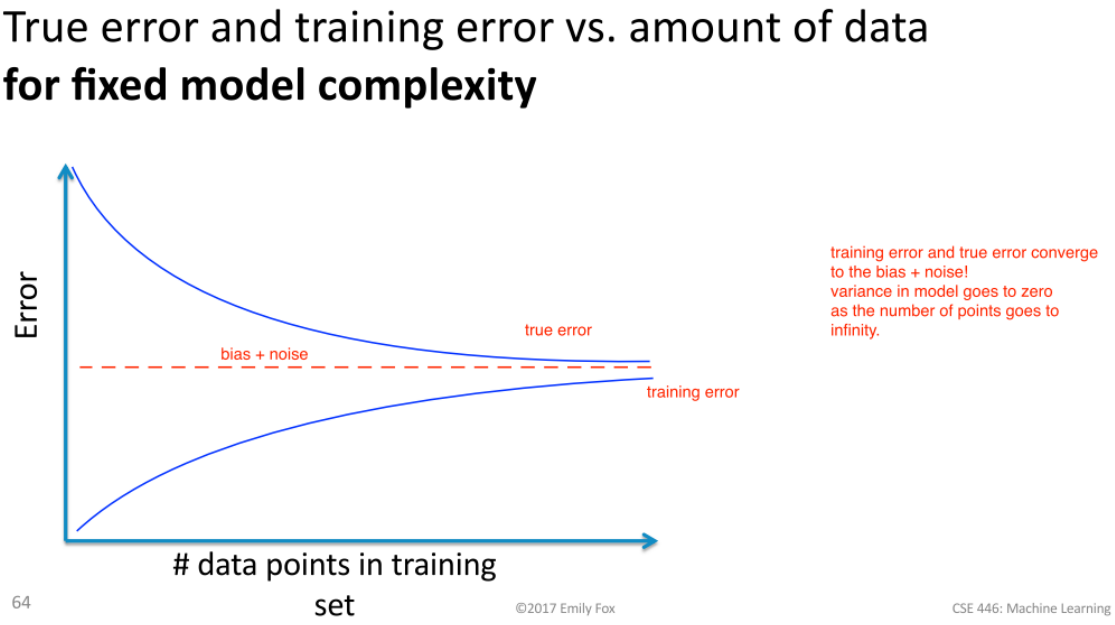

Figure 3: Bias Variance tradeoff

1.4 Convexity and Norms

A **norm** is any function $f(x) = ||x|| : \mathbb{R}^n \to \mathbb{R}$ with the following properties:

1. **Non-negativity:** $||x|| \geq 0$ for all $x \in \mathbb{R}^n$ and $||x|| = 0$ iff $x = \vec{0}$
2. **Absolute scalability:** $||\alpha x|| = |\alpha|||x||$ for all $x \in \mathbb{R}^n$, $\alpha \in \mathbb{R}$
3. **Triangle Inequality:** $||x + y|| \leq ||x|| + ||y||$ for all $x, y \in \mathbb{R}^n$

A very special family of norms is the L_p norm $||x||_p$. This is define as:

$$\boxed{||x||_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}}$$

We say that a set C is a **convex set** if:

$$x, y \in C \to \lambda x + (1 - \lambda)y \in C \quad \forall \lambda \in [0, 1]$$

In words, this says that the line connecting any two points in the set is completely contained within the set.

We say that a function $f : \mathbb{R}^n \to \mathbb{R}$ is a **convex function** if:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y) \quad \forall \lambda \in [0, 1], x, y \in \mathbb{R}^n$$

In words, this says that the secant line between any two points on the function surface is always above the function between those two points.

Properties

- Sum of convex functions is convex.
- all norms $||x||$ are convex
- if $g(x)$ is a concave function then $-g(x)$ is convex.
- Any local minimizers of a convex function are also global minimizers (important for our iterative min-finding algorithms like gradient descent; how do we know our endpoint is a global min rather than just a local one if our function is not convex?).

1.5 Regularization

Regularization is the process of restricting solution space by placing extra constraints on possible solutions. This allows us to enforce desirable qualities in the solutions such as sparsity or small coefficient values.

1.5.1 Ridge Regression

A problem arises when systems start to become underdetermined: when there are more features/inputs than there are data measurements ($d > n$). The objective function is flat in some directions and there are many possible solutions $\hat{w}$. As a result, small changes in the training dataset can have large changes in the predicted $\hat{w}$ and the coefficients in $\hat{w}$ can become very large. As a result of this latter issue, small changes in the input can drastically change the predictions of the model. To rectify this, we add an extra constraint: an L_2 norm regularization. Thus, the standard LLS problem becomes:

$$\hat{w}_{ridge} = \underset{w \in \mathbb{R}^d}{\arg\min} \left(\sum_{i=1}^{n} (y_i - w^T x_i)^2 + \lambda ||w||_2^2 \right)$$

With some algebraic manipulation, we can rewrite this as an LLS problem. Thus, its solution can be determined exactly via the normal equations. This yields

$$\boxed{(X^T X + \lambda I)\hat{w}_{ridge} = X^T y}$$

Note that as $X^T X$ is positive semidefinite always and λI is positive definite for all $\lambda > 0$, the matrix $X^T X + \lambda I$ is invertible and thus the solution $\hat{w}_{ridge}$ is unique (if $\lambda > 0$).

If we make some assumptions about our system, we can make some qualitative observations about the behavior of the solutions to ridge regression. These assumptions are not necessarily valid in general, but the give us some idea of the behavior of ridge regression. If we assume that $X^T X = nI$ then we see:

$$\hat{w}_{ridge} = \frac{n}{n + \lambda} \hat{w}_{LLS}$$

where $\hat{w}_{LLS}$ is the solution that would be obtained from the unregularized Linear Least Squares regression problem. Here, we see that λ is actively pushing the solution towards zero, discouraging large coefficient values. This is why Ridge Regression is said to have **shrinkage** properties. In general, there is not this clear inverse relationship between λ and the coefficient magnitudes, but it still captures the rough trend. If we also assume that $y = f(x) + \epsilon = Xw + \epsilon$ with $\epsilon \sim N(0, \sigma^2 I)$ then we can see:

$$\hat{w}_{ridge} = \frac{n}{n + \lambda} + \frac{1}{n + \lambda} X^T \epsilon$$

$$E[pred\_error] = E[(y - x^T \hat{w}_{ridge})^2 \mid X = x] = \sigma^2 + \frac{\lambda^2}{(n + \lambda)^2} (w^T x)^2 + \frac{d\sigma^2 n}{(n + \lambda)^2} ||x||_2^2$$

Here, the first term is the irreducible error, the second term is the $bias^2$, and the third term is the variance. We can see that as $\lambda \to \infty$ the $bias^2$ goes to $(w^T x)^2$ and the variance goes to

zero. As we saw earlier, as $\lambda \to \infty$ then $\hat{w}_{ridge} \to 0$. Thus, the variance goes to zero as you always predict $\hat{w}_{ridge} = 0$ regardless of the data and the bias goes to $(w^T x)^2$ as this is the squared difference between the expected prediction, which is always zero as $\hat{w}_{ridge} = 0$, and the truth $w^T x$. As $n \to \infty$, both the $bias^2$ and the variance go to zero: if you have infinite data, we can always generate a perfect model (up to the irreducible error).

1.5.2 LASSO

LASSO is just another kind of regularized regression. Formally, it is defined as:

$$\hat{w} = \arg\min_{w \in \mathbb{R}^d} \left(\sum_{i=1}^{n} \left((y_i - x_i^T w)^2 \right) + \lambda ||w||_1 \right)$$

Promoting Sparsity
One of the main benefits of using L_1 regularization in procedures like LASSO is that it tends to promote sparsity in the solution: it generates solutions $\hat{w}_{lasso}$ with few nonzero coefficients. Not only does this appear to make the model more interpretable (we could call all the features associated to nonzero coefficients "important" and all the others irrelevant [1]) but it also makes evaluating the model more computationally efficient. We could throw away all the features that had zero coefficients and just use the nonzero ones as our predictors.

If all we care about is the number of nonzero entries, why not use the L_0 norm which calculates exactly that? Well, for all $p \geq 1$ the L_p norm is convex and as we have talked about before convex optimization is much more well-understood and easier to work with than nonconvex problems. Essentially, the L_1 norm is a proxy for the L_0 norm that maintains the important property of convexity.

Coordinate Descent: Solving a LASSO problem

Here, the idea is to improve our $\hat{w}$ one coordinate at a time. For each coordinate, we look how far we should travel down that direction. Keep iterating that until we reach an optimal solution. Unfortunately, due to the L_1 norm which relies on absolute value functions, our objective function is no longer differentiable everywhere. To handle this, we will introduce a generalization to the gradient called the **subgradient**. A vector g is called a subgradient of a CONVEX function f at x if:

$$f(y) \geq f(x) + g^T(y - x) \quad \forall y$$

Note in the case that f is differentiable at x, we see the subgradient MUST be the gradient. If f is not differentiable, g can be several different vectors. As with the gradient, **the function f is minimized at x if the subgradient can take on the value zero at x** (it may be able to take on other values at x as well). Now, we are ready to discuss coordinate descent.

First, note that we can rewrite the LASSO minimization problem in a coordinate-by-coordinate way as seen in the figure above. Now, we are able to find the optimal value for a single coordinate w_j using the subgradient. It is easy to verify that the subgradient for w_j is:

[1] Although we should be careful in these interpretations. Remember, the feature you are measuring might not truly be important to the response, but simply highly coorelated to some other feature that actually is. Just because firetrucks and fires tend to come together does not imply firetrucks cause fires.

Fix any $j \in \{1, \ldots, d\}$

$$\sum_{i=1}^{n} \left(y_i - x_i^T w\right)^2 + \lambda ||w||_1 = \sum_{i=1}^{n} \left(y_i - \sum_{k=1}^{d} x_{i,k}\, w_k\right)^2 + \lambda \sum_{k=1}^{d} |w_k|$$

$$= \sum_{i=1}^{n} \left(\left(y_i - \sum_{k \neq j} x_{i,k}\, w_k\right) - x_{i,j}\, w_j\right)^2 + \lambda \sum_{k \neq j} |w_k| + \lambda |w_j|$$

this term is fixed with respect to w_j so we can simply throw it out of the minimization problem

Initialize $\widehat{w}_k = 0$ for all $k \in \{1, \ldots, d\}$

Loop over $j \in \{1, \ldots, n\}$: minimize in jth direction.

$$r_i^{(j)} = y_i - \sum_{k \neq j} x_{i,k} \widehat{w}_k$$

$$\widehat{w}_j = \arg\min_{w_j} \sum_{i=1}^{n} \left(r_i^{(j)} - x_{i,j}\, w_j\right)^2 + \lambda |w_j|$$

Figure 4: Rewriting the LASSO minimization problem into a form that can be used in coordinate descent. Pretty much just separates the minimization problem into cases for each coordinate w_j separately.

$$\delta_{w_j} \left(\sum_{i=1}^{j} (r_i^{(j)} - x_{ij} w_j)^2 + \lambda |w_j|\right) = \begin{cases} a_j w_j - c_j - \lambda & w_j < 0 \\ [-c_j - \lambda, -c_j + \lambda] & w_j = 0 \\ a_j w_j - c_j + \lambda & w_j > 0 \end{cases}$$

$$a_j = 2 \sum_{i=1}^{n} x_{ij}^2 \qquad c_j = \sum_{i=1}^{n} r_i^{(j)} x_{ij}$$

Note that the term $r_i^{(j)}$ does not depend on w_j by definition, so find the derivative of the first term (the sum) is simple.

$$\delta_{w_j} \sum_{i=1}^{n} \left(\sum_{i=1}^{j} (r_i^{(j)} - x_{ij} w_j)^2\right) = \sum_{i=1}^{n} -2 x_{ij} (r_i^{(j)} - x_{ij} w_j)$$

$$= 2 w_j \sum_{i=1}^{n} x_{ij}^2 - 2 \sum_{i=1}^{n} r_i^{(j)} x_{ij}$$

$$= a_j w_j - c_j$$

It is that pesky absolute value that complicates the subgradient. We can see that when $w_j < 0$ we have $\lambda|w_j| = -\lambda w_j$ and thus $\delta_{w_j}(-\lambda w_j) = -\lambda$. When $w = 0$ we are at the peak of the absolute value. The $a_j w_j$ goes to zero and as we saw any value in $[-1, 1]$ is a suitable subgradient to $|w_j|$ so any value between $[-\lambda, \lambda]$ is suitable for the subgradient of $\lambda|w_j|$. The $-c_j$ term still reamins giving us our final interval. By the same logic as above, we can find the case for $w_j > 0$.

Now, we know the best choice for w_j, the one that minimizers the loss function, is the one where the subgradient can be zero (it might be able to take on other values). We can see that if $|c_j| < \lambda$ then clearly zero lies in the interval $[-c_j - \lambda, c_j + \lambda]$ so choosing $w_j = 0$ will suffice (as this will put us in the middle case for the subgradient above). Now, suppose the optimal $w_j < 0$. This would put us in the first case of the subgradient, $\delta_{w_j} = a_j w_j - c_j - \lambda$, and we can see this is zero exactly when $w_j = \frac{c_j + \lambda}{a_j}$. But if w_j equals this, $w_j < 0$ iff $c_j < -\lambda$ as a_j is clearly positive. Thus, if $c_j < -\lambda$ we will choose $w_j = \frac{c_j + \lambda}{a_j}$. Lastly, suppose $w_j > 0$. This puts us in the third case for the subgradient $\delta_{w_j} = a_j w_j - c_j + \lambda$. We can see this is zero exactly when $w_j = \frac{c_j - \lambda}{a_j}$. Again if we set w_j in this way, w_j would only be positive if $c_j > \lambda$. This describes our algorithm for defining w_j:

$$w_j = \begin{cases} \frac{c_j + \lambda}{a_j} & c_j < -\lambda \\ 0 & |c_j| \leq \lambda \\ \frac{c_j - \lambda}{a_j} & c_j > \lambda \end{cases} \tag{10}$$

In this way, we can see how λ enforces sparsity: larger λ values expand the region where w_j would be set to zero! We can repeat this coordinate descent, looping for the coordinates one at a time then repeating the process, until our $\hat{w}$ stops changing.

1.5.3 Logistic Regression

Linear/Ridge/Lasso regression is great for trying to predict a continuous response variable from some input, but what if you were more concerned with classification: placing each datapoint into one of several categories. Instead of a continuous response variable, the variable would represent one of k different categories $y \in \{1, 2, ..., k\}$. Here, what you really want to know is:

$$P(Y = y \mid X = x)$$

This asks: Given some input of features x, what is the probability that the datapoint comes from category y? A natural way to estimate this would be to use the definition of conditional probability. We count the number of times we see x and y occur together and divide it by the number of times x occured at all:

$$P(Y = y \mid X = x) = \frac{count(x \cap y)}{count(x)}$$

However, this has an issue. If $x \in \mathbb{R}^d$, even if x is discrete and its entries are binary (i.e. $x = [1, 0, 1, 1, 0, .., 1]$ or something of the sorts) there are 2^d possible x! Unless you have infinite data, it's likely you there will be some x you never see in your data. So how would you estimate these probabilities? How would you know the probability of y given x if you never see x occur? We will go back to our linear regression but this time use a **link function**, a function $f :\to \mathbb{R} \to [0, 1]$

which squishes all numbers into the range $[0, 1]$ giving them an interpretation as a probability. One example of a link function is the **sigmoid**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

1.6 Cross-Validation

We have seen some pretty cool techniques using regularization, but in all of them we have seen the results depend heavily on our choice of the hyperparameter λ. But how do we choose λ? The answer is cross-validation!

As we showed in the regression subsection, the main issue lies in the fact that our training error is not an unbiased estimator of the true error: the expected error across all possible datapoints. In fact, it tends to underestimate the true error. However, we saw that the testing error WAS an unbiased estimator of the true error because it played no part in training the model. Cross-validation follows the same philosophy: we set aside some data and use that to gauge the performance of our model and hence evaluate our choice of λ. The ideal λ minimizes the error on this dataset so we repeat this several times while varying λ to find a good choice.

1.6.1 Leave One Out Cross-Validation

Fix λ. Suppose you have a training dataset T. Remove one datapoint x_j and train a model $\hat{f}_{T \setminus x_j}$ on the remaining $(n-1)$ datapoints. Determine the loss (squared error) in predicting y_j using x_j and this model $\hat{f}_{T \setminus x_j}$. Do this for every datapoint in the dataset and average the errors to get an approximation of the true error

$$error_{LOO} = \frac{1}{n} \sum_{j=1}^{n} (y_j - \hat{f}_{T \setminus x_j}(x_j))^2 \tag{11}$$

Repeat this process across many different λ values and choose the one that minimizes the leave one out error $error_{LOO}$. Assuming $n >> 1$ then our model $\hat{f}_{T \setminus x_j}$ does not change significantly between each iteration j. As a result, the $error_{LOO}$ is almost an unbiased estimator of the true error. However, a huge downside of this approach is you have to train a model on EVERY datapoint which can be really slow especially if iterative algorithms are used.

1.6.2 K-fold Cross-Validation

downsides = less training data due to validation dataset.

1.7 Gradient Descent

Gradient descent is an algorithm for iteratively finding a local minimizer of some differentiable function f. The main idea is that the gradient points in the direction of greatest increase so simply move in the opposite direction:

$$x_{k+1} = x_k - \eta \nabla f(x_k)$$

where η is the learning rate, a hyperparameter controlling the step size. If η is too high, gradient descent will "leap over" the minima and never converge. If η is too low, gradient descent will be slow to converge. In the context of machine learning, the function to be optimized is some sort of loss function over all samples in the training set:

$$f(w) = \sum_{i=1}^{n} loss((x_i, y_i) \mid w) + \lambda ||w||_2^2$$

One example of a loss function is the squared error $loss((x_i, y_i) \mid w) = (y_i - w^T x_i)^2$. And of course we can use different kinds of regularizers like an L_1 norm instead of an L_2 one. Often, the losses are averaged to remove the dependence of $f(w)$ on the number of training samples n:

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} loss((x_i, y_i) \mid w) + \lambda ||w||_2^2$$

Below are some common gradients that can be used in gradient descent. In each case $X \in \mathbb{R}^{n \times d}$ is the data matrix with each measurement stored as a row and y is a column vector of the response variable:

$$\nabla_w LLS(w) = -2X^T(y - Xw)$$
$$\nabla_w Ridge(w) = -2X^T y + 2(X^T X + \lambda I)w$$

1.7.1 Stochastic Gradient Descent

The true gradient is the sum (or average if loss is normalized with $\frac{1}{n}$) of the gradient across many datapoints in the training sample. This can become computationally inefficient to compute as the number of datapoints n becomes large. The full gradient across all training datapoints gives the direction of steepest decrease in the training loss function, but there are likely many directions that will still decrease the function value. If we approximate the full gradient across the entire training dataset with just a few datapoints, we are more likely than not to find a descent direction (opposite to this approximated gradient) as we know the true mean must be the full gradient. Since this only requires a few datapoints the update is much faster. We call this subsample of the training data a **batch** and its size the **batchsize**.

Support Vector Machines (SVM)

So far, the classification techniques we have covered like logistic regression have simply aimed to generate a hyperplane which separates the data into the two classes. However, often times there could be many hyperplanes that perfectly separate the data or do any equally good job of doing so. SVM seeks to expand upon this challenge by finding a hyperplane that comes with a margin, an error of space surrounding the plane that has no points in it. The main idea here is that this extra space separates the two classes by as much as possible, making it less likely any new points coming from one distribution jump across the hyperplane and gets classified in the other class. Figure 5 provides a visual of what this process looks like.

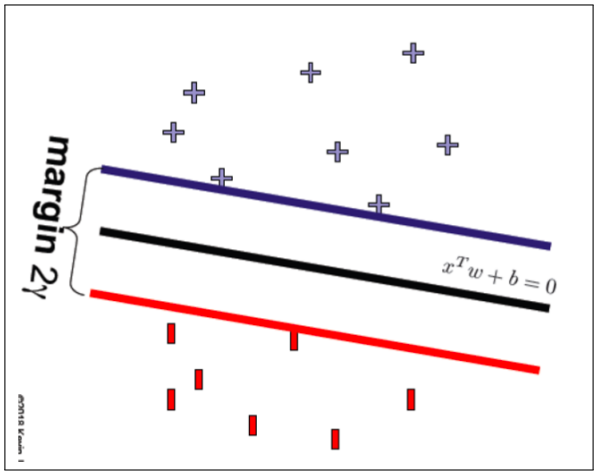

Figure 5: SVM: classifying hyperplane with a margin γ on both sides

Thus, our goal is the maximize the size of this margin γ while still correctly classifying all the points (and not letting any points be inside the margin). Formally, this is:

$$\begin{aligned} &\max_{w,b} \gamma \\ &st \quad \frac{1}{||w||_2} y_i(w^T x_i + b) \geq \gamma \quad \forall i \end{aligned}$$

Here, the constraint enforces the margin: no points can be incorrectly classified by the hyperplane defined by w, b, otherwise the sign of the LHS is negative and certainly less than γ, and no points can be closer to the hyperplane than γ. It turns out we can rewrite this system in a more convenient form:

$$\begin{aligned} &\min_{w,b} ||w||^2 \\ &st \quad y_i(w_i^T x_i + b) \geq 1 \quad \forall i \end{aligned}$$

Now, our system no longer depends on the variable γ which was unrelated to the classification procedure.

Of course, all this relies on the fact that the data is linearly separable. Otherwise, there would no way to build a margin to separate the classes. We could try to use a feature map or kernelize the SVM algorithm (see next section), but these might just overcomplicate a model that is roughly linearly separable plus an outlier or two. Instead, we could also try to relax our margins by adding **slacks** ξ_i for each datapoint which describe how far the datapoint is from being on the correct side of the margin. Specifically:

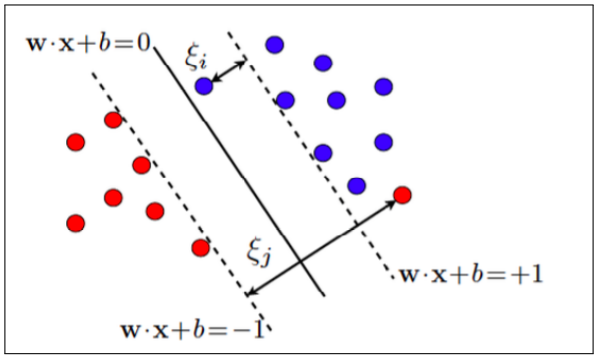

Figure 6: The slacks ξ_i for incorrect datapoints outside their side of the margin.

$$\xi_i = max(0, 1 - y_i(w^T x_i + b)) \tag{12}$$

Now, we can reframe our optimization problem to consider these slacks. Let C be a constant that adjusts how important it is to avoid these slacks:

$$\begin{aligned} & \max_{w,b,\xi} ||w||^2 + C \sum_i \xi_i \\ st \quad & y_i(w_i^T x_i + b) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned}$$

This optimization problem cannot be solved exactly, but as usually we can apply one of our iterative techniques like gradient descent. Thus, the final SVM (with slacks) minimization problem is:

$$\min_{w,b} loss(\{x_i, y_i\}_{i=1}^n) = \min_{w,b} \frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b)) + \lambda ||w||_2^2 \tag{13}$$

This kind of loss function, $max(0, 1 - y_i(w^T x_i + b))$, is called the **hinge loss function**. As it is not differentiable when $1 - y_i(w^T x_i + b) = 0$ we will have to apply subgradient descent. Here is the subgradient for hinge loss.

$$\partial_w = \begin{cases} 0 & y_i(w^T x_i + b) > 1 \\ [-y_i x_i, 0] & y_i(w^T x_i + b) = 1 \\ -y_i x_i & y_i(w^T x_i + b) < 1 \end{cases} \tag{14}$$

Since we can choose any suitable subgradient, we will just choose $-y_i x_i$ for the case when $y_i(w^T x_i + b) = 1$. Thus we can concisely write it as:

$$\partial_w = -(y_i x_i) I\{y_i(w^T x_i + b) \leq 1\}$$

This yields the stochastic gradient descent update with batch size $|batch|$ and learning rate η:

$$w_{k+1} = w_k + \frac{\eta}{|batch|} \sum_{i \in batch} \left(y_i x_i I\{y_i(w^T x_i + b) \leq 1\} \right) - 2\eta\lambda w_k$$

Kernel Trick

A lot of the classifications algorithms we have showed so far, and will show later, often rely on data being linearly separable. However, in reality we know there are many kinds of data where the classes are poorly separable in a linear sense. For example, consider the following simple example in Figure 7.

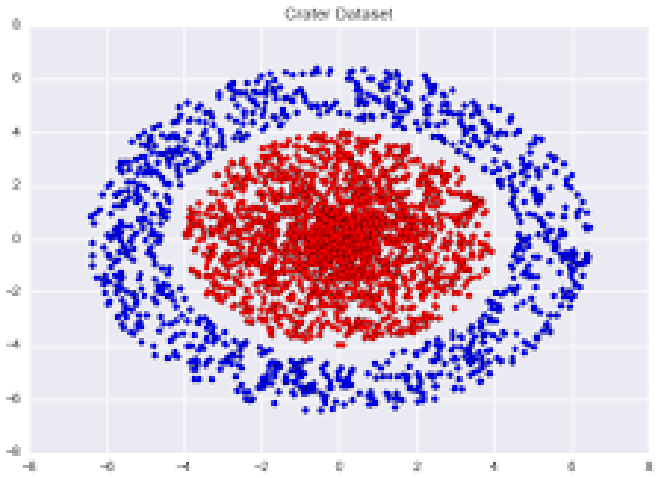

Figure 7: A dataset which is clearly not linearly separable. However, if we apply the feature map $(x, y) \to (x, y, x^2 + y^2)$ it will be separable in this new space.

Typically, we handle this issue by using a feature map to project the data into some other space where the data is linearly separable. In the case of Figure 7 a good feature map may be:

$$(x, y) \to (x, y, x^2 + y^2)$$

However, this projection often increases the dimensionality of the input as many new features may be needed to make the data linearly separable. In doing so, however, the classification algorithms start to run into the curse of dimensionality and become too slow to utilize. For example, if we had only two variables (u, v) and we wanted use a feature map for all polynomials of degree at most d, $(u, v) \to (1, u, v, uv, u^2, v^2, ..., u^k v^{d-k})$, then our feature map would have $\binom{d+2}{2}$ terms. In general, if your input is size n then there will be $\binom{d+n}{n}$ (this is just a weak composition of d into $n+1$ parts: one part for each input and an extra part to hold the degrees that aren't used). Quickly then, these problems become intractable, but we still want to be able to map into these high dimensional spaces. The Kernel trick finds a workaround to this problem.

Let $\phi(x)$ be our feature vector x mapped into the high-dimensional space we want to work in (i.e. polynomials of degree at most degree d as in the example above). The Kernel trick relies on one fundamental assumption: whenever we use $\phi(x)$ it is only part of an inner product. This may seem restrictive, but many of the classification algorithms we have seen so far rely on these inner products (SVM, perceptron training, etc). It just so happens that $\langle \phi(x_i), phi(x_j) \rangle$ can often be much easier to compute. For example, in the polynomial basis of degree at most d discussed above it just so happens that for $x, y \in \mathbb{R}^n$:

$$\langle \phi(x), \phi(y) \rangle = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \dots \\ x_n \\ x_1 x_2 \\ \dots x_1^3 x_4^7 \\ \dots \end{bmatrix} \bullet \begin{bmatrix} 1 \\ y_1 \\ y_2 \\ \dots \\ y_n \\ y_1 y_2 \\ \dots y_1^3 y_4^7 \\ \dots \end{bmatrix} = C(1 + \langle x, y \rangle)^d \qquad C \in \mathbb{R}$$

Define $K(u, v) = \langle \phi(u), \phi(v) \rangle$. Then we call K the **kernel function**. Above, the kernel function is $(1 + \langle u, v \rangle)^r$.

Theorem 1 *Mercer's Theorem*

$K(x, x')$ is a valid kernel if there exists a $\phi(x)$ such that $K(x, x') = \langle \phi(x), \phi(x') \rangle$

Furthermore, $K(x, x')$ is a valid kernel IFF $\mathbf{K}$ is symmetric and positive semi-definite for any pointset $(x_1, ..., x_n)$, where $\mathbf{K}_{ij} = K(x_i, x_j)$

Common Kernel Functions

- Polynomials of degree exactly d

$$K(u, v) = (\langle u, v \rangle^d)$$

- Polynomials of degree at most d

$$K(u, v) = (1 + \langle u, v \rangle^d)$$

- Gaussian Kernel (aka Radial Basis Function (RBF))

$$K(u, v) = exp\left(\frac{||u - v||^2}{2\sigma^2}\right)$$

- Sigmoid Kernel

$$K(u, v) = tanh(\eta \langle u, v \rangle + \mu)$$

Kernelizing a Machine Learning Algorithm

1. Prove that a solution (i.e. optimal classifier $\hat{w}$) is always in the span of the training points.
2. Rewrite algorithm so that all training/test datapoints are accessed only through an inner product with other datapoints.
3. Choose a kernel function and substitute $x_i^T x_j$ with $K(x_i, x_j)$ in the training algorithm.

1.7.2 Example: Perceptron Training Algorithm

See Neural Networks section (2) for a reference to what this algorithm aims to accomplish.

1. In the perceptron training algorithm, we update the weights w in response to our mistakes. Specifically, if I is the set of indices where mistakes were made (i.e. point x_i was classified $\hat{y}_i \neq y_i$) then if we initialize $w_0 = \vec{0}$:

$$w = \sum_{i \in I} y_i x_i$$

   Thus clearly the solution w is a linear combination of the training datapoints.

2. Thus our prediction rule for a new datapoint x_{new} becomes:

$$\hat{y} = sign\left(x_{new} \bullet w\right) = sign\left(x_{new} \bullet \sum_{i \in I} y_i x_i\right) = sign\left(\sum_{i \in I} y_i \langle x_i, x_{new} \rangle\right)$$

3. Thus, we can kernelize this by replacing our prediction rule with:

$$\hat{y} = sign\left(w, \phi(x_{new})\right) = sign\left(\sum_{i \in I} y_i K(x_i, x_{new})\right)$$

1.7.3 Example: Kernelized Ridge Regression

As a reminder, ridge regression is defined to be:

$$\hat{w} = \arg\min_w \sum_{i=1}^{n} (y_i - w^T x_i)^2 + \lambda ||w||_2^2 = \arg\min_w ||y - Xw||^2 + \lambda ||w||_w^2$$

In the kernelized case under a feature map $\phi(x)$ this becomes (where $\Phi \in \mathbb{R}^{n \times p}$ is a matrix whose i^{th} row is $\phi(x_i)$):

$$\hat{w} = \arg\min_w ||y - \Phi w||_2^2 + \lambda ||w||_w^2$$

Note that we can write $\hat{w} = \Phi^T \hat{\alpha}$ with:

$$\hat{\alpha} = \arg\min_{\alpha} ||y - \Phi\Phi^T\alpha||_2^2 + \lambda\alpha^T\Phi\Phi^T\alpha = \arg\min_{\alpha} ||y - \mathbf{K}\alpha||_2^2 + \lambda\alpha^T\mathbf{K}\alpha \tag{15}$$

where $\mathbf{K}$ is the kernel function matrix: $K_{ij} = K(x_i, x_j) = \langle\phi(x_i), \phi(x_j)\rangle$. The argument for why we can do this is a bit nuanced (see slide 5 of lecture 12 CSE 446 for more details). The basic idea is if $\hat{w}$ was not in the span of the columns of Φ^T, then it must have a component orthogonal to this subspace. However, when we project $\hat{w}$ into Φ^T (i.e. during the multiplication $\Phi^T w$) we lose this component. As such, this component won't affect the squared error term and only increases the L_2 norm term. Thus, we might as well have chosen the projection of $\hat{w}$ onto Φ^T as it performs the same on the squared error and has a smaller norm! Thus, $\hat{w}$ can only be optimal if its in the range of Φ^T. Regardless, from equation 15 above we can solve for $\hat{\alpha}$ explicitly by setting its gradient with respect to α equal to zero.

$$\hat{\alpha} = (\mathbf{K} + \lambda I)^{-1} y \tag{16}$$

Then, our prediction for a new datapoint x_{new} would be:

$$\hat{y} = \hat{w}^T\phi(x_{new}) = \hat{\alpha}^T\Phi\phi(x) = \boxed{\sum_{i=1}^{n} \hat{\alpha}_i K(x_i, x_{new})} \tag{17}$$

K-Nearest Neighbors

Linear classifiers are great and the use of properly chosen feature maps can introduce nonlinearities into the classifier, but this often greatly increases their complexity. The K-nearest Neighbors (k-NN) algorithm is a simple, nonlinear model for supervised learning. All it says is for a new datapoint, find its the k closest datapoints to it (using a distance measure like L_2, L_1, L_∞) and assign that point a label based on the most common label in all neighbors.

Properties

- Does not handle scaling of the features well: scaling one of the features can drastically change which points are "close" to each other. Thus, we have to be careful about how we normalize these features.
- In the limit of infinite data (where datapoints are dense in $\mathbb{R}^n$), the 1-NN algorithm makes at most twice as many errors as the optimal Bayes classifier.
- Finding nearest neighbors can be slow to compute.

Nearest Neighbor Regression

Create a model $\hat{f}(x)$ by finding the k nearest datapoints x_i to x and predicting the mean of their associated $f(x_i)$ values.

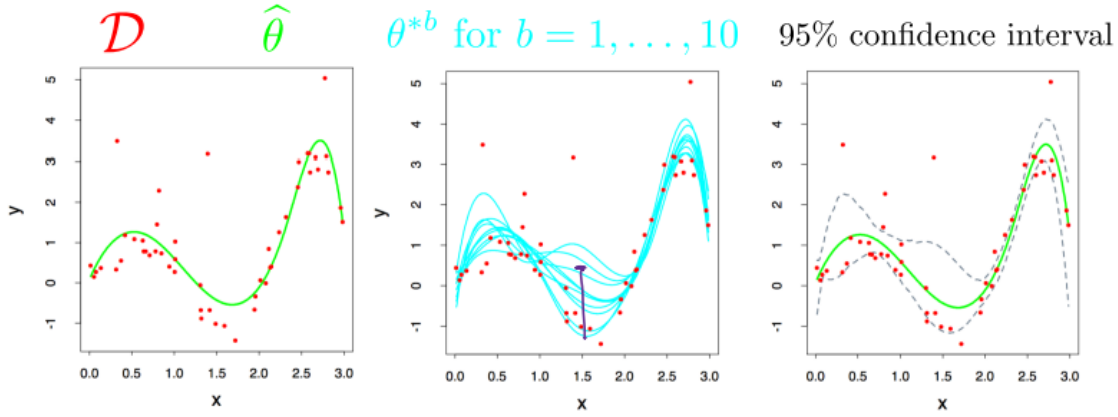

Figure 8: Building confidence intervals using the bootstrap technique. In the left image, we see the original dataset drawn from the overall distribution and the statistic $\hat{\theta}$, a regression model, generated for that data. In the middle, we see the same statistic calculated for 10 different bootstrap samples. In the right, we see the confidence intervals for this statistic from the bootstrap samples.

Bootstrap

Cross-validation is a great technique for trying to assess the true performance of a model, but it comes with several down-sides. First and foremost, a large chunk of the data is used up as part of the validation dataset. Secondly, it only gives you a single number and its hard to draw meaningful statistics from this like confidence intervals, the amount of error at a particular point x in feature space (likely the model performs better or worse in some areas compared to others), or information of the distribution of models. Bootstrap attempts to answer some of these questions.

Given a dataset $\mathcal{D} = \{x_i\}_{i=1}^n \sim F_X$ drawn iid from a distribution with CDF F_X we are going to generate B bootstrap samples. For $b = 1, ..., B$ sample n datapoints from our dataset $\mathcal{D}$ (NOT the overall distribution) **with replacement**. From this bootstrap sample, compute the statistic of interest (i.e. the mean, percentiles, etc). In doing this many times, we just a hint at the variation in the statistic as we see in Figure 8.

K-Means

A form of unsupervised learning for clustering. The basic idea is that we choose some number k of classes to look for (k different groups of data in our dataset) and we want to classify points based on a nearest-neighbor sort of approach. However, we do not know the labels of any of the points.

Algorithm 1 *Lloyd's Algorithm (K-means)*

1. *Initialize k points in $\mathbb{R}^n$ to be the initial means. Often, these are either chosen randomly or simply randomly chosen points from the dataset.*

2. *Partition all datapoints into groups based on which of the k means each datapoint is closest to.*

3. *Set the new means to be the mean of datapoints in each of the groups from step 2.*

4. *Repeat steps 2 and 3 until convergence (i.e. let the loss be the total squared distance from points to their respective means and wait for loss to stop changing)*

Note that the cost function (total squared distance between points and their corresponding means) is decreasing at every iteration!

PCA

The main idea of PCA is that data as it is collected is often not the best way to represent the system at hand. The data may be measuring the system from a subpar angle, it may be full of redundancies, and it may include features that are actually meaningless. PCA aims to construct a new coordinate system for the data that captures its variability (in a **LINEAR** sense) as good as possible. Specifically it means to find **principal components** $v_1, ..., v_d$ such that:

$$x_i \approx \alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_d v_d \rightarrow x_i = (\alpha_1, ..., \alpha_d)_{PCA} \qquad x_i, v_i \in \mathbb{R}^n$$

Thus, we rewrite x in the new coordinate system given by the principal components $\{v_1, v_2, ..., v_d\}$ as the coordinates $(\alpha_1, ..., \alpha_d)$. Note that $\alpha_i = \langle x, v_i \rangle$ as the v_i are orthogonal. In fact, in doing so it is often the case that a $d << n$ can be found that captures the data incredibly well, drastically reducing the dimensionality of the system. Not only does this make the data computationally easier to work with by reducing the number of operations required to manipulate it, making it a valuable preprocessing step before classification/learning, but it also reduces the number of parameters models generated on the data have, reducing the chance of overfitting. But how do we find these principal components v_i?

To find these principal components, we are going to build our coordinate system one vector, v_i, at a time. The overall goal is to find an **orthonormal** set of vectors $v_1, ..., v_d$ that maximizes the variance in our data $\{x_i\}_{i=1}^n$ captured by the subspace $S = span\{v_1, ..., v_d\}$. More precisely, we want to maximize the magnitude of the projection of $\{x_i\}_{i=1}^n$ onto S. Since the $v_1, ..., v_n$ are orthonormal, this is the same as maximizing the components of x_i in each of the vectors v_i. Thus:

$$\max_{v_1,...,v_d} \left(\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{d} \langle x_i, v_j \rangle \right) = \max_{v_1,...,v_d} \frac{1}{n} \sum_{j=1}^{d} (Xv_j)^T (Xv_j) \tag{18}$$

where X is a matrix whose rows are x_i. Note that this is equivalent to minimizing the overall distance between the subspace S and our data $\{x_i\}$ (the components of x_i NOT in the direction of v_i). When written in the matrix form, it is clear what PCA is doing. Let $A = \frac{1}{n} X^T X$, the empirical covariance matrix for our system (as we subtract the mean out of X). Naturally, A is symmetric in which case its eigenvector decomposition is $A = Q \Lambda Q^T$ where Q is an orthogonal matrix whose columns are eigenvectors of A and Λ is a diagonal matrix whose diagonal entries are the corresponding eigenvalues.

Theorem 2 *Finding Principal Components*

The first k principal components of a data matrix X whose rows are x_i are the first k eigenvectors (i.e. columns in Q) for the empirical covariance matrix (for the features) $A = \frac{1}{n}X^TX$ where A has eigenvector decomposition $A = Q\Lambda Q^T$ as its symmetric.

NOTE: It is important to remember to subtract means and normalize variances among the features before performing PCA

Singular Value Decomposition

The SVD is a matrix decomposition of the form:

$$X = U\Sigma V^*$$

for unitary matrices U, V (i.e. orthogonal matrices if X is real so $UU^T = U^TU = I$ and $VV^T = V^TV = I$). As U, V are unitary/orthogonal, the columns of U, V form a basis for the columnspace and rowspace of X respectively. Furthermore, note that:

$$XX^T = U\Sigma V^*V\Sigma U^* = U\Sigma^2U^*$$
$$X^TX = V\Sigma U^*U\Sigma V^* = V\Sigma^2V^*$$

Thus we can see that the columns of U are just the set of eigenvectors for the covariance matrix among the rows of X (typically data measurements) which is why it forms a basis for the columns of X. Furthermore, the columns of V are the set of eigenvectors for the covariance matrix among the columns of X (typically the features of the data) which is why it forms a basis for the rows of X. In both cases, Σ^2 are the eigenvalues so we can see $\sigma_i = \sqrt{\lambda_i}$.

Theorem 3 *Error Bounds*

Let $\hat{A}$ be an $n \times d$ matrix of random variables, each with variance bounded by σ^2. If

$$A = E[\hat{A}]$$

is rank k then with high probability

$$||A - \hat{A}_k||_F \in O(k\sigma^2(n + d)$$

Mixture Models and Expectation Maximization

Mixture models are a form of unsupervised learning. The main idea is that we assume our datapoints $\{x_i\}_{i=1}^n$ are coming from a set of distributions of a specific form (i.e. multivariate gaussians). The randomness of these distributions accounts for the spread of the datapoints in $\mathbb{R}^n$. The goal would then to be to figure out the parameters of these distributions and assign each point to the distribution it is most likely to come from. Suppose there are k distributions $f_i(x \mid \theta)$ parameterized by θ (which may be include multiple parameters, i.e. μ, Σ in the case of the multivariate gaussian). Let $z_{ij} = 1$ of datapoint $i \in \{1, 2, ..., n\}$ came from distribution $j \in \{1, 2, ..., k\}$ and zero otherwise. Then to goal is to maximize the likelihood of getting the datapoints from these distributions:

$$\max_{\theta,z} L(z,\theta) = \max_{\theta,z} \prod_{i=1}^{n} \prod_{j=1}^{k} P(x_i \mid x_i \sim f_i(x \mid \theta_i))^{I\{z_{ij}=1\}} \tag{19}$$

$$\max_{\theta,z} log(L(z,\theta)) = \max_{\theta,z} \sum_{i=1}^{n} \sum_{j=1}^{k} z_{ij} * log(P(x_i \mid x_i \sim f_i(x \mid \theta_i))) \tag{20}$$

Here, the indicator random variable just makes sure we only include the probability of getting x_i from the distribution x_i is determined to come from. Note that $I\{z_{ij} = 1\} = z_{ij}$. The only problem is that in order to determine the parameters θ, we need to know which points came from which distribution (we need to know z). However, in order to know which points came from which distribution, z, we need to know the parameters of the distributions. Quite the pickle! Our way around this is the Expectation Maximization Algorithm.

Expectation Maximization

The idea of the EM algorithm is to alternate between solving the two problems mentioned above: finding which points came from which distribution and finding the parameters of the distribution.

E step

Since we want to incorporate the idea of uncertainty in these assignments, rather than trying to approximate z_{ij} (i.e. predict exactly which distribution datapoint x_i came from) we will approximate the relative probabilities each point x_i came from each distribution j which we will call the **responsibility** r_{ij}. Unlike z_{ij} which is one for a single j and zero elsewhere, r_{ij} is a relative probability: how likely is it that point x_i came from distribution j relative to the other distributions?

$$r_{ij} = \frac{\pi_j P(x_i \mid x_i \sim f_j(x \mid \theta_j))}{\sum_{\ell=1}^{k} \pi_\ell P(x_i \mid x_i \sim f_\ell(x \mid \theta_\ell))} \tag{21}$$

M step

In this step, fix all the assignments/responsibilities r_{ij}. Once these are fixed, we want to adjust our parameters θ_j and π_j so that they maximize the probability of observing these assignments. π_j is simple and intuitive (same for all EM problems): we choose the probability of drawing a point from distribution j to be the average of the probabilities that a point in our sample was assigned to that distribution (r_{ij}).

$$N_j = \sum_{i=1}^{n} r_{ij} \tag{22}$$

$$\pi_j = \frac{N_j}{n} \tag{23}$$

Our next goal in this step is to choose these model parameters θ_j to maximize the likelihood/log-likelihood of observing the given datapoints. This is just a MLE problem

$$\max_{\theta} \sum_{i=1}^{n} ln \left(\sum_{j=1}^{k} \pi_j P(x_i \mid x_i \sim f_j(x_j \mid \theta_j))) \right) \tag{24}$$

Neural Networks

Theorem 4 *Neural networks with one hidden layer are arbitrary function approximators*

Perceptron

The perceptron is a simple linear classifier. Unlike other linear classifier (SVM? Logistic?)

Algorithm 2 *Perceptron Training Algorithm*

```
for i in range(n):
    # Observe x_i

    # make a prediction
    y_hat = sign(dot(w, x))

    # observe true label y_k in {1,-1}

    # Update the model if a mistake was made
    if not y_hat == y_k:
        w = w + y_k * x_k
```

Theorem 5 *Perceptron Mistake Limit*

If the feature vectors $\{x_i\}_{i=1}^{n}$ *have bounded norm,* $\max_i ||x_i|| \leq R$*, and the labels are linearly separable in the feature space, there exists some margin* γ *on the optimal separating hyperplane (think back to the margins in SVM), then the number of errors made in the perceptron training algorithm is bounded above by* $\frac{R^2}{\gamma^2}$

Decision Trees, Bagging, and Boosting

This is a form of supervised learning. The idea here is to build some sort of decision system to classify points. An example is shown in Figure 9. Essentially, for categorical features we just split the data into the different values that feature can take on, and for continuous features we choose some sort of threshold and split the data into above/below this threshold. Once we decide to stop, we simply decide to classify the point based on the majority remaining in that leaf node. For example, in the example of Figure 9 we could be trying to determine whether or not someone will pay back their loan. From previous experience/data, we see that 90% of people, a majority, who have poor credit and low income (the far right branch in the decision tree) they have failed to pay back their loan. Thus, in the future if another datapoint falls in this category we will predict they too will fail to pay back the loan and decide to reject their application.

Building A Decision Tree

Often times, you won't want to split on every possible feature as this tends to generate overly complicated decision trees. At this point, any datapoints in the same leaf node are exactly the same in all their categorical variables and roughly the same in their continuous. This might be too specific of a focus and may cause you to miss some overarching trends. Instead, we tend to prefer simpler decision trees. So how do you go about choosing good features to split on? We will use a greddy (but not optimal) algorithm as finding the best tree is NP-hard.

1. For some subset of the data at a given node in the tree (i.e. a node after previous splitting has already occurred) choose a feature of the data ϕ
2. Split the data according to that feature.
3. Predict the majority of response variable in each resulting branch.
4. Calculate the classification error (correctly classified) / (total at node)
5. Choose to split on the feature with the smallest classification error.
6. Repeat!
7. Stop when a given stopping condition is met:
    - no features left to split on
    - no classification error; all datapoints in each leaf node have the same response variable y
    - **Bad idea:** stop if classification error does not change during an iteration

To choose the threshold for continuous features, simply choose the threshold that produces the lowest classification error at that node (not necessarily lowest overall).

Avoiding Overfitting

- stop early; hard to know exactly when to stop however.
- make full tree then trim (pruning)

Bagging

Unfortunately, basic decision trees do not really work all that well. One possible solution is Bagging (Bootstrap aggregation). The idea here is that we use bootstrapping from our dataset to generate many different decision trees. When trying to make a prediction on a new datapoint, make a prediction for each of these trees and choose the majority decision (for classification; or average for regression). This too can give an estimate for the error. Let $\mathcal{D}_b$ be the b^{th} bootstrapped dataset (drawn from our dataset $\mathcal{D}$ with replacement). Then define the "out of bag" error to be:

$$\epsilon_b = \frac{1}{n} \sum_{(x_i, y_i) \in \mathcal{D}}$$

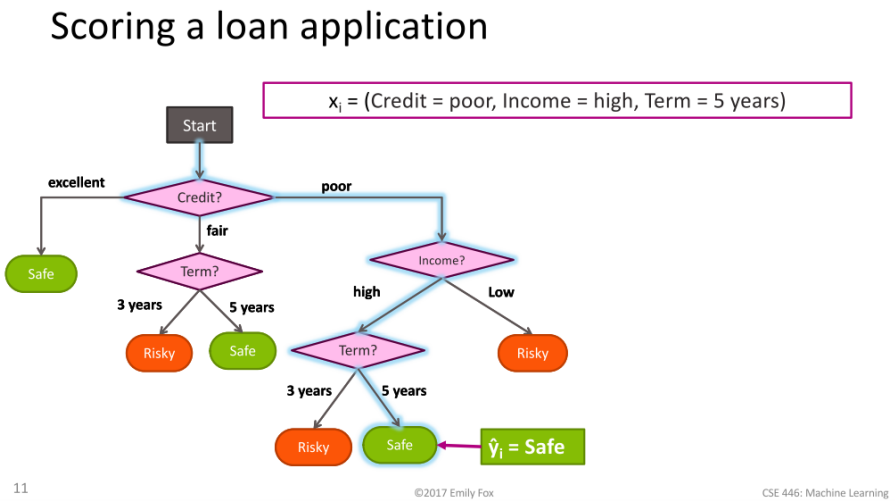

Figure 9: Caption

From this, we could also try to build an **ensemble classifier** which encorporates the (weighted) predictions from all these models? Let $\{f_i(x)\}$ be our different decision trees/classifiers.

$$\hat{y}_{new} = sign\left(\sum_{i=1}^{k} w_i f_i(x_{new})\right)$$

Random Forest

Same as bagging except when training a decision tree, for each split randomly choose some subsample of the features to consider for the split rather than considering all of them.

Boosting

Basic idea of an ensemble classifier: combine a bunch of simple, weak classifiers to make good one, shown in Figure 10. Boosting aims to improve this process by building each simple classifier using information from the previous classifiers. Specifically, learn where the previous classifiers are making a lot of errors and focus the next classifier on being good in those areas. How it does this is by associating weights α_i to each of the datapoints which are high if the current classifier makes a mistake predicting that datapoint and low otherwise. These weights α_i are passed on to the next classifier and affect how it calculates the classification error rate when building a decision tree. Rather than counting the number of mistakes in makes, it weights these mistakes. Let $\mathcal{D}'$ be the subset of datapoints at a given node, let $\hat{y}_i$ be the predicted response variable and y_i the true response. Then the weighted error for a specific feature split at that node is:

$$error = \frac{\sum_{i \in \mathcal{D}'} \alpha_i I\{\hat{y}_i \neq y_i\}}{\sum_{i \in \mathcal{D}'} \alpha_i}$$

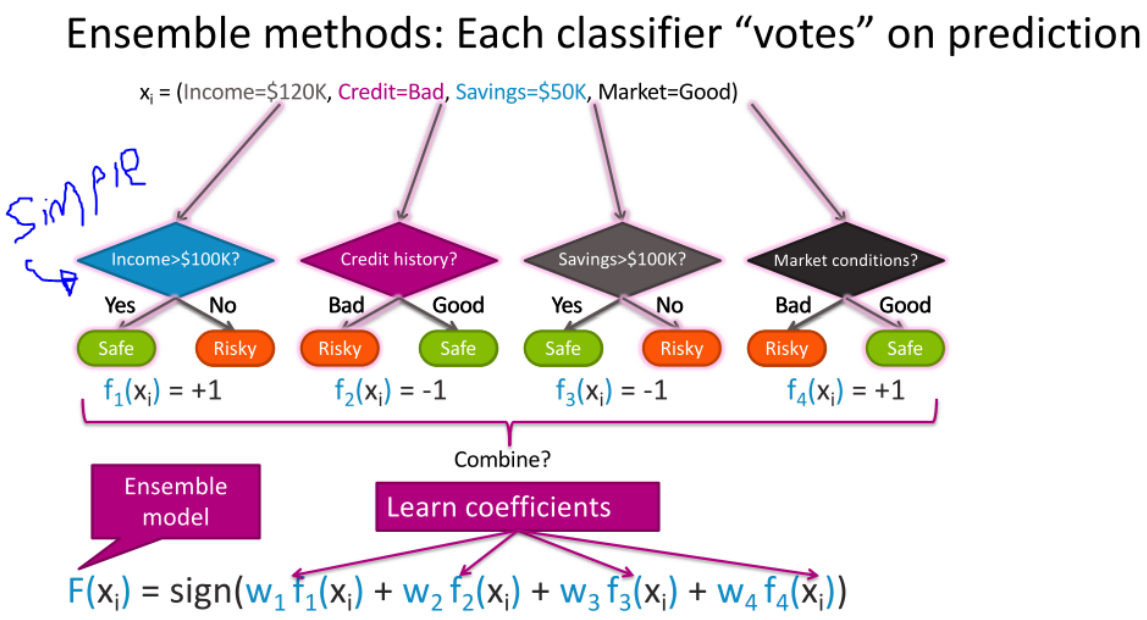

Figure 10: Combine the predictions of a bunch of simple classifiers.