Machine Learning Notes

February 19, 2018

I. Necessary background n sh't

A. Distance measures

When we think of distance in space, our brains usually default to euclidean distance, or straight line distance. However, there are actually infinitely many different ways to measure distance. In this section we briefly note some of the more useful distance measures in machine learning.

norm: *a function that assigns a strictly positive length or size to each vector in a vector space, save* $\vec{0}$*, which is assigned a length of zero.* (for a formal definition, google it.)

Euclidean Norm: The one we are all used to (straight line distance), denoted by $|\vec{x}|_2$, $||\vec{x}||_2$, L_2, or ℓ_2. For a given n-dimensional vector $\vec{x}$

$$||\vec{x}||_2 = \sqrt{\sum_{i=1}^{n} \vec{x}_i^2}$$

Manhattan Norm: denoted by subscript 1 ($||\vec{x}||_1$, ℓ_1, etc), the distance a taxi has to drive in a rectangular street grid to get from the origin to the point x. For a given n-dimensional vector $\vec{x}$

$$||\vec{x}||_1 = \sum_{i=1}^{n} |x_i|$$

p-norm: a generalization of the norms. $p = 1$ gives manhattan norm and $p = 2$ gives euclidean norm.

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

as $p \to \infty$, the p-norm approaches the infinity norm, or maximum norm: $\|\mathbf{x}\|_\infty := \max_i |x_i|$. TODO add figure (maybe more distance measures)

B. Calculus and Optimization

TODO

Linear Algebra

The rank is basically the number on non zero eigen value. A positive definite matrix if all eigenvalues are positive. Positive semidefinite matrix all eigenvalues are 0 or greater. A d x d positive semidefinite matrix could have rank less than d. We'll only talk about symmetric matrices (i.e. all eigen values are real).

C. Problems in Statistical Learning

- Regression: making a real valued prediction (e.g. predict house value given sq. feet, number of beds, number of baths) Examples are linear regression, k-NN where labels are continuous (we use average of k nearest neighbors to determine y), regression tree.
- Classification: predict a value within a finite set of possibilities. Examples are k-NN with plurality of k nearest neighbors, logistic regression, decision tree.
- Ranking: Put a set of objects in order of relevance.

C. Probability

Probability can be approached with two mindsets. One views probability theory as the study of frequencies and hence, is called the frequentist approach. A more general approach is called Bayesian. It views probability as a quantification of uncertainty. Jaynes 2003 showed that a Bayesian approach to probability could be regarded as an extension of boolean logic to situations involving uncertainty. The Bayesian approach is particularly useful in machine learning.

Let X, Y denote events. Then we have,

$$\textbf{sum rule:} \quad P(X) = \sum_{Y} P(X, Y)$$

$$\textbf{product rule:} \quad P(X, Y) = P(Y \mid X) P(X)$$

applying the product rule twice, we arrive at

$$\textbf{Bayes theorem:} \quad P(Y \mid X) = \frac{P(X \mid Y) P(Y)}{P(X)}$$

In the above, $P(X)$ is called the prior and $P(X \mid Y)$ is called the posterior. We also define

Independence: $P(X \mid Y) = P(X) \implies P(X, Y) = P(X)P(Y)$

1. Sample space, Random variables. Expectation, variance, covariance 2. Probability distributions (mass vs density). 3. Examples, bernoulli, poisson, gaussian.

D. Statistics:

TODO what is a statistic?

A *statistical model* $\mathcal{P} = \{P_\theta : \vec{\theta} \in \Theta\}$ is a family of probability distributions indexed by a set Θ called the parameter space. In a *parametric statistical model*, the distributions are indexed by a finite number of parameters (i.e. $\Theta \subset \mathbb{R}^k$ for some $k < \infty$). Examples: Bernoulli, Poisson, Gaussian, Multivariate Gaussian (MVD) distributions.

Model fitting: Suppose we have an observation x, and a parametric model $\mathcal{P}$ with parameter space Θ. Pick some $P_\theta \in \mathcal{P}$ (i.e. some $\vec{\theta} \in \Theta$) that best "fits" the observation x.

Maximum Likelihood Estimation: The most popular model fitting method. Say we have set $\mathbf{X}$ of points drawn from some distribution $P_{\vec{\theta}}$ (see next section). However, we don't know $\vec{\theta}$. How can we estimate it? If we had no data points, we would want to chose $arg\,max(P(\vec{\theta}))$, the $\vec{\theta}$ that maximizes the prior. But we do have $\mathbf{X}$, so let's use Bayes Theorem:

$$P(\vec{\theta} \mid \mathbf{X}) = \frac{P(\mathbf{X} \mid \vec{\theta})P(\vec{\theta})}{P(\mathbf{X})}$$

we want to maximize the posterior,

$$arg_\theta\, max(P(\vec{\theta} \mid \mathbf{X})) = arg_\theta\, max(\frac{P(\mathbf{X} \mid \vec{\theta})P(\vec{\theta})}{P(\mathbf{X})}) = arg_\theta\, max(P(\mathbf{X} \mid \vec{\theta})P(\vec{\theta}))$$

We make the further assumption that $P(\vec{\theta})$ is uniformly distributed over Θ. Finally, we arrive at:

$$arg_\theta\, max\, P(\mathbf{X} \mid \vec{\theta})$$

$P(\mathbf{X} \mid \vec{\theta})$ is called the *likelihood function* $\mathcal{L}$. This is why this method is called maximum likelihood estimation. To greatly simplify calculations, often allowing us to arrive at a closed form solution, we make the further assumption that the x's in $\mathbf{X}$ are independent. Note that this assumption is not strictly necessary. We have already assumed the points are drawn from the same distribution. Taking these two assumptions together, we say the x_i's are i.i.d or independent identically distributed. We can write

$$\mathcal{L} = P(\mathbf{X} \mid \vec{\theta}) = \prod_{i=1}^{n} P(x_i \mid \vec{\theta})$$

Since $\mathcal{L}$ is a monotonic function, taking the log of $\mathcal{L}$ does not change the maximizing $\vec{\theta}$ but it does greatly simplify calculations since it allows us to write:

$$ln\,\mathcal{L} = \sum_{i=1}^{n} ln(P(x_i \,|\, \vec{\theta}))$$

Parametric vs Non-Parametric
Linear vs Non-Linear

E. Classifiers

Fundamental statistical learning theory assumption: *labeled examples come from the same source as future examples.* More formally $\{(\vec{x}, y)\}_i^n$ is an i.i.d sample from a probability distribution over $\mathcal{X} \times \mathcal{Y}$

let $\hat{f}_D$ be a classifier trained on data set D.

$$\text{err}_D(\hat{f}) := \mid \{(\vec{x}, y) \in D \,|\, \hat{f}(\vec{x}) \neq y\} \mid$$

further, define the *true* error rate of classifier $f : \mathcal{X} \to \mathcal{Y}$

$$\text{err}_{\mathcal{P}}(f) := P(f(\vec{x}) \neq y)$$

note that the true error is with respect to a probability distribution $\mathcal{P}$ not a set D and $(\vec{x}, y) \sim \mathcal{P}$

Bayes Optimal Classifier: given data point $\vec{x}$ along with labels Y,

$$f^*(\vec{x}) = arg_y\, max\ P(Y = y \,|\, \vec{x}) = arg_y\, max\ P(\vec{x} \,|\, Y = y)P(Y = y)$$

$P(Y = y_i)$, denoted by π_i is called the class prior. $P(\vec{x} \,|\, Y = y)$, denoted by $P_i(x)$, is called the class conditional. In layman terms, the Bayes Optimal classifier always return the label with the highest probability of being correct.

Naive Bayes Classifier: makes the assumption that the individual features are independent of each other given the class label. Hence the term naive. We can write the naive Bayes Classifier as:

$$f^*(\vec{x}) = arg_y\, max\ \prod_{j=1}^{d} P(\vec{x}_j \,|\, Y = y)P(Y = y)$$

Naive Bayes Classifier is computationally very simple and it's quick to code. However, if the data set in question has correlated features (many do) it can give very bad estimates.

Generative Models: In the context of classification problems, a generative model is a statistical model $\mathcal{P}$ on $\mathcal{X} \times \{1, 2, 3, ..., k\}$ where each $P_\theta \in \mathcal{P}$ is

$$P_\theta(\vec{x}, y) = \pi_y \cdot P_y(\vec{x})$$

$$\vec{\theta} = (\pi_1, \pi_2, ..., \pi_k, P_1, P_2, ...P_k)$$

This theory is good an all, but how do we actually go about building a classifier? Say we have a data set $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ regarded as an i.i.d. sample.

1. Partition $\{x^{(i)}\}_{i=1}^n$ into $D_1, D_2, ...D_k$ where
$$D_y = \{x^{(i)} \ : \ y^{(i)} = y\}$$
2. Estimate π_i's (*i.e.* $\hat{\pi}_i = \frac{|D_i|}{n}$) and the P_i's for each class using maximum likelihood estimation. Obtaining $\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{P}}$
3. classifier $\hat{f}$ is Bayes classifier for distribution $P_{\hat{\theta}}$ corresponding to parameter estimates:
$$\hat{\boldsymbol{\theta}} = (\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{P}})$$

to classify,

$$\hat{f}(x) = arg_y \, max \ \pi_y \cdot P_y(x)$$

TODO finish this. important. Lecture 2 McInerney

F. Cross Validation

Having built a classifier, how can you know if it is any good? You can't just fit the model to your training data and hope it would accurately work for the real data it has never seen before. You need some kind of assurance that your model has got most of the patterns from the data correct, and its not picking up too much on the noise, or in other words its low on bias and variance.

Validation is the process of deciding whether numerical results quantifying hypothesized relationships between variables are acceptable as descriptions of the data. One validation technique is evaluation of residuals: it measures the number of misclassified points on the training set (or distances between predicted and training points in the regression case). Thus, evaluation of residuals is just a measure of how well the classifier performs on the training data. How well does the classifier perform on unseen data points?

Cross Validation is a technique that gives an approximation of how well the classifier will perform on unseen data points.

- **K-Fold Cross Validation:** The data is divided into k subsets where $k-1$ subsets are used to train the model and the remaining subset is used for validation. This process is repeated k times, each time changing the validation set. The error estimation is averaged over all k trials. As a general rule of thumb, $K = 5$, but a good choice of K obviously depends on the data set.

- **Leave P Out Validation:** This approach leaves p data points out of the training set, resulting in a training set of size $n - p$. Every combination of p points is considered. Clearly, for any reasonably sized data set, even small values of p can be computationally infeasible. A popular case of this method is $p = 1$.

G. Finding the G-spot: Overfitting vs. Underfitting

TODO

H. Feature Scaling

TODO

II. The Methods: Supervised Learning

1. k Nearest Neighbors (k-NN)

Layman definition:

Non-parametric method used for classification *and* regression. Given labeled training data $\mathcal{X}, \mathcal{Y}$ and $\vec{x} \notin \mathcal{X}$, find the k nearest (measured by some norm, default is euclidean) $\vec{x}_j \in \mathcal{X}$ along with corresponding $y_j \in \mathcal{Y}$. Denote the collection of the labels of the k nearest neighbors to $\vec{x}$ as $Y_{\vec{x}}^k$. Then we have, for

classification: output $y \in Y_{\vec{x}}^k$ such that y appears the most times in $Y_{\vec{x}}^k$. If two y appear the same number of times, chose y arbitrarily.

regression: output the average of all $y \in Y_{\vec{x}}^k$

Issues and How to Deal with Them

1. **Choosing the optimal value of k:** A very small value of k leads to overfitting while a large value of k leads to underfitting. Let $\hat{f}^i$ be the i-NN classifier. We want $k = arg_i\, min(\text{err}_D(\hat{f}^i))$ where D is the test set. Finding the optimal k is usually done by graphing training error and test error as a function of k then chooing k where test error flattens out as k increases.

2. **Speeding up k-NN Search:** use k-D Trees (see next section).

Improving the Method

TODO.. is it possible?

Pros vs. Cons

Pros

- No preprocessing
- No assumptions for underlying distribution (non-parametric)
- easy to understand and implement

Cons

- predictions are very costly, requiring a full scan through the data. Each prediction runs in $O(dN)$ (d dimensions, N data set size). Note that methods exist to speed up search.
- All the data (or some large subset of it) must be kept in memory for every prediction. This can be prohibitive for large N.
- How do you compare inputs with non ordered values? Which is nearer to green, yellow or blue?

Keep in Mind

- k-NN classifier is often successful where each class has many possible prototypes, and the decision boundary is very irregular
- k-NN is likely to do poorly with data having only a few relevant features and lots of irrelevant features
- All features are equally important to k-NN classifier. This becomes an issue when features are poorly scaled, leading to the effective marginalization of certain features.

Detour Uno: K Dimensional Trees or K-D Trees

Motivation

With 1-dimensional data we can speed up k-NN by sorting the data set and running binary search for each prediction. This gives runtime of $O(log(N))$ for each prediction, pretty damn good. How can we generalize the binary tree data structure for n-dimensions? In comes k-D Trees.

Layman Definition:

A k-D Tree is a space-partitioning data structure for organizing points in a k-dimensional space. A leaf may have 1 or more data points. It's instructive to see a simple algorithm for building a k-D Tree. (note that there are many possible algorithms, one of which results in the popular ML method called Decision Tree).

The algorithm: Given a set of points $\mathcal{X} \subset \mathbb{R}^d$ and a desired leaf number n (if none is given $n = \text{N}$; this is the k-D tree used to speed up k-NN search). TODO how to deal with repeats??

1. arbitrarily choose a dimension j s.t. $1 \leq j \leq d$.
2. let m equal the median of $\{\vec{x}_j \mid \vec{x} \in \mathcal{X}\}$
3. partiton the data into sets L,R s.t.
   $\text{L} := \{\vec{x} \mid \vec{x}_j \leq m\}$
   $\text{R} := \{\vec{x} \mid \vec{x}_j > m\}$
4. Recursively apply steps 1-3 on sets L and R until tree has n leaves.

k-NN + k-D Trees

For 1-NN, we build a k-D Tree with N (number of data points) leaf nodes. To make a prediction given $\vec{x}$ we can greedily route $\vec{x}$ down the nodes of the tree, going left or right depending on if the node condition is satisfied, until reaching a leaf node. Return the y of the data point in that leaf node.

IMPORTANT NOTE: the above algorithm is an example of where a greedy algorithm fails. It does not necessarily yield the nearest neighbor, it yields some near neighbor. It is very possible for the nearest neighbor to be in the other subtree. The diagram below shows an example of this pitfall.

There are several ways to resolve this issue. One way is to introduce additional condition(s) that must be satisfied at each inner node. Another way is to find nearest neighbor in the current leaf, denote its distance by r, and draw a circle of radius r around the query point. Search for leaves whose domain intersect the circle, and search for NN in those leaves. If another point is a distance $r' < r$ from query point, then that point becomes new NN and a new circle of radius r' is drawn. The search is repeated until no nearer neighbor is found.

The k-D Tree method for k-NN $k \geq 2$ is a bit more involved and, as always, there are several different approaches. One of the simplest approaches is to build a tree with leaves with a minimum of k data points. To make predictions, route query point $\vec{x}$ down the tree until reaching a leaf. Pick the data point in the leaf furthest from the query point. Let r denote the distance. Then draw a circle of radius r and repeat the steps in 1-NN, the difference being the circle must always maintain k points inside. The limitation of having leaves of size at least k can be prohibitive for large k. For other, perhaps better, methods, google them.

TODO draw tree, show image of defeatest search

2. Decision Trees

Motivation

Given labeled training data $\mathcal{X}$, along with labels $\mathcal{Y} = \{1, 2, ..., k\}$ the simplest classifier $\hat{f}$ we can build is: for each label y_j

$$P(Y = y_j \mid \vec{x}) = P(Y = y_j) := \frac{\sum \mathbb{1}\{y^{(i)} = y_j\}}{N}$$

Although the method is simple, easy to compute, requires no memory, and is easy to understand, it's a piece of shit. $\hat{f}$ leaves a lot of valuable information on the table each time it makes a prediction. The relation $P(Y = y_i \mid \vec{x}) = P(Y = y_i)$ shows this clearly. This says the label is independent of the query vector. But this feels wrong. The feature values of $\vec{x}$ must tell us *something*; they should make y_i more, or less, likely. We should expect $P(Y = y_i \mid \vec{x}) \neq P(Y = y_i)$ to hold in most cases (notice that changing the prior due to some new information to get a posterior is a Bayesian approach). So, how do we extract information from $\vec{x}$? We make the same assumption as in k-NN: data points near each other have similar y values. This gives rise to the following idea: What if we partition $\mathcal{X}$ into m regions where most points in a region have the same y value. By doing this, we have that for each region

$$\exists i \;\; 1 \leq i \leq d \quad \text{s.t.} \quad P(Y = y_i \mid \vec{x}) \approx 1$$

This is the idea behind Decision Trees. Stop and think, what is the above saying? It is saying our prediction will be right almost all of the time. If this gives you an odd feeling that we are 'cheating the system', you are not alone. In fact, you'd be correct. We seem to extract *too* much information from $\mathcal{X}$. We cannot believe everything $\mathcal{X}$ tells us. Data is usually naturally noisy, meaning that even the best possible classifier will still be wrong some of the time. This issue with 'over trusting' the data is in fact overfitting. We will see how to ameliorate this issue a bit later but it's important to see that Decision Trees have a strong tendency to overfit.

From k-NN to Decision Trees

If you read the above, you can skip this. But it's informative to continue our quixotic quest for the *most accurate, the most fastest, the most least-memory-userest, and the most versatilest* learning method on the face of the earth.

k-NN with k-D Trees is a definite improvement over regular k-NN. However, it still has considerable limitations. Data must be kept in memory for every prediction. Moreover, how do we deal with a data set where a vector contains both ordered and non ordered points (e.g. $\vec{x}_i \in \{\text{Blue, Green, Yellow}\}$ and $\vec{x}_j \in \mathbb{R}$)? To address the memory issue, lets take a moment to think about the assumptions we make when we choose k-NN as our classifier (or regressor). We assume

that $\vec{x}$'s that are near each other have similar y values. In k-NN, using k-D Trees to search for $\vec{x}$'s nearest neighbors, when we get to a leaf, we know that all points m in that leaf are near $\vec{x}$. They may or may not be the m nearest, but they are nearer than most other points. Does it really matter if we get the k nearest points instead of some m relatively very near points? What if, for every leaf, we take the most commonly occurring y value, and use it as representative of that leaf? In other words, for every $\vec{x}$ routed to that leaf, we predict the same value. Is this good enough to produce results similar to those of k-NN? Is the accuracy of the model negatively affected? If so, is there something we can do to compensate? The answers to the last three are no, yes, and yes.

So what can we do to compensate? Notice that the algorithm shown in the previous section for building a k-D Tree makes no attempt whatsoever at maximizing model accuracy. It solely partitions the data for efficient searching. What if we could partition the data in such a way that maximizes model accuracy? In come Decision Trees.

Layman Definition

A Decision Tree is a non-parametric method that can be used for classification. Geometrically, it is a k-D Tree where each splitting threshold t is chosen such that leaf uncertainty is minimized (we'll cover uncertainty measures shortly). Again, it is instructive to see a generic algorithm for building a Decision Tree. Note that it is the same algorithm as building a k-D Tree, the only difference is *how* you pick dimension j and the splitting threshold t, which in a generic k-D Tree, is the median.

The algorithm: Given a set of points $\mathcal{X} \subset \mathbb{R}^d$ and a stopping criterion (e.g. a desired leaf number n, or leaves must have at least m points)

1. choose a dimension $1 \leq j \leq d$ and splitting threshold t such that leaf uncertainty is minimized.
2. partiton the data into sets L,R s.t.
   L := $\{\vec{x} \mid \vec{x}_j \leq t\}$
   R := $\{\vec{x} \mid \vec{x}_j > t\}$
3. Recursively apply steps 1-2 on sets L and R until stopping criterion is true.

**note that splitting threshold t need not be numeric. For example, say $\vec{x}_j \in \{Blue, Yellow, Green\}$. $Yellow$ could be a splitting threshold for j. L becomes $\{\vec{x} \mid \vec{x}_j = t\}$ and R $\{\vec{x} \mid \vec{x}_j \neq t\}$

To make a prediction given $\vec{x}$ we route $\vec{x}$ down the nodes of the tree, going left or right depending on if the node condition is satisfied, until reaching a leaf node. Assume this leaf node has m data points. Denote the collection of the labels

of the m near neighbors to $\vec{x}$ as $Y_{\vec{x}}^m$. Output $y \in Y_{\vec{x}}^m$ such that y appears the most times in $Y_{\vec{x}}^m$ (plurality of y). If more than one y appear the same number of times, chose y arbitrarily.

How do we compute uncertainty doe? Let $p_{y_i} := \frac{\sum \mathbb{1}\{y^{(i)}=y_i\}}{n}$. The three most popular uncertainty measures are:

Entropy: $\sum_{y \in \mathcal{Y}} p_y \log \frac{1}{p_y}$

Gini: $1 - \sum_{y \in \mathcal{Y}} p_y^2$

Classification error: $1 - \sum_{y \in \mathcal{Y}} p_y^2$

Each measure results in a different tree, so it's important to chose the measure that makes sense for your data. You could also just try them all and chose the one that results in the lowest classification error.

Notes:

After building the tree, we can discard the training data. To make predictions we only need the condition at each inner tree node and the predicted y value at each leaf node. Decision Trees solve the memory issue with k-NN.

You may be wondering, how exactly the optimal j, t values are determined at each node. Step 1 is in fact the most important and time consuming step in the algorithm. Note that for each j, there are infinite possible t splitting values. To address this issue, we can implement the following algorithm for picking good values for j, t. Given some subset size N of training data:

for each j, sort $\vec{x}_j^{(i)}$ for all $1 \leq i \leq N$. Set t equal to the midpoint between ith and ith $+1$ element in sorted the sorted $\vec{x}_j^{(i)}$ for $i \in [1, N-1]$. Check uncertainty with these values of j, t. Return j, t with minimum uncertainty.

In python pseudocode:

```
min_list = []
for j in range(0, d):
    for i in range(0, N):
        list.append(X[i][j])
    list = list.remove_duplicates().sort()

    min_uncert = 1
    for i in range(0, len(list)-1):
        t = (list[i] + list[i+1]) / 2.0
        tmp = calculate_uncert(j,t)
        if tmp <= min_uncert:
```

```
                min = (j, t, tmp)
            min_list.append(min)
        return min(min_list, key = lambda g:g[2])[:-1]
```

**this is a shit algorithm. It's just here to give you an idea of how it could work.

Another important aspect to note is that, when we interpret the splits as hyperplanes, our splitting algorithm limits the hyperplanes to be perpendicular to the splitting dimension's axis. What if we removed this limitation? That is, instead we denoted our splits by $\vec{\theta} \cdot \vec{x} \leq t$ (we chose $\vec{\theta},\ t$ to minimize uncertainty, as before) instead of $\vec{x}_j \leq t$. These more complex splits are called surrogate splits. At first, it seems the increased generality of surrogate splits would lead to a much better model. It turns out this is not the case. The simple splitting method can achieve very similar results to surrogate splits, the only catch is that it requires a bigger tree. This makes sense if you think about it. Since we measure decision tree model complexity by tree size, then surrogate splits are theoretically superior. However, computational costs can be prohibitively expensive. Hence the simple splitting method is the standard for building the tree. There are cases where looking into surrogate splits can be beneficial, so don't be too quick to rule them out.

Issues and How to Deal with Them

1. **Tree size** A very large tree (i.e. many leaf nodes) leads to overfitting, while a small tree leads to underfitting. So when do you stop building the tree? A popular method is to build a very large tree, and then 'prune it'. That is, remove a leaf node and test prediction accuracy against a hold-out set. If model accuracy is improved, the leaf node is truly removed. This is continued until removing any leaf decreases accuracy.

Pros vs. Cons

Pros

- Easy to interpret results and present resulting tree to a layman
- Do not have to keep training data around for predictions
- Very fast predictions
- Can deal with data containing ordered and non-ordered values.
- Ignores features that do not contribute to label, since it won't split at these features

Cons

- It is very easy to overfit the data, even after pruning.
- Needs a lot of data to perform well.

Keep in Mind

Overfitting is a serious issue that must be addressed when using a DT in any serious work. This may require tuning the pruning process. Also remember there are methods (most popular are boosting and random forests) that can considerably improve tree performance.

3. Regression Trees

This is a very, very brief section since Regression Trees are literally just Decision Trees with a different uncertainty measure.

Motivation

What if we need a model that is a regression (i.e. it must predict continuous and real values)? For example, we must predict house values given room numbers and square feet. In this simple example we can use a regular linear regression. But we usually are trying to solve much more complex problems where dimensions are very high and relationships are highly non linear. We could then resort to a non-linear regression. However, notice that we have a single regression over the whole data space. Could we split the data space in 2 and fit a better regression on each subset of data? Most likely it's a yes. What if we split each subset further, and fit regressions on those? This is the idea behind the regression tree.

Definition

I will skip the definition since it's the same as Decision Trees. I will just briefly address the different uncertainty measure, which, in the simples case, is the variance, as shown below.

$$\frac{1}{n}\sum_{i=1}^{n}(\vec{x}_j^{(i)} - \mu_j)^2$$

where n is the data points in that leaf. To make a prediction, we route $\vec{x}$ down to a leaf and return the sample mean of Y in that leaf. Note that the resulting model is piecewise constant. We are not truly fitting a line at each node. Although this piecewise constant model is easy to understand and simple to compute, the prediction accuracy of these models often lags behind smoother models. It is straightforward to see how we can a fit line at each node, but the costs are usually prohibitively high. There are several algorithms (M5, GUIDE) that do some attempt at smoothing to get better results.

Regression tress and decision trees are collectively referred to as CART (Classification and Regression Trees).

4. Bayes Classifier

Motivation

Consider the following imaginary scenario. We are faced with the task of predicting sex based only on height measurements. Since we are dealing with one dimensional data, we decide to plot the points, labelling male values with an x and female values with an o. When we look at the points on the line, we see that the o's cluster around one value and the x's cluster around another. We then decide to build a histogram of the male and female height values. We see that both histograms form a rough bell curve. If we have some basic knowledge of probability theory, we'd say, "hey, it seems like male histogram delineate a gaussian curve and women's delineate another. If we modeled height as a random variable drawn from a gaussian distribution, we could make some pretty accurate predictions." This is the rough idea behind probabilistic models.

The search continues: from geometry to probability

k-NN and Decision Trees are good and all, but they aren't perfect. Being the perfectionists and idealists that we are, we have this feeling in our gut that the perfect method is still out there, restlessly waiting to be found, anxious to share with us the world's knowledge, eager to bloom and display the true beauty of data. Let us take a step back and again ask the fundamental learning question: given $\mathcal{X}$, $\mathcal{Y}$ and $\vec{x} \notin \mathcal{X}$ how can we predict the correct $y \in \mathcal{Y}$ associated with $\vec{x}$? In k-NN and Decision Trees, the only assumption we made is that points near each other have similar y values. What if we take on a more probabilistic view of the world? Namely, what if we assume there exists some probability distribution $\mathcal{P}$ from which all points $(\vec{x}, y)$ are drawn? This assumption would allow us to use the full power of probabilistic tools to tackle our learning question. Armed with these, maybe we can finally build the ultimate optimal classifier.

Layman Definition

(You should go back and review the sections on probability and statistics if any of this is not immediately obvious)

Remember the Bayes Optimal Classifier?

$$f^*(\vec{x}) = arg_y max\ P(Y = y \,|\, \vec{x}) = arg_y max\ P(\vec{x} \,|\, Y = y)P(Y = y)$$

Notice $P(\vec{x} \,|\, Y = y)P(Y = y) = P(X = \vec{x}\,, Y = y)$. In other words, we are maximizing the joint distribution. In summary, it's a classifier that returns the label with the highest probability conditional on $\vec{x}$. The Bayes Optimal Classifier we'll use is an estimator of the *true* Bayes Optimal Classifier. To see this, consider the following case. We have a data set $D = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ of i.i.d. samples from $P_{\boldsymbol{\theta}} \in \mathcal{P}$ where $\mathcal{P}$ is a generative statistical model with parameter space Θ. However, we don't know $P_{\boldsymbol{\theta}}$. How do we go about estimating it? Estimating $\boldsymbol{\theta}$

is what we call model fitting. There are several methods to solve this problem. A popular and generally very robust method is maximum likelihood estimation (MLE). Notice that since $\mathcal{P}$ is a generative model, we need to estimate both the class priors $P(Y = y)$ and the class conditionals $P(X = \mathbf{x} \,|\, Y = y)$. Once we have $P_{\hat{\boldsymbol{\theta}}}$, we build a Bayes Optimal Classifier $\hat{f}$ for it.

The key thing to realize here is that f^* is the Bayes Optimal Classifier for $P_{\boldsymbol{\theta}}$ while $\hat{f}$ is the Bayes Optimal Classifier for $P_{\hat{\boldsymbol{\theta}}}$. Also notice that using MLE requires us to set a length (i.e. number of parameters) for $\boldsymbol{\theta}$ and, perhaps more importantly, requires us to chose an underlying distribution (gaussian, poisson, etc). We could chose $P_{\hat{\boldsymbol{\theta}}}$ to have 10 parameters (five labels) and be normally distributed, but what if the data's true underlying distribution has 14 (seven labels) parameters$P_{\boldsymbol{\theta}}$ is strongly bimodal? Our classifier would not perform very well. This is the main drawback from using probabilistic models, they often require us to make assumptions that are above our pay grade. There are other estimation models that do not as assumption-needy, such as non-parametric and graphical models. These are usually more involved and also have their drawbacks (non-parametric often performs poor in high dimensions). After studying the data however, we can often glean valuable information that allows us to make some pretty good assumptions about the underlying probability distribution. In these cases, Baysian Classifiers really shine.

How to use:

The definition offered above wasn't formatted in 'definition' format. So I will show an example of how to actually build a Bayesian Classifier. Say we have a data set $D = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ regarded as an i.i.d. sample.

1. (We don't actually do this in practice. We just do step 2 directly. But it's useful to think of it this way.) Partition $\{\vec{x}^{(i)}\}_{i=1}^{n}$ into $D_1, D_2, ...D_k$ where
$$D_y = \{\vec{x}^{(i)} \,:\, y^{(i)} = y\}$$

2. Estimate $P_{\boldsymbol{\theta}}$. That is, estimate the class priors π_i's (*i.e.* $\hat{\pi}_i = \frac{|D_i|}{n}$) and the class conditionals P_i's (i.e. $P(\vec{x} \,|\, Y)$) for each class using maximum likelihood estimation, obtaining $\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{P}}$. We have several options for modeling the class conditionals. We could use non-parametric models, graphical models, or simple distribution models (e.g. multivariate gaussian). The first two are above my clearance so for this example, let's assume multivariate gaussian. Using MLE to fit $\boldsymbol{\theta}$, we arrive at
$$\hat{\mu}_i = \frac{1}{|D_i|} \sum_{i=1}^{|D_i|} \vec{x}^{(i)} \qquad \hat{\Sigma}_i = \frac{1}{|D_i|} \sum_{i=1}^{|D_i|} (\vec{x}^{(i)} - \hat{\mu}_i)(\vec{x}^{(i)} - \hat{\mu}_i)^T$$

3. $\hat{f}(x) = arg_y max\ \pi_y \cdot P_y(x)$ is Bayes classifier for distribution $P_{\hat{\theta}}$ corresponding to parameter estimates: $\hat{\boldsymbol{\theta}} = (\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{P}})$

Naive Bayes Classifier: The Slutty Sister

Sometimes, due to various circumstances (complexity of data, need for speed, and whatnot), we are cornered into making some further assumptions (jk, sometimes we can actually make the naive assumption). The Naive Bayes Classifier assumes that individual features are independent of each other given the class label. Hence the term naive.

$$f(\vec{x}) = arg_y max\ \prod_{j=1}^{d} P(\vec{x}_j \,|\, Y = y)P(Y = y)$$

Naive Bayes Classifier is computationally very simple and it's quick to code. However, if the data set in question has correlated features (many do) it can give very bad estimates. Extending the "How to use" section to Naive Bayes is straightforward so I won't rewrite it.

Issues and How to Deal with Them

Choosing a model P_θ, choosing an estimation method.

Pros vs. Cons

Pros

- straightforward recipe for building a classifier
- Allows us to leverage domain knowledge of class conditional distributions
- Can be very efficient when number of labels is large
- If our estimated distribution is close the real distribution, than Bayes Classifier can perform well with a small amount of training data.

Cons

- Classifier assumes data comes from estimated distribution $P_{\hat{\theta}}$, which is generally not true.
- A more philosophical critique is that there is wasted effort in defining the joint probabilities. Since we are solving a classification problem, all that matters is the boundary

Keep in Mind

Although the distribution assumptions we must make for Bayes Classifier are a great weakness, they can at times be a great strength. If our assumptions are approximately correct, Bayes Classifier needs much less data than k-NN and CART to achieve low error rates. When training data is scarce and we have some prior knowledge of it's structure, then Bayes Classifer could very well be the best choice. Also

Detour Dos: Discriminative vs. Generative

Now that we have learned three popular and powerful methods, k-NN, CART, and Bayes Classifier, we should start to have somewhat of a feel for what a learning method truly is. We often hear the term "extract information from the data", but what does it actually mean? In classification problems, it means we can characterize each of the k classes by a set $\mathcal{C}_k \subset \mathbb{R}^d$ where d is the number of features (the sets $\mathcal{C}_k$ are usually non-convex and disjoint). Thus, given a new observation $\vec{x}$ (i.e. a point in $\mathbb{R}^d$), we assign it to class k such that $\vec{x} \in \mathcal{C}_k$. Although we have not thought about it in this way, all the methods we have seen partition $\mathbb{R}^d$ into k subsets, one for each label. They all do so in a different manner. If we compare the methods seen so far, to which method would you say k-NN is most similar to, Decision Trees (DT) or Bayes Classifier (BC)? k-NN is clearly closer to DT than BC. In k-NN and DT we only make simple geometric assumptions; we directly build the decision boundary. We don't worry about probability distributions and whatnot. BC, on the other hand, requires us to make serious assumptions about the underlying distribution of the data, then we fit a model, then we make predictions. k-NN and Decision Trees are examples of *discriminative classifiers* while Bayes Classifier is an example of a *generative classifier*. Generally, discriminative classifiers are preferred over generative ones. Since all that really matters in classification is the decision boundary (i.e. the conditional probability), generative classifiers do 'extraneous work', when estimating the joint distribution. This actually increases our chances of model mismatch. Discriminative classifiers are straight to the point. However, the ability of generative classifiers to leverage prior knowledge (when it is correct) is useful when we have relatively few observations, often outperforming discriminative classifiers in this scarce data context. Generative models also handle missing data and unlabeled training data (semi-supervised learning) more elegantly by using probability theory.

5. The Perceptron:

Motivation

As we discussed in Detour Dos, all we really care about (in classification) is the decision boundary. The non-parametric methods seen so far draw some

pretty crazy decision boundaries. While the Bayesian Classifier draws more tame boundaries, they are still non-linear. Consider the case where we are trying to build a binary classifier. What if we assume that our data set is linearly separable? That is, there exists some hyperplane that cleanly partitions the data set so that all points of class 1 are on or above the hyperplane and all points of class 2 are in below it? (You may be thinking this is an overly optimistic, even ridiculous, assumption. It turns out we can make a non-linearly separable data set linearly separable). Note that if there exists a hyperplane that separates the data, there exists infinitely many such hyperplanes. The key idea here is that we can use this hyperplane as a classifier. How do we go about finding these hyperplanes? If our data set contains d continuous features, then we have ∞^d possible values for the vector defining the hyperplane. Brute force search is not an option. We resort then to algorithms to help us find a hyperplane, if there exist any. One such algorithm is the perceptron.

From Hyperplane Geometry to Classification

I could just list out the equations and the perceptron algorithm, but that would leave you feeling a void in you chest. I will try to characterize hyperplanes in an intuitive way so that the perceptron naturally follows. Consider a plane in three dimensional space $z = ax + by + d$. In vector notation

$$z = < a, b > \cdot < x, y > + d \;=\; \vec{w} \cdot \vec{x} + d$$

The intersection of this and the xy-plane is a line given by

$$0 = ax + by + d \implies y = -\frac{a}{b}x - \frac{d}{b}$$

We will refer to the line resulting from the intersection as the 'decision line', we'll see why shortly. Notice $z = 0 \iff$ the point(x, y) is on the decision line. Notice also that the sign of z tells us on what side of the decision line is the point (x, y). If $z > 0$, then we know the point (x, y) is above the decision line; $z < 0$ tells us the point (x,y) is below it. An interesting and very surprising fact (for me at least) is that the signed distance from a point on the xy-plane to the decision line is given by

$$\frac{ax + by + d}{\sqrt{a^2 + b^2}} \;=\; \frac{\vec{w} \cdot \vec{x} + d}{||\vec{w}||} \;=\; \frac{z}{||\vec{w}||}$$

It follows that if we interpret the decision line as partioning the xy-plane into two subsets, then z tells us on which subset is a point (x, y). Converting z into a binary classifier then is straightforward. Let's define a non-linear transformation f, which we'll call the activation function as follows,

$$f(\vec{w} \cdot \vec{x} + d) = f(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ -1, & \text{if } z < 0 \end{cases}$$

Notice the two classes are specified by -1, 1 not 0, 1. This modification allows us to use the perceptron algorithm.

So, how do we go from data set to classifier? Let's go through the steps. Say we have a labeled (as -1,1) data set D with 2 features. Further assume the data set is linearly separable (C_1, C_2 are convex sets whose boundary is linear). Therefore, there exists a hyperplane $\vec{w}\cdot\vec{x}+w_0$ that defines this boundary. What does this mean mathematically? It means that for every point

$$\vec{x} \in C_1 \quad \vec{w}\cdot\vec{x} + w_0 \geq 0$$

$$\vec{x} \in C_2 \quad \vec{w}\cdot\vec{x} + w_0 < 0$$

Using the activation function f, we can rewrite the above with our desired labels (-1,1)

$$\vec{x} \in C_1 \quad f(\vec{w}\cdot\vec{x} + w_0) = 1$$

$$\vec{x} \in C_2 \quad f(\vec{w}\cdot\vec{x} + w_0) = -1$$

Thus our classifier is defined by a hyperplane $\vec{w}\cdot\vec{x} + w_0$ and an activation function f. Clearly, our prediction is correct if $f(\vec{w}\cdot\vec{x}^{(i)} + w_0) = y^{(i)}$. Equivalently, a correct prediction yields $f(\vec{w}\cdot\vec{x}^{(i)} + w_0)y^{(i)} = 1$. This will be useful for the perceptron algorithm. Note that if the data set is linearly separable, we would expect $f(\vec{w}\cdot\vec{x}^{(i)} + w_0)y^{(i)} = 1 \quad \forall i$.

For more terse notation, let's define $\vec{w}' := (w_0, \vec{w})$ and $\vec{x}' := (1, \vec{x})$. We rewrite

$$\vec{w}\cdot\vec{x} + w_0 = \vec{w}'\cdot\vec{x}'$$

How should we evaluate the 'correctness' of an arbitrary hyperplane in classifying D? An obvious metric would be to count the number of misclassified points. This approach, however, weighs all misclassified points equally. What if we decide that misclassified points that are far from the decision boundary are in a way more incorrect than misclassified points near the decision boundary? Mathematically, the classification error for a given $\vec{w}'$ would be given by,

$$E(\vec{w}') = -\sum_{j\in\mathcal{M}} (\vec{w}\cdot\vec{x}^{(j)} + w_0)y^{(j)}$$

Where $\mathcal{M}$ is the set of misclassified points. Note that $E > 0$ if $\mathcal{M} \neq \emptyset$. Therefore we want to minimize E. Also note that $\vec{w}\cdot\vec{x}^{(j)} + w_0$ does not give the distance of point $\vec{x}$ to the decision line, but it is directly proportional to the distance and therefore we are are penalizing the classifier more strongly for points that are further away. To minimize E, we use stochastic gradient descent,

$$\vec{w}^{(\tau+1)} = \vec{w}^{(\tau)} - \eta\nabla E(\vec{w}') \;=\; \vec{w}^{(\tau)} + \eta\vec{x}^{(j)}y^{(j)}$$

Ladies and gentlemen, I present to you the perceptron algorithm. The above analysis readily generalizes to d dimensions.

Layman Definition:

The perceptron is an algorithm for finding a hyperplane that separates the data set iff the data set is linearly separable. Notice that implicit in this definition is that where dealing with only two labels. A hyperplane takes the form $\vec{w} \cdot \vec{x} + w_0$. For notation purposes, we will redefine $\vec{w} :=< w_0, \vec{w} >$ and $\vec{x} :=< 1, \vec{x} >$ allowing us to write the hyperplane equation more tersely: $\vec{w} \cdot \vec{x}$.

The algorithm: Given labeled training data $D = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ starting at $t = 0$:

1. initialize $\vec{w} = \vec{1}$. (any value works)
2. for τ in range(0, maxIterations) loop through steps 3-4 (each iteration through the whole data set is called an epoch)
3. for all $(x, y) \in D$ do
4. if $f(\vec{w} \cdot \vec{x}_i) y_i < 0$ then $\vec{w} = \vec{w} + \vec{x}^{(j)} y^{(j)}$

algorithm ends when $f(\vec{w} \cdot \vec{x}_i) y_i > 0 \quad \forall (x, y) \in D$. What happened to η? Don't really understand why and I am pretty tired rn, but η doesn't affect the hyperplane in a relevant way, so we just set it to 1.

The algorithm is actually quite different than either the decisiom tree algorithm or the k-NN algorithm. First, since we are using stochastic gradient descent, it is *online*. This means that instead of considering the entire data set for each update, it only looks at one example. Second, it is *error driven*. This means that, so long as it is doing well, it doesn't bother updating its parameters. One aspect of the perceptron algorithm that is left underspecified is line 3, which says: loop over all the training examples. The natural implementation of this would be to loop over them in a constant order. The is actually a bad idea. You need to avoid is presenting the examples in some fixed order. We can do this by permuting the order of examples at each epoch and iterating over this permuted set.

Notes

The perceptron algorithm is guaranteed to find a hyperplane that linearly separates the data, if one exists. However, remember that if one exists, then infinitely many exist. Does the perceptron consistently return the same $\vec{w}$? It does not. The resulting $\vec{w}$ varies depending on the initialization of $\vec{w}$ and the order in which the points are presented. This brings us to the question, are some hyperplanes better at partitioning the data than others? Yes. Support Vector Machines, the method we will see next, finds the optimal separating hyperplane.

What is the runtime of the perceptron algorithm? Since this algorithm is error driven, we expect the runtime to be in terms of m, or number of mistakes, not

the number of data points. Another aspect to note is that we would expect the perceptron to converge more quickly for easy learning problems than for hard learning problems. This certainly fits intuition. The question is how to define easy and hard. One way to make this definition is through the notion of margin. If I give you a data set and hyperplane that separates it, then the margin is the distance between the hyperplane and the nearest point. Intuitively, problems with large margins should be easy (there's lots of wiggle room to find a separating hyperplane); and problems with small margins should be hard (you really have to get a very specific well tuned weight vector).

The Perceptron Convergence Theorem: Suppose the perceptron algorithm is run on a linearly separable data set D with margin $\gamma > 0$. Define $R := max_{\vec{x} \in D} ||\vec{x}||$. Then the algorithm will make at most $(\frac{R}{\gamma})^2$ mistakes.

Issues and How to Deal with Them:

1. **Voted Perceptron:** If a data set is approximately linearly separable, then we would expect the resulting hyperplane from the perceptron to be approximately correct. This is somewhat true. The issue is that the resulting $\vec{w}$ varies considerably depending on the order the data is iterated over. This means that the hyperplane after the final update t^* could be worse than hyperplane at update $t^* - 1$ or even $t^* - 10$. Put another way, the regular perceptron algorithm gives more weight to points seen later than those seen earlier. If the hyperplane at update t correctly classifies a big chunk of the data (i.e. goes a long time without updating), then this should indicate that the error minimizing hyperplane is closer to it. Consequently, we should give it more weight than the hyperplane resulting from update $t+k$ if that hyperplane only saw a few points before updating. This is the idea behind the voted perceptron. We weigh every hyperplane by the number of points it correctly classified. Let k denote hyperplane after the kth update and $c^{(k)}$ be the number of points the kth hyperplane correctly classified before updating.

$$\hat{y} = f(\sum_{k=1}^{K} c^{(k)} f(\vec{w}^{(k)} \cdot \vec{x} + w_0^{(k)}))$$

2. **Linear Separability:** Interesting data sets are never linearly separable or likely even approximately linearly separable. Does that mean the perceptron is completely useless? Not at all. We can still fit non linear boundaries using the perceptron concepts. One method is to kernlize the feature space, effectively "linearly separating" the data (we explore kernel methods in the next detour). Another, perhaps more elegant way, is to combine multiple perceptrons into a single framework called a neural network.

Keep in Mind:

There is no Pros vs. Cons section for the perceptron since it is not very common to use the perceptron on it's own. Its true applications lie in neural networks, which we'll cover later. There, we will reintroduce the perceptron in a very different light.

Detour Tres: Kernel Space and the Kernel Trick

The main draw back of linear classifiers is that a data set is rarely linearly separable. This non linear-separability arises due to two reasons. The data could just be noisy or the decision boundary is strongly non linear (or both). Consider a case of the latter. Data points near the origin belong to class $\mathcal{C}_1$ and points away from the origin belong to class $\mathcal{C}_2$. A linear decision boundary is hopeless. However, a spherical decision boundary could perform very well. Now assume that a hypersphere cleanly partitions the data set into the two regions. If we mapped $x_i \to x_i^2$ and redefined the x_i axis to x_i^2 for every i, then we would see that, in this new space, the data set is linearly separable. Thus if a data set is separable in some non-linear way, by applying a non-linear transformation to the input vector $\vec{x}$ to get an augmented feature vector $\vec{x}'$, we make the data linearly separable. We call the non-linear input transformation a kernel function, denoted by $\phi(\vec{x})$. More formally, $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$. We call $\mathbb{R}^{d'}$ kernel space.

This is a very powerful result that allows us to apply linear classifiers to a much wider array of problems. However, there is a drawback. Consider the kernel

$$\phi : \mathbb{R}^d \to \mathbb{R}^{\frac{d(d-1)}{2}}$$

That maps a vector $\langle x_1, ..., x_n \rangle$ to $\langle x_1^2, x_1x_2, ..., x_2^2, x_2x_3, ..., x_n^2 \rangle$. You may recognize $\frac{d(d-1)}{2}$ is $\binom{d}{2}$. This kernel allows us to greatly increase the expressive power of our model and may allow us to apply linear methods to a previously non linear problem. Note however, that computational complexity increased quadratically with respect to d for every point. This is not good if our input vector is high dimensional (In the hw image classification problem, $d = 784$. This kernel converts a 784 dimensional vector to a 307,328 dimensional vector. Not good). It turns out however, that there is a trick that allows us to expand the input vector without having to ever deal with the kernalized vector in computation. We thus gain a lot of expressive power without *any* additional cost.

Consider a kernel that maps to the interactions between features up to the quadratic power.

To get a feel for kenel methods, let's kernlize the perceptron algorithm.

As you can probably already tell, coming up with a good kernel function is an art. Good feature transformations are what separate a good data scientist

from a mediocre one. This is one of the tougher parts of building a classifier. We'll see later a way to overcome this using neural networks.

Support Vector Machines:

TODO

Neural Networks

Motivation:

Remember how hard it is to come up with a good kernel function? Neural networks make life much easier by

Unsupervised Learning:

All of the methods for classification seen so far assume the data set is labeled. We assume the data set can be partitioned into sets that characterize each label. Sometimes, we are faced with a different problem. We have a data set D of unlabled data and, moreover, sometimes we don't even know how many classes there are. Extracting information from a data set means something different in this context than in classification problems. In unsupervised learning we are trying to find *possible* partitions of the data that

Time Series Models

Motivation

What if the our data set is not i.i.d.
Notes 11/13

Graphical Models

definition A graph consists of nodes $s \in \mathcal{V}$ and edges $e \in \epsilon$