An Overview of Machine Learning and Pattern Recognition

Alexander Sverdlov*

June 26, 2015

Abstract

Machine learning is a branch of computer science that is concerned with the automation of learning. It is an integral part of artificial intelligence, and relates to fields as diverse as data mining, statistics, philosophy, information theory, biology, and computational complexity. The goal of machine learning is to build computer programs that improve with experience; ultimately that may automate science and discovery, and bring us computer programs that get progressively better at programming themselves. While machine learning subject spans far and wide, in this review we look at a segment related to pattern recognition, specifically reviewing decision trees, hyperplane methods, neural networks and clustering.

*alex@theparticle.com

Contents

1 Introduction

The US financial industry collectively generates tens of billions of records *every day*. Everything from order entry, order routes, order executions, quotes, trades, etc., everything is logged. Regulatory agencies have access to large percentage of this data—with the sole purpose of regulating markets and brokerage firms. SEC Rule 613 mandates the creation of a Consolidated Audit Trail—this level of data will be a lot more available once that rule is implemented in the next few years.

Regulation mostly consists of looking for rule violations—mostly those specified in Securities Exchange Act of 1934. For example, there is a "Prohibition Against Manipulation of Security Prices"—which in part is defined as "creating a false or misleading appearance of active trading in any security." This definition goes through a few rounds of rule interpretation (the rules are often very vague), and eventually a program is created to detect such activity in the sea of data.

How exactly would such a program work? We could arbitrarily define 'price manipulation' as "a market participant trades at least 5 buy *and* 5 sell orders within 3 minutes at progressively increasing or decreasing prices." Where did we get 5 orders? Where did we get 3 minutes? Those are arbitrary cut off points.

We write the program (no easy task, when dealing with billions of records), and find millions of instances. We tighten the thresholds—make it 10 orders, and 1 minute, etc. We continue to do this until we get a few dozen instances—something that an analyst can *manually* examine by hand to confirm that this really is 'price manipulation' as *defined* (which we just did).

The analyst, upon manually examining the kick-outs labels a dozen as 'valid' and the rest are dismissed as being 'normal market activity'. Now we go back to the drawing board, and tighten the requirements even more, so only those dozen would show up in the next run

of the program (and even then, we still get false positives!).

What about false negatives? Let's pretend they don't exist.

Clearly there has to be a better way. Machine Learning may be the answer, but there are *major* challenges.

- Size of data—Bagging and other ensemble methods are a requirement. Algorithms that cannot be parallelized will not work.
- Sampling does not work—we could sample a million records, and not find a single instance (not to mention that someone would need to manually examine them).
- Each instance is different—and often requires analysts to manually examine and piece things together—often taking hours of time.
- Most instances are not isolated to one record—the situations often involve many (sometimes thousands) of records spanning a long time (sometimes across days—meaning trillions of records).
- Records have numeric, categorical, bit-fields, and just plain text fields. There are also data problems almost every day (when there is a one in a billion chance of something happening, it is a sure to happen a dozen times a day).
- Whatever mechanism is used to detect instances has to be explainable at the business level (it cannot be a black box), and be defendable in court (be sound from the regulation perspective and in spirit of the rule).
- The solution has to be efficient—programs that take days to run are a huge resource drain.

And so it is with these things in mind that we approach Machine Learning. In this paper we review the core algorithms and concepts that underpin the whole field, starting with the ultimate question of...

1.1 What is Machine Learning?

Machine Learning is a "Field of study that gives computers the ability to learn without being explicitly programmed". Quote due to Arthur Samuel [Sam59], who created a checkers program that, by playing itself, learned to play checkers far better than Arthur Samuel himself.

1.2 What is Learning?

There are essentially three ways of acquiring new knowledge: *memorization*, *deduction*, and *induction*. With memorization, the learner simply memorizes (stores in a database or file) all facts and observations. These facts can then be recalled, or matched to input. A rote memorizer cannot cope with inputs that were not previously observed, nor can any predictions be drawn from the data.

With deduction, the learner gets predictive power; starting with knowledge that the learner knows to be *true* (facts, observations, or previously proved items), the learner attempts to derive other truths by applying laws of logic and math. No unproven assumptions allowed.

Induction is what we are interested in for machine learning; it is deduction with assumptions. The inductive learner makes observations (usually input/output pairs) and assumes (guesses) that they were generated by some *process*, e.g., a normally distributed random variable. Then working backwards, the learner uses these observations to construct a model of this process that mimics the observations in some way, e.g., the learner may construct a

stochastic process that has the same *mean* and *variance* as the observations. This model is the 'learned knowledge', and can be used in place of the process to make predictions.

The above applies equally to humans and machines. Humans do have an advantage of common sense—and perhaps some genetic memory (our eyes learn to recognize shapes without us consciously trying to learn how to). Machines have the capacity advantage—it is trivial to chug through billions of records.

1.3 What is a Pattern?

The concept of a "dog" may be generalized from a list of attributes—animal, pet, domesticated, furry, small, affectionate towards owner, protective of owner, and requires daily walks. Changing some of those attributes may more appropriately generalize to a "cat" concept.

Given a list of attributes, we can determine if they are a better match for a "dog" concept or a "cat" concept (or perhaps not applicable to either). Those attributes define points in concept space, and "dog" and "cat" concepts are regions of that concept space.

A pattern is some structure in concept space. As with the above concept example, it is often a region, or subspace. It could also be as simple as a point, or as complicated as an arbitrary N-dimensional object. It could be defined in numeric terms or non-numeric attributes.

Our task is to learn that pattern. Since we do not actually know what it is, we need some way of sampling the concept space. Let us imagine we have a function $f(\boldsymbol{x})$, with $\boldsymbol{x} = (x_1, \ldots, x_n)$ where x_i is a number or string. This function $f(\boldsymbol{x})$ will return 0 for all $\boldsymbol{x}$ that are part of the pattern, and a non-zero value otherwise, perhaps indicating how far $\boldsymbol{x}$ is away from the pattern.

In other words, the pattern recognition task is to learn some mechanism to estimate $f(\boldsymbol{x})$. The $\boldsymbol{x}$s we use for learning may be generated by some process that is either sampling

or somehow moving through the concept space. To complicate matters $\boldsymbol{x}$s may be noisy, and some x_i values may be irrelevant, random, or wildly transformed. The pattern itself may be changing in time, perhaps chaotically. Think of how "driving a car" concept might look in N-dimensional concept space—it's a function that millions of people manage to learn, even though the input is audio-visual and ever changing.

Classification is related to pattern recognition—the $f(\boldsymbol{x})$ returns the name of the concept region the $\boldsymbol{x}$ is in.

1.4 Methods & Models

Concept space is big. Too big. When learning structures in concept space, we cannot rely on memorization—it is highly likely that every $\boldsymbol{x}$ we observe during training, and every $\boldsymbol{x}$ we attempt to recall during production will be different. This leaves us in fuzzy land of *induction* discussed above.

One practical way to model things in concept space is to define representative points or examples. If we are trying to recognize a "dog" concept, perhaps we memorize all the dog and non-dog examples we see during training. When presented with an unknown pattern in the future, we check the *distance* to all the previously memorized examples, and pick the label of the closest known example. To avoid noise, perhaps we can use the majority label of the k nearest examples. This is known as k-NN or k nearest neighbors algorithm, described by Cover & Hart and Fix. [FJ51, CH67].

The k-NN is incredibly easy to implement and use. However, it raises the question of: What is *distance* anyway? How distant is "dog" concept from the "cat" concept? Is it any more distant than say a "hamster" concept? Those are some tough questions and no easy answers—there are different measures of distance though (e.g. Euclidean distance, Manhattan distance, Cosine distance, Hamming distance, edit distance, etc.). None of these

are universally applicable, and some only make sense for numeric data or string data.

Efficient implementations of k-NN cannot compare distance to every remembered training sample—since runtime would grow linearly with the training set size. Nearly all implementations use some hierarchical structure, such as axis-aligned K-D trees described by Bentley [Ben75] or Ball Trees described by Omohundro. [Omo89] Such hierarchical structures define regions of concept space—using just a simple rule. To generalize, what form would such rules take?

What separates one concept from another is some sort of a decision surface (or multiple surfaces) in concept space. For numeric $\boldsymbol{x}$, that surface is often a hyperplane. In fact, K-D tree mentioned above recursively divide concept space by defining hyperplanes. Any $\boldsymbol{x}$ can be easily checked against a hyperplane to see if it is in *front* or the *back* of the plane. When $\boldsymbol{x}$ has string values, the decision rule incorporates equality of said strings in its operation. For example, samples that have $attrib = Yes$ defines a subspace which is different from the $attrib = No$ subspace. They do not have to be binary—a string attribute that has 30 different values may define 30 subspaces.

Estimating such decision surfaces to separate out subspaces of concept space is the theme of all pattern recognition methods. The differences are mostly in what kind of decision rules are used, how many of them are used, whether they are layered, and how good they are at separating concepts. There is also a lot of variation due to particular domain areas: how much data is there, how noisy is it, what are the relevant dimensions and distance measures, how is data prepared, cleaned, etc.

Some concepts are inherently statistical. The concept of a "fair coin" for example. No coin will ever land half heads and half tails on any single trial, and yet that is the expected number of heads and tails for a fair coin. For such patterns, it is often more useful to gather statistics and apply decision rules on those measures as opposed to on the individual samples. In the

end, it still boils down to some decision surface using some distance measure—either distance between means (for example, the Student's t-test[1]) or some measure between distributions (such as Kullback-Leibler divergence[2] or chi-squared test[3]).

1.5 Verification

To verify whether our pattern recognition algorithm actually works we need to test it on data it has never seen before. This is often done by training the model on 70% of the training data, and testing it on the remaining 30%. We are always interested in the accuracy rate, and often also interested in the false positive and false negative rates.

When data is scarce, a technique of k-fold validation may be used. Training data is partitioned into k buckets. Training is done k times on $k-1$ datasets and tested on the remaining kth dataset. The accuracy rates are averaged across all k runs. It is not unusual to set $k = 10$. In the extreme case (perhaps all we have is handful of samples) we can do n-fold validation, where we n is the number of training samples.

1.6 Notation

In this paper, vectors are lowercase bold, e.g. $\boldsymbol{x}$, matrices are upper case bold, e.g. $\boldsymbol{X}$, and scalars are not bold, e.g. x. A scalar such as x_i is the ith element of a vector $\boldsymbol{x}$, and $\boldsymbol{X}_i$ is the ith vector of a matrix $\boldsymbol{X}$.

[1]One sample t-test: $t = \frac{\bar{x}-\mu_0}{s/\sqrt{n}}$ where $\bar{x}$ is the sample mean we are testing, μ_0 is the population mean, s is sample standard deviation, and n is the sample size. Significance of t is then determined by where it falls in the Student's t-distribution.

[2]Kullback-Leibler divergence of Q from P is $D_{KL}(P||Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)}$. The $D_{KL}(P||Q)$ represents number of bits required to code samples from P using a code optimized for Q.

[3]Chi-squared test: $\chi^2 = \sum_{i=1}^{n} \frac{(O_i-E_i)^2}{E_i}$ where O_i is number of observations of type i, n is number of different categories, E_i is the expected frequency of type i observations. Significance of χ^2 is then determined by where it falls in the Chi-squared distribution for a given degree of freedom.

2 Decision Trees

We begin the pattern recognition review with decision trees. They are a graphical representation of decision making that is easily interpretable by humans. Despite their simplicity, they are quite robust. Their step-by-step explanatory capability is very valuable in getting people to trust the output. Often, another mechanism is trained for the sole purpose of labeling enough examples for a 'final model' decision tree to be constructed—that way we get the flexibility of another model, and the readability of a decision tree.

To label an unknown example $\boldsymbol{x}$, we start from the root of the tree, and at each node apply a test that tests a particular attribute x_i. The result of this test tells us which child branch to explore for subsequent tests. This continues recursively until a leaf node is reached—the label on the leaf node is the output. When the target is numeric, we may call this setup a Regression Tree.

In this scenario, the training data are often represented as a set of tuples $\boldsymbol{X}$ with each tuple being $(x_1, \ldots, x_n, x_t)$ where each x_i is either a string or a number. The x_t attribute is the target label. The classification task is: Given a new $(x_1, \ldots, x_n)$, predict the target label x_t.

Learning decision trees is done by accepting a set of labeled tuples (often very expensive to label), and recursively deciding what test (on which attribute) to apply at each node. The goal is to induce a decision tree that has good accuracy on the training set, and generalizes well on verification set.

A related view of decision trees is to consider the subspaces decision rules create—the tree recursively partitions the concept space into disjoint subspaces—with each subspace representing a particular label.

We can also view a decision tree as a collection of tests. Every path from the root node to the leaf represents a list of tests that need to be applied to classify an example with the

leaf label. With this view (treating the tree as a bag of tests as opposed to a hierarchical tree), it is easier to spot undesirable tests and remove them.

Most decision tree learning algorithms are variations on the top-down greedy search algorithm, with the most notable example being ID3 (Interactive Dichotomizer 3) by Quinlan. [Qui86]. Before Quinlan, Breiman et al. presented the concept in an 1984 book *Classification and Regression Trees* [BFSO84], and before that, Hunt did experiments with a Concept Learning System (CLS) [Hun66], which Quinlan references as inspiration and a precursor to ID3.

Hunt's Concept Learning System was a divide and conquer scheme that could handle binary (positive and negative) target values, with the decision attribute being decided by a heuristic based on the largest number of positive cases. Hunt speculated about using information theoretic measure—noting that humans in the game of 20 questions will ask questions that maximize information gain.

ID3 improves on Hunt's approach by using an information theoretic measure of *information gain* to decide on the test attribute, and allowing for multi-valued target labels.

C4.5, also proposed by Quinlan [Qui93], improves on ID3 with ability to handle numeric attributes and dealing with missing values. C4.5 also introduced a rule pruning mechanism—to avoid over-fitting the tree to the training data.

2.1 ID3

Starting with a set of tuples $\boldsymbol{X}$, where each tuple is $(x_1, \ldots, x_n, x_t)$, ID3 algorithm calculates *information gain* on every attribute, and uses the highest information gain attribute to split the dataset. The algorithm then proceeds recursively on each resulting tuple set.

Information gain measures the reduction in entropy of the target attribute minus the

entropy of the target attribute after the split on attribute i:

$$Gain(\boldsymbol{X}, i) = Entropy(\boldsymbol{X}) - \sum_{v \in Values(i)} \frac{|\boldsymbol{X}_v|}{|\boldsymbol{X}|} Entropy(\boldsymbol{X}_v)$$

where *Entropy* is a measure of bits it takes to represent the *target attribute* x_t and $\boldsymbol{X}_v$ is the set of tuples resulting from the split on value v of attribute i. The $|\boldsymbol{X}_v|/|\boldsymbol{X}|$ scales the split entropy by the number of elements in the set after the split on v.

$$Entropy(X) = -\sum_{v \in Values(\boldsymbol{X}_t)} P(v) \log_2 P(v)$$

where $\boldsymbol{X}_t$ are all the values of the target attribute x_t.

Consider the dataset in Table 1. We would like to use this dataset to decide on the form of transportation, given whether we are in a hurry, how much money we have, and whether the train is late. For example, if we are in a hurry, and have \$50, then we should take the taxi. Similarly, even if we are in a hurry, but only have \$10, then we are stuck taking the train.

Hurry	Money	TrainLate	Method
N	50	N	Train
N	50	Y	Taxi
Y	50	N	Taxi
Y	10	N	Train
Y	10	Y	Train
N	10	N	Train
N	10	Y	Train

Table 1: Decision Tree training tuple.

To build a decision tree using this dataset, we need to compute information gain of splitting on every column.

The entropy of the $Method$ column is:

$$-\left(\frac{2}{7}\log\left(\frac{2}{7}\right)+\frac{5}{7}\log\left(\frac{5}{7}\right)\right)=0.59827$$

The entropy of $Method$ after splitting on each other column is:

$$\begin{aligned}
\text{Hurry:} &\quad -((\tfrac{1}{4})\log(\tfrac{1}{4})+\tfrac{3}{4}\log(\tfrac{3}{4}))\tfrac{4}{7}+-(\tfrac{1}{3}\log(\tfrac{1}{3})+\tfrac{2}{3}\log(\tfrac{2}{3}))\tfrac{3}{7}=0.59413\\
\text{Money:} &\quad -(\tfrac{1}{3}\log(\tfrac{1}{3})+\tfrac{2}{3}\log(\tfrac{2}{3}))*\tfrac{3}{7}+0=0.27279\\
\text{TrainLate:} &\quad -((\tfrac{1}{4})\log(\tfrac{1}{4})+\tfrac{3}{4}\log(\tfrac{3}{4}))\tfrac{4}{7}+-(\tfrac{1}{3}\log(\tfrac{1}{3})+\tfrac{2}{3}\log(\tfrac{2}{3}))*\tfrac{3}{7}=0.59413
\end{aligned}$$

So we decide to split on the $Money$ column, this leaves us with two datasets, Table 2. The dataset where the only $Method$ is $Train$ cannot be split—in fact, the entropy for that one is 0. We only have to worry about the smaller dataset.

Hurry	Money	TrainLate	Method
Y	10	N	Train
Y	10	Y	Train
N	10	N	Train
N	10	Y	Train

Hurry	Money	TrainLate	Method
N	50	N	Train
N	50	Y	Taxi
Y	50	N	Taxi

Table 2: Dataset after splitting on the $Money$ column.

Now for the smaller of the two tables, the entropy of $Method$ after splitting on each other column is:

$$\begin{aligned}
\text{Hurry:} &\quad -(\tfrac{1}{2}\log(\tfrac{1}{2})+\tfrac{1}{2}\log(\tfrac{1}{2}))\tfrac{1}{2}+0=0.34657\\
\text{TrainLate:} &\quad -(\tfrac{1}{2}\log(\tfrac{1}{2})+\tfrac{1}{2}\log(\tfrac{1}{2}))\tfrac{1}{2}+0=0.34657
\end{aligned}$$

Since we get the same information gain, we can just pick to split on $Hurry$ column, Table 3.

We do the same logic for the $TrainLate$ column, and end up with a tree illustrated in Figure 1.

Using such a decision tree is very intuitive and it is easy to understand what the tree is

Hurry	Money	TrainLate	Method
N	50	N	Train
N	50	Y	Taxi

Hurry	Money	TrainLate	Method
Y	50	N	Taxi

Table 3: Splitting the smaller dataset on *Hurry* column.

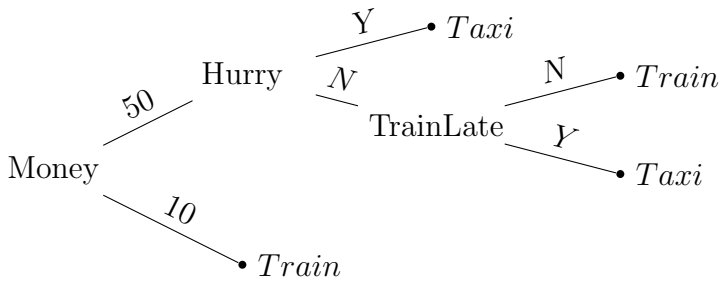

Figure 1: Decision tree built from the sample data.

doing. We start at the root, and determine if our *Money* amount allows for any options—if not, we take the *Train*. If we have enough money, we need to determine if we are in a *Hurry*, if yes, we take *Taxi*. If we are not in a hurry, we check to see if the train is late, if yes, we take *Taxi*, otherwise we take *Train*. Easy to understand and apply.

The major limitation of ID3 is the inability to calculate information gain for a numeric attribute. It is lucky that our *Money* variable only had two values. In general, numerical attributes, especially those that function as keys, will have very high information gain, without any predictive power.

Running ID3 to conclusion (recursively until the entire dataset is exhausted), results in over fitting the training data. This can be avoided by stopping the algorithm early or later pruning branches.

2.2 C4.5

The ID3 method is not without problems, and C4.5 is essentially a set of adjustments to the basic ID3 algorithm to make it work better. For one, the *Gain* has a tendency of favoring

unique identifiers. If we apply *Gain* on a database table, it will pick out all the keys, dates, ids, etc—none of which generalize.

When calculating which value to split on, C4.5 takes the number of distinct values into consideration. If a node will branch out a million different children, then we generally do not want to use that attribute.

$$Split(\boldsymbol{X}, i) = -\sum_{j=1}^{n} \frac{|\boldsymbol{X}_j|}{|\boldsymbol{X}|} \log_2 \frac{|\boldsymbol{X}_j|}{|\boldsymbol{X}|} \qquad GainRatio(\boldsymbol{X}, i) = \frac{Gain(\boldsymbol{X}, i)}{Split(\boldsymbol{X}, i)}$$

where $\boldsymbol{X}_j$s are n subsets resulting from partitioning $\boldsymbol{X}$ by n values of attribute i.

C4.5 introduces a way of dealing with numerical values. If an attribute i is numeric, it has N distinct numeric values (the datasets are finite). Such an attribute presents $N-1$ potential splits (we can split that attribute at any of the $N-1$ values.

To efficiently calculate the information gain for each of the $N-1$ split points of a numerical attribute we need to sort the dataset on values of that attribute. Once the numeric attribute is sorted, it is feasible to calculate information gain for each of the $N-1$ split points using a single iteration over the data.

Missing values are addressed by calculating the ratio of non-missing values within the split, and weighing the tuple with the missing values according to the ratio of the non-missing values. For example, if a node is to be split in two, and has 2 negative values, and 3 positive values, and two missing values, then the two missing values will be weighted as 0.4 negative, and 0.6 positive.

Another major improvement C4.5 brings is pruning. We start out over-fitting the tree—apply the tree building algorithm to completion. Then we convert the resulting tree into a bag of rules (each path from root to leaf becomes a conjunction rule). For each such conjunction rule, remove individual attribute tests if such a removal does not hurt the rule's classification performance (using validation tests). Sort the conjunction rules by their *estimated accuracy*

(estimated by applying rule to either training samples or a different verification set), and apply them in order until a rule succeeds—once a rule succeeds, we have our classification. This scheme avoids the dangers of over-fitting and under fitting.

3 Hyperplanes

Decision trees carve concept space using axis-aligned rules. In other words, each test (usually) goes after one attribute. This makes rules very easy to understand (a key feature of decision trees), but may cause a lot of unnecessary noise if there is a linear relationship between two attributes. An alternative is to use arbitrary hyperplanes. Unlike decision trees however, these are limited to numeric inputs.

A hyperplane is a fancy name for a 'plane' in N-dimensions: this is just a line in 2D, and a plane in 3D as illustrated in Figure 2.

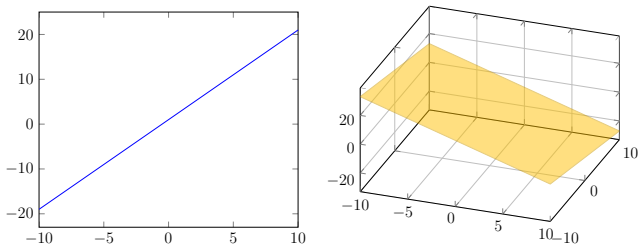

Figure 2: A line in 2D: $Ax + By = D$, and a plane in 3D: $Ax + By + Cz = D$

This can be extended indefinitely. To avoid running out of variables, we often write the plane as:

$$w_1x_1 + w_2x_2 + \cdots + w_nx_n = D$$

and to avoid that awkward D at the end, we often create $w_0 = -D$ and always set $x_0 = 1$. That way the whole thing becomes:

$$w_0x_0 + w_1x_1 + \cdots + w_nx_n = 0 \qquad \text{in vector notation: } \boldsymbol{w}^T\boldsymbol{x}$$

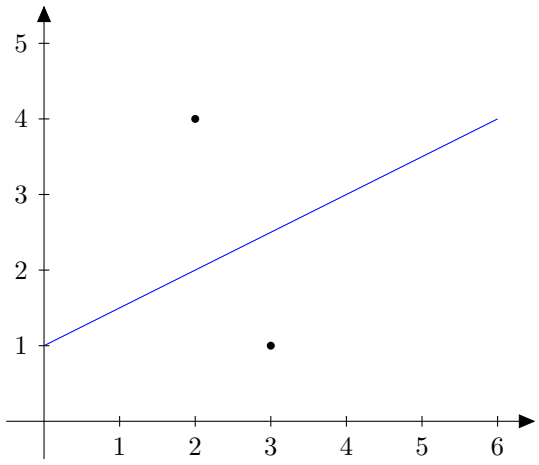

Figure 3: Dividing Line: $-x + 2y - 2 = 0$.

To check if a point $\boldsymbol{x}$ is exactly on the hyperplane we check $\boldsymbol{w}^T\boldsymbol{x} = 0$. To check if the point is in *front* of the plane, we check for $\boldsymbol{w}^T\boldsymbol{x} > 0$, and *back* of the plane as $\boldsymbol{w}^T\boldsymbol{x} < 0$.

For example, Figure 3, we have: $-x + 2y - 2 = 0$. That is, $w_1 = -1$, $w_2 = 2$, and $w_0 = -2$. Testing point $(2, 4)$ against it we get $-(2) + 2(4) - 2 > 0$, or *front* of the line. That is, $x_1 = 2$, $x_2 = 4$, and we always set $x_0 = 1$. Doing an inner product $\boldsymbol{w}^T\boldsymbol{x} > 0$. The second point $(3, 1)$ is checked the same exact way:

$$[w_0, w_1, w_2] \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = [-2, -1, 2] \times \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} < 0$$

Because $\boldsymbol{w}^T\boldsymbol{x}$ (or $(-2 \times 1) + (-1 \times 3) + (2 \times 1)$) is less then zero, we know $(3, 1)$ is on the *back* of the line. This is really the power of hyperplanes—they are a convenient linear modeling tool. The notion of *front* vs *back* is arbitrary (we can always flip the 'direction' of a line by multiplying all weights by -1).

3.1 Perceptrons

Perceptrons are literally hyperplanes, oriented in any direction. This power comes at a cost of readability—unlike decision trees, it is very hard to figure out what, if any, meaning exists in a hyperplane.

The idea of artificial neurons dates back to McCulloch & Pitts (1943) [MP43], when they proposed using an artificial neuron for computation—defining a mathematical abstraction of a biologically inspired neuron.

The McCulloch & Pitts neuron contains a set of real valued weights $(w_1, \ldots, w_n)$ and accepts $(x_1, \ldots, x_n)$ as input (without the $x_0 = 1$, that is handled separately by *threadhold*). The function it applies is:

$$o(\boldsymbol{x}) = step(\boldsymbol{x}^T \boldsymbol{w}) \qquad step(x) = \begin{cases} 1 & \text{if } y \geq threadhold \\ 0 & \text{otherwise} \end{cases}$$

where $\boldsymbol{x}^T \boldsymbol{w}$ is the inner product of $\boldsymbol{x}$ and $\boldsymbol{w}$ column vectors, and *step* a linear step function at *threshold*. The w_i values are normalized to a $(0, 1)$ or $(-1, 1)$ range, and both inputs and outputs are binary. See Figure 4.

Simple as they are, arrangements of such neurons were shown to compute any binary function. The inability to calculate XOR function (famously presented as a major limitation by Minsky & Papert (1969) [MP69]) is only applicable to a single layer of neurons. Since a neuron is a hyperplane, it cannot split XOR since that is not linearly separable, see Figure 5. Layering perceptrons into networks (as McCulloch & Pitts have done in their paper) overcomes this limitation.

McCulloch & Pitts did not define any training method; like programming, one had to adjust the neuron weights to compute different binary functions.

Perceptrons were developed by Frank Rosenblatt in 1958. [Ros58] Heavily based on the

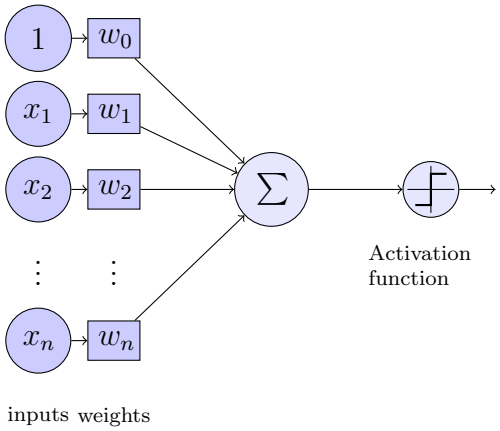

Figure 4: The Perceptron. Thought different models differ in detail, they all follow this general approach. (picture by *m0nhawk* on TEX StackExchange)

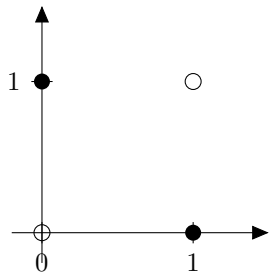

Figure 5: The XOR function. There is no single line that can separate the filled circles from non-filled ones.

McCulloch & Pitts neuron, this model used more flexible weights, and had an adaptive component. A perceptron is a function:

$$o(\boldsymbol{x}) = sign(\boldsymbol{x}^T \boldsymbol{w}) \qquad\qquad sign(x) = \begin{cases} 1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

where $\boldsymbol{x}^T \boldsymbol{w}$ is the inner product of input $\boldsymbol{x}$ and weight vector $\boldsymbol{w}$. The first input x_0 is always assumed to have value 1, and weight w_0 is the corresponding threshold.

Training a perceptron is iterative. For every training sample $(\boldsymbol{x}, y)$ we adjust the weights by

$$w_i = w_i + \lambda(y - o(\boldsymbol{x}))x_i$$

The $y - o(\boldsymbol{x})$ gets the classification error (or 0 if no error was made on the training example). The adjustment is then weighted by learning rate parameter λ and input x_i. In other words, if there was an error, and input x_i was tiny, we want to make a tiny adjustment to w_i. If x_i was large, then we want to make a similarly large adjustment to w_i.

For example, suppose the training example has $x = 2$ and $y = 1$, and our perceptron is $w = -1$ with 0 threshold. The $o(x)$ is threfore -1. We need to adjust weight *upwards* (governed by $y - o(x)$, in our example $1 - (-1) = 2$) by some fraction λ (perhaps set to 0.1) of x. Therefore we adjust weight by:

$$\lambda(y - o(\boldsymbol{x}))x_i = 0.1 \times (1 - (-1)) \times 2 = 0.4$$

A limitation of the above learning rule is that there is no notion of good $\boldsymbol{w}$ beyond the correct or incorrect value of 1 or -1, see Figure 6. If the sets are not separable, the perceptron will semi-randomly stumble in adjusting the weight vector until stopped. It is not a robust algorithm by any means, but it was a good start in the right direction.

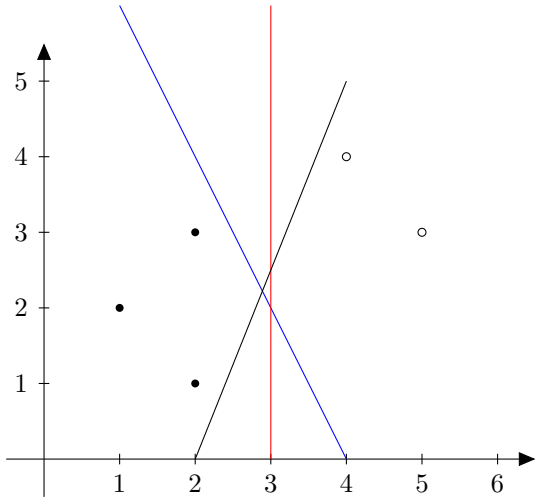

Figure 6: The perceptron learning rule will not adjust weights once they produce the correct classification—in other words, the different lines in this figure are all equally correct.

3.2 Delta Rule

Bernard Widrow (1965) [WML90] and Ted Hoff came up with a much better way to train the perceptron. Let us start by defining the total error for a neuron with weights $\boldsymbol{w}$:

$$E(\boldsymbol{w}) \equiv \frac{1}{2} \sum_{i \in D} (y_i - \boldsymbol{x}_i^T \boldsymbol{w})^2$$

where D is the set of all training data. The E function is essentially the *sum of squares* of all errors on the dataset D. We can differentiate E with respect to $\boldsymbol{w}$:

$$\Delta E(\boldsymbol{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right] \qquad \frac{\partial E}{\partial w_i} = \sum_{i \in D} (y_i - \boldsymbol{x}_i^T \boldsymbol{w})(-x_i)$$

The $\Delta E(\boldsymbol{w})$ is the gradient vector, with a component for each weight. We can adjust the weights via:

$$w_i = w_i + -\lambda \Delta E(\boldsymbol{w})$$

where λ is the learning rate. The $\Delta E(\boldsymbol{w})$ points in the direction that *increases* $E(\boldsymbol{w})$, so we need the negative sign to adjust in the decreasing direction. This derivation is adopted from Mitchell. [Mit97]

The iterative version of the delta rule is:

$$w_i = w_i + \lambda(y - \boldsymbol{x}^T\boldsymbol{w})x_i$$

This method has a smoother learning behavior than perceptron learning rule, and will continue to adjust $\boldsymbol{w}$ until it reaches an optimum value, even when all the examples are correctly classified.

The function is identical to the non-thresholded Perceptron learning—it is amazing what that small adjustment has done.

3.3 Least Squares

It turns out there is a more direct way to solve the weight adjustment problem. When the target label y is a real number (and $\boldsymbol{x}$s are numeric), the 'labeling' task becomes regression—we are fitting some function to data points. Least squares finds a hyperplane that best fits the $\boldsymbol{x}$s. Our model is:

$$\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$$

where $\boldsymbol{X}$ is a matrix with individual observables $\boldsymbol{x}$ as rows, $\boldsymbol{y}$ is a column vector of target variables. This is the heart of linear algebra. It seems we should be able to solve $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$ for $\boldsymbol{w}$ by solving a system of linear equations (e.g. $\boldsymbol{w} = \boldsymbol{X}^{-1}\boldsymbol{y}$). For example:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 2 \\ 1 & 4 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

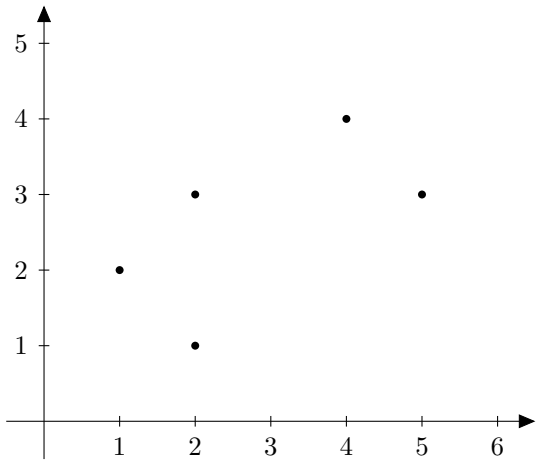

Figure 7: Sample points for Least Squares regression.

We can solve it by inverting $\boldsymbol{X}$, and solving for $\boldsymbol{w}$

$$\boldsymbol{X}^{-1} = \begin{bmatrix} 1 & 2 \\ 1 & 4 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -1 \\ -0.5 & 0.5 \end{bmatrix} \qquad \text{then} \qquad \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -0.5 & 0.5 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}$$

Unfortunately there are usually many more observables than $\boldsymbol{w}$s, such as:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 5 \end{bmatrix} \quad \boldsymbol{y} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 4 \\ 3 \end{bmatrix}$$

Such problems are overdetermined ($\boldsymbol{X}$ is not square), and training data contains noise—we would not be able to fit a line through the data, because no such line exists, see Figure 7.

We can rewrite the problem as: $\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w} = \boldsymbol{\gamma}$, where $\boldsymbol{\gamma}$ is the *error*, either positive or negative. We can square the error to get a positive number, then just as in Delta Rule,

minimize the square error. Rewriting the squared error function:

$$\begin{aligned} E(\boldsymbol{w}) = \|\boldsymbol{\gamma}\|^2 &= (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^2 \\ &\quad (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) \\ &\quad (\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}) \end{aligned}$$

One clever way of finding minimums (or maximums) is to differentiate, then set derivative to zero, and solve. The derivative is:

$$\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} = -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = 0$$

Which leads to what are called 'normal equations':

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y} \qquad \text{which leads to:} \qquad \boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

If $\boldsymbol{X}^T\boldsymbol{X}$ is invertible, that is it, we can directly solve for $\boldsymbol{w}$. If some columns of $\boldsymbol{X}$ are not independent, then is $\boldsymbol{X}^T\boldsymbol{X}$ not invertible, and we need to make an adjustment. Adding $\lambda\boldsymbol{I}$ to $\boldsymbol{X}^T\boldsymbol{X}$ ensures the inverse always exists. λ here is some tiny number, like 0.001.

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

This is the regression solution for situations when $\boldsymbol{X}$ is $N \times M$ matrix, and N is much *bigger* than M (has many more rows than columns). The $\boldsymbol{X}^T\boldsymbol{X}$ is a square matrix sized $M \times M$. Its running time depends on inverting $\boldsymbol{X}^T\boldsymbol{X}$—the algorithm is very fast for tall and skinny

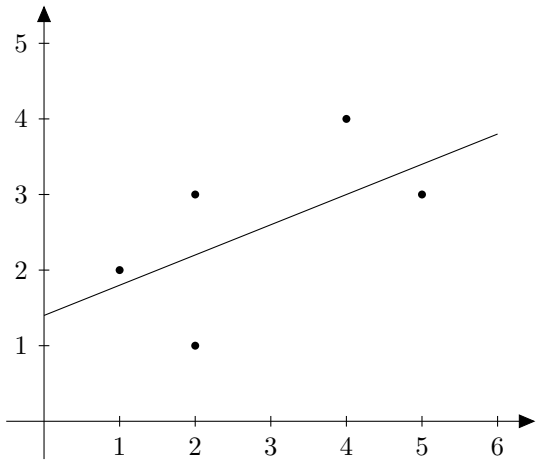

Figure 8: Sample points plotted along with the 'best' line: $y = 0.4x + 1.4$

matrices.

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, \quad \boldsymbol{X}^T\boldsymbol{X} = \begin{bmatrix} 5 & 14 \\ 14 & 50 \end{bmatrix}, \quad (\boldsymbol{X}^T\boldsymbol{X})^{-1} \approx \begin{bmatrix} 0.92 & -0.26 \\ -0.26 & 0.09 \end{bmatrix}$$

Plugging that into $\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$ we get

$$\boldsymbol{w} \approx \begin{bmatrix} 0.92 & -0.26 \\ -0.26 & 0.09 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 \\ 1 \\ 3 \\ 4 \\ 3 \end{bmatrix} = \begin{bmatrix} 1.4 \\ 0.4 \end{bmatrix}$$

This example dataset is probably not the best illustration of line fitting—the points do not appear to be on or near the line at all, see Figure 8 for the plot of $\boldsymbol{w}$. That said, the

least squares algorithm found the 'best' line to fit them anyway.

3.4 Least Squares Duality

Now for a bit of magic (Shawe-Taylor & Cristianini (2004) [STC04] derivation). Using *Sherman-Morrison-Woodbury formula* [GVL96] we can rewrite $(\boldsymbol{X}^T\boldsymbol{X}+\lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T$ as $\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T+\lambda\boldsymbol{I})^{-1}$, giving us another way of solving for $\boldsymbol{w}$ [4]:

$$\boldsymbol{w} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

This is the regression solution for situations when $\boldsymbol{X}$ is $N \times M$ matrix, and N is much *smaller* than M (has many more columns than rows). The $\boldsymbol{X}\boldsymbol{X}^T$ is a square matrix sized $N \times N$. Its running time depends on inverting $\boldsymbol{X}\boldsymbol{X}^T$.

Since most datasets have more records than attributes it seems this second derivation does not gain us much. A useful thing to notice is that $\boldsymbol{w}$s are now a linear combination of inputs.

$$\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{\alpha} \qquad \boldsymbol{\alpha} = (\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{y} = (\boldsymbol{G} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

where $\boldsymbol{G}_{ij} = \boldsymbol{x}_i^T\boldsymbol{x}_j$. To apply this $\boldsymbol{w}$ to a new sample $\boldsymbol{x}$:

$$\boldsymbol{w}^T\boldsymbol{x} = (\boldsymbol{X}^T\boldsymbol{\alpha})^T\boldsymbol{x} = (\boldsymbol{X}^T(\boldsymbol{G} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y})^T\boldsymbol{x} = \boldsymbol{y}^T(\boldsymbol{G} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}\boldsymbol{x} = \boldsymbol{\alpha}\boldsymbol{k}$$

where $\boldsymbol{k}$ is a vector where each $k_i = \boldsymbol{X}_i^T\boldsymbol{x}$.

[4] This does not imply $(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T$ is equal to $\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}$; pseudo-inverses are not exact.

Using our original example matrix, the $\boldsymbol{X}\boldsymbol{X}^T$ is much bigger than $\boldsymbol{X}^T\boldsymbol{X}$ for this problem:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, \quad \boldsymbol{X}\boldsymbol{X}^T = \begin{bmatrix} 2 & 3 & 3 & 5 & 6 \\ 3 & 5 & 5 & 9 & 11 \\ 3 & 5 & 5 & 9 & 11 \\ 5 & 9 & 9 & 17 & 21 \\ 6 & 11 & 11 & 21 & 26 \end{bmatrix}$$

Not surprisingly, $\boldsymbol{X}\boldsymbol{X}^T$ is not invertible (2nd and 3rd rows are the same). This is where the $\lambda\boldsymbol{I}$ adjustmenet becomes important. The $(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})$ is definitely invertible.

3.5 Discriminators

While the 'least squares' method described above is used primarily for interpolation and extrapolation, a similar technique can be used for classification. [LV00] The idea is to find the hyperplane that *splits* the provided training data.

Given a training set:

$$\boldsymbol{X} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_L, y_L)\}$$

where $y_i \in \{-1, +1\}$ indicates the class, we will use $\boldsymbol{X}_+$ as shorthand for all the positive training cases, and similarly $\boldsymbol{X}_-$ for all the negative ones. Our model is a hyperplane, with weights $\boldsymbol{w}$, and distance D, such that:

$$w_1 x_1 + \cdots + w_N x_N = D$$

With such a hyperplane, we get a notion of things being in 'front' of the plane and in the 'back' of the plane. If we plug $\boldsymbol{x}$ into the plane equation (represented by $\boldsymbol{w}$ and D), and get a positive value, then $\boldsymbol{x}$ is in front of the plane, etc. For this section, we will use the same

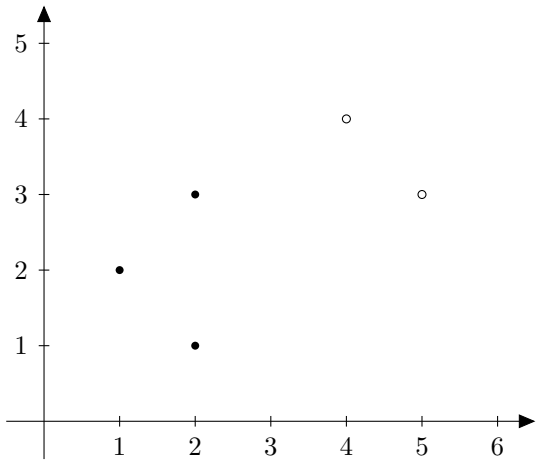

Figure 9: Example data for classification.

dataset as before, Figure 9.

$$\boldsymbol{X}_+ = \begin{bmatrix} 1 & 2 & +1 \\ 2 & 1 & +1 \\ 2 & 3 & +1 \end{bmatrix}, \quad \boldsymbol{X}_- = \begin{bmatrix} 4 & 4 & -1 \\ 5 & 3 & -1 \end{bmatrix},$$

3.5.1 Fisher's Linear Discriminant

Perhaps the simplest idea for a classifier is to calculate means of positive and negative examples: $\boldsymbol{\mu}_+$ and $\boldsymbol{\mu}_-$, then create a vector from one to the other and normalize:

$$\boldsymbol{w} = \frac{\boldsymbol{\mu}_+ - \boldsymbol{\mu}_-}{\|\boldsymbol{\mu}_+ - \boldsymbol{\mu}_-\|}$$

The D can be used to place the plane right between the two means, e.g.

$$D = \boldsymbol{w}\frac{1}{2}(\boldsymbol{\mu}_+ + \boldsymbol{\mu}_-)$$

This simple method is a special case of Fisher's Linear Discriminant (1936) [Fis36], and it works when covariance matrices for $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ are mostly multiples of identify matrix: both $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ are spheres around their respective means, with very little skew. When this is not the case, we need to incorporate the covariance matrices ($\boldsymbol{\Sigma}_+$ and $\boldsymbol{\Sigma}_-$) into the calculation:

$$\boldsymbol{w} = (\boldsymbol{\Sigma}_+ + \boldsymbol{\Sigma}_-)^{-1}(\boldsymbol{\mu}_+ - \boldsymbol{\mu}_-)$$

We are still creating a hyperplane from one mean to the next—we just twist it by the inverse of covariance matrices. For example, if $\boldsymbol{X}_+$ is particularly stretched out in direction i then the resulting $\boldsymbol{w}$ will be more orthogonal in that direction i. The D for the hyperplane can be chosen using the same method as above—or we can scale it by standard deviation away from each class (for situations when deviations of $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ are not the same).

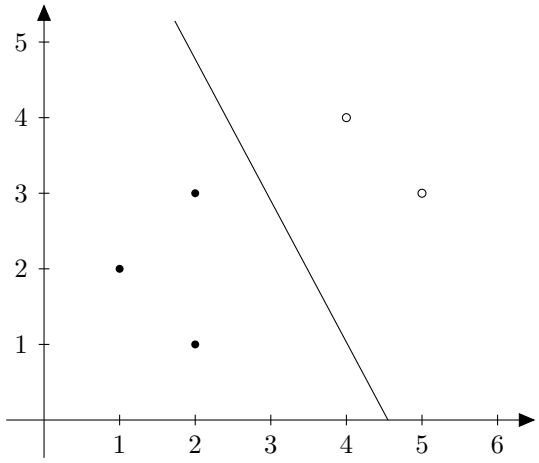

Figure 10: Fisher's Linear Discriminant, $0.88 * x + 0.47 * y - 4 = 0$

3.6 Maximal Margin Separator

When we apply the Fisher's Linear Discriminant $(0.88 * x + 0.47 * y - 4)$ to the two closest points, $(2, 3)$ and $(4, 4)$ we get -0.83 and 1.4 respectively. They are correctly classified as far as their sign is concerned, but their scale is different—if each is the closest point to the

hyperplane, what makes one more positive than the other negative? In other words, had we used just the closest points to build a linear discriminant we would get a different classifier.

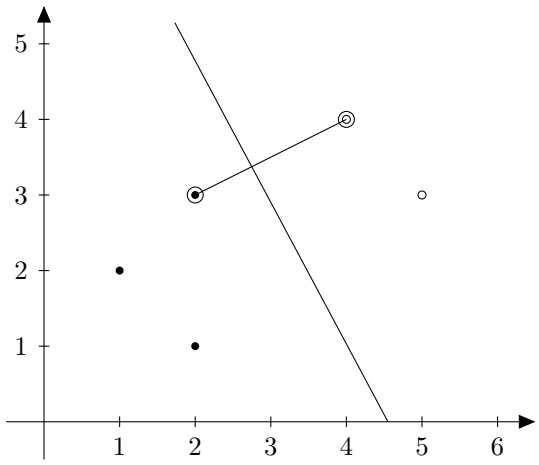

Figure 11: The two closest points to Fisher's Linear Discriminant are not the same distance to the separating hyperplane.

The maximal margin separator is essentially the idea that to achieve maximum generalizability, the separating surfaces should maximize distance to both negative and positive examples.

Let us start with the linear hyperplane model, $\boldsymbol{w}^T\boldsymbol{x} = D$, and assume that $\boldsymbol{w}$ is normalized (has a norm of 1). We can rewrite this as: $\boldsymbol{w}^T\boldsymbol{x} - D = 0$. For all $\boldsymbol{x}$ that are on the hyperplane, this model will produce 0.

For other $\boldsymbol{x}$ that are not on the hyperplane, this model produces the *distance* from the hyperplane to $\boldsymbol{x}$. We would like to maximize this distance, provided $\boldsymbol{x}$ is on the correct side of the hyperplane. In other words, maximize $\boldsymbol{w}^T\boldsymbol{x} - D$ subject to $y_i(\boldsymbol{w}^T\boldsymbol{x}_i - D) \geq 1$ for all training data i.

Suppose the maximum of $\boldsymbol{w}^T\boldsymbol{x} - D = \gamma$, we can then divide out the γ:

$$\frac{\boldsymbol{w}^T\boldsymbol{x}}{\gamma} - \frac{D}{\gamma} = \frac{\gamma}{\gamma} = 1$$

In other words, we can turn the maximization of $\boldsymbol{w}^T\boldsymbol{x} - D$ problem into the minimization of $\|\boldsymbol{w}\|$ problem, subject to same constraints. Note that here $\boldsymbol{w}$ is not normalized.

A way to solve such optimization problems is to use Lagrange multipliers. We rewrite the optimization problem as:

$$f(\boldsymbol{w}, D) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_i \alpha_i[y_i(\boldsymbol{w}^T\boldsymbol{x}_i - D) - 1]$$

Take derivatives, set to zero, and solve for $\boldsymbol{w}$:

$$\begin{array}{lll} \frac{\partial f(\boldsymbol{w},D)}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_i \alpha_i y_i \boldsymbol{x}_i = 0 & \Rightarrow & \boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x}_i \\ \frac{\partial f(\boldsymbol{w},D)}{\partial D} = \sum_i \alpha_i y_i = 0 & & \end{array}$$

The problem of course is that we need to solve for $\boldsymbol{\alpha}$ before we solve for $\boldsymbol{w}$. However, now that we have a solution for $\boldsymbol{w}$ we can just plug into the original formula and simplify:

$$\begin{aligned} f(\boldsymbol{\alpha}) &= \tfrac{1}{2}\left(\textstyle\sum_i \alpha_i y_i \boldsymbol{x}_i\right)^T \left(\textstyle\sum_j \alpha_j y_j \boldsymbol{x}_j\right) - \textstyle\sum_i \alpha_i \left[y_i \left(\left(\textstyle\sum_j \alpha_j y_j \boldsymbol{x}_j\right)^T \boldsymbol{x}_i - D\right) - 1\right] \\ &= \tfrac{1}{2}\textstyle\sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j - \left[\textstyle\sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j - 0 - \textstyle\sum_i \alpha_i\right] \\ &= \textstyle\sum_i \alpha_i - \tfrac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j \end{aligned}$$

Flipping the sign, we aim to minimize $f(\boldsymbol{\alpha})$:

$$\min_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) = \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j - \sum_i \alpha_i$$

subject to $\alpha_i \geq 0$ for all i, and $\sum_i \alpha_i y_i = 0$. This is a simple Quadratic Programming

problem that can be solved by setting up a linear KKT[5] system:

$$\left[\begin{array}{c|c} 0 & \boldsymbol{y}^T \\ \hline \boldsymbol{y} & \boldsymbol{H} \end{array}\right] \left[\begin{array}{c} -D \\ \hline \boldsymbol{\alpha} \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline \mathbf{1} \end{array}\right]$$

where $\boldsymbol{H}_{ij} = y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j$, $\boldsymbol{y} = (y_1, \ldots, y_L)$, $\mathbf{1} = (1_1, \ldots, 1_L)$. $\boldsymbol{H}$ is what is known as a Hessian matrix, and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_L)$ are Lagrange multipliers from the dual solution. Note that we can directly solve for $\boldsymbol{\alpha}$ and D (via least squares). We can then solve for $\boldsymbol{w}$ via:

$$\boldsymbol{w} = \boldsymbol{X}^T [\boldsymbol{\alpha} \times \boldsymbol{y}]$$

where $\boldsymbol{\alpha} \times \boldsymbol{y}$ is an element-wise multiplication. This $\boldsymbol{w}$ is not yet the maximal margin separator—next section on sparseness addresses the maximal margin.

3.7 Sparseness & Support Vector Machines

Above we saw that we can build regression and classification models using robust solutions, grounded firmly in linear algebra methods. The problem of course is the size of the $\boldsymbol{G}$ or $\boldsymbol{H}$ matrix. If we have a moderate number of samples, perhaps 10000, then we are talking about a non-sparse 10000×10000 matrix—and inverting something like that is a challenge—if we have a million samples, the whole solution becomes impractical.

A key insight comes from the $\boldsymbol{\alpha}$ vector. This is really what we are optimizing. Perhaps we can avoid solving for all of them at the same time? This is exactly the thinking behind Support Vector Machines, first proposed by Vapnik (1963) [VL63, CV95] and later developed by many others.

The SVM algorithms iteratively solves for some subset of $\boldsymbol{\alpha}$s, and continue to iterate

[5] Karush-Kuhn-Tucker

until all the $\boldsymbol{\alpha}$s satisfy the KKT. At this point, for most practical problems, only a tiny subset of the $\boldsymbol{\alpha}$s will be non-zero—meaning that only a tiny subset of the whole dataset is contributing to the $\boldsymbol{w}$. It also makes kernel appliation (Section 3.10) practical. Figure 12 shows an example of SVM for the example dataset.

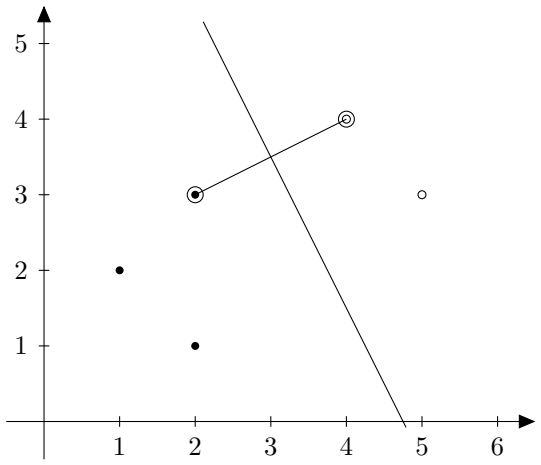

Figure 12: The SVM discriminator: $0.89443 * x + 0.44721 * y - 4.2485$, The $(2, 3)$ and $(4, 4)$ samples have $\alpha_i = 1$, the rest of the $\boldsymbol{\alpha}$ vector is 0.

The mechanical question then becomes how to setup the iteration to efficiently clamp down the value of most unimportant αs to zero. Practical considerations are: the iteration has a matrix inversion as the inner loop—the more α values we try to solve for, the more complicated this inner loop becomes.

Vapnik proposed a "chunking" algorithm. The key idea is that rows/columns with corresponding $\alpha_i = 0$ are irrelevant and can be skipped. Each iteration begins by collecting all non-zero Lagrange multipliers from last step and the M worst examples that violate KKT conditions (for some value of M). Since there is no hard control on how many non-zero Lagrange multipliers may exist from iteration to iteration, this solution has unpredictable runtime performance.

This was later improved by Osuna (1997) [OFG97] who proposed to have a constant size matrix. Every iteration would operate on the same number of Lagrange multipliers.

Osuma also proved that a large QP problem can be broken down into a series of smaller QP problems. As long as at least one example violates KKT conditions, then the overall problem will converge. Because the size of each sub-problem is fixed and limited, this solution can work on arbitrary large inputs—each subproblem doing a fixed amount of work.

Osuna's solution led Platt (1998) [Pla98] to develop the Sequential Minimal Optimization (SMO) algorithm, that uses just two Lagrange multipliers per iteration. It turned out that this can be solved analytically, avoiding the whole QP optimization as an inner loop, and is the fastest way of doing general SVMs right now.

Smola (2004) [SS04, WF05] applied SMO ideas to regression—the Lagrange multipliers are bounded to a certain margin around the hyperplane. Joachims (2006) [Joa06] has developed Boosting-like methods can be used to train linear SVMs in linear time. Syed (1999) [SLS99] developed methods to train SVMs incrementally.

3.8 Non-Linear Embedding

If we wanted to fit an exponential function, none of the above mentioned linear methods would work. One easy tweak we could do is 'embed' our linear data in non-linear space. We can do this by defining a non-linear function Φ, and transforming our data using that function. In other words, instead of working with $\boldsymbol{x}$, we would work with $\Phi(\boldsymbol{x})$.

The function Φ can be anything at all. It can reduce or increase the dimensionality of the sample point $\boldsymbol{x}$. For example, a 2-dimensional $\boldsymbol{x}$ may be turned into a 3-dimensional $\Phi(x)$:

$$\Phi(x_1, x_2) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

This has the capacity of turning our 'learning lines' method into a 'learning curves' method.

To see why this works, consider fitting points to $y = Be^{Ax}$. We can take log of both sides to get $\ln(y) = Ax + \ln(B)$, which is linear. The Φ embedding would apply the ln function

to Y, and upon output, apply exponential to get B.

Similarly, to fit power function $y = B * x^a$ we take log of both sides to get $\ln(y) = \ln(B) + a * \ln(x)$, which is now also linear.

To fit polynomials we "embed" higher dimensions that are powers of x. For example, instead of

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 5 \\ 1 & 7 \\ 1 & 11 \\ 1 & 13 \end{bmatrix}$$

which would fit a line, we can fit a 3rd degree polynomial just by tweaking that matrix to be:

$$\begin{bmatrix} 1 & 2 & 2^2 & 2^3 \\ 1 & 3 & 3^2 & 3^3 \\ 1 & 5 & 5^2 & 5^3 \\ 1 & 7 & 7^2 & 7^3 \\ 1 & 11 & 11^2 & 11^3 \\ 1 & 13 & 13^2 & 13^3 \end{bmatrix}$$

The resulting solutions will have the form $y = D + Cx + Bx^2 + Ax^3$. This can be extended to any degree polynomial you care to fit.

3.9 Logistic Regression

Logistic regression is a form of non-linear Φ embedding discussed above. Often we have problems that require learning and extrapolating probabilities—and those are always in the

0 to 1 range. To use a linear model, we need to project that 0 to 1 range onto the $-\infty$ to $+\infty$ range. We do this via the logit function

$$\text{logit}(x) = \log\left(\frac{x}{1-x}\right)$$

The logit function is the log of the Odds ratio, and is illustrated in Figure 13. Notice that 'probabilities' that are very close to 1, will get an extremely high y value (we need to clamp it at some high value, since ∞ is kind of hard to represent on a computer), and vice versa.

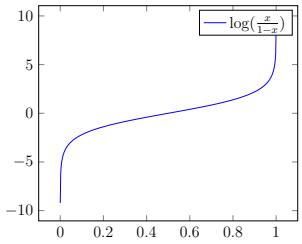

Figure 13: The logit function.

The embedding transforms the $\boldsymbol{y}$ target probability into logit($\boldsymbol{y}$). The model is trained on the transformed values. Once we have our linear model, the outputs of $\boldsymbol{w}^T\boldsymbol{x}$ will be linear and in $-\infty$ to $+\infty$ range. We need to turn those into probabilities—in other words, we need the inverse of the logit function, which happens to be the sigmoid function, Figure 14:

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

The output of logistic regression is determined via sigmoid($\boldsymbol{w}^T\boldsymbol{x}$) and it is always a value between 0 and 1; something that can be interpreted as 'probability'.

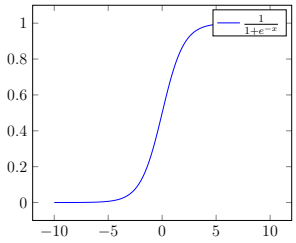

Figure 14: The sigmoid function.

3.10 Kernels

Kernels are just a clever method to do Φ embedding, utilizing the $\boldsymbol{G}$ matrix computation pipeline, without actually computing the Φ embedding. For example (due to [STC04]):

$$\begin{aligned} K(\Phi(\boldsymbol{x}), \Phi(\boldsymbol{z})) &= (x_1^2, x_2^2, \sqrt{2}x_1x_2)^T(z_1^2, z_2^2, \sqrt{2}z_1z_2) \\ &= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 \\ &= (x_1z_1 + x_2z_2)^2 \\ &= (\boldsymbol{x}^T\boldsymbol{z})^2 \end{aligned}$$

In other words, we can avoid a lot of calculation by simply squaring the elements of the $\boldsymbol{G}$ matrix. There are many more of these kernels—some even embed the data into an infinite dimensional space, such as the Gaussian kernel—this would be impossible to calculate without using this kernel trick.

Kernels have become a key piece in algorithm creation—regression and classification are domain and data independent: they will work on any data in any domain. Kernels on the other hand are often crafted along with data preparation.

Kernels take any two data examples, and produce what amounts to as a similarity score. This could be calculated via an algorithm, heuristically assigned, etc. For text data, this could be counting words. For image data, comparing histograms, etc.

4 Connectionism

The next logical progression from using hyperplane based rules is to connect many of them together. There are several broad architectures for doing this: fully connected, randomly connected, layered, etc. The links can be setup to send signals in either direction or just one. Hopfield Networks are examples of fully connected recurrent networks. Artificial Neural Networks, despite the broad name, are generally seen as layered feed forward networks with one or more 'hidden' layers, see Figure 15.

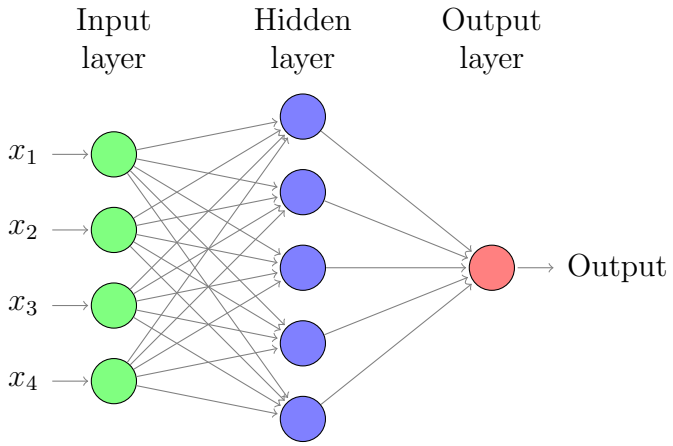

Figure 15: An example of a feed-forward Neural Network with one hidden layer. (picture by *Kjell Magne Fauske* on TEX StackExchange)

4.1 Hopfield Networks

A Hopfield Network is a fully connected recurrent artificial neural network developed by John Hopfield in 1982. [Hop82] It can serve as content addressable memory, meaning that the network can remember patterns, and when shown a partial pattern, can recall the full pattern. It is based on the idea introduced by Hebb (1949) [Heb49], that connections within

the brain roughly follow this pattern:

$$w_{ij} \sim \text{correlation}(x_i, x_j)$$

This leads to Hebbian Learning rule, which is to increase or decrease the w_{ij} depending on whether x_i and x_j correlate.

The activation function in a Hopfield network is:

$$a_i = \sum_j w_{ij} x_j \qquad\qquad x_i = \tanh(a_i)$$

The synchronous update first calculates all a_i values, then updates all x_is, the asynchronous does it one node at a time, perhaps for random i. The tanh function is similar to sigmoid but returns values in -1 to 1 range—in line with what a correlation output should be, see Figure 16.

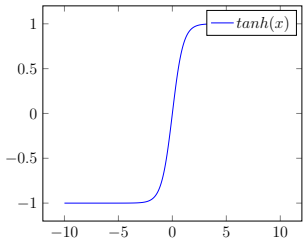

Figure 16: The tanh function.

For example (from MacKey [Mac02]), we could train the network with these patterns: `london-england`, `tokyo----japan`, `ottawa--canada`, `oslo----norway`, `paris---france`. The network, when presented with `::::::--canada` pattern, will iterate and recall the `ottawa--canada` pattern. Similarly, if the presented pattern is corrupted, such as, `otowa---canada`, the network will recall the corrected `ottawa--canada`.

4.2 Artificial Neural Networks

The term Artificial Neural Network almost always refers to a layered feed forward network, with one or more 'hidden' layers, as in Figure 15.

The biggest problem with these (and other) networks is training. Getting a single unit to output the correct values is mostly well understood (Support Vector Machines with meaningful Kernels (meaningful distance metric), are the pinnacle of such knowledge). How to train an entire network of units has not yet reached that level of understanding.

The original networks were either hand crafted, or randomly connected. Everyone was excited by the computing capabilities of networks (they can approximate any function, as shown by Hornik (1989) [HSW89])—but there was no automated way of getting networks to do the right things. The errors at the outer edge of the network were easily visible—and modifying the weights of the last layer via the Delta Rule works fine, but a method to propagate appropriate amount of error deeper into the network was needed.

The Backpropagation algorithm works by propagating the 'appropriate amount' of error from the outer layers into the 'hidden' layers. This appropriate amount is: weighted (by link weight) sum of errors this neuron is contributing to. Once we know the error for every neuron, we can use the Delta Rule to adjust the weight. This turned out to work very well for one hidden layer.

For more than one hidden layer, the backpropagated error loses its meaning. For first hidden layer, we the error is a weighted average of all forward errors—that in itself is vague. For two hidden layers, the error is a weighted average of weighted averages. In other words, backpropagation is only appropriate for shallow networks.

Training deep networks (with many hidden layers) can be achieved by building up the deep network one layer at a time—and training that layer via the shallow-training method. A method to do this is an autoencoder (feed forward neural network architecture), and

contrastive divergence (restricted Boltzmann machines). Like an onion, the outer layer is trained first, once the outer layer learns relevant features, a hidden layer is trained on those features. This continues for several layers—referred to as Deep Learning, as every layer adds depth to the network.

An interesting question to ask is what kind of functions are such layers computing/learning—what exactly is the representation that the network holds? A single layer of a neural network has N by M weights, where N is the number of inputs into a given layer, and M outputs. Let us define the $\boldsymbol{x}$ vector as the inputs, and the $\boldsymbol{W}$ matrix as all the weights. To calculate the linear output (activation), we just need to do:

$$\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} \qquad\qquad \boldsymbol{o} = trans(\boldsymbol{a})$$

The $\boldsymbol{a}$ is transformed via some threshold function, which can be a sigmoid function or just an identity (returning $\boldsymbol{a}$). This single layer computation has an incredible amount of power—for example, a whole family of frequency domain to time domain transforms fit this pattern: with appropriate weights, a single layer of a neural network is capable of picking out certain frequency bands.

Layering such transformations amounts to computing (and learning) very deep structural information of concept space. The downside of course is that the learned weights, even for only a single layer, are not human readable the same way that decision trees are. One can argue that this neural network AI is somehow weaker than logic or decision tree based AI, since it lacks the competence to explain how it is solving the problems—then again, humans cannot explain how to recognize faces either.

4.3 Backpropagation

Until late 1980s there was no systematic way of adjusting weights of hidden layers. Randomly adjusting weights did not work (garbage in, garbage out), and hand crafting networks is not practical—especially when the task they are solving cannot readily be explained in rules.

Breakthrough came with Rumelhart et al. (1986) [RHW86, Wer90] development of a mechanism to propagate output errors deeper into the hidden layers: backpropagation. Now there was a method of training that actually worked—for shallow networks.

For deeper networks, the backpropagation hit the *vanishing gradient problem*, first identified by Hochreiter (1998). [Hoc98, BSF94]. The general idea is that the output layer's gradient, when propagated back, gets averaged out with every layer backwards—eventually resulting in no meaningful gradient.

Backpropagation algorithm is essentially gradient descent of all the weights of the network. For gradient descent, we need to know the *error* or direction of where to move—once we know that, we can advance in that direction using the learning rate. Finding the direction of the move for a single perceptron was simple; it is just a difference between the expected (target) value and the actual output of the neuron. For a layered network, it gets a bit trickier. Essentially, we need to compute the derivative of the whole neural network—which is difficult. Backpropagation is a clever method to compute the network derivative a small piece at a time, using only the local information available at each neuron. Backpropagation is a two step process. We first feed forward the input through the network, keeping track of all output values at each layer. We then calculate the error of the final layer as we did in the single perceptron case (just take difference between expected and actual outputs, multiplied by the derivative of the *sigmoid* function[6]). We then propagate that error down the layers, adjusted (weighted average) by the appropriate weight (if the weight is high, then that link

[6]The derivative of *sigmoid* is just $sigmoid(x)(1 - sigmoid(x))$

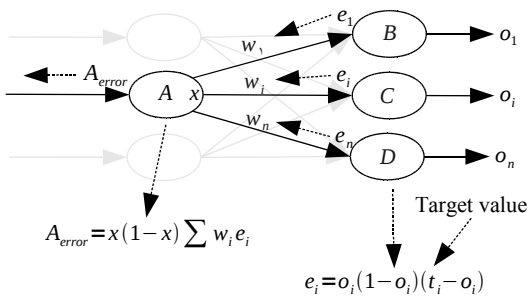

Figure 17: Backpropagation for a single link.

contributed a lot to that error, etc.). Refer to Figure 17: we directly calculate e_i (i.e.: error), and to *backpropagate* that error to a previous layer, we calculate the A_{error} by taking a weighted (the w_is) average of e_is, and multiplying that by derivative of *sigmoid* for that neuron. This can be repeated for any number of layers. Once we have an error for each neuron, we can use the perceptron learning algorithm on it.

Neural network training would normally iterate through the training data multiple times, applying backpropagation every time, until some condition is met. Either we iterate a fixed number of times, or stop when sum squared error stops improving or reaches some threshold.

In general, backpropagation tends to do very badly on non-trivial problems (such as networks with more than 2 layers), and requires quite a bit of tweaking. Somewhat paradoxically, it also tends to do badly on many trivial problems—simple functions with few dimensions. The reason is that with low dimensional inputs, there is a high chance of quickly getting stuck in a bad local minima, while with high dimensional inputs, there is a high chance of getting out of local minima via some downward leading dimension.

Some of the improvements involve adding momentum and/or adding regularization. Momentum considers previous weight update as part of current weight update—allowing gradient descent to roll over small bumps (local minima).

Regularization [Mac02] addresses a problem often associated with learning weights: the

perceptron learning rule encourages weights to get outrageously huge; over fitting the training set. This can often be avoided by starting with very small weights, and quitting before over fitting occurs. The regularization technique essentially adds a weight decay value, so with every iteration, weights tend to get smaller—even while the learning rule is trying to make them bigger. One of the problems with regularization is that the perceptron weights become 'stuck' near the origin.

4.4 Autoencoders & Deep Learning

The idea behind *autoencoders* is to train the network of the form pictured in Figure 18.

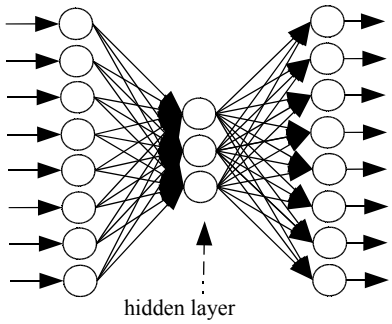

Figure 18: Learning the hidden layer finds a new compact representation of the data.

If we feed this network samples such as: 10000000, 01000000, 00100000, 00010000, 00001000, 00000100, 00000010, 00000001, and use input itself as the target, something remarkable happens: the network learns to equate the input with a binary encoding of the input, such as 001, 010, 011, 100, 101, 110, and 111. In other words, it finds a more compact representation of the data [Mit97].

Now consider a more extreme example in Figure 19. The input and output is a 256×256 image, and the goal is to learn the hidden layer. Once trained, the hidden layer will have a compact representation of an image—it will have the most important features of the input

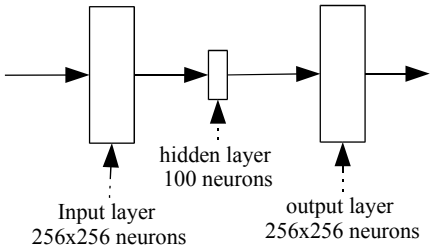

Figure 19: Learning the hidden layer finds the most important features of the input image.

image. Trained on a collection of images, it will pick out the most important feature out of all of them [HS06, SH07].

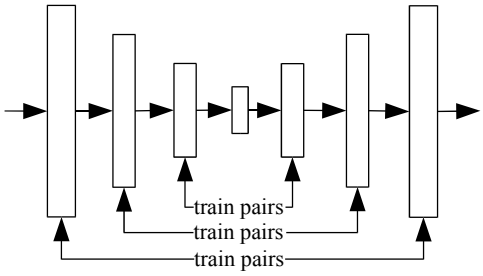

Figure 20: Autoencoders are trained in layers.

We can layer such architectures like onions, and train them in steps, as in Figure 20. First train the outer layer, then unroll, and train the inner layer using the inputs/outputs of the first layer. Then unroll again, etc. Such an approach allows us to build *deep neural networks*.

Applications for such layered autoencoders include image processing and character recognition [OH08, HOT06, HO06]. In computer vision, such techniques are used for object recognition [LHB04, LCH+06].

Essentially a deep network learns (unsupervised!) what the relevant features are of a particular dataset. Considering that a single-layer network can perform advanced transformations similar to a Fourier transform, and then apply a filter on them—layering such

transformations (while at the same time letting only the most important features deeper into the network) is an incredibly powerful modeling technique.

5 Probabilities

Probability is a tricky word—usually meaning the likelyhood of something occurring—or how frequent something is. Obviously, if something happens frequently, then its probability of happening is high. Below subsections go over some basics before diving into more advanced topics.

5.1 Basics

Probabilities always involve three things: a random variable X, an alphabet $\boldsymbol{\mathcal{A}_x}$, and the corresponding probabilities $\boldsymbol{\mathcal{P}_x}$. In this setup, X takes on values $x \in \boldsymbol{\mathcal{A}_x}$ with probability $\boldsymbol{\mathcal{P}_x}$. Probabilities of subsets are just sums of the individual elements of the subsets; if $\boldsymbol{T} \subseteq \boldsymbol{\mathcal{A}_x}$, then

$$P(\boldsymbol{T}) = P(x \in \boldsymbol{T}) = \sum_{a_i} P(x = a_i)$$

When more than one variable are involved, we have a *joint probability*. For two variables, we may write $P(x, y)$. For five, we may write $P(a, b, c, d, e)$.

For example, for a single die, the alphabet is $\{1, 2, 3, 4, 5, 6\}$, since any single throw can

land on some number 1 through 6. Consider throwing *two* dice, the outcomes may be:

$$
\begin{aligned}
2 &= \{1,1\} \\
3 &= \{1,2\} \text{ or } \{2,1\} \\
4 &= \{1,3\} \text{ or } \{2,2\} \text{ or } \{3,1\} \\
5 &= \{1,4\} \text{ or } \{2,3\} \text{ or } \{3,2\} \text{ or } \{4,1\} \\
6 &= \{1,5\} \text{ or } \{2,4\} \text{ or } \{3,3\} \text{ or } \{4,2\} \text{ or } \{5,1\} \\
7 &= \{1,6\} \text{ or } \{2,5\} \text{ or } \{3,4\} \text{ or } \{4,3\} \text{ or } \{5,2\} \text{ or } \{6,1\} \\
8 &= \{2,6\} \text{ or } \{3,5\} \text{ or } \{4,4\} \text{ or } \{5,3\} \text{ or } \{6,2\} \\
9 &= \{3,6\} \text{ or } \{4,5\} \text{ or } \{5,4\} \text{ or } \{6,3\} \\
10 &= \{4,6\} \text{ or } \{5,5\} \text{ or } \{6,4\} \\
11 &= \{5,6\} \text{ or } \{6,5\} \\
12 &= \{6,6\}
\end{aligned}
$$

That's 36 outcomes, each having 1 in 36 chance of occurring. For example, if we throw two dice, our chances of getting a "2", or $P(2)$ are $1/36$. Our chances of getting "11", or $P(11)$ are $2/36$ (since there are two subsets that add up to 11, namely, $\{5,6\}$ and $\{6,5\}$). What about $P(7)$? We can get that any number of ways:

$$\{1,6\} \text{ or } \{2,5\} \text{ or } \{3,4\} \text{ or } \{4,3\} \text{ or } \{5,2\} \text{ or } \{6,1\}$$

There are six ways of getting a "7". Each one of those has a $1/36$ chance of coming up, thus $P(7) = 6/36$.

What are the chances of us throwing a "7" where one of the die comes up as "1"? Here are the outcomes when at least one die is a 1:

$$\{1,1\}, \{1,2\}, \{1,3\}, \{1,4\}, \{1,5\}, \{1,6\}, \{2,1\}, \{3,1\}, \{4,1\}, \{5,1\}, \{6,1\}$$

That makes 11/36. We already know that chance of getting a "7" is 6/36. To add the probabilities gets us:

$$P(\text{at least one die is 1}) + P(7) = 11/36 + 6/36 = 17/36$$

But we counted some of them twice! $\{1, 6\}$ and $\{6, 1\}$ show up for both $P(\text{at least one die is 1})$ and $P(7)$, so we must subtract them. The end result is:

$$P(\text{at least one die is 1}) + P(7) - P(\{1, 6\} \text{ or } \{6, 1\}) = 11/36 + 6/36 - 2/36 = 15/36$$

To put that into set nation:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

We obtain a *marginal probability* $P(x)$ from a joint probability $P(x, y)$ via summation:

$$P(x) = \sum_{y \in \boldsymbol{A_y}} P(x, y)$$

This is often called *marginalization*, or *summing out*. For example, we can find the probabilities for a single die by summing out the 2nd die from example above.

Events tend to occur one after the other. Probability of x given y is called *conditional probability*, and is written $P(x|y)$. This is just a ratio:

$$P(x|y) = \frac{P(x, y) stupendously}{P(y)}$$

Rewriting conditional probability gives us the *product rule*:

$$P(x,y) = P(x|y)P(y) \qquad \text{or} \qquad P(x,y) = P(y|x)P(x)$$

If x and y are independent (have no influence on each other's occurrence), the product rule becomes:

$$P(x,y) = P(x)P(y)$$

A practical note on the product rule is that often we don't need to compute actual products of probabilities, but can work with sums of logarithms.

A variation on the product rule and marginalization gives us *conditioning*:

$$P(x) = \sum_{y \in \boldsymbol{\mathcal{A}_y}} P(x|y)P(y)$$

Simiarly, we can get a *joint probability* from conditional probabilities via the *chain rule*:

$$P(x) = \prod_{i=1}^{n} P(x_i|x_1, ..., x_{i-1})$$

In other words (writing out the above $\prod$ loop), we get:

$$\begin{aligned} P(a,b) &= P(a|b)P(b) \\ P(a,b,c) &= P(a|b,c)P(b|c)P(c) \\ P(a,b,c,d) &= P(a|b,c,d)P(b|c,d)P(c|d)P(d) \end{aligned}$$

and so on.

5.2 Bayes' theorem

Thomas Bayes (1702-1761) gave rise to a new form of statistical reasoning—the inversion of probabilities. We can view it as

$$Posterior = Likelihood \times Prior$$

where *Posterior* is the probability that the *hypothesis* is true given the evidence. *Prior* is the probability that the hypothesis was true *before* the evidence (ie: an assumption). *Likelihood* is the probability of obtaining the observed evidence given that the hypothesis is true.

Bayes' rule is derived from the *product rule*, by noting:

$$P(x|y)P(y) = P(y|x)P(x) \qquad \text{which leads to:} \qquad P(x|y) = \frac{P(y|x)P(x)}{P(y)}$$

It is worth thinking about this a bit. Consider the meaning of $P(y|x)$ above—if we fix x then $P(y|x)$ represents the probability of y given x—probabilities sum to 1. If on the other hand we fix y, then $P(y|x)$ turns into the *likelyhood* function—it does not sum to 1!

Another perspective: We can visualize $P(x, y)$ as a matrix, with all values of x being rows, and all values of y being columns. All entries in that matrix sum to exactly 1. If we wanted a matrix where each row sums to 1, then we would normalize by row—we would sum each row and divide each element of that row by that sum. By marginalization we get $P(x)$ which is that sum by row, and the matrix where each row sums to 1 is $P(x, y)/P(x)$.

What this really means is there is now a two step process. First, we pick a row, with probability $P(x)$, then within this row we pick an appropriate y with probability $P(x, y)/P(x)$, or to rewrite the same thing:

$$P(y|x) = \frac{P(x, y)}{P(x)}$$

Now the magic: before any observations, the probability of any particular row is $P(x)$, we call it the prior probability.

Let us say we observed a particular y, what is the probability of $P(x)$ after this observation? Well, it is obviously $P(x|y)$, but all we have is:

$$P(y|x) = \frac{P(x,y)}{P(x)}$$

Pretend we wanted to get back to the joint distribution $P(x,y)$, we would multiply

$$P(x,y) = P(y|x) * P(x)$$

Then to calculate $P(x|y)$. We would divide $P(x,y)$ or $P(y|x) * P(x)$ by $P(y)$. Note that we don't actually need this last step—since we know probabilities sum to 1, we can just calculate $P(y|x) * P(x)$, and then normalize the *columns* (not rows), and we'd get $P(x|y)$, which is a process that first picks a column (values of y), and then within that column picks a value of x with probability $P(x|y)$.

Once we know the probability $P(x|y)$ of picking a particular value of y (the event observation), we can replace $P(x)$, our prior, with the newly calculated $P(x|y)$, so next time we apply this rule again, we would be working with the new prior $P(x)$, that is adjusted for observing y.

5.3 Naive Bayes Classifier

We can use the Bayes rule to do document classification—commonly used to classify emails into spam/nospam categories. [SDHH98] For this to work, we need (either assumed, or

calculated) prior probabilities of certain word occurring in a certain document category

$$P(w_i|C)$$

where w_i is some word, and C is some document category (e.g.: spam, nospam, etc.). The probability of a given document D given a certain document category is:

$$P(D|C) = \Pi_i P(w_i|C)$$

note that none of the probabilities can be zero, otherwise the whole thing is zero. In practice, this is usually accomplished by specifying a very small number as the minimum probability (even if the word doesn't exist in a particular category). The product is also almost always replaced by a sum of logarithms.

Now we do the Bayes rules:

$$P(C|D) = \frac{P(D|C)P(C)}{P(D)}$$

Both $P(D|C)$ and $P(C)$ can be easily estimated from the training data (just count words by category). We never have to estimate $P(D)$ as it is only normalizing the results (making probabilities sum to 1)—which we don't need to determine which category is more likely.

5.4 Bayesian Networks

Given a probability distribution, say $P(a, b, c, d, e, f, g)$ we can calculate probability of anything we feel is useful. For example, if we wanted to know what is the probability of $P(a, c, g)$ we can just sum over the other variables. Similarly if some variables have definite values, e.g.: $P(a = true, c, f = false, g)$.

Note that the a, b, c, d etc., above can be all the everyday things. For example, a may be "fire alarm goes off", b may be "someone calls the fire department", c may be "all phones are dead", d may be "alarm clock goes off" and d may be "alarm clock gets confused with fire alarm", and so on.

In other words, having a probability distribution and the ability to extract information from it is incredibly useful. The major problem is that marginalization (summing out) is terribly expensive to do: it is an exponential operation. To go from $P(a, b, c)$ to $P(a, b)$ we need to sum over *all* the possible values of c—and this adds up to impractical very quickly.

Recall joint probability distributions rewritten as conditional probabilities:

$$\begin{aligned} P(a, b) &= P(a|b)P(b) \\ P(a, b, c) &= P(a|b, c)P(b|c)P(c) \end{aligned}$$

Now imagine that we knew that a was not dependent on b, c. For example, "fire alarm goes off" is independent of "alarm clock goes off". The above then could be rewritten as:

$$\begin{aligned} P(a, b) &= P(a)P(b) \\ P(a, b, c) &= P(a)P(b|c)P(c) \end{aligned}$$

This is suddenly much simpler to deal with—in terms of marginalization.

Bayesian Networks [Nea03, RN02] then is a method to write a probability distribution that explicitly specifies conditional *independence*. In a joint probability, everything is assumed to be dependent on everything else—in Bayesian networks we are explicitly saying that some things have nothing to do with each other.

Practically speaking, in the worst case, evaluating Bayesian networks is still exponential (NP-Complete in fact [DL93, GJ79]). We can remove some dependencies, but the ones that are left are still going to cause exponential calculations.

Folks have came up with various sampling algorithms to poke at the exponential search space and make predictions—without actually going through the entire search space.

The above just deals with *evaluating* a Bayesian network, but not with actually coming up with one. The simplest way of coming up with one is to ask an expert to draw it (and specify probabilities). This is similar to having an expert specify rules of an expert system.

As a fall back, the expert can specify the structure, and the probabilities can be learned from data (perhaps using Bayes rule for learning, or maybe just finding aggregates in the dataset).

If we don't have an expert to specify a Bayesian network, and we need to learn both the structure and probabilities from data, the task becomes hard. NP-Complete hard. [Chi96] There are many folks doing research into this, but so far, for any interesting size problem, the task is impractical. The main problem is that it is very hard to claim independence with only a finite number of input samples.

There does appear to be a simple workaround: if the training dataset is big, say 100 gigabytes, with maybe 1000 variables (this is currently *very* impractical by the best of the best Bayesian network learning approaches), then instead of learning the Bayesian network and then using it for inference (which is impractical), just run through the dataset (in linear time). We can answer *any* statistics question on this dataset by scanning through it once and maintaining aggregates. If we need to answer another question, just scan through the dataset again—certainly better than spending an exponential amount of time learning a Bayesian network, and then spending an equally exponential amount of time evaluating (even via sampling) the learned network.

5.5 Monte Carlo Integration

The key difficulty in evaluating Bayesian Networks is evaluating integrals. *Monte Carlo* integration is an algorithm that relies on random sampling to compute an integral, possibly of a very complicated multidimensional function. This is accomplished by generating samples within the integral and then averaging them together (weighted average, via probability). The major problem, of course, is how to sample an arbitrary probability distribution. Given any integral:

$$\int_a^b h(x)dx$$

If we can decompose $h(x)$ into function $f(x)$ and a probability distribution $p(x)$ defined over an interval (a, b), then we have:

$$\int_a^b h(x)dx = \int_a^b f(x)p(x)dx = E_{p(x)}\left[f(x)\right] \simeq \frac{1}{n}\sum_{i=1}^{N} f(x_i)$$

As a simple example, we can use this technique to estimate area under a quarter circle, and with that, get an estimate for value of π:

$$\pi \approx \frac{4}{N}\sum_{i=1}^{N} \begin{cases} 1 & \text{if } x_{random}^2 + y_{random}^2 \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

where x_{random} and y_{random} are random numbers from 0 to 1, generated on every loop. The bigger N is (more samples in the integral), the more accurate the estimate for π. Now, consider marginalization:

$$P(x) = \sum_{y \in \mathcal{A}_y} P(x|y)P(y)$$

What if instead of going through all the values of $\mathcal{A}_y$, we randomly sample them? This is the gist of a Monte Carlo method.

Often, the distribution we wish to sample is very complex and has many dimensions. The *Markov Chain Monte Carlo* method [Wal, GC92] constructs a Markov Chain with stationary distribution we wish to sample. The sampling process then just has to walk the chain for a long enough time to produce accurate samples.

A popular MCMC algorithm is a special case of Metropolis-Hastings, called *Gibbs sampling*. If we have random variables X, and Y, and wish to sample $f(x)$, but only have $f(x|y)$ and $f(y|x)$, we can generate a *Gibbs sequence* via:

$$\begin{aligned} X_j &\Leftarrow_{sample} f(x|Y_j = y_j) \\ Y_{j+1} &\Leftarrow_{sample} f(y|X_j = x_j) \end{aligned}$$

where y_0 is specified (or guessed); remainder of the sequence:

$$X_0, Y_1, X_1, Y_2, X_2, \ldots$$

is generated by iteratively applying the above rules. Such a rule can easily be generalized to more than two random variables:

$$\begin{aligned} X_j &\Leftarrow_{sample} f(x|Y_j = y_j, Z_j = z_j) \\ Y_{j+1} &\Leftarrow_{sample} f(y|X_j = x_j, Z_j = z_j) \\ Z_{j+1} &\Leftarrow_{sample} f(z|X_j = x_j, Y_{j+1} = y_{j+1}) \end{aligned}$$

In the above case, the initial values for y_0 and z_0 are specified, etc.

Gibbs sampling sits at the core of most efficient (practical) methods involving Bayesian inference. Applications for the above are mostly simulations to compute some value that doesn't have a closed form, such as valuation of equity indexed annuities [HC07], or computing the credit score [SBEP+02].

6 Ensemble Learning

Meta algorithms (sometimes referred to as ensembles) are algorithms that use multiple copies of other algorithms as their building blocks. By combining several algorithms, we can often achieve better performance than we can get from any single one. For example, a search engine that queries a dozen other search engines and presents the top results from all, may actually be better than any individual search engine.

This is similar to asking a crowd the weight of Mnt.Washington. While individuals will give wild numbers between a few grams to possibly the weight of the whole planet, the average of all results will be pretty close to the actual weight of Mnt.Washington. There's an interesting account of this by Francis Galton, who in 1906 observed a contest where villagers were guessing the weight of a publicly displayed ox. While none of the 800 villagers guessed the correct weight of 1,198 pounds, the average of their votes came very close: 1,197 pounds [Sur04].

6.1 Bagging

Bootstrap aggregating or Bagging is a meta algorithm to improve training stability [Bre96, BY01]. The gist of the algorithm is: Given a training set, uniformly sample the set with replacement to make many training sets. Use each of those samples to train a corresponding classifier (perhaps in parallel). The resulting final classifier is an average (or a vote) of the individual trained classifiers.

The motivation is to generate smaller training sets that highlight some feature of the training data—feature that would be difficult to train for given the entire data set. For example, if we apply bagging on a Decision Tree, we are likely to find different splits, especially deeper in the tree.

Bagging is a very popular method for parallelizing algorithms across a cluster of machines—

the data is already 'randomly' sharded (distributed), and each machine builds a model using only the data it has local access to.

6.2 Boosting

Boosting [Fre90, Sch] is similar to Bagging, except the generation of training sets is serialized. After each iteration of the algorithm, the samples that were misclassified are given higher weight (for sampling) for the next iteration. This ensures that the problematic examples, those that are misclassified, get the most attention from subsequent classifiers.

Unlike Bagging, it is very tricky to apply boosting in a clustered environment, as each iteration must reweigh the training data—almost certainly causing data movement—unless such re-weightings are confined to within each node.

7 Clustering

Clustering is a mechanism to group similar items together—primarily so we could work with groups instead of individual samples. There are two problems with this: what does similarity mean, and what exactly is a group? We need to define *both*.

The choices are often domain specific, for example, clustering locations might use Euclidean distance, but clustering financial instruments like Bonds might require some heuristic 'distance' measures. If distance measure is not appropriate to the task, the clusters will often be meaningless.

Next we need to decide how clusters will be formed, and how many we want. If we have one unlabeled sample, then obviously it is in the group of one, all by itself. There really is no need for such clusters. If we have *two* samples, we suddenly have a choice to make: do we create one cluster or two? It turns out that this choice is arbitrary—if we prefer to have

one cluster, we create one cluster. Or we could create two.

7.1 k-Means

k-Means is perhaps the most popular clustering algorithm. It was first introduced by Hugo Steinhaus in 1956 [Ste56], and later revised and extended by [Llo57, Llo82, Mac67]. The algorithm is designed to cluster data into k buckets, usually based on Cartesian distance between input samples. It is an example of an *Expectation Maximization* algorithm, which is a general class of two step algorithms, where first step projects our expectations into the model, and second step adjusts the model so as to maximize our projections.

The algorithm starts with k random means. During each iteration, each training sample gets the label of the closest mean. Once all training samples are labeled, we recalculate k means using the samples assigned with each label. This repeats while training samples are changing labels from iteration to iteration.

The output of this procedure is obviously the k mean points. To find a cluster for any new input sample, we compute its distance to every mean, and assign it to the closest one.

7.2 Hierarchical Clustering

Hierarchical clustering looks for a whole hierarchy of clusters. This can be Agglomerative or Divisive.

Agglomerative clustering starts with each sample point in a cluster of its own. Each iteration merges two closest points or clusters (to form bigger clusters). The end product is one cluster that contains everything—along with the sequence of what falls into what cluster when.

Divisive clustering is the reverse of Agglomerative: starts off with one big cluster, and starts splitting off farthest things out.

7.3 Manifold Clustering

Traditional clustering method implicitly assume that clusters are centered around points, and even when customized, distance measures often assume coordinate independence. When trying to cluster abstract things like Bonds, for example, some of those assumptions may not apply. For example, for bonds, time to maturity, yield, and current price are all related and coordinates have different scale and units. It is not unreasonable to assume that individual bonds sit on some higher dimensional structure—not a single point—that represents relationships among different bonds.

Manifold clustering attempts to discover these higher dimensional structures. The methods and the types of manifolds detected differ.

Linear manifold clustering [HH07] attempts to find linear manifolds by sampling a certain number of points (for example, 3 for a plane), constructing the manifold, and then seeing if other points are also on the manifold. Using the distance-to-manifold histogram for all samples, this method determines if this really is a genuine manifold (the histogram indicates a separation of points on the manifold and not on the manifold).

Other methods to find manifolds involve looking for sparse manifolds near sample points. The idea is that neighboring sample points are very likely part of the same manifold and can be linked (with a weight). This appears to work for sparse manifolds without much clutter. [EV11]

8 Dimension Reduction

Imagine we take a picture of a 3D room—we get a million pixels. We now have a million dimensional "sample" of that 3D room. We can take a great many of such pictures of the same room—each one a million dimensional sample—all representing the same 3D room. It

quickly becomes clear that all these pictures are really representing a 3D room (that is what we are photographing), and the variation is in lighting, direction of camera, etc., perhaps adding a few more dimensions—but certainly not a million. A great many problems have this ridiculously high dimensionality.

8.1 Principal Component Analysis

Perhaps the simplest and most overused method of dimension reduction is PCA, or Principal Component Analysis. Intuitively, principle components represent directions of variation. If we had a point cloud the shape of an average car, the first principle component would be along the length of the car (the longest dimension). The second principle component would be along the height or width, depending which was the second longest dimension of the car. For a 3D car, we would find three principle components—even if the original dataset had 500 dimensions. PCA is an automated mechanism to find such dimensions—for arbitrary data.

Practically this often means taking high dimensional data, and arranging it as columns of a matrix. If we are dealing with pictures of faces, where each picture is perhaps 256x256 pixels, then each column of a matrix will be 65536 numbers, and there will be a column for every training picture. A huge matrix. Applying PCA on such a matrix will result in a set of orthogonal vectors, that are called eigenvectors. For the 'pictures of faces', such eigenvectors (these are 65536 numbers, or 256x256 'images') represent typical components of faces—all other faces are linear combinations of these eigenfaces. The key here is that there are only perhaps a dozen such eigenfaces. To compare two face images, we project both of them onto these few eigenfaces, and compare distance within this low dimensional space. This is exactly the method used by Sirovich & Kirby [SK87], and Turk & Pentland [TP91] for face recognition, and it turned out that faces are relatively low dimensional—the

Turk paper only used 7 eigenfaces.

The PCA approach is a lot more general than face recognition. It is a by product of one of the most useful matrix factorization methods ever, the Singular Value Decomposition: $X = U\Sigma V^*$

8.2 Vector Quantization

Vector quantization [Gra84] is a clustering-like method to reduce dimensions. It has applications in speech recognition—the high dimensional frequency spectrum of speech is quantized into several thousand distinct 'states'—which are then used to train the Hidden Markov Model.

The algorithm for vector quantization is surprisingly simple. A cluster is defined by the mean of all samples it contains. Each of the first N samples becomes its own cluster. For each subsequent example, test distance to each cluster. If that distance is smaller than the largest inter-cluster distance, then add example to the closest cluster, otherwise merge two closest clusters, and new sample becomes a new cluster. An efficient method to implement it is to realize that the algorithm only needs to maintain sums and counts—not the actual cluster membership of all past samples.

9 Conclusion

Machine Learning and Pattern Recognition is an amazingly broad field with a wide assortment of applications in all sorts of domains. In this paper we reviewed key concepts that glue the field together.

Decision trees, unlike the purely numerical methods, are human readable: we can follow along step-by-step through whole decision process. When people understand how the decision

was reached, they trust the decisions.

In numeric settings, decisions often take the form of hyperplanes. These are just a linear combinations of components—where each component may be an arbitrary function of the input. This has incredible flexibility—in modeling both linear and non-linear concepts.

Connecting multiple decision mechanisms together gets us networks. They appear to offer endless computational possibilities, if only we could figure out a way to train them properly. Each layer of linear decision rules is capable of non-trivial transformations, along with filtering. Unfortunately the backpropagation algorithm proved incapable of training anything but shallow networks.

For deeper networks, unsupervised approaches based on layered auto-encoders and layered restricted Boltzmann machines are gaining traction. These methods attempt to model the data distributions—with each layer output attempting to recreate that layer's input. This leads to very powerful and deep feature extraction.

Clustering and dimension reduction are in the same class as deep networks: they are an unsupervised mechanism to identify (hopefully meaningful) features.

Bayes' theorem touches at the heart of knowledge acquisition. What do we know, and what are the assumptions? In many situations, assumptions are the key to the whole problem. For example, a medicine that works 5% of the time is useless. But that same exact medicine works 100% of the time for 5% of the people.

Ensemble learning algorithms, such as Bagging, are pretty much the only way to approach learning in the world of Big Data. It is not just an optimization mechanism anymore—it is mostly about practical data locality.

This is an exciting field. One can think of it as attempting to learn how understanding works. How to model and automate intelligence. It is perhaps the most exciting and most intellectually challenging area not just in computer science but in all of human endeavors.

References

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[BFSO84] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification And Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[BSF94] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term Dependencies With Gradient Descent Is Difficult, 1994.

[BY01] P. Buhlmann and B. Yu. Analyzing bagging. *Annals of Statistics*, 2001.

[CH67] Thomas M. Cover and Peter E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[Chi96] David M. Chickering. Learning Bayesian networks is NP-Complete. In D. Fisher and H. Lenz, editors, *Learning from Data: Artificial Intelligence and Statistics V*, pages 121–130. Springer-Verlag, 1996.

[CV95] Corinna Cortes and Vladimir Vapnik. Support Vector Networks. In *Machine Learning*, volume 20, pages 273–297, 1995.

[DL93] Paul Dagum and Michael Luby. Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artif. Intell.*, 60(1):141–153, 1993.

[EV11] Ehsan Elhamifar and René Vidal. Sparse manifold clustering and embedding. In J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger,

editors, *Advances in Neural Information Processing Systems 24*, pages 55–63. Curran Associates, Inc., 2011.

[Fis36] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(7):179–188, 1936.

[FJ51] Evelyn Fix and Jr. Discriminatory analysis: Nonparametric discrimination: Consistency properties. Technical Report Project 21-49-004, Report Number 4, USAF School of Aviation Medicine, Randolf Field, Texas, 1951.

[Fre90] Yoav Freund. Boosting a weak learning algorithm by majority. In *COLT: Proceedings of the Workshop on Computational Learning Theory, Morgan Kaufmann Publishers*, 1990.

[GC92] Edward I. George George Casella. Explaining the gibbs sampler. *The American Statistician*, 46(3):167–174, 1992.

[GJ79] Michael R. Garey and David S. Johnson. *Computers And Intractability: A Guide To The Theory Of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[Gra84] R.M. Gray. Vector quantization. *ASSP Magazine, IEEE*, 1(2):4–29, April 1984.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[HC07] Ming-hua Hsieh and Yu-fen Chiu. Monte carlo methods for valuation of ratchet equity indexed annuities. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 998–1003, Piscataway, NJ, USA, 2007. IEEE Press.

[Heb49] Donald O. Hebb. *The Organization Of Behavior: A Neuropsychological Theory.* Wiley, New York, June 1949.

[HH07] Robert Haralick and Rave Harpaz. Linear manifold clustering in high dimensional spaces by stochastic search. *Pattern Recogn.*, 40(10):2672–2684, October 2007.

[HO06] Geoffrey E. Hinton and Simon Osindero. A fast learning algorithm for deep belief nets. *Neural Computation*, 18, 2006.

[Hoc98] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116, April 1998.

[Hop82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558, Apr 1982.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comp.*, 18(7):1527–1554, July 2006.

[HS06] G. E. Hinton and R. R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[Hun66] Earl B Hunt. *Concept Learning: An Information Processing Problem.* Wiley, 1966.

[Joa06] Thorsten Joachims. Training linear SVMs in linear time. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 217–226, New York, NY, USA, 2006. ACM.

[LCH+06] Yann Lecun, S. Chopra, R. Hadsell, F. J. Huang, and M. A. Ranzato. *A Tutorial On Energy-Based Learning*. MIT Press, 2006.

[LHB04] Y. Lecun, F. J. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. volume 2, 2004.

[Llo57] S. P. Lloyd. Least squares quantization in PCM. Bell Telephone Laboratories Paper., 1957.

[Llo82] S. P. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[LV00] S. Lukas and L. Vandewalle. Sparse least squares support vector machine classiers. *European Symposium on Articial Neural Networks*, pages 37–42, 2000.

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. Fifth Berkeley Symp. on Math. Statist. and Prob*, volume 1, pages 281–297. Univ. of Calif. Press, 1967.

[Mac02] David J. C. Mackay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, June 2002.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[MP43] Warren McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction To Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[Nea03] Richard E. Neapolitan. *Learning Bayesian Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[OFG97] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, pages 276–285, Sep 1997.

[OH08] Simon Osindero and Geoffrey Hinton. Modeling image patches with a directed hierarchy of Markov random fields, 2008.

[Omo89] Stephen M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

[Pla98] John C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report, Advances in Kernel Methods, Support Vector Learning, 1998.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[Qui93] J. Ross Quinlan. *C4.5: Programs For Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[Ros58] F. Rosenblatt. The perception: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959.

[SBEP+02] Monte Carlo Search, Bart Baesens, Michael Egmont-Petersen, Roberto Castelo, and Jan Vanthienen. Learning Bayesian network classifiers for credit scoring using Markov Chain Monte Carlo search. In *Proc. International Congress on Pattern Recognition*, pages 49–52, 2002.

[Sch] R. Schapire. The boosting approach to machine learning: An overview. In MSRI Workshop on Nonlinear Estimation and Classification, Berkeley, CA, Mar. 2001.

[SDHH98] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[SH07] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. In *Proceedings of the SIGIR Workshop on Information Retrieval and Applications of Graphical Models*, Amsterdam, 2007.

[SK87] L. Sirovich and M. Kirby. Low-Dimensional Procedure For The Characterization Of Human Faces. *Journal of the Optical Society of America A*, 4(3):519–524, 1987.

[SLS99] N. Syed, H. Liu, and K. Sung. Incremental learning with support vector machines. *International Joint Conference on Artificial Intelligence*, 1999.

[SS04] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, August 2004.

[STC04] John Shawe-Taylor and Nello Cristianini. *Kernel Methods For Pattern Analysis*. Cambridge University Press, June 2004.

[Ste56] Hugo Steinhaus. Sur la division des corp materiels en parties (in french). In *Bull. Acad. Polon. Sci.*, volume 4, pages 801–804. Univ. of Calif. Press, 1956.

[Sur04] James Surowiecki. *The Wisdom Of Crowds*. Doubleday, 2004.

[TP91] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *J. Cognitive Neuroscience*, 3(1):71–86, January 1991.

[VL63] V. Vapnik and A. Lerner. Pattern Recognition Using Generalized Portrait Method. *Automation and Remote Control*, 24, 1963.

[Wal] B. Walsh. Markov Chain Monte Carlo and Gibbs Sampling. Lecture Notes for EEB 581, version 26 April 2004.

[Wer90] P.J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools And Techniques, Second Edition (Morgan Kaufmann Series In Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[WML90] Bernard Widrow, Michael, and A. Lehr. 30 years of adaptive neural networks. In *Proceedings of the IEEE*, pages 1415–1442. IEEE press, 1990.