# Chapter 11

# Machine Learning

> Any sufficiently advanced form of cheating is indistinguishable from learning.
>
> – Jan Schaumann

For much of my career, I was highly suspicious of the importance of machine learning. I sat through many talks over the years, with grandiose claims and very meager results. But it is clear that the tide has turned. The most interesting work in computer science today revolves around machine learning, both powerful new algorithms and exciting new applications.

This revolution has occurred for several reasons. First, the volume of data and computing power available crossed a magic threshold where machine learning systems started doing interesting things, even using old approaches. This inspired greater activity in developing methods that scale better, and greater investment in data resources and system development. The culture of open source software deserves to take a bow, because new ideas turn into available tools amazingly quickly. Machine learning today is an exploding field with a great deal of excitement about it.

We have so far discussed two ways of building models based on data, linear regression and nearest neighbor approaches, both in fairly extensive detail. For many applications, this is all you will need to know. If you have enough labeled training data, all methods are likely to produce good results. And if you don't, all methods are likely to fail. The impact of the best machine learning algorithm can make a difference, but generally only at the margins. I feel that the purpose of my book is to get you from crawling to walking, so that more specialized books can teach you how to run.

That said, a slew of interesting and important machine learning algorithms have been developed. We will review these methods here in this chapter, with the goal of understanding the strengths and weaknesses of each, along several relevant dimensions of performance:




S.S. Skiena, *The Data Science Design Manual,*
Texts in Computer Science, https://doi.org/10.1007/978-3-319-55444-0_11

- *Power and expressibility*: Machine learning methods differ in the richness and complexity of the models they support. Linear regression fits linear functions, while nearest neighbor methods define piecewise-linear separation boundaries with enough pieces to approximate arbitrary curves. Greater expressive power provides the possibility of more accurate models, as well as the dangers of overfitting.

- *Interpretability*: Powerful methods like deep learning often produce models that are completely impenetrable. They might provide very accurate classification in practice, but no human-readable explanation of why they are making the decisions they do. In contrast, the largest coefficients in a linear regression model identify the most powerful features, and the identities of nearest neighbors enable us to independently determine our confidence in these analogies.

  I personally believe that interpretability is an important property of a model, and am generally happier to take a lesser-performing model I understand over a slightly more accurate one that I don't. This may not be a universally shared opinion, but you have a sense whether you really understand your model and its particular application domain.

- *Ease of use*: Certain machine learning methods feature relatively few parameters or decisions, meaning they work right out of the box. Both linear regression and nearest neighbor classification are quite simple in this regard. In contrast, methods like support vector machines (SVMs) provide much greater scope to optimize algorithm performance with the proper settings. My sense is that the available tools for machine learning will continue to get better: easier to use and more powerful. But for now, certain methods allow the user enough rope to hang themselves if they don't know what they are doing.

- *Training speed*: Methods differ greatly in how fast they fit the necessary parameters of the model, which determines how much training data you can afford to use in practice. Traditional linear regression methods can be expensive to fit for large models. In contrast, nearest neighbor search requires almost no training time at all, outside that of building the appropriate search data structure.

- *Prediction speed*: Methods differ in how fast they make classification decisions on a new query $q$. Linear/logistic regression is fast, just computing a weighted sum of the fields in the input records. In contrast, nearest neighbor search requires explicitly testing $q$ against a substantial amount of the training test. In general there is a trade-off with training speed: you can pay me now or pay me later.

Figure 11.1 presents my subjective ratings of roughly where the approaches discussed in this chapter fit along these performance dimensions. These ratings are not the voice of G-d, and reasonable people can have different opinions.

| Method | Power of Expression | Ease of Interpretation | Ease of Use | Training Speed | Prediction Speed |
|---|---|---|---|---|---|
| Linear Regression | 5 | 9 | 9 | 9 | 9 |
| Nearest Neighbor | 5 | 9 | 8 | 10 | 2 |
| Naive Bayes | 4 | 8 | 7 | 9 | 8 |
| Decision Trees | 8 | 8 | 7 | 7 | 9 |
| Support Vector Machines | 8 | 6 | 6 | 7 | 7 |
| Boosting | 9 | 6 | 6 | 6 | 6 |
| Graphical Models | 9 | 8 | 3 | 4 | 4 |
| Deep Learning | 10 | 3 | 4 | 3 | 7 |

Figure 11.1: Subjective rankings of machine learning approaches along five dimensions, on a 1 to 10 scale with higher being better.

Hopefully they survey the landscape of machine learning algorithms in a useful manner. Certainly no single machine learning method dominates all the others. This observation is formalized in the appropriately named *no free lunch theorem*, which proves there does not exist a single machine learning algorithm better than all the others on all problems.

That said, it is still possible to rank methods according to priority of use for practitioners. My ordering of methods in this book (and Figure 11.1) starts with the ones that are easy to use/tune, but have lower discriminative power than the most advanced methods. Generally speaking, I encourage you to start with the easy methods and work your way down the list if the potential improvements in accuracy really justify it.

It is easy to misuse the material I will present in this chapter, because there is a natural temptation to try all possible machine learning algorithms and pick whichever model gives the highest reported accuracy or $F1$ score. Done naively through a single library call, which makes this easy, you are likely to discover that all models do about the same on your training data. Further, any performance differences between them that you do find are more likely attributable to variance than insight. Experiments like this is what statistical significance testing was invented for.

The most important factor that will determine the quality of your models is the quality of your features. We talked a lot about data cleaning in Chapter 3, which concerns the proper preparation of your data matrix. We delve more deeply into feature engineering in Section 11.5.4, before discussing deep learning methods that strive to engineer their own features.

One final comment. Data scientists tend to have a favorite machine learning approach, which they advocate for in a similar manner to their favorite programming language or sports team. A large part of this is experience, meaning that because they are most familiar with a particular implementation it works best in their hands. But part of it is magical thinking, the fact that they noticed one library slightly outperforming others on a few examples and inappropriately generalized.

Don't fall into this trap. Select methods which best fit the needs of your application based on the criteria above, and gain enough experience with their various knobs and levers to optimize performance.

## 11.1 Naive Bayes

Recall that two events $A$ and $B$ are *independent* if $p(A \text{ and } B) = p(A) \cdot p(B)$. If $A$ is the event that "my favorite sports team wins today" and $B$ is "the stock market goes up today," then presumably $A$ and $B$ are independent. But this is not true in general. Consider the case if $A$ is the event that "I get an A in Data Science this semester" and $B$ is "I get an A in a different course this semester." There are dependencies between these events: renewed enthusiasms for either study or drinking will affect course performance in a correlated manner. In the general case,

$$p(A \text{ and } B) = p(A) \cdot p(B|A) = p(A) + P(B) - p(A \text{ or } B).$$

If everything was independent, the world of probability would be a much simpler place. The naive Bayes classification algorithm crosses its fingers and assumes independence, to avoid the need to compute these messy conditional probabilities.

### 11.1.1 Formulation

Suppose we wish to classify the vector $X = (x_1, \ldots x_n)$ into one of $m$ classes $C_1, \ldots, C_m$. We seek to compute the probability of each possible class given $X$, so we can assign $X$ the label of the class with highest probability. By Bayes theorem,

$$p(C_i|X) = \frac{p(C_i) \cdot p(X|C_i)}{p(X)}$$

Let's parse this equation. The term $p(C_i)$ is the *prior* probability, the probability of the class label without any specific evidence. I know that you the reader are more likely to have black hair than red hair, because more people in the world have black hair than red hair.[1]

The denominator $P(X)$ gives the probability of seeing the given input vector $X$ over all possible input vectors. Establishing the exact value of $P(X)$ seems somewhat dicey, but mercifully is usually unnecessary. Observe that this denominator is the same for all classes. We only seek to establish a class label for $X$, so the value of $p(X)$ has no effect on our decision. Selecting the class with highest probability means

$$C(X) = \arg\max_{i=1,\ldots,m} \frac{p(C_i) \cdot p(X|C_i)}{p(X)} = \arg\max_{i=1,\ldots,m} p(C_i) \cdot p(X|C_i).$$

[1] Wikipedia claims that only 1–2% of the world's population are redheads.

| Day | Outlook | Temp | Humidity | Beach? |
|---|---|---|---|---|
| 1 | Sunny | High | High | Yes |
| 2 | Sunny | High | Normal | Yes |
| 3 | Sunny | Low | Normal | No |
| 4 | Sunny | Mild | High | Yes |
| 5 | Rain | Mild | Normal | No |
| 6 | Rain | High | High | No |
| 7 | Rain | Low | Normal | No |
| 8 | Cloudy | High | High | No |
| 9 | Cloudy | High | Normal | Yes |
| 10 | Cloudy | Mild | Normal | No |

| P(X\|Class) | Probability in Class | |
|---|---|---|
| Outlook | Beach | No Beach |
| Sunny | 3/4 | 1/6 |
| Rain | 0/4 | 3/6 |
| Cloudy | 1/4 | 2/6 |
| Temperature | Beach | No Beach |
| High | 3/4 | 2/6 |
| Mild | 1/4 | 2/6 |
| Low | 0/4 | 2/6 |
| Humidity | Beach | No Beach |
| High | 2/4 | 2/6 |
| Normal | 2/4 | 4/6 |
| P(Beach Day) | 4/10 | 6/10 |

Figure 11.2: Probabilities to support a naive Bayes calculation on whether today is a good day to go to the beach: tabulated events (left) with marginal probability distributions (right).

The remaining term $p(X|C_i)$ is the probability of seeing input vector $X$ given that we know the class of the item is $C_i$. This also seems somewhat dicey. What is the probability someone weighs 150 lbs and is 5 foot 8 inches tall, given that they are male? It should be clear that $p(X|C_i)$ will generally be very small: there is a huge space of possible input vectors consistent with the class, only one of which corresponds to the given item.

But now suppose we lived where everything was independent, i.e. the probability of event $A$ *and* event $B$ was always $p(A) \cdot p(B)$. Then

$$p(X|C_i) = \prod_{j=1}^{n} p(x_j|C_i).$$

Now anyone who really believes in a world of independent probabilities is quite naive, hence the name *naive Bayes*. But such an assumption really does make the computations much easier. Putting this together:

$$C(X) = \arg\max_{i=1,\ldots,m} p(C_i) \cdot p(X|C_i) = \arg\max_{i=1,\ldots,m} p(C_i) \prod_{j=1}^{n} p(x_j|C_i).$$

Finally, we should hit the product with a log to turn it to a sum, for better numerical stability. The logs of probabilities will be negative numbers, but less likely events are more negative than common ones. Thus the complete naive Bayes algorithm is given by the following formula:

$$C(X) = \arg\max_{i=1,\ldots,m} (\log(p(C_i)) + \sum_{j=1}^{n} \log(p(x_j|C_i))).$$

How do we calculate the $p(x_j|C_i)$, the probability of observation $x_j$ given class label $i$? This is easy from the training data, particularly if $x_j$ is a categorical variable, like "has red hair." We can simply select all class $i$ instances

in the training set, and compute the fraction of them which have property $x_j$. This fraction defines a reasonable estimate of $p(x_j|C_i)$. A bit more imagination is needed when $x_j$ is a numerical variable, like "age=18" or "the word *dog* occurred six times in the given document," but in principle is computed by how often this value is observed in the training set.

Figure 11.2 illustrates the naive Bayes procedure. On the left, it presents a table of ten observations of weather conditions, and whether each observation proved to be a day to go to the beach, or instead stay home. This table has been broken down on the right, to produce conditional probabilities of the weather condition given the activity. From these probabilities, we can use Bayes theorem to compute:

$$\begin{aligned} &P(\text{Beach}|(\text{Sunny,Mild,High})) \\ &\quad = (P(\text{Sunny}|\text{Beach}) \times P(\text{Mild}|\text{Beach}) \times P(\text{High}|\text{Beach}) \times P(\text{Beach}) \\ &\quad = (3/4) \times (1/4) \times (2/4) \times (4/10) = 0.0375 \end{aligned}$$

$$\begin{aligned} &P(\text{No Beach}|(\text{Sunny,Mild,High})) \\ &\quad = (P(\text{Sunny}|\text{No}) \times P(\text{Mild}|\text{No}) \times P(\text{High}|\text{No})) \times P(\text{No}) \\ &\quad = (1/6) \times (2/6) \times (2/6) \times (6/10) = 0.0111 \end{aligned}$$

Since $0.0375 > 0.0111$, naive Bayes is telling us to hit the beach. Note that it is irrelevant that this particular combination of (Sunny,Mild,High) appeared in the training data. We are basing our decision on the aggregate probabilities, not a single row as in nearest neighbor classification.

### 11.1.2 Dealing with Zero Counts (Discounting)

There is a subtle but important feature preparation issue particularly associated with the naive Bayes algorithm. Observed counts do not accurately capture the frequency of rare events, for which there is typically a long tail.

The issue was first raised by the mathematician Laplace, who asked: What is the probability the sun will rise tomorrow? It may be close to one, but it ain't exactly 1.0. Although the sun has risen like clockwork each morning for the 36.5 million mornings or so since man started noticing such things, it will not do so forever. The time will come where the earth or sun explodes, and so there is a small but non-zero chance that tonight's the night.

There can always be events which have not yet been seen in any finite data set. You might well have records on a hundred people, none of whom happen to have red hair. Concluding that the probability of red hair is $0/100 = 0$ is potentially disastrous when we are asked to classify someone *with* red hair, since the probability of them being in each and every class will be zero. Even worse would be if there was exactly one redhead in the entire training set, say labeled

with class $C_2$. Our naive Bayes classifier would decide that every future redhead just had to be in class $C_2$, regardless of other evidence.

*Discounting* is a statistical technique to adjust counts for yet-unseen events, by explicitly leaving probability mass available for them. The simplest and most popular technique is *add-one discounting*, where we add one to the frequency all outcomes, including unseen. For example, suppose we were drawing balls from an urn. After seeing five reds and three greens, what is the probability we will see a new color on the next draw? If we employ add-one discounting,

$$P(\text{red}) = (5+1)/((5+1)+(3+1)+(0+1)) = 6/11, \text{ and}$$

$$P(\text{green}) = (3+1)/((5+1)+(3+1)+(0+1)) = 4/11,$$

leaving the new color a probability mass of

$$P(\text{new-color}) = 1/((5+1)+(3+1)+(0+1)) = 1/11.$$

For small numbers of samples or large numbers of known classes, the discounting causes a non-trivial damping of the probabilities. Our estimate for the probability of seeing a red ball changes from $5/8 = 0.625$ to $6/11 = 0.545$ when we employ add-one discounting. But this is a safer and more honest estimate, and the differences will disappear into nothingness after we have seen enough samples.

You should be aware that other discounting methods have been developed, and adding one might not be the best possible estimator in all situations. That said, *not* discounting counts is asking for trouble, and no one will be fired for using the add-one method.

Discounting becomes particularly important in natural language processing, where the traditional *bag of words* representation models a document as a word frequency count vector over the language's entire vocabulary, say 100,000 words. Because word usage frequency is governed by a power law (Zipf's law), words in the tail are quite rare. Have you ever seen the English word *defenestrate* before?[2] Even worse, documents of less than book length are too short to contain 100,000 words, so we are doomed to see zeros wherever we look. Add-one discounting turns these count vectors into sensible probability vectors, with non-zero probabilities of seeing rare and so far unencountered words.

## 11.2 Decision Tree Classifiers

A *decision tree* is a binary branching structure used to classify an arbitrary input vector $X$. Each node in the tree contains a simple feature comparison against some field $x_i \in X$, like "is $x_i \geq 23.7$?" The result of each such comparison is either true or false, determining whether we should proceed along to the left or right child of the given node. These structures are sometimes called *classification and regression trees* (CART) because they can be applied to a broader class of problems.

[2]It means to throw someone out the window.

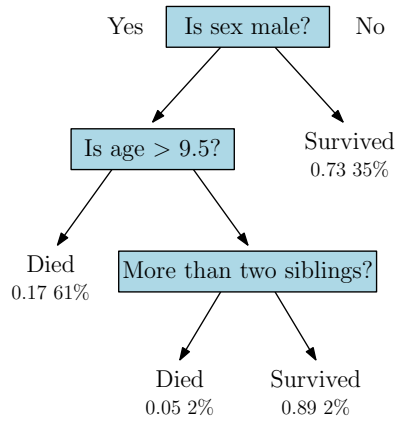


Figure 11.3: Simple decision tree for predicting mortality on the Titanic.

The decision tree partitions training examples into groups of relatively uniform class composition, so the decision then becomes easy. Figure 11.3 presents an example of a decision tree, designed to predict your chances of surviving the shipwreck of the Titanic. Each row/instance travels a unique root-to-leaf path to classification. The root test here reflects the naval tradition of women and children first: 73% of the women survived, so this feature alone is enough to make a prediction for women. The second level of the tree reflects children first: any male 10 years or older is deemed out of luck. Even the younger ones must pass one final hurdle: they generally made it to a lifeboat only if they had brothers and sisters to lobby for them.

What is the accuracy of this model on the training data? It depends upon what faction of the examples end on each leaf, and how pure these leaf samples are. For the example of Figure 11.3, augmented with coverage percentage and survival fraction (purity) at each node, the classification accuracy $A$ of this tree is:

$$A = (0.35)(73\%) + (0.61)(83\%) + (0.02)(95\%) + (0.02)(89\%) = 78.86\%.$$

An accuracy of 78.86% is not bad for such a simple decision procedure. We could have driven it up to 100% by completing the tree so each of the 1317 passengers had a leaf to themselves, labeling that node with their ultimate fate. Perhaps 23-year-old second-class males were more likely to survive than either 22- or 24-year-old males, an observation the tree could leverage for higher training accuracy. But such a complicated tree would be wildly overfit, finding structure that isn't meaningfully there. The tree in Figure 11.3 is interpretable, robust, and reasonably accurate. Beyond that, it is every man for himself.

Advantages of decision trees include:

- *Non-linearity*: Each leaf represents a chunk of the decision space, but reached through a potentially complicated path. This chain of logic permits decision trees to represent highly complicated decision boundaries.

- *Support for categorical variables*: Decision trees make natural use of categorical variables, like "if hair color = red," in addition to numerical data. Categorical variables fit less comfortably into most other machine learning methods.

- *Interpretability*: Decision trees are explainable; you can read them and understand what their reasoning is. Thus decision tree algorithms can tell you something about your data set that you might not have seen before. Also, interpretability lets you vet whether you trust the decisions it will make: is it making decisions for the right reasons?

- *Robustness*: The number of possible decision trees grows exponentially in the number of features and possible tests, which means that we can build as many as we wish. Constructing many random decision trees (CART) and taking the result of each as a vote for the given label increases robustness, and permits us to assess the confidence of our classification.

- *Application to regression*: The subset of items which follow a similar path down a decision tree are likely similar in properties other than just label. For each such subset, we can use linear regression to build a special prediction model for the numerical values of such leaf items. This will presumably perform better than a more general model trained over all instances.

The biggest disadvantage of decision trees is a certain lack of elegance. Learning methods like logistic regression and support vector machines use *math*. Advanced probability theory, linear algebra, higher-dimensional geometry. You know, *math*.

By contrast, decision trees are a hacker's game. There are many cool knobs to twist in the training procedure, and relatively little theory to help you twist them in the right way.

But the fact of the matter is that decision tree models work very well in practice. *Gradient boosted decision trees* (GBDTs) are currently the most frequently used machine learning method to win Kaggle competitions. We will work through this in stages. First decision trees, then boosting in the subsequent section.

### 11.2.1 Constructing Decision Trees

Decision trees are built in a top-down manner. We start from a given collection of training instances, each with $n$ features and labeled with one of $m$ classes $C_1, \ldots, C_m$. Each node in the decision tree contains a binary predicate, a logic condition derived from a given feature.

Features with a discrete set of values $v_i$ can easily be turned into binary predicates through equality testing: "is feature $x_i = v_{ij}$?" Thus there are $|v_i|$ distinct predicates associated with $x_i$. Numerical features can be turned into binary predicates with the addition of a threshold $t$: "is feature $x_i \geq t$?"

The set of potentially interesting thresholds $t$ are defined by the gaps between the observed values that $x_i$ takes on in the training set. If the complete set of observations of $x_i$ are $(10, 11, 11, 14, 20)$, the meaningful possible values for $t \in (10, 11, 14)$ or perhaps $t \in (10.5, 11.5, 17)$. Both threshold sets produce the same partitions of the observations, but using the midpoints of each gap seems sounder when generalizing to future values unseen in training.

We need a way to evaluate each predicate for how well it will contribute to partitioning the set $S$ of training examples reachable from this node. An ideal predicate $p$ would be a *pure partition* of $S$, so that the class labels are disjoint. In this dream all members of $S$ from each class $C_i$ will appear exclusively on one side of the tree, however, such purity is not usually possible. We also want predicates that produce *balanced splits* of $S$, meaning that the left subtree contains roughly as many elements from $S$ as the right subtree. Balanced splits make faster progress in classification, and also are potentially more robust. Setting the threshold $t$ to the minimum value of $x_i$ picks off a lone element from $S$, produces a perfectly pure but maximally imbalanced split.

Thus our selection criteria should reward both balance and purity, to maximize what we learn from the test. One way to measure the purity of an item subset $S$ is as the converse of disorder, or *entropy*. Let $f_i$ denote the fraction of $S$ which is of class $C_i$. Then the information theoretic entropy of $S$, $H(S)$, can be computed:

$$H(S) = -\sum_{i=1}^{m} f_i \log_2 f_i$$

The negative sign here exists to make the entire quantity positive, since the logarithm of a proper fraction is always negative.

Let's parse this formula. The purest possible contribution occurs when all elements belong to a single class, meaning $f_j = 1$ for some class $j$. The contribution of class $j$ to $H(S)$ is $1 \log_2(1) = 0$, identical to that of all other classes: $0 \cdot \log_2(0) = 0$. The most disordered version is when all $m$ classes are represented equally, meaning $f_i = 1/m$. Then $H(S) = \log_2(m)$ by the above definition. The smaller the entropy, the better the node is for classification.

The value of a potential split applied to a tree node is how much it reduces the entropy of the system. Suppose a Boolean predicate $p$ partitions $S$ into two disjoint subsets, so $S = S_1 \cup S_2$. Then the *information gain* of $p$ is defined

$$IG_p(S) = H(S) - \sum_{j=1}^{2} \frac{|S_i|}{|S|} H(S_i)$$

We seek the predicate $p'$ which maximizes this information gain, as the best splitter for $S$. This criteria implicitly prefers balanced splits since both sides of the tree are evaluated.

Alternate measures of purity have been defined and are used in practice. The *Gini impurity* is based on another quantity $(f_i(1 - f_i))$, which is zero in

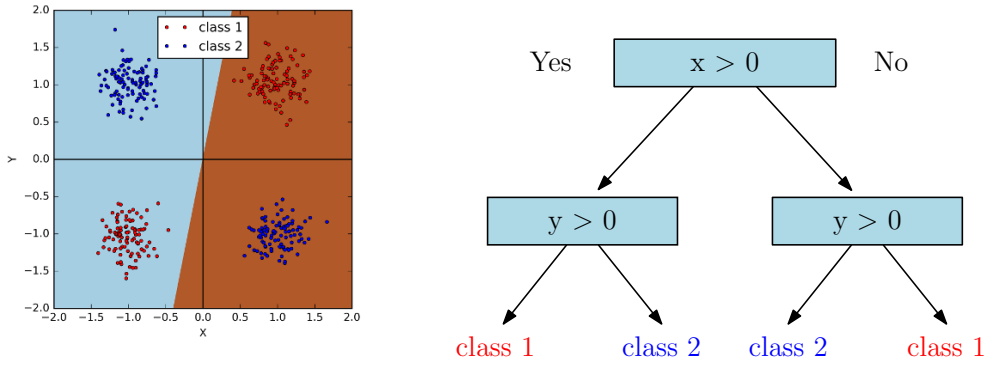


Figure 11.4: The exclusive OR function cannot be fit by linear classifiers. On the left, we present four natural clusters in $x - y$ space. This demonstrates the complete inability of logistic regression to find a meaningful separator, even though a small decision tree easily does the job (right).

both cases of pure splits, $f_i = 0$ or $f_i = 1$:

$$I_G(f) = \sum_{i=1}^{m} f_i(1 - f_i) = \sum_{i=1}^{m} (f_i - f_i^2) = \sum_{i=1}^{m} f_i - \sum_{i=1}^{m} f_i^2 = 1 - \sum_{i=1}^{m} f_i^2$$

Predicate selection criteria to optimize Gini impurity can be similarly defined.

We need a stopping condition to complete the heuristic. When is a node pure enough to call it a leaf? By setting a threshold $\epsilon$ on information gain, we stop dividing when the reward of another test is less than $\epsilon$.

An alternate strategy is to build out the full tree until all leaves are completely pure, and then prune it back by eliminating nodes which contribute the least information gain. It is fairly common that a large universe may have no good splitters near the root, but better ones emerge as the set of live items gets smaller. This approach has the benefit of not giving up too early in the process.

### 11.2.2 Realizing Exclusive Or

Some decision boundary shapes can be hard or even impossible to fit using a particular machine learning approach. Most notoriously, linear classifiers cannot be used to fit certain simple non-linear functions like eXclusive OR (XOR). The logic function $A \oplus B$ is defined as

$$A \oplus B = (A \text{ or } \bar{B}) \text{ or } (\bar{A} \text{ or } B).$$

For points $(x, y)$ in two dimensions, we can define predicates such that $A$ means "is $x \geq 0$?" and $B$ means the "is $y \geq 0$?". Then there are two distinct regions where $A \oplus B$ true, opposing quadrants in this $xy$-plane shown in Figure

11.4 (left). The need to carve up two regions with one line explains why XOR is impossible for linear classifiers.

Decision trees are powerful enough to recognize XOR. Indeed, the two-level tree in Figure 11.4 (right) does the job. After the root tests whether $A$ is true or false, the second level tests for $B$ are already conditioned on $A$, so each of the four leaves can be associated with a distinct quadrant, allowing for proper classification.

Although decision trees can recognize XOR, that doesn't mean it is easy to find the tree that does it. What makes XOR hard to deal with is that you can't see yourself making progress toward better classification, even if you pick the correct root node. In the example above, choosing a root node of "is $x > 0$?" causes no apparent enrichment of class purity on either side. The value of this test only becomes apparent if we look ahead another level, since the information gain is zero.

Greedy decision tree construction heuristics fail on problems like XOR. This suggests the value of more sophisticated and computationally expensive tree building procedures in difficult cases, which look-ahead like computer chess programs, evaluating the worth of move $p$ not now, but how it looks several moves later.

### 11.2.3 Ensembles of Decision Trees

There are an enormous number of possible decision trees which can be built on any training set $S$. Further, each of them will classify *all* training examples perfectly, if we keep refining until all leaves are pure. This suggests building hundreds or even thousands of different trees, and evaluating a query item $q$ against each of them to return a possible label. By letting each tree cast its own independent vote, we gain confidence that the most commonly seen label will be the right label.

For this to avoid group-think, we need the trees to be diverse. Repeatedly using a deterministic construction procedure that finds the best tree is worthless, because they will all be identical. Better would be to randomly select a new splitting dimension at each tree node, and then find the best possible threshold for this variable to define the predicate.

But even with random dimension selection, the resulting trees often are highly correlated. A better approach is *bagging*, building the best possible trees on relatively small random subsets of items. Done properly, the resulting trees should be relatively independent of each other, providing a diversity of classifiers to work with, facilitating the wisdom of crowds.

Using ensembles of decision trees has another advantage beyond robustness. The degree of consensus among the trees offers a measure of confidence for any classification decision. There is a big difference in the majority label appearing in 501 of 1000 trees vs. 947 of them.

This fraction can be interpreted as a probability, but even better might be to feed this number into logistic regression for a better motivated measure of confidence. Assuming we have a binary classification problem, let $f_i$ denote the

fraction of trees picking class $C_1$ on input vector $X_i$. Run the entire training set through the decision tree ensemble. Now define a logistic regression problem where $f_i$ is input variable and the class of $X_i$ the output variable. The resulting logit function will determine an appropriate confidence level, for any observed fraction of agreement.

## 11.3 Boosting and Ensemble Learning

The idea of aggregating large numbers of noisy "predictors" into one stronger classifier applies to algorithms as well as crowds. It is often the case that many different features all weakly correlate with the dependent variable. So what is the best way we can combine them into one stronger classifier?

### 11.3.1 Voting with Classifiers

*Ensemble learning* is the strategy of combining many different classifiers into one predictive unit. The naive Bayes approach of Section 11.1 has a little of this flavor, because it uses each feature as a separate relatively weak classifier, then multiplies them together. Linear/logistic regression has a similar interpretation, in that it assigns a weight to each feature to maximize the predictive power of the ensemble.

But more generally, ensemble learning revolves on the idea of *voting*. We saw that decision trees can be more powerful in aggregate, by constructing hundreds or thousands of them over random subsets of examples. The wisdom of crowds comes from the triumph of diversity of thought over the individual with greatest expertise.

Democracy rests on the principle of one man, one vote. Your educated, reasoned judgment of the best course of action counts equally as the vote of that loud-mouthed idiot down the hall. Democracy makes sense in terms of the dynamics of society: shared decisions generally affect the idiot just as much as they do you, so equality dictates that all people deserve an equal say in the matter.

But the same argument does not apply to classifiers. The most natural way to use multiple classifiers gives each a vote, and takes the majority label. But why should each classifier get the same vote?

Figure 11.5 captures some of the complexity of assigning weights to classifiers. The example consists of five voters, each classifying five items. All voters are pretty good, each getting 60% correct, with the exception of $v_1$, who batted 80%. The majority option proves no better than the worst individual classifier, however, at 60%. But a perfect classifier results if we drop voters $v_4$ and $v_5$ and weigh the remainders equally. What makes $v_2$ and $v_3$ valuable is not their overall accuracy, but their performance on the hardest problems ($D$ and especially $E$).

There seem to be three primary ways to assign weights to the classifier/voters. The simplest might be to give more weight to the votes of classifiers who have

| Item/voter | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | Majority | Best weights |
|---|---|---|---|---|---|---|---|
| A | * | | * | * | * | * | * |
| B | * | | * | * | * | * | * |
| C | * | * | | * | * | * | * |
| D | * | * | | | | | * |
| E | | * | * | | | | * |
| % correct | 80% | 60% | 60% | 60% | 60% | 60% | 100% |
| best weight | 1/3 | 1/3 | 1/3 | 0 | 0 | | |

Figure 11.5: Uniform weighting of votes does not always produce the best possible classifier, even when voters are equally accurate, because some problem instances are harder than others (here, $D$ and $E$). The "*" denotes that the given voter classified the given item correctly.

proven accurate in the past, perhaps assigning $v_i$ the multiplicative weight $t_i/T$, where $t_i$ is the number of times $v_i$ classified correctly and $T = \sum_{i=1}^{c} t_i$. Note that this weighting scheme would do no better than majority rule on the example of Figure 11.5.

A second approach could be to use linear/logistic regression to find the best possible weights. In a binary classification problem, the two classes would be denoted as 0 and 1, respectively. The 0-1 results from each classifier can be used as a feature to predict the actual class value. This formulation would find non-uniform weights that favor classifiers correlated with the correct answers, but do not explicitly seek to maximize the number of correct classifications.

### 11.3.2 Boosting Algorithms

The third idea is *boosting*. The key point is to weigh the examples according to how hard they are to get right, and reward classifiers based on the weight of the examples they get right, not just the count.

To set the weights of the classifier, we will adjust the weights of the training examples. Easy training examples will be properly classified by most classifiers: we reward classifiers more for getting the hard cases right.

A representative boosting algorithm is *AdaBoost*, presented in Figure 11.6. We will not stress the details here, particularly the specifics of the weight adjustments in each round. We presume our classifier will be constructed as the union of non-linear classifiers of the form "is $(v_i \geq t_i)$?", i.e. using thresholded features as classifiers.

The algorithm proceeds in $T$ rounds, for $t = \{0, \ldots, T\}$. Initially all training examples (points) should be of equal weight, so $w_{i,0} = 1/n$ for all points $x_1, \ldots, x_n$. We consider all possible feature/threshold classifiers, and identify the $f_i(x)$ which minimizes $\epsilon_t$, the sum of the weights of the misclassified points. The weight $\alpha_t$ of the new classifier depends upon how accurate it is on the

**AdaBoost**

For $t$ in $1 \ldots T$:

- Choose $f_t(x)$:
  - Find weak learner $h_t(x)$ that minimizes $\epsilon_t$, the weighted sum error for misclassified points $\epsilon_t = \sum_i w_{i,t}$
  - Choose $\alpha_t = \frac{1}{2} \ln \left( \frac{1-\epsilon_t}{\epsilon_t} \right)$
- Add to ensemble:
  - $F_t(x) = F_{t-1}(x) + \alpha_t h_t(x)$
- Update weights:
  - $w_{i,t+1} = (w_{i,t}) e^{-y_i \alpha_t h_t(x_i)}$ for all i
  - Renormalize $w_{i,t+1}$ such that $\sum_i w_{i,t+1} = 1$

Figure 11.6: Pseudocode for the AdaBoost algorithm.

current point set, as measured by

$$\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon_t})$$

The point weights are normalized so $\sum_{i=1}^{n} w_i = 1$, so there must always be a classifier with error $\epsilon_t \leq 0.5$.[3]

In the next round, the weights of the misclassified points are boosted to make them more important. Let $h_t(x_i)$ be the class ($-1$ or 1) predicted for $x_i$, and $y_i$ the correct class or that point. The sign of $h_t(x_i) \cdot y$ reflects whether the classes agree (positive) or disagree (negative). We then adjust the weights according to

$$w'_{i,t+1} = w_{i,t} e^{-y_i \alpha_t h_t(x_i)}$$

before re-normalizing all of them so they continue to sum to 1, i.e.

$$C = \sum_{i=1}^{n} w'_{i,t+1}, \quad \text{and} \quad w_{i,t+1} = w'_{i,t+1}/C.$$

The example in Figure 11.7 shows a final classifier as the linear sum of three thresholded single-variable classifiers. Think of them as the simplest possible

[3]Consider two classifiers, one which calls class $C_0$ if $x_i \geq t_i$, the other of which calls class $C_1$ if $x_i \geq t_i$. The first classifier is right exactly when the second one is wrong, so one of these two must be at least 50%.

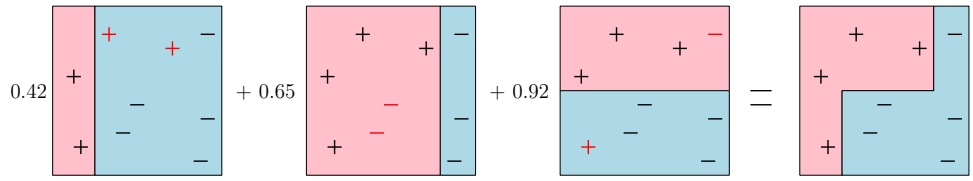


Figure 11.7: The final classifier is a weighted ensemble that correctly classifies all points, despite errors in each component classifier which are highlighted in red.

decision trees, each with exactly one node. The weights assigned by AdaBoost are not uniform, but not so crazily skewed in this particular instance that they behave different than a majority classifier. Observe the non-linear decision boundary, resulting from the discrete nature of thresholded tests/decision trees.

Boosting is particularly valuable when applied to decision trees as the elementary classifiers. The popular *gradient boosted decision trees* (GBDT) approach typically starts with a universe of small trees, with perhaps four to ten nodes each. Such trees each encode a simple-enough logic that they do not overfit the data. The relative weights assigned to each of these trees follows from a training procedure, which tries to fit the errors from the previous rounds (residuals) and increases the weights of the trees that correctly classified the harder examples.

Boosting works hard to classify every training instance correctly, meaning it works particularly hard to classify the most difficult instances. There is an adage that "hard cases make bad law," suggesting that difficult-to-decide cases make poor precedents for subsequent analysis. This is an important argument against boosting, because the method would seem prone to overfitting, although it generally performs well in practice.

The danger of overfitting is particularly severe when the training data is not a perfect gold standard. Human class annotations are often subjective and inconsistent, leading boosting to amplify the noise at the expense of the signal. The best boosting algorithms will deal with overfitting though regularization. The goal will be to minimize the number of non-zero coefficients, and avoid large coefficients that place too much faith in any one classifier in the ensemble.

*Take-Home Lesson*: Boosting can take advantage of weak classifiers in an effective way. However, it can behave in particularly pathological ways when a fraction of your training examples are incorrectly annotated.

## 11.4 Support Vector Machines

*Support vector machines* (SVMs) are an important way of building non-linear classifiers. They can be viewed as a relative of logistic regression, which sought

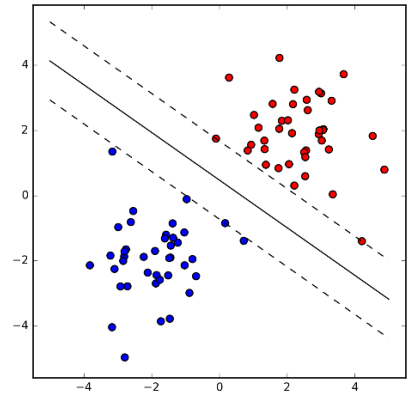


Figure 11.8: SVMs seek to separate the two classes by the largest margin, creating a channel around the separating line.

the line/plane $l$ best separating points with two classes of labels. Logistic regression assigned a query point $q$ its class label depending upon whether $q$ lay above or below this line $l$. Further, it used the logit function to transform the distance from $q$ to $l$ into the probability that $q$ belongs in the identified class.

The optimization consideration in logistic regression involved minimizing the sum of the misclassification probabilities over all the points. By contrast, support vector machines work by seeking maximum margin *linear* separators between the two classes. Figure 11.8(left) shows red and blue points separated by a line. This line seeks to maximize the distance $d$ to the nearest training point, the maximum margin of separation between red and blue. This a natural objective in building a decision boundary between two classes, since the larger the margin, the farther any of our training points are from being misclassified. The maximum margin classifier should be the most robust separator between the two classes.

There are several properties that help define the maximum margin separator between sets of red and blue points:

- The optimal line must be in the midpoint of the channel, a distance $d$ away from both the nearest red point and the nearest blue point. If this were not so, we could shift the line over until it did bisect this channel, thus enlarging the margin in the process.
- The actual separating channel is defined by its contact with a small number of the red and blue points, where "a small number" means at most twice the number of dimensions of the points, for well-behaved point sets avoiding $d + 1$ points lying on any $d$-dimensional face. This is different than with logistic regression, where *all* the points contribute to fitting the

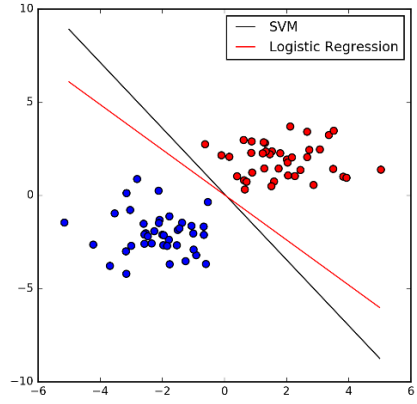


Figure 11.9: Both logistic regression and SVMs produce separating lines between point sets, but optimized for different criteria.

best position of the line. These contact points are the *support vectors* defining the channel.

- Points inside the convex hull of either the red or blue points have absolutely no effect on the maximum margin separator, since we need all same-colored points to be on the same side of the boundary. We could delete these interior points or move them around, but the maximum margin separator will not change until one of the points leaves the hull and enters the separating strip.

- It is not always possible to perfectly separate red from blue by using a straight line. Imagine a blue point sitting somewhere within the convex hull of the red points. There is no way to carve this blue point away from the red using only a line.

Logistic regression and support vector machines both produce separating lines between point sets. These are optimized for different criteria, and hence can be different, as shown in Figure 11.9. Logistic regression seeks the separator which maximizes the total confidence in our classification summed over all the points, while the wide margin separator of SVM does the best it can with the closest points between the sets. Both methods generally produce similar classifiers.

### 11.4.1 Linear SVMs

These properties define the optimization of *linear support vector machines*. The separating line/plane, like any other line/plane, can be written as

$$w \cdot x - b = 0$$

for a vector of coefficients $w$ dotted with a vector of input variables $x$. The channel separating the two classes will be defined by two lines parallel to this and equidistant on both sides, namely $w \cdot x - b = 1$ and $w \cdot x - b = 1$.

The actual geometric separation between the lines depends upon $w$, namely $2/||w||$. For intuition, think about the slope in two dimensions: these lines will be distance 2 apart for horizontal lines but negligibly far apart if they are nearly vertical. This separating channel must be devoid of points, and indeed separate red from blue points. Thus we must add constraints. For every red (class 1) point $x_i$, we insist that

$$w \cdot x - b \geq 1,$$

while every blue (class $-1$) point $x_i$ must satisfy

$$w \cdot x - b \leq -1,$$

If we let $y_i \in [-1, 1]$ denote the class of $x_i$, then these can be combined to yield the optimization problem

$$\max ||w||, \quad \text{where} \quad y_i(w \cdot x_i - b) \geq 1 \quad \text{for all} \quad 1 \leq i \leq n.$$

This can be solved using techniques akin to linear programming. Note that the channel must be defined by the points making contact with its boundaries. These vectors "support" the channel, which is where the provocative name *support vector machines* comes from. The optimization algorithm of efficient solvers like LibLinear and LibSVM search through the relevant small subsets of support vectors which potentially define separating channels to find the widest one.

Note that there are more general optimization criteria for SVMs, which seek the line that defines a wide channel and penalizes (but does not forbid) points that are misclassified. This sort of dual-objective function (make the channel wide while misclassifying few points) can be thought of as a form of regularization, with a constant to trade off between the two objectives. Gradient descent search can be used to solve these general problems.

### 11.4.2 Non-linear SVMs

SVMs define a hyperplane which separates the points from the two classes. Planes are lines in higher dimensions, readily defined using linear algebra. So how can this linear method produce a non-linear decision boundary?

For a given point set to have a maximum margin separator, the two colors must first be linearly separable. But as we have seen this is not always the case. Consider the pathological case of Figure 11.10 (left), where the cluster of red

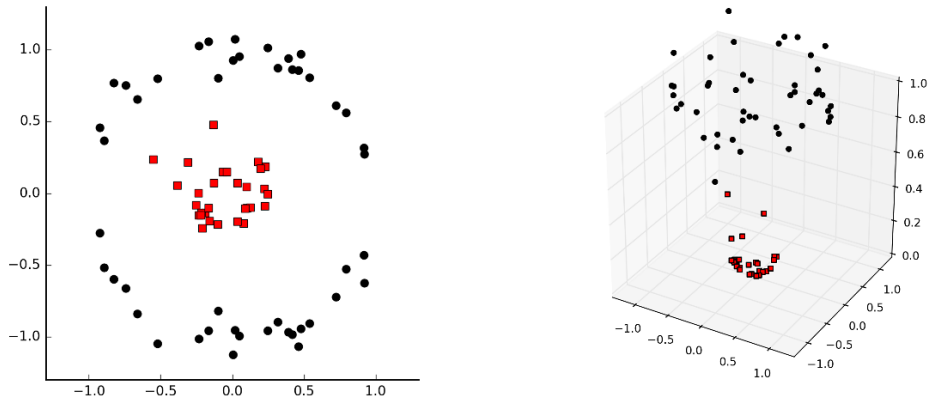


Figure 11.10: Projecting points to higher dimensions can make them linearly separable.

points is surrounded by a ring-shaped cluster of black points. How might such a thing arise? Suppose we partition travel destinations into *day trips* or *long trips*, depending upon whether they are close enough to our given location. The longitude and latitudes of each possible destination will yield data with exactly the same structure as Figure 11.10 (left).

The key idea is that we can project our $d$-dimensional points into a higher-dimensional space, where there will be more possibilities to separate them. For $n$ red/blue points along a line in one dimension, there are only $n-1$ potentially interesting ways to separate them, specifically with a cut between the $i$th and $(i+1)$st points for $1 \leq i < n$. But this blows up to approximately $\binom{n}{2}$ ways as we move to two dimensions, because there is more freedom to partition as we increase dimensionality. Figure 11.10 (right) demonstrates how lifting points through the transformation $(x, y) \rightarrow (x, y, x^2 + y^2)$ puts them on a paraboloid, and makes it possible to linearly separate classes which were inseparable in the original space.

If we jack the dimensionality of any two-class point set high enough, there will always be a separating line between the red and black points. Indeed, if we put the $n$ points in $n$ dimensions through a reasonable transform, they will always be linearly separable in a very simple way. For intuition, think about the special case of two points (one red and one blue) in two dimensions: obviously there must be a line separating them. Projecting this separating plane down to the original space results in some form of curved decision boundary, and hence the non-linearity of SVMs depends upon exactly how the input was projected to a higher-dimensional space.

One nice way to turn $n$ points in $d$ dimensions into $n$ points in $n$ dimensions

might be to represent each point by its distances to all $n$ input points. In particular, for each point $p_i$ we can create a vector $v_i$ such that $v_{ij} = dist(i, j)$, the distance from $p_i$ to $p_j$. The vector of such distances should serve as a powerful set of features for classifying any new point $q$, since the distances to members of the actual class should be small compared to those of the other class.

This feature space is indeed powerful, and one can readily imagine writing a function to turn the original $n \times d$ feature matrix into a new $n \times n$ feature matrix for classification. The problem here is space, because the number of input points $n$ is usually vastly larger than the dimension $d$ that they sit in. Such a transform would be feasible to construct only for fairly small point sets, say $n \leq 1000$. Further, working with such high-dimensional points should be very expensive, since every single distance evaluation now takes time linear in the number of points $n$, instead of the data dimension $d$. But something amazing happens...

### 11.4.3 Kernels

The magic of SVMs is that this distance-feature matrix need not *actually* be computed explicitly. In fact, the optimization inherent in finding the maximum margin separator only performs the dot products of points with other points and vectors. Thus we could imagine performing the distance expansion on the fly, when the associated point is being used in a comparison. Hence there would be no need to precompute the distance matrix: we can expand the points from $d$ to $n$ dimensions as needed, do the distance computation, and then throw the expansions away.

This would work to eliminate the space bottleneck, but we would still pay a heavy price in computation time. The really amazing thing is that there are functions, called *kernels*, which return what is essentially the distance computation on the larger vector without ever constructing the larger vector. Doing SVMs with kernels gives us the power of finding the best separator over a variety of non-linear functions without much additional cost. The mathematics moves beyond the scope of what I'd like to cover here, but:

> *Take-Home Lesson*: Kernel functions are what gives SVMs their power to separate project $d$-dimensional points to $n$ dimensions, so they can be separated *without* spending more that $d$ steps on the computation.

Support vector machines require experience to use effectively. There are many different kernel functions available, beyond the distance kernel I presented here. Each have advantages on certain data sets, so there is a need to futz with the options of tools like LibSVM to get the best performance. They work best on medium-sized data sets, with thousands but not millions of points.

## 11.5 Degrees of Supervision

There is a natural distinction between machine learning approaches based on the degree and nature of the *supervision* employed in amassing training and evaluation data. Like any taxonomy, there is some fuzziness around the margins, making it an unsatisfying exercise to try to label exactly what a given system is and is not doing. However, like any *good* taxonomy it gives you a frame to guide your thinking, and suggests approaches that might lead to better results.

The methods discussed so far in this chapter assume that we are given training data with class labels or target variables, leaving our task as one to train classifier or regression systems. But getting to the point of having labeled data is usually the hard part. Machine learning algorithms generally perform better the more data you can give them, but annotation is often difficult and expensive. Modulating the degree of supervision provides a way to raise the volume so your classifier can hear what is going on.

### 11.5.1 Supervised Learning

*Supervised learning* is the bread-and-butter paradigm for classification and regression problems. We are given vectors of features $x_i$, each with an associated class label or target value $y_i$. The annotations $y_i$ represent the supervision, typically derived from some manual process which limits the potential amount of training data.

In certain problems, the annotations of the training data come from observations in interacting with the world, or at least a simulation of it. Google's AlphaGo program was the first computer program to beat the world champion at Go. A position evaluation function is a scoring function that takes a board position and computes a number estimating how strong it is. AlphaGo's position evaluation function was trained on all published games by human masters, but much more data was needed. The solution was, essentially, to build a position evaluator by training against itself. Position evaluation is substantially enhanced by search – looking several moves ahead before calling the evaluation function on each leaf. Trying to predict the post-search score without the search produces a stronger evaluation function. And generating this training data is just a result of computation: the program playing against itself.

This idea of learning from the environment is called *reinforcement learning*. It cannot be applied everywhere, but it is always worth looking for clever approaches to generate mechanically-annotated training data.

### 11.5.2 Unsupervised Learning

Unsupervised methods try to find structure in the data, by providing labels (clusters) or values (rankings) without any trusted standard. They are best used for exploration, for making sense of a data set otherwise untouched by human hands.

The mother of all unsupervised learning methods is *clustering*, which we discussed extensively in Section 10.5. Note that clustering can be used to provide training data for classification even in the absence of labels. If we presume that the clusters found represent genuine phenomenon, we can then use the cluster ID as a label for all the elements in the given cluster. These can now serve as training data to build a classifier to predict the cluster ID. Predicting cluster IDs can be useful even if these concepts do not have a name associated with them, providing a reasonable label for any input record $q$.

### Topic Modeling

Another important class of unsupervised methods is *topic modeling*, typically associated with documents drawn over a given vocabulary. Documents are written about topics, usually a mix of topics. This book is partitioned into chapters, each of which is about a different topic, but it also touches on subjects ranging from baseball to weddings. But what is a topic? Typically each topic is associated with a particular set of vocabulary words. Articles about baseball mention *hits*, *pitchers*, *strikeouts*, *bases*, and *slugging*. *Married*, *engaged*, *groom*, *bride*, *love*, and *celebrate* are words associated with the topic of wedding. Certain words can represent multiple topics. For example *love* is also associated with tennis, and *hits* with gangsters.

Once one has a set of topics $(t_1, \ldots, t_k)$ and the words which define them, the problem of identifying the specific topics associated with any given document $d$ seems fairly straightforward. We count the number of word occurrences of $d$ in common with $t_i$, and report success whenever this is high enough. If given a set of documents manually labeled with topics, it seems reasonable to count the frequency of each word over every topic class, to construct the list of words most strongly associated with each topic.

But that is all very heavily supervised. *Topic modeling* is an unsupervised approach that infers the topics and the word lists from scratch, just given unlabeled documents. We can represent these texts by a $w \times d$ frequency matrix $F$, where $w$ is the vocabulary size and $d$ the number of documents and $F[i, j]$ reflects how many times work $i$ appears in document $j$. Suppose we factor $F$ into $F \approx W \times D$, where $W$ is a $w \times t$ word–topic matrix and $D$ is a $t \times d$ topic–document matrix. The largest entries in the $i$th row of $W$ reflect the topics word $w_i$ is most strongly linked to, while the largest entries in the $j$th column of $D$ reflect the topics best represented in document $d_j$.

Such a factorization would represent a completely unsupervised form of learning, with the exception of specifying the desired number of topics $t$. It seems a messy process to construct such an approximate factorization, but there are a variety of approaches to try to do so. Perhaps the most popular method for topic modeling is an approach called *latent Dirichlet allocation* (LDA), which produces a similar set of matrices $W$ and $D$, although not strictly produced by factorization.

Figure 11.5.2 presents a toy example of LDA in action. Three excellent books were analyzed, with the goal of seeing how they were organized among

| Text | $T_1$ | $T_2$ | $T_3$ |
|---|---|---|---|
| The Bible | 0.73 | 0.01 | 0.26 |
| Data Sci Manual | 0.05 | 0.83 | 0.12 |
| Who's Bigger? | 0.08 | 0.23 | 0.69 |

| $T_1$ | | $T_2$ | | $T_3$ | |
|---|---|---|---|---|---|
| Term | Weight | Term | Weight | Term | Weight |
| God | 0.028 | CPU | 0.021 | past | 0.013 |
| Jesus | 0.012 | computer | 0.010 | history | 0.011 |
| pray | 0.006 | data | 0.005 | old | 0.006 |
| Israel | 0.003 | program | 0.003 | war | 0.004 |
| Moses | 0.001 | math | 0.002 | book | 0.002 |

Figure 11.11: Illustration of topic modeling (LDA). The three books are represented by their distribution of topics (left). Each topic is represented by a list of words, with the weight a measure of its importance to the topic (right). Documents are made up of words: the magic of LDA is that it simultaneously infers the topics and word assignments in an unsupervised manner.

three latent topics. The LDA algorithm defined these topics in an unsupervised way, by assigning each word weights for how much it contributes to each topic. The results here are generally effective: the concept of each topic emerges from its most important words (on the right). And the word distribution within each book can then be readily partitioned among the three latent topics (on the left).

Note that this factorization mindset can be applied beyond documents, to any feature matrix $F$. The matrix decomposition approaches we have previously discussed, like singular value decomposition and principle components analysis, are equally unsupervised, inducing structure inherent in the data sets without our lead in finding it.

### 11.5.3 Semi-supervised Learning

The gap between supervised and unsupervised learning is filled by *semi-supervised learning* methods, which amplify small amounts of labeled training data into more. Turning small numbers of examples into larger numbers is often called *bootstrapping*, from the notion of "pulling yourself up from your bootstraps." Semi-supervised approaches personify the cunning which needs be deployed to build substantive training sets.

We assume that we are given a small number of labeled examples as $(x_i, y_i)$ pairs, backed by a large number of inputs $x_j$ of unknown label. Instead of directly building our model from the training set, we can use it to classify the mass of unlabeled instances. Perhaps we use a nearest neighbor approach to classify these unknowns, or any of the other approaches we have discussed here. But once we classify them, we assume the labels are correct and retrain on the larger set.

Such approaches benefit strongly from having a reliable evaluation set. We need to establish that the model trained on the bootstrapped examples performs better than one trained on what we started with. Adding billions of training examples is worthless if the labels are garbage.

There are other ways to generate training data without annotations. Often it seems easier to find positive examples than negative examples. Consider the

problem of training a grammar corrector, meaning it distinguishes proper bits of writing from ill-formed stuff. It is easy to get a hold of large amounts of proper examples of English: whatever gets published in books and newspapers generally qualifies as good. But it seems harder to get a hold of a large corpora of incorrect writing. Still, we can observe that randomly adding, deleting, or substituting arbitrary words to any text almost always makes it worse.[4] By labeling all published text as correct and all random perturbations as incorrect, we can create as large a training set as we desire without hiring someone to annotate it.

How can we evaluate such a classifier? It is usually feasible to get enough genuine annotated data for evaluation purposes, because what we need for evaluation is typically much smaller than that for training. We can also use our classifier to suggest what to annotate. The most valuable examples for the annotator to vet are those that our classifier makes mistakes on: published sentences marked incorrect or random mutations that pass the test are worth passing to a human judge.

### 11.5.4 Feature Engineering

*Feature engineering* is the fine art of applying domain knowledge to make it easier for machine learning algorithms to do their intended job. In the context of our taxonomy here, feature engineering can be considered an important part of supervised learning, where the supervision applies to the feature vectors $x_i$ instead of the associated target annotations $y_i$.

It is important to ensure that features are presented to models in a way that the model can properly use them. Incorporating application-specific knowledge into the data instead of learning it sounds like cheating, to amateurs. But the pros understand that there are things that cannot be learned easily, and hence are better explicitly put into the feature set.

Consider a model to price art at auctions. Auction houses make their money by charging a commission to the winning bidder, on top of what they pay the owner. Different houses charge different rates, but they can amount to a substantial bill. Since the total cost to the winner is split between purchase price and commission, higher commissions may well lower the purchase price, by cutting into what the bidder can afford to pay the owner.

So how can you represent the commission price in an art pricing model? I can think of at least three different approaches, some of which can have disastrous outcomes:

- *Specify the commission percentage as a feature*: Representing the house cut (say 10%) as a column in the feature set might not be usable in a linear model. The hit taken by the bidder is the product of the tax rate and the final price. It has a multiplicative effect, not an additive effect,

[4] Give it a try someplace on this page. Pick a word at random and replace it by *red*. Then again with *the*. And finally with *defenestrate*. Is my original text clearly better written than what you get after such a change?

and hence cannot be meaningfully exploited if the price range for the art spans from $100 to $1,000,000.

- *Include the actual commission paid as a feature*: Cheater... If you include the commission ultimately *paid* as a feature, you pollute the features with data not known at the time of the auction. Indeed, if all paintings were faced with a 10% tax, and the tax paid was a feature, a perfectly accurate (and completely useless) model would predict the price as ten times the tax paid!

- *Set the regression target variable to be the total amount paid*: Since the house commission rates and add-on fees are known to the buyer before they make the bid, the right target variable should be the total amount paid. Any given prediction of the total purchase price can be broken down later into the purchase price, commission, and taxes according to the rules of the house.

Feature engineering can be thought of as a domain-dependent version of data cleaning, so the techniques discussed in Section 3.3 all apply here. The most important of them will be reviewed here in context, now that we have finally reached the point of actually building data-driven models:

- *Z-scores and normalization*: Normally-distributed values over comparable numerical ranges make the best features, in general. To make the ranges comparable, turn the values into Z-scores, by subtracting off the mean and dividing by the standard deviation, $Z = (x - \mu)/\sigma$. To make a power law variable more normal, replace $x$ in the feature set with $\log x$.

- *Impute missing values*: Make sure there are no missing values in your data and, if so, replace them by a meaningful guess or estimate. Recording that someone's weight equals $-1$ is an effortless way to mess up any model. The simplest imputation method replaces each missing value by the mean of the given column, and generally suffices, but stronger methods train a model to predict the missing value based on the other variables in the record. Review Section 3.3.3 for details.

- *Dimension reduction*: Recall that *regularization* is a way of forcing models to discard irrelevant features to prevent overfitting. It is even more effective to eliminate irrelevant features before fitting your models, by removing them from the data set. When is a feature $x$ likely irrelevant for your model? Poor correlation with the target variable $y$, plus the lack of any qualitative reason you can give for why $x$ *might* impact $y$ are both excellent indicators.

  Dimension reduction techniques like singular-value decomposition are excellent ways to reduce large feature vectors to more powerful and concise representations. The benefits include faster training times, less overfitting, and noise reduction from observations.

- *Explicit incorporation of non-linear combinations*: Certain products or ratios of feature variables have natural interpretations in context. Area or volume are products of length, width, and height, yet cannot be part of any linear model unless explicitly made a column in the feature matrix. Aggregate totals, like career points scored in sports or total dollars earned in salary, are usually incomparable between items of different age or duration. But converting totals into rates (like points per game played or dollars per hour) usually make more meaningful features.

  Defining these products and ratios requires domain-specific information, and careful thought during the feature engineering process. You are much more likely to know the right combinations than your non-linear classifier is to find it on its own.

Don't be shy here. The difference between a good model and a bad model usually comes down to quality of its feature engineering. Advanced machine learning algorithms are glamorous, but it is the data preparation that produces the results.

## 11.6 Deep Learning

The machine learning algorithms we have studied here do not really scale well to *huge* data sets, for several reasons. Models like linear regression generally have relatively few parameters, say one coefficient per column, and hence cannot really benefit from enormous numbers of training examples. If the data has a good linear fit, you will be able to find it with a small data set. And if doesn't, well, you didn't really want to find it anyway.

*Deep learning* is an incredibly exciting recent development in machine learning. It is based on *neural networks*, a popular approach from the 1980s which then fell substantially out of style. But over the past five years something happened, and suddenly multi-layer (deep) networks began wildly out-performing traditional approaches on classical problems in computer vision and natural language processing.

Exactly why this happened remains somewhat of a mystery. It doesn't seem that there was a fundamental algorithmic breakthrough so much as that data volume and computational speeds crossed a threshold where the ability to exploit enormous amounts of training data overcame methods more effective at dealing with a scarce resource. But infrastructure is rapidly developing to leverage this advantage: new open source software frameworks like Google's *TensorFlow* make it easy to specify network architectures to special-purpose processors designed to speed training by orders of magnitude.

What distinguishes deep learning from other approaches is that it generally avoids feature engineering. Each layer in a neural network generally accepts as its input the output of its previous layer, yielding progressively higher-level features as we move up towards the top of the network. This serves to define a hierarchy of understanding from the raw input to the final result, and indeed

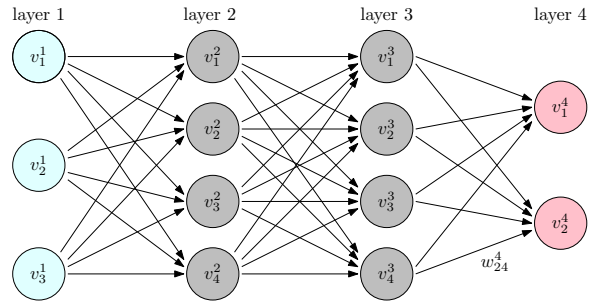


Figure 11.12: Deep learning networks have hidden layers of parameters.

the penultimate level of a network designed for one task often provides useful high-level features for related tasks.

Why are neural networks so successful? Nobody really knows. There are indications that for many tasks the full weight of these networks are not really needed; that what they are doing will eventually be done using less opaque methods. Neural networks seem to work by overfitting, finding a way to use millions of examples to fit millions of parameters. Yet they generally manage to avoid the worst behavior of overfitting, perhaps by using less precise ways to encode knowledge. A system explicitly memorizing long strings of text to split out on demand will seem brittle and overfit, while one representing such phrases in a looser way is liable to be more flexible and generalizable.

This is a field which is advancing rapidly, enough so that I want to keep my treatment strictly at the idea level. What are the key properties of these networks? Why have they suddenly become so successful?

*Take-Home Lesson*: Deep learning is a very exciting technology that has legs, although it is best suited for domains with enormous amounts of training data. Thus most data science models will continue to be built using the traditional classification and regression algorithms that we detailed earlier in this chapter.

### 11.6.1 Networks and Depth

Figure 11.12 illustrates the architecture of a deep learning network. Each node $x$ represents a computational unit, which computes the value of a given simple function $f(x)$ over all inputs to it. For now, perhaps view it as a simple adder that adds all the inputs, then outputs the sum. Each directed edge $(x, y)$ connects the output of node $x$ to the input of a node $y$ higher in the network. Further, each such edge has an associated multiplier coefficient $w_{x,y}$. The value actually passed to $y$ is the $w_{x,y} \cdot f(x)$, meaning node $y$ computes a weighted sum of its inputs.

The left column of Figure 11.12 represents a set of input variables, the values of which change whenever we ask the network to make a prediction. Think of this

as the interface to the network. Links from here to the next level propagate out this input value to all the nodes which will compute with it. On the right side are one or more output variables, presenting the final results of this computation. Between these input and output layers sit *hidden layers* of nodes. Given the weights of all the coefficients, the network structure, and the values of input variables, the computation is straightforward: compute the values of the lowest level in the network, propagate them forward, and repeat from the next level until you hit the top.

*Learning* the network means setting the weights of the coefficient parameters $w_{x,y}$. The more edges there are, the more parameters we have to learn. In principle, learning means analyzing a training corpus of $(x_i, y_i)$ pairs, and adjusting weights of the edge parameters so that the output nodes generate something close to $y_i$ when fed input $x_i$.

### Network Depth

The depth of the network should, in some sense, correspond to the conceptual hierarchy associated with the objects being modeled. The image we should have is the input being successively transformed, filtered, boiled down, and banged into better and better shape as we move up the network. Generally speaking, the number of nodes should progressively decrease as we move up to higher layers.

We can think of each layer as providing a level of abstraction. Consider a classification problem over images, perhaps deciding whether the image contains a picture of a cat or not. Thinking in terms of successive levels of abstraction, images can be said to be made from pixels, neighborhood patches, edges, textures, regions, simple objects, compound objects, and scenes. This is an argument that at least eight levels of abstraction could potentially be recognizable and usable by networks on images. Similar hierarchies exist in document understanding (characters, words, phrases, sentences, paragraphs, sections, documents) and any other artifacts of similar complexity.

Indeed, deep learning networks trained for specific tasks can produce valuable general-purpose features, by exposing the outputs of lower levels in the network as powerful features for conventional classifiers. For example, *Imagenet* is a popular network for object recognition from images. One high-level layer of 1000 nodes measures the confidence that the image contains objects of each of 1000 different types. The patterns of what objects light up to what degree are generally useful for other tasks, such as measuring image similarity.

We do not impose any real vision of what each of these levels should represent, only to connect them so that the potential to recognize such complexity exists. Neighborhood patches are functions of small groups of connected pixels, while regions will be made up of small numbers of connected patches. Some sense of what we are trying to recognize goes into designing this topology, but the network does what it feels it has to do during training to minimize training error, or *loss*.

The disadvantages of deeper networks is that they become harder to train

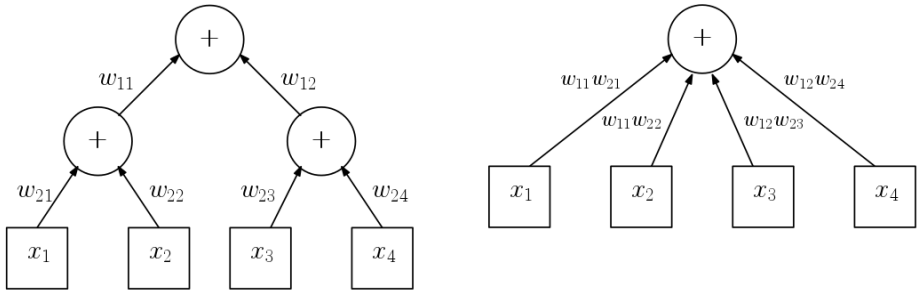


Figure 11.13: Addition networks do not benefit from depth. The two layer network (left) computes exactly the function as the equivalent one layer network (right).

the larger and deeper they get. Each new layer adds a fresh set of edge-weight parameters, increasing the risks of overfitting. Properly ascribing the effect of prediction errors to edge-weights becomes increasingly difficult, as the number of intervening layers grows between the edge and the observed result. However, networks with over ten layers and millions of parameters have been successfully trained and, generally speaking, recognition performance increases with the complexity of the network.

Networks also get more computationally expensive to make predictions as the depth increases, since the computation takes time linear in the number of edges in the network. This is not terrible, especially since all the nodes on any given level can be evaluated in parallel on multiple cores to reduce the prediction time. Training time is where the real computational bottlenecks generally exist.

### Non-linearity

The image of recognizing increasing levels of abstraction up the hidden layers of a network is certainly a compelling one. It is fair to ask if it is real, however. Do extra layers in a network *really* give us additional computational power to do things we can't with less?

The example of Figure 11.13 seems to argue the converse. It shows addition networks built with two and three layers of nodes, respectively, but both compute exactly the same function on all inputs. This suggests that the extra layer was unnecessary, except perhaps to reduce the engineering constraint of node degree, the number of edges entering as input.

What it really shows is that we need more complicated, non-linear node *activation* functions $\phi(v)$ to take advantage of depth. Non-linear functions cannot be composed in the same way that addition can be composed to yield addition. This nonlinear activation function $\phi(v_i)$ typically operates on a weighted sum

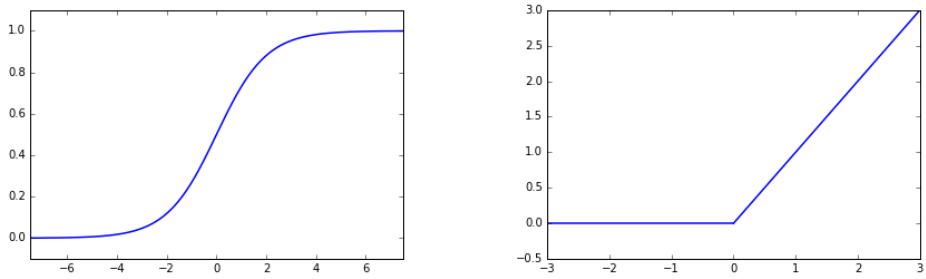


Figure 11.14: The logistic (left) and ReLU (right) activation functions for nodes in neural networks.

of the inputs $x$, where

$$v_i = \beta + \sum_i w_i x_i.$$

Here $\beta$ is a constant for the given node, perhaps to be learned in training. It is called the *bias* of the node because it defines the activation in the absence of other inputs.

That computing the output values of layer $l$ involves applying the activation function $\phi$ to weighted sums of the values from layer $l-1$ has an important implication on performance. In particular, neural network evaluation basically just involves one matrix multiplication per level, where the weighted sums are obtained by multiplying an $|V_l| \times |V_{l-1}|$ weight matrix $W$ by an $|V_{l-1}| \times 1$ output vector $V_{l-1}$. Each element of the resulting $|V_l| \times 1$ vector is then hit with the $\phi$ function to prepare the output values for that layer. Fast libraries for matrix multiplication can perform the heart of this evaluation very efficiently.

A suite of interesting, non-linear activation functions have been deployed in building networks. Two of the most prominent, shown in Figure 11.14, include:

- *Logit*: We have previously encountered the *logistic function* or logit, in our discussion of logistic regression for classification. Here

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

  This unit has the property that the output is constrained to the range [0,1], where $f(0) = 1/2$. Further, the function is differentiable, so back-propagation can be used to train the resulting network.

- *Rectified linear units (ReLU)*: A *rectifier* or diode in an electrical circuit lets current flow in only one direction. Its response function $f(x)$ is linear when $x$ is positive, but zero when $x$ is negative, as shown in Figure 11.14

(right).

$$f(x) = x \text{ when } x \geq 0$$
$$= 0 \text{ when } x < 0$$

This kink at $x = 0$ is enough to remove the linearity from the function, and provides a natural way to turn off the unit by driving it negative. The ReLU function remains differentiable, but has quite a different response than the logit, increasing monotonically and being unbounded on one side.

I am not really aware of a theory explaining why certain functions should perform better in certain contexts. Specific activation functions presumably became popular because they worked well in experiments, with the choice of unit being something you can change if you don't feel your network is performing as well as it should.

Generally speaking, adding one hidden layer adds considerable power to the network, with additional layers suffering from diminishing returns. The theory shows that networks without any hidden layers have the power to recognize linearly separable classes, but we turned to neural nets to build more powerful classifiers.

*Take-Home Lesson*: Start with one hidden layer with a number of nodes between the size of the input and output layers, so they are forced to learn compressed representations that make for powerful features.

### 11.6.2 Backpropagation

*Backpropagation* is the primary training procedure for neural networks, which achieves very impressive results by fitting large numbers of parameters incrementally on large training sets. It is quite reminiscent of stochastic gradient descent, which we introduced in Section 9.4.

Our basic problem is this. We are given a neural network with preliminary values for each parameter $w_{ij}^l$, meaning the multiplier that the output of node $v_j^{l-1}$ gets before being added to node $v_i^l$. We are also given a training set consisting of $n$ input vector-output value pairs $(x_a, y_a)$, where $1 \leq a \leq n$. In our network model, the vector $x_i$ represents the values to be assigned to the input layer $v^1$, and $y_i$ the desired response from the output layer $v_l$. Evaluating the current network on $x_i$ will result in an output vector $v_l$. The error $E_l$ of the network at layer $l$ can be measured, perhaps as

$$E_l = ||y_i - v^l||^2 = \sum_j (\phi(\beta + \sum_j w_{ij}^l v_{ij}^{l-1}) - y_{ij})^2$$

We would like to improve the values of the weight coefficients $w_{ij}^l$ so they better predict $y_i$ and minimize $E_l$. This equation above defines the loss $E_l$ as a function of the weight coefficients, since the input values from the previous

| Source | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Apple | iPhone | iPad | apple | MacBook | iPod |
| apple | apples | blackberry | Apple | iphone | fruit |
| car | cars | vehicle | automobile | truck | Car |
| chess | Chess | backgammon | mahjong | checkers | tournaments |
| dentist | dentists | dental | orthodontist | dentistry | Dentist |
| dog | dogs | puppy | pet | cat | puppies |
| Mexico | Puerto | Peru | Guatemala | Colombia | Argentina |
| red | blue | yellow | purple | orange | pink |
| running | run | ran | runs | runing | start |
| write | writing | read | written | tell | Write |

Figure 11.15: Nearest neighbors in word embeddings capture terms with similar roles and meaning.

layer $v^{l-1}$ is fixed. As in stochastic gradient descent, the current value of the $w_{ij}^l$ defines a point $p$ on this error surface, and the derivative of $E_l$ at this point defines the direction of steepest descent reducing the errors. Walking down a distance $d$ in this direction defined by the current step size or *learning rate* yields updated values of the coefficients, whose $v_l$ does a better job predicting $y_a$ from $x_a$.

But this only changes coefficients in the output layer. To move down to the previous layer, note that the previous evaluation of the network provided an output for each of these nodes as a function of the input. To repeat the same training procedure, we need a target value for each node in layer $l-1$ to play the role of $y_a$ from our training example. Given $y_a$ and the new weights to compute $v^l$, we can compute values for the outputs of these layers which would perfectly predict $y_i$. With these targets, we can modify the coefficient weights at this level, and keep propagating backwards until we hit the bottom of the network, at the input layer.

### 11.6.3 Word and Graph Embeddings

There is one particular unsupervised application of deep learning technology that I have found readily applicable to several problems of interest. This has the extra benefit of being accessible to a broader audience with no familiarity with neural networks. *Word embeddings* are distributed representations of what words actually *mean* or *do*.

Each word is denoted by a single point in, say, 100-dimensional space, so that words which play similar roles tend to be represented by nearby points. Figure 11.15 presents the five nearest neighbors of several characteristic English words according to the *GloVe* word embedding [PSM14], and I trust you will agree that they capture an amazing amount of each word's meaning by association.

The primary value of word embeddings is as general features to apply in

specific machine learning applications. Let's reconsider the problem of distinguishing spam from meaningful email messages. In the traditional bag of words representation, each message might be represented as a sparse vector $b$, where $b[i]$ might report the number of times vocabulary word $w_i$ appears in the message. A reasonable vocabulary size $v$ for English is 100,000 words, turning $b$ into a ghastly 100,000-dimensional representation that does not capture the similarity between related terms. Word vector representations prove much less brittle, because of the lower dimensionality.

We have seen how algorithms like singular value decomposition (SVD) or principle components analysis can be used to compress an $n \times m$ feature matrix $M$ to an $n \times k$ matrix $M'$ (where $k \ll m$) in such a way that $M'$ retains most of the information of $M$. Similarly, we can think of word embeddings as a compression of a $v \times t$ word–text incidence matrix $M$, where $t$ is the number of documents in corpus, and $M[i, j]$ measures the relevance of word $i$ to document $j$. Compressing this matrix to $v \times k$ would yield a form of word embedding.

That said, neural networks are the most popular approach to building word embeddings. Imagine a network where the input layer accepts the current embeddings of (say) five words, $w_1, \ldots, w_5$, corresponding to a particular five word phrase from our document training corps. The network's task might be to predict the embedding of the middle word $w_3$ from the embeddings of the flanking four words. Through backpropagation, we can adjust the weights of the nodes in the network so it improves the accuracy on this particular example. The key here is that we continue the backpropagation past the lowest level, so that we modify the actual input parameters! These parameters represented the embeddings for the words in the given phrase, so this step improves the embedding for the prediction task. Repeating this on a large number of training examples yields a meaningful embedding for the entire vocabulary.

A major reason for the popularity of word embeddings is *word2vec*, a terrific implementation of this algorithm, which can rapidly train embeddings for hundreds of thousands of vocabulary words on gigabytes of text in a totally unsupervised manner. The most important parameter you must set is the desired number of dimensions $d$. If $d$ is too small, the embedding does not have the freedom to fully capture the meaning of the given symbol. If $d$ is too large, the representation becomes unwieldy and overfit. Generally speaking, the sweet spot lies somewhere between 50 and 300 dimensions.

### Graph Embeddings

Suppose we are given an $n \times n$ pairwise similarity matrix $S$ defined over a universe of $n$ items. We can construct the adjacency matrix of similarity graph $G$ by declaring an edge $(x, y)$ whenever the similarity of $x$ and $y$ in $S$ is high enough. This large matrix $G$ might be compressed using singular value decomposition (SVD) or principle components analysis (PCA), but this proves expensive on large networks.

Programs like *word2vec* do an excellent job constructing representations from sequences of symbols in a training corpus. The key to applying them in new

domains is mapping your particular data set to strings over an interesting vocabulary. *DeepWalk* is an approach to building *graph embeddings*, point representations for each vertex such that "similar" vertices are placed close together in space.

Our vocabulary can be chosen to be the set of distinct vertex IDs, from 1 to $n$. But what is the text that can represent the graph as a sequence of symbols? We can construct random walks over the network, where we start from an arbitrary vertex and repeatedly jump to a random neighbor. These walks can be thought of as "sentences" over our vocabulary of vertex-words. The resulting embeddings, after running *word2vec* on these random walks, prove very effective features in applications.

DeepWalk is an excellent illustration of how word embeddings can be used to capture meaning from any large-scale corpus of sequences, irrespective of whether they are drawn from a natural language. The same idea plays an important role in the following war story.

## 11.7 War Story: The Name Game

My brother uses the name *Thor Rabinowitz* whenever he needs an alias for a restaurant reservation or online form. To understand this war story, you first have to appreciate why this is very funny.

- *Thor* is the name of an ancient Norse god, and more recent super-hero character. There are a small but not insignificant number of people in the world named Thor, the majority of whom presumably are Norwegian.
- *Rabinowitz* is a Polish-Jewish surname, which means "son of the rabbi." There are a small but not insignificant number of people in the world named Rabinowitz, essentially none of whom are Norwegian.

The upshot is that there has never been a person with that name, a fact you can readily confirm by Googling "Thor Rabinowitz". Mentioning this name should trigger cognitive dissonance in any listener, because the two names are so culturally incompatible.

The specter of Thor Rabinowitz hangs over this tale. My colleague Yifan Hu was trying to find a way to prove that a user logging in from a suspicious machine was really who they said they were. If the next login attempt to my account suddenly comes from Nigeria after many years in New York, is it really me or a bad guy trying to steal my account?

"The bad guy won't know who your friends are," Yifan observed. "What if we challenge you to recognize the names of two friends from your email contact list in a list of fake names. Only the real owner will know who they are."

"How are you going to get the fake names?" I asked. "Maybe use the names of other people who are not contacts of the owner?"

"No way," said Yifan. "Customers will get upset if we show their names to the bad guy. But we can just make up names by picking first and last names and sticking them together."

"But Thor Rabinowitz wouldn't fool anybody," I countered, explaining the need for cultural compatibility.

We needed a way to represent names so as to capture subtle cultural affinities. He suggested something like a word embedding could do the job, but we needed training text be that would encode this information.

Yifan rose to the occasion. He obtained a data set composed of the names of the most important email contacts for over 2 million people. Contact lists for representative individuals[5] might be:

- *Brad Pitt*: Angelina Jolie, Jennifer Aniston, George Clooney, Cate Blanchett, Julia Roberts.
- *Donald Trump*: Mike Pence, Ivanika Trump, Paul Ryan, Vladimir Putin, Mitch McConnell.
- *Xi Jinping*: Hu Jintao, Jiang Zemin, Peng Liyuan, Xi Mingze, Ke Lingling.

We could treat each email contact list as a string of names, and then concatenate these strings to be sentences in a 2 million-line document. Feeding this to word2vec would train embeddings for each first/last name token appearing in the corpus. Since certain name tokens like *John* could appear either as first or last names, we created separate symbols to distinguish the cases of *John/1* from *John/2*.

Word2vec made short work of the task, creating a one hundred-dimensional vector for each name token with marvelous locality properties. First names associated with the same gender clustered near each other. Why? Men generally have more male friends in their contact list than women, and visa versa. These co-locations pulled the genders together. Within each gender we see clusterings of names by ethnic groupings: Chinese names near Chinese names and Turkish names near other Turkish names. The principle that birds of a feather flock together (*homophily*) holds here as well.

Names regularly go in and out of fashion. We even see names clustering by age of popularity. My daughter's friends all seem to have names like *Brianna*, *Brittany*, *Jessica*, and *Samantha*. Sure enough, these name embeddings cluster tightly together in space, because they do so in time: these kids tend to communicate most often with peers of similar age.

We see similar phenomena with last name tokens. Figure 11.16 presents a map of the 5000 most frequent last names, drawn by projecting our one hundred-dimensional name embeddings down to two dimensions. Names have been color-coded according to their dominant racial classification according to U.S. Census data. The cutouts in Figure 11.16 highlight the homogeneity of regions by cultural group. Overall the embedding clearly places White, Black, Hispanic, and Asian names in large contiguous regions. There are two distinct

[5]Great care was taken throughout this project to preserve user privacy. The names of the email account owners were never included in the data, and all uncommon names were filtered out. To prevent any possible misinterpretation here, the examples shown are not *really* the contact lists for Brad Pitt, Donald Trump, and Xi Jinping.

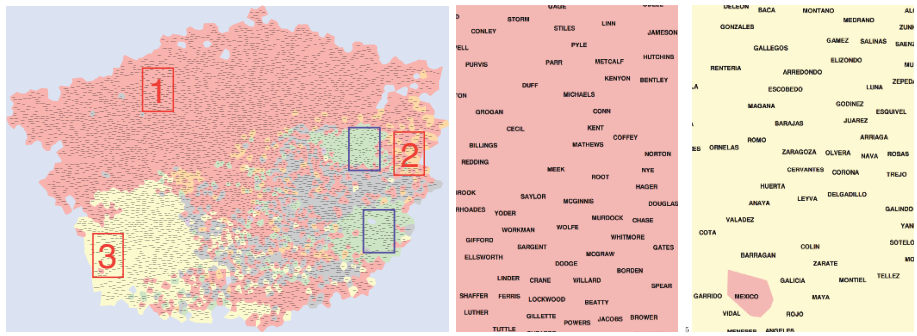


Figure 11.16: Visualization of the name embedding for the most frequent 5000 last names from email contact data, showing a two-dimensional projection view of the embedding (left). Insets from left to right highlight British (center), and Hispanic (right) names.

Asian regions in the map. Figure 11.17 presents insets for these two regions, revealing that one cluster consists of Chinese names and the other of Indian names.

With very few *Thors* corresponding with very few *Rabinowitzes*, these corresponding name tokens are destined to lie far apart in embedding space. But the first name tokens popular within a given demographic are likely to lie near the last names from the same demographic, since the same close linkages appear in individual contact lists. Thus the nearest last name token $y$ to a specific first name token $x$ is likely to be culturally compatible, making $xy$ a good candidate for a reasonable-sounding name.

The moral of this story is the power of word embeddings to effortlessly capture structure latent in any long sequence of symbols, where order matters. Programs like word2vec are great fun to play with, and remarkably easy to use. Experiment with any interesting data set you have, and you will be surprised at the properties it uncovers.

## 11.8 Chapter Notes

Good introductions to machine learning include Bishop [Bis07] and Friedman et al. [FHT01]. Deep learning is currently the most exciting area of machine learning, with the book by Goodfellow, Bengio, and Courville [GBC16] serving as the most comprehensive treatment.

Word embeddings were introduced by Mikolov et al. [MCCD13], along with their powerful implementation of word2vec. Goldberg and Levy [LG14] have shown that word2vec is implicitly factoring the pointwise mutual information matrix of word co-locations. In fact, the neural network model is not really fundamental to what it is doing. Our DeepWalk approach to graph embeddings is described in Perozzi et al. [PaRS14].

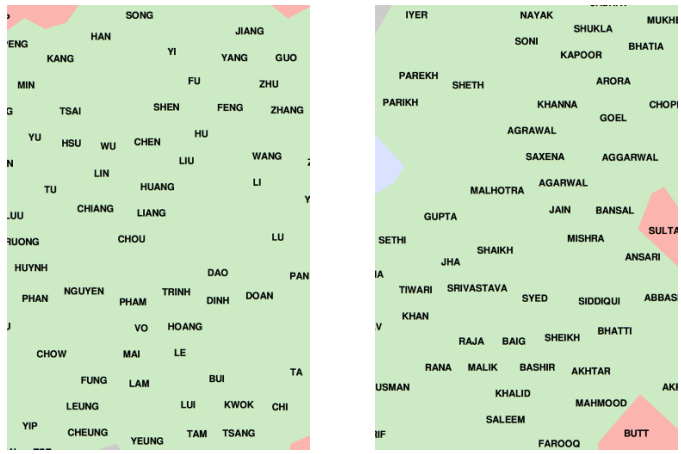


Figure 11.17: The two distinct Asian clusters in name space reflect different cultural groups. On the left, an inset showing Chinese/South Asian names. On the right, an inset from the cluster of Indian family names.

The Titanic survival examples are derived from the Kaggle competition `https://www.kaggle.com/c/titanic`. The war story on fake name generation is a result of work with Shuchu Han, Yifan Hu, Baris Coskun, and Meizhu Liu at Yahoo labs.

## 11.9 Exercises

### Classification

11-1. *[3]* Using the naive Bayes classifier of Figure 11.2, decide whether (Cloudy,High, Normal) and (Sunny,Low,High) are beach days.

11-2. *[8]* Apply the naive Bayes technique for multiclass text classification. Specifically, use *The New York Times Developer API* to fetch recent articles from several sections of the newspaper. Then, using the simple Bernoulli model for word presence, implement a classifier which, given the text of an article from *The New York Times*, predicts which section the article belongs to.

11-3. *[3]* What is regularization, and what kind of problems with machine learning does it solve?

### Decision Trees

11-4. *[3]* Give decision trees to represent the following Boolean functions:

(a) $A$ and $\bar{B}$.

(b) $A$ or ($B$ and $C$).

(c) ($A$ and $B$) or ($C$ and $D$).

11-5. *[3]* Suppose we are given an $n \times d$ labeled classification data matrix, where each item has an associated label *class A* or *class B*. Give a proof or a counterexample to each of the statements below:

(a) Does there always exist a decision tree classifier which perfectly separates $A$ from $B$?

(b) Does there always exist a decision tree classifier which perfectly separates $A$ from $B$ if the $n$ feature vectors are all distinct?

(c) Does there always exist a logistic regression classifier which perfectly separates $A$ from $B$?

(d) Does there always exist a logistic regression classifier which perfectly separates $A$ from $B$ if the $n$ feature vectors are all distinct?

11-6. *[3]* Consider a set of $n$ labeled points in two dimensions. Is it possible to build a finite-sized decision tree classifier with tests of the form "*is* $x > c$*?*", "*is* $x < c$*?*", "*is* $y > c$*?*", and "*is* $y < c$*?*" which classifies each possible query exactly like a nearest neighbor classifier?

### Support Vector Machines

11-7. *[3]* Give a linear-time algorithm to find the maximum-width separating line in one dimension.

11-8. *[8]* Give an $O(n^{k+1})$ algorithm to find the maximum-width separating line in $k$ dimensions.

11-9. *[3]* Suppose we use support vector machines to find a perfect separating line between a given set of $n$ red and blue points. Now suppose we delete all the points which are not support vectors, and use SVM to find the best separator of what remains. Might this separating line be different than the one before?

### Neural Networks

11-10. *[5]* Specify the network structure and node activation functions to enable a neural network model to implement linear regression.

11-11. *[5]* Specify the network structure and node activation functions to enable a neural network model to implement logistic regression.

### Implementation Projects

11-12. *[5]* Find a data set involving an interesting sequence of symbols: perhaps text, color sequences in images, or event logs from some device. Use word2vec to construct symbol embeddings from them, and explore through nearest neighbor analysis. What interesting structures do the embeddings capture?

11-13. *[5]* Experiment with different discounting methods estimating the frequency of words in English. In particular, evaluate the degree to which frequencies on short text files (1000 words, 10,000 words, 100,000 words, and 1,000,000 words) reflect the frequencies over a large text corpora, say, 10,000,000 words.

### Interview Questions

11-14. *[5]* What is deep learning? What are some of the characteristics that distinguish it from traditional machine learning

11-15. *[5]* When would you use random forests vs. SVMs, and why?

11-16. *[5]* Do you think fifty small decision trees are better than a large one? Why?

11-17. *[8]* How would you come up with a program to identify plagiarism in documents?

### Kaggle Challenges

11-18. How relevant are given search results to the user?
`https://www.kaggle.com/c/crowdflower-search-relevance`

11-19. Did a movie reviewer like or dislike the film?
`https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews`

11-20. From sensor data, determine which home appliance is currently in use.
`https://www.kaggle.com/c/belkin-energy-disaggregation-competition`