1 Introduction

Recommender system find and summarizes patterns in some structure (and those patterns can include how, in the past, users have explored that structure). One way to find those patterns is to use data mining algorithms.

The rest of this book focuses specifically on recommender systems. But, just to get us started, this chapter is a tutorial introduction to data mining algorithms:

- This chapter covers C4.5, k-Means, Apriori, AdaBoost, kNN, Naive Bayes, CART, and SVM.
- Also mentioned will be Random Forests, DbScan, canopy clustering, mini-batch K-means, single-pass K-Means, GENIC, the Fayyad-Irani discretizer, Infogain, Tf*Idf, PDDP, PCA, and LSI.
- There will also be some discussion on how to use the above for text mining.
- Data mining is a very active field. Hence, any summary of that field must be incomplete. Therefore this chapter ends with some suggested readings for those who want to read more about this exciting field.

Every learning method is biased in some way and it is important to understand those biases. Accordingly, it is important to understand two biases of this chapter.

Firstly, it will be the view of this chapter that it is a mistake to use data miners as black box tools. In that black-box view, the learners are applied without any comprehension of their internal workings. To avoid that mistake, it is useful for data mining novices to reflect on these algorithms as a menu of design options can be mixed and matched and mashed-up as required.

Tim Menzies
Lane Department of Computer Science and Electrical Engineering, West Virginia University, USA e-mail: tim@menzies.us
Accordingly, where appropriate, this chapter will take care to show how parts of one learner might be used for another.

Secondly, this chapter discusses newer methods such as CLIFF, WHERE, W2, and the QUICK active learner. By newer methods, we refer to the work of the author, his collaborators, or his graduate students. Caveat emptor!

2 Different Learners for Different Data

Let’s start at the very beginning (a very good place to start). When you read you begin with A-B-C. When you mine, you begin with data.

Different kinds of data miners work best of different kinds of data. Such data may be viewed as tables of examples:

- Tables have one column per feature and one row per example.
- The columns may be numeric (has numbers) or discrete (contain symbols).
- Also, some columns are goals (things we want to predict using the other columns).
- Finally, columns may contain missing values.

For example, in text mining, where there is one column per word and one row per document, the columns contain many missing values (since not all words appear in all documents) and there may be hundreds of thousands of columns.

While text mining applications can have many columns. Big Data applications can have any number of columns and millions to billions of rows. For such very large data sets, a complete analysis may be impossible. Hence, these might be sampled probabilistically (e.g. using the Naive Bayes algorithm discussed below).

On the other hand, when there are very few rows, data mining may fail since there are too few examples to support summarization. For such sparse tables, k-th nearest neighbors (kNN) may be best. kNN makes conclusions about new examples by looking at their neighborhood in the space of old examples. Hence, kNN only needs a few (or even only one) similar examples to make conclusions.

If a table has no goal columns, then this is an unsupervised learning problem that might be addressed by (say) finding clusters of similar rows using, say, K-means or EM. An alternate approach, taken by the APRORI association rule learner, is to assume that every column is a goal and to look for what combinations of any values predict for any combination of any other.

If a table has one goal, the this is a supervised learning problem where the task is to find combinations of values from the other columns that predict for the goal values. Note that for data sets with one discrete goal feature, it is common to call that goal the class of the data set.

For example, here is a table of data for a simple data mining problem:
In this table, we are trying to predict for the goal of play?, given a record of the weather. Each row is one example where we did or did not play golf (and the goal of data mining is to find what weather predicts for playing golf).

Note that temp and humidity are numeric columns and there are no missing values.

Such simple tables are characterized by just a few columns and not many rows (say, dozens to thousands). Traditionally, such simple data mining problems have been explored by C4.5 and CART. However, with some clever sampling of the data, it is possible to scale these traditional learners to Big Data problems [6, 7].

### 3 Association Rules

The APRIORI learner seeks associations; i.e. sets of ranges that after often found in the same row. First published in the early 1990s [45], APRIORI is a classic recommendation algorithm for assisting shopper. It was initially developed to answer the shopping basket problem; i.e. if a customer buys X, what else might they buy?

APRIORI can be used by, say, an on-line book store to make recommendations about what else a user might like to see. To use APRIORI, all numeric values must be discretized; i.e. the numerics ranges replaced with a small number of discrete symbols. Later in this chapter, we discuss several ways to perform discretization but a X% chop is sometimes as good as anything else. In this approach, numeric feature values are sorted and then divided into X equal sized bins. X=10 is a standard default but the above table is very small so we will use X=2 to generate:
In the discretized data, APRORI then looks for sets of ranges where the larger set is found often in the smaller. For example, one such rule in our table is:

play=yes \Rightarrow \text{humidity=up to 82.5} \& \text{windy=False}

That is, sometimes when we play, humidity is high and there is no wind.

Other associations in this data set include:

- humidity=up to 82.5 \& windy=False \Rightarrow play = no
- humidity=over 82.5 \Rightarrow play = no
- temperature=up to 73.5 \Rightarrow outlook = rainy
- outlook=overcast \Rightarrow play = yes
- outlook=rainy \Rightarrow temperature = up to 73.5
- play=yes \Rightarrow humidity = over 82.5
- play=yes \Rightarrow outlook = overcast

Note that in association rule learning, the left or right hand side of the rule can contain one or more ranges. Also, while all the above are associations within our play data, some are much rarer than others. APRORI can generate any number of rules depending on a set of tuning parameter that define, say, the minimum number of examples needed before we can print a rule.

Formally, we say that an association rule learner takes as input D “transactions” of items I (e.g. see the above example table). As shown above, association rule learners return rules of the form LHS \Rightarrow RHS where LHS \subset I and RHS \subset I and LHS \cap RHS = \emptyset. In the terminology of APRORI, an association rule X \Rightarrow Y has support s if s\% of D contains X \& Y; i.e., \( s = \frac{|X \& Y|}{|D|} \)
(where |X \& Y| denotes the number of |D| transactions/examples containing both X and Y). The confidence c of an association rule is the percent of transactions/examples containing X which also contain Y; i.e., \( c = \frac{|X \& Y|}{|X|} \).

As an example of these measures, consider the following rule:

play=yes \Rightarrow outlook = overcast

In this rule, LHS = X = play=yes and RHS = Y = outlook=overcast.

Hence:

- Support = \( \frac{|X \& Y|}{|D|} = \frac{4}{14} = 0.29 \)
- Confidence = \( \frac{|X \& Y|}{|X|} = \frac{4}{4} = 1.0 \).
APRIORI was the first association rule pruning. When it was first proposed (1992) it was famous for its scalability. Running on a 33MHz machine with 64MB of RAM, APRIORI was able to find associations in 838MB of data in under 100 seconds- which was quite a feat for those days. To achieve this, APRIORI explored progressively larger combinations of ranges. Further, the search for larger associations was constrained to smaller associations that occurred frequently. These frequent item sets were grown incrementally and APRIORI only explored items sets of size $N$ using items that occurred frequently of size $M < N$. Formally speaking, APRIORI uses support-based pruning i.e. when searching for rules with high confidence, sets of items $I_1, \ldots, I_k$ are only be examined if all its subsets are above some minimum support value. After that, confidence-based pruning is applied to reject all rules that fall below some minimal threshold of adequate confidence.

3.1 Technical Aside: How to Discretize?

In the above example, we used a discretization policy before running APRIORI. Such discretization is a useful technique for many other learning schemes (and we will return to discretization many times in this chapter). For now, we just say that discretization need not be very clever [49]. For example, a 10% chop is often as good as anything else (exception: for small tables of data like that shown above, it may be necessary to use fewer chops, just in case not enough information falls into each bin).

A newer method for discretization is to generate many small bins (e.g. 10 bins) then combine adjacent bins whose mean values are about the same. To apply this newer approach, we need some definition of “about the same” such as the Hedges test of Figure 1.

4 Learning Trees

APRIORI find sets of interesting associations. For some applications this is useful but, when the query is more directed, another kind of learner may be more suited.

4.1 C4.5

The C4.5 decision tree learner [44] tries to ignore everything except the minimum combination of feature ranges that lead to different decisions. For example, if C4.5 reads the raw golf data (from §2), it would focus of the play?
The Hedges test explores two populations, each of which is characterized by its size, their mean, and standard deviation (denoted $n$, $\text{mean}$, $sd$, respectively). When testing if these two populations are different, we need to consider the following:

- If the standard deviation is large, then this *confuses* our ability to distinguish the bins.
- But if the sample size is large then we can *attenuate* the effects of the large standard deviation; i.e. the more we know about the sample, the more certain we are of the mean values.

Combining all that, we arrive at an informal measure of the difference between two means (note that this expression weights *confusion* by how many samples are trying to confuse us):

$$
\text{attenuate} = n_1 + n_2 \\
\text{confusion} = \frac{n_1*sd_1 + n_2*sd_2}{\text{attenuate}} \\
\text{delta} = \frac{\text{abs}(\text{mean}_1 - \text{mean}_2)}{\text{confusion}}
$$

A more formally accepted version of the above, as endorsed by Kampenes et al. [27] is the following *Hedges' test*. To explain the difference between the above expression and *Hedges' test*, note that:

- This test returns true if the *delta* is less than some *small* amount. The correct value of *small* is somewhat debatable but the values shown below are in the lower third of the *small* values seen in the 284 tests from the 64 experiments reviewed by Kampenes et al.
- A c term is added to handle small sample sizes (less than 20).
- Standard practice in statistics is to:
  - Use $n - 1$ in standard deviation calculations;
  - Use variance $sd^2$ rather than standard deviation.

```python
function hedges(n1,mean1,sd1, n2,mean2,sd2):
    small = 0.17 # for a strict test. for a less severe test, use 0.38
    m1 = n1 - 1
    m2 = n2 - 1
    attenuate = m1 + m2
    confusion = sqrt( ( m1 * (sd1)^2 + m2 * (sd2)^2 ) / attenuate)
    delta = abs(mean1 - mean2) / confusion
    c = 1 - 3/(4*(m1 + n1 ) - 1)
    return delta * c < small
```

Fig. 1 A tutorial on the Hedges test of the *effect size* of the difference between two populations.

feature. It would then report what other feature ranges lead to such playful behavior. That report would take the form of the following tree:

```plaintext
outlook = sunny
  | humidity <= 75: yes
  | humidity > 75: no
outlook = overcast: yes
outlook = rainy
  | windy = TRUE: no
  | windy = FALSE: yes
```
To read this decision tree, note that sub-trees are indented and that any line containing a colon (:) is a prediction. For example, the top branch of this tree says:

\[ \text{IF outlook=sunny AND humidity} \leq 75 \text{ THEN we will play golf.} \]

Note that this decision tree does not include `temp`. This is not to say that golf playing behavior is unaffected by cold or heat. Rather, it is saying that for this data, the other features are more important.

C4.5 looks for a feature value that simplifies the data. For example consider the above table with five examples of no playing of golf and nine examples of yes, we played golf. Note that the baseline distributions in the table are \( p_1 = 5/14 \) and \( p_2 = 9/14 \) for no and yes (respectively). Now look at the middle of the above tree, at the branch `outlook=overcast`. C4.5 built this branch since within that region, the distributions are very simple indeed: all the rows where the outlook is overcast have `play=yes`. That is, in this sub-tree \( p_1 = 0 \) and \( p_2 = 100\% \).

Formally, we say that decision tree learners look for splits in the data that reduces the diversity of the data. This diversity is measured by the entropy equation discussed in Figure 2. For example, in the golf example, the relative frequency of each class was \( p_1 = 5/14 \) and \( p_2 = 9/14 \). In that case:

\[
\begin{align*}
\text{e} &= \text{entropy}([5/14, 9/14]) \\
&= -5/14 \cdot \log_2(5/14) - 9/14 \cdot \log_2(9/14) = 0.94
\end{align*}
\]

For the subtree selected by `outlook=overcast`, where \( p_1 = 0 \) and \( p_2 = 100\% \), we ignore the zero value (since there is no information there) and compute:

\[
\begin{align*}
n_1 &= 4 \\
\text{e}_1 &= \text{entropy([1])} = -1 \cdot \log_2(1) = 0
\end{align*}
\]

Note that for the subtree with five rows selected by `outlook=sunny`, there are two yes and one no. That is:

\[
\begin{align*}
n_2 &= 5 \\
\text{e}_2 &= \text{entropy([2/5, 3/5])} = 0.97
\end{align*}
\]

Also, and for the subtree with five rows selected by `outlook=rainy`, there are three yes and two no. Hence:

\[
\begin{align*}
n_3 &= 5 \\
\text{e}_3 &= \text{entropy([3/5, 2/5])} = 0.97
\end{align*}
\]

From the above, we can compute the expected value of the entropy after dividing the 14 rows in our table using the above tree:

\[
\begin{align*}
n &= n_1 + n_2 + n_3 = 14 \\
\text{expect} &= 4/14 \cdot 0 + 5/14 \cdot 0.97 + 5/14 \cdot 0.97 = 0.65
\end{align*}
\]

That is, the above tree has simplified the data from \( e = 0.94 \) to the new expected value of 0.65.
How to measure diversity? For numeric classes, it is customary to use standard deviation. However, for discrete classes, we need something else. To define such a diversity measure, we start with the following intuition:

- A population that contains only one thing is not diverse;
- A population that contains many things is more diverse.

Consider some sheep and cows in a barnyard, which we will represent as a piece of paper. Imagine that the animals do not like each other so they huddle in different corners. Say the sheep cover 10% of the yard and the cows cover 30%. To get some wool, we keep folding the piece of paper in half till we find all those sheep—a process we can represent mathematically as \( \log_2(0.1) \) The same cost to find the cows takes \( \log_2(0.3) \). The expected value of that search is the probability of each population times the cost of finding that population; i.e.

\[
-1 \times (0.1 \times \log_2(0.1) + 0.3 \times \log_2(0.3))
\]

The log of a probability less than one is negative so, by convention, we multiply by a minus sign. This informal example, while illustrative, has limitations (e.g. it ignores details like the 60% grass). The formal definition of symbol diversity comes from Claude Shannon’s famous information entropy expression:

\[
\text{entropy}(\{p_1, p_2, \ldots, p_n\}) = - \sum_{i} p_i \log_2(p_i)
\]

For the formal derivation of this equation (that says nothing about sheep or cows or folding little bits of paper) see the excellent discussion in http://goo.gl/SRaC2. Shannon used entropy as a way to measure how much signal was in a transmission:

- A piece of paper that is full of only one thing has, by definition, one thing everywhere. In terms of the above discussion, this is the population that is not diverse.
- Such a piece of paper has no distinctions; i.e. no regions where one thing becomes another.
- Hence, to transmit that information takes zero bits since there is nothing to say.

Note that Shannon’s equation captures this “zero bits” case. If you only have one thing then \( p=1 \) and entropy is zero:

\[
\text{entropy}(\{1\}) = 0
\]

On the other hand, as we increase diversity, the more bits are required to transmit that signal. For example, having three similar things is less diverse than having four or five similar things (as we might expect):

\[
\begin{align*}
\text{entropy}(\{10, 10, 10\}) &= 1.58 \\
\text{entropy}(\{10, 10, 10, 10\}) &= 2 \\
\text{entropy}(\{10, 10, 10, 10, 10\}) &= 2.32
\end{align*}
\]

Fig. 2 A tutorial on measuring diversity using Shannon entropy.

4.2 CART

The CART regression tree learner is another traditional learner first developed for simple data mining problems [5]. Like C4.5, CART has been a frame-
work within which many researchers have proposed exciting new kinds of data miners (for example, the research on CART lead to the invention of random forests, discussed below).

For certain classes of problems, CART is known to work as well or better as more complex schemes [13]. The lesson here is that before rushing off to try to latest and greatest new learner, it is worthwhile spend some effort on simpler learners like CART and C4.5. At the very least, these simple learners will offer baseline results against which supposedly more sophisticated methods can be compared.

The entropy equation of C4.5 assumes that the goal class is discrete. The CART regression tree learner applies the same recursive split procedure of C4.5, but it assumes the goal class is numeric. CART generates regression trees which look the same as decision trees but their leaves contain numeric predictions. For such numeric goals, we can measure the diversity of the class distribution using standard deviation.

For reasons of speed, it is useful to compute standard deviation using a single-pass algorithm. Suppose we have a \( n \) measures of a numeric goals in a class distribution \( x_1, x_2, x_3, \ldots \). If \( t \) is the sum of all \( x \) variables and \( t^2 \) is the sum of the square of all \( x \) variables, then

\[
\text{stdev} = \sqrt{\frac{t^2 - (t^2/n)}{n-1}}
\]

Apart from the handling of the class variable, C4.5 and CART work in very same ways: they try split on all features, then they use the split that most reduces diversity:

- CART finds splits that divide the \( n \) numeric goals into \( n_i \) divisions, each with standard deviation \( s_i \).

Both algorithms then apply some weighted sum to compute the expected value of the split:

- C4.5 : expected diversity = \( \sum_{i} \frac{n_i}{n} \times e_i \)
- CART: expected diversity = \( \sum_{i} \frac{n_i}{n} \times s_i \)

Once they find the feature that generates the split with the lowest diversity, they then apply that split and recurse on each division.

### 4.3 Hints and Tips for CART and C4.5

Any recursive algorithm such as CART (or C4.5) needs a stopping criteria; e.g. stop when there are less than \( M \) examples falling into each sub-tree.
As $M$ gets larger, it becomes harder to form new sub-trees so the total tree size shrinks. That is, the tree becomes easier to read.

As $M$ gets smaller, it becomes easier for the learner to explore special cases within the data. That is, the predictions of the tree can become more accurate. Random forests, discussed below, use very small $M$ values (e.g., $M = 2$).

For example, the housing\(^1\) data set describes 506 houses from Boston. Each house is described in terms of 14 features (and the last feature “PRICE” is the target concept we seek to predict):

- CRIM: per capita crime rate by town
- ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS: proportion of non-retail business acres per town
- CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise);
- NOX: nitric oxides concentration (parts per 10 million);
- RM: average number of rooms per dwelling
- AGE: proportion of owner-occupied units built prior to 1940
- DIS: weighted distances to five Boston employment centers
- RAD: index of accessibility to radial highways
- TAX: full-value property-tax rate per $10,000
- PRTATIO: pupil-teacher ratio by town
- B: $1000(B - 0.63)^2$ where $B$ is the proportion of blocks by town
- LSTAT: lower status of the population
- PRICE: Median value of owner-occupied homes.

With the default value of $M = 4$, CART generates a tree with 28 leaves. But with $M = 100$, we generate a much more readable and smaller tree with only 9 leaves. This tree is shown below:

```
STAT <= 9.725 :
  | RM <= 6.941 :
  | | DIS <= 3.325 : PRICE = 27.4
  | | DIS > 3.325 :
  | | | RM <= 6.545 : PRICE = 23.8
  | | | RM > 6.545 : PRICE = 26.8
  | RM > 6.941 : 36.0
LSTAT > 9.725 :
  | LSTAT <= 15 :
  | | DIS <= 4.428 :
  | | | TAX <= 300 : PRICE = 21.9
  | | | TAX > 300 : PRICE = 20.3
  | | DIS > 4.428 : PRICE = 19.7
  | LSTAT > 15 :
  | | CRIM <= 5.769 : PRICE = 17.0
  | | CRIM > 5.769 : PRICE = 13.6
```

Any line containing a colon (:) is a prediction. For example, the top branch of this decision tree is saying:

```
IF STAT < 9.725 AND RM <= 6.941 and DIS <= 3.325 THEN PRICE = 27.4
```

\(^1\) http://archive.ics.uci.edu/ml/datasets/Housing.
The smaller tree, shown above, is less accurate than the tree grown with $M = 4$. However, the difference is not large:

- The predictions of the larger tree correlate with actuals at $R^2 = 91\%$.
- While the smaller tree is nearly as accurate with $R^2 = 86\%$.

(In the above, $R^2$ is a measure of how much one (say) class feature $f$ is determined by another variable $x$ and is called the Pearson correlation coefficient$^2$.)

Note that the smaller tree is much easier to read and understand while being nearly as accurate as the larger and more complex tree. When discussing a learned model with users, sometimes it is worth losing a few points in performance in order to display a smaller, more easily understood, tree. Note also that this trick of selecting $M$ in order to balance performance vs readability can be applied to any tree learning procedure including CART or C4.5.

Finally, it is worth mentioning that tree learners often include a post-pruning step where the data miner experiments with removing sub-trees. In this post-pruning, if the predictive power of the pruned tree is not worse than the original tree, then the pruner recurses on the reduced tree.

4.4 Random Forests

Traditional tree learners like CART and C4.5 cannot scale to Big Data problems since they assumes data is loaded into main memory and executed within one thread. There are many ways to address these issues such as Catlett’s classic peepholing method [7]. One of the most interesting, and simplest, is Brieman’s random forest method [6]. The motto of random forests is this:

> If one tree is good, why not build a whole bunch?

To build one tree in a random forest, pick a number $m$ less than the number of features. Then, to build a forest, build many trees as follows:

- Select some subset $d$ of the training data;
- Build a tree as above, but at each split, only consider $m$ features (selected at random);
- Do not bother to post-prune.

Finding the right $d$ and $m$ values for a particular data set means running the forests, checking the error rates in the predictions, then apply engineering judgement to select better values. Note that $d$ cannot be bigger than what

---

$^2$ Given a mean value for $x$ over $n$ measurements $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$, then the total sum of squares is $SS_{tot} = \sum_{i}(x_i - \bar{x})^2$ and the sum of squares of residuals is $SS_{err} = \sum_{i}(x_i - f_i)^2$. From this, the amount by which $x$ determines $f$ is $R^2 = 1 - (SS_{err}/SS_{tot})$. 

can fit into RAM. Also, a useful default for $m$ is the log of the number of features.

Random forests make predictions by passing test data down each tree. The output is the most common conclusion made by all trees.

Random forests have certain drawbacks:

- Random forests do not generate a single simple model that users can browse and understand. On the other hand, the forests can be queried to find the most important features (by asking what features across all the trees were used most as a split criteria).
- Some commonly used data mining toolkit insist that all the data loads into RAM before running random forests\(^3\).

Nevertheless, random forests are remarkably effective:

- Random forests generates predictions that are often as good, or better, than many other learners [6].
- They are fast. In 2001, Breiman reports experiments where running it on a data set with 50,000 cases and 100 variables, it produced 100 trees in 11 minutes on a 800Mhz machine. On modern machines, random forest learning is even faster.
- They scale to data sets with a very large numbers of rows or features: just repeatedly sample as much data as can fit into RAM.
- They extends naturally into cloud computing: just build forests on different CPUs.

Like C4.5 and CART, it might be best to think of random forests as a framework within which we can explore multiple data mining methods:

- When faced with data that is too big to process:
  - Many times repeat:
    - Learn something from subsets of the rows and features.
  - Then make conclusions by sampling across that ensemble of learners.

As seen with random forests, this strategy works well for decision tree learning, but it is useful for many other learners as well (later in this chapter we discuss an analog of random forests for the Naive Bayes classifier).

Note that for this style of random learning to be practical, each model must be learned very fast. Hence, when building such a learner, do not sweat the small stuff. If something looks tricky then just skip it (e.g. random forests do not do post-pruning). The lesson of random forests is that multiple simple samples can do better than fewer and more complex methods. Don’t worry, be happy.

\(^3\) But it should be emphasized that this is more an issue in the toolkit’s implementation than some fatal flaw with random forests.
A final note on random forests: they are an example of an ensemble learning method. The motto of ensemble learning is that if one expert is good, then many are better. While $N$ copies of the same expert is clearly a waste of resources, $N$ experts all learned from slightly different data can offer $N$ different perspectives on the same problem. Ensemble learning is an exciting area of data mining- and one that has proved most productive. For example:

- The annual KDD-cup is an international competition between data mining research teams. All the first and second-placed winners for 2009-2011 used ensemble methods.
- In our own work, our current best-of-breed learner for effort estimation is an ensemble method [29].

Later in this chapter we will discuss other kinds of ensemble learners such as AdaBoost.

### 4.5 Applications of Tree Learning

C4.5 and CART are widely influential algorithms. The clarity and simplicity of this kind of learning has allowed many researchers to develop innovative extensions. Those extensions include random forests (discussed above) as well as:

- **The Fayyad-Irani discretizer** is a cut-down version of C4.5 that builds a tree from a single feature. The leaves of that tree are returned as the learned bins [FI1993].
- **The InfoGain feature selector** [22]. This algorithm does not build any tree. Rather, it acts like C4.5’s first split when it conducts a what-if query over all features. InfoGain sorts the features by the entropy reduction that would result if the data was split on that data. A standard InfoGain algorithm requires discrete data and so is typically run over data that has been discretized by Fayyad-Irani.
- **PDDP** is a tree learner that splits on synthesizes features [2] At each level of the recursion, PDDP finds the eigenvectors of the correlation matrix of the data that falls into each sub-branch. Technically, this is a principle component analysis (PCA) that transforms the data to a new coordinate system such that the greatest variance by any projection of the data comes to lie on the first coordinate (called the first principal component).
- **WHERE** is a version of PDDP that uses the FASTMAP trick to find an approximation to the first component [35]. WHERE uses the Aha distance function (defined later in this chapter). After picking any row $X$ at random, WHERE finds
  - The row Right that is furthest from $X$;
  - It then finds the row Left that is furthest from Right.
WHERE then projects all rows onto the line that runs from Left to Right by finding:
- The distance $c$ between Left and Right;
- For every row, the distance $a$ and $b$ to rows Left and Right.
With that information, the projection of a row along the Left and Right line is $x = (a^2 + c^2 - b^2)/(2c)$.
WHERE finds the median $x$ value, then splits the data above and below that split point. The algorithm then recurses on each split.

These variants can be much simpler than standard C4.5 (or CART). For example:

- WHERE and PDDP can find much shallower trees than CART or C4.5. The reason for this is that if there exists $N$ correlated features, then the principle component found by PCA (or the approximation found by WHERE) can model those $N$ features with $M < N$ dimensions.
- WHERE runs much faster than PDDP. Finding the principle component takes polynomial time while WHERE’s projections take linear time (only $4N$ distance measures between rows).
- InfoGain does not even have to model the recursive tree data structure. This algorithm is widely used in text mining since it runs in linear time and takes very little memory. Hall and Holmes [22] comment that other feature selectors can be more powerful, but are slower and take more memory.
- Fayyad-Irani only needs to reason about two features at any one time (the numeric feature being discretized and the class feature). Hence, even if all the data cannot fit into RAM, it may still be possible to run Fayyad-Irani (and if memory problems persist, a simple pre-processor to the data that selects X% of the rows at random may suffice for learning the discretized ranges).

Just an aside, this Fayyad-Irani discretizer is useful for more than just building decision trees. It is also useful a procedure for ranking performance results from different learners (see Figure 3).

5 Naive Bayes

When working with a new data set, it is prudent to establish a baseline result using the most direct and simplest approach. Once that baseline performance is known, then it is possible to know how much more work is required for this data.

The most direct and simplest learning method discussed in this chapter is a Naive Bayes classifier. Despite their name, Naive Bayes classifiers are hardly naive. In fact, they offer a range of important services such as learning from very large data sets, incremental learning, anomaly detection, row pruning,
and feature pruning—all in near linear time (i.e. very fast). Better yet, as discussed below, implementing those services is trivially simple.

The reason these classifiers are called \textit{naive} is that they assume that within one class, all features are statistically independent. That is, knowledge about the value of one feature does not tell us anything about the value of any other. So a Naive Bayes classifier can never look at a table of medical diagnosis to infer that $\text{pulse}=0$ is associated with $\text{temperature}=\text{cold}$.

Proponents of tree learning would dispute this Naive Bayes assumption. They prefer algorithms like C4.5 or CART or random forests since tree learners always collect information in context: that is, all sub-trees refer to data in the context of the root of that tree.

Strange to say, Naive Bayes performs often as well, or better, as decision tree learning—an observation that is carefully documented and explained by Domingos and Pazzini [14]. In short, they defined the volume of the zone where Naive Bayes would make a different decision to some optimal Bayes (one that knows about correlations between features). That zone is very small.

Consider a two-column data set where column1 is the performance score of some learner and column2 is the name of that learner. If we sort on column1 then apply Fayyad-Irani, all the learners with the same scores will be grouped together in the one bin. In this rig, the \textit{best} learner is the most common learner found in the bin with greatest value. Alternatively, consider a small modification of Fayyad-Irani, but this time we are recursing through a sorted list of buckets, sorted on their mean score. Each bucket contains all the performance scores on one learner. The first level split finds the index of the list that divides the buckets into two lists, which we will call $\text{list1}$ and $\text{list2}$, that are up to and above the split:

- Let the mean of the performance scores in entire list and $\text{list1}$ and $\text{list2}$ be $\mu_0$, $\mu_1$ and $\mu_2$.
- Let the number of performance scores in $\text{list1}$ and $\text{list2}$ be $n_1, n_2$. A good split is the one that maximizes the expected value of the sum of square of the mean differences before and after divisions. If $n = n_1 + n_2$ then that expected value is:

$$\frac{n_1}{n} \cdot \text{abs}(\mu_1 - \mu_0)^2 + \frac{n_2}{n} \cdot \text{abs}(\mu_2 - \mu_0)^2$$

This is the Scott-Knott procedure for ranking different treatments. This procedure recurses, on the bins in each split, but only if some statistical test agrees that distributions in $\text{list1}$ and $\text{list2}$ are statistically different. This procedure is quite clever in that it can divide $T$ treatments using $\log_2(T)$ comparisons. Mittas and Angelis [38] recommend using ANOVA to test if $\text{list1}$ and $\text{list2}$ are significantly statistically different. If the distributions are non-Gaussian, they also recommend applying the Bloom transform to the data as a pre-processor. It turns out that if we use the simpler Hedges procedure (described above), then the learners are grouped in the same way as using an ANOVA+Bloom test.

\textbf{Fig. 3} A tutorial on using tree learners to rank treatments.
and grows vanishingly smaller as the number of features in the data set increases.

Which is another way of saying that the space where the Naive Bayes assumption is truly Naive is quite tiny. This is very good news since it means that a Naive Bayes classifier can store data using a simple frequency table. Hence, a Naive Bayes classifier has:

- A tiny memory footprint
- Is very fast to training and very fast to make conclusions;
- Simple to build.

Naive Bayes classifiers use the famous Bayes theorem to make predictions. Tables of data are separated into their various classes. Statistics are then collected for each class. For example, recall the play? data from section 2 of this chapter. Here are the statistics for that data. Note that all these numbers are divided into the two class variables play?=yes and play?=no.

<table>
<thead>
<tr>
<th>outlook</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>sunny</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>overcast</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rainy</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>mean 73</td>
<td>74.6</td>
</tr>
<tr>
<td></td>
<td>std dev 6.2</td>
<td>7.9</td>
</tr>
</tbody>
</table>

Underneath each feature are some cumulative statistics. For example:

- The mean humidity is different for yes and no: 79.1 and 86.2 (respectively).
- Overcast appears 4/9 times when we play and zero times with we do not play. That is, if in the future we see outlook=overcast then it is far more likely that we play that otherwise (to be precise, it is infinitely more likely that we will play).

One way to view a Naive Bayes classifier is a clustering algorithm where we have clustered together all things with the same class into the same cluster. When new data arrives we travel to every cluster and ask it to vote on does this look familiar to you?. And the cluster (class) that offers the most votes is used to make the prediction.

The voting procedure is to use Bayes’ rule. Bayes’ rule says that our new belief in an hypothesis is the product of our old beliefs times any new evidence; i.e.

\[ new = now \ast old \]
Just as aside: this simple equation hides a trick. If the test case (which is the \textit{now} term in the above equation) has a missing feature, we just assume that it offers no evidence for a conclusion. Hence, we can just skip over that feature. Note that this is a much simpler scheme (to say the least) for handling missing values than other learners like C4.5. In C4.5, if a sub-tree starts with some feature that is missing in the test case, C4.5 then performs an intricate what-if query of all sub-trees.

In Bayes’ rule, the probability of an hypothesis \( H \) given evidence \( E \):

\[
P(H|E) = P(E|H) \ast P(H)/P(E)
\]

In this expression:

- The \( P(E) \) term is the probability of the evidence. Since this is the same for each test case, it can be ignored (and a good thing too since it is very unclear how to calculate that term).
- The \textit{old} term is the prior probability of the hypothesis, denoted \( P(H) \). This is just the frequency of each hypothesis. In our playing example, \( H \) is \textit{yes} or \textit{no} so \( P(\text{yes})=9/14 \) and \( P(\text{no})=5/14 \).
- The probability of the evidence given the hypothesis, denoted \( P(E|H) \), is looked up from the table of statistics. This is the \textit{now} term.

For example, if we were told that tomorrow’s forecast was sunny, the classes would offer the following votes on how likely it is that we will play or not:

- \( P(\text{yes}|\text{outlook}=\text{sunny}) = 2/9 \ast 9/14 = 0.39 \)
- \( P(\text{no}|\text{outlook}=\text{sunny}) = 3/5 \ast 5/14 = 0.14 \)

Since \textit{yes} offers more votes, we would conclude that tomorrow we will play. Note that as more evidence arrives, the more information we can add to \textit{now}.

For example:

- Let the forecast be for sunshine, 66 degrees, 90% humidity and wind.
- That is \( E = (\text{outlook}=\text{sunshine}, \text{temp}=66, \text{humid}=90 \text{ and wind}=\text{true}) \)
- To handle this conjunction, we multiply the individual probabilities:

\[
P(\text{yes}|E) = 2/9 \ast 0.0340 \ast 0.0221 \ast 3/9 \ast 9/14 = 0.000036
\]
\[
P(\text{no}|E) = 3/5 \ast 0.0291 \ast 0.0380 \ast 3/5 \ast 5/14 = 0.000136
\]

- To report the above, we normalized the probabilities:

\[
P(\text{yes}|E) = 0.00036 \div (0.00036 + 0.000136) = 21\%
\]
\[
P(\text{no}|E) = 0.000136 \div (0.00036 + 0.000136) = 79\%
\]

- That is, for this forecast, \textit{no} is offering more votes than \textit{yes}. Hence, for this forecast, we would predict \textit{no}.  


The above calculation had some long decimal numbers (e.g. 0.0340). Where did these come from? Recall that temp=66 and for the class yes, the above table reports that the mean temperature was 73 with a standard deviation of 6.2. How likely is 66 on a bell-shaped curve whose mean is 73 and whose standard deviation is 6.2? We say that:

- The closer we get to the mean value, the higher the likelihood;
- The greater the diversity in the data, the lower the likelihood of any one value.

Both these notions can be expressed in terms of a bell-shaped curve; a.k.a. the normal or Gaussian distribution:

- This curve reaches maximum at the mean value;
- As the variance increases, the curve grows wider. Since the area under this probability curve must sum to one, the wider the curve, the lower the top-of-hill (where the mean is). That is increasing diversity decreases our confidence in a particular value.

Hence, a standard Naive Bayes classifier uses a Gaussian probability distribution function to compute the likelihood of any particular number:

```python
function gaussianPdf(mean, stdev, x)
    return 1/(stdev * sqrt(2*pi))* e ^ (-1*(x-mean)^2/(2*stdev*stdev))

print gaussianPdf(73, 6.2, 66)
===> 0.0340
```

Note that this calculation assumes that the underlying distribution is a bell-shaped Gaussian curve. While this can be a useful engineering approximation, it may not be true in many situations. Many Bayes classifiers discretize their numerics before making predictions, thus avoiding the need for this Gaussian assumptions. A repeated result is that the performance of Naive Bayes is improved by discretization [15]. A standard discretizer is the Fayyad-Irani approach discussed above in Section 4.5.

### 5.1 Bayes and Anomaly Detection

If data miners are used for mission critical or safety critical applications, it is important to understand when they cannot be trusted. This is the role of the anomaly detector. Such detectors trigger when the new example is outside the range of the examples used to train the learner. Several recent high-profile disasters could have been averted if anomaly detectors were running on learned models (see Figure 4).

Anomaly detection is a very active area of research in data mining. For a detailed survey, see [8]. But to give the reader a sample of how to build an anomaly detector, we mention here one anomaly detector that can be built using the above Bayesian statistics table.
Farnstrom and his colleagues [17] use a statistical approach to detect anomalies. Suppose we have access to the above Bayesian statistics table:

- Read each row and replace its class value with some single symbol (e.g. global).
- For this global class, build the above Bayes statistics table. With that table, compute the suspect region for each feature:
  - For numeric attributes, it is outside the mean value plus or minus 1.96 times the standard deviation (this corresponds to a 95% confidence interval).
  - For discrete attributes, list all values that occur less than 5% of the time in that row.
- When new data arrives, count how many times each feature falls into the suspect region:
  - Reject any row that has more than \( n \) features with suspect values\(^5\).

In 1999, NASA’s $125M Mars climate orbiter burned up in the Martian atmosphere after a mix-up in the units used to control the system:

- The problem was that meters were confused with feet and, as a result, the orbiter passed 60km, not 150km above the Martian atmosphere.
- The confusion in the units was apparent on-route to Mars—the spacecraft required unusually large course corrections\(^4\).
- Sadly, the ground crew had no anomaly detector to alert them to how serious this deviation was from the craft’s expected behavior.

In 2003, anomaly detection might have also saved the crew of the Columbia space shuttle. On re-entry, a hypersonic shock wave entered a hole in the craft’s wing and tore the craft apart:

- The hole was formed when the shuttle was struck at launch by a block of frozen foam measuring 1200 in\(^3\) and traveling at 477 mph (relative to the vehicle).
- Engineers concluded that such a strike was not hazardous using a program called CRATER. CRATER was trained on much smaller and much slower projectiles: a normal CRATER example was a 3 in\(^3\) piece of debris traveling at under 150 mph.
- An anomaly detector could have alerted NASA to mistrust the engineers’ conclusions since they were drawn from a region well outside of CRATER’s certified expertise.

![Fig. 4](image-url) High-profile disasters that might have been averted via anomaly detection.

\(^5\) Note that Farnstrom uses \( n = 1 \) but this is a parameter that can be tuned. In the next section, we discuss incremental learner where, at least during the initial learning phase, all the data will be anomalous since this learner has never seen anything before. For learning from very few examples, \( n \) should be greater than one.
The above shows how to detect anomalies—not what to do with them. This is a domain decision. One possibility is to store the anomalies in a bucket and, when that bucket gets too big, run a second learner just on those harder examples. For example, the trees generated by WHERE could be incrementally modified as follows:

- Build a tree from X% of the data;
- When new data arrives, push it down the tree to its nearest leaf cluster;
- At every level of that descent, check the Left and Right pairs. If the new, instance falls outside the range Left and Right:
  - Add the instance to a bucket of anomalies;
  - Mark which of Left and Right is closest to the anomaly.
- If the number of anomalies grows beyond some threshold, then rebuild any sub-tree with those anomalies. In that rebuild, use all the anomalies and any Left and Right row not marked by the last point.

Note that this approach implies we only need to rebuild parts of the tree—which is useful for any incremental learning scheme.

A more general version of this approach is the AdaBoost algorithm [18]. This is a meta-learner scheme; i.e. it can be applied to any learner (e.g. Naive Bayes, CART, C4.5, etc). Like random forests, it is one of the most important ensemble methods, since it has solid theoretical foundation, very accurate prediction, great simplicity (a few dozen lines of code to implement), and wide and successful applications. Also, this algorithm is can “boost” the performance of a weak learner to a higher classification accuracy.

 AdaBoost builds a sequence of T classifiers using some learner $t = 1, 2, \ldots, T$:

- To build a training set for classifier $t$, then $M$ times, sample with replacement from the data set according to the $D_t$ distribution described below.
- All $i \in m$ examples are given some weight. Initially that weight is $D_1(i) = \frac{1}{m}$. Subsequently, this weight is changed if classifier $t$ incorrectly classifies example $i$.
- Examples with the greater weights are used with higher probability by the next classifier $t+1$. That is, AdaBoost builds a sequence of classifiers $t = 1, 2, \ldots, T$ each of which focuses on the examples that were problematic for classifier $t - 1$.

AdaBoost’s updates the weights as follows:

- Let $\epsilon_t = (\sum_i D_t(i))/m$ be sum of the weights of the examples with incorrect classifications made by classifier $t$.
- Let $\beta_t = \epsilon_t/(1 - \epsilon_t)$.
- Let $D_{t+1}(i) = \frac{D_t(i)}{Z_t} \gamma$ where:
  - $\gamma = \beta_t$ if $i$ was correctly classified by classifier $t$. Otherwise $\gamma = 1$.
  - $Z_t$ is a normalization constant such that $\sum D_{t+1}(i) = 1$.  

To use this ensemble, each classifier $t = 1, 2, \ldots, T$ proposes its own classification $c$ with weight $\log \left(\frac{1}{\beta_t}\right)$ (and AdaBoost returns the classification with the largest weighted vote).

### 5.2 Incremental Bayes

Naive Bayes is an excellent candidate for mining very large of very long streams of data. This is due to the fact that the working memory of a Naive Bayes classifier can be very small:

- A summary of the data seen so far;
- The next test case to classify.

Such an incremental Naive Bayes classifier might work as follows:

- When a new test case arrives, classify it using the existing statistics.
- Then (and only then) update the statistics with the new case.

Some heuristics for incremental Bayes learning include the following:

1. Use a learner that updates, very fast. In this respect, a Naive Bayes classifier is a good candidate since its memory footprint is so small.
2. If the numerics in this domain do not conform to a Gaussian curve, use an incremental discretizer to convert the numbers to discrete values. Implementing such incremental discretizers is not a complex task [19].
3. If you use the Gaussian assumption, then be wary of floating point errors (particularly for very long data streams). Incrementally compute standard deviation using Knuth’s method\(^6\).
4. To emulate something like a random forest ensemble, split the incoming data into 10 streams (each containing 90% of the incoming data) and run a separate learner for each stream. Let each stream make a prediction and report the majority decision across all streams (perhaps weighted by accuracy performance statistic seen for all these learners). Note that since Naive Bayes has such a small footprint, then the memory overhead of running ten such classifiers is not excessive.
5. Skip any suspect examples (as defined by the Farnstrom detector) or run the anomalies in their own separate stream.
6. Add an initial randomizer buffer that reads the input data in blocks of (say) 10,000 examples, then spits them out in a random order (for a linear-time randomizer, use the Fisher Yates shuffle\(^7\)). This random buffer minimizes order effects where the learned model is just some quirk of the ordering of the data. Also, this buffer is useful for collecting preliminary statistics on the data such as the minimum and maximum value of numeric values.

---

\(^6\) [http://goo.gl/ytk6B](http://goo.gl/ytk6B)

\(^7\) [http://goo.gl/x8mNA](http://goo.gl/x8mNA)
7. Do not trust the classifier till it has seen enough data. Experience with simple data sets is that the performance of incremental Naive Bayes plateaus after a few hundred examples or less, but you need to check that point in your own data. In any case, it is wise to have some start-up period where classification is disabled.

8. To check for the performance plateau, divide the data into eras of, say, 100 examples in each era. Collect performance statistics across 10 streams. Compute the mean and standard deviation of accuracy in each era $i$. Declare a plateau if the performance of era $j = i + 1$ is about the same as era $i$ (e.g. using the Hedge’s effect size rule shown above).

For some data sets, the number of examples to reach plateau may be surprisingly brief. For example, certain defect data sets plateau after a 100 (or less) examples [36]. Recent results with the QUICK active learner suggest that this can be reduced to even fewer if we intelligently select the next example for training. For example, in some software effort estimation data sets, we have plateaued after just a dozen examples, or even less [30].

**5.3 Incremental Learning and Data Set Shift**

David Hand [23] warns that classifiers can make mistakes when their models become out-dated. This can occur when some structure is learned from old data, then the data generating phenomenon changes. For example, software effort estimations trained on COBOL must be recalibrated if ever those programmers move to JavaScript.

There are many ways to handle such data set shift and for a state of the art report, see the work of Minku and Yao [37]. But just to describe a simple way to handle data set shift, consider the incremental Bayes classifier described above. In the following circumstances, a very simple data set shift scheme can be applied:

- The number of examples required to reach the performance plateau is $t_1$;
- The rate of data set shift is $t_2$;
- The data shifts at a slower rate than the time required to reach plateau; i.e. $t_1 < t_2$.

In this (not uncommon) situation, a data scientist (or a recommender system) can handle data set shift by running two learners:

- An incremental Bayes classifier (described above);
- And any other learner they like (which, in fact, could also be a Bayes classifier).

While the performance plateau remains flat, the data scientist (or the recommender systems) can apply the other learner to all data seen too date. But
if the performance plateau starts changing (as measured by, say, the Hedges test of Figure 1) then data mining algorithm needs to dump the old model and start learning afresh.

6 SVM = Support Vector Machines

The learners discussed above try to find cuts between the features in tables of data that, say, predict for some class variable. An assumption of that approach is that the data divides nearly along the lines of the existing features.

Sometimes, lining up with the current features may not be the best policy. Consider the data shown at right: note that there is no simple split parallel to the x-axis or y-axis that best separates the black and the weight dots.

To solve this problem, we note that what looks complex in lower dimensions can actually be simpler in higher dimensions. If we map the two-dimensional points \( x_1, x_2 \) from the above figure into the three dimensions \( \phi(x_1, x_2) \rightarrow (x_1^2, x_2^2, \sqrt{2}x_1x_2) \), we arrive at the figure shown on the right. Note that there now exists a hyperplane that separates the black and white dots. Formally, \( \phi \) is a “kernel function” that transforms co-ordinates in one space into another.

Support vector machines (SVMs) are algorithms that can learn a hyperplane that separates classes in a hyperdimensional space [11] (for an example of such a hyperplane, see the rectangular shape in the above figure). Combined with some kernel function \( \phi \) (like the one shown above), SVMs can handle very complex data sets.

Internally, SVMs are quadratic optimizers that search for a hyperplane that best separates the classes. A maximum margin SVMs strives to maximize the distance of this hyperplane to the support

---

8 This example comes from Norvig and Russell’s excellent text *AI: A Modern Approach* [46]
vectors; i.e. examples from different classes that fall either side of the hyperplane.

In the absence of expert knowledge, a Radial Basis Function is the usual default kernel. For example, here is a simple Gaussian RBF:

$$\phi(r) = e^{-\epsilon r^2}$$

(here, $r$ is some distance of each example from an origin point where all values are zero).

Note that each kernel has a set of magic parameters that need to be tuned to local domain. A standard architecture is to use some tuning learner to find the best parameter settings— for example, when predicting software development effort, Corazza et al. use a tabu search to learn the parameters for their radial bias function [10].

7 Pruning Data

In the previous section, we solved the data mining problem via adding dimensions. In this section, we explore another approach where we delete dimensions (i.e. the features) and/or rows in a data set.

Many real-world data sets contain spurious or noisy or irrelevant or redundant data. For this reason, it is often useful to strategically prune some of the training data. Many researchers, such as, Levina and Bickel report that it is possible to prune seemingly complex data since:

"... the only reason any methods work in very high dimensions is that, in fact, the data are not truly high-dimensional. Rather, they are embedded in a high-dimensional space, but can be efficiently summarized in a space of a much lower dimension" [33].

The rest of this section discusses two kinds of pruning: feature pruning and row pruning.

7.1 Feature Pruning

One advantage of tree learners over Naive Bayes is that the trees can be a high-level and succinct representation of what was learned from the data. On the other hand, the internal data structures of a Bayes classifier are not very pleasant to browse. Many users require some succinct summary of those internals.

In terms of this chapter, dimensionality reduction means pruning some subset of the features in a table. Feature selection is an active area of data mining research. Two classic references in this area come from Kohavi [32]
and Hall and Holmes [22]. A repeated result is that in many data sets, most features can be removed without damaging our ability to make conclusions about the data.

As to specific feature pruning methods, we will introduce them in three steps:

1. Near linear-time pruning methods such as InfoGain and CLIFF.
2. Polynomial-time pruning methods such as PCA, or LSI.
3. Then, in Section 8, we will discuss other feature pruners especially designed for text mining such as stop lists, stemming, and Tf*Idf.

### 7.1.1 Near-Linear-time Feature Pruning

For near linear-time pruning method, recall the InfoGain method discussed above in Section 4.5 (*Applications of Tree Learning*). After discretization, each feature can be scored by the entropy resulting for dividing the class variable into the discrete values for this feature (and better features have lower entropy; i.e. their values select for a smaller range of class values). For each feature, this process requires an $O(n \log(n))$ sort over all the feature, but this is the slowest aspect of this feature selector.

As to other near linear-time methods, recent results show that a Bayes classifier can easily be converted into a feature pruner. We call this algorithm CLIFF [41]. It is detailed here as yet another example of what can be done with a supposedly naive Naive Bayes classifier.

CLIFF discretizes all numeric data (using a 10% chop) then counts how often each discretized range occurs in each class.

```python
Classes = all classes
for one in Classes:
    two = Classes - one
    c1 = |one|  # number of examples in class "one"
    c2 = |two|  # all other examples
    for f in features:
        ranges = discretized(f)
        for range in ranges:
            n1, n2 = frequency of range in one, two
            r1, r2 = n1/c1, n2/c2
            f.ranges[range].value[one] = r1*r1/(r1+r2)
```

The equation on the last row rewards ranges that are:
- More frequent in class *one* (this is the $r1^2/r1$ term);
- And which are relatively more frequent in class *one* than classes *two* (this is in the fraction).

A range with high value is powerful in the sense that it is frequent evidence for something that selects for a particular class. In CLIFF, we say that:

- The power of a range is maximum of its values over all classes (as calculated by the above);
- The power of a feature is the maximum power of all its ranges.
CLIFF can prune features by discarding those with least power. Further, within each feature, it can prune ranges with lower power. CLIFF is very simple for implement. Once a programmer has built a Naive Bayes classifier and a discretizer, CLIFF is 30 lines or less in a high-level language such as Python. Our experience with this algorithm is that it can convert large data sets into a handful of most powerful ranges that can be discussed with a user.

7.1.2 Other Feature Pruning Methods

As to more complex polynomial methods like:

- PCA: Principle Component Analysis; invented in 1901 [40];
- or LSI: Latent Semantic Indexing; invented in 1990 [12].

are based on synthesizing a small number of most-informative features. The key to this synthesis is the rotation of the higher dimensions into lower dimensions. For example, consider the data shown at right. Clearly, for this data, the standard (x,y) co-ordinate system is less informative than a new synthesized dimension that runs through the middle of the data. Formally, we say that this two-dimensional data can be approximated by a rotation into one-dimension along a synthesized dimension. Such rotations are quite standard in software engineering. For example, Nagappan & Ball & Zeller [39] using principle components analysis (PCA) to learn synthesized dimensions. When we applied their technique to defect prediction, we found that we could find one simple rule within 24 features of a defect data set:

\[
\text{defective if domain1 greater than 0.317}
\]

Here, \(\text{domain1}\) is a dimension synthesized by PCA that is a linear combination of the 24 features in the data set:

\[
\begin{align*}
\text{domain1} &= 0.241 \times \text{loc} + 0.236 \times v(g) + 0.222 \times e(g) + 0.236 \times t(v) + \\
&+ 0.241 \times a + 0.238 \times a + 0.086 \times d + 0.199 \times e + \\
&+ 0.236 \times b + 0.221 \times t + 0.241 \times \text{loCode} + 0.179 \times \text{loComment} + \\
&+ 0.221 \times \text{loBlank} + 0.158 \times \text{loCodeAndComment} + 0.163 \times \text{uniqOp} + \\
&+ 0.234 \times \text{uniqOpnd} + 0.241 \times \text{totalOp} + 0.241 \times \text{totalOpnd} + \\
&+ 0.236 \times \text{branchCount}
\end{align*}
\]

The important thing of this equation are the weights on the features. These weights range from 0.158 (for number of lines of code and comment) to 0.241 (for many things including the number of branches inside this code). That is, PCA is telling us that for the purposes of predicting defects, code branching is \(0.241 / 0.158 = 1.5\) more important than counts of lines of code and comment.

Under-the-hood, PCA builds \(N\) domains like the \(\text{domain1}\) shown above using a matrix-oriented approach. Standard-PCA uses the correlation matrix where cell \(i, j\) is the correlation between feature \(i\) and \(j\). It then sorts the eigenvector of that matrix decreasing on their eigenvalue (so \(\text{domain1}\), shown
above, would have been the first eigenvector in that sort)\(^9\).

LSI is another matrix-oriented method [12] for dimensionality reduction. LSI decomposes a matrix of data \(D\) into three matrices \(U,S,V\) which can be combined in order to regenerate \(D\) using \(D = USV^T\). The middle matrix \(S\) is special:

- The non-diagonal elements of \(S\) are zero;
- The diagonal elements of \(S\) are the weights of each feature;
- The rows of \(S\) are sorted in descending order by this weight.

A smaller data set can now be generated by removing the \(k\) lowest rows in \(S\) and the \(k\) most right columns in \(V\) (i.e. that data relating to the least interesting features). We denote the truncated \(V\) matrix as \(V'\). Now the document \(D_i\) can be described using the fewer number of features in \(V'_i\).

### 7.2 Row Pruning

Science seeks general principles; i.e. a small number of factors that appear in many examples. For data sets that support such generality, we should therefore expect that a large number of rows are actual echoes of a smaller number of general principles.

Row pruning is the art of finding exemplars for those general principles. Just like with feature pruning, a repeated result is that most data sets can be pruned back to a small percentage of exemplars (also known as prototypes). For example, Chang’s prototype generators [9] replaced training sets of size \((514, 150, 66)\) with prototypes of size \((34, 14, 6)\) (respectively). That is, prototypes maybe as few as \((7,9,9)\)% of the original data.

If we prune a data set in this way, then all subsequent reasoning runs faster. Also, performance quality can improve since we have removed strange outlier examples.

There are many ways to do row pruning (also called instance selection). This is an active area of data mining [20]. For example, Peter Hart [24] proposes an incremental procedure where, starting with a random selection of the data, if a new test case is misclassified, then it is deemed to be different to all proceeding examples. Hart recommends selecting just those novel instances. Kocaguneli prefers a clustering approach, followed the deletion of all clusters that would confuse subsequent inference (i.e. those with the larger variance or entropy of any numeric or discrete class variable) [28].

Hart’s and Kocaguneli’s methods are much slower than a newer methods based on CLIFF [41]. Recall that CLIFF finds \textit{power} ranges; i.e. those ranges

---

\(^9\) It turns out that WHERE (described above in §4.5) is a heuristic method for finding the first domain of PCA. But whereas PCA takes polynomical time, WHERE runs in linear time. For more on the relationship of WHERE to PCA, see [42].
that tend to select for a particular class. If a row contains no power ranges, then it is not an interesting row for deciding between one class and another. To delete these dull rows, CLIFF scores and sorts each row by the product of the power of the ranges in that row. If then returns the top 10% scoring rows. Note that this procedure runs in linear time, once the power ranges are known.

8 Text Mining

Up until now, this chapter has considered well-structured tables of examples. Most real-world data does not appear in such well-structured tables. One study\textsuperscript{10} concluded that:

\begin{itemize}
  \item 80 percent of business is conducted on unstructured information;
  \item 85 percent of all data stored is held in an unstructured format (e.g. unstructured text descriptions);
  \item Unstructured data doubles every three months.
\end{itemize}

That is, if we can tame the text mining problem, it would be possible to reason and learn from wide range of naturally occurring data.

Text mining data has a different “shape” to the tables of examples discussed above. Consider the text of this chapter. At the time of writing this sentence, it contains 1574 unique words in 304 paragraphs and each chapter has 25 words (median). That is, if we represented this document as one example per paragraph, then:

\begin{itemize}
  \item Each row would be 1574 features wide;
  \item Each row would have entries for around 25 cells. Which is another way of saying that this table would be $1 - \frac{25}{1574} = 98.4\%$ empty.
\end{itemize}

For such mostly-empty data sets, before we can apply machine learning, we have to prune that empty space lest our learners get lost in all that nothingness.

That, is the essential problem of text mining is dimensionality reduction. There are several standard methods for dimensionality reduction such as tokenization, stop lists, stemming, $Tf*Idf$.

8.1 Tokenization

Tokenization replaces punctuation with spaces. Optionally, tokenizers also send all uppercase letters to lower case.

\textsuperscript{10} http://www.b-eye-network.com/view/2008
8.2 Stopping

Stop lists remove “dull” words found in a “stop list” such as the following:

- a
- about
- across
- again
- against
- almost
- alone
- along
- already
- also
- although
- always
- an
- among
- amongst
- amount
- an
- an
- and
- another
- any
- anyhow
- anyone
- anything
- anyway
- anywhere
- are
- around
- as
- at

There are many on-line lists of stop words but before you use one, you should review its contents. For example, the word “amount” is in the above stop list. However, in an engineering domains, the word “amount” might actually be vital to understanding the units of the problem at hand.

Note also that stop word removal may be done in a procedural manner. For example, in one domain, we find it useful to stop all words less than four characters long.

8.3 Stemming

Stemming removes the suffixes of words with common meaning. For example, all these words relate to the same concept.

- CONNECT
- CONNECTED
- CONNECTING
- CONNECTION
- CONNECTIONS

Porter’s stemming algorithm [43] is the standard stemming tool. It repeatedly replies a set of pruning rules to the end of words until the surviving words are unchanged. The pruning rules ignore the semantics of a word and just perform syntactic pruning:

http://www.ranks.nl/resources/stopwords.html
Porter’s stemming algorithm has been coded in many languages\textsuperscript{12} and comes standard with many text mining toolkits. The algorithm runs so quickly that there is little overhead in applying it.

### 8.4 Tf*Idf

Tf*Idf is shorthand for “term frequency times inverse document frequency”. This calculation models the intuition that jargon usually contains technical words that appear a lot, but only in a small number of paragraphs. For example, in a document describing a spacecraft, the terminology relating to the power supply may be appear frequently in the sections relating to power, but nowhere else in the document.

Calculating Tf*Idf is a relatively simple matter. If there are $Words$ number of document and each word $I$ appear $Word[I]$ number of times inside a set of $Documents$ and if $Document[I]$ be the documents containing $I$, then:

$$Tf*Idf = \frac{Word[i]}{Words} \times \log \left( \frac{Documents}{Document[i]} \right)$$

In this equation, the $\log$ is used to stop the inverse document frequency growing so large (for some entries) that it dominates all over entries.

There are three important aspects of $Tf*Idf$ that deserve our attention:

- It takes linear time to compute (so it scales to large data sets) and that computation is very simple to code (one implementation of the author, in a high-level scripting language, was less than 20 lines long).

---

\textsuperscript{12} http://www.tartarus.org/martin/PorterStemmer
• When applied to natural language texts, it is often than case that the the majority of words have very low $Tf*Idf$ scores. That is, we just use the (say) to 100 terms (as ranked by $Tf*Idf$) then we can prune away vast numbers of uninformative words.

### 8.5 Other Text Mining Methods

The above sequence is summarized as follows:

$$\text{Tokenize} \rightarrow \text{Stop} \rightarrow \text{Stem} \rightarrow Tf*idf$$

Note that this sequence takes linear time (so its scales to very large data sets). For complex domains, other methods may be required such as the $InfoGain$, PCA, and LSI methods discussed above. For example, a standard implementation is to use the $V'$ matrix (discussed above with LSI) and find nearby words using cosine similarity (discussed below, see Equation 1). For a catalog of state-of-the-art text mining methods, see the other chapters in this book.

### 9 Nearest-Neighbor Methods

*Model-based methods* such as decision tree learners first build some model (e.g. a decision tree), then use that model to reason about future examples.

Another approach is *instance-based reasoning* that reasons about new examples according to their nearest neighbors. This style of learning is called *lazy learning* since nothing happens till the new text example arrives. This lazy approach can be quite slow so it is customary to add a pre-processor step that clusters the data. Once those clusters are known, then it is faster to find similar examples, as follows:

• Find the nearest cluster to the new test example;
• Ignore everything that is not in that cluster;
• Find similar examples, but only looking in that cluster.

Clustering is an *unsupervised* learning algorithm in that, during clustering, it ignores any class feature. This is a very different approach to nearly all the algorithms above, which were *supervised*; i.e. they treated the class feature in a way that was special and different to all the other features. The only exception was APRIORI that does not treat any feature different to any other (so it is *unsupervised*).

A core issue within clustering is how to measure the distance between rows of examples. For example, some use a kernel function to compute a
weighted distance between rows. Much could be said on these more complex approaches but presenting that material would be a chapter all on its own. Suffice to say that with a little column pruning, somethings very simple functions suffice [30].

For example consider the overlap defined for discretized data where the distance between two rows is the number of ranges that occur in both rows. Note that this overlap measure scales to very large data sets, using a reverse index that records what ranges occur in what rows. Given that reverse index, simple set intersection operators on the reverse index can then quickly find:

- If any two rows have no shared ranges (so distance = infinity);
- Otherwise, the distance between two rows is the number of shared ranges.

This measure is used by McCallum and others [34] in their work on canopy clustering (discussed below) as well as the W2 instance-based planner [4]. W2 accepts as input the context that is of interest to some manager; i.e. a subset of the data features that mention a subset of the feature ranges. W2 is discussed in Figure 5.

Two other distance measures, that make more use of the specific distances between values are cosine disimilarity and the Minkowhs distance. Consider two rows in the database $x$ with $y$ and features $1, 2, 3, ..., f$ etc. In cosine disimilarity, the distance is least when the angle between two rows is zero. This can be calculated using the following equation:

W2 is an example of nearest neighbor inference using a simple overlap distance measure:

- A manager might pose the question “what is the best action for projects with programmer capability equals low or very low and the database size is large or very large”.
- W2 finds the $K$ projects nearest this context (sorted by their overlap).
- Next, for the first time in all this processing, W2 looks at the class variable which might contain information (say) about the development time of the projects. It then divides the projects into the $k_1$ projects it likes (those with lowest development effort) and the $k_2$ is does not (and $K = k_1 = k_2$).
- Next, it sorts the ranges by the value of each range (as defined above in discussion of CLIFF in the above section on feature pruning).
- Lastly, it conducts experiments where the first $i$ items in that sorted ranges of values are applied to some hold-out set.

As output, W2 prints its recommendations back to the manager. That manager is a list of things to change in the project. That list contains the first $i$ terms in the list of sorted items that select for projects in the hold out set with least (say) development time.

---

This text is a natural representation of the document content. No hallucinations were detected.
The Minkowski distance is really a family of distance measures controlled by the variable \( p \):

\[
dist(x, y) = \left( \sum_{i} (w_{i}|x_{i} - y_{i}|^{p})^{1/p} \right)
\]  

(2)

Here, \( w_{i} \) is the some weight given to each feature (and the the larger this weight, the more importance is given to that feature). At \( p = 1 \), then this is the Manhatten (or city-block) distance. At \( p = 2 \), then this is the standard Euclidean distance measure you probably studied in high school.

This Euclidean distance is used in David Aha's classic paper on instance-based reasoning [1]. Aha's method allows us to handle missing data as well as data sets with both numeric and non-numeric features. This measure begins by normalizing all numerics min to max, 0 to 1. Such normalization has many benefits:

- **Normalization lets us compare distances between numerics of different scale.** To see why this is an issue, consider a database which lists rocket speeds and astronaut shoe sizes. Even if an astronaut shoe size increases by 50% from 8 to 12. That difference would be lost when compared to rocket speeds (that can range from zero to 41*10^6 meters per second). If we normalize all numerics zero to one, then a 100% change in shoe size (that can range from zero to 20) will not be lost amongst any changes to the rocket speed.

- **Normalization let us compare distances between numeric and discrete features.** Aha offers the following distance measure for non-numerics: if two non-numeric values are the same, then their separation is zero; else, it is one. That is, the maximum difference for non-numerics is the same maximum difference for any normalized numeric value. If we combine numeric normalization with Aha’s rule, then we can compare rows that contain numeric and non-numeric features.

Aha also offers a method for handling missing data. The intuition behind this method is that if a value is missing, assume the worst case and return the maximum possible value. To implement that, Aha recommends the following procedure. When finding the difference between two values \( x_{i} \) and \( y_{i} \) from feature \( i \):

- If both values are missing, return the maximum distance (assuming normalized data, then this maximum value is one).
- If the feature is non-numeric:
  - If one is absent, return one.
  - Else if values are the same, return zero.
- Else, return one.

- If the feature is numeric then:
  - If only one value is present, then return the largest of $\text{value}$ and $\text{value1}$.
  - Otherwise return $x_i y_i$.

Now that we can compute $x_i - y_i$, we can use the Euclidean measure $\sqrt{\sum w_i * (x_i - y_i)^2 / \sum w_i}$ where $w_i$ is some weight that defaults to one (but might otherwise be set by the feature pruning tools described above). Note that, if we use normalized values for $x_i$ and $y_i$, then this value returns a number in the range zero to one.

Note that for small data sets, it is enough to implement this distance function, without any clustering. The standard k-th nearest neighbor algorithm generates predictions by finding the $k$ rows nearest any new test data, then reporting the (say) median value of the class variable in that sample.

Returning now to clustering, this is a large and very active area of research [25, 26]. The rest of this chapter discusses a few fast and scalable clustering algorithms.

## 10 Some Fast Clustering Methods

DBScan [16] use some heuristic to divides the data into neighborhoods. For all neighborhoods that are unclustered, it finds any one with enough examples in an adjacent neighborhood (where enough is a domain-specific parameter). A cluster is then formed of these two neighborhood and the process repeats for any neighbors of this expanding cluster.

Canopy clustering [34] a very fast distance measure used extensively at Google to divide data into groups of nearby items called canopies. Then it perform more elaborate (and more CPU expensive) analysis, but only within these canopies.

Farnstrom (who we mentioned above) proposed a modification to the standard K-means algorithm. These modifications allow for incremental learning of clusters. In standard K-means, $K$ rows are picked at random to be the centroids. All the other rows are then labelled according to which row is nearest. Each centroid is then moved to the middle of all the rows with that label. The process repeats until the centroid positions stabilize. Note that K-means can be slow since it requires repeated distance measurements between all centroids and all rows. Also, it demands that all the rows are loaded into RAM at one time. Farnstrom fixes both these problems with his simple single pass $k$-means (SSK) algorithm:

- Work through the data in batches of size, say, 1% (ideally, selected at random).
- Cluster each batch using K-means.
• For each new batch, call K-means to adjust the clusters from the last batch for the new data.
• New data is only added to the old clusters if it is not anomalous (as defined by the Färnstrom anomaly detector, mentioned above).
• In theory, this might result in some old centroid now have no data from the new 1% of the data. In this case, a new centroid is created using the most distant point in the set of anomalous data (but in practice, this case very rarely happens).

Another fast clusterer based on K-means is Mini-Batch K-means [47]. Like SSK, this algorithm process the data in batches. However this algorithm is much simpler than SSK:
• Each centroid maintains a count \( v \) of the number of rows that where its nearest neighbor.
• The data is read in batches of size \( M \).
• The fewer the rows that used that centroid, the more it must be moved to a new position. Hence, after each batch has updated \( v \):
  - Compute \( n = 1/v \);
  - For each centroid:
    - Recall all rows \( r \) in the batch that found this centroid to be its nearest neighbor
    - For each such row, move all values \( c \) in that towards \( r \) by an amount \( (1 - n) * c + nr \) (note that large \( v \) implies small \( n \) which translates to “do not move the heavily used centroids very much”).

Sculley reports that this approach runs orders of magnitude faster than standard k-means [47]. Also, it is an incremental algorithm that only needs RAM for the current set of centroids and the next batch. But how to determine the right number of centroids? One method, which we adapt from the the GENIC incremental cluster [21], is to pick a medium number of cluster then, after each prune:
• Let each cluster contain \( c \) items.
• Find and delete the \( x \) dull centroids with \( c/C * N < \text{rand}(\)\), where \( C \) is the sum of all the \( c \) values from the \( N \) clusters.
• If \( x > C/4 \) then set \( x \) to \( C/4 \).
• Select any \( x \) rows at random and add them to the set of centroids. Note that this either replaces dull centroids or adds new centroids if we do not have enough.

Note that the Färnstrom approach and Mini-Batch K-means satisfy the famous Data Mining Desiderata [3]. According to that decree, a scalable data miner has the following properties:
• Require one scan (or less) of the database if possible: a single data scan is considered costly, early termination if appropriate is highly desirable.
On-line “anytime” behavior: a “best” answer is always available, with status information on progress, expected remaining time, etc. provided.

- Suspensible, stoppable, resumable; incremental progress saved to resume a stopped job.
- Ability to incrementally incorporate additional data with existing models efficiently.
- Work within confines of a given limited RAM buffer.
- Utilize variety of possible scan modes: sequential, index, and sampling scans if available.
- Ability to operate on forward-only cursor over a view of the database. This is necessary since the database view may be a result of an expensive join query, over a potentially distributed data warehouse, with much processing required to construct each row (case).

Once the clusters are created, then it not uncommon practice to apply a supervised learner to the data in each cluster. For example:

- NBTree uses a tree learner to divide the data, then builds one Naive Bayes classifier for each leaf [31].
- WHERE applies the principle of envy to clustered data. Each cluster asks “who is my nearest cluster with better class scores than me?” Data mining is then applied to that cluster and the resulting rules are applied back to the local cluster [35]. WHERE built those rules using a more intricate version of W2, described above.

There are several advantage to intra-cluster learning:

- The learner runs on fewer examples: For example, WHERE builds a tree of clusters whose leaves contain the square root of the number of examples in the whole data set. For any learner that takes more than linear time to process examples, running multiple learners on the square root of the data is faster than running one learner on all the data.
- The learner runs on a more homogeneous set of examples: To increase the reliability of the predictions from a data miner, it is useful to build the learner from similar examples. By learning on a per-cluster basis, a learner is not distracted by dissimilar examples in other clusters. In the case of WHERE, we found that the predictions generated per-cluster where much better than those found after learning from all the data (and by “better” we mean lower variance in the predictions and better median value of the predictions.).
11 Where to Learn More

This chapter has been a quick overview of a range of data mining technology. If the reader wants to read further than this material, then the following material may be of interest:

- The many excellent data mining texts such as Witten and Frank text on data mining [48] or all the learning-related sections of the Norvig and Russell text [46].
- The proceedings of the annual Mining Software Repositories and PROMISE conference.

Also, for a sample of state-of-the-art data mining methods, see the other chapters in this book.

References


