Chapter 10

Data Mining (Under the Hood)

The last three chapters listed *application areas* of data mining in software engineering. This chapter discusses the *internals of a data miner*. In particular, it answers the question "just what is data mining?".

This chapter is meant for data mining novices. Hence, our running example will be a very simple database (15 examples of "when to play golf").

Figure 10.1 Horse, carved from stone, 15,000 BC. From wikipedia.org/wiki/Sculpture; license: goo.gl/C4byU4.

10.1 Data Carving

According to the Renaissance artist Michelangelo di Lodovico Buonarroti Simoni:

> *Every block of stone has a statue inside it and it is the task of the sculptor to discover it.*

While sculptors like Michelangelo carved blocks of marble, data scientists carve into blocks of data. So, with apologies to Señor Simoni, we say:

> ~~*Every*~~ *Some* ~~*stone*~~ *databases have* ~~*statue*~~ *a model inside and it is the task of the* ~~*sculptor*~~ *data scientist to* ~~*discover*~~ *check for it.*

Enter data mining. Decades of research has resulted in many automatic ways to wield the knife that slices and refines the data. All these data miners have the following form:

1. Find the crap (where "crap" might also be called "superfluous details");
2. Cut the crap;
3. Go to step 1

For example, here is a table of data. 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).

```
# golf data, version1.

outlook  , temp , humidity , windy , play
-------- , ---- , -------- , ----- , -----
overcast , 64   , 65       , TRUE  , yes
rainy    , 65   , 70       , TRUE  ,  no
rainy    , 68   , 80       , FALSE , yes
sunny    , 69   , 70       , FALSE , yes
rainy    , 70   , 96       , FALSE , yes
rainy    , 71   , 91       , TRUE  ,  no
overcast , 72   , 90       , TRUE  , yes
sunny    , 72   , 95       , FALSE ,  no
rainy    , 75   , 80       , FALSE , yes
sunny    , 75   , 70       , TRUE  , yes
sunny    , 80   , 90       , TRUE  ,  no
overcast , 81   , 75       , FALSE , yes
overcast , 83   , 86       , FALSE , yes
sunny    , 85   , 85       , FALSE ,  no
```

This chapter "cuts" into this data using a variety of pruning methods. The result will be a much smaller summary of the essential details of this data, without superfluous details.

Before that, some digressions:

- Later in this chapter, we will call the above table a *simple* data mining problem in order to distinguish it from other kinds of problems such as *text mining* or *Big Data* problems.
- We use the above table of data to illustrate some basic principles of data mining. That said, in practice, we probably need more data than shown above before daring to report any kind of conclusions to business users. Data mining is an *inductive* process that learns from examples and the above table is probably too small to support effective induction. For a discussion on how to learn models from very small data sets, see [304].
- The following notes do not survey *all* of data mining technologies since such a survey would fill many books. Rather, they just focus on the technologies used frequently in this book. If the reader wants to explore further, see [112, 452].

Enough digressions; back to the tutorial on data mining. Note that the following notes describe learning algorithms that could be applied to software engineering data, or indeed data from many other sources.

10.2 About the Data

Before we cut up the data, it is insightful to reflect on the structure of that data since different kinds of data miners work best of different kinds of data.

Data mining executes on *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.

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, *nearest neighbors* methods [112] may be best that makes conclusions about new examples by looking at their neighborhood in the space of old examples. Hence, such methods only needs a few (or even only one) similar examples to make conclusions. This book uses nearest neighbor methods many times. For example, Chapter 13 uses them as a *relevancy filter* to find pertinent from other organizations.

If a table has many goals, then some may be competing; e.g. it may not be possible to find a car with the twin goals of low cost and low miles per gallon. Such competing multi-goal problems can be studied using a *multi-objective optimizer* like the genetic algorithms used in NSGA-II [96] or the optimizers of Chapter 24.

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, using algorithms like k-means of EM [452] or the CHUNK tool of Chapter 12. An alternate approach, taken by the APRORI association rule learner [371], 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.

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 algorithms like C4.5 and CART [51]. However, with some clever sampling of the data, it is possible to scale these traditional learners to Big Data problems [53,72]. C4.5 and CART are both examples of *iterative dichotomization*, which is a general divide-and-conquer strategy that splits the data, then recurses on each split. For more on *iterative dichotomization*, see below (in *contrast pruning*).

10.3 Cohen Pruning

Now that we know what data looks like, we can cut it up.

There are many ways to prune superfluous details from a table of data. One way is to apply *Cohen pruning*:

Cohen pruning: Prune away small differences in numerical data.

For example, the users might know that they can control or measure data in some column c down to some minimum precision ϵ_c. With that knowledge, we could simplify all numerics by rounding them to their nearest ϵ_c. Without that knowledge, we might apply some domain general rule such as Cohen's rule that says values are different by a *small amount* if they differ by some fraction of the standard deviation σ. And, just to remind us all, the standard deviation of n numbers from some list X measures how much they usually differ from the mean value $X.\mu$; i.e.

$$\sigma = \sqrt{\frac{\sum_i^n (x_i - X.\mu)^2}{n-1}} \tag{10.1}$$

In the above table of data, the standard deviation of *temperature* and *humidity* in the above columns are $\sigma_{temperature} = 6.6$ and $\sigma_{humidity} = 10.3$ (respectively). If we round those numerics to $\sigma/2$ of those values (then round those values to the nearest integer), we get the following table:

```
# golf data, version2.
# Numerics rounded to half of the
# standard deviation in each column.

outlook  , temp , humidity , windy , play
-------- , ---- , -------- , ----- , -----
overcast , 62   , 67       , TRUE  , yes
rainy    , 66   , 72       , TRUE  ,  no
rainy    , 69   , 82       , FALSE , yes
sunny    , 69   , 72       , FALSE , yes
rainy    , 69   , 98       , FALSE , yes
rainy    , 72   , 93       , TRUE  ,  no
overcast , 72   , 93       , TRUE  , yes
sunny    , 72   , 93       , FALSE ,  no
rainy    , 76   , 82       , FALSE , yes
sunny    , 76   , 72       , TRUE  , yes
sunny    , 79   , 93       , TRUE  ,  no
overcast , 82   , 77       , FALSE , yes
overcast , 82   , 87       , FALSE , yes
sunny    , 85   , 87       , FALSE ,  no
```

In this case, the change is quite small but for other data sets, Cohen pruning can avoid silly discussions about (say) 10.00111 versus 10.003112. For more on how to avoid being distracted by *small effects*, see [206, 238, 440].

One frequently asked question about Cohen pruning is how big to select a good value for ϵ. Standard values are to use $\epsilon \in \{0.3, 0.5, 0.8\} * \sigma$ for *small, medium*, or *large* differences within a set of numbers. Note that the justification for those values is more "engineering judgment" than "rigorous statistical argument"[1].

[1]For a more rigorous approach to defining ϵ, use equations 2,3,4 from [206] to calculate the Hedges' "g" measure. Then, from Table 9 of that paper, use $g \leq \{0.38, 1.0, \infty\}$ to determine if some value is a *small, medium* or *large* difference.

10.4 Discretization

In theory, numbers might range from negative to positive infinity. In practice, there may be only a few important distinctions within that infinite range of values. As an example of this, consider the *age* values collected from humans. In theory, this number can range from 0 to 120 (or 0 to 200,000 if we include all *homo sapiens* from the fossil record). For most purposes it is enough to *discretize* those numbers into a small of bins such as *baby, infant, child, teenager, adult, old, dead*.

That is, one way to remove spurious details in a table is:

> *Discretization pruning: prune numerics back to a handful of bins.*

For example, in the above table, notice that:

- Up to *temperature=69*, nearly all the *play* variables are *yes*.
- Similarly, if we where to sort on *humidity*, we would see that up to *humidity=80*, that $\frac{7}{8}$ of the *play* values are *play=yes*.

Based on the above discussion, we might use discretization to divide our numerics into two bins *(lo,hi)* at the following cuts:

- If *temperature*≤ 69 then *temperate=lo* else *temperature=hi*.
- If *humidity*≤ 80 then *humidity=lo* else *humidity=hi*.

This results in the following data where the numeric data has are been replaced with symbols:

```
# golf data, version 3: numerics discretized

outlook  , temp , humidity , windy , play
-------- , ---- , -------- , ----- , -----
overcast , lo   , lo       , TRUE  , yes
rainy    , lo   , lo       , TRUE  ,  no
rainy    , lo   , hi       , FALSE , yes
sunny    , lo   , lo       , FALSE , yes
rainy    , lo   , hi       , FALSE , yes
rainy    , hi   , hi       , TRUE  ,  no
overcast , hi   , hi       , TRUE  , yes
sunny    , hi   , hi       , FALSE ,  no
rainy    , hi   , hi       , FALSE , yes
sunny    , hi   , lo       , TRUE  , yes
sunny    , hi   , hi       , TRUE  ,  no
overcast , hi   , lo       , FALSE , yes
overcast , hi   , hi       , FALSE , yes
sunny    , hi   , hi       , FALSE ,  no
```

10.4.1 Other Discretization Methods

In the above example, discretization reflected over some target class column in order to find useful breaks (and in the above data, *golf* is the class column). Formally speaking, the discretization of numerics based on the class variable is called *supervised discretization*. A widely-used supervised discretization method is the "Fayyad-Irani"

method [127] that uses information content to decide where to cut up numeric ranges. For more on the use of information content, see the next section.

The alternate to discretized supervision is *unsupervised discretization* that just reflects over the column of numbers without considering values from elsewhere. For example, in *equal-width discretization*, we round the data to the value nearest to $\frac{max-min}{bins}$. Alternatively, in *equal- frequency discretization*, we sort the numbers and divide them into $bins$ number of equal-sized intervals. In these approaches, a frequently-asked-question is how many *bins* should be used. In our experience, some small number between two and sixteen (median five) is often useful (and the best number for a particular data set requires some experimentation).

Rather than guess how many *bins* are required, it is possible to use the distribution of the numbers to infer where to insert the breaks. This approach, which we call MEANGAIN, recursively splits a list of numbers X into sublists Y and Z in order to most isolate the larger or smaller values. This requires reflecting over $X.\mu, Y.\mu.Z.\mu$ (i.e. the mean values seen *before* and *after* each split). After sorting the numbers, we recursively divide the data X of n items at position i to generate $Y = X[1..i]$ and $Z = Y[i+1..n]$. For that division, we seek the index "i" that most maximizes the expected value of the square of the differences in the means before and after the split. i.e.

$$\arg\max_i \left(\frac{i}{n}(X.\mu - Y.\mu)^2 + \frac{n-i}{n}(X.\mu - Z.\mu)^2 \right)$$

This recursion halts when:

- Either division is too small, where "small" might be less than 3 items in Y or Z;
- The means of the two splits differ by only a small amount; i.e. $|Y.\mu - Z.\mu| < \epsilon$. Recalling the above discussion in §10.3, we could set $\epsilon = 0.3\sigma$, where σ is the standard deviation of the entire column of data (note that in this procedure, ϵ is set once before starting the recursion).

If MEANGAIN is applied to the *temperature* and *humidity* columns, then we learn to break *temperature* at {69,76} and *humidity* at {87}.

As to which discretization method is best- that is always data set dependent. But note that seemingly more sophisticated methods are not necessarily better or, indeed, lead to different conclusions. For example, in the above discussion, we eye-balled the *temperature* and *humidity* numerics and proposed breaks at *temperature=69* and *humidity=87*. The rest of the chapter will use those manually generated breaks since they are not far off the breaks proposed by MEANGAIN.

For a good discussion on other discretization methods, see [110, 144, 461].

10.5 Column Pruning

Discretization can be the first step in another pruning process called column pruning or, more formally, *feature subset selection* [160].

A repeated observation is that most data sets have tightly correlated columns or columns containing noisy or irrelevant data [160, 240]. Consequently, in practice, N

columns can often be pruned back to $\sqrt{N}$ columns (or less) without damaging our ability to mine useful patterns from that data. Hence, we recommend:

Column pruning: prune away columns that are not redundant and/or noisy.

To find ignorable columns of numbers, we might apply supervised discretization and reflect on the results:

1. If discretization fails to find any breaks in the column, then that means this column does not influence the class variable (and can be ignored).
2. If discretization divides the data, and the distribution of the classes *within each break* is similar to the *original* class distribution, then dividing this column does not influence the class (and can be ignored).

This second rule can be generalized to include numeric and non-numeric columns. Once the numerics are discretized in a table with R rows, then all the columns are divided into ranges containing $r_1, r_2, ...$ rows. For each range i with r_i rows:

- Let those rows have $n_1, n_2, .., n_j$ examples of class $c_1, c_2, ..., c_j$ each with probability in this range of $p_j = n_j/r_i$.
- We can score those ranges using *entropy*, which is like the inverse of information content. Entropy measures the *disorder* of some part of the data so *less* entropy is *better*. The *entropy* of a range r_i is

$$e_i = -\sum p_j log_2(p_j) \tag{10.2}$$

The expected value E of the entropy of a column is the weighted sum of the entropy e_i in each of its ranges r_i; that is $E = \sum_i e_i r_i/R$. Now the *smaller* the entropy E, the *better* that column since it means that its ranges correspond most to particular classes. If we normalize all the column entropies to 0..1 for E_{min} to E_{max} then there is usually a small number of *best* columns with normalized $E < 0.25$ and many more *worse* columns that can be pruned.

The above procedure is usually called INFOGAIN [160] and has the advantage that, after the numerics have been discretized, it requires only two passes over the data. While one of the fastest known methods, other column pruners can perform better (albeit, slower). For more details, see [160, 240].

Later is this book, we present another column pruning method called CHUNK. CHUNK finds two distant rows and asks "what separates them?" by (a) drawing a line between the two rows and (b) mapping all the other points onto that line[2]. In doing so, it can be learned that some columns do not contribute to distinguishing distant rows. Such columns can then be ignored.

10.6 Row Pruning

Discretization can also be the first step towards pruning away irrelevant rows. This is interesting since, just as many columns can be safely pruned, so too is it possible to

[2]To be precise, it uses the cosine rule to project all the other points somewhat along that line. The details of that process need not concern us in this introductory chapter, but see the *fastdiv* procedure of Chapter 12 for more details.

prune away many rows. If that surprises the reader, then consider this:

- If data contains a model then that data contain multiple examples of the relationships within that model.
- This means that within the R rows, there exists $P < R$ rows that are *prototypes*; i.e. best examples of different aspects of the model.

That is, another way to prune data is:

> *Row pruning: Prune the rows in a table back to just the prototypes.*

This chapter presents two methods for finding prototypes (and for a long list of many other methods, see [351]). The first way is to *cluster* the data then replace each cluster with its *centroid* (the central point of that cluster). For more on that method, see the next section on *cluster pruning*.

Another way to implement row pruning is to (slightly) modify the INFOGAIN procedure, described above. Recall that INFOGAIN calculated the entropy e_i associated with each range r_i which covered n_i rows in each column c_k. Our simple row pruner, called RANGEGAIN, uses the same information. It scores each range in each column using:

$$S(c_k, r_i) = e_i(R - n_i)/R$$

where R is the total number of rows and $R = \sum_i r_i$.

Once we know the value of each range, we can then score each row using the sum of the scores S of the ranges in that row (and *smaller* scores are *better*). If we normalize all those row scores 0..1 , min to max, then there is usually a small number of *best* rows with a normalized score $S < 0.25$ and many more *worse* rows that can be pruned.

RANGEGAIN has all the advantages and disadvantages as INFOGAIN. It is very fast (only two passes over the data) and if INFOGAIN is already available, then there is little else to implement. On the other hand, like INFOGAIN, the RANGEGAIN procedure it is not a detailed analysis of the structure of the data. Hence, other row pruners [351] might perform better (albeit slower).

Note the similarities of this row pruner to the above column pruning procedure. There are deep theoretical reasons for this: Lipowezky [268] notes that column and row pruning are similar tasks since both remove cells in the hypercube of all rows times all column. When we prune rows or columns, it is like we are playing an accordion with the ranges. We can "squeeze in" or "pull out" that hypercube as required, which makes that range cover more or less rows and/or columns. Later in this book we will take advantage of this connection when the POP1 row pruner (in Chapter 17) is extended to become the QUICK column pruner (in Chapter 18).

If the reader wants more information on row pruning, then:

- For notes on other row pruners, see [351].
- For a discussion of a tool that uses *both* column *and* row pruning, see Chapter 15.
- For a discussion on the connection of row pruning to privacy (that uses a variant of the RANGEGAIN procedure), see Chapter 16.
- For a discussion on the connection of row pruning to clustering, see the next section.

10.7 Cluster Pruning

Discretization, column pruning and row pruning are like a fine-grained analysis of data that prunes some parts of a table of data. *Clustering* is a more coarse-grained approach that finds very large divisions in the data.

Clusters are groups of similar rows. Once those clusters are known, this can simply how we reason over data. Instead of reasoning over R rows, we can reason over C clusters- which is very useful if there are fewer clusters than rows; i.e. $C < R$. That is, another way to remove spurious information about data is:

> *Cluster pruning: Prune many rows down to a smaller number of clusters containing similar examples.*

There are many different clustering methods [452], including the CHUNK algorithm of Chapter 12. When clustering, some *distance* function is required to find similar rows. One reason for discretizing data is that simplifies finding those distances. Firstly, in the case of discretized data, that distance function can be as simple as the number of column values from the same ranges. Secondly, a simple reverse index *from* column range *to* row id means that it is very fast to find some *row2* with some overlap to *row1*.

One other thing about clustering: it is usual to *ignore* the class variable (in this case, *play*) and cluster on the rest. If we CHUNK on those non-class columns, then we find the following four clusters:

```
# golf data, version 4: clustered.
# Note that 'c' is the cluster id.

c, outlook,temp,humid,wind,play
--------------------------------
1,    rainy, lo, lo,  TRUE,  no
1, overcast, lo, lo,  TRUE, yes
1, overcast, hi, lo, FALSE, yes
--------------------------------
2,    sunny, lo, lo, FALSE, yes
2,    sunny, hi, lo,  TRUE, yes
2, overcast, hi, hi,  TRUE, yes
2,    sunny, hi, hi,  TRUE,  no
--------------------------------
3, overcast, hi, hi, FALSE, yes
3,    sunny, hi, hi, FALSE,  no
3,    sunny, hi, hi, FALSE,  no
--------------------------------
4,    rainy, hi, hi,  TRUE,  no
4,    rainy, hi, hi, FALSE, yes
4,    rainy, lo, hi, FALSE, yes
4,    rainy, lo, hi, FALSE, yes
```

To find the pattern in these clusters, we ask what ranges are found in one cluster, but not the others. It turns out if you first decide a value for *humidity* and then decide a value for *outlook*, you can build a predictor for cluster membership. We will return to that point in the next section on *contrast pruning*.

The above clusters can be used to reduce the golfing data to four prototypes. For each cluster, we write one *centroid* from the majority value in each column (and if there is no clear majority, we will write "?"):

```
# golf data, version 5: clusters replaced with the centroids
# Note that 'c' is the cluster id.

c, outlook,temp,humid, windy, play
----------------------------------
1,overcast,  lo,   lo,  TRUE, yes
2,   sunny,  hi,    ?,  TRUE, yes
3,   sunny,  hi,   hi, FALSE,  no
4,   rainy,   ?,   hi, FALSE, yes
```

That is, to a first-level approximation, we can prune the original golfing data to just four rows. Just as an aside, this kind of prototype generation is not recommended for something as small as the golfing data. However, for larger data sets with hundreds of rows or more, show in Chapter 15 that that CHUNKing R rows down to $\sqrt{R}$ clusters is an effective method for defect and effort estimation.

10.7.1 Advantages of Prototypes

There are many advantages of reasoning via prototypes. For example, they are useful for handling unusual features in the data, If we collapse each cluster to one prototype (generated from the central point in that cluster), then each prototype is a report of the average effects within a cluster. If we restrict the reasoning to just those average effects, we can mitigate some of the confusing effects of noisy data.

Also, prototype-based reasoning can be faster than reasoning about all the rows [75]. For example, a common method for classifying new examples is to compare them to similar older examples. A naive approach to finding those nearest neighbor reasoning is to search through all rows. However, if we index all the rows according to their nearest prototype, then we can find quickly find nearest neighbors as follows:

1. For all prototypes, find the nearest neighbor.
2. For all rows associated with that prototype, find the nearest neighbors.

Note that step 2 is sometimes optional. In Chapter 15, we generate effect detect predictions by extrapolating between the known effort/defect values seen in their two nearest prototypes. In that approach, once the prototypes are generated, we can ignore the original set of rows.

Other benefits of prototypes are compression, anomaly detection, and incremental model revision:

- *Compression:* Suppose we generate, say, $\sqrt{N}$ clusters and keep only say, $M = 10$ rows from each cluster. This approach means we only need to store some fraction $M/\sqrt{N}$ of the original data.
- *Anomaly detection:* When new data arrives, we can quickly search that space to find its nearest neighbors. If those "nearest neighbors" are unusually far away, then this new data is actually an anomaly since it is an example of something *not* seen when during the initial clustering.
- *Incremental model revision:* If we keep the anomalies found at each prototype then if that number gets too large, then that would a signal to reflect over all those anomalies and the M examples to, say, generate new clusters around those

examples and anomalies. This approach means we rarely have to reorganize all the data- which can lead to very fast local incremental updates of a model [150].

Finally, another benefit of prototype learning is implementing privacy algorithms. If we reduce R rows to (say) $\sqrt{R}$ centroids then, by definition, we are ignoring the particular details or $R - \sqrt{R}$ individuals in the data. This has implications for sharing data, while preserving confidentiality (see Chapter 16).

Note that the above benefits of using prototypes can only be achieved if the cost of finding the prototypes is not exorbitant. While some clustering methods can be very slow, tools like CHUNK run in near-linear time.

10.7.2 Advantages of Clustering

Apart from prototype generation, there are many other advantages of cluster-based reasoning. For one thing, when exploring new data sets, it is good practice to spend some initial time just looking around the data for interesting or suspicious aspects of the data. Clustering can help such an unfocused search since it reduces the data to sets of similar examples.

Also, clusters can be used to impute missing data. If a clustered row has some value missing in a column, then we might guess a probably replacement value using the most common value of the other columns in that cluster.

Further, clusters are useful for reducing uncertainty in the conclusions reached from data. Any special local effects (or outliers) that might confuse general reasoning can be isolated in their own specialized cluster [295]. Then, we can reason about those outliers in some way that is different to the rest of the data.

More generally, clustering can find regions where prediction certainty *increases* or *decreases*. Raw tables of data contain many concepts, all mixed up. If we first cluster data into regions of similar examples, then it is possible that our prediction certainty will *increase* To see this, we apply Equation 10.2 to the original golf data with $\frac{9}{14} \approx 64\%$ examples of *play=yes* and $\frac{5}{14} \approx 36\%$ examples of *play=no*. This has entropy $e_0 = 0.94$. In the four clusters shown above (see version 4 of the golf data), the entropy of clusters 1,2,3,4 are $e_{1,2,3,4} = \{0.92, 0.81, 0.92, 0.92\}$, respectively. The rows in these clusters comprise different fractions $f_{1,2,3,4} = \frac{3}{14}, \frac{4}{14}, \frac{3}{14}, \frac{4}{14}$ (respectively) of the data. From this, we can see that the expected value of entropy of these clusters is $\sum e_i f_i = 0.86$, which is less than the original entropy of 0.94. This is to say that clustering has clarified our understanding of the data by finding a better way to divide the data.

On the other hand, clustering can also find regions where prediction certainty *decreases*. Suppose our clustering had generated the following cluster:

```
# golf data, imagined cluster

outlook,temp,humid,wind,play
----------------------------
  sunny, hi, hi, FALSE,  no
  rainy, hi, hi,  TRUE,  no
  rainy, hi, hi, FALSE, yes
  rainy, lo, hi, FALSE, yes
```

The problem with the this cluster is that is no longer clear if we are going to play golf (since the frequency of *play*=*yes* is the same as *play*=*no*). For such a cluster, Equation 10.2 tells us that the entropy is $-2 * 0.5 * \log_2(0.5) = 1$. Since this us more than $e = 0.94$, then we would conclude that in this cluster the entropy is *worse* than in the original data. This is a *confusing cluster* since, if we entered this cluster, we would be *less* about what sports we play that if we just stayed with the raw data.

How to handle such *confusing clusters*? One approach is to declare them "no go" zones and delete them. The TEAK system of Chapter 14 takes that approach: if regions of the data are found with high conclusion variance, then those regions are deleted. TEAK then runs clustering again, but this time on just the examples from the non-confusing regions. This proves to be remarkable effective. TEAK can look into data from other organizations (or from very old prior projects) and find the small number of relevant and non-confusing examples that are relevant to current projects.

10.8 Contrast Pruning

The universe is a complex place. Any agent recording *all* their stimulation will soon run out of space to store that information. Lotus founder Mitchell Kapor once said, "Getting information off the Internet is like drinking from a fire hydrant." We should take Kapor's observation seriously. Unless we can process the mountain of information surrounding us, we must either ignore it or let it bury us.

For this reason, many animals react more to the *difference* between things rather than the *presence* of those things. For example, the neurons in the eyes of frog are most attuned to the *edges* of shapes and the *changes* in the speed of their prey [94]. As with frogs, so too with people. Cognitive scientists and researchers studying human decision making note that humans often use simple models rather than intricate ones [148]. Such simple models often list *differences* between sets of things rather than extensive *descriptions* of those things [209].

If we listen to frogs and cognitive scientists, then it seems wise to apply the following pruning method to our data:

> *Contrast pruning: Prune away ranges that do not contribute to differences within the data.*

To illustrate this point, we use the C4.5 *Iterative dichotomization* algorithm [369] to learn the distinctions between the clusters shown above (in version 4 of the golf data).

C4.5 recursively applies the INFOGAIN procedure to find the best column on which to split the data. Recall that INFOGAIN sorts the columns based on their entropy (and *lower* entropy is *better*). C4.5 then splits the data, once for every range in the column with *lowest* entropy. C4.5 then calls itself recursively on each split.

C4.5 returns a decision tree where the root of each sub-tree is one of the splits. For example, we applied C4.5 to version 4 of the cluster data (ignoring the *play* variable) calculating entropy using the cluster id of each row). The result is shown below:

```
# rules that distinguish different clusters
#
#                     : conclusion (all, missed)
-------------------------------------------------
humidity = lo
|   outlook = overcast : cluster1   (  2,     0)
|   outlook = rainy    : cluster1   (  1,     0)
|   outlook = sunny    : cluster2   (  2,     0)
humidity = hi
|   outlook = rainy    : cluster4   (  4,     0)
|   outlook = overcast : cluster2   (  2,     1)
|   outlook = sunny    : cluster3   (  3,     1)
```

In this tree, indentation indicates the level of tree. Also, anything after ":" is a conclusion reached by one branch of the tree. For example, the branch in the first two rows of this tree can be read as:

IF *humidity=lo* AND *outlook=overcast* THEN *cluster1*

Induction is not certain process and sometimes, we cannot fit all the data to our models. Hence, after every conclusion, ID counts *all* the examples fell down each branch as well how many *missed* on being in the right leaf. For most of the conclusions shown above, all the examples fall down branches that place them into the correct clusters. Hence, their *missed* number is zero. However, for two last branches, there were some issues. For example, of the 3 examples that match to *humidity=hi and outlook=rainy*, two of them are in *cluster3* but one actually comes from *cluster2*. Hence the number on this last branch are $3/1$.

Now here's the important point: notice what is *not* in the above tree? There is no mention of *temperature* and *windy* since, in terms of distinguishing between structures in our data, that data is superfluous. Hence, if were engaging business users in discussions about this data, we would encourage them to discuss issues relating to *humidity* and *outlook* before discuss issues relating to the less relevant columns of *temperature* and *wind*.

More generally, contrast pruning is a more refined version of column pruning. Contrast pruning selects the ranges that help us distinguish between different parts of the data. Whereas column pruning ranks and removes all ranges in uninformative *columns*, contrast pruning can rank *ranges* to select some parts of some of the columns.

In the case of the golf data, the contrast ranges shown in the above rules use all the ranges of *humidity* and *outlook*. But in larger data sets, a repeated result is that the contrast sets are some small subset of the ranges in a small number of columns [300]. For example, Figure 10.2 shows the 14 columns that describe 506 houses from Boston (and the last column "PRICE" is the target concept we might want to predict). If we discretize the housing pricing to $\{$*low, medlow, medhigh, high*$\}$ then C4.5 could learn the predictor for housing prices shown in Figure 10.3. For this discussion, the details of that tree do matter. What does matter are the *details* of that tree. C4.5 is a zealous data miner that looks for numerous nuanced distinctions in increasingly smaller and specialized parts of the data. Hence, this tree mentions nearly all of the columns (all except for *ZN* and *B*).

On the other hand, if we use contrast pruning then we can generate models that are much smaller than Figure 10.3. To do this, we first assign some *weights* to the classes

Figure 10.2 Columns in the Boston *housing* table.
From http://archive.ics.uci.edu/ml/datasets/Housing.

- 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
- PTRATIO: 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.

such that higher priced houses are increasingly more desired:

$$w_{low} = 1 \qquad w_{medlow} = 2$$
$$w_{medhigh} = 4 \qquad w_{high} = 8$$

After discretizing all numerics, we then divide the houses into {*low, medlow, medhigh, high*} and weight each range by its frequency in that range times the w_i value for that division. Those range scores are then normalized 0..1 and we build rules from just the top quarter scoring ranges (i.e. with normalized range score over 0.75). Starting with those ranges, the TAR3 contrast set learner [299] learns the following selector for houses with large resale value:

$$(12.6 \leq PTRATION < 16) \cap (6.7 \leq RM < 9.78) \tag{10.3}$$

Figure 10.4 shows the effects of this rule. The red,green bars shows the distributions of houses before,after applying Rule 10.3. Note that the houses selected in the green bars are mostly 100% *high* priced houses; i.e. the learned rule is effective for selecting desired subsets of the data..

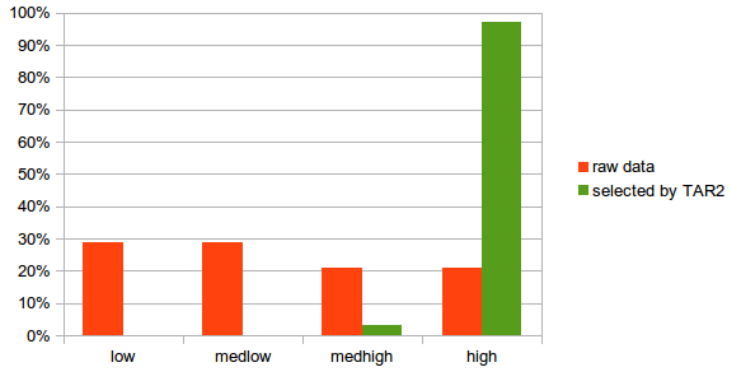
It turns out, in many data sets, the pruning offered by contrast sets can be very large- which is to say that in many domains, here are a small number of most powerful ranges and many more ranges that are superfluous (and can be safely ignored) [300]). For example, the model learned after contrast pruning on the housing data was one line long and referenced only two of the 13 possible observable columns. Such succinct models are quick to read and audit and apply. On the other hand, models learned without pruning can be more complicated to read and and harder to understand; e.g. Figure 10.3.

For notes on other contrast set learners, see [348].

Figure 10.3 A decision tree to predict housing prices.

```
LSTAT <= 14.98
 |   RM <= 6.54
 |   |   DIS <= 1.6102
 |   |   |   DIS <= 1.358: high
 |   |   |   DIS > 1.358
 |   |   |   |   LSTAT <= 12.67: low
 |   |   |   |   LSTAT > 12.67: medlow
 |   |   DIS > 1.6102
 |   |   |   TAX <= 222
 |   |   |   |   CRIM <= 0.06888: medhigh
 |   |   |   |   CRIM > 0.06888: medlow
 |   |   |   TAX > 222: medlow
 |   RM > 6.54
 |   |   RM <= 7.42
 |   |   |   DIS <= 1.8773: high
 |   |   |   DIS > 1.8773
 |   |   |   |   PTRATIO <= 19.2
 |   |   |   |   |   RM <= 7.007
 |   |   |   |   |   |   LSTAT <= 5.39
 |   |   |   |   |   |   |   INDUS <= 6.41: medhigh
 |   |   |   |   |   |   |   INDUS > 6.41: medlow
 |   |   |   |   |   |   LSTAT > 5.39
 |   |   |   |   |   |   |   DIS <= 3.9454
 |   |   |   |   |   |   |   |   RM <= 6.861
 |   |   |   |   |   |   |   |   |   INDUS <= 7.87: medhigh
 |   |   |   |   |   |   |   |   |   INDUS > 7.87: medlow
 |   |   |   |   |   |   |   |   RM > 6.861: medlow
 |   |   |   |   |   |   |   DIS > 3.9454: medlow
 |   |   |   |   |   RM > 7.007: medhigh
 |   |   |   |   PTRATIO > 19.2: medlow
 |   |   RM > 7.42
 |   |   |   PTRATIO <= 17.9: high
 |   |   |   PTRATIO > 17.9
 |   |   |   |   AGE <= 43.7: high
 |   |   |   |   AGE > 43.7: medhigh
LSTAT > 14.98
 |   CRIM <= 0.63796
 |   |   INDUS <= 25.65
 |   |   |   DIS <= 1.7984: low
 |   |   |   DIS > 1.7984: medlow
 |   |   INDUS > 25.65: low
 |   CRIM > 0.63796
 |   |   RAD <= 4: low (13.0)
 |   |   RAD > 4
 |   |   |   NOX <= 0.655
 |   |   |   |   AGE <= 97.5
 |   |   |   |   |   DIS <= 2.2222: low
 |   |   |   |   |   DIS > 2.2222: medlow
 |   |   |   |   AGE > 97.5: medlow
 |   |   |   NOX > 0.655
 |   |   |   |   CHAS = 0: low
 |   |   |   |   CHAS = 1
 |   |   |   |   |   DIS <= 1.7455: low
 |   |   |   |   |   DIS > 1.7455: medlow
```

Figure 10.4 Distributions of houses, before and after applying Rule 10.3.



10.9 Goal Pruning

Our final pruning levers the known goal for the learning process. For example, if the goal is to learn when we might play golf, then we could report just a summary of that data that affects that decision. That is:

> *Goal pruning: Prune away ranges that do not effect final decisions.*

For example, what if had the goal "what predicts for playing for golf"? This question has been made very simple, thanks to our pruning operators:

1. Cohen pruning has simplified the numerics;
2. Discretization has collapsed the simplified numerics into a few ranges;
3. Clustering has found that the data falls into four groups;
4. Contrast pruning has found a minority of ranges and columns that can distinguish between those groups (which are all the ranges in *humidity* and *outlook*).

From all this pruning, we now run C4.5 looking for ways to separate the classes *play=yes* and *play=no*. For that run, we only use the ranges found in the contrast pruning:

```
# golf data, version 6
# Only discertized ranges found
# in contrasts between clusters.

outlook  , humidity , play
-------- , -------- , -----
overcast ,  lo      ,  yes
rainy    ,  lo      ,   no
rainy    ,  hi      ,  yes
sunny    ,  lo      ,  yes
rainy    ,  hi      ,  yes
rainy    ,  hi      ,   no
overcast ,  hi      ,  yes
sunny    ,  hi      ,   no
rainy    ,  hi      ,  yes
sunny    ,  lo      ,  yes
sunny    ,  hi      ,   no
overcast ,  lo      ,  yes
overcast ,  hi      ,  yes
sunny    ,  hi      ,   no
```

When given to C4.5, this yields the following tree.

```
#                  : conclusion (all, missed)
-----------------------------------------------
outlook = overcast : yes        (4,        0)
outlook = rainy    : yes        (5,        2)
outlook = sunny
|   humidity = lo  : yes        (2,        0)
|   humidity = hi  : no         (3,        0)
```

Observe how all this pruning has lead to a very succinct model of golf-playing behavior. For 4+5=9 of the rows, a single test on *outlook* is enough to make a decision. It is true that for 2+3=5 rows, the decision making is slightly more complex (one more test for *humidity*). That said, the tree is usually very accurate:

- In three of its four branches, there are no *missed* conclusions
- In the remaining branch, errors are in the minority (only 2 of 5 cases).

10.10 Extensions for Continuous Classes

Most of the examples in the chapter focuses on predicting for discrete classes, e.g. symbols like *play=yes* and *play=no*. Another category of learners used in this book are learners that predict for continuous classes. There are many such learners such as linear regression, support-vector machines, etc, but the one we use most is a variant of the C4.5 iterative dichotomization algorithm.

Recall that C4.5 recursively divides data on the columns whose ranges have the lowest entropy. This approach can be quickly adapted to numerical classes by switching the entropy equation of Equation 10.2 with the standard deviation equation of Equation 10.1. Like entropy, the standard deviation is *least* when we know *most* about that distribution (since lower standard deviations means we have less uncertainty). This is the approach taken by the "regression tree" learners such as the CART algorithm used in Chapters 17, 18, and 22.