Insightful Code:
Common Lisp edition
(a.k.a. "ICCLE")

by the Mountain LISP Gang and Tim Menzies (ed)

/tmp/revision.tex /tmp/revision.tex

September 13, 2008

Contents

I Preface **9**

1 What is ICCLE? **11**
- 1.1 Reporting Bugs . . . 12
- 1.2 Join the Gang . . . 12
- 1.3 License . . . 12
- 1.4 Acknowledgments . . . 12

II Introducing Insightful Code **13**

2 What's It All About? **15**
- 2.1 Insightful Code . . . 15
- 2.2 What's Different about Insightful Code? . . . 16
- 2.3 What are the Components of Insightful Code? . . . 16

3 Data Mining **17**
- 3.1 A Simple Example . . . 17
- 3.2 Other Kinds of Learning . . . 20
- 3.3 Biases . . . 26
- 3.4 So Why Study Data Mining? . . . 29

4 Tasks **33**
- 4.1 What are "Tasks"? . . . 33
- 4.2 Star Trek, Episode 80 . . . 35
- 4.3 Data Mining as Tasks . . . 37
- 4.4 Enough . . . 38

III Tips (for Beginners) **41**

5 Setting up **43**
- 5.1 Before you Begin . . . 43
- 5.2 Joining, Downloading, Installing, Testing . . . 44

6 Using ICCLE **47**
6.1 ICCLE Makefile Utilities 47
6.2 Sharing the Code 48
6.3 Unit Tests 51

7 What to Read **53**
7.1 LISP 53
7.2 GNUPLOT 54
7.3 LATEX 54

IV Tricks (for Programmers) **55**

8 Using The Code Base **57**
8.1 Directories and Code 57
8.2 Loading ICCLE 58
8.3 Dependencies 58

9 Unit Testing : deftest **65**

10 Basic Tricks **69**
10.1 Arrays 69
10.2 Hash Tables 69
10.3 Strings 72
10.4 Maths 74
10.5 Lists 76
10.6 Matrix 78
10.7 Random 79

11 Some Useful Destructs **81**
11.1 Tree 81
11.2 Counted 81
11.3 Caution 82
11.4 Line 82

12 Advanced Tricks **87**
12.1 Debugging Tricks 87
12.2 OS Tricks 89

13 Statistical Functions **91**
13.1 Visualizations with “Distograms” 91
13.2 Plotting Tricks 93
13.3 Performance Statistics 96
13.4 Hypothesis Testing 102

V Tools (for Data Mining) 113

14 Storage 115
14.1 Building Tables 115
14.2 Data 120
14.3 Xindex 122

15 Discretization 129

VI Tasks (for Supporting the Business) 131

VII Tests (of Different Tools) 133

VIII Traps (for the unwary) 135
15.1 Subversion 137

IX To do (What's Next?) 139

X Team (About Us) 141

16 Teams 143
16.1 Team 0: Cradle 143

XI Theory (Only if you Care) 145

List of Figures

3.1 Contact lens data. 18
3.2 PRISM 19
3.3 Iterative dichomotization 21
3.4 Six clusters within some data. 22
3.5 K-Means 22
3.6 Clusters found by EM, grouped by expert 23
3.7 Kernel Trick 25
3.8 A decision tree from HOUSING data 25
3.9 Minimal contrast sets from HOUSING data 26
3.10 Different biases. From [FF05]. 27
3.11 Performance results statistically similar 28
3.12 Comparing results from many learners. From [DP97]. 29
3.13 Results from the same data sets as Figure 3.11 31

4.1 A simple hierarchy of knowledge-level tasks. 34
4.2 A more complex hierarchy of knowledge-level tasks. 34
4.3 Inside the diagnosis task. From [vHA96]. 34
4.4 Regions in data space. 36

8.1 CODE: `lib/icicle.lisp` 58
8.2 CODE: `lib/icicle.asd` 58
8.3 CODE: `lib/tests.asd` 59
8.4 CODE: `lib/tricks.asd` 60
8.5 CODE: `lib/store.asd` 61
8.6 CODE: `lib/toolslearn.asd` 62
8.7 CODE: `lib/toolspre.asd` 63

9.1 CODE: `lib/tests/deftest.lisp` 67
9.2 CODE: `lib/tests/tests.lisp` 68
9.3 CODE: `lib/tests/tricks.lisp` 68
9.4 CODE: `lib/tests/tools.lisp` 68

10.1 CODE: `lib/tricks/array.lisp` 70
10.2 CODE: `lib/tricks/hash.lisp` 71

10.3 CODE: lib/tricks/strings.lisp . . . 72
10.4 CODE: lib/tricks/align.lisp . . . 73
10.5 CODE: lib/tricks/maths.lisp . . . 75
10.6 CODE: lib/tricks/list.lisp . . . 77
10.7 CODE: lib/tricks/matrix.lisp . . . 78
10.8 CODE: lib/tricks/random.lisp . . . 80

11.1 CODE: lib/tricks/tree.lisp . . . 81
11.2 CODE: lib/tricks/counted.lisp . . . 82
11.3 CODE: lib/tricks/caution.lisp . . . 83
11.4 CODE: lib/tricks/line.lisp . . . 85

12.1 CODE: lib/tricks/debug.lisp . . . 88
12.2 CODE: lib/tricks/os.lisp . . . 90

13.1 CODE: lib/tricks/dist.lisp . . . 92
13.2 CODE: lib/tricks/distogram.lisp . . . 94
13.3 A plot generated by GNUPLOT. . . . 95
13.4 CODE: lib/tricks/gnuplot.lisp . . . 97
13.5 ROC sheet . . . 98
13.6 CODE: lib/tricks/abcd.lisp . . . 99
13.7 CODE: lib/tricks/abcdmatrix.lisp . . . 101
13.8 CODE: lib/tricks/abcdtest.lisp . . . 102
13.9 Selecting statistical functions . . . 103
13.10CODE: lib/tricks/ttest.lisp . . . 104
13.11CODE: lib/tricks/mannwhitney.lisp . . . 106
13.12CODE: lib/tricks/wilcoxon.lisp . . . 108
13.13CODE: lib/tricks/wilcoxon2.lisp . . . 109
13.14CODE: lib/tools/pre/fastmap.lisp . . . 110
13.15CODE: lib/tools/pre/kmeans.lisp . . . 111

14.1 Contact lens data. . . . 116
14.2 CODE: lib/store/unknown.lisp . . . 116
14.3 CODE: lib/store/header.lisp . . . 117
14.4 CODE: lib/store/table.lisp . . . 118
14.5 CODE: lib/store/tableselectors.lisp . . . 119
14.6 CODE: lib/store/data.lisp . . . 121
14.7 CODE: lib/store/crossindex.lisp . . . 124
14.8 CODE: lib/store/xindex.lisp . . . 125
14.9 CODE: lib/store/f.lisp . . . 126
14.10CODE: lib/store/xindexselectors.lisp . . . 127

Part I

Preface

Chapter 1

What is ICCLE?

ICCLE is a teaching experiment where senior year and graduate data mining students were asked to co-write a data mining textbook. To seed the text, students were given:

- Some data mining code written in LISP;
- Some 1990s knowledge engineering theory (about cognitive tasks);
- A challenge re-express that theory in terms of data mining tools and techniques;
- An example textbook to emulate (Peter Seilebl's Practical Common Lisp).
- A skeleton Latex document comprising dozens of little files. Student projects could then perform dozens of little projects contributing:
  - Tutorial introduction (motivation for what is to come);
  - Tips, for beginners;
  - Tricks, for programmers;
  - Tools, for data mining;
  - Tasks, for supporting the business;
  - Tests, of different tools;
  - Traps, for the unwary;
  - To do (what's next?);
  - Team (about us)
  - Theory (only if you care)

  Note that "tips" and much of "tricks" relate to general programming issues. Hard-core data mining does not begin till "tools".

Students shared all code in a SUBVERSION repository, licensed under GPL3.0.

Share and enjoy, but don't treat this material too seriously (yet). This is the first year of ICCLE and it is anticipated that it will benefit from a few years of development and maturation.

1.1 Reporting Bugs

If you find bugs in this code, please post them to `http://code.google.com/p/iccle/issues/list`.

1.2 Join the Gang

If you want to join the gang fixing the bugs and writing the next version of this book, then send a GOOGLE email to one of the ICCLE project owners (see the list, bottom right, of `http://code.google.com/p/iccle`).

1.3 License

1.4 Acknowledgments

Tim Menzies wishes to thank Peter Norvig, Paul Graham, and Peter Seibel for offering samples of how good a textbook on programming can be. Also, thanks to Bart Massey for his on-going example of the power of open source development ("Tim... use the source..."). And extra thanks to Helen Burgess for her wonderful support during this project.

Part II

Introducing Insightful Code

Chapter 2

What's It All About?

"Too many doings, not enough learnings."

When was the last time your code added insight to your business?

For example, how often do your business users interrupt the code demo to phone the CEO?

"Wendy," they shout down the line, "we have to change the western division. The programmers have just shown us a way to optimizing production by 30%."

Then they storm out of the demo, spilling coffee and tipping over chairs, in their race to reorganize the business.

This is a very rare event and this is our fault. We geek too much. After matching brackets for days/ weeks/ years, all our beautiful code achieves only low-level programming goals and not the high-level business goals they are meant to serve.

2.1 Insightful Code

We need more insightful code- systems that can:

- can learn from experience.
- that can reveal to business users new and exciting knowledge.

Current data mining methods are often not insightful since they are focus on their internal algorithmic details, rather than their impact on business goals. Also, some of those methods may be needlessly complex. As Janez Demsar[1] observes:

- In many business situations, data miners should not solve the problem. Rather, they should help people solve the problem. And when helping people, more complex methods like neural networks and support vector machines are not as useful as, say, simple symbolic classifiers like classification trees and naive Bayesian classifier with appropriate visualizations.

[1] `http://www.ailab.si/janez/`

- No classifier can beat a good scatter plot- just show the data appropriately and let the user decipher what it tells. Forget about the fancy modeling techniques: we are the best hardware with the best software there is. Computers are only good at drawing and at searching through possible visualizations, so let's use them for that.

We therefore seek a simple code library that offers new insights to the business users about their business. To do this, we combine data mining with the cognitive patterns seen at the business-level:

- The business goals are expressed as a cognitive tasks; e.g. monitor, predict, construct, etc. This is the whole story but if contains holes : numerous details not specified in the high level story.
- Data mining is used to fill in the holes; i.e. learn the details of the high-level tasks that tune it to a particular organizational context.

2.2 What's Different about Insightful Code?

Our library is different from standard data mining toolkits.

- Decades of data mining research has shown us how to build effective data miners using very simple components.
- Here, we exploit that research to build simple components and show how they can service the business users' cognitive tasks.

Consequently, insightful coding is:

- Model-light: the high-level cognitive tasks;
- And data-heavy: we assume that an organization has enough data available to tunes its business model. And if this data is absent, we will build a high-level simulator of the business to generate the data needed for this approach.

This is different to standard business modeling practices.

- Such standard modeling practices are model-heavy: many people, brainstorming complex business models.
- But data-light: very little feedback from the real world to refine, maintain, and optimize those models.

In standard software design, modeling is a mostly-manual early life cycle activity with very little support for model updates. Insightful coding, on the other hand, offers much support for model maintenance. Models are maintained from day one and are continually maintained through-out the life cycle.

2.3 What are the Components of Insightful Code?

The next two chapters describe two concepts define insightful coding: data mining and tasks.

Chapter 3

Data Mining

Data miners find diamonds in the dust. In the21^{st} century, any Internet user can easily access a mountain of data. But just because you can *access* data, does not necessarily mean you can *understand* it. Buried within that mountain data might be *useful* knowledge but, without data mining support, it may be invisible.

This chapter describes some useful data mining tricks, tools, and concepts. By the end of the chapter, you'll know enough about data mining to impress people at parties and succeed in round one of a job interview for a data mining position.

More seriously, by the time this chapter is over, you'll have a sense of the space of options inside a data mining and maybe even start thinking about the design of new data miners.

Since our goal is offering beginners a "quick start" in data mining, we will focus more on techniques that are effective and relatively simple. For more elaborate data mining methods, see many texts (e.g. [DHS00, WF05]) or (my current personnel favorite) the Markov logic work of Richardson and Domingos [RD06][1].

Students of data mining should note all the underlined terms- they are important concepts that you will need to thoroughly understand.

3.1 A Simple Example

Data miners build walls around regions of interest in feature space. Think of feature space as a chess board with bits of data piled on each square. Unlike chess boards, feature spaces are rarely two-dimensional. Rather, they a n-dimensional hypercubes that are difficult to visualize. The goal of data mining is to offer insight into how the data is spread around feature space.

For example, Figure 3.1 shows numerous examples of who wore what kind of contact lens. If spread around feature space, what patterns can be found?

[1] For an excellent video describing learning and Markov logics, see `http://www.uwtv.org/programs/displayevent.aspx?rID=3900&fID=810`

This data, called *contact-lens* has five features and can be used to build predictors that target, say, what kind of lens you should buy. If you look closely at the data you can see that in the four examples where we need a *hard* lens then the feature *astigmatism* has the range *yes* in $\frac{4}{4}$ examples. We can therefore extract the rule with the test:

```
R1 if astigmatism=yes then lens=hard
```

Now we can take a ruler and a pen and cross out all the examples with $astigmatism = yes$ and $lens = hard$. The remaining examples can be explored recursively looking for the next best range that predicts for some target range. In the remaining data, the next best rule looks like:

```
R2 if astigmatism=no then lens=soft
```

After that, everything is all of one class *none*, so the final predictor is:

```
(defun contact-lens ()
  (data
   :name 'contact-lens
   :columns '(age            perscription astigmatism tear-production lens)
             ;-------------- ------------ ----------- --------------- ----
   :egs    '((young          myope        yes         normal          hard)
             (young          hypermetrope yes         normal          hard)
             (presbyopic     myope        yes         normal          hard)
             (pre-presbyopic myope        yes         normal          hard)
             ;-------------- ------------ ----------- --------------- ----
             (young          hypermetrope no          reduced         none)
             (young          hypermetrope yes         reduced         none)
             (pre-presbyopic hypermetrope yes         reduced         none)
             (pre-presbyopic hypermetrope yes         normal          none)
             (young          myope        no          reduced         none)
             (young          myope        yes         reduced         none)
             (presbyopic     myope        no          reduced         none)
             (presbyopic     myope        no          normal          none)
             (presbyopic     hypermetrope yes         reduced         none)
             (presbyopic     hypermetrope yes         normal          none)
             (presbyopic     myope        yes         reduced         none)
             (pre-presbyopic hypermetrope no          reduced         none)
             (pre-presbyopic myope        no          reduced         none)
             (pre-presbyopic myope        yes         reduced         none)
             (presbyopic     hypermetrope no          reduced         none)
             ;-------------- ------------ ----------- --------------- ----
             (pre-presbyopic myope        no          normal          soft)
             (pre-presbyopic hypermetrope no          normal          soft)
             (young          myope        no          normal          soft)
             (young          hypermetrope no          normal          soft)
             (presbyopic     hypermetrope no          normal          soft))))
```

Figure 3.1: Contact lens data.

1. For all classes $c \in C$
2. Create a R with an empty condition that predicts for class C.
3. Until R is pure (or there are no more features) do
   (a) For each feature F not mentioned in R
      - For each value $v \in F$, consider adding $F = v$ to the condition of R
   (b) Select F and v to maximize $\frac{p}{t}$ where t is total number of examples of class C and p is the number of examples of class C selected by $F = v$. Break ties by choosing the condition with the largest p.
   (c) Add $F = v$ to R
4. Print R
5. Remove the examples covered by R.
6. If there are examples left, loop back to (1)

Figure 3.2: Learning rules via PRISM (favor rules that predict for majority classes).

```
     if astigmatism=yes then lens=hard
else if astigmatism=no  then lens=soft
else lens=non
```

Technically speaking, this a rule covering algorithm of the kind proposed by Cendrowska in his PRISM algorithm [Cen87][2]. Rule-covering algorithms delete examples covered by the rules, then recursively exploring the remaining data. In the general case, rule covering algorithms generate rules with multiple tests (not just the single tests of R1 and R2), each time honing in on just the examples selected by the tests. For more details on the PRISM algorithm, see Figure 3.2.

This simple example offered above illustrates many of themes that echo through out data mining research. For example, R1 and R2 could have been generated in any order since both are pure; i.e. they 'cover" one and only one target range. But R2 actually has more support than R1 (since it covers 5 examples, not 4). A reasonable heuristic when data mining is to favor models generated from well-supported parts of the data. If we'd followed that heuristic, then R2 would have been extracted before R1. Such a heuristic (of favoring models with greatest support) is an example of a search bias.

The flip side of that heuristic is to avoid models generated from under-supported parts of the data. That is, when building models, have some minimum support requirement that terminates learning before making spurious conclusions from too few examples. For example, if we tried to continued to build rules, but just for the *none* cases of our contact data, we would find that our naive rule-covering algorithm would start generating rules using just a single example. This is incorrect: generalizations should be, by definition, general and not come from just a handful of examples. Models learned from too little data are too susceptible to noise; i.e. spurious signals not connected to the output

[2]And refined by Gaines' INDUCT algorithm [Gai89] and Cohen's RIPPER algorithm [Coh95]; then rigorously analyzed by Furnkranz and Flach [FF05].

class. The symptoms of noise are:

- Overly complex theory
- Poorer performance on future examples

There are many ways to manage noise

- Pre-pruning: terminate when working with less than (say) four examples;
- Post-pruning: after a model is built, try throwing bits away; e.g. remove rules that use less than (say) four examples;
- Testing: test a model via hold-out sets; i.e. on data not used during training. (Technically speaking, this does not actually manage noise, but it does let us detect it. If the performance on hold-outs is radically different to the performance on the training, then the learner may have built models from noise.)

3.2 Other Kinds of Learning

Apart from rule covering algorithms, there are many different kinds of learners. The following list is hardly inclusive. Data mining is a very active area of research and there exist many mature, well-engineered, and well-studied algorithms that can be quickly applied to new data. It is standard for public domain data mining toolkits to contain 100s of different learners and pre-processors and visualization techniques. The trick is knowing what to apply and when (so keep reading...).

3.2.1 Decision Tree Learning

A decision tree learner would convert Figure 3.1, into a a tree whose leaves predict for target ranges.

```
tear-production = normal
|   astigmatism = yes
|   |   perscription = myope: hard
|   |   perscription = hypermetrope: none
|   astigmatism = no: soft
tear-production = reduced: non
```

The most famous tree learner is Quinlan's C4.5 algorithm [Qui92b] (also known as J48 [WF05]). C4.5 belongs to a class of *iterative dichomotization* algorithms defined in Figure 3.3.

3.2.2 Association Rules

Rule covering algorithm and decision tree learners predict for target ranges. Association rule learners, on the other hand, find patterns of co-occurring feature ranges (regardless whether they are target ranges, or not).

Given examples with mixtures of target ranges:

1. Split: Find a feature that most divides the examples into subsets and each subset with fewer mixtures (note: needs a measure of purity; i.e. mixed-up-ed-ness). The feature with the best splitting ranges becomes the root of a sub-tree.
2. Recurse: repeat for each subset
3. Stop when subsets too small to justify further splitting.
4. Condense the examples that fall into each lead
5. Prune back the generated tree (over-fitting avoidance)

Different tree learners result from different selections of the above underlined operators:

- CART: regression trees [BFOS84]
  - purity: standard deviation
  - condense: report the average of the instances in each bag.
- M5Prime: model trees [Qui92a]
  - purity: standard deviation
  - condense: generate a linear model of the form $a + b * x1 + c * x2 + d * x3 + ...$
- J48: decision trees [Qui92b]
  - purity: the "entropy" measure
  - condense: report majority class
- Random forests: by build just one tree? [Bre01]
  - purity: the "gini" index;
  - split: given M features, only consider splitting on $m \ll M$ randomly selected variables. Assess splits using the "gini" index.
  - prune: disabled
  - Repeat the above hundreds of times. Classify by seeking a majority vote amongst all trees. For more, see `http://stat-www.berkeley.edu/users/breiman/RandomForests/cc_home.htm`

Figure 3.3: Learning trees using iterative dichotomization.

```
tear-production=reduced                             ==> lens=none
perscription=myope tear-production=reduced          ==> lens=none
perscription=hypermetrope tear-production=reduced ==> lens=none
astigmatism=yes tear-production=reduced             ==> lens=none
astigmatism=no tear-production=reduced              ==> lens=none
lens=soft                                           ==> astigmatism=no
lens=soft                                           ==> tear-production=normal
tear-production=normal lens=soft                    ==> astigmatism=no
astigmatism=no lens=soft                            ==> tear-production=normal
lens=soft                                            ==> astigmatism=no tear-production=normal
```

Association rule learning research was sparked by the APRIORI learner of Agrawal et.al. [RTA93]

3.2.3 Clustering

Clustering algorithms return all the examples sorted into piles (where each pile contains examples that are closer to each other than items in other piles). For example, Figure 3.6 shows some housing data divided into six piles (i.e. the clusters) using the K-means algorithm of Figure 3.5. A generalization of K-means is the EM algorithm. In K-means, all examples belong to one and only

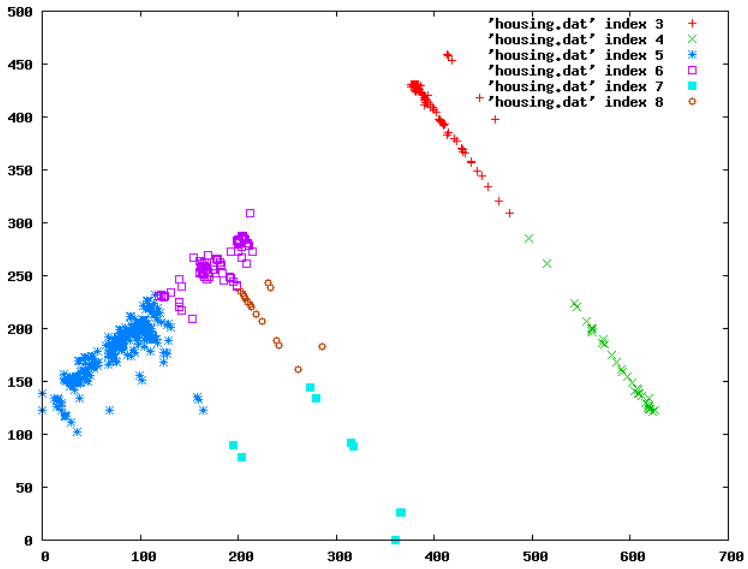

Figure 3.4: Six clusters within some data.

1. Pick K examples at random as centroids.
2. In each round, label each other example with its nearest centroid.
3. For all examples with the same label, computed the weighted mean position.
4. Move the centroid to that position.
5. Go to the next round.
6. Repeat till the centroids move very little from round i to round $i + 1$.

Figure 3.5: Learning clusters via K-MEANS.

one class. In EM, all examples belong to all clusters at some degree of (possibly very low) probability.

Note that clustering in a unsupervised learning method (makes no special use of the target variable). Rather than report that "this goes with that target", unsupervised learners report "this goes with that" where "this" and "that" are observables or controllables. All the other learning methods in this chapter (except for association rule learning) are supervised learners.

Supervised learning problems have the advantage that users can control the output of the learner by carefully engineering targets that reflect their goals and priorities. However, supervised learning also assumes that some supervisor has labeled the training data- which is not always possible. For example, try labeling every person in the world "I like them" or "I do not like them". You may not have the resources to complete that task.

It is possible to convert unsupervised problems into supervised ones:

- The unsupervised data can be clustered, then the cluster id be inserted into the data as a target variable. For example, Figure 3.6 (left-hand-side) shows data reflecting land slides in steep suburban areas. After clustering the data, a user can declare all the clusters to be one of three types: red for dangerous, blue for boring, and black denoting beach erosion. These regions are then grouped together to form the right-hand-side of Figure 3.6. These grouped regions can then become the targets for supervised learning (e.g. to learn features that predict for dangerous landslides).
- In active learning, users are offered a small sub-sample of the unsupervised data and asked to offer a target setting for each member of that sub-sample. What happens next depends on the particular active learning method. One possibility is that some predictor is built from the sub-sample and applied to the rest of the data. The user then reviews a sub-sample of the prediction and marks the correct decisions. These correct decisions get added to the learning set and the process repeats. Schohn and Cohn report that by carefully selecting what is shown to the user, then active learning on semi-supervised data can out-perform learning from fully supervised data [SC00].

3.2.4 Näive Bayes Classifiers

Näive Bayes classifiers don't even return a symbolic model. Rather, they summarize the frequency counts of the ranges seen in different targets, then offer predictions based on those ranges:

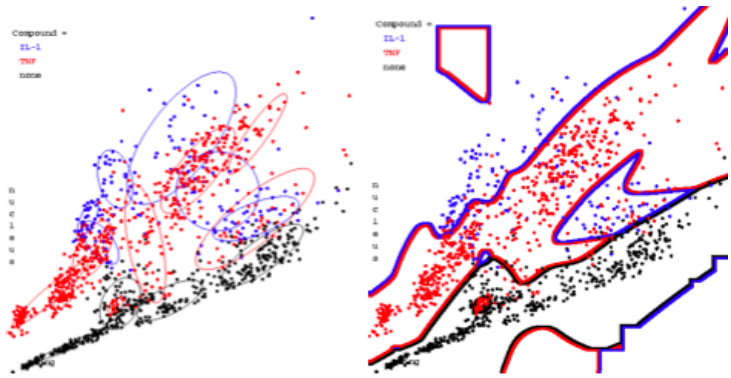

Figure 3.6: Clusters found by EM (left), grouped together by an expert (right).

```
Class none: Prior probability = 0.59

age:  Discrete Estimator. Counts =  5 7 6  (Total = 18)
perscription:  Discrete Estimator. Counts =  8 9  (Total = 17)
astigmatism:  Discrete Estimator. Counts =  9 8  (Total = 17)
tear-production:  Discrete Estimator. Counts =  4 13  (Total = 17)

Class soft: Prior probability = 0.22

age:  Discrete Estimator. Counts =  3 2 3  (Total = 8)
perscription:  Discrete Estimator. Counts =  3 4  (Total = 7)
astigmatism:  Discrete Estimator. Counts =  1 6  (Total = 7)
tear-production:  Discrete Estimator. Counts =  6 1  (Total = 7)
```

Näive Bayes classifiers are a very simple learning scheme with numerous advantages:

- Tiny memory footprint (just the frequency table and space for one or two current examples);
- Fast training (just update the frequency table)
- Fast testing
- Simplicity (50 lines of code, max)

These classifiers are called näive since they assume that features are all equally important and statistically independent (i.e. knowledge about the value of a particular feature doesn't tell us anything about the value of another attribute). Although these assumptions are rarely true, this scheme works well in practice for reasons we will return too later [DP97].

3.2.5 Regression

Regression equation learners that return equations such as:

$$a = b + k_1 * c + k_2 * d + ...$$

Note that such regression learners need numeric targets and prefer all the other features to be numeric as well.

3.2.6 Support Vector Machines

Support vector machines (SVM) separate examples with different target ranges by a "wall" (actually, a hyper-plane) that is as far way as possible from the nearest examples (called the support vectors. Once the hyper-plane is constructed, further processing need only concern itself with these support vectors (since the other examples are not "on the border" between different concepts).

Note that not all data is linearly separable by a hyper-plane into different target ranges in which case, SVMs have to handle soft boundaries. Alternatively, it is sometimes possible to redefine the examples such that they become separable using a kernel trick (e.g. Figure 3.7).

For more on SVMs, see `http://www.nature.com/nbt/journal/v24/n12/fig_tab/nbt1206-1565_F1.html` or `http://www.cs.iastate.edu/~honavar/hearst-svm.pdf`).

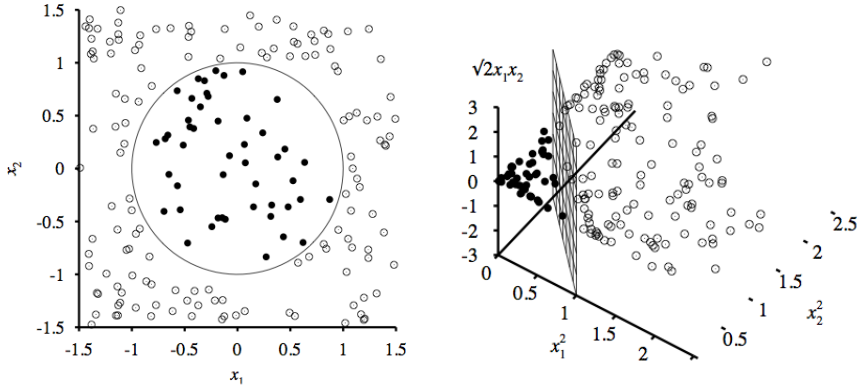

Figure 3.7: An example kernel trick that makes data linearly separable. The left plot displays $\{x_1, x_2\}$ and no straight line separates the dots from the circles. The right plot displays $\{x_1^2, x_2^2, \sqrt{2}x_1x_2\}$ and now the dots can be separated from the circles by a hyper-plane. Example from [RN95].

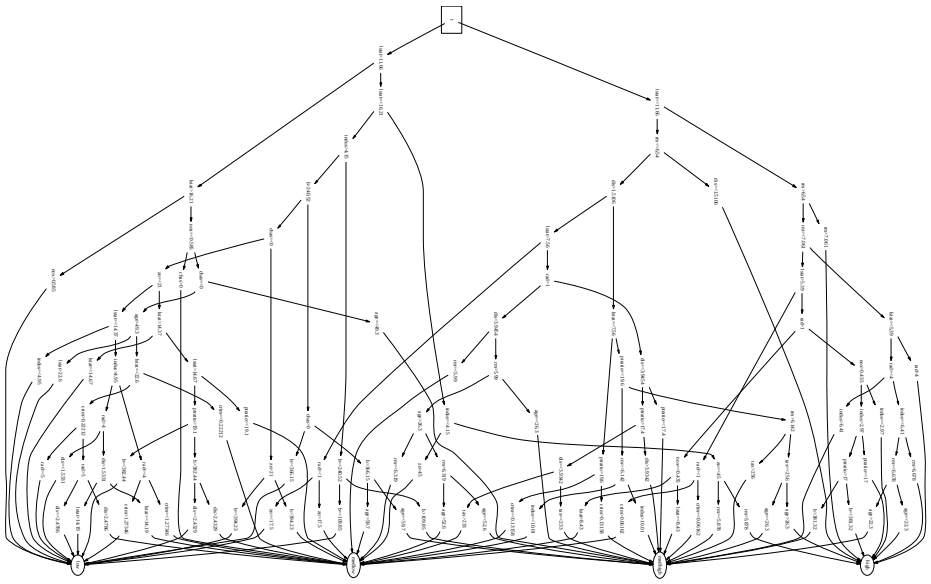

Figure 3.8: A decision tree learned from 506 cases in HOUSING example set from the UC Irvine repository [BM98]. Classes (at bottom), left-to-right denote purchase price: “high”, “medhigh”, “medlow”, and “low” This indicates median value of owner-occupied homes in $1000’s.

3.2.7 Contrast Set Learning

Contrast set learners are best described with an example. The decision tree of Figure 3.8 was learned from the same data that generated the contrast sets of Figure 3.9. While decision trees built elaborate descriptions of each target range, minimal contrast set learners like TAR3 [MH03, MH07] return just the smallest number of ranges that distinguish better ranges. When designing a controller, Figure 3.9 can be quite useful for specifying the least you need to do to achieve some desired result.

3.3 Biases

Different learners generate different models according to the biases. Some examples of search bias were given above (favor models with greatest empirical support).

Different learners have different ways to avoid over-fitting. Since each method biases what model is generated, this another kinds of bias called over-fitting bias.

Yet another bias is the language bias of the learner. Rule covering algorithms generate rules and there are many other target languages for models. For example, regression learners output equations; decision trees output (wait for it...) decision trees; contrast set learners out sets of differences between target ranges; bayes classifiers output distribution information on each class; etc. That is, a learner's language bias can radically change the what models are generated.

As shown in Figure 3.10, the space of (search*language*over-fitting avoidance) biases is quite large. You may be shocked at this- surely, you protest, we know which bias is best? Well, no. There is no universal "best" learner/model. Every learner/model has built in biases and these biases are essential to gen-

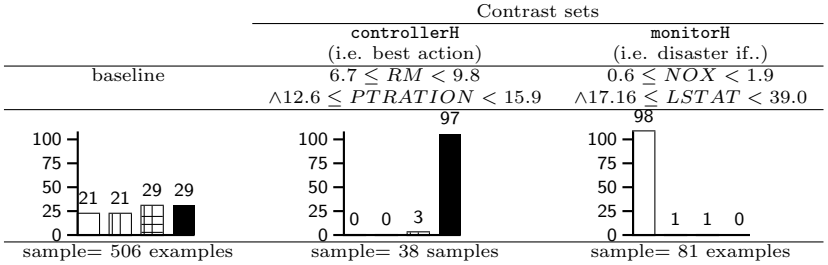

Figure 3.9: Minimal contrast sets learned from the same data used to generate Figure 3.8. This dataset has the class distribution shown in the bottom table, left-hand-side. KEY: LSTAT= lower status of the population; NOX= nitric oxides concentration (parts per 10 million); PTRATIO= pupil-teacher ratio by town; RM= average number of rooms per dwelling; KEY: low; medlow; medhigh; high.

	Language Bias							Search Bias							Overfitting Avoidance		
	Static					Dyn.		Algorithm				Strategy					
Algorithm	Selectors	Literals	Synt. Restr.	Rel. Clichés	Rule Models	Lang. Hier.	Constr. Ind.	Hill-Climbing	Beam Search	Best First	Stochastic	Top-Down	Bottom-Up	Bidirectional	Pre-Pruning	Post-Pruning	Integrated
AQ	×							×	×			×					
AQ15	×							×	×			×				×	
AQ17	×						×	×	×			×					
ATRIS	×							×			×			×		×	
BEXA	×							×	×			×			×	×	
CHAMP	×	×	×				×	×	×			×			×		
CiPF	×						×	×				×				×	
CN2	×							×	×			×			×		
CN2-MCI	×						×	×	×			×			×		
CLASS	×									×		×					
DLG	×							×	×				×				
FOCL	×	×		×				×				×			×		
FOIL	×	×	×					×				×			×		
FOSSIL	×	×	×	×				×				×			×		
GA-SMART	×	×		×	×						×	×			×		
GOLEM		×	×					×					×				
GREEDY3	×							×				×				×	
GRENDEL					×			×				×					
GROW	×							×				×				×	
HYDRA	×	×						×				×					
IBL-SMART	×	×		×						×				×	×		
INDUCE	×	×						×	×			×					
I-REP, I²-REP	×	×	×	×				×				×					×
JoJo	×	×						×						×			
m-FOIL	×	×	×					×	×			×			×		
MILP	×	×	×								×	×			×		
ML-SMART	×	×		×				×	×	×		×			×		
NINA					×	×		×					×				
POSEIDON	×							×	×			×				×	
PREPEND	×							×				×					
PRISM	×							×				×					
PROGOL	×	×	×							×		×					
REP	×	×		×				×				×				×	
RIPPER	×							×				×				×	×
RDT					×			×				×					
SFOIL	×										×	×			×		
SIA	×										×		×			×	
SMART+	×	×		×	×			×	×	×	×	×			×		
SWAP-1	×							×						×		×	
TDP	×	×	×	×				×				×			×	×	

Figure 3.10: Different biases. From [FF05].

eralization. When we generalize, we need to be able to declare that parts of the data/model are superfluous and can be discarded. Without biases, we don't know what to prune and what to keep. Without bias we can build predictors for future events. But with bias, we learn models particular to one way of looking

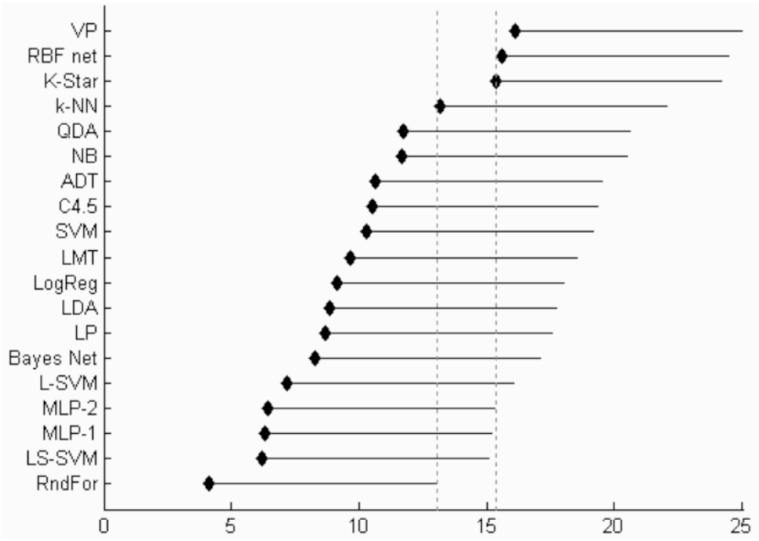

Figure 3.11: Vertical lines divide the results into regions where the performance results of models are statistically similar. All these results divide into three groups of differing performance. The left-hand-side 15 methods are statistically *insignificantly different*. From [LBMP09].

at the problem.

There are other biases to consider as well:

- Sample bias: We build models from the available examples, not the space of all possible examples. For example, this data only comes from the east coast and they do things differently over there.
- Evaluation bias: Given that different biases produce different models, which is the best one? To answer that, we need an evaluation bias that ranks the models.

Evaluation turns out to be quite tricky and there is much evidence that it is often done poorly [Dem06]. When evaluation is done properly, a repeated and humbling result is that many learners generate models whose performance is indistinguishable from other learners.

Figure 3.11 shows one such experiment by Lessman et.al. [LBMP09] with 19 learners trying to build predictors for faulty software modules. In that figure the vertical dashed lines divide the results into regions where the results are statically similar. For example, 15 of the learners shown there are indistinguishable.

In other results, Domingos and Pazzini compared several different learners on 28 data sets (see [DP97]). The performance scores are shown in Figure 3.12: note how they are very similar across all learners.

Table 1. Classification accuracies and sample standard deviations, averaged over 20 random training/test splits. "Bayes" is the Bayesian classifier with discretization and "Gauss" is the Bayesian classifier with Gaussian distributions. Superscripts denote confidence levels for the difference in accuracy between the Bayesian classifier and the corresponding algorithm, using a one-tailed paired t test: 1 is 99.5%, 2 is 99%, 3 is 97.5%, 4 is 95%, 5 is 90%, and 6 is below 90%.

Data Set	Bayes	Gauss	C4.5	PEBLS	CN2	Def.
Audiology	73.0±6.1	73.0±6.1^{6}	72.5±5.8^{6}	75.8±5.4^{3}	71.0±5.1^{5}	21.3
Annealing	95.3±1.2	84.3±3.8^{1}	90.5±2.2^{1}	98.8±0.8^{1}	81.2±5.4^{1}	76.4
Breast cancer	71.6±4.7	71.3±4.3^{6}	70.1±6.8^{5}	65.6±4.7^{1}	67.9±7.1^{1}	67.6
Credit	84.5±1.8	78.9±2.5^{1}	85.9±2.1^{3}	82.2±1.9^{1}	82.0±2.2^{1}	57.4
Chess endgames	88.0±1.4	88.0±1.4^{6}	99.2±0.1^{1}	96.9±0.7^{1}	98.1±1.0^{1}	52.0
Diabetes	74.5±2.4	75.2±2.1^{6}	73.5±3.4^{5}	71.1±2.4^{1}	73.8±2.7^{6}	66.0
Echocardiogram	69.1±5.4	73.4±4.9^{1}	64.7±6.3^{1}	61.7±6.4^{1}	68.2±7.2^{6}	67.8
Glass	61.9±6.2	50.6±8.2^{1}	63.9±8.7^{6}	62.0±7.4^{6}	63.8±5.5^{6}	31.7
Heart disease	81.9±3.4	84.1±2.8^{1}	77.5±4.3^{1}	78.9±4.0^{1}	79.7±2.9^{3}	55.0
Hepatitis	85.3±3.7	85.2±4.0^{6}	79.2±4.3^{1}	79.0±5.1^{1}	80.3±4.2^{1}	78.1
Horse colic	80.7±3.7	79.3±3.7^{1}	85.1±3.8^{1}	75.7±5.0^{1}	82.5±4.2^{2}	63.6
Hypothyroid	97.5±0.3	97.9±0.4^{1}	99.1±0.2^{1}	95.9±0.7^{1}	98.8±0.4^{1}	95.3
Iris	93.2±3.5	93.9±1.9^{6}	92.6±2.7^{6}	93.5±3.0^{6}	93.3±3.6^{6}	26.5
Labor	91.3±4.9	88.7±10.6^{6}	78.1±7.9^{1}	89.7±5.0^{6}	82.1±6.9^{1}	65.0
Lung cancer	46.8±13.3	46.8±13.3^{6}	40.9±16.3^{5}	42.3±17.3^{6}	38.6±13.5^{3}	26.8
Liver disease	63.0±3.3	54.8±5.5^{1}	65.9±4.4^{1}	61.3±4.3^{6}	65.0±3.8^{3}	58.1
LED	62.9±6.5	62.9±6.5^{6}	61.2±8.4^{6}	55.3±6.1^{1}	58.6±8.1^{2}	8.0
Lymphography	81.6±5.9	81.1±4.8^{6}	75.0±4.2^{1}	82.9±5.6^{6}	78.8±4.9^{3}	57.3
Post-operative	64.7±6.8	67.2±5.0^{3}	70.0±5.2^{1}	59.2±8.0^{2}	60.8±8.2^{4}	71.2
Promoters	87.9±7.0	87.9±7.0^{6}	74.3±7.8^{1}	91.7±5.9^{3}	75.9±8.8^{1}	43.1
Primary tumor	44.2±5.5	44.2±5.5^{6}	35.9±5.8^{1}	30.9±4.7^{1}	39.8±5.2^{1}	24.6
Solar flare	68.5±3.0	68.2±3.7^{6}	70.6±2.9^{1}	67.6±3.5^{6}	70.4±3.0^{2}	25.2
Sonar	69.4±7.6	63.0±8.3^{1}	69.1±7.4^{6}	73.8±7.4^{1}	66.2±7.5^{5}	50.8
Soybean	100.0±0.0	100.0±0.0^{6}	95.0±9.0^{3}	100.0±0.0^{6}	96.9±5.9^{3}	30.0
Splice junctions	95.4±0.6	95.4±0.6^{6}	93.4±0.8^{1}	94.3±0.5^{1}	81.5±5.5^{1}	52.4
Voting records	91.2±1.7	91.2±1.7^{6}	96.3±1.3^{1}	94.9±1.2^{1}	95.8±1.6^{1}	60.5
Wine	96.4±2.2	97.8±1.2^{3}	92.4±5.6^{1}	97.2±1.8^{6}	90.8±4.7^{1}	36.4
Zoology	94.4±4.1	94.1±3.8^{6}	89.6±4.7^{1}	94.6±4.3^{6}	90.6±5.0^{1}	39.4

Figure 3.12: Comparing results from many learners. From [DP97].

One interesting feature of Figure 3.12 is the *Lung cancer* and *Primary tumor* results. The performance scores in these data sets is lower than the other data sets and these performance dips occur *across all the learners*. From that figure, we say that data can be more important than algorithms in determining the success of learning.

3.4 So Why Study Data Mining?

"Hang on", you might be thinking, "all this data mining research and all the learners produce the same result? Why should I study a field which is obviously

wasting its time?". This is a fair question and there are three replies.

Firstly, data mining is not about algorithms. Other fields discuss algorithms: elegant beautiful algorithms with provable properties. Here, we're about algorithms working on data. And real-world data has many, many quirks. What seems like good ideas, in theory, may be irrelevant in practice. So one of the lessons of data mining are not about the algorithms (though the algorithms are exciting to study). Rather, what data mining really reveals is the strange wonderful state of the world around us. In particular, from Figure 3.11 and Figure 3.12, that the patterns are there to be found and they can be found by any number of methods. Which is wonderful news.

Secondly, once we know that different methods can yield similar results, we can are free to optimize other aspects such as

- Scalability: how can we change the algorithms such that they can process much larger data sets?
- Implementation simplicity: what can we throw away from our algorithms such that they still work but are simpler to code;
- Stability: all the biases mentioned above mean that minor changes to the learner/data can result in very large changes to the generated model. Is it possible to build learners such that their conclusions will still hold true in the future?
- Explainability: how can we improve the presentation of the models such that humans can better understand them?

Regarding explainability, my own work has explored a certain keys effect where a small number of features control the setting to everything else. When *keys* exist, the learning task reduces to just finding and exploring the keys. This kind of thinking has results in a new kind of minimal contrast set learner called TAR3 (shown above, see Figure 3.9). TAR3 assumes that keys exist and looks for the smallest number of ranges that have the most different frequencies in different target ranges. TAR3 is both a test and an application of this notion of *keys*. If keys exist, then TAR3 should return very small theories that users find fast and simple to understand. In nearly all test cases, TAR3 returns such small theories which is supportive (but into conclusive) evidence for the commonality of keys. However, there is much evidence for keys not only in the TAR3 research, but in other fields as well (see see [MDR07]).

Thirdly, sometimes, it is possibly to offer breakthrough results where some technique performs radically better than the alternatives. One example of this is offered below. Note the following research was conducted by a student of this class, two years ago. Zach Mitlon's work began a simple observation: evaluation bias is not search bias, but sometimes it should be. More precisely:

- A search bias is used during the learning to control what is generated.
- An evaluation bias is used after the model is generated and does not offer any feedback to the learners, except perhaps "please delete yourself, you are inferior to the other model, over there".

rank	treatment	median score	2nd quartile, median and 3rd quartile
1	which	70.9	
2	bayes classifier	56.8	
3	decision tree	27.6	
4	rule covering	21.9	
			50%

Figure 3.13: Results from the same data sets as Figure 3.11, assessing the results on combined from 10 repeats of a 3-way cross-val, sorted by *median* performance score.

- Strange to say, in many situations, learners use some internal search bias X to generate models, which are then evaluated using some different evaluation bias Y.

Zach didn't like this separation of the search bias from the evaluation bias. So he built a new kind of rule learner that supported moving the evaluation biases back into the search bias. His case study was predicting defecting software modules. Rather that search for "accurate" predictors (one whose's predictions matched those found in the test data), he searched for predictors that maximized:

$$\frac{\#\text{faulty modules found}}{\text{Total lines of code in modules predicted to be faulty}}$$

Note if a predictor scores well on this criteria, then test engineers could read less code and find more bugs (something that would please any member of a software testing team).

The results (shown in Figure 3.13) were quite exciting since they were quite different to the Figure 3.11 and Figure 3.12 results:

- Different learners now had statistically different behaviors.
- The learner whose search bias was the same as the evaluation bias (see WHICH), performed best[3]

Figure 3.13 shows those results using data similar to that explored by Lessmann et.al. from Figure 3.11. WHICH was the learner took the user's evaluation bias and used it as a search bias. The other learners used their internal search biases to generate models which were then evaluated according to the user's local biases. Note that WHICH2 out-performed them all, by quite a large margin.

What is interesting here is that the data used here was very similar to that used in Figure 3.11. The first column shows the results of a statistical analysis that checks if each learner is truly different to the others. Note how, under a certain domain-specific bias, all the learners have truly different performance

[3]For full details on this experiment, see `http://menzies.us/pdf/08which.pdf`.

(unlike in Figure 3.11. That is, the evaluation bias really matters since particular evaluation biases can reveal differences that are not apparent in other views.

One last thing before ending this chapter. Note that rule covering is the worst performing learner in Figure 3.13. This is not a general result: sometimes rule covering algorithms are indeed the right tool for the job. But, as shown above, rule covering is just one a large number of data mining techniques. You need to know many techniques in order to be prepared for the next problem you might encounter. So, don't stop reading.

Chapter 4

Tasks

Welcome back from your lightning fast tour of data mining. Now you a little about all the data mining tools, lets take our dialogue up one level.

In this chapter we offer a fanciful example of some mission critical data mining. Our heroes have been lured into a trap and must escape. To do so, they will apply a set of data mining tools in order to achieve number high-level tasks such as detection, diagnosis, alternative generation, evaluation, and monitoring.

But here's the key point- they'll rarely do one task at a time. Rather, they will mix and match the tasks, skipping rapidly between them. So if we just given them (say) a decision tree learner to classify some new data, that will only support a small fraction of their tasks.

What is the lesson here? Professional data mining analysts should not try to fit one tool to every problem. Rather, they should own a toolbox stuffed with different data mining methods that they can wire together into some ensemble solution.

4.1 What are "Tasks"?

There are many *kinds* of learners such as linear regression, neural nets, decision trees, Bayes classifiers, rule-covering algorithms, support vector machines, etc, etc. But are they *useful*? What *value-added* to they offer an organization?

To answer that question, we need to know the *task* of the organization and what is done to achieve that task. Specific companies perform specific tasks and if we explore every one of those, this book may stretch to the horizon but never reach any conclusions. Before we can offer general principles of data mining, we first need a general model of "task".

This notion of "task" has been well explored by the AI community. After decades of rule-based and model-based programming, certain cliched model-based tasks were identified. New tools were built around these cliches that relegated rules and models to lower level constructs, controlled by higher-level

- Finding problems
  - Detection
  - Diagnosis
- Solving problems
  - Alternative generation
  - Evaluation
  - Judgment
- Resolution
  - Monitoring

Figure 4.1: A simple hierarchy of knowledge-level tasks.

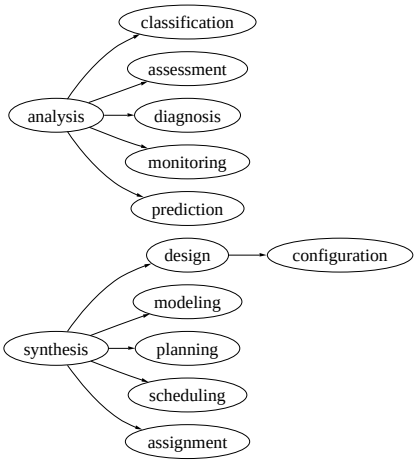

Figure 4.2: A more complex hierarchy of knowledge-level tasks.

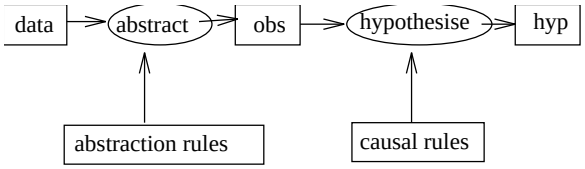

Figure 4.3: Inside the diagnosis task. From [vHA96].

knowledge-level [New82] tasks. Various researchers worked to identify standard sets of knowledge-level operators. For example, researchers in decision-support systems [Bro86] offered the simple hierarchy of Figure 4.1.

This list wasn't detailed enough for some an more complex hierarchies were proposed, such as Figure 4.2. Figure 4.3 offers the inner details of the diagnosis task. In that figure ovals in that figure denote functions and rectangles denote data structures. Any number of design and code libraries were built

around these knowledge-level tasks such as cognitive patterns [GRC+98]; CommonKADS [WSB92, SWA+94, Sch99]; configurable role-limiting methods [SG96, GM96]; MIKE [AFS96]; the Method-To-Task approach [EST+95]; generic tasks [CJS92]; SPARK/ BURN/ FIREFIGHTER [MDK+92]; model construction operators [Cla92]; components of expertise [Ste90]; and other systems described in [Ben95, BdVe94, CJS92, MZ96, TH93, Sch99, GRC+98].

But here's the thing: that line of work never stabilized. Every year, I'd go to conferences and there would be a new knowledge level task defined, or an old one redefined. There seemed no generality or stability in that line of research. For example:

- Figure 4.1 and Figure 4.2 both seem useful hierarchies. Which is best? How to assess alternative sets of "best" knowledge-level tasks? There was never any consensus on this issue.
- Because there was no way to assess the tasks, they kept multiplying. For example, I once cataloged eight different definitions of the diagnosis task [Men97].

Unhappy with the standard approach to knowledge-level tasks, I started thinking about simpler frameworks that let you combine tasks, as required. Several researchers including Allen Newell, the original author of the knowledge level principle, never endorsed structures like Figure 4.2. Instead, he preferred a single rule-based meta-level-search algorithm [RLN93]. In that approach, the observation that a device is performing (e.g.) the diagnosis task is a user-interpretation of the interaction of a set of lower-level tools. I speculated that there was something simpler that Newell's approach and tried various approaches including ripple-down-rules [CKPM93] and abductive logic [MM97]. Sadly, my code failed the scalability test. For example, the computational costs of abduction is quite high [BATJ91] so my abductive tools never worked for large examples.

For some years, I stopped knowledge-level modeling and worked on data mining instead. This was a field with rigorous empirical standards, large libraries of standard problems, and much successful work in scaling these algorithms to very large problems. Recently, I started wondering in data mining tools could be combined together to support knowledge level tasks. This idea is best explained via an example. We take you now to the deck of the U.S.S. Enterprise.

4.2 Star Trek, Episode 80

As the scene opens, the black hats have kidnapped our gallant crew and teleported them to the E2-zone. As the crew struggle to regain consciousness after their traumatic abduction, the captain barks an order.

"Status!" he demands and the science officer displays Figure 4.4 on the screen. Peering forward intently, our heroes discuss their situation.

"What are we looking at Mr. Spock?" asks the captain.

	1	2	3	4	5	6
A	++++++ +++++				- - - - - - - - - - -	
B						
C	++		-	-	-	+
D					-	
E	- - - - - - - -	X			+++ - - -	

Figure 4.4: Regions in data space.

"I have accessed the ship's TIVO", explains the science officer, "and have plotted the number of times a star ship has exploded in the next five minutes. Plus signs denote survival and minus signs ... "

"Are things we want to avoid," interrupted the captain. "I get it. What are the axes?"

"The axes represent things we can control such as red-shirted lackies on the bridge and the volume of the incidental music. And our position," says the science officer pointing to cell E2, "is marked with this X symbol."

The captain considered their position. "E5 looks... ambiguous", he remarked.

"Indeed," was the reply. "There is no conclusive evidence that E5 is unequivocally dangerous or safe. However, the same cannot be said for E1 where in 100% of cases, there was death in five minutes. Captain, while we consider our position, I suggest we move to E3 but refrain from entering E4."

"But that takes us closer to sites of old explosions at C3 and C4. "

"Indeed, but those regions are poorly supported by the historical record. Compared to the mountains of support in the A and E row, I would argue that the C row is just as ambiguous as E5. With such low support, it would be illogical to make any definitive conclusion about row C."

The captain agreed. While the ship moved from E2 to E3, they stared at the screen. Clearly, they had to plot a course for A1 in time for the closing credits. But how to get there?

Suddenly, the captain speaks up. "Spock... can the black hats access the same data as us?"

"Quite possibly caption. The networks have been broadcasting reruns of this show for decades now."

"Then we need to ask, they did they send us here? What do they win?"

The science considered the data. "I believe we are canaries."

"?"

"Earth miners used canaries to test for gases in mines. I conjecture that we are are being used to sample the uncertainties in the E-zone between the danger of E1 and the ambiguous E5."

"You know Spock, I think you might have something there. No lose to them

if we explode, and they might learn one more '-' along the way. Hmmmm... but there must be a back-up plan, just in case we survive..... (pause).... Spock!?"

"Yes captain?"

"Not canaries. Honey pots!"

"Captain?"

"Look at C1- it seems the safest path home to A1."

"But captain, as I've already said, that region has such comparative low support compared to the rest of the data that it is illogical to make definite conclusions in that sector."

"Yes Spock, but we have to cross row-C to get home. And the black hats know that too. I'm betting they're betting that we'll run through C1 on our way home."

"Where their fleet will, no doubt, be ready to destroy us. Impeccable logic, captain. But how do you suggest we proceed?"

"Well, there's no data on C2. By extrapolation, it could be '+' but then, as you say, there is so little data on that row that it could be anything at all."

"You you plan to navigate through C2?"

"No Mr. Spock, C3".

"But captain, that is a '-' sector."

"Yes Mr. Spock. You know that, I know that, and the black hats know we know that."

"And they will place some of their fleet there. I see. And the C3 sample is so small that it could be inaccurate."

The captain raised engineering on the intercom. "Scotty,", he said, "I need full power. We're going to blast through C3, hopefully before the black hats notice us and hopefully faster than five minutes."

Queue dramatic music as the Enterprise streaks through space, racing to safety.

4.3 Data Mining as Tasks

Returning now to Figure 4.1, let us consider what support data mining offers the tasks faced by our gallant crew.

4.3.1 Finding problems

Detection

Initially, our crew have an unsupervised learning problem. While in data gathering mode, the science officer displayed a clustering analysis of the local space and the majority classification of each cluster could be computed (which is difficult in the case of ambiguous regions like E5).

Diagnosis

Diagnosis of a problem (that are in a region of space that might kill them) could follow from either:

- A nearest neighbor analysis of the current position to the nearest cluster.
- Running a classifier like rule covering or decision tree learning or a Näive Bayes classifier to determine what region of space they most belong to.

4.3.2 Solving problems

Alternative generation

Contrast set learning could compute the delta between the current location and the desired goal. This would return (say) a least cost path from the E2 to A1.

Evaluation

By changing the evaluation function, it would be possible to find a set of paths connecting the current situation to a desired goal.

Judgment

The paths could be sorted and one could be selected at random from the space of paths with similar probabilities of success (this is how the captain selected C3).

4.3.3 Resolution

Monitoring

For the crew of the Enterprise, the monitoring involves tracking their state changes and computing their odds of survival at every step. Perhaps, add in some high drama with a temporary engine failure as the ship crosses C3 (while the engineers race to the Jeffry's tubes to repair some electrical thing, shooting sparks in all directions). Burst through to A1, music blaring. End scene. Roll credits.

Note also, that the black hats in the above scenarios are also engaged in an active monitoring process. They deliberately dropped the Enterprise into an under-sampled part of the space in order to increase their knowledge of the domain. That is, knowledge of the feature space of Figure 4.4 could be used for the design of experiments and selecting the minimal decisions required to increase information content.

4.4 Enough

That the highest-level picture I can offer of data mining. The rest of this book describes a data mining LISP library. The intent of that library is allow the rapid

deployment of data miners that support high-level business tasks by mixing and matching various data mining tools.

Part III

Tips (for Beginners)

Chapter 5

Setting up

This chapter contains some setting up instructions. This are prone to rapid change.

5.1 Before you Begin

Before starting, you'll need to install EMACS, StonyBrook Common LISP (SBCL), and the Superior Lisp Interaction Mode for Emacs (SLIME).

- For WVU LCSEE students, these are already installed in your school machines.
- For users of the LOUD UBUNTU distribution, these come with the distribution.
- For LINUX users, these are readily available via your package manager (e.g. SYNAPTIC).
- For WINDOWS users, rumor has it that these are available under CYGWIN.
- For OS/X users, download and install READY LISP from `http://www.newartisans.com/software/readylisp.html`.

5.1.1 Notes on READY LISP (OS/X users only)

I can't stand READY LISP's built-in paredit mode. Too restrictive! So I edit out all the paredit stuff in READY LISP's initialization file:
/Applications/Ready Lisp.app/Contents/Resources/site-lisp/init-lisp/el

You may want to do the same.

Also, at the time of this writing, READY LISP does not read $HOME/.emacs. To enable customizations, edit the READY LISP initialization file and add to the end:

```
(setq-default indent-tabs-mode nil) ; indent code with spaces, not tabes
(one-buffer-one-frame-mode 0)       ; open new files in current frame
(setq frame-title-format "%f")      ; show full pathname in window title
(setq backup-directory-alist nil)   ; get rid of those irritationg x~ files
(setq backup-directory-alist
      (cons (cons "\\.*$" (expand-file-name "~/opt/tmp/backup"))
    backup-directory-alist))
```

Note that, after installing ICCLE (see below), these lines are also available in */opt/iccle/osx/etc/dotaquamacs*. So instead of retyping the above every time you update READY LISP, you could use

```
(load "~/opt/iccle/osx/etc/dotaquamacs")
```

5.1.2 Notes for Non-OS/X Users

SLIME needs some set up in *$HOME/.emacs*:

```
(setq inferior-lisp-program "/usr/bin/sbcl --noinform") ;; or wherever you put SLIME
;(add-to-list 'load-path "/usr/share/common-lisp/source/slime/") ;; this path is WVU CSEE specific
;(setq slime-path "/usr/share/common-lisp/source/slime/")        ;; this path is WVU CSEE specific
(require 'slime)
(slime-autodoc-mode)
(slime-setup)
(global-set-key "\C-cs" 'slime-selector)
(add-hook 'lisp-mode-hook (lambda () ;; there are rumors of this crashing at CSEE.
                                     ;; if so, just delete this hook.
(slime-mode t)
(local-set-key "\r" 'newline-and-indent)
(setq lisp-indent-function 'common-lisp-indent-function)
(setq indent-tabs-mode nil)))
```

Once SBCL/SLIME/EMACS are installed, you need to join up before you can begin.

5.2 Joining, Downloading, Installing, Testing

5.2.1 Joining

Create a GOOGLE email account.

- Go to `http://gmail.com` and look bottom-right for "sign-up for gmail".
- Once you have that email account, you can continue.

Join the ICCLE code team.

- Go to `http://code.google.com/p/iccle`;
- Look bottom right for a list of project owners. Send your GOOGLE email to one of them, asking for ICCLE access.
- Once you are a member of the ICCLE group, got to `http://code.google.com/hosting/settings` and make your "Issue change notification" settings.

Once they have added you as a project member, you can continue.

5.2.2 Download and Install

- Pick a team number that no one else is using. Call it N.
- Sign into GOOGLE.
- Go to http://code.google.com/hosting/settings, see your GOOGLE code password.
- In a UNIX shell, type

```
cd
mkdir svns
cd svns
svn checkout https://iccle.googlecode.com/svn/ iccle --username YOUR.GOOGLE.NAME
cd iccle
svn cp trunk/ branches/team/N     # <=== your team number.
make install
make doc
```

This could take a few minutes to download and set up. When this terminates:

- There will a copy of the ICCLE manual (this document) in *$HOME/tmp/iccle.pdf*.
- There will be the core ICCLE code in *svn/iccle/trunk*.
- You will have your own copy of the code, which you can update and destroy at will, in *svn/iccle/branches/team/N*.

5.2.3 Configure

Configure your BASH environment

Check your file *$HOME/.bashrc*. It should contain the lines

```
export SVN_EDITOR=‘which nano‘      # or whatever text editor your prefer
export ICCLE=$HOME/svns/iccle/branches/team/N  # where N is your team number
```

Now, either login and log back in or source this file; e.g.

```
. $HOME/.bashrc
```

This step is required before you can continue.

5.2.4 Configure your EMACS environment

Note: skip this section if you are a READY LISP user.

Add these lines to your *$HOME/.emacs* file

```
(setq backup-directory-alist nil)
(setq backup-directory-alist ; get the backups out of the working directories
    (cons (cons "\\.*$" (expand-file-name "~/opt/tmp/backup"))
  backup-directory-alist))
(setq inhibit-startup-message t)    ; why isn't this the default?
(setq scroll-step 1)                ; don't scroll in large jumps
(setq require-final-newline   t)    ; every file has at least one new line
(global-font-lock-mode t 1)         ; enable syntax highlighting
(line-number-mode t)                ; show line numbers and time in status line
(setq display-time-24hr-format nil) ; show line numbers and time in status line
(display-time)
(xterm-mouse-mode t)                ; let the mouse work
(menu-bar-mode nil)                 ; remove that space wasting bar
(show-paren-mode t)                 ; match brackets
(transient-mark-mode t)             ;  show search
(setq load-path (cons "~/opt/iccle/color-theme" load-path))
(require 'color-theme)
(color-theme-initialize)
(color-theme-hober) ; default color theme
```

Note that you might want to change the default color theme. If so, fire up EMACS and type "M-x color-theme-selector" and pick another scheme.

5.2.5 Testing ICCLE

Booting ICCLE Inside EMACS

- For READY LISP users, in the shell, type these commands

  ```
  cd $ICCLE
  make macedit
  ```

  In EMACS, type this commands

  ```
  M-x load file <RET> iccle.lisp <RET>
  ```

- For other users, in the shell, type these commands

  ```
  cd $ICCLE
  make edit
  ```

  In EMACS, type this command

  ```
  M-x slime
  M-x load-file <RET> iccle.lisp <RET>
  ```

Run the test suite

Inside EMACS, run this command. You should see zero failures.

```
(test-all)
```

Once you've tested ICCLE, you are ready to develop new code.

Chapter 6

Using ICCLE

6.1 ICCLE Makefile Utilities

The Makefile in the *$ICCLE* directory offers some handy development utilities. To see these, type:

```
cd $ICCLE
make
```

At the time of this writing, those utilities are:

```
 1 clean   : delete editor temporary files
 2 commit  : save work to repository
 3 doc     : generate pdf documentation
 4 edit    : general edit
 5 help    : show help
 6 install : create local files and directories
 7 license : print license
 8 macedit : os/x edit
 9 status  : show modified, new files
10 update  : update work from repository
11 zip     : make a tag
```

At the start of each day's work, it is recommended that developers use:

```
cd $ICCLE
make update # updates from repository
```

To boot EMACS and ICCLE, use

```
cd $ICCLE
make macedit # READY LISP users only
make edit    # everyone else
```

At least once a day (and more is better), it is recommend that developers use:

```
cd $ICCLE
make commit   # deletes temporary files before committing to repository
```

Before committing, it is useful (a) remove any temporary files and (b) check for "rogue" files that need to be added to the repository. This is a one-liner:

```
cd $ICCLE
svn clean status # prints the commands needed to add the rogues
```

To generate a PDF of the current ICCLE manual, use:

```
cd $ICCLE
make doc
```

To generate a zip file, suitable for adding to a tag directory:

```
cd $ICCLE
make zip # deletes temp files, compiles the doc, adds it to trunk/doc/iccle.pdf
```

To generate the above help text, use:

```
cd $ICCLE
make help
```

To modify the help text, read *$ICCLE/Makefile* and look for "#:" comments after some of the rules. The "make help" command is just a grep and pretty print of those comments.

6.2 Sharing the Code

This section has noted on how to use and share versioned software like ICCLE.

6.2.1 Trunks, Branches, and Tags

Trees grow, mostly in the trunk. Sometimes, they branch out in novel directions. Some branches get sick and have to be pruned. Otherwise, when we like a branch or trunk, we take a snapshot of them and keep that snapshot safely stored in the vault.

As with trees, so with code. Code repositories can contain "branches", a "trunk", and "tags" (the vault where we store snapshots):

```
iccle
   branches
       solo
         ashesh
         timm
         ...
       team
         3
         9
         ...
   tags
       0.1
         COPYRIGHT.txt
         iccle_v0_1.zip
```

```
        INSTALL.txt
        Makefile
        README.txt
        ...
trunk
    Makefile
    ...
```

For the most part, groups working together to extend code in the "trunk". Sometimes, they'll try something experimental and branch out with a copy of part of the trunk (this copy is stored in "branches"). At some landmark, something is ready for release to a broader community and that landmark is kept under tags. For example, in the above directories, there is a version "0.1" of ICCLE stored under "tags".

6.2.2 Good Manners

Everyone in a code repository can muck up the repository for everyone else. So all users of a repository need to work with caution. At the very least:

- Everyone working on the same repository should read their email a few times a day looking for messages along the lines of "help! you've broken XYZ! fix it!".
- Everyone working on the same repository should join the mailing list for that repository. For ICCLE, that mailing list is `http://groups.google.com/group/iccle`.

Also, since files can be checked out to multiple operating systems so file names must be selected to be legal on all those platforms:

- Always use lower case.
- Avoid spaces in file names
- Keep file names under 100 characters
- Avoid funny punctuation. Never use {# % ; : ,}. Underscores are legal for SUBVERSION, but LATEX has a problem with them as file names. So they are forbidden as well. In short, try and stay with a..z and 0..9. For more details on file name limitations, see `http://en.wikipedia.org/wiki/Filename`.
- Finally, before committing your changes every day, make sure you delete any temporaries. In ICCLE, the "make clean" command in trunk deletes:
  - certain temporary files generated by some commonly used editors
  - Any files of compiled LISP code (the .fasl files).

6.2.3 Branch, a lot

Branching is useful when you only want to share part of other people's code. For example:

- The main load file of ICCLE is found in *trunk/lib/iccle.lisp*. This main calls a bunch of load control files like *trunk/lib/table.asd*.
- There is nothing stopping your from writing your own main and load control files such as *branches/team/3/iccle.lisp* and *branches/team/3/table.asd*. These main and load files can still reference some of the trunk files, as well as some local work you store in your own branch/team.

You should feel free to generate as many branches as you want. Under the hood, a version control system is free to do magic when copying from trunk to branches to tags. The underlying database does the standard tricks with aliases etc such the copy only requires a little more space than the original. So feel free to copy a lot since such copies have very low overhead. For example:

- "Ashesh" and "timm" have decided to do a little of their own exploration under the "solo" branch.
- And the team "3" and "9" have decided that they want to work together away from the rest of the pack. Their work can be found under "team".

6.2.4 After Branching

The end result of branching should either be recognizing a dead end or new work that is to be added to the trunk. So after branching, comes either pruning or merging:

- Pruning is easy- just delete that particular sub-branch. Note that, since the code is under version control, you can always get the deleted code restored, using the versioning system.
- Merging is more fun and must be done with great care. Your version control system should offer support for finding differences and automating the merges (e.g. in SUBVERSION, see the *svn diff* and *svn merge* commands). Also, if programmers have written test suites, these should be run after the merge to ensure that what works before, still works now.

6.2.5 Tagging

Once the code is at some stable point, it becomes a landmark that can be issued to a wider community. It is useful to label that landmark with a version number and copy all the code in that landmark to some location where everyone can get to it. In the terminology of version control, this is called "tagging".

Making a tag is a big thing. The tag should be as clean as possible- all temporary files deleted, all test suites working, etc. Also, as shown above, the root of tag should include some magic files:

- A copyright notice[1]

[1]ICCLE uses GPL 3.0. The GPL permits anyone to see, modify and redistribute that source code, as long as they make changes available publicly and license them under the GPL. That contrasts with some licenses used in open-source projects that permit source code to

- A single zip file containing the whole tag. Note: this zip should be free of any magic files added by the version control system. The ICCLE Makefile has a rule for this; see "make zip".
- An file describing the install instructions at a fresh site.
- Maybe, a Makefile to automate common tasks (installation, etc).
- A README file describing the system.

6.3 Unit Tests

Two issues with sharing code are:

- Explaining your code to other people;
- Making sure that your new changes to break other people's work.

Unit tests are a useful way to address these issues. A unit test is a canned example that runs some functions. The outputs are automatically compared to some expectation in order to see if a test has *passed* or *failed* the tests.

Since it is so useful, ICCLE makes extensive use of unit tests. If you look at the code, you'll see many `deftest` commands. ICCLE developers are *strongly* urged to use and extend ICCLE's unit tests. In particular:

- Run the unit tests before committing.
- As far as possible, do not commit code to the repository that breaks other people's tests. If that is not possible, at least alert other people if your new work breaks their old work.

Test-Driven Development

Unit tests are also very useful when writing your own code. Authors like Kent Beck argue that software development should be structured around building and running unit tests. Test-driven development (TDD) requires the programmer to first fail the test cases. The idea is to ensure that the test really works and can catch an error. Once this is shown, the normal cycle will commence. This has been coined the "Test-Driven Development Mantra", known as red/green/refactor where red means fail and green is pass.

In a pure TDD approach, work proceeds as follows[2]:

Planning rules:

- User stories are written. (a.k.a. lots of tests);
- Make frequent small releases (a.k.a. shows a string of tests);
- Fix the program only when it breaks (aak.a. code-run-break-fix is the usual cycle).

Coding rules:

be made proprietary. Hence, for any tag of ICCLE, the COPYRIGHT.txt file should include the GPL 3.0 text; see http://www.gnu.org/licenses/gpl-howto.html and `http://www.gnu.org/licenses/quick-guide-gplv3.html`.

[2]From `http://en.wikipedia.org/wiki/Test_driven_development`

- Code the unit test first.

Testing rules:

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

6.3.1 Demo-driven Development

After reading Kent Beck's book *Test-driven development* [Bec00] I was struck with how small each test was. When checking operations on a new abstract data type, just mini-tests are certainly appropriate. However, when working with large granularity programs Kent-style tests seemed to small.

So I propose a (small) variation to TDD. Think about how you might present the code to someone else, what you'd show off, what you'd use to demonstrate the core principles. This variant is called "demo-driven development" (DDD) and it differs from TDD in that each "demo" usually comprises "large-grain tests" that demonstrate some functionality.

In short, TDD demands thousands of tests while DDD demands dozens (or less) of demonstration scripts. Currently, ICCLE's `deftests` are more DDD than TDD.

Chapter 7

What to Read

7.1 LISP

7.1.1 Web-based Material

- The absolute best Common LISP quick reference ever devised: `http://clqr.berlios.de/clqr-a4-consec.pdf`;
- The very cool LispForum `http://www.lispforum.com/index.php`;
- Common Lisp Educational Resources: `http://www.lisp.org/alu/res-lisp-education`
- Common-Lisp.net: `http://common-lisp.net/`
- Marty Hall's LISP links: `http://www.apl.jhu.edu/~hall/lisp.html`

7.1.2 Tutorials

- LISP quick start: `http://cs.gmu.edu/~sean/lisp/LispTutorial.html`
- How to wrote LISP macros (advanced material): `http://foner.www.media.mit.edu/people/foner/Miscellaneous/macros.html`

7.1.3 Books

On-Line Books

- Paul Graham's *On Lisp*: `http://www.paulgraham.com/onlisptext.html` (source code available from `http://www.paulgraham.com/lispcode.html`);
- Peter Seibel's excellent text *Practical Common Lisp*: `http://gigamonkeys.com/book/` (you've got to read this one)

Other Books

- Paul Graham's ANSI Common Lisp (source code available from `http://www.paulgraham.com/lispcode.html`);
- Peter Norvig's Paradigms of AI: `http://norvig.com/paip.html`.

7.2 GNUPLOT

The on-line GNUPLOT manual can be found at `http://www.gnuplot.info/docs/gnuplot.html`. This manual is somewhat verbose so some simpler tutorials are shown below.

7.2.1 Tutorials

- GNUPLOT 101: `http://www.duke.edu/~hpgavin/gnuplot.html`
- Further help for beginners: `http://cs.gmu.edu/~sean/lisp/LispTutorial.html`
- Help for advanced users: `http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`

7.3 LATEX

7.3.1 Tutorials

- LATEX 101: `http://www.jgsee.kmutt.ac.th/exell/General/LaTeX.html`
- A really great guide to LATEX: `http://ctan.tug.org/tex-archive/info/lshort/english/lshort.pdf`.
- On-line links: `http://www.latex-project.org/guides/`

Part IV

Tricks (for Programmers)

Chapter 8

Using The Code Base

8.1 Directories and Code

ICCLE code is divided as follows:

```
 1    COPYRIGHT.txt
 2    INSTALL.txt
 3    Makefile
 4    README.txt
 5    doc/             # latex source files for make
 6         Makefile    # make doc creates iccle.pdf
 7    etc/
 8    lib/
 9        iccle.asd
10        iccle.lisp
11        store.asd
12        store/
13        tests.asd
14        tests/
15        toolspre.asd
16        toolslearn.asd
17        tools/
18        tricks.asd
19        tricks/
20    share/
21        data/
22        img/          # .png .jpg .gif .pdf files
```

Lines 1,2,3,4 are system files, the most important being $Makefile$, described in §6.1.

The *doc/* directory stores the LATEX source files and can be built using

```
cd $ICCLE/doc
make doc
```

The *etc/* directory stores various files used during installation.

```
; disable an irritating SBCL flag
#+SBCL (DECLAIM (SB-EXT:MUFFLE-CONDITIONS CL:STYLE-WARNING))

(defun make  () (make1 "iccle"))
(defun make! () (make1 "iccle" t))

(defun make1  (sys &optional forcep)
  (asdf:operate 'asdf:load-source-op sys :force forcep))

(make)
```

Figure 8.1: CODE: `lib/iccle.lisp`

```
; -*- mode: lisp; -*-
(defsystem "iccle"
  :description "some data mining tricks in LISP"
  :version     "0.01"
  :author      "The Mountain Lisp Gang and Tim Menzies (ed)"
  :licence     "GPL3.0"
  :depends-on ("tests" "tricks" "store" "toolslearn" "toolspre"))
```

Figure 8.2: CODE: `lib/iccle.asd`

The *share/data* directory stores sample data files.
The *share/img* directory stores image files.
The LISP source code is stored in the *lib/* directory:

- The main load file is *lib/iccle.lisp*.
- The *lib/* $*$ *asd* files store the file dependencies.

8.2 Loading ICCLE

All the ICCLE code is loaded by *iccle.lisp* in Figure 8.1. A standard boot sequence is therefore as simple as:

```
(load  "iccle")
```

This defines two simple functions that controls the loading of Figure 8.2:

make `(make)` Updates the system. Initially, loads all files. Subsequently, just uploads any updated files.

make! `(make!)` Loads (or reloads) the entire system.

8.3 Dependencies

The dependencies within ICCLE are defined by

```
;;;; -*- mode: lisp; -*-

(defsystem "tests"
    :description "unit testing for LISP code.
                  Based on Peter Seibel's code"
    :version     "0.01"
    :author      "The Mountain Lisp Gang and Tim Menzies (ed)"
    :licence     "GPL3.0"
    :components
    ((:file "tests/deftest")
     (:file "tests/tools")
     (:file "tests/tricks")
     (:file "tests/tests"
            :depends-on ("tests/deftest"
                         "tests/tools" "tests/tricks"))))
```

Figure 8.3: CODE: `lib/tests.asd`

- *lib/tests.asd*: the unit testing framework of Figure 8.4
- *lib/tricks.asd*: the application independent LISP tricks of framework of Figure 8.4
- *lib/store.asd*: the data storage and indexing functions of Figure 8.5
- *lib/toolslearn.asd*: the data mining tools of Figure 8.6
- *lib/toolspre.asd*: continued data mining tools of Figure 8.7

Note that if you want to start a new *.asd* file, you need to:

- Create it in *lib*;
- Reference it in the main *.asd* file *lib/iccle.asd*.

```
; -*- mode: lisp; -*-
(defsystem "tricks"
  :description "application independent LISP tricks"
  :version     "0.01"
  :author      "The Mountain Lisp Gang and Tim Menzies (ed)"
  :licence     "GPL3.0"
  :depends-on ("tests")
  :components
  ((:file "tricks/array")
   (:file "tricks/hash")
   (:file "tricks/strings")
   (:file "tricks/maths")
   (:file "tricks/list")
   (:file "tricks/matrix"
          :depends-on ("tricks/list"))
   (:file "tricks/random")
   (:file "../share/data/housing")
   (:file "tricks/tree")
   (:file "tricks/counted")
   (:file "tricks/caution")
   (:file "tricks/line")
   (:file "tricks/lispfuns")
   (:file "tricks/os" :depends-on ("tricks/strings"))
   (:file "tricks/gnuplot" :depends-on ("tricks/os"))
   (:file "tricks/debug" :depends-on ("tricks/lispfuns"))
   (:file "tricks/ttest" :depends-on ("tricks/line"))
   (:file "tricks/dist" :depends-on ("tricks/ttest"))
   (:file "tricks/distogram" :depends-on ("tricks/dist"))
   (:file "tricks/mannwhitney" :depends-on
          ("tricks/list" "tricks/maths" "tricks/ttest"))
   (:file "tricks/wilcoxon2")
   (:file "tricks/wilcoxon" :depends-on ("tricks/wilcoxon2"))
   (:file "tricks/align" :depends-on ("tricks/matrix" "tricks/strings"))
   (:file "tricks/abcdmatrix")
   (:file "tricks/abcd")
   (:file "tricks/abcdtest"
          :depends-on ("tricks/abcd" "tricks/abcdmatrix"))))
```

Figure 8.4: CODE: `lib/tricks.asd`

```
; -*- mode: lisp; -*-

(defsystem "store"
  :description "data storage and indexing"
  :version     "0.01"
  :author      "The Mountain Lisp Gang and Tim Menzies (ed)"
  :licence     "GPL3.0"
  :depends-on ("tricks")
  :components
   ((:file "store/unknown")
    (:file "store/header")

    (:file "store/table"  :depends-on ("store/header"))
    (:file "store/tableselectors" :depends-on ("store/table"))
    (:file "store/data" :depends-on ("store/table"))
    (:file "store/datatest" :depends-on ("store/data"))

    (:file "store/xindex" :depends-on ("store/table"))
    (:file "store/f" :depends-on ("store/xindex"))
    (:file "store/xindexselectors" :depends-on ("store/xindex"))
    (:file "store/crossindex"
           :depends-on ("store/table" "store/xindex"))
))
```

Figure 8.5: CODE: `lib/store.asd`

```
; -*- mode: lisp; -*-
(defsystem "toolslearn"
  :description "application independent LISP tricks"
  :version     "0.01"
  :author      "The Mountain Lisp Gang and Tim Menzies (ed)"
  :licence     "GPL3.0"
  :depends-on ("tricks" "store")
  :components
   (
    (:file "../share/data/weather")
    (:file "../share/data/weather2")
    (:file "../share/data/contactlens")
    (:file "tools/pre/folds")
    (:file "tools/learn/nway" :depends-on ("tools/pre/folds"))
    (:file "tools/learn/rangedetails")

    (:file "tools/learn/iterativedichotomize"
           :depends-on ("tools/learn/nway"))
    (:file "tools/learn/info"
           :depends-on ("../share/data/weather2"
                        "tools/learn/rangedetails"
                        "tools/learn/iterativedichotomize"))

    (:file "tools/learn/id3"
           :depends-on ("tools/learn/iterativedichotomize"
                        "tools/learn/info"))

    (:file "tools/learn/nb"
           :depends-on ("../share/data/weather2"
                        "tools/learn/nway"))

    (:file "tools/learn/prism")
    (:file "tools/learn/prismrule"
           :depends-on ("tools/learn/prism"
                        "../share/data/contactlens"))

    (:file "tools/learn/prismtest"
           :depends-on ("tools/learn/prism"))

    (:file "tools/learn/zeror"
           :depends-on ("../share/data/weather2"
                        "tools/learn/nway"))
     ))
```

Figure 8.6: CODE: `lib/toolslearn.asd`

```
; -*- mode: lisp; -*-
(defsystem "toolspre"
  :description "application independent LISP tricks"
  :version     "0.01"
  :author      "The Mountain Lisp Gang and Tim Menzies (ed)"
  :licence     "GPL3.0"
  :depends-on ("tricks" "store")
  :components
   (
    (:file "../share/data/weathernumerics")
    (:file "tools/pre/folds")
    (:file "tools/pre/unsuperviseddiscretization"
             :depends-on ("../share/data/weathernumerics"))
    (:file "tools/pre/efd"
           :depends-on ("tools/pre/unsuperviseddiscretization"))
    (:file "tools/pre/ewd"
           :depends-on ("tools/pre/unsuperviseddiscretization"))
    (:file "tools/pre/fastkplot")
    (:file "tools/pre/fastmap4kmeans2")
    (:file "tools/pre/fastmap4kmeans3" :depends-on
          ("tools/pre/fastmap4kmeans2"))
    (:file "tools/pre/quicksort")
    (:file "tools/pre/saveload")
    (:file "tools/pre/kmeanstest")
    (:file "tools/pre/fastmap"
           :depends-on ("tools/pre/fastmap4kmeans3"))
    (:file "tools/pre/kmeans"
           :depends-on ("tools/pre/fastmap4kmeans3"))
    (:file "tools/pre/fastkmain" :depends-on
           ("tools/pre/fastmap" "tools/pre/kmeans"
            "tools/pre/saveload" "tools/pre/quicksort"))
    (:file "tools/pre/discretehandler" :depends-on
           ("tools/pre/fastmap" "tools/pre/kmeans"))
     ))
```

Figure 8.7: CODE: `lib/toolspre.asd`

Chapter 9

Unit Testing : deftest

ICCLE uses an adaption of Peter Seibel's unit unit testing code. This defines

`(deftest test () ..)` Defines a unit test. *deftest*

`(tests test1 test2...)` (macro) High-level driver for the unit tests. First, it resets all the pass, fail counters. Second, it runts `test1`, `test2`, etc. Third, report the pass fail counts. Internally, this macro uses `tests-reset` and `tests-report`. *tests*

`(tests-report)` Reports the test report counters. *tests-report*

`(tests-reset)` Initializes the test report counters. *tests-reset*

The `(tests-reset)` and `tests-report` functions manager counters for test `passes` and `fails`. These counters are incremented by the unit test code as follows (note that this example demonstrates the use of *closures* in LISP):

```
(let ((passes 0) (fails 0))
  (defun tests-reset()    (setf passes 0) (setf fails 0))
  (defun tests-report ()
    (format t "~%PASSES: ~a (~a %)~%FAILS : ~a~%"
            passes (* 100 (/ passes (+ passes fails)))
            fails))
  (defun report-result (result form)
    (if result
        (and (incf passes)
             (format t "."))
        (and (incf fails)
             (format t "~%fail ... ~a: ~a~%"  *test-name* form)))
    result)
)
```

The idiom for using these counters is demonstrated by the `(test-all)` function of *lib/tests/test.lisp*:

ICCLE's unit tests are written in a hierarchy. If some nested test is called within a super-test, then the code recurses into sub-tests. In this way, so centralized test facility can be defined in terms of multiple layers of tests spread through out the entire code.

test-all `(test-all)` Top-level call to all the unit tests: see Figure 9.1. Developers should develop extensive test suites called by `(test-all)`. At the time of this writing, test suites have been built for *tricks* and *tools*.

test-tricks `(test-tricks)` Top-level call to the unit tests for the ICCLE tricks code.

test-tools `(test-tools)` Top-level call to the unit tests for the ICCLE tools code.

One useful function for unit tests is the `samep` function that compares a LISP term against some sting, while ignoring any whitespace. For example:

```
(deftest test-align ()
  (let ((tmp (with-output-to-string (s)
               (align '((peace love understanding)
                        (34.1 33.4 324.2)
                        (334 222222 33)) :stream s))))
    (samep tmp "
         PEACE |   LOVE | UNDERSTANDING
         ----- | ------ | -------------
         34.1 |   33.4 |         324.2
         334 | 222222 |            33
")))
```

This is very useful when writing unit tests since it means the test comparison code is not damaged by a single added character. For example, the output of `align` has very little left-hand-side white space:

```
 PEACE |   LOVE | UNDERSTANDING
 ----- | ------ | -------------
  34.1 |   33.4 |         324.2
   334 | 222222 |            33
```

Nevertheless, it was useful when writing to test include some left-hand-side space in order to pretty format `test-align`.

```
; From Peter Seiblel's excellent text: http://gigamonkeys.com/book
(defvar *test-name* nil)

(defmacro deftest (name parameters &body body)
  `(defun ,name ,parameters
    (let ((*test-name* (append *test-name* (list ',name))))
      ,@body)))

(defmacro check (&body forms)
  `(combine-results
    ,@(loop for f in forms collect `(report-result ,f ',f))))

(defmacro with-gensyms ((&rest names) &body body)
  `(let ,(loop for n in names collect
               `(,n (make-symbol ,(string n))))
     ,@body))

(defmacro combine-results (&body forms)
  (with-gensyms (result)
    `(let ((,result t))
      ,@(loop for f in forms collect
              `(unless ,f (setf ,result nil)))
      ,result)))

(let ((passes 0) (fails 0))

  (defun tests-reset()
    (setf passes 0)
    (setf fails 0))

  (defun tests-report ()
    (format t "~%PASSES: ~a (~a %)~%FAILS : ~a~%"
            passes (* 100 (/ passes (+ passes fails)))
            fails))

 (defun report-result (result form)
    (if result
        (and (incf passes)
             (format t "."))
        (and (incf fails)
             (format t "~%fail ... ~a: ~a~%"  *test-name* form)))
    result)
  )

(defmacro tests (&body body)
  `(progn
     (make)
     (tests-reset)
     ,@body
     (tests-report)))
```

Figure 9.1: CODE: `lib/tests/deftest.lisp`

```
(defun test-all ()
  (tests
    (test-tricks)
    (test-tools)
  ))
```

Figure 9.2: CODE: `lib/tests/tests.lisp`

```
(deftest test-tricks ()
  (test-do-array)
  (test-showh)
  (test-remove-nths)
  (test-align)
  (test-distogram)
  (test-random)
  (test-line)
  (test-caution)
  (test-stats)
  (test-ttest)
  (test-mann-whitney)
  (test-wilcoxon)
  (test-caution)
  (test-k-means)
  (test-fastmap)
  (test-ancillary)
  (test-quicksort)
  )
```

Figure 9.3: CODE: `lib/tests/tricks.lisp`

```
(deftest test-tools ()
  (test-data)
  (test-discretize)
  (test-xindex-f)
  (test-xindex-majority-class)
  (test-index)
  (test-folds)
  (test-zeror)
  (test-prism)
  (test-nb)
  (test-abcd)
  (test-info-splitter)
  (test-id-trainer)
)
```

Figure 9.4: CODE: `lib/tests/tools.lisp`

Chapter 10

Basic Tricks

This chapter describes some basic tricks with standard LISP data types.

10.1 Arrays

The file *lib/tricks/array.lisp* in Figure 10.1 defines:

`(array2list a)` Return a list from the array `a`. *array2list*

`(doarray (item a &optional out) body)` (macro) Execute `body` for each `item` in array `a`. Return `out` (default nil). *doarray*

`(doarray (item index a &optional out)` (macro) Execute `body` for each `item` at position `index` in array `a`. Return `out` (default nil). *doarray*

10.2 Hash Tables

The file *lib/tricks/hash.lisp* in Figure 10.2 defines:

`(dohash (key value hash &optional out) body)` Execute `body` for each `key` and `value` in the hash table `hash`. Return `out` (default nil). *dohash*

`(showh h &key indent stream before after if-empty show lt)` Print the hash table `h` onto `stream` (default, standard output), indenting each item with `indent` (default, 0) number of spaces. Print the strings `before` and `after` before and after showing the table. If the hash table is empty, just print `before, if-empty, after`. Print the hash keys in the sort order defined by `lt` (default, alphanumeric). Print each key value pair using the pretty print function `show`. This pretty print defaults to:
indent key = value newline *showh*

```
(defun array2list (a)
  (let (l)
    (dotimes (i (length a) (reverse l))
      (push (aref a i) l))))

(defmacro doarrayi ((one n array &optional out) &body body)
  (let ((i (gensym)))
    `(let ((,i (length ,array)))
       (dotimes (,n ,i ,out)
         (let ((,one (aref ,array ,n)))
           ,@body)))))

(defmacro doarray ((one array &optional out) &body body)
  (let ((j (gensym)))
    `(doarrayi (,one ,j ,array ,out) ,@body)))

(deftest test-do-array ()
  (let* ((count 0)
         (r1    (doarrayi (x i #(0 1 2) count)
                  (incf count x)
                  (incf count i)))
         (r2    (doarray (x #(0 1 2) count)
                  (incf count x))))
   (check
     (= r1 6)
     (= r2 9))))
```

Figure 10.1: CODE: `lib/tricks/array.lisp`

```
(defmacro dohash ((key value hash &optional end) &body body)
  `(progn (maphash #'(lambda (,key ,value) ,@body) ,hash)
          ,end))

(defun showh (h &key
              (indent 0) (stream t) (before "") (after "")
              (if-empty "empty")
              (show #'(lambda (x)
                        (format stream "~a~a = ~a~%"
                           (nchars indent) (first x) (rest x))))
              (lt #'lt))
 (if (zerop (hash-table-count h))
     (format stream "~a~a~a" before if-empty after)
     (let (l)
       (format stream "~a" before)
       (maphash #'(lambda (k v) (push (cons k v) l)) h)
       (mapc show
             (sort l #'(lambda (a b)
                         (funcall lt (car a) (car b)))))
       (format stream "~a" after)
       h)))

(deftest test-showh ()
  (let ((h (make-hash-table)))
    (dolist (one '(apple pear banana))
      (setf (gethash (length (string one)) h) one))
   (check
     (samep (with-output-to-string (s) (showh h  :stream s))
            "4 = PEAR
             5 = APPLE
             6 = BANANA"))))
```

Figure 10.2: CODE: `lib/tricks/hash.lisp`

```
(defun line1 (s)
  (format nil "~a" (read-from-string s)))

(defun longest-string (l)
  (let ((max -1) tmp)
    (dolist (x l max)
      (let ((tmp (length (format nil "~a" x))))
        (if (> tmp max)
            (setf max tmp))))))

(defun whiteout (seq)
  (remove-if #'whitespacep  seq))

(defun whitespacep (char)
  (member char '(#\Space #\Tab #\Newline #\Page) :test #'char=))

(defun samep (form string)
  (string= (whiteout (format nil "~a" form)) (whiteout string)))

(defun lt (x y)
  (if (string-lessp (format nil "~a" x) (format nil "~a" y))
      t
      nil))

(defmacro list2string ((item list stream) &body body)
  `(with-output-to-string (,stream)
     (dolist (,item ,list) ,@body)))

(defun nchars (n &optional (char #\Space))
  (with-output-to-string (s)
    (dotimes (i n)
      (format s "~a" char ))))
```

Figure 10.3: CODE: `lib/tricks/strings.lisp`

10.3 Strings

The file *lib/tricks/strings.lisp* in Figure 10.3 defines:

line1 `(line1 string)` Returns the first line in `string`.

list2string `(list2string (item list stream) body)` Calls `body` for every `item` in `list`. Within the `body`, if print functions write to `stream` then this function will return a string storing all those prints.

longest-string `(longest-string list)` Returns the number of characters in the longest string within `list`.

lt `(lt form1 form2)` A generic sort predicate for ordering any LISP structure. Returns true if the printed representation of `form1` is the same as `form2`.

nchars `(nchars n &optional char)` Returns a string `n` long filled with `char` (defaults to space).

```
(defun align (lines &key
              (stream t) (fs " |") (rs #\Newline)
              (underline #\-) (before #\Newline) (after ""))
 (let* ((header (first lines))
        (data   (rest  lines))
        (cols   (transpose lines))
        (widths (mapcar #'longest-string cols))
        (sep "")
        (fmt   (with-output-to-string (s)
                 (dolist (width widths)
                    (format s "~a~~a<~a~~>" sep (1+ width) "~a")
                    (setf sep fs))
                 (format s "~a" rs))))
    (format stream "~a" before)
    (apply #'format `(,stream  ,fmt ,@header))
    (setf sep "")
    (dolist (w widths)
      (format stream "~a ~a" sep (nchars w underline))
      (setf sep fs))
    (format stream "~a" rs)
    (dolist (line data)
      (apply #'format `(,stream  ,fmt ,@line)))
    (format stream "~a" after)))

(deftest test-align ()
  (let ((tmp (with-output-to-string (s)
               (align '((peace love understanding)
                        (34.1 33.4 324.2)
                        (334 222222 33)) :stream s))))
    (samep tmp "
        PEACE |   LOVE | UNDERSTANDING
        ----- | ------ | -------------
         34.1 |   33.4 |          324.2
          334 | 222222 |             33
")))
```

Figure 10.4: CODE: `lib/tricks/align.lisp`

samep `(samep form string)` Returns true if the printed representation of `form` is the same as `string` (ignoring whitespace).

whiteout `(whiteout seq)` Removes all whitespace from `seq`.

whitespacep `(whitespace char)` Removes true if `char` is whitespace; i.e. is one of space, tab, newline, or page break.

The file *lib/tricks/align.lisp* in Figure 10.4 defines a simple report utility that pretty prints a lists of lists, lining up all the fields, aligned to the same column breaks.

align `(align lines &key underline stream fs rs before after)` `Lines` stores a list of list, the first line of which is expected to store a set of text headings. These headings are printed, underlined by the `underline` character. The pretty print is written on `stream` (default: standard output) with each field separated by `fs` (default "|") and each line separated by the record separator `rs` (default, new line). The strings `before` (default, newline) and `after` (default, empty string) are printed before and after the rest of the report.

For example:

```
> (align '((peace love understanding)
         (34.1 33.4 324.2)
         (334 222222 33)))

 PEACE |   LOVE | UNDERSTANDING
 ----- | ------ | -------------
  34.1 |   33.4 |         324.2
   334 | 222222 |            33
```

10.4 Maths

The file *lib/tricks/maths.lisp* in Figure 10.5 defines

between `(between n1 n2 n3)` Returns true if the number tt n2 lies between `n1` and `n2`.

e `e` Defines the constant 2.7182817...

harmonic-mean `(harmonic-mean x y)` Returns the harmonic mean of the two numbers `x` and `y`.

mean `(mean l)` Returns the mean of the list of numbers `l`

median `(median l)` Returns:

- the 50% percentile value a list of numbers `l`;
- the *spread* of that list, i.e. the 75% percentile minus the 50% percentile;
- and the 75% point and the 25% percentile point.

```
(defconstant e (exp 1))

(defun sum (l)
  (let ((n 0))
    (dolist (one l n) (incf n one))))

(defun mean (l)
  (let ((n 0) (sum 0))
    (dolist (one l (/ sum n))
      (incf sum one)
      (incf n))))

(defun harmonic-mean (x y)  (/ (* 2 x y) (+ x y)))

(defun weighted-sum (l total)
  (sum (mapcar
        #'(lambda (one)
            (let ((freq   (first one))
                  (weight (rest one)))
              (* (/ freq total) weight)))
        l)))

(defun stdev (n sum sumSq)
  (sqrt (/ (- sumSq(/ (* sum sum) n)) (- n 1))))

(defun list2stdev (l)
  (let ((n 0) (sum 0) (sumSq 0))
    (dolist (x l (stdev n sum sumSq))
      (incf n)
      (incf sum x)
      (incf sumSq (* x x )))))

(defun median (nums)
  (labels ((mean (x y) (/ (+ x y ) 2)))
    (let* ((n1         (sort nums #'<))
           (l          (length n1))
           (mid        (floor (/ l 2)))
           (midval     (nth mid  n1))
           (25percent  (nth (floor (* l 0.25)) n1))
           (75percent  (nth (floor (* l 0.75)) n1))
           (50percent  (if (oddp l)
                           midval
                           (mean midval (nth (- mid 1) n1)))))
      (values 50percent
              (- 75percent 50percent)
              75percent
              25percent))))

(defun between (a b c)  (<= a b c))

(deftest test-stats ()
  (let ((nums '(1 1 1 1 2 2 4 100)))
    (check
      (= 112 (sum nums))
      (= 3/2 (median nums))
      (= 14  (mean nums)))))
```

Figure 10.5: CODE: `lib/tricks/maths.lisp`

list2stdev `(list2stdev l)` Returns the standard deviation of a list of numbers `l`.

stdev `(stdev n sum sumSq)` Given `n` numbers summing to `sum` and whose squares sum to `sumSq`, return the standard deviation. A sub-routine called by `list2stdev`

sum `(sum l` Returns the sum of a list of numbers `l`.

weighted-sum `(weighted-sum l total)` Returns the weighted sum of the list of cons `((frequency1 . weight1) (frequency1 . weight2)..)` as follows:

$$\sum_i \left(\frac{frequency_i}{total} * weight_i \right)$$

10.5 Lists

The file *lib/tricks/lists.lisp* in Figure **??** defines

allbut `(allbut l n)` Returns the list `l` without the item at position `n`.

doitems `(doitems (item index list &optional out) body)` (macro) Execute `body` for each `item` at position `index` from `list`. Afterwards, return `out` (default, nil).

geta `(geta key list &optional default)` (macro) Returns the value associated with `key` from the association list `l`. If no such value exists, return `default`.

list2array `(list2array l)` Return an array of same size as the list `l`, with all that items lists copied into the array.

rank `(rank l)` Return a hash table showing the rank of all numbers in the list `l`. Assumes that `l` is sorted. For example:

```
(showh (rank '(10 11 11 12 12 46 68 68 68 212 1000)))
10 = 1
46 = 6
68 = 8
11 = 5/2
12 = 9/2
212 = 10
1000 = 11
```

Note that repeated numbers are assigned the average rank of all the repeats.

remove-nths `(remove-nths doomed l)` Returns a list without the items of `l` at the positions listed in `doomed`. For example:

```
> (remove-nths '(1 3 5) '(i have gone to seek a great perhaps))
(I GONE SEEK GREAT PERHAPS)
```

```
(defun geta (key list &optional default)
  (or (cdr (assoc key list))
      default))

(defmacro doitems ((one n list &optional out) &body body )
  `(let ((,n -1))
     (dolist (,one ,list ,out)  (incf ,n) ,@body)))

(defun list2array (l)
  (let* ((len (length l))
         (a   (make-array `(,len))))
    (doitems (one n l a)
      (setf (aref a n) one))))

(defun shuffle (l)
  (dotimes (i (length l) l)
    (rotatef
     (elt l i)
     (elt l (random (length l))))))

(defun tally (l &key (key #'eql))
  (let ((h (make-hash-table :test key)))
    (dolist (one l h)
      (incf (gethash one h 0)))))

(defun remove-nths (doomeds l &optional (n 0))
  (if (null doomeds)
      l
      (remove-nths1 doomeds (first l) (rest l) n)))

(defun remove-nths1 (doomeds first rest n)
  (if (= n (first doomeds))
      (remove-nths (rest doomeds)        rest (1+ n))
      (cons first (remove-nths doomeds rest (1+ n)))))

(deftest test-remove-nths ()
  (check
    (equal (remove-nths '(0 1 5) '(a b c d e f g)) '(c d e g))
    (equal (remove-nths '(0 1 5 6) '(a b c d e f g)) '(c d e))
    (equal (remove-nths '() '(a b c d e f g)) '(a b c d e f g))))

(defun rank (l &key (ranks (make-hash-table)) (n 0))
  (if (null l)
      ranks
      (let (repeats sum now)
        (labels ((walk () (incf n) (pop l))
                 (new  () (setf repeats 1) (setf sum n))
                 (same () (incf sum n) (incf repeats))
                 (spin () (when (eql now (car l))
                            (walk) (same) (spin))))
          (setf now (walk))
          (new)
          (spin)
          (setf (gethash now ranks) (/ sum repeats))
          (rank  l :ranks ranks :n n )))))

(defun allbut (l n)
  (if (zerop n)
      (values (rest l) (first l))
      (cons (first l) (allbut (rest l) (1- n)))))
```

Figure 10.6: CODE: `lib/tricks/list.lisp`

```
;; matrix stuff
(defun transpose (x)
   (apply #'mapcar (cons #'list x)))

(defun lists2m (ls)
  (let* ((height (length ls))
         (width  (length (first ls)))
         (a      (make-array `(,height ,width))))
    (doitems (row rown ls a)
      (doitems (cell coln row)
        (setf (aref a rown coln) cell)))))
```

Figure 10.7: CODE: `lib/tricks/matrix.lisp`

Note that it is assumed that (a) the list `doomed` is sorted in ascending order and that (b) the list positions start at zero.

shuffle `(shuffle l)` Returns a list with all the items in list `l` in a random order. Caution: the original list `l` is destructively modified.

tally `(tally l &key key)` Return a hash table with one key for every unique item in the list `l`. Each key will contain a count of the number of relates of each item in `l`. For example:

```
(showh (tally '(a b r c a d a b r a)))
A = 4
B = 2
C = 1
D = 1
R = 2
```

Note that the hash table computers item equality via `key` (default, #'eql).

10.6 Matrix

The file *lib/tricks/matrix.lisp* in Figure 10.7 defines

transpose `(transpose l)` Returns a list of lists, with all items transposed. For example:

```
> (let ((programmers        '(tim tom tam))
        (favorite-languages '(lisp ruby prolog))
        (ages               '(47 21 56)))
    (transpose (list programmers favorite-languages ages)))

((TIM LISP 47) (TOM RUBY 21) (TAM PROLOG 56))
```

lists2m `(lists2m l)` Return a two-dimensional matrix filled in from the list of lists `t`.

10.7 Random

When debugging code containing random selections, it is useful to recreate a prior run that lead to an error. Hence, most languages offer a *pseudo-random* number generator which generates a sequence pf random numbers from some *seed*. Exactly the same sequence of "random" numbers can be recreated by resetting the seed and re-running the random sequence.

Sadly, I can't make LISP's built in random number generator work that way[1]. So the file *lib/tricks/random.lisp* in Figure 10.8 implements a simple standard pseudo-random number generator:

`(my-random n)` Return a floating point number in the range $0 \leq x < n$. *my-random*

`(my-random-int n)` Return an integer in the range $0 \leq x < (n-1)$. *my-random-int*

`(reset-seed)` Reset the seed to a built in initial state. *reset-seed*

[1] Yes, I've tried reseting `*random-state*` and that did not work. Also, I began to worry that what ever I did would not be portable to other LISPs.

```
(defparameter *seed0* 10013)
(defparameter *seed*  *seed0*)

(defun reset-seed () (setf *seed* *seed0*))

(defun park-miller-randomizer ()
  "The Park-Miller multiplicative congruential randomizer
  (CACM, October 88, Page 1195). Creates pseudo random floating
  point numbers in the range 0.0 < x <= 1.0."
  (let ((multiplier
         16807.0d0);16807 is (expt 7 5)
        (modulus
         2147483647.0d0)) ;2147483647 is (- (expt 2 31) 1)
    (let ((temp (* multiplier *seed*)))
      (setf *seed* (mod temp modulus))
      (/ *seed* modulus))))

(defun my-random (n)
  "Returns a pseudo random floating-point number
  in range 0.0 <= number < n"
  (let ((random-number (park-miller-randomizer)))
    ;; We subtract the randomly generated number from 1.0
    ;; before scaling so that we end up in the range
    ;; 0.0 <= x < 1.0, not 0.0 < x <= 1.0
    (* n (- 1.0d0 random-number))))

(defun my-random-int (n)
  "Returns a pseudo-random integer in the range 0 <= n-1."
  (let ((random-number (/ (my-random 1000.0) 1000)))
    (floor (* n random-number))))

(defun random-demo ()
  (let (counts out)
    (labels
        ((sorter (x y) (< (car x) (car y)))
         (zap    ()    (setf out nil)
                       (reset-seed)
                       (setf counts (make-hash-table)))
         (inc    (n)   (setf (gethash n counts)
                             (1+  (gethash n counts 0))))
         (cache  (k v) (push (list k v) out)))
      (zap)
      (dotimes (i 10000)
        (inc  (my-random-int 5)))
      (maphash #'cache counts)
      (sort out #'sorter))))

(deftest test-random ()
  (check
    (equalp (random-demo) (random-demo))))
```

Figure 10.8: CODE: `lib/tricks/random.lisp`

Chapter 11

Some Useful Destructs

11.1 Tree

The file *lib/tricks/tree.lisp* in Figure 11.1 shows a destruct for managing trees.

`(make-node &key key kids)` Returns a node in a tree with a `key` and `kids` (defaults, "nil") — *make-node*

`(node-key n)` Returns the `key` of node `n`. — *node-key*

`(node-key n)` Returns the `kids` of node `n`. — *node-kids*

11.2 Counted

The file *lib/tricks/counted.lisp* in Figure 11.2 shows an abstract data type for storing a sorted and counted set of things.

`(counted+ x c)` Adds `x` to the counted set `c`. Returns the updated set `c`. As a side-effect the `sorted` flag within `c` is cleared. — *counted+*

`(counted-all c)` Returns the contents of the counted set `c`. — *counted-all*

`(counted-n c)` Returns the number of items in the counted set `c`. — *counted-n*

`(counted-sort c pred key)` Returns the counted-set `c` with its contents sorted according to `pred` and `key`. Also, this function sets the sorted-flag of `c`. — *counted-sort*

`(counted-sorted c)` Returns the status of the contents on the counted set `c` ("t" or "nil"). — *counted-sorted*

```
(defstruct node  key kids)
```

Figure 11.1: CODE: `lib/tricks/tree.lisp`

```
(defstruct counted all (n 0) sorted)

(defun counted+ (one c)
  (incf (counted-n c))
  (push one (counted-all c))
  (setf (counted-sorted c) nil)
  c)

(defun counted-sort (c pred key)
  (unless (counted-sorted c)
    (setf (counted-all c) (sort (counted-all c) pred :key key))
    (setf (counted-sorted c) t))
  c)
```

Figure 11.2: CODE: `lib/tricks/counted.lisp`

make-caution `(make-caution &key all n sorted)` Returns a new `caution` with `b` counts (default, 0) for `all` items and the sorted flag set to `sorted` (default, "nil").

11.3 Caution

The file *lib/tricks/caution.lisp* in Figure 11.3 stores a container for warning messages. If the number of warnings exceeds a certain number (default, 20), then we say that this caution has run out of "patience", in which case it calls the LISP `error` function.

caution-all `(caution-all c)` Return all the warning messages in cautions `c`.

caution-killed `(caution-killed c)` Return "t" if the caution `c` has exceeded its patience.

caution-patience `(caution-patience c)` Returns the remaining patience in this caution.

make-caution `(make-caution)` Create a new caution.

ok `(ok test caution format-str &rest args)` If `test` fails (returns "nil"), then

- A warning message is called using `format-str` and `args`.
- `caution` losses some patience;
- If we run out of patience, the the LISP error function is called.

The standard way to use this a caution is to create one, then carry it across some extensive processing.

11.4 Line

The file *lib/tricks/line.lisp* in Figure 11.4 enables interpolating some `x` value from an arbitrary piece-wise linear function defined by `((x1 . y1) (x2 . y2) ...)`.

```
(defstruct (caution (:print-function caution-print))
  all (patience 20) killed)

(defun caution-print (c s depth)
  (declare (ignore depth))
  (format s "#(CAUTION :ALL ~a :PATIENCE ~a)"
          (caution-all c)
          (caution-patience c)))

(defun ok (test cautions format-str &rest args)
  (or test
      (let ((message (apply #'format `(nil ,format-str ,@args))))
        (push message (caution-all cautions))
        (decf (caution-patience cautions))
        (format t "% ~a~%" message)
        (when (< (caution-patience cautions) 0)
          (setf (caution-killed cautions) t)
          (error "too many warnings"))
        nil)))

(defun die (cautions format-str &rest args)
  (apply #'ok  `(nil ,cautions ,format-str ,@args))
  (setf (caution-killed cautions) t)
  (error "gasp... wheeze... rosebud... (thud)"))

(defun test-out-of-patience ()
  "can't be a defftest cause it crashes on too many errros"
  (let ((c (make-caution :patience 5)))
    (dotimes (i 5)
      (ok (= 1 2) c "bad ~a ~a" 1 2))))

(deftest test-caution ()
 (let ((c (make-caution)))
    (ok (= 1 2) c "bad ~a ~a" 1 2)
    (check
      (= 19 (caution-patience c) 19)
      (equalp "bad 1 2" (first (caution-all c))))))
```

Figure 11.3: CODE: `lib/tricks/caution.lisp`

interpolates `(interpolates x l)` Returns the y value associated with `x` over the piece-wise linear function defined by the list `l` containing `((x1.y1)(x2.y2)...)`. For example:

```
> (interpolates 25 '((0 . 0) (5 . 5) (30 . 30)))
25
```

line-b `(line-b line)` Returns the y-intercept of `line`.

line-m `(line-m line)` Returns the slope of `line`.

line-verticalp `(line-verticalp line)` Returns "t" if `line` runs vertical.

line-y `(line-y x line)` Returns the y value associated with `x` on a single `line`.

point-to-line `(point-to-line x1 y1 x2 y2)` Creates a single line that runs from `(x1,y1)` to `(x2,y2)` with slope `m` and y-intercept `b`.

```
(defstruct line  x1 y1 x2 y2 m b verticalp)

(defun point-to-line (x1 y1 x2 y2)
  (if (> x1 x2)
      (point-to-line x2 y2 x1 y1)
      (let* ((rise      (- y2 y1))
             (run       (- x2 x1))
             (verticalp (zerop run))
             m
             b)
        (unless verticalp
          (setf m    (/ rise run)
                b    (- y2 (* m x2))))
        (make-line :x1 x1 :y1  y1 :x2 x2 :y2 y2
                   :m m :b b :verticalp verticalp))))

(defun line-y (x line)
  (if (line-verticalp line)
      (line-y1 line)
      (+ (* (line-m line)  x)
         (line-b line))))

(defun interpolate (x x1 y1 x2 y2 &optional too-big )
  (cond ((<   x x1)   y1)
        ((eql x x1)   y1)
        ((eql x x2)   y2)
        ((< x1 x x2)  (line-y x
                              (point-to-line x1 y1 x2 y2)))
        (t            too-big)))

(defun interpolates (x l)
  (let* ((one  (pop l))
         (two  (first l)))
    (or (if (null l) (rest one))
        (interpolate  x (first one) (rest one)
                        (first two) (rest two))
        (interpolates x l))))

(deftest test-line ()
  (let ((line (point-to-line 0 10 1 0)))
    (check
      (samep line
        "#S(LINE :X1 0 :Y1 10 :X2 1 :Y2 0 :M -10 :B 10 :VERTICALP
NIL)")
      (= 5 (line-y 0.5 line))
      (= 25 (interpolates 25 '((0 . 0) (5 . 5) (30 . 30)))))))
```

Figure 11.4: CODE: `lib/tricks/line.lisp`

Chapter 12

Advanced Tricks

12.1 Debugging Tricks

The file *lib/tricks/debug.lisp* in Figure 12.1 contains some useful debugging tricks.

`(o &rest forms)` For each item in `forms`, print the unevaluated item and then the result of evaluating that item. This is very useful for quickly writing debug print statements. Return the result of the last evaluation. For example: *o*

```
> (let ((a 1) (b 2) (c 3)) (o a b c))
[A]=[1] [B]=[2] [C]=[3]
3
```

`(watch code)` Collect runtime statistics on `code` and all functions called by `code`. For example, the following trace shows that the code `(demo-distogram)` spends most of its time making 1000 calls to the function `(park-miller-randomizer)`: *watch*

```
> (watch (demo-distogram))

  seconds  |  consed | calls |  sec/call  |  name
-----------------------------------------------------
     0.002 | 114,680 | 1,000 |   0.000002 | PARK-MILLER-RANDOMIZER
     0.000 |  45,696 |     1 |   0.000000 | DISTOGRAM
     0.000 |   8,192 |     1 |   0.000000 | DEMO-DISTOGRAM
     0.000 |  40,960 | 1,000 |   0.000000 | DIST-ADD
     0.000 |       0 |     1 |   0.000000 | MAKE-DIST
     0.000 |       0 |     1 |   0.000000 | AS-DIST
     0.000 |       0 |     1 |   0.000000 | RESET-SEED
     0.000 |  32,768 | 1,000 |   0.000000 | MY-RANDOM
     0.000 | 203,872 |     6 |   0.000000 | NCHARS
     0.000 |       0 |     1 |   0.000000 | DIST-SORT
     0.000 |  73,728 | 1,000 |   0.000000 | MY-RANDOM-INT
-----------------------------------------------------
     0.002 | 519,896 | 4,012 |            | Total
```

```
(defmacro o (&rest l)
  (let ((last (gensym)))
  `(let (,last)
     ,@(mapcar #'(lambda(x) `(setf ,last (oprim ,x))) l)
     (terpri)
     ,last)))

(defmacro oprim (x)
  `(progn (format t "[~a]=[~a] " (quote ,x) ,x) ,x))

(defmacro watch (code)
  `(progn
    (sb-profile:unprofile)
    (sb-profile:reset)
    (sb-profile:profile ,@(my-funs))
    (eval ,code)
    (sb-profile:report)
    (sb-profile:unprofile)
    t)
)

(defun my-funs ()
  (let ((out '()))
    (do-symbols  (s)
      (if (and (fboundp s)
               (find-symbol  (format nil "~a" s) *package*)
               (not (member s *lisp-funs*)))
          (push s out)))
    out))

(defmacro time-it (n &body body)
  (let ((n1 (gensym))
        (i  (gensym))
        (t1 (gensym)))
    `(let ((,n1 ,n)
           (,t1 (get-internal-run-time)))
       (dotimes (,i ,n1) ,@body)
       (float (/ (- (get-internal-run-time) ,t1)
                 (* ,n1 internal-time-units-per-second))))))

(defun test-time-it (&key (repeats 100) (loops 100) (max 10))
  (let (out)
    (dotimes (i loops)
      (push (random max) out))
    (time-it repeats
      (list2stdev out))))
```

Figure 12.1: CODE: `lib/tricks/debug.lisp`

12.2 OS Tricks

The file *lib/tricks/os.lisp* in Figure 12.2 contains some tricks for talking to the operating system from within LISP.

`(my-command-line)` Return the command-line that called this LISP — *my-command-line*

`(my-getenv)` Return a variable set from the environment outside of LISP. — *my-getenv*

`(sh command &rest args)` Return a string containing the output from running `command` (a string) with the arguments `args` (more strings). — *sh*

`(tmp-file)` Create a new temporary file and return the name of that file as a string. — *tmp-file*

`(tmp-dir)` Create a new directory file and return the name of that directory as a string. — *tmp-dir*

```
(defun my-command-line ()
  (or
   #+SBCL *posix-argv*
   #+LISPWORKS system:*line-arguments-list*
   #+CMU extensions:*command-line-words*
   nil))

(defun my-getenv (name &optional default)
  #+CMU
  (let ((x (assoc name ext:*environment-list*
                  :test #'string=)))
    (if x (cdr x)  default))
  #-CMU
  (or #+Allegro   (sys:getenv name)
      #+CLISP     (ext:getenv name)
      #+ECL       (si:getenv name)
      #+SBCL      (sb-unix::posix-getenv name)
      #+LISPWORKS (lispworks:environment-variable name)
      default))

(defun sh (com &rest args)
  (let ((path (which com)))
    (with-output-to-string (s)
      (with-open-stream
          (files
           (sb-ext:process-output
            (sb-ext:run-program path args
                                :output s)))))))

(defun tmp-file ()
  (line1 (sh "mktemp")))

(defun tmp-dir ()
  (let* ((dir  (format nil "~a" (random 1000000000)))
         (path (make-pathname :directory
                              `(:absolute "tmp" ,dir))))
    (ensure-directories-exist path)
    (format nil "/tmp/~a" dir)))

(defun which (exe)
  (if (eql #\/ (char exe 0))
      exe
      (let* ((com   (format nil "which ~a" exe)))
        (line1  (sh "/bin/bash" "-c" com)))))
```

Figure 12.2: CODE: `lib/tricks/os.lisp`

Chapter 13

Statistical Functions

When assessing the performance of one data miner (or comparing the performance of multiple data miners) it may be required to perform many experiments and summarize the results. The functions in this chapter offer simple support for data summary, ranging from simple visualizations to a more rigorous statistical analysis.

13.1 Visualizations with "Distograms"

Distograms are combinations of a `dist` structure and a histogram generator. Distograms can be used to store and show the ranges of numbers generated by some process.

For example, here is a generator that computes, 10000 times, $\sqrt(n)$ for ranges $1 \leq n \leq 10$.

```
> (let ((log (make-dist)))
    (dotimes (i 10000)
        (let ((one (sqrt (/ (random 1000) 100))))
           (dist-add one log)))
    (distogram log :shrink 200))

  0.50 *   262 =   2% < *
  1.50 *  1950 =  19% < **********
  2.50 *  4042 =  40% < ********************
  3.50 *  3746 =  37% > ******************
```

Note how the square root function biases the generated values towards the large numbers. For example, only 2% of the generated values (262 out out 10000) are under one while 37% of the numbers (3746 out of 10000) are over three.

The file *lib/tricks/dist.lisp* in Figure 13.1 shows the dist structure. As numbers are thrown at a dist (via the `dist-add` function), a running log is kept of the max and min number seen so far, the sum of the numbers, how many numbers were seen etc.

By default, dists are *lean*; i.e. they only keep summaries of the logged numerics and not the numerics themselves (this saves on storage space). However,

```
(defstruct dist
  (min    most-positive-double-float)
  (max    most-negative-double-float)
  (bars   '())
  (sum    0)
  (sumsq  0)
  (n      0)
  (sorted t)
  (fuzz   1)
  (fat   nil)
  (all    '())
)

(defun dist-compare (d1 d2 &key (conf 95))
  "compares means of two independent samples d1 d2"
  (ttest (dist-sum d1) (dist-sumsq d1) (dust-n d1)
         (dist-sum d2) (dist-sumsq d2) (dust-n d2)
         :conf conf))

(defun as-dist (l)
  "Return a dist with all the numbers of l"
  (if (eq 'dist (type-of l))
      l
      (dist-adds l)))

(defun dist-adds (l &optional (d (make-dist)))
  "Add the numbers in the list 'l' to a dist."
  (dolist (x l d)
    (dist-add x d)))

(defun dist-mean (d)
  (/ (dist-sum d) (dist-n d)))

(defun dist-sd (d)
  (stdev (dist-n d) (dist-sum d) (dist-sumsq d)))

(defun dist-add (num &optional (d (make-dist)))
  (incf (dist-n      d))
  (incf (dist-sum    d) num)
  (incf (dist-sumsq  d) (* num num))
  (if (dist-fat d) (push num (dist-all d)))
  (if (< num (dist-min d)) (setf (dist-min d) num))
  (if (> num (dist-max d)) (setf (dist-max d) num))
  (let* ((fuzz (dist-fuzz d))
         (num1 (* fuzz (round (/ num fuzz))))
         (val  (cdr (assoc num1 (dist-bars d)))))
    (if val
        (setf (cdr (assoc num1 (dist-bars d))) (1+ val))
        (push (cons num1 1) (dist-bars d)))
    (setf (dist-sorted d) nil))
  d)

(defun dist-sort (d)
  (labels ((car< (a b) (< (car a ) (car b))))
    (unless (dist-sorted d)
      (setf (dist-bars d) (sort (dist-bars d) #'car<)
            (dist-sorted d) t))
    d))
```

Figure 13.1: CODE: `lib/tricks/dist.lisp`

if the dist is created as `fat` (see `make-dist`, below) then all the logged numbers are kept by the dist.

`(as-dist x)` Returns a dist structure. If a `x` is already a dist, then just return `x`. Else, convert the numbers in `x` into a dist. *as-dist*

`(dist-add x d)` Return the dist structure `d`, augmented by the number `n`. *dist-add*

`(dist-adds l &optional d)` Returns a dist structure summarizing the list of numbers `l`. If `d` is not supplied, store the summary in a new dist. *dist-adds*

`(dist-all d)` Return all the numbers ever stored in the dist. If the dist is *lean*, then this will be nil. *dist-all*

`(dist-compare d1 d2 &key conf)` Check if two dists are statistically different using the t-test of §**??**. *dist-compare*

`(dist-max d)` Return the maximum of the numbers added to the dist. *dist-max*

`(dist-mean d)` Return the mean value of the dist contents. *dist-mean*

`(dist-min d)` Return the minimum of the numbers added to the dist. *dist-min*

`(dist-n d)` Return the number of additions made to the log. *dist-n*

`(dist-sd d)` Return the standard deviation of the dist contents. *dist-sd*

`(dist-sort d)` Return the dist `d` with its cache of key value pairs sorted numerically using #'¡. *dist-sort*

`(make-dist &key fat fuzz)` Create a new dist structure. If `fat` is nil (which is the default), then the dist will not retain all the added values. If `fuzz` is non-nil, then all added numbers will be rounded to the nearest `fuzz` value. *make-dist*

The file *lib/tricks/distogram.lisp* in Figure 13.2 implements `distogram`, a function for pretty-printing dists:

`(distogram d &key str header lwdith rwidth decimals pad decimals)` Pretty print the dist `d` on stream `str` (default: standard output). Before the pretty print, display the `header`. Print numbers with `decimals` number of decimals. Set the width of the numeric columns with `lwidth` and `rwidth` (default for both: 5). If the display is too wide, then it can be made narrower using the `shrink` factor (default: 1). *distogram*

13.2 Plotting Tricks

Distograms are a very simple text-based method to visualize a log of data. For more complex tasks, the GNUPLOT tool simplifies the generation of elaborate n-dimensional plots such as Figure 13.3. For tutorials on GNUPLOT, see §**??**.

```
(defun distogram (d0 &key
                  header
                  (lwidth 5) (decimals 2) (rwidth 5)
                  (shrink 1)  (str t))
  (let ((d (as-dist d0)))
    (unless (dist-sorted d)
      (dist-sort d))
    (let ((sum   (dist-sum  d))
          (bars  (dist-bars d))
          (min   (dist-min  d))
          (max   (dist-max  d))
          (n     (dist-n    d))
          (fuzz  (dist-fuzz d))
          (percentSum 0)
          (fmt
           (format nil " ~~~a,~af * ~~~ad = ~~3d% ~~a "
                   lwidth decimals rwidth)))
      (if header
          (format t "~a~%" header))
      (dolist (bar bars t)
        (let* ((key     (car bar))
               (value   (cdr bar))
               (percent (floor (* 100 (/ value n))))
               (stars   (round (/ value shrink)))
               (halfp   (< percentSum 50)))
          (incf percentSum percent)
          (format str fmt (+ key (/ fuzz 2))
                          value
                          percent
                          (if halfp "<" ">"))
          (format str "~a~%" (nchars stars "*")))))))

(defun demo-distogram ()
  (let ((log (make-dist :fuzz 2)))
    (reset-seed)
    (dotimes (i 1000)
      (dist-add (sqrt (my-random-int 100)) log))
    (with-output-to-string (s)
      (distogram log :shrink 10 :str s))
    ))

(deftest test-distogram ()
  (check
    (samep (demo-distogram)
  "  1.00 *    12 =   1% < *
     3.00 *    68 =   6% < *******
     5.00 *   163 =  16% < ****************
     7.00 *   222 =  22% < **********************
     9.00 *   327 =  32% < ********************************
    11.00 *   208 =  20% > ********************")))
```

Figure 13.2: CODE: `lib/tricks/distogram.lisp`

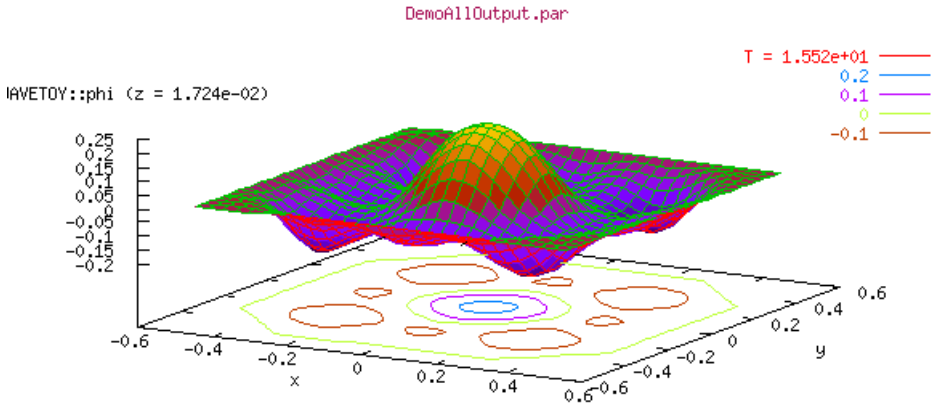

Figure 13.3: A plot generated by GNUPLOT.

The file *lib/tricks/gnuplot.lisp* in Figure 13.4 defines a simple interface from LISP to the GNUPLOT plotting program. It can be used to generate simple text-based plots.

`(gnuplot &rest strings)` Execute the GNUPLOT commands in strings. For *gnuplot* example:

```
>  (gnuplot "set terminal dumb"
           "set size 0.75,1"
           "set xrange [-5:5]"
           "set yrange [-1.2:1.2]"
           "plot sin(x)")
"
      +----+---------+---------+---------+---------+----+
      |    +         +         +         +sin(x) ****** |
    1 *****                            *****            ++
      |   **                          *    **            |
      |    **                        *      **           |
      |     *                       *        *           |
  0.5 ++     *                     **         *          ++
      |       *                    *           *         |
      |       *                   *            *         |
      |        *                  *             *        |
    0 ++       *                 *               *       ++
      |         *               *                 *      |
      |          *             *                  *      |
      |          *            *                   *      |
 -0.5 ++          *          **                     *     ++
      |            *        *                        *    |
      |            **      *                         **   |
      |              **    *                          **  |
   -1 ++               *****                           *****
      |    +         +         +         +         +    |
      +----+---------+---------+---------+---------+----+
          -4        -2         0         2         4"
```

13.3 Performance Statistics

Suppose we have some historical log that can comment on the correct prediction of each row. Suppose some device that predicts labels on test data. By comparing those predictions with historical logs, we can define several measures of success. Let $\{A, B, C, D\}$ denote the true negatives, false negatives, false positives, and true positives (respectively) found by a binary detector (binary detectors work on data sets with two classes, like Figure **??**). A, B, C, D can be combined in many ways. For example, accuracy (or *acc*) is the percentage of true positives (D) and negatives (A) found by the detector.

$$acc = accuracy = (A + D)/(A + B + C + D) \tag{13.1}$$

Also, recall (or *pd*) comments on how much of the target was found.

$$pd = recall = D/(B + D) \tag{13.2}$$

Precision (or *prec*) comments on how many of the instances that triggered the detector actually containing the target concept.

$$prec = precision = D/(D + C) \tag{13.3}$$

In practice, engineers *balance* between pf and pd. To operationalize this notion of *balance*, we define *bal* to be the Euclidean distance from the sweet

```
(defun gnuplot (&rest coms)
 (let ((str  (with-output-to-string (s)
               (format s "~a<<EOF~%" (which "gnuplot"))
               (dolist (com coms)
                 (format s "~a~%" com))
               (format s "~a" "EOF")))
       (file   (tmp-file)))
   (with-open-file (s file :direction :output
                           :if-exists :overwrite)
     (format s "~a~%" str))
   (prog1
       (sh "bash" file)
     (delete-file file))))

(defun demo-gnuplot1 ()
  (gnuplot "set terminal dumb"
           "set size 0.75,1"
           "set xrange [-5:5]"
           "set yrange [-1.2:1.2]"
           "plot sin(x)"))

(defun demo-gnuplot2-mac-osx ()
  (let* ((file (format nil "~a/plot.png" (tmp-dir))))
    (gnuplot "set terminal pdf"
             (format nil "set output '~a'~%" file)
             "plot sin(x)")
    (sh "xpdf" file  "&")))

(deftest test-gnuplot ()
  (check
   (samep (demo-gnuplot1) "
      +----+---------+---------+---------+---------+----+
      |    +         +         +         +sin(x) ****** |
    1 *****                             *****           ++
      |   **                          *    **            |
      |    **                        *      **           |
      |     *                       *        *           |
  0.5 ++     *                     **         *          ++
      |       *                    *          *          |
      |       *                   *            *         |
      |        *                  *             *        |
    0 ++       *                 *               *       ++
      |         *               *                 *       |
      |          *              *                 *       |
      |          *             *                   *       |
 -0.5 ++          *          **                     *     ++
      |            *        *                       *     |
      |            **      *                        **    |
      |             **    *                           **   |
   -1 ++              *****                            *****
      |    +         +         +         +         +    |
      +----+---------+---------+---------+---------+----+
          -4        -2         0         2         4")))
```

Figure 13.4: CODE: `lib/tricks/gnuplot.lisp`

		module found in defect logs?	
		no	yes
signal prediction	no (i.e. $v(g) < 10$)	A = 395	B = 67
	yes (i.e. $v(g) \geq 10$)	C = 19	D = 39

$pd =$	$Prop.detected =$	37%
$pf =$	$Prob.falseAlarm =$	5%
$prec =$	$precision$	= 67%
$Acc =$	$accuracy =$	83%
$bal =$	$Balance =$	45%

Figure 13.5: A ROC sheet assessing the predictor $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules that fall into each cell of this ROC sheet.

spot $pf = 0, pd = 1$ to a pair of $< pf, pd >$. For convenience, we (a) normalize *bal* by the maximum possible distance across the ROC square ($\sqrt{2}$); (b) subtract this from 1; and (c) express it as a percentage; i.e.

$$balance = bal = 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}} \tag{13.4}$$

Hence, better and *higher* balances fall *closer* to the desired sweet spot of $pf = 0, pd = 1$. See Figure **??** for an example of these calculations.

Some measures, like balance, combine the rest. Another commonly used combiner is the the f-measure. This measure is the harmonic mean of precision and recall. It has the property that if *either* precision or recall is low, then the f-measure is decreased. The f measure is useful for dual assessments that include *both* precision and recall.

$$f\text{-}measure = \frac{2 \cdot prec \cdot pd}{prec + pd} \tag{13.5}$$

All these measures fall in the range $0 \leq \{pd, pf, prec, f, acc, balance\} \leq 1$. Also, except for pf the *larger* these values, the better the model.

One detail before continuing: if the predictions are not just binary, then they must be divided by extracting one binary problem for each prediction labels. For example here are predictions for three labels, each row describes data from one label and each column shows the labels assigned to each row.

```
X Y Z    <--- predicted as
-----
1 2 3   X
3 2 1   Y
0 4 5   Z
```

(Note that the above report is called a "confusion matrix".)

These ternary predictions have to be divided into the three binary sets of predictions for $(X, \neg X)$, $(Y, \neg Y)$ and $(Z, \neg Z)$:

```
(defstruct (abcd (:print-function abcd-print))
  for (a 0) (b 0) (c 0) (d 0))

(labels ((a (x) (abcd-a x))
         (b (x) (abcd-b x))
         (c (x) (abcd-c x))
         (d (x) (abcd-d x)))
  (defun pd     (x) (if (zerop (d x)) 0 (/ (d x) (+ (b x) (d x)))))
  (defun pf     (x) (if (zerop (c x)) 0 (/ (c x) (+ (a x) (c x)))))
  (defun all    (x) (+ (a x) (b x) (c x) (d x)))
  (defun recall    (x) (pd x))
  (defun precision (x) (if(zerop (d x)) 0 (/ (d x) (+ (c x)(d x)))))
  (defun accuracy  (x) (if (and (zerop (a x)) (zerop (d x))) 0
                           (/ (+ (a x) (d x)) (all x))))
  (defun f-measure (x) (if (or (zerop (a x)) (zerop (d x))) 0
                           (harmonic-mean (pd x) (pf x))))
  (defun balance   (x &optional (goalpd 1) (goalpf 0))
    (- 1 (/ (sqrt (+  (expt (- goalpf (pf x)) 2)
                      (expt (- goalpd (pd x)) 2)))
            (sqrt 2))))
)

(defun abcd-stats (pairs &key (verbose t))
  "stats from cons of (want . got)"
  (let* ((h (make-hash-table :test #'equal))
         classes out)
    (dolist (pair pairs)
      (unless (member (first pair) classes)
        (push (first pair) classes))
      (unless (member (rest  pair) classes)
        (push (rest pair)  classes))
      (incf (gethash pair h 0))); while here, collect abcd-stats
    (if verbose (abcd-matrix pairs classes h))
    (dolist (class classes out)
      (unless (null class)
        (let ((abcd (make-abcd :for class)))
          (maphash ; for each item in the hash counts, do
           #'(lambda (pair count)
               (abcd-stat (first pair) (rest pair)
                          count class h abcd))
           h)
          (push abcd out))))))

 (defun abcd-stat (want got count goal h abcd)
   (if (eql got goal)
       (if (eql want goal)
           (incf (abcd-d abcd) count)
           (incf (abcd-c abcd) count))
       (if (eql want goal)
           (incf (abcd-b abcd) count)
           (incf (abcd-a abcd) count))))
```

Figure 13.6: CODE: `lib/tricks/abcd.lisp`

```
notX X             notY Y             notZ Z
------             ------             ------
 12  3 notX    |     9  6 notY    |     8  4  notZ
  5  1 X       |     4  2 Y       |     4  5  Z
```

The file *lib/tricks/abcd.lisp* in Figure 13.6 defines the abcd structure that stores code for generating performance statistics on some device that predicts labels on test data.

abcd-stats `(abcd-stats pairs &key verbose)` Given a list of pairs of the form `((got1 . want1) (got2 . want2) ..)`, `abcd-stats` returns a list of `abcd` structures storing the (a, b, c, d) scores for every label seen in the pairs. For example:

```
> (abcd-stats '((x . x) (x . x) (x . y) (x . z)
                (x . z) (y . x) (y . x) (y . x) (y . y)
                (z . y) (z . y) (z . x)))

(
#(ABCD :FOR X :A 3 :B 3 :C 4 :D 2 :ACC .42  :PD .40 :PF .57
                                  :PREC .33 :F .47  :BAL .41)
#(ABCD :FOR Y :A 5 :B 3 :C 3 :D 1 :ACC .50  :PD .25 :PF .38
                                  :PREC .25 :F .30  :BAL .41)
#(ABCD :FOR Z :A 7 :B 3 :C 2 :D 0 :ACC .58  :PD .00 :PF .22
                                  :PREC .00 :F .00  :BAL .28)
)
```

Note that the `abcd-print` function for computes and shows the accuracy, pd, pf, precision, f-measures, and balance statistics. If the `verbose` flag is "t", then the `abcd-stats` function also prints the confusion matrix:

```
a    b    c  <--- classified as
0    2    1 | a = Z
0    1    3 | b = Y
2    1    2 | c = X
```

The `abcd-print` function and the functions that print the confusion matrix are found in file *lib/tricks/abcdmatrix.lisp* (see Figure 13.7).

It is possible to write other reports (apart from the confusion matrix) using the statistics calculators defined for the abcd structure:

accuracy `(accuracy abcd)` Returns `abcd`'s accuracy.

all `(all abcd)` Returns the sum of the counts in `abcd`.

balance `(balance abcd)` Returns `abcd`'s balance.

f-measure `(f-measure abcd)` Returns `abcd`'s f-measure.

pd `(pd abcd)` Returns `abcd`'s probability of detection (a.k.a. recall).

pf `(pf abcd)` Returns `abcd`'s probability of false alarm.

precision `(precision abcd)` Returns `abcd`'s precision.

```
(defun abcd-print (x s d)
  (declare (ignore x))
  (format s "#(ABCD :FOR ~a :A ~a :B ~a :C ~a :D ~a :ACC ~2,2f "
          (abcd-for x) (abcd-a x) (abcd-b x)
          (abcd-c x) (abcd-d x) (accuracy x))
  (format s ":PD ~2,2f :PF ~2,2f :PREC ~2,2f :F ~2,2f :BAL ~2,2f)"
          (pd x) (pf  x) (precision x)
          (f-measure x) (balance   x)))

(defun abcd-matrix (pairs classes h)
  (let ((n 0)
        renames
        (syms '((0  . a)(1  . b)(2  . c) (3 . d) (4 . e)
                (5  . f)(6  . g)(7  . h) (8 . i) (9 . j)
                (10 . k)(11 . l)(12 . m)(13 . n)(14 . o)
                (15 . p)(16 . q)(16 . r)(18 . s)(19 . t)
                (20 . u)(21 . v)(22 . w)(23 . x)(24 . y)
                (25 . z)(26 . aa)(27 . ab) (28 . ac)(29 . ad)
                (30 . ae)(31 . af)(32 . ag)(33 . ah)(34 . ai)
                (35 . aj)(36 . ak)(37 . al)(38 . am)(39 . an)
                (40 . ao)(41 . ap)(42 . aq)(43 . ar)(44 . as)
                (45 . at)(46 . au)(47 . av)(48 . aw)(49 . ax)
                (50 . ay)(51 . az))))
    (showh h)
    (terpri)
    (doitems (class i classes)
      (format t "~5<~(~a~)~>" (cdr (assoc i syms)))
      (push (cons class (cdr (assoc i syms ))) renames))
    (format t "  <--- classified as~%")
    (setf n -1)
    (doitems (class1 i1 classes)
      (incf n)
      (doitems (class2 i2 classes)
        (format t "~5d" (gethash (cons class1 class2) h 0)))
      (format t " | ~(~a~) = ~a~%" (cdr (assoc i1 syms)) class1))))
```

Figure 13.7: CODE: `lib/tricks/abcdmatrix.lisp`

```
(deftest test-abcd ()
  (let* ((tmp (abcd-stats '((a . a) (a . a) (a . b) (a . c) (a . c)
                            (b . a) (b . a) (b . a) (b . b)
                            (c . b) (c . b) (c . a)) :verbose nil))
         (b   (second tmp)))
    (check
      (samep tmp "(
     #(ABCD :FOR A :A 3 :B 3 :C 4 :D 2 :ACC .42
                :PD .40 :PF .57 :PREC .33 :F .47 :BAL .41)
     #(ABCD :FOR B :A 5 :B 3 :C 3 :D 1 :ACC .50
                   :PD .25 :PF .38 :PREC .25 :F .30 :BAL .41)
     #(ABCD :FOR C :A 7 :B 3 :C 2 :D 0 :ACC .58
                   :PD .00 :PF .22 :PREC .00 :F .00 :BAL .28))")
      (= (accuracy  b) (/ 6 12))
      (= (all       b) 12)
      (= (pd        b) (/ 1 4))
      (= (pf        b) (/ 3 8))
      (= (precision b) (/ 1 4))
      (= (f-measure b) (/ 3 10))
      (= (balance   b) 0.40707296)
      )))
```

Figure 13.8: CODE: `lib/tricks/abcdtest.lisp`

recall `(recall abcd)` Returns `abcd`'s recall (a.k.a. probability of detection).

See also the file *lib/tricks/abcdtest.lisp* (see Figure 13.8 for some demo code.
.

13.4 Hypothesis Testing

When the visual pattern is not clear (or misleading) is it useful to apply some decision procedure that concludes if the performance scores from one treatment is the same, better, or worse than another. Ideally, the decision procedure has been well-studied and (within the community of people that you are reporting too), has been accepted as an appropriate and acceptable method for judging performance output.

There are any number of statistical decision procedures and we list four in Figure 13.9. These appropriate tests are selected by considering:

- Where the performance scores collected by different treatments on the same population?
- Are we comparing the actual performance scores or the *ranked* scores?

Regarding the first point, consider kicking a car tire to gauge air pressure using N different boots. The boots are the different treatments applied to the same population (the car tire). On the other hand, if we use two instruments to measure airport passenger height , the both the treatment is different as well as

	Values	
populations?	actual?	ranked?
same	T-tests 1	Wilcoxon
different	T-tests 2	Mann-Whitney

Figure 13.9: Selecting statistical tests based on properties of the task. T-tests (types 1 and 2), Wilcoxon, and Mann-Whitney procedures are described later in this text.

the population being measured (since there is improbable that exactly the same people will walk past both detectors).

Turning now to the issue of *actual* vs *ranked* scores, consider two treatments A, B that have generated eleven performance measures. Treatment A has generated $N_1 = 5$ values {5,7,2,0,4} and treatment B has generated $N_2 = 6$ values {4,8,2,3,6,7}. The mean actual performance of treatment A is higher than the mean of B:

```
> (float (mean '(5 7 2 0 4)))
3.6

> (float (mean '(4 8 2 3 6 7)))
5.0
```

Such mean actual measures are inappropriate when (e.g.) the performance scores contain large outliers. For example, if Mr. Bill Gates (a famous software multi-billionaire) was placed in a room with 20 unemployed people, then the mean actual annual income in that room with be around $2 billion. This actual number characterizes neither the financial situation of Mr. Gates or the other 20 unemployed people in that room.

An alternative to actual values are their ranks. Returning to the above example, we can rank the numbers in treatment A and B by first sorting the numbers and labeling according to which treatment generated them:

Samples	A	A	B	B	A	B	A	B	A	B	B
Values	0	2	2	3	4	4	5	6	7	7	8

The numbers can now be ranked, left to right, first rank to last rank. On ranking, averages are used when values are the same:

Samples	A	A	B	B	A	B	A	B	A	B	B
Values	0	2	2	3	4	4	5	6	7	7	8
Ranks	1	2.5	2.5	4	5.5	5.5	7	8	9.5	9.5	11

Note that, when ranked in this manner, the largest value (8 in this case) gets the same rank even if it was ten to a hundred times larger. That is, such rank tests are less susceptible to large outliers.

13.4.1 T-tests 1

The file *lib/tricks/ttest.lisp* in Figure 13.10 implements the test that two treatments generating actual values are similar or different. This code assumes that

```
(defun ttest-from-lists (one two &optional (conf 95))
  (let ((as 0) (asq 0) (an 0)(bs 0) (bsq 0) (bn 0))
    (dolist (a one)
      (incf an) (incf as a) (incf asq (* a a)))
    (dolist (b two)
      (incf bn) (incf bs b) (incf bsq (* b b)))
    (ttest as asq an bs bsq bn :conf conf)))

(defun ttest (as asq an bs bsq bn &key (conf 95))
  (labels
      ((less ()  (< (/ as an) (/ bs bn)))
       (same ()  (let* ((tcrit    (tcritical (+ an bn -2) conf))
                        (ssa      (- asq (/ (* as as) an)))
                        (ssb      (- bsq (/ (* bs bs) bn)))
                        (pooled   (/ (+ ssa ssb) (+ bn  an -2)))
                        (sxasb    (sqrt (* pooled (+ (/ 1 an)
                                                     (/ 1 bn)))))
                        (tvalue   (abs (/ (- (/ bs bn) (/ as an))
                                          sxasb))))
                   ;(o tcrit tvalue)
                   (> tcrit tvalue))))
    (cond ((same)  0)    ; H0  : mean of a same as mean of b
          ((less) -1)    ; H1a : mean of a <       mean of b
          (t       1)))) ; H1b : mean of a >       mean of b

(let ((ttable '((95 . ((   1  . 12.70   )
                       (   3  .  3.1820 )
                       (   5  .  2.5710 )
                       (  10  .  2.2280 )
                       (  20  .  2.0860 )
                       (  80  .  1.99   )
                       ( 320  .  1.97   )))
                (99 . ((   1  . 63.6570 )
                       (   3  .  5.8410 )
                       (   5  .  4.0320 )
                       (  10  .  3.1690 )
                       (  20  .  2.8450 )
                       (  80  .  2.64   )
                       ( 320  .  2.58   ))))))
 (defun tcritical (n conf)
   (interpolates n (geta conf ttable))))

;www.cas.buffalo.edu/classes/psy/segal/2072001/ttests/t-tests1.html
(defun ttest-demo (&optional (fudge 1))
  (let ((one '(105 112  96 124 103  92 97  108 105 110))
        (two '( 98 108 114 106 117 118 126 116 122 108)))
    (setf one (mapcar #'(lambda (x) (* x fudge)) one))
    (ttest-from-lists one two)))

(deftest test-ttest ()
  (check
    (= 0 (ttest-demo))
    (= 0 (ttest-demo 1.1))
    (= 1 (ttest-demo 1.2))))
```

Figure 13.10: CODE: `lib/tricks/ttest.lisp`

the values are normally distributed (the dreaded bell-shaped curve) and that the data comes from different treatments being applied to the same data.

Internally, the test works by comparing some summary value against a *critical* t-test value.

`(tcritical n conf)` Return the critical value for a population of size `n` at confidence `conf`. The only two confidences defined are 95% and 99%. *tcritical*

`(ttests-from-lists a b &optional conf)` *ttest-from-lists*

`(ttest as asq an bn bsq bn &key conf)` Return 0,-1,1 if the values in A are statistically the same, less than, more than treatment B. The populations are characterized by their sum `as,bs`, the sum of the squares of their items `asq bsq`, and the number of items in each population `an,bn`. Perform at some statistical level of confidence `conf` (default: 95%). The only two confidences defined are 95% and 99%. *ttest*

`(ttests-from-lists a b &optional conf)` Perform at some statistical level of confidence `conf` (default: 95%). The only two confidences defined are 95% and 99%. Return the result of running a `ttest` on results from two lists of treatments: `a`, `b`. *ttest-from-lists*

13.4.2 Mann-Whitney

The file *lib/tricks/mannwhitney.lisp* in Figure 13.11 defines code for testing if the ranks of the performance of treatment A are the same, less than, or greater than treatment B/

`(mann-whitney a b &optional conf)` Return 0,-1,1 if the ranks of the values in A are statistically the same, less than, more than treatment B. Perform at some statistical level of confidence `conf` (default: 95%). The only two confidences defined are 95% and 99%. *mann-whitney*

```
(defun mann-whitney (a b &optional (conf 95))
  "The 'method 1' of http://faculty.vassar.edu/lowry/ch11a.html."
  (labels ((as-ranks (l r) (mapcar #'(lambda (x) (gethash x r)) l))
           (sum      (l)   (let ((s 0)) (dolist (x l s)(incf s x))))
           (median   (l n) (let ((sorted (sort l #'<))
                                 (midv (floor (/ n 2))))
                             (if (oddp n)
                                 (nth midv sorted)
                                 (/ (+ (nth (1- midv) sorted)
                                       (nth  midv
sorted)) 2)))))
    (let* ((all    (sort (copy-list (append a  b)) #'<))
           (ranks  (rank all))
           (ranksa (as-ranks a ranks))
           (ranksb (as-ranks b ranks))
           (na     (length a))
           (nb     (length b))
           (n      (+ na nb))
           (tcrict (tcritical n conf))
           (suma   (* 1.0 (sum ranksa)))
           (ta     (/ (* na (+ n 1)) 2.0))
           (sigma  (sqrt (/  (* na nb (+ n 1)) 12.0)))
           (za     (/ (+ (- suma ta) 0.5) sigma)))
                                      ;
(o suma ta sigma tcrict) ;debug line
      (cond  ((< (abs za) tcrict)                          0)
             ((< (median ranksa na) (median ranksb nb)) -1)
             (t                                            1))
      )))

(defun mann-whitney-demo-small ()
  (mann-whitney '(4.6   4.7   4.9  5.1   5.2   5.5
                  5.8   6.1   6.5  6.5   7.2)
                '(5.2   5.3   5.4  5.6   6.2   6.3
                  6.8   7.7   8.0  8.1)))

(defun mann-whitney-demo-big (&optional (fudge 1))
  "Generate two lists of 10000 random floats. Multiple the
  second  list by 'fudge'. Check if the median ranks of list one
  is the same as list two (return '0'), smaller than list two
  (return '-1'), or larger than list two (return '1')."
  (labels ((big (n s) (let (out)
                        (dotimes (i n out)
                          (push (* s (round (random 100))) out)))))
    (let ((one (big 10000 1))
          (two (big 10000 fudge)))
      (mann-whitney one two))))

(deftest test-mann-whitney ()
  (check
    (= 0 (mann-whitney-demo-small))
    (= 0 (mann-whitney-demo-big))
    (= 1 (mann-whitney-demo-big -2))
    (= -1 (mann-whitney-demo-big 2))
  ))
```

Figure 13.11: CODE: `lib/tricks/mannwhitney.lisp`

13.4.3 Wilcoxon

The file *lib/tricks/wilcoxon.lisp* in Figure 13.12 and *lib/tricks/wilcoxon2.lisp* in Figure 13.13 defines code for testing if there is a significant difference between the two lists of data ($X.A$ and $X.B$).

`(wilcoxon x.a x.b &optional (signum 95)` If there is a significant difference, 1 is returned. Else, 0 is returned. Wilcoxon works when comparings lists with more than 5 values. Acceptable `signum` values are 90, 95, 98, 99, 99.9. *wilcoxon*

13.4.4 Fastmap

The file *lib/tools/pre/fastmap.lisp* in Figure 13.14 defines code for reducing the dimensions of a given set of data while attempting to preserve the distances between the items.

`(fastmap k datamatrix)` `k` is the number of dimensions you want fastmap to reduce `datamatrix` to. *fastmap*

13.4.5 K-means

The file *lib/tools/pre/kmeans.lisp* in Figure 13.15 defines code for clustering data into k distinct clusters.

`(k-means k data)` `k` is the number of clusters you want the function to find. `data` is the matrix of data that is to be clustered. Clusters are determined by the euclidean distance between items in the matrix. *k-means*

```
;;;;Peter Santiago, Started June 16th 2008
;;;;Wilcoxon algorithm and information shamelessly stolen from
;;;;Richard Lowry:
;;;;http://faculty.vassar.edu/lowry/ch12a.html

(defun wilcoxon (x.a x.b &optional (signum 95))
  "Runs a wilcoxon signed-rank test"
  (labels ((combine (a &rest elements) `(,a ,@ elements))
           (as-ranks (l r) (mapcar #'(lambda (x) (gethash x r)) l))
           (simple-quad (c) (/ (+ -1 (sqrt (- 1 (* -1 8 c)))) 2)))

    (let* ((x.dif (mapcar #'- x.a x.b)) ;x.a-x.b
           (x.abs (mapcar #'abs x.dif)) ;absolute value of x.dif
           (x (sort (mapcar #'combine x.abs x.dif x.b x.a)
                    #'< :key #'first)) ;sorts by absolute value
           (ranks (as-ranks (mapcar #'first x)
                            (wilcoxon-rank (mapcar #'first x))))
                                        ;gets the ranks
           (x (mapcar #'cons ranks x)) ;adds absolute ranks to x
           (x (mapcar #'cons
                      (mapcar #'*
                              (mapcar #'first x)
                              (mapcar #'signum
                                      (mapcar #'third x))) x))
                                        ;adds signed ranks
           (n (simple-quad
               (loop for item in ranks
                     summing item))) ;number of ranks used
           (w (loop for item in (mapcar #'first x)
                    summing item)) ;total weight of all signed ranks

           (z (/ (- w .5) (sqrt (/ (* n (+ n 1) (+ (* 2 n) 1)) 6))))
                                        ;z-value

      (test-significance signum n w z))))

(deftest test-wilcoxon ()
  "runs wilcoxon on data with known results."
  (check
   (= 1 (wilcoxon '(72 90 40 84 22 78 50 64 50 30 52 64 45 64 24 78)
                  '(32 58 20 68 36 68 40 56 44 25 56 68 48 62 24 78)
                  95)
           )))
```

Figure 13.12: CODE: `lib/tricks/wilcoxon.lisp`

```
(defun test-significance (signum total.ranks weight z)
  "Tests for significance: 0 if not, 1 if"
  (let ((val 0))
    (cond ((< total.ranks 5))
          ((< total.ranks 10)
           (< weight (get-wcrit signum total.ranks))
           (if (> weight (get-wcrit signum total.ranks))
               (setf val 1)))
          ((< 10 total.ranks)
           (< z (get-zcrit signum))
           (if (> z (get-zcrit signum))
               (setf val 1))))
    val))
(defun get-zcrit (signum)
  "for tests with greater than 10 ranks, use this table"
  (let ((table '((90   1.645)
                 (95   1.960)
                 (98   2.326)
                 (99   2.576)
                 (99.9 3.291))))
    (dotimes (i (list-length table))
      (if (= (first (nth i table)) signum)
          (return (second (nth i table)))))))
(defun get-wcrit (signum total.ranks)
  "for tests run with 5-9 ranks, you must use this table"
  (let* ((inf most-positive-fixnum)
         (table `((90   (5 15)    (6 17)    (7 22)   (8 26)
(9 29))
                  (95   (5 ,inf)  (6 21)    (7 24)   (8 30)
(9 35))
                  (98   (5 ,inf)  (6 ,inf)  (7 28)   (8 34)
(9 39))
                  (99   (5 ,inf)  (6 ,inf)  (7 ,inf) (8 36)
(9 43))
                  (99.9 (5 ,inf)  (6 ,inf)
(7 ,inf) (8 ,inf)(9 ,inf)))))
    (dotimes (i (list-length table))
      (if (= (first (nth i table)) signum)
          (dotimes (j 5)
            (if (= total.ranks (first (nth (+ j 1) (nth i table))))
                (return-from get-wcrit
                  (second (nth (+ j 1) (nth i table))))))))))
(defun wilcoxon-rank (l &optional (ranks (make-hash-table)) (n 0))
  "Returns a hash of the ranks in a sorted list. All numbers in a
   run of repeated entries get the average rank of that run."
  (if (null l)
      ranks
      (let (repeats sum now)
        (labels ((walk () (incf n) (pop l))
                 (new  () (setf repeats 1) (setf sum n))
                 (same () (incf sum n) (incf repeats))
                 (spin () (when (eql now (car l))
                            (walk) (same) (spin))))
          (setf now (walk))
          (new)
          (spin)
          (cond ((= now 0)
                 (setf repeats 1 sum 0 now 0 n 0)))
          (setf (gethash now ranks) (/ sum repeats))
          (wilcoxon-rank l ranks n)))))
```

Figure 13.13: CODE: `lib/tricks/wilcoxon2.lisp`

```
(defun fastmap (k datamatrix &optional
                (distancematrix (set-distance-matrix datamatrix))
                (x (make-array (list (array-dimension datamatrix 0)
                                     k) :initial-element nil))
                (pa (make-array (list 2 k) :initial-element nil))
                (cnum -1)) ;k is the desired number of dimensions,
  ;datamatrix if the data; all the rest are optional and generally
  ;computed by fastmap for subsequent passes
  "fastmap: does important things; generally awesome."
  (let ((templist 0) ;for storing the output of
                     ;choose-distance-objects
        (dprimematrix 0) ;for storing the reduced distancematrix
        (o.a 0) ;stores object a
        (o.b 0) ;stores object b
        (x.i 0)) ;stores the calculated value of x.i
    ;(a bit extraneous, but I think it helps with readability)

    ;;1 base case conditional
    (if (<= k 0)
        (return-from fastmap x))
                           ;returns the new coordinates of objects
    (setf cnum (+ cnum 1)) ;This is here because that's how
                   ;faloutsos and lin organized their algorithm
    ;;2 choose pivot objects
    (setf templist (choose-distance-objects distancematrix))
    (setf o.a (first templist))
    (setf o.b (second templist))
    ;;3 record the ids of the pivot objects
    (setf (aref pa 0 cnum) o.a)
    (setf (aref pa 1 cnum) o.b)
    ;;4 distance something
    (if (= (aref distancematrix o.a o.b) 0)
        (dotimes (i (array-dimension x 0))
          (setf (aref x i cnum) 0)))
    ;;5 project the objects on the line(o.a,o.b)
    (dotimes (i (array-dimension datamatrix 0))
      (setf x.i (/ (- (+ (expt (aref distancematrix o.a i) 2)
                         (expt (aref distancematrix o.a o.b) 2))
                      (expt (aref distancematrix o.b i) 2))
                   (* 2 (aref distancematrix o.a o.b))))
      (setf (aref x i cnum) x.i))
    ;;6 consider the projections of the objects on a hyper-plain
    ;perpendicular to the line (o.a,o.b)
    ; find new distance function
    (setf dprimematrix (reduce-dist distancematrix x cnum))
    (fastmap (- k 1) datamatrix dprimematrix x pa cnum)))
                                       ;end fastmap

(deftest test-fastmap()
  (check
    (let ((x (fastmap 1 (make-array '(4 2)
                                    :initial-contents '((0 0)
                                                        (5 3)
                                                        (10 24)
                                                        (60 70))))))
      ;;fastmap chooses o.a to be 3, and o.b to be 0
      (and (between 92 (aref x 0 0) 93)
           (between 86 (aref x 1 0) 87)
           (between 67 (aref x 2 0) 68)
           (between 0 (aref x 3 0) 0)))))
```

Figure 13.14: CODE: `lib/tools/pre/fastmap.lisp`

```
(defun k-means (k data)
  (k-means-prim k (make-array `(,(length data)
                                ,(length (first data)))
                              :initial-contents data)))
(defun k-means-prim (k data)
  "k-means: performs a k-means clustering."
  (let ((memberof (make-array (array-dimension data 0)
                              :initial-element nil))
        (oldmemberof (make-array (array-dimension data 0)
                                 :initial-element nil))
        (num.in.cluster (make-array k :initial-element 0))
        (centroids (make-array (list k (array-dimension data 1))
                               :initial-element 0))
        (distances (make-array (list (array-dimension data 0) k)
                               :initial-element nil))
        (tempmin nil)
        (finaldata
         (adjust-array data
                       (list (array-dimension data 0)
                             (+ (array-dimension data 1) 1)))))
    (dotimes (i (array-dimension data 0))
      (setf (aref memberof i) (mod i k))
      (setf (aref num.in.cluster (mod i k))
            (+ (aref num.in.cluster (mod i k)) 1)))
    (dotimes (i (array-dimension data 0))
      (dotimes (j (array-dimension data 1))
        (setf (aref centroids (aref memberof i) j)
              (+ (/ (aref data i j)
                    (aref num.in.cluster (aref memberof i)))
                 (aref centroids (aref memberof i) j)))))
    (dotimes (iteration (array-dimension data 0) memberof)
      (if (equalp oldmemberof memberof)
          (return memberof))
      (dotimes (i (array-dimension memberof 0))
        (setf (aref oldmemberof i) (aref memberof i)))
      (setf distances (set-distance-matrix2 data centroids))
      (dotimes (i (array-dimension distances 0))
        (setf tempmin (aref distances i 0))
        (dotimes (j (array-dimension distances 1))
          (cond ((<= (aref distances i j) tempmin)
                 (setf tempmin (aref distances i j))
                 (setf (aref memberof i) j)))))
      (setf num.in.cluster (make-array k :initial-element 0))
      (dotimes (i (array-dimension memberof 0))
        (setf (aref num.in.cluster (aref memberof i))
              (+ (aref num.in.cluster (aref memberof i)) 1)))
      (setf centroids (make-array (list k (array-dimension data 1))
                                  :initial-element 0))
      (dotimes (i (array-dimension data 0))
        (dotimes (j (array-dimension data 1))
          (setf (aref centroids (aref memberof i) j)
                (+ (/ (aref data i j)
                      (aref num.in.cluster (aref memberof i)))
                   (aref centroids (aref memberof i) j))))))
    (dotimes (i (array-dimension data 0))
      (setf (aref finaldata i (- (array-dimension finaldata 1) 1))
            (aref memberof i)))
    finaldata))                          ;end k-means
```

Figure 13.15: CODE: `lib/tools/pre/kmeans.lisp`

Part V

Tools (for Data Mining)

Chapter 14

Storage

A surprising amount of the ICCLE code is not related to data mining. Most of it, in fact, relates to general LISP utilities and functions to store and access training files.

This chapter describes the storage code system. The dependences of that system were described above in Figure 8.5. This code process data sets like those shown in Figure 14.1. We say that that Figure 14.1 contains *cells* of data stored in *rows* (also known as *instances*). Data column names are called either *columns* or *features*.

Note that the top-level driver of the data storage system is the `data` function. This is described below after an initial discussion about some support routines.

14.1 Building Tables

This `data` function returns an *unindexed* `table` data structure. For some applications this suffices but for applications that perform multiple queries, there are tools to cross- index the data (see `xindex`, below). For example, while `table` stores rows as lists, `xindex` stores them as arrays. This means that accessing a row is fundamentally faster in `xindex` than in `table`s.

14.1.1 Unknown

Sometimes, data is not known and some "don't know" symbol must be added to a data file. The file *lib/store/unknown.lisp* in Figure 14.2 defines the ICCLE "don't known character".

`*unknown*` Stores the value of "don't know" (currently, "?"). **unknown**

`(unknownp x)` (Macro) Returns "t" if `x` is the unknown value. *unknownp*

```
(defun contact-lens ()
  (data
   :name 'contact-lens
   :columns '(age            prescription astigmatism tear-production lens)
             ;-------------- ------------ ----------- --------------- ----
   :egs    '((young          myope        yes         normal          hard)
             (young          hypermetrope yes         normal          hard)
             (presbyopic     myope        yes         normal          hard)
             (pre-presbyopic myope        yes         normal          hard)
             ;-------------- ------------ ----------- --------------- ----
             (young          hypermetrope no          reduced         none)
             (young          hypermetrope yes         reduced         none)
             (pre-presbyopic hypermetrope yes         reduced         none)
             (pre-presbyopic hypermetrope yes         normal          none)
             (young          myope        no          reduced         none)
             (young          myope        yes         reduced         none)
             (presbyopic     myope        no          reduced         none)
             (presbyopic     myope        no          normal          none)
             (presbyopic     hypermetrope yes         reduced         none)
             (presbyopic     hypermetrope yes         normal          none)
             (presbyopic     myope        yes         reduced         none)
             (pre-presbyopic hypermetrope no          reduced         none)
             (pre-presbyopic myope        no          reduced         none)
             (pre-presbyopic myope        yes         reduced         none)
             (presbyopic     hypermetrope no          reduced         none)
             ;-------------- ------------ ----------- --------------- ----
             (pre-presbyopic myope        no          normal          soft)
             (pre-presbyopic hypermetrope no          normal          soft)
             (young          myope        no          normal          soft)
             (young          hypermetrope no          normal          soft)
             (presbyopic     hypermetrope no          normal          soft))))
```

Figure 14.1: Contact lens data.

14.1.2 Header

Data miners assume some *meta-knowledge* about the data they are processing; e.g.

- what is the target concept (called the `class` column),
- what is the name of each column,
- what is columns are numeric,
- what columns are non-numeric;
- which columns should be ignored by the learning;
- etc.

```
(defparameter *unknown* '?)
(defmacro unknownp (x) `(eq ,x *unknown*))
```

Figure 14.2: CODE: `lib/store/unknown.lisp`

```
(defstruct header name numericp classp ignorep)

(defun columns-new (cols klass)
  (let ((counted (make-counted)))
    (doitems (col i cols)
      (counted+ (make-header :name col
                             :numericp (numericp col)
                             :classp  (= i klass))
                counted))
    (setf (counted-all counted)
          (reverse (counted-all counted)))
    counted))

(defun columns-header (cols)
  (mapcar #'header-name (counted-all cols)))

(defun numericp (x)
  (equal (char (symbol-name x) 0) #\$))

(defun ignorep (x)
  (equal (char (symbol-name x) 0) #\?))
```

Figure 14.3: CODE: `lib/store/header.lisp`

This knowledge is stored in the *columns* line of Figure 14.1:

```
'(age prescription astigmatism tear-production lens)
```

By default, all columns are not numeric and the last column is the class (`lens`). Columns are declared numeric by adding the "`$`" symbol to the column name. Also, columns are declared numeric by adding the "`?`" symbol to the column name. So, this following header makes `age` a numeric and tell us to ignore `tear-production`.

```
'($age prescription astigmatism ?tear-production lens)
```

The file *lib/store/header.lisp* in Figure 14.3 defines the `header` struct.

`columns-new cols klass)` Convents the column header list to a list of column header struct. Each item of the `cols` list is a symbol containing zero of one symbols "`$`", "`?`" denoting numeric or ignorable columns. The numeric index of the class column is given by `klass`. *columns-new*

`(columns-header cols)` Converts the list of column header structs `cols` into a list of column header names. *columns-header*

`(header-name header)` Returns the name of the header *header-name*

`(header-numericp header)` Returns "`t`" if the header is for numeric values. *header-numericp*

`(header-classp header)` Returns "`t`" if the header is for a target class. *header-classp*

```
(defstruct (table (:print-function table-print))
  name
  columns
  class
  (cautions (make-caution))
  (all      (make-counted)))

(defun table-print (tbl s depth)
  (declare (ignore depth))
  (format s
  "#(TABLE~%~T:NAME ~a~%~T:COLUMNS ~a~%~T :CLASS ~a~%~T :CAUTIONS~a~%~T :ALL ~a~T)"
          (table-name     tbl)
          (table-columns  tbl)
          (table-class    tbl)
          (table-cautions tbl)
          (table-all      tbl)))

(defun table-copy (tbl &optional (new (table-egs tbl)))
  (data :egs     new
        :name    (table-name tbl)
        :klass   (table-class tbl)
        :columns (columns-header (table-columns tbl))))
```

Figure 14.4: CODE: `lib/store/table.lisp`

header-ignorep `(header-ignorep header)` Returns "`t`" if the header is marked as "ignore".

numericp `(numericp x)` Returns "t" if `x` contains the marker for numeric columns.

unknownp `(unknownp x)` Returns "t" if `x` contains the marker for unknown columns.

14.1.3 Table

The file *lib/store/table.lisp* in Figure 14.4 defines the `table` struct. Tables store:

- A unique name;
- A list of the column `header`s defined in §**??**;
- A list of rows stored inside `eg` (or "example") structs (defined in §**??**).
- A unique `name`; A `class` variable storing the index of the target concept;
- A list of any caution messages generated when the table was pre-processed (for more on cautions, see §**??**).
- A counted list of all the rows (for more on counted lists, see §**??**).

14.1.4 Table Selectors

Tables are used very frequently by ICCLE. the file *lib/store/tableselectors.lisp* in Figure 14.5 defines a set of useful accessors to table contents.

isa `(isa n tbl)` Return the target in the `n`-th row of the table `tbl`.

```
(defun isa (x tbl)  (nth (table-class tbl) x))

(defun table-feature-names (tbl)
  (let (features
        (classi (table-class tbl)))
    (doitems (feature i (counted-all (table-columns tbl))
                     (reverse features))
      (unless (= i classi)
        (push (header-name feature) features)))))

(defun table-width (tbl)
  (counted-n (table-columns tbl)))

(defun table-height (tbl)
  (counted-n (table-all tbl)))

(defun table-rows (tbl)
  (counted-all (table-all tbl)))

(defun table-egs (tbl)
  (mapcar #'eg-features (table-rows tbl)))

(defun table-majority-class (tbl &optional (egs (table-egs tbl)))
  (let ((counts (make-hash-table))
        (max most-negative-fixnum)
        majority)
    (dolist (eg egs majority)
      (let* ((class (isa eg tbl))
             (new   (incf (gethash class counts 0))))
        (if (> new max)
            (setf max new
                  majority class))))))
```

Figure 14.5: CODE: `lib/store/tableselectors.lisp`

table-all `(table-all tbl)` Returns the list of `eg` structs containing the data.

table-cautions `(table-cautions tbl)` Returns the list of any cautions seen while loading the data.

table-class `(table-class tbl)` Returns the index of the table column with the class variables.

table-columns `(table-columns tbl)` Returns a list of the `header` structs describing the table columns.

table-egs `(table-egs tbl)` Returns the data used to create the table.

table-feature-names `(table-feature-names tbl)` Returns a list of the column names, excluding the class name, in the table `tbl`.

table-height `(table-height tbl)` Returns the number of rows of data in a table (same as `table-rows`).

table-majority-class `(table-majority-class tbl)` Returns the class name that appears most frequently in the table.

table-name `(table-name tbl)` Returns the name of the table.

table-rows `(table-rows tbl)` Returns the number of rows of data in a table (same as `table-height`).

table-width `(table-width tbl)` Returns the number of columns in a table, including the class column.

14.2 Data

The file *lib/store/data.lisp* in Figure 14.6 defines `data`, a simple table creation. This function batches up all the row and column definition work into a single interface. For an example usage of `data`, see Figure **??**.

data `(data &key name columns egs change klass shufflep)` Converts a list of `egs` to a `table` struct called `name`. Checks that all the rows are the correct width (otherwise, creates new `caution` warnings). `Columns` is a list of column names containing perhaps the symbols "`$`" denoting numeric features and "`?`" denote features to ignore. `Klass` stores the index of the class feature. If negative, `klass` counts in from the right-hand-side+1; otherwise it is an offset from the left. The default value for `klass` is -1; i.e. the most right-hand-side column is the class. If `shufflep` is true, the rows are rearranged at random. The optional `change` parameter enables fast alternation of table contents (e.g. replacing all class symbols `yes,no` with `2,1`, respectively).

```
(defstruct eg features class)

(defun data (&key name  columns egs change (klass -1) shufflep)
  (let* ((tbl
          (make-table
           :name name
           :columns (columns-new
                     columns
                     (class-index klass (length columns)))))
         (egs1  (if shufflep (shuffle egs) egs)))
    (setf (table-class tbl)
          (class-index klass (table-width tbl)))
    (dolist (eg egs1 tbl)
      (data1 eg change klass tbl))))

(defun data1 (one change klass tbl)
  (when
      (ok (= (table-width tbl) (length one))
          (table-cautions tbl) "~a wrong size" one)
    (if change
        (setf one (sublis change one)))
    (counted+
     (make-eg :class (isa one tbl) :features one)
     (table-all tbl))))

(defun class-index (klass width)
  (if (< klass 0) (+ klass width) klass))

(defun make-data1 ()
  (data
   :name   'weather
   :columns '(forecast temp humidty $windy play)
   :egs     '((sunny    hot  high   FALSE no)
              (sunny    hot  high   TRUE  yes)
              (sunny    hot  high         yes)
              )
   :change '((yes . 2) (no . 1))))
```

Figure 14.6: CODE: `lib/store/data.lisp`

The following code offers an example of `data`. In this example, note that the third row has a missing feature, the fourth column has been marked as a numeric column, and some `changes` have been defined that should swap all `yes,no` with `2,1`, respectively.

```
(defun make-data1 ()
  (data
   :name   'weather
   :columns '(forecast temp humidity $windy play)
   :egs    '((sunny    hot  high    FALSE  no)
             (sunny    hot  high    TRUE   yes)
             (sunny    hot  high           yes)
             )
   :change '((yes . 2) (no . 1))))
```

When executed, this produces the following output:

```
> (make-data1)
#(TABLE
 :NAME WEATHER
 :COLUMNS
    #S(COUNTED
       :ALL (#S(HEADER :NAME FORECAST :NUMERICP NIL :CLASSP NIL :IGNOREP NIL)
             #S(HEADER :NAME TEMP     :NUMERICP NIL :CLASSP NIL :IGNOREP NIL)
             #S(HEADER :NAME HUMIDTY  :NUMERICP NIL :CLASSP NIL :IGNOREP NIL)
             #S(HEADER :NAME $WINDY   :NUMERICP T   :CLASSP NIL :IGNOREP NIL)
             #S(HEADER :NAME PLAY     :NUMERICP NIL :CLASSP T   :IGNOREP NIL))
        :N 5
        :SORTED NIL)
  :CLASS 4
  :ALL #S(COUNTED
          :ALL (#S(EG :FEATURES (SUNNY HOT HIGH TRUE 2) :CLASS 2)
                #S(EG :FEATURES (SUNNY HOT HIGH FALSE 1) :CLASS 1))
          :N 2
          :SORTED NIL)
  :CAUTIONS#(CAUTION :ALL ((SUNNY HOT HIGH YES) wrong size) :PATIENCE 19)
)
```

In `tables`, `columns` and `all` are counted structs. The slot `columns/all` contains the `header` structs: note that the last feature (`play`) has a `classp` of "`t`" while the others are all "`nil`"". Also, only `windy` has a `numericp` of "`t`'.

As to the actual data, this is stored in an `eg` struct that stores the cells as well as the `class` of each row. Note that the class symbol has been changed from `yes,no` to `2,1`, respectively.

On the last line of the above figure, we can see that the `cautions` slot is not empty. Recall that one row had the wrong length so a `caution` struct was generated. Observe how any row generating a caution does not get entered into `all`. Also, each caution decrements the `patience` counter of the `table/cautions`. In the above example, we have 19 cautions left.

14.3 Xindex

Tables are a relatively simple structure that store rows in a straight-forward manner (as lists). For query-intensive applications, or for very large tables, an

cross-indexing scheme is useful to pre-compute and cache some standard queries to a table.

The `cross-index` function input defined in the file *lib/store/crossindex.lisp* (see Figure 14.7) implements such a cross-indexing scheme. The function inputs a table and returns an `xindex` struct that stores the table, as well as some cached information about that table. `Xindex` is defined in *lib/store/xindex.lisp* (see Figure 14.8).

For example, consider the following function.

```
(defun make-data2 ()
 (data
  :name    'weather
  :columns '($forecast $play)
  :egs     '((sunny   yes)
             (rainy   yes)
             (sunny   yes))))
```

The following call returns a `xindex` struct. The original table is stored in the `table` slot while the other slots contain some pre-computed and cached information:

- The table has 2 rows.
- All the rows have the same class.
- The ranges of the zero-column is 2 (`rainy,sunny`).
- The range of the first column is just `yes`.
- The values `sunny` and `rainy` appear twice and once (respectively) in column zero.
- In column one, the value `yes` appears once.
- There are three instances of class yes and no instances of any other class.

```
> (cross-index (make-index2))
#(XINDEX
 :TABLE <table>
  :ALL #(#S(EG :FEATURES (SUNNY YES) :CLASS YES)
         #S(EG :FEATURES (RAINY YES) :CLASS YES)
         #S(EG :FEATURES (SUNNY YES) :CLASS YES))
  :N 3
  :RANGES #(#S(COUNTED :ALL (RAINY SUNNY) :N 2 :SORTED NIL)
            #S(COUNTED :ALL (YES) :N 1 :SORTED NIL))
  :UNIQUES
     (0 RAINY) = 1
     (0 SUNNY) = 2
     (1 YES) = 3
   :CLASS-COUNTS
     YES = 3
   :COUNTS
     (YES 0 RAINY) = #S(COUNTED :ALL (1) :N 1 :SORTED NIL)
     (YES 0 SUNNY) = #S(COUNTED :ALL (2 0) :N 2 :SORTED NIL)
     (YES 1 YES) = #S(COUNTED :ALL (2 1 0) :N 3 :SORTED NIL))
```

14.3.1 Simple Access to the Cross-Index

The `xindex` struct is a little difficult to work with. For optimization reasons, the internal structures take many forms and programmers struggle with arrays

```
(defun cross-index (tbl)
  (let* ((x       (xindex-new tbl))
         (uniques (xindex-uniques x))
         (class-counts       (xindex-class-counts x)))
    (dotimes (i (table-width tbl))
      (setf (aref (xindex-ranges x) i) (make-counted)))
    (dotimes (j (xindex-ns x) x)
      (cross-index1 j (aref (xindex-all x) j)
x uniques class-counts))))

(defun cross-index1 (j eg  x uniques class-counts)
  (let* ((i -1)
         (class (eg-class eg)))
    (incf (gethash class class-counts 0))
    (dolist (range (eg-features eg))
      (incf  i)
      (unless (unknownp range)
        (let* ((xkey  `(,class ,i ,range))
               (ukey  `(,i ,range))
               (counted  (gethash xkey (xindex-counts x)
                                  (make-counted))))
          (setf (gethash xkey (xindex-counts x))
                (counted+  j counted))
          (when (= 1 (incf (gethash ukey uniques 0)))
            (setf (aref (xindex-ranges x) i)
                  (counted+ range
                            (aref (xindex-ranges x) i)))))))))

(deftest test-index ()
  (let ((tmp (cross-index (make-data2))))
    (check
      (samep tmp "
#(XINDEX
 :TABLE <table>
  :ALL #(#S(EG :FEATURES (SUNNY YES) :CLASS YES)
         #S(EG :FEATURES (RAINY YES) :CLASS YES)
         #S(EG :FEATURES (SUNNY YES) :CLASS YES))
  :N 3
  :RANGES #(#S(COUNTED :ALL (RAINY SUNNY) :N 2 :SORTED NIL)
            #S(COUNTED :ALL (YES) :N 1 :SORTED NIL))
  :UNIQUES
     (0 RAINY) = 1
     (0 SUNNY) = 2
     (1 YES) = 3
   :CLASS-COUNTS
     YES = 3
   :COUNTS
     (YES 0 RAINY) = #S(COUNTED :ALL (1) :N 1 :SORTED NIL)
     (YES 0 SUNNY) = #S(COUNTED :ALL (2 0) :N 2 :SORTED NIL)
     (YES 1 YES) = #S(COUNTED :ALL (2 1 0) :N 3 :SORTED NIL)
 )"))))

(defun make-data2 ()
  (data
   :name    'weather
   :columns '($forecast $play)
   :egs     '((sunny   yes)
              (rainy   yes)
              (sunny   yes))))
```

Figure 14.7: CODE: `lib/store/crossindex.lisp`

```
(defstruct (xindex (:print-function xindex-print))
  table
  all
  (n 0)
  ranges
  (counts (make-hash-table :test #'equalp))
  (class-counts (make-hash-table))
  (uniques (make-hash-table :test #'equal)))

(defun xindex-print (x s depth)
  (declare (ignore depth))
  (labels ((show (x y) (showh x :before y :stream s
                               :indent 5 :after "")))
    (format s
            "#(XINDEX~%~T:TABLE <table>~%~T :ALL ~a~%~T :N ~a~%~T :RANGES ~a~%~T"
            (xindex-all x)
            (xindex-n x)
            (xindex-ranges x))
    (show (xindex-uniques      x)  " :UNIQUES
") (show (xindex-class-counts x) "   :CLASS-COUNTS
") (show (xindex-counts       x) "   :COUNTS
"  ))
  (format s "~T)"))

(defun xindex-new (tbl)
  (make-xindex
   :table    tbl
   :all      (make-array
              (table-height tbl)
              :initial-contents (table-rows tbl))
   :n        (table-height tbl)
   :ranges   (make-array
              (table-width tbl))))
```

Figure 14.8: CODE: `lib/store/xindex.lisp`

```
(defun f (x class &optional
          (index      (table-class (xindex-table x)))
          (range      class)
          (access     #'counted-n)
          (if-missing 0))
  (let ((tmp (gethash `(,class ,index ,range)
                      (xindex-counts x))))
    (if tmp
        (funcall access tmp)
        if-missing)))

(defun uses (x class index range)
  (f x class index range #'counted-all nil))

(deftest test-xindex-f ()
  (unless (fboundp 'weather2) (loaddata 'weather2))
  (let ((x (cross-index (weather2))))
    (check
      (= (f x 'no) 5)
      (= (f x 'no 0 'rainy) 2))))
```

Figure 14.9: CODE: `lib/store/f.lisp`

versus lists versus hash-tables versus counted lists. Therefore, the "`f`" function found in *lib/store/f.lisp* (see Figure 14.9) was designed as a simple interface into cross-indices.

f `(f x class column range access if-missing)` Returns the frequency counts of {`class`} or {`class column`} or {class column range}. If the value is not found then `if-missing` is returned.

The function can be called in one of two standard ways:

`(f x class)` Returns the frequency of `class` in the table "`x`".

`(f x class column value)` Returns the frequency of `class,column,value` in the table "`x`".

14.3.2 Elaborate Access to the Cross-Index

The "`f`" function is not enough for all purposes, The file *lib/store/xindexselectors.lisp* shown at Figure 14.10 defines a more elaborate API into an `xindex` struct.

Some of these selectors just return simple slot values from an `xindex` struct:

xindex-all `(xindex-all x)` Returns an array of the row `eg` slots. Note that these examples are stored as lists inside a `table` struct, so accessing a row is faster in `xindex` than in `table`s.

xindex-counts `(xindex-counts x)` Returns a hash table whose keys are triples {`class column value`} and whose keys are a counted list of rows containing that value in that row for that class.

```
(defun xindex-classi (x)
  (table-class (xindex-table x)))

(defun xindex-classes-n (x)
  (counted-n
   (aref (xindex-ranges x)
         (table-class
          (xindex-table x)))))

(defun xindex-features (x)
  (xindex-ranges x))

(defun xindex-feature (i x)
  (counted-all (aref (xindex-ranges x) i)))

(defun xindex-feature-names (x)
  (table-feature-names (xindex-table x)))

(defun xindex-ns (x)
  (counted-n (table-all (xindex-table x))))

(defun xindex-classes-all (x)
  (counted-all
   (aref (xindex-ranges x)
         (table-class
          (xindex-table x)))))

(defun xindex-unique-n (x feature range)
  (gethash `(,feature ,range) (xindex-uniques x) 0))

(defun xindex-width (x)
  (table-width (xindex-table x)))

(defun xindex-majority-class (x)
  (let* ((classes       (xindex-classes-all x))
         (max           -1)
         most-frequent)
    (dolist (class classes  most-frequent)
      (let* ((count  (f x class)))
        (if (> count max)
            (setf max           count
                  most-frequent class))))))

(deftest test-xindex-majority-class ()
  (or (fboundp 'weather) (loaddata 'weather))
  (check
    (= 2 (xindex-majority-class (cross-index (weather))))))
```

Figure 14.10: CODE: `lib/store/xindexselectors.lisp`

xindex-ranges `(xindex-ranges x)` Returns a list of counted ranges for each feature in the `xindex` struct `x`. If the table holds N features, this this list is N items long.

xindex-table `(xindex-table x)` Returns the table inside the index `x`.

xindex-uniques `(xindex-uniques x)` Returns a hash table whose keys are the pairs {`column value`} and whose values are the number of times those pairs appear in all classes.

Other selectors do little more than pass the query down to the `table` struct stored inside the `xindex`.

xindex-classi `(xindex-classi x)` Returns the index of the class column inside the index `x`'s table.

xindex-feature-names `(xindex-feature-names x)` Returns a list of the column names, excluding the class name, inside the index `x`'s table.

xindex-width `(xindex-width x)` Returns the number of columns inside the index `x`'s table.

Yet other selectors do more work:

xindex-classes-all `(xindex-classes-all x)` Returns a list of class values.

xindex-classes-n `(xindex-classes-n x)` Returns the number of different class values known to the index.

xindex-feature `(xindex-feature i x)` Return the i-th feature from the list of features.

xindex-features `(xindex-features x)` Returns an array of counted symbols per feature.

xindex-majority-class `(xindex-majority-class n)` Return the class with highest frequency.

xindex-ns `(xindex-ns x)` Returns the number of examples in the table.

xindex-unique-n `(xindex-unique-n x feature range)` Return the number of times {`feature range`} appears in the index (default, 0).

Chapter 15

Discretization

Think of learning like an accordion- some target concept is spread out across all the data and our task is to squeeze it together till it is dense enough to be visible. That is, learning is like a compression algorithm.

One trick that helps compressions is *discretization*: i.e. clumping together observations taken over a continuous range into a small number of regions. Humans often discretize real world data. For example, parents often share tips for "toddlers"; i.e. humans found between the breaks of $age = 1$ and $age = 3$. Many researchers report that discretization improves the performance of a learner since it gives a learner a smaller space to reason about, with more examples in each part of the space [DKS95, LYW06, YW03, FI93].

Discretization can generally be described as a process of assigning data attribute instances to bins or buckets that they fit in according to their value or some other score. The general concept for discretization as a binning process is dividing up each instance of an attribute to be discretized into a number distinct buckets or bins. The number of bins is most often a user-defined, arbitrary value; however, some methods use more advanced techniques to determine an ideal number of bins to use for the values while others use the user-defined value as a starting point and expand or contract the number of bins that are actually used based upon the number of data instances being placed in the bins. Each bin or bucket is assigned a range of the attribute values to contain, and discretization occurs when the values that fall within a particular bucket or bin are replaced by identifier for the bucket into which they fall.

After Gama and Pinto [GP06], we say that discretization is the process of converting a continuous range into a histogram with k break points $b_1 \dots b_k$ where

$$\forall i < j : b_i \leq b_j$$

. The histogram divides a continuous range into bins (one for each break) and many observations from the range may fall between two break points b_i and b_{i+1} at frequency counts c_i.

Simple discretizers are *unsupervised* methods that build their histograms without exploiting information about the target class; e.g.

- *equal width*: $(\forall i, j : (b_i - b_{i-1}) = (b_j - b_{j-1}))$;
- *equal frequency*: $(\forall i, j : (c_i = c_j))$. For Näive Bayes classifiers working on n instances, Yang & Webb [YW03] advocate equal frequency with $c_i = c_j = \sqrt{n}$.

Part VI

Tasks (for Supporting the Business)

Part VII

Tests (of Different Tools)

Part VIII

Traps (for the unwary)

15.1 Subversion

15.1.1 The File Character Trap

Don't use file names that are illegal in other operating systems (e.g. containing ":"). Try not use spaces, stay with 0123456789, A-Z, a-z, and ";"

15.1.2 The File Browser Trap

Don't move files without telling subversion about it

Part IX

To do (What's Next?)

Part X

Team (About Us)

Chapter 16

Teams

16.1 Team 0: Cradle

This team built some initial infrastructure and defined some of the early coding standards for ICCLE.

16.1.1 Tim Menzies

Tim Menzies, Ph.D. (`tim@menzies.us`) has worked on advanced modeling and AI since 1986. He received a PhD from the University of New South Wales, Sydney, Australia and is the author of 160+ refereed papers.

A former research chair for NASA, Dr. Menzies is now a associate professor at the West Virginia University's Lane Department of Computer Science and Electrical Engineering.

For this book, he build the initial framework and sample code, then herded the LISP gang while they wrote the rest.

For more information, visit `http://menzies.us`.

16.1.2 Peter Santiago

Peter Santiago is an undergraduate in Computer Science at West Virginia University.

A dynamic figure, often seen scaling walls and crushing ice, he has been known to remodel train stations on my lunch breaks, making them more efficient in the area of heat retention.

He also translates ethnic slurs for Cuban refugees, writes award-winning operas, and manage time efficiently. Occasionally, he treads water for three days in a row.

For his sins, he was the first user of Tim Menzies' ICCLE code.

Part XI

Theory (Only if you Care)

Bibliography

[AFS96] J. Angele, D. Fensel, and R. Studer. Domain and task modelling in mike. In A. Sutcliffe et.al., editor, *Domain Knowledge for Interactive System Design*. Chapman & Hall, 1996.

[BATJ91] T. Bylander, D. Allemang, M.C. Tanner, and J.R. Josephson. The Computational Complexity of Abduction. *Artificial Intelligence*, 49:25–60, 1991.

[BdVe94] J. Breuker and W. Van de Velde (eds). *The CommonKADS Library for Expertise Modelling*. IOS Press, Netherlands, 1994.

[Bec00] Kent Beck. *Test-Driven Development*. Addison-Wesley Professional, 200.

[Ben95] R. Benjamins. Problem-solving methods for diagnosis and their role in knowledge acquisition. *International Journal of Expert Systems: Research & Applications*, 8(2):93–120, 1995.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.

[BM98] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. URL: `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[Bre01] L. Breimann. Random forests. *Machine Learning*, pages 5–32, October 2001.

[Bro86] C.H.P. Brookes. Requirements Elicitation for Knowledge Based Decision Support Systems. Technical Report 11, Information Systems, University of New South Wales, 1986.

[Cen87] J. Cendrowska. Prism: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987.

[CJS92] B. Chandrasekaran, T.R. Johnson, and J. W. Smith. Task structure analysis for knowledge modeling. *Communications of the ACM*, 35(9):124–137, 1992.

[CKPM93] P. Compton, B. Kang, P. Preston, and M. Mulholland. Knowledge acquisition without analysis. In *European Knowledge Acquisition Workshop*, 1993.

[Cla92] W.J. Clancey. Model Construction Operators. *Artificial Intelligence*, 53:1–115, 1992.

[Coh95] W.W. Cohen. Fast effective rule induction. In *ICML'95*, pages 115–123, 1995. Available on-line from `http://www.cs.cmu.edu/~wcohen/postscript/ml-95-ripper.ps`.

[Dem06] J. Demsar. Statistical comparisons of clasifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006. Avaliable from `http://jmlr.csail.mit.edu/papers/v7/demsar06a.html`.

[DHS00] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, 2000.

[DKS95] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995. Available from `http://www.cs.pdx.edu/~timm/dm/dougherty95supervised.pdf`.

[DP97] Pedro Domingos and Michael J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[EST+95] H. Eriksson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen. Task modeling with reusable problem-solving methods. *Artificial Intelligence*, 79(2):293–326, 1995.

[FF05] Johannes Fürnkranz and Peter A. Flach. Roc 'n' rule learning: towards a better understanding of covering algorithms. *Machine Learning*, 58(1):39–77, 2005.

[FI93] U M Fayyad and I H Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[Gai89] B.R. Gaines. An ounce of knowledge is worth a ton of data: quantitative studies of the trade-off between expertise and data based on statistically well-founded empirical induction. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 156–159, 1989. Available on-line at `http://ksi.cpsc.ucalgary.ca/articles/Induct/ML89/`.

[GM96] Y. Gil and E. Melz. Explicit representations of problem-soving strategies to support knowledge acquisition. In *Proceedings AAAI' 96*, 1996.

[GP06] Joao Gama and Carlos Pinto. Discretization from data streams: applications to histograms and data mining. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 662–667, New York, NY, USA, 2006. ACM Press. Available from `http://www.liacc.up.pt/~jgama/IWKDDS/Papers/p6.pdf`.

[GRC+98] Karen M. Gardner, Alexander R. Rush, Michael Crist, Rober Konitzer, James J. Odell, Bobbin Teegarden, and Robert Konitzer. *Cognitive Patterns: Problem-Solving Frameworks for Object Technology*. Cambridge University Press, June 1998.

[LBMP09] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *accepted for publication IEEE Transactions on Software Engineering*, 2009.

[LYW06] J. Lu, Y. Yang, and G.I. Webb. Incremental discretization for naive-bayes classifier. In *Lecture Notes in Computer Science 4093: Proceedings of the Second International Conference on Advanced Data Mining and Applications (ADMA 2006)*, pages 223–238, 2006. Available from `http://www.csse.monash.edu/~webb/Files/LuYangWebb06.pdf`.

[MDK+92] D. Marques, G. Dallemagne, G. Kliner, J. McDermott, and D. Tung. Easy Programming: Empowering People to Build Their own Applications. *IEEE Expert*, pages 16–29, June 1992.

[MDR07] T. Menzies, D.Owen, and J. Richardson. The strangest thing about software. *IEEE Computer*, 2007. `http://menzies.us/pdf/07strange.pdf`.

[Men97] T.J. Menzies. OO patterns: Lessons from expert systems. *Software Practice and Experience*, 27(12):1457–1478, December 1997. Available from `http://menzies.us/pdf/97patern.pdf`.

[MH03] T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from `http://menzies.us/pdf/03tar2.pdf`.

[MH07] T. Menzies and Y. Hu. Just enough learning (of association rules): The TAR2 treatment learner. In *Artificial Intelligence Review*, 2007. Available from `http://menzies.us/pdf/07tar2.pdf`.

[MM97] T.J. Menzies and A. Mahidadia. Ripple-down rationality: A framework for maintaining psms. In *Workshop on Problem-Solving Methods for Knowledge-based Systems, IJCAI '97, August 23.*, 1997. Available from `http://menzies.us/pdf/97rdra.pdf`.

[MZ96] E. Motta and Z. Zdrahal. Parametric design problem solving. In *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop*, 1996.

[New82] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.

[Qui92a] J. R. Quinlan. Learning with Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992. Available from `http://citeseer.nj.nec.com/quinlan92learning.html`.

[Qui92b] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.

[RD06] M. Richardson and Pedro Domingos. Markov logic networks, February 206.

[RLN93] P.S. Rosenbloom, J.E. Laird, and A. Newell. *The SOAR Papers*. The MIT Press, 1993.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: Modern Approach*. Prentice Hall, 1995.

[RTA93] R.Agrawal, T.Imeilinski, and A.Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD Conference, Washington DC, USA*, 1993. Available from `http://citeseer.nj.nec.com/agrawal93mining.html`.

[SC00] G. Schohn and D. Cohn. Less is more: Active learning with support vector machines. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 839–846, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[Sch99] Guus Schreiber, editor. *Knowledge Engineering and Management : The CommonKADS Methodology*. MIT Press, 1999.

[SG96] B. Swartout and Y. Gill. Flexible knowledge acquisition through explicit representation of knowledge roles. In *1996 AAAI Spring Symposium on Acquisition, Learning, and Demonstration: Automating Tasks for Users*, 1996.

[Ste90] L. Steels. Components of Expertise. *AI Magazine*, 11:29–49, 2 1990.

[SWA+94] A. TH. Schreiber, B. Wielinga, J. M. Akkermans, W. Van De Velde, and R. de Hoog. Commonkads. a comprehensive methodology for kbs development. *IEEE Expert*, 9(6):28–37, 1994.

[TH93] D.S.W. Tansley and C.C. Hayball. *Knowledge-Based Systems Analysis and Design*. Prentice-Hall, 1993.

[vHA96] F. van Harmelen and M. Aben. Structure-preserving specification languages for knowledge-based systems. *International Journal of Human-Computer Studies*, 44:187–212, 1996.

[WF05] Ian H. Witten and Eibe Frank. *Data mining. 2nd edition.* Morgan Kaufmann, Los Altos, US, 2005.

[WSB92] B.J. Wielinga, A.T. Schreiber, and J.A. Breuker. KADS: a Modeling Approach to Knowledge Engineering. *Knowledge Acquisition*, 4:1–162, 1 1992.

[YW03] Y. Yang and G. Webb. Weighted proportional k-interval discretization for naive-bayes classifiers. In *Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003)*, 2003. Available from `http://www.csse.monash.edu/~webb/Files/YangWebb03.pdf`.

Index

code
- *unknown*, 115
- abcd-stats, 100
- accuracy, 100
- align, 74
- all, 100
- allbut, 76
- array2list, 69
- as-dist, 93
- balance, 100
- between, 74
- caution-all, 82
- caution-killed, 82
- caution-patience, 82
- columns-header, 117
- columns-new, 117
- counted+, 81
- counted-all, 81
- counted-n, 81
- counted-sort, 81
- counted-sorted, 81
- data, 120
- deftest, 65
- dist-add, 93
- dist-adds, 93
- dist-all, 93
- dist-compare, 93
- dist-max, 93
- dist-mean, 93
- dist-min, 93
- dist-n, 93
- dist-sd, 93
- dist-sort, 93
- distogram, 93
- doarray, 69
- dohash, 69
- doitems, 76
- e, 74
- f, 126
- f-measure, 100
- fastmap, 107
- geta, 76
- gnuplot, 95
- harmonic-mean, 74
- header-classp, 117
- header-ignorep, 117
- header-name, 117
- header-numericp, 117
- interpolates, 82
- isa, 118
- k-means, 107
- line-b, 84
- line-m, 84
- line-verticalp, 84
- line-y, 84
- line1, 72
- list2array, 76
- list2stdev, 74
- list2string, 72
- lists2m, 78
- longest-string, 72
- lt, 72
- make, 58
- make-caution, 81, 82
- make-dist, 93
- make-node, 81
- mann-whitney, 105
- mean, 74
- median, 74
- my-command-line, 89
- my-getenv, 89
- my-random, 79
- my-random-int, 79
- nchars, 72

node-key, 81
node-kids, 81
numericp, 118
o, 87
ok, 82
pd, 100
pf, 100
point-to-line, 84
precision, 100
rank, 76
recall, 100
remove-nths, 76
reset-seed, 79
samep, 72
sh, 89
showh, 69
shuffle, 78
stdev, 76
sum, 76
table-all, 118
table-cautions, 120
table-class, 120
table-columns, 120
table-egs, 120
table-feature-names, 120
table-height, 120
table-majority-class, 120
table-name, 120
table-rows, 120
table-width, 120
tally, 78
tcritical, 105
test-all, 65
test-tools, 66
test-tricks, 66
tests, 65
tests-report, 65
tests-reset, 65
tmp-dir, 89
tmp-file, 89
transpose, 78
ttest, 105
ttest-from-lists, 105
unknownp, 115, 118
watch, 87
weighted-sum, 76
whiteout, 74
whitespacep, 74
wilcoxon, 107
xindex-all, 126
xindex-classes-all, 128
xindex-classes-n, 128
xindex-classi, 128
xindex-counts, 126
xindex-feature, 128
xindex-feature-names, 128
xindex-features, 128
xindex-majority-class, 128
xindex-ns, 128
xindex-ranges, 126
xindex-table, 128
xindex-unique-n, 128
xindex-uniques, 128
xindex-width, 128

files
lib/iccle.asd, 58
lib/iccle.lisp, 58
lib/store.asd, 61
lib/store/crossindex.lisp, 124
lib/store/data.lisp, 121
lib/store/f.lisp, 126
lib/store/header.lisp, 117
lib/store/table.lisp, 118
lib/store/tableselectors.lisp, 119
lib/store/unknown.lisp, 116
lib/store/xindex.lisp, 125
lib/store/xindexselectors.lisp, 127
lib/tests.asd, 59
lib/tests/deftest.lisp, 67
lib/tests/tests.lisp, 68
lib/tests/tools.lisp, 68
lib/tests/tricks.lisp, 68
lib/tools/pre/fastmap.lisp, 110
lib/tools/pre/kmeans.lisp, 111
lib/toolslearn.asd, 62
lib/toolspre.asd, 63
lib/tricks.asd, 60
lib/tricks/abcd.lisp, 99
lib/tricks/abcdmatrix.lisp, 101
lib/tricks/abcdtest.lisp, 102
lib/tricks/align.lisp, 73

lib/tricks/array.lisp, 70
lib/tricks/caution.lisp, 83
lib/tricks/counted.lisp, 82
lib/tricks/debug.lisp, 88
lib/tricks/dist.lisp, 92
lib/tricks/distogram.lisp, 94
lib/tricks/gnuplot.lisp, 97
lib/tricks/hash.lisp, 71
lib/tricks/line.lisp, 85
lib/tricks/list.lisp, 77
lib/tricks/mannwhitney.lisp, 106
lib/tricks/maths.lisp, 75
lib/tricks/matrix.lisp, 78
lib/tricks/os.lisp, 90
lib/tricks/random.lisp, 80
lib/tricks/strings.lisp, 72
lib/tricks/tree.lisp, 81
lib/tricks/ttest.lisp, 104
lib/tricks/wilcoxon.lisp, 108
lib/tricks/wilcoxon2.lisp, 109

teams
0 : cradle
Menzies, Tim, 143
Santiago, Peter, 143