The **TreeQ** Manual V0.8

Jonathan Foote

September 24, 2003

Chapter 1

Overview

The TreeQ package is a set of C-language applications and libraries that implement a automatic machine learning algorithm. Simply put, the package lets you construct a system that will *learn the differences* between complicated data sets. Though the system is data-driven and thus will work on any arbitrary data, I have experimented primarily with audio.

The basic technology lets you find the similarity, expressed as a number, between two given audio files. Applications include:

- Speaker identification: Find the speaker whose speech most resembles speech from an unknown talker.
- Speech/music classification: determine whether a given audio file is speech, music, or neither. Other classification/identification tasks like language, should work in a similar manner.
- Music/audio retrieval by similarity: Given an audio example, produce a ranked list (like Google) of similar-sounding audio from a large database.
- Audio segmentation: given a large audio file, locate speech, music, and silence, and detect which speaker is speaking.

1.1 Applications of the TreeQ Package

The TreeQ package is a set of library routines and applications (in vanilla C) that can be used to do the following:

- Given suitably labeled training data, construct “signatures” or “templates” from audio files that characterize the audio. This is a three-stage process:

  1. Calculate spectral (or other) parameters for the audio. This is currently done using the CSLU CLSUC routines, but the HTK package can be used as well.
  2. Construct a quantization tree from labeled, parameterized data. Given training data from different audio classes, this step learns the salient differences between the classes, and learns to ignore other differences.
  3. Use the tree to construct templates from parameterized source audio. This step is extremely efficient once the tree has been constructed.

- Given a set of templates, calculate the “distance” between them. Applications of this distance measure include:

  – Speaker identification and audio classification Given unknown speech data, the “closest” template from a set of known speaker templates will identify the unknown speaker. Similarly for general audio classification: train a tree and templates for the types of audio you wish to distinguish between (e.g. male/female speech, musical genres). The closest template to that from an unknown audio source will identify that unknown audio.
  – Audio retrieval-by-similarity. Given a set of audio files, construct templates for each. The set can then be ranked in order of similarity to any template.
  – Audio segmentation. Given an audio stream, calculate a template for a moving window, and compare to reference templates for, e.g., silence, music, speech, or any desired events. This can be used to segment the audio stream. Also, the difference between short- and long-term statistics can be used as a measure of audio novelty.

1.2 Block Diagram

Figure 1.1 shows a block diagram of the various TreeQ functional units. On the left is a conceptual schematic; on the right are the corresponding names

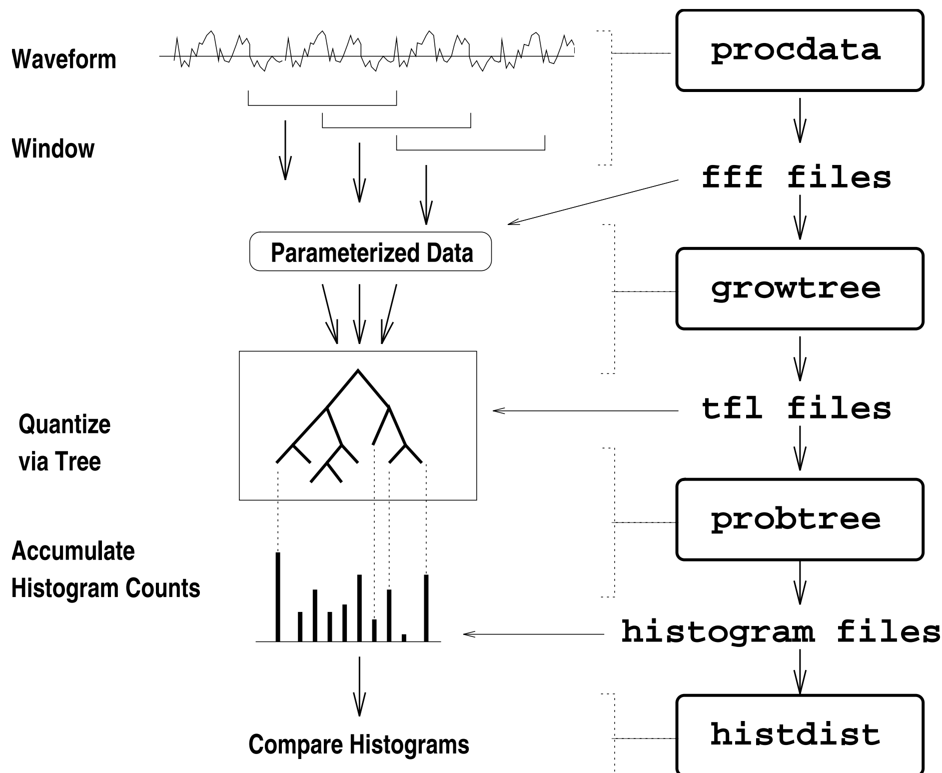

Figure 1.1: Block diagram of TreeQ functions.

of the TreeQ programs and file types. The basic function of most TreeQ modules is to convert complex and bulky data, here typified by audio files, into compact representations called histogram files. The first thing to be done is to parameterize the audio; this is done using the procdata program, which produces parameterized data files in the fff format. Once data has been parameterized, a tree-structured vector quantizer is grown using the growtree program, which produces a tree in a tfl file. Given a tree and (more) parameterized data, the program probtree generates histogram files, which can be compared and classified using the histdist program. Note that this diagram is oversimplified; most programs take several kinds of files, and data from different classes. Read on for more in-depth descriptions.

1.2.1 procdata Overview

This section describes the data pre-processing tool procdata. The basic idea is to reduce the large amount of data in an audio waveform by extracting the salient features using spectral analysis. A detailed explanation is beyond the scope of this manual; however there are some pretty good references available; I like the one in the HTK Book, but Rabiner's book is probably the best reference. Most spectral analysis techniques assume the signal is stationary (unchanging) over the analysis period. For speech, that's a pretty poor assumption. So what people do is to look at overlapping short segments of speech, around 10-25 mS long, called frames or windows. Each frame is converted to a spectral representation, typically by using a Fast Fourier Transform (FFT) or Linear Predictive Analysis (LPC). Most of my work has been done using Mel-scaled Cepstral Coefficients, or MFCCs, which are described in a little more detail in the next section. Other common analyses are Perceptual Linear Prediction (PLP) and filterbank (FFT) methods which are available using procdata.

Figure cepanalhows the steps in cepstral parameterization. First, the audio is Hamming-windowed in overlapping steps. For each window, the log of the power spectrum is computed using a discrete Fourier transform (DFT). The log spectral coefficients are perceptually weighted by a non-linear map of the frequency scale. This operation, called Mel-scaling, emphasizes mid-frequency bands in proportion to their perceptual importance. The final stage is to further transform the Mel-weighted spectrum (using another DFT) into "cepstral" coefficients. This results in features that are reasonably dimensionally uncorrelated, thus the final DFT is a good approximation of the Karhunen-Loeve transformation for the mel spectra. The audio waveform, sampled at several kHz, is thus transformed into a low-dimensional feature vector (12 coefficients typical) at a frame rate of a few hundred per seconds (100 is typical). Thus procdata reduces the amount of data by several orders of magnitude.

1.2.2 growtree Overview

This section details how to use the tree growing utilities growtree, choptree, and probtree. growtree, as you might expect from the name, grows a qtree by examining the characteristics of input files. Each input file is considered to be a "class." All data in a class is assumed to have similar characteristics, but data from different classes is assumed to be different. This is important:

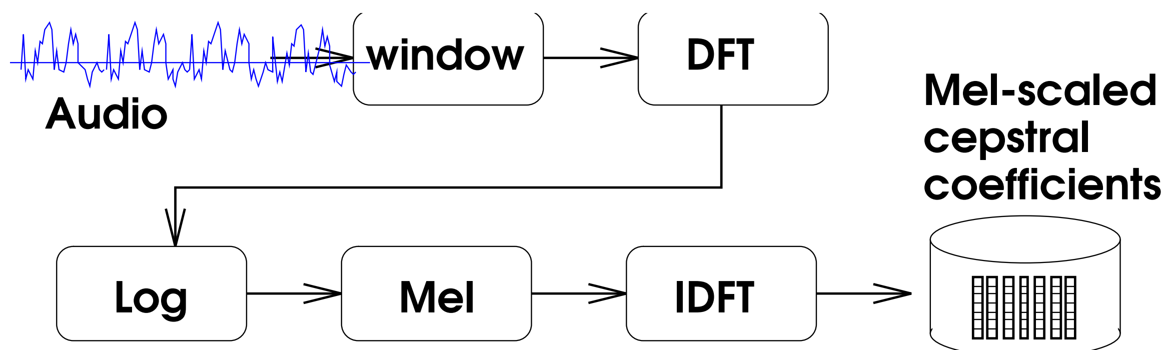

Figure 1.2: cepanal.eps

growtree spends considerable effort partitioning the feature space such that data from each class wind up in different partitions. Thus the class training data is important: it must contain sufficient examples of the things you are trying to differentiate between. Because classes will typically overlap a great deal, this is difficult to do perfectly. Nonetheless, growtree tries its best. By giving growtree appropriate kinds of data, you can teach it to differentiate between different things. For example, if you want to do talker identification, you should make each class wide examples of a particular talker's speech. Growtree will attempt to learn the differences between speakers, while ignoring the (possibly large) variations in the speech of a particular speaker. On the other hand, if you want to distinguish between phones (parts of speech), for example vowels and consonants, you should make the training classes examples of the same phones from many speakers. Thus the tree will try to learn the differences between the phones while ignoring the (possibly large) variations between speakers, for example between males and females.

A tree-structured quantizer is especially practical because it may be pruned to different sizes depending on the amount of data. Because there is one histogram bin for each leaf in the tree, the tree size directly determines the size of the histogram template. If data is sparse, a large histogram will be suboptimal as many bin counts will be zero. Pruning the tree will result in fewer bins, which may be able to better characterize the data. In this fashion the number of free parameters can be adjusted to suit the application.

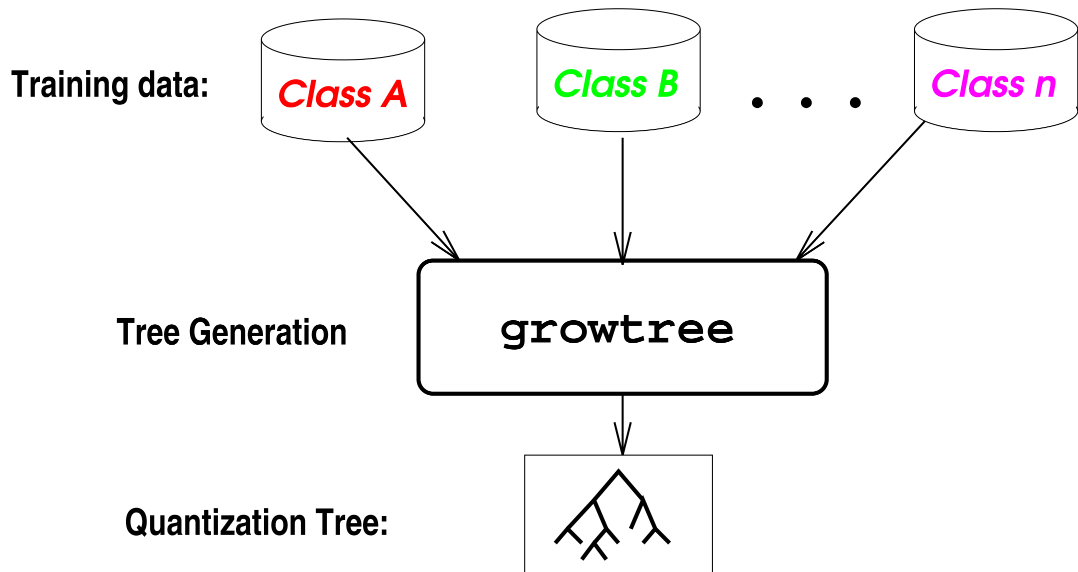

Figure 1.3: growtree.eps

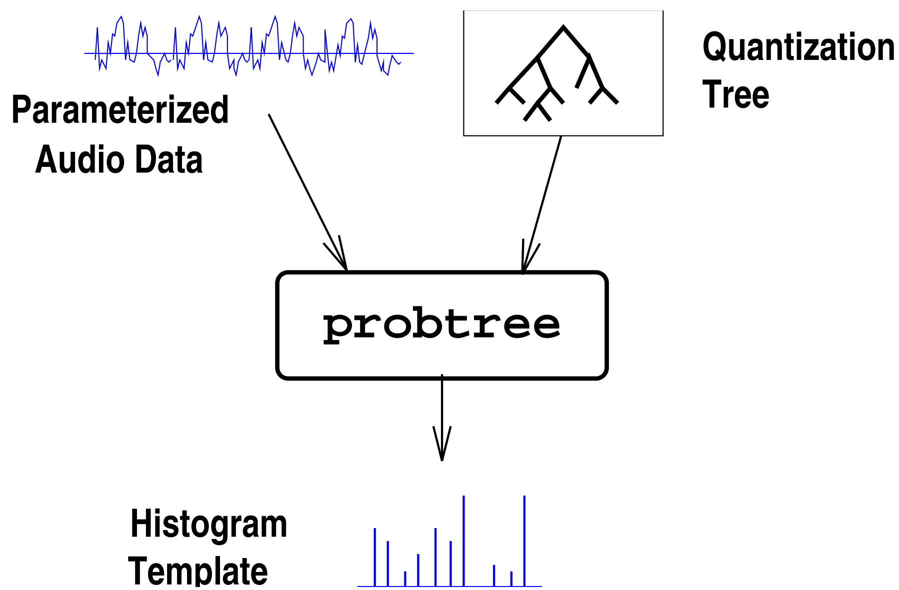

Figure 1.4: probtree.eps

1.2.3 probtree Overview

The tree partitions the feature space into L non-overlapping regions or "cells," each of which corresponds to a leaf of the tree. Though the tree can be used as a classifier, by labeling each leaf with a particular class. When classifying a sufficient amount of class-labeled heterogeneous data, each leaf will get data from a number of classes "routed" to it. By choosing the most popular class, each leaf can be labeled with the the class whose data is most likely to end up there. Such a classifier will not be robust, as in general classes will overlap such that a typical leaf will contain data from many different classes. A better way to capture class attributes is to look at the the ensemble of leaf probabilities from the quantized class data. A second of data will result in 100 feature vectors (ignoring window effects), and thus 100 different leaf labels. If a histogram is kept of the leaf probabilities, such that if, say, 14 of the 100 unknown vectors are classified at leaf j then leaf j is given a value of 0.14 in the histogram. The resulting histogram captures essential class qualities, and thus serves as a reference template against which other histograms may be compared.

1.2.4 histdist Overview

1.3 Frequently Asked Questions

Q: *Vector quantizers suck. We tried them for speech recognition back in the old days, and Gaussian models rule — they make better use of training data, generalize better, have better classification performance, etc. etc. etc...*

A: Yep.
Several points to make here. 1: We're not trying to *model* the data, we're trying to *find discriminative features*. The difference is this: given the complicated distributions we are looking at, we don't want to waste effort modeling variation that doesn't involve class boundaries. EM-trained Gaussian classifiers do just that; and they are subject to the curse of dimensionality as well. 2: The tree is supervised, which is a much more difficult proposition in the continuous domain. Dealing with insanely complicated feature spaces such as music requires supervision to pick out the important variations from the general wildness. There's no guarantee that things will even be Gaussian.

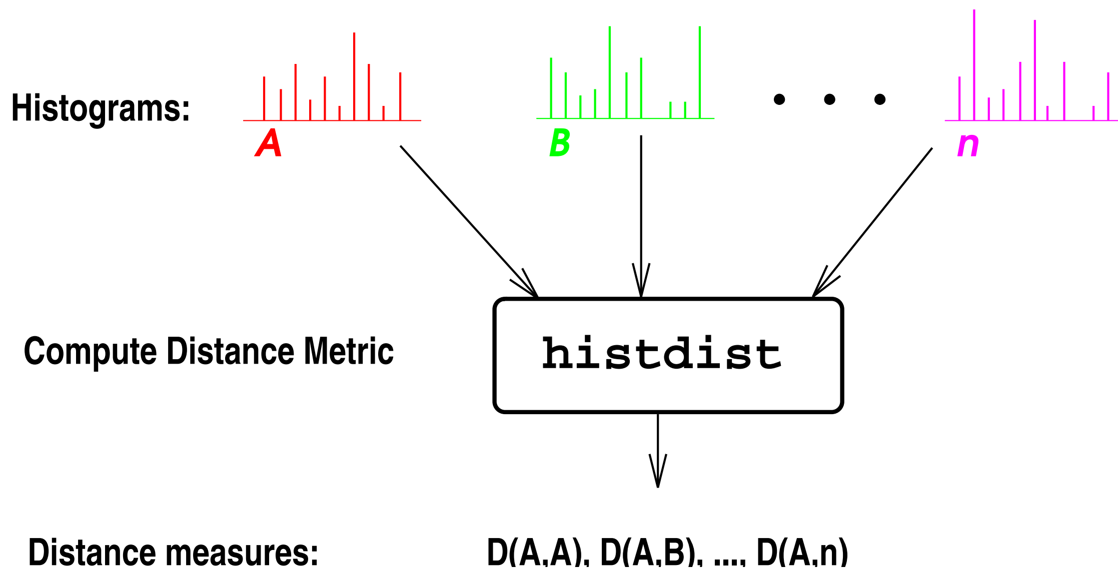

Figure 1.5: histdist.eps

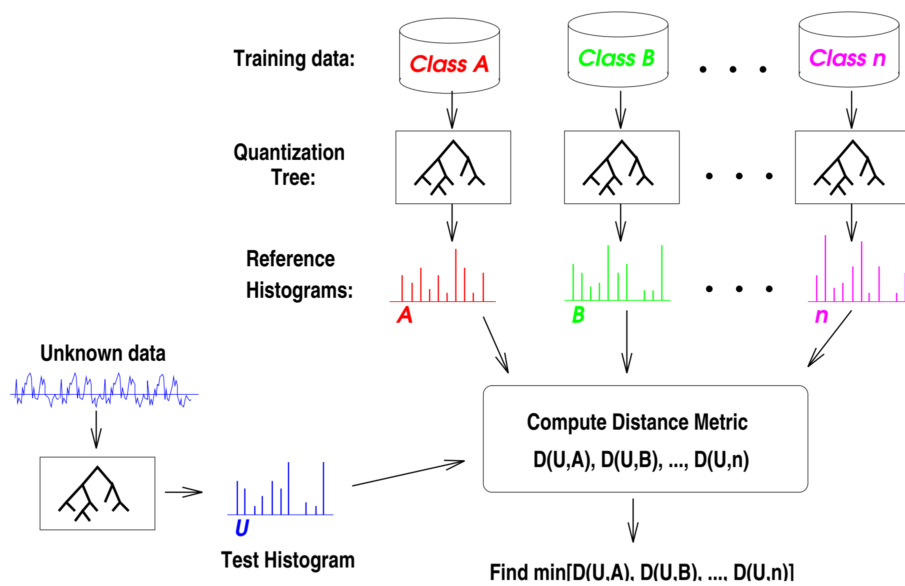

Figure 1.6: dist.eps

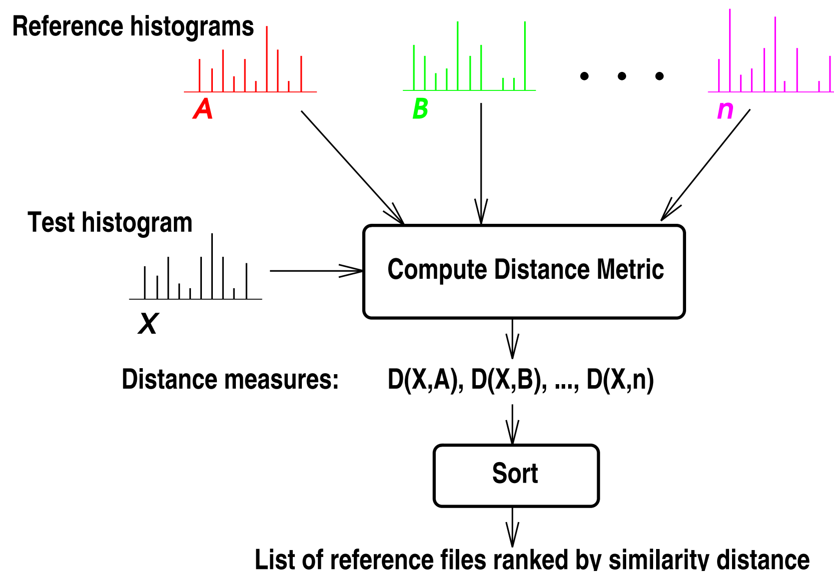

Figure 1.7: compare.eps

Q: *Why do you allow zero histogram counts? Aren't those an artifact of sparse data? Wouldn't smoothing help?*

A: Ahh — your background in language modeling is showing. Zero counts are not good when you're trying to model n-gram probabilities, which are probably never zero for real languages, and if you estimate them as such you will do badly when they do occur. For this domain, I don't see any reason why a bin count shouldn't be zero. Even if it isn't zero (just very small), it will then make only a very small difference to the distance measure—not a catastrophe. Smoothing would only add noise (though it might improve zero-sensitive distance measures such as the Kullback Liebler distance). If it was desirable to avoid zero bin counts, my approach would be to prune the tree, so that fewer bins were available, and hence will have fewer zero counts (but less "resolution").

Q: *Why are your decision boundaries so constrained? Wouldn't general hyperplanes, or even more flexible boundaries from a neural network, be better?*

A: Yes *but*. Certainly, more general hyperplanes would be better, but the

problem is how to find them! The space of possible hyperplanes is too vast to search efficiently, and you typically need iterative algorithms. By restricting hyperplanes to be parallel to the feature axes, you reduce the n-dimensional best hyperplane search to n 1-dimensional searches, which is far more tractable—it's easy to find the (provably) "best" hyperplane in a linear search. Yes, more flexible decision boundaries would be more efficient, and reduce the size of the tree, but my philosophy has been that it's far easier just to throw a few more nodes on the tree to get the desired boundaries. Note that the tree, constrained as it is, can still model *any* n-dimensional boundary or boundaries, albeit coarsely. Things which will break more conventional classifiers, such as bull's-eye or onion-skin class distributions, won't break the tree (though you may need a whole lot of data to train one).

Chapter 2

Tutorial

2.1 Signal Processing

2.2 Using Quantization Trees

In this tutorial, you will see how to grow a tree based on training data and use it to classify unknown data. In addition, you will see how to generate histograms from training data and the tree you just grew, how to use the histograms to find a distance measure between unknown data files.

Commands in this tutorial are in the shell script `tutorial.csh` in the distribution's `test` directory. Let's go through the commands one at a time. Things you type in are prefixed with a ">" character; everything else is output. This tutorial uses two-dimensional, normally-distributed data from two classes, called (starting from 0) class 0 and class 1. (The file `makendata.m` is a Matlab script that will generate additional random data with the same distribution.) Each class has two data files: a large file for training and a file with rather less data for testing. Figure 2.1 shows a plot of the two test sets: you can see there is a substantial degree of overlap. We are going to attempt two things:

1. Classification: given an unknown data point, figure out which class it came from.

2. Similarity measure: Given a *number* of data points, calculate a measure of similarity to a different set of points. Hopefully, the test data for a particular class will be more similar to the training data for the same class than to the test or training data for the other class.

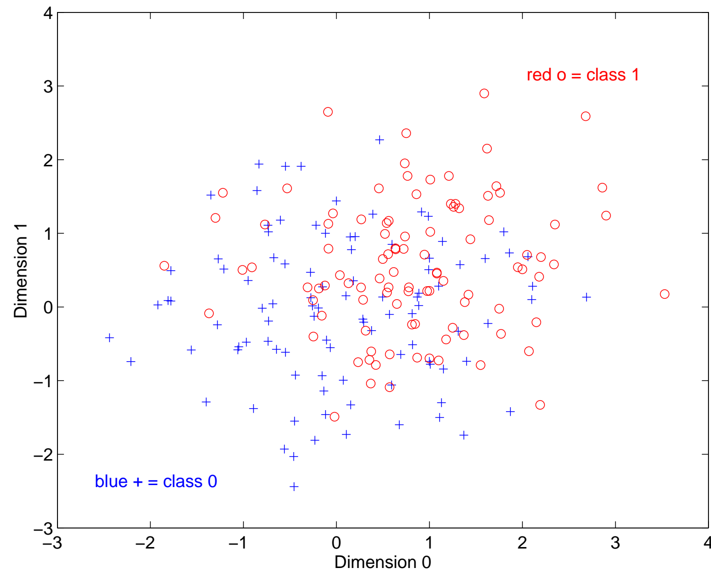

Figure 2.1: Tutorial two-class test data: 200 points.

Let's start. First, the tutorial script expects that you have installed the TreeQ executables in the bin subdirectory of the distribution. If you haven't (or don't know what I am taking about), set the `$bin` variable in the `tutorial.csh` script to the directory containing the compiled programs.

2.2.1 Growing Trees

The first thing we need to do is to convert the ASCII data files to the binary fff format. The program text2fff does this.

```
> text2fff -T 1 t*.txt
train0.txt --> ./train0.fff
train1.txt --> ./train1.fff
test0.txt --> ./test0.fff
test1.txt --> ./test1.fff
```

This produces 4 new fff files corresponding to each test and train input file. The files have a `.fff` extension, suggesting they are in the fff format,

which they are. The `-T 1` flag sets the trace variable; if you don't specify it the program will work silently. (Most TreeQ programs follow the same convention; larger trace number will give you increasing amounts of increasingly useless information. Invoking a TreeQ program with no arguments will print a usage description.)

OK, that was easy. Let's use growtree to grow a tree using the training data:

```
> growtree -t 0.0002 train0.fff train1.fff
 max tree depth     :  20
 mass weight factor :  1.000000
 min MI threshold   :  0.000200
 time extent        :  1
 center time        :  0
 vector size        :  2
 training vectors   :  10000
 number of classes  :  2


Finding root split...
L child 2    Level 1  sd:  0 tx: 0 npts: 5302    MI: 0.017380
L child 3    Level 2  sd:  1 tx: 0 npts: 1972    MI: 0.004138
L child 4    Level 3  sd:  0 tx: 0 npts: 538     MI: 0.001084
L child 5    Level 4  sd:  0 tx: 0 npts: 355     MI: 0.000444
R child 6    Level 5  sd:  0 tx: 0 npts: 312     MI: 0.000385
R child 7    Level 6  sd:  0 tx: 0 npts: 308     MI: 0.000251
        .
        .
        .
```

OK, that gives several more pages of output, which I've deleted. The tree growing procedure gives you a line of output for every split it makes in the data, (and thus every node it adds to the tree). It tells you whether the node was a left or right child of the parent, it's level in the tree, the dimension and time extent of the split, the number of training points used to find the split, and the amount of mutual information given by the split. The `-t` flag sets the *split threshold* below which growtree will stop adding nodes. Setting the split threshold is sometimes tricky as it depends on both the amount of training data and the distribution; I typically start with a

large threshold, then decrease it exponentially (by adding zeros) until I find a reasonable value. (The exact tree size isn't crucial because we can always prune the tree using choptree, as we will see a bit later.)

Let's skip to the end:

```
        .
        .
        .
Forcing leaf 307
Forcing leaf 308
Forcing leaf 309
Forcing leaf 310
Writing 311 nodes to tree.tfl
```

So, assuming everything worked, we now we have a tree file `tree.tfl`. Cool. The "Forcing leaf" business is a consequence of the tree structure. Every non-leaf node is a split, while the leaves correspond to the partitions on either side of the parent node's split. If we set the threshold high enough to produce only one split (node), we need to add two children leaves to represent the left and right partitions. Thus for a n-leaf tree, regardless of how balanced it is, there are always $2n - 1$ nodes (counting the leaves as nodes).

But a 311-node tree is probably too big for our toy problem (especially because we only have 200 test data points). Let's use choptree to create a smaller tree, having, say, 50 leaves:

```
> choptree -T 1 -keep 50 tree.tfl
tree --> tree-50.tfl (50 leaves)
```

This creates a new tree file pruned to 50 leaves, identical to the tree produced by growtree if we had the foresight to set the threshold that would stop growth at precisely 50 leaves. Let's take a look at our new tree, using showtree:

```
> showtree -h -m tree-50.tfl
```

```
=================== Tree Information =====================

                 Number of nodes: 99
                Number of leaves: 50
                               .
                               .
                               .
```

I won't show the entire output, because you can see that for yourself. You can also use showtree to generate a Matlab file that will plot the tree partition of the feature space (well, of the first two dimensions, anyway). Figure 2.2 shows what that looks like for our two-class distribution, plotted over the test data.Some things to note: the tree puts most of its splits near the boundary between the two classes. Also, the tree was trained on the training data and is thus independent of the test data shown in the plot.

Showtree can tell us a lot of information about the tree, and thus about our training data. It might look from Figure 2.2 that most of the splits are horizontal (in dimension 1). This is counter-intuitive because the classes are most separable in dimension 0. (The means are separated by 0.87 in dimension 0 and only by 0.5 in dimension 1). Let's use showtree to count how many splits are in each dimension:

```
> showtree -n tree-50.tfl | fgrep 'Dimension: 0' | wc
      60     240    1680
> showtree -n tree-50.tfl | fgrep 'Dimension: 1' | wc
      39     156    1092
```

This tells us that 60 of the 99 splits were in dimension 0 while only 39 of the splits were in dimension 1. Thus dimension 0 is more "important" for splitting, and hence classification. This is a useful thing to know; if you have lots of dimensions, you might be able to throw some away if they don't help the classification task. (A better metric takes into account where each split is in the tree; the better the split in terms of its MI the more "important" it is. See [])

2.2.2 Using Trees for Classification

OK. Now that we have a tree, we can do stuff with it. Perhaps the most straightforward thing we can do is to use it as a classifier. Let's label each

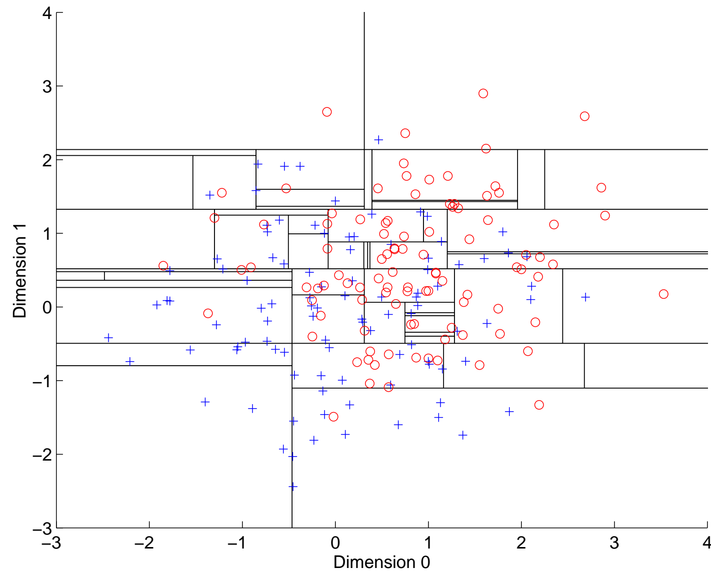

Figure 2.2: Tree partition of feature space.

leaf with the class that has the most training data in that leaf's partition. This is what probtree does. Let's try it:

```
> probtree -c -tfl tree-50.tfl train0.fff train1.fff
Writing 99 nodes to tree-50.tfl
```

Exciting, huh? If you use showtree -n you will see that every node now has a class label. We can plot the tree partition, and color every leaf cell according to its label, as in Figure 2.3. See how the tree partitions the space into two regions corresponding to the two classes. As you add leaves to the tree, the zig-zag boundary will more closely approximate the optimal linear boundary.

Now we can use this tree as a classifier. We just take unknown data, throw it at the tree, and see in which leaf it ends up. Looking at the class label of the leaf gives us a reasonable guess as to the class of the unknown data point. The quantize program does exactly this: you give it a tree and some unknown data, and for each data point it finds the class of the associated leaf. Let's try it:

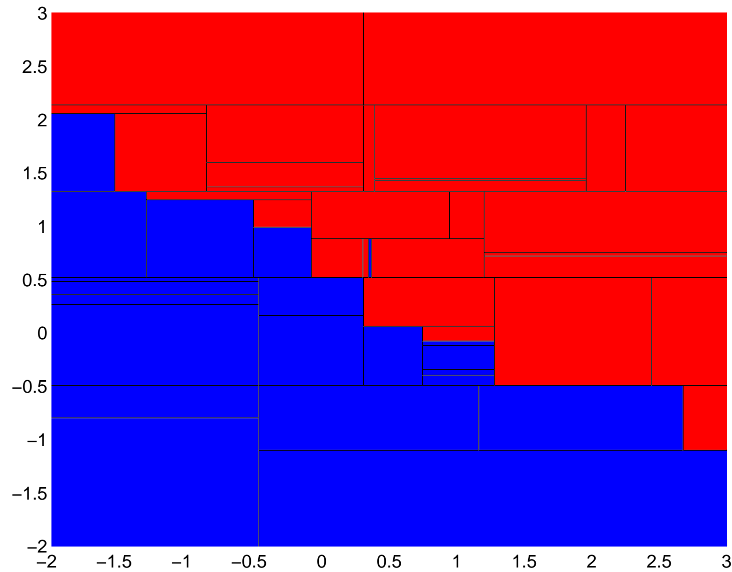

Figure 2.3: Tree partition, colored by class label.

```
> quantize -c -tfl tree-50.tfl test0.fff | fgrep 0 | wc
      70       70      140
> quantize -c -tfl tree-50.tfl test0.fff | fgrep 0 | wc
      67       67      134
```

What these commands did is to count how many test points were correctly classified for each class data set. Since there were 100 points in each class, the tree correctly classified 70% of the class 0 data and 67% of the class 1 data. This is actually not bad, considering the degree of overlap of the classes—the means are only 1 standard deviation apart. Note that the best we can *possibly* do, assuming I've done the math right, is:

$$\int_{-\infty}^{0.5} \frac{1}{\sqrt{\pi}} e^{(-t^2)} dt = 0.76$$

or 76% correct.

2.2.3 Using Histogram Templates

Well, for real-world problems we won't be looking at data points one at a time; rather we'll be looking at an ensemble of them. This is why I've introduced the concept of a "histogram template." What this is is just a count of the number of data points that end up in each tree leaf. This turns out to be a usefully compact way of representing data: instead of this huge waveform file, you just have this small array of counts.

The program probtree takes a tree file and some data, and generates a histogram file for each input data file. Here we go:

```
probtree -T 1 -tfl tree-50.tfl train*.fff test*.fff
train0.fff --> ./train0.tab
train1.fff --> ./train1.tab
test0.fff --> ./test0.tab
test1.fff --> ./test1.tab
```

What we've just done is to generate histogram tab file for each data file. Let's take a look at these (using showhist +h -f) and plotting them to get something like Figure 2.4. This shows the two histograms for the two classes' test data. Besides a few zero counts caused by the small amount of test data, they look pretty different. This is good, because the *are* different; we've trained the tree to make them that way. How different are they? The program histdist will calculate a numerical measure. Let's try it:

```
>histdist -T 3 -v train*.tab test*.tab
Using Euclidean distance

Reference files: train0.tab train1.tab test0.tab test1.tab
   1 train0.tab  0.000000 0.849268 0.300670 0.909555
   2 train1.tab  0.849268 0.000000 0.828947 0.489738
   3 test0.tab   0.300670 0.828947 0.000000 0.912944
   4 test1.tab   0.909555 0.489738 0.912944 0.000000
```

We just calculated a Euclidean distance measure between every combination of the four input histograms. Row i and column j is the distance between files i and j, counting as they are ordered on the command line.

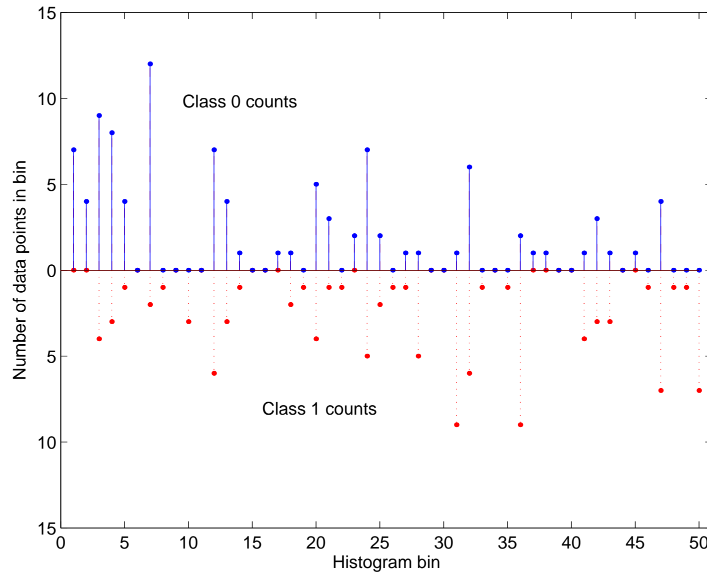

Figure 2.4: Histogram counts for the two classes.

Thus the diagonal is always zero as a file is always similar to itself, and has a distance of zero. The distance netric increases as files become less similar. Here, we only really care about the distances between the test and training templates. histdist can handle this by specifying a first group of *reference* files and a second group of *test* files using a "+"; only the intra-group distances are calculated. Watch:

```
> histdist -T 3 -v train*.tab + test*.tab
Using Euclidean distance

Reference files: train0.tab train1.tab
   1 test0.tab   0.300670 0.828947
   2 test1.tab   0.909555 0.489738
```

This tells us what we want to know: that the test0 template is closest to the train0 reference and the test1 template is closest to the train1 reference. (The fact that distances aren't symmetric is a result of the absurdly small amount of test data. If you use more data you will see the distance matrix

become more symmetric.) histdist can optionally sort the test templates by distance; this is how the music retrieval demonstration works. Given an unknown query template, histdist will return a list of reference files ranked by similarity.

Well, that's the end of the tutorials. Hope you now have a better idea of how things work, and can start using the package for something useful.

Chapter 3

Man Pages

3.1 cep2spec

cep2spec

3.1.1 usage

cep2spec

3.2 choptree

One of the many advantages of a tree-structured quantizer is that it can be pruned to yield a smaller tree. Choptree does exactly that. It reads in a tree file, and depending on the command-line arguments, trims the tree to the desired number of leaves, and writes the resulting smaller tree to an output file.

3.2.1 usage

choptree

3.3 fff2text

Fff2text is a simple sausage-grinder program that takes fff-format files and prints them to stdout in ASCII format. You can use this for debugging or use Matlab to plot or otherwise manipulate the data, using the `>> load -ascii` command. Each output line contains one ASCII sample. Multiple dimensions are printed on the same line. The result is an ASCII table having nSamps rows and FFFVecSize columns. See what these numbers mean in the fff (Chapter4.1.1).

3.3.1 usage

Flag	*Param*	*Description*	*Default*
-h		Print header	no

The one option is pretty obvious; it will print the header and omit the data. the default is to print only the data. If you want both, stop whining and run it twice.

3.4 growtree

growtree

3.4.1 usage

growtree

3.5 histdist

histdist

3.5.1 usage

histdist

3.6 nearhist

nearhist

3.6.1 usage

nearhist

3.7 probtree

probtree

3.7.1 usage

probtree

3.8 corrdist

This program computes the simple vector autocorrelation distance of a parameterized audio file in fff format. First, note that a fff file is a sequence of vectors, called frames, indexed by time. Call an vector o (for observation) observed at time t $O(t)t$. The vector correlation (similarity) between any two frames in the file can be calculated as the inner (dot) product of the vectors. More usefully, we can calculate the similarity $S(i,j)$ between two windows of length W starting at frames i and j as the sum of the inner products:

$$S(i,j) = \sum_{t=0}^{W-1} O(i+t) \cdot O(j+t),$$

This calculation is exactly what corrdist produces.

3.8.1 usage

```
corrdist [options] test.fff
```

Where `test.fff` is an input fff file and the options are:

Flag	*Param*	*Description*	*Default*
-T	i	Trace level	0
-help		Print usage help	
-w	i	Window width	2
-s	i	Advance window by this step	1
-z		zero mean data before correlating	False

Given a file of N frames, and specifying W on the command line (with the `-w` option), corrdist produces a square matrix of ascii values such that the (i,j)th value is $S(i,j)$. If the default step size of 1 is used, the size of the output matrix will be $(N-W) \times (N-W)$. Note that the matrix is symmetric $S(i,j) = S(j,i)$, and any row/column will be a maximum where $i = j$ (on the diagonal of the square). The matrix is written to `stdout`, and is typically large, so be sure to pipe it to a file.

The `-s` option allows to select a "step size" such that i and j vary incrementally by this step size. In other words a step size of 2 means $S(i,j)$ is computed only for even numbers hence the output matrix is half the size of one produced with the default step size of 1. The `-z` option will subtract the global mean from each data vector before doing the autocorrelation. This will not affect the output.

3.9 procdata

Obsolete. Use the `HCopy` tool from the `HTK` toolkit to process audio data into MFCCs (`http://htk.eng.cam.ac.uk/`). Here's an example config file that will help:

```
TARGETLABEL=HTK
TARGETKIND=MFCC_0
TARGETRATE=100000.0
SAVECOMPRESSED=F
SAVEWITHCRC=F
WINDOWSIZE=250000.0
USEHAMMING=T
PREEMCOEF=0.97
NUMCHANS=26
CEPLIFTER=22
NUMCEPS=12
SOURCEFORMAT=WAV
SOURCERATE=625
SOURCEKIND=WAVEFORM
```

If you are working on Windows (including Ming or Cygwin) or DOS, you might want to make sure the byte ordering is compatible by adding the following options. (TreeQis not smart enough to figure this out automatically.)

```
NATURALREADORDER=T
NATURALWRITEORDER=T
```

3.10 quantize

quantize

3.10.1 usage

quantize

3.11 showtree

To examine a tree file, use showtree. Without arguments, showtree prints the tree file header, which is probably more information than you really care about. With arguments, it will optionally display the node data (which can get complicated for large trees) or even output the structure of the tree partition for graphical display.

3.11.1 usage

showtree

3.12 text2fff

As you might expect, text2fff does the reverse of fff2text — it takes an ASCII table with nSamps rows and FFFVecSize columns, and outputs a fff-format files.

3.12.1 usage

It has a few more options, as it must produce a sensible fff header:

Flag	*Param*	*Description*	*Default*
-T	i	Trace level	
-rate	f	Sampling rate in Hz	100
-kind	i	Parameterization kind	MFCC

The "parameterization kind" is the parmKind integer code that's specified in the file fffio.h. This is for compatibility with HTK; nothing in TreeQ really pays much attention, though some programs will complain if asked to operate on multiple files with different sample rates or parmKinds.

Bugs: I was too lazy to dynamically allocate the input buffer, which may overflow for extremely large input files. In this case, text2fff will crap out gracefully and tell you what to do to fix the problem. (simply redefine a constant and recompile)[1].

[1]Hey, maybe that *really is* dynamic reallocation!

Chapter 4

tqlib Library Reference

I'm afraid this chapter will have to wait until I am retired and have enough time to properly document the library functions. For now I will give only a brief overview of the file structure; you will have to rely on comments in the code for more detailed information. Lucky for you, I've been liberal with the comments and hopefully they make some sense.

The tqlib is organized into the following files. Each has a `.c` code file and a `.h` include file as you might expect.

TreeQ Basic file I/O, housekeeping and error functions; also contains some useful macro definitions.

dbase Contains code to efficiently manage large amounts of vector data.

fffio Contains I/O code for fff format data files, which is similar to that of HTK.

hist Contains functions for processing and I/O of tree-generated histograms.

procargs Command-line argument processing functions.

treeio Code for managing, reading, and writing tfl-format tree files.

4.1 File Formats

4.1.1 Fungible File Format (fff)

The fff file format is a way of easily storing and accessing multidimensional numerical data. In the TreeQ package, it use primarily used for storing

int	n Samples
int	sampPeriod
short	sampSize
short	parmKind

Table 4.1: FFF file header

int	bins
float	tot
float	range
float*	count

Table 4.2: Histogram file header

parametrized audio data, though it can be used for arbitrary data as demonstrated in the Tutorial (Chapter 2).

Rather then create *ANOTHER* bloody data file format, the fff format is nearly identical to, and hence compatible with, the HTK format. Note that this code *is NOT intended as a replacement for HTK*, which is a powerful package well worth obtaining. In particular, several nice features of the HTK format are not supported, such as compressed files and checksums.

The structure of an fff file is straightforward: a 12-byte header followed by a variable-length array of either floats or shorts. Table 4.1 shows the header of an fff file. The remainder of the file is `Samples x sampSize` data bytes, whose datatype (`float` or `int`) is encoded in the `parmKind` field (see the tqlib header file `fffio.h` for a list). The `sampPeriod` parameter specifies the period between samples in 100 nS units. A fff file can be conveniently manipulated using the tqlib calls and macros.

4.1.2 Histogram Files

These are pretty dead-simple. A histogram is just an array of counts, so that is what the file looks like, plus a header as shown in Table 4.2.

There are `bins` count values following the header, stored as floats (though they will always be integer-valued). The maximum count is stored in the `range` header field while the sum of all counts is in `tot`. The `count` field is

used to store a memory pointer and is thus meaningless in the file.

4.1.3 Tree Files (tfl)

The output file from growtree is a tree file. This is a binary file and hence may not be portable across different platforms (PC users beware!). The tree file has a fixed-length header followed by a variable number of node records. The header contains information about how many nodes and some statistics on the data used to construct the tree. Each node structure contains an ID number, split threshold information, the MI generated from the split at that node, and "pointers" to parent and children nodes. In the file, the "pointers" are actually ID numbers of the relative nodes – these are converted back to real pointers (memory addresses) when the file is read in. But you don't have to worry about that; unless you are doing extreme hacking (you sad git) you can access the tree files using the tqlib routines.

Chapter 5

Acknowledgments

I've been inspired by, and cheerfully stolen code from, a number of fine people over the years. All are due much thanks. First of all, Les Niles wrote the argument command-line processing package `procarg` while at LEMS at Brown University. For excellent examples of clean, crisp, concise, and caffinated[1] code, you could do far worse than check out `procarg.c` in the tqlib library. I am indebted to Tony Robinson's speech cookbook package which inspired some of the `tqlib` routines. I've based the Fungible File Format (fff) directly on the HTK file format developed by Steve Young at the Cambridge University Engineering department. Finally, thanks to colleagues and staff over the years at Brown University, Cambridge University, and the Institute of Systems Science at the National University of Singapore, for invaluable help and support. Much of this work was completed under a Fulbright Fellowship administered by the Committee for the International Exchange of Scholars.

5.1 Add-ons and helpers

5.1.1 Matlab

Matlab, from The Mathworks[2]. This I've found useful for both generating test as well as analyzing the output. The figures in (see the tutorial, Chapter2) were all done using Matlab.

[1] (but cryptic)
[2] http://www.mathworks.com

5.1.2 HTK

Anyone doing serious speech research needs the HTK Hidden Markov Toolkit from Cambridge University[3].

5.1.3 Cygwin

Cygwin is an awesome Linux-like environment for Windows `http://www.cygwin.com/`. TreeQ has been built and runs naturally on Cygwin.

5.1.4 SOX

SOX is an invaluable tool for converting audio file formats. In particular, it can convert just about any common file into a raw format suitable for procdata. If you don't have it, you can get it at the author's web site: `http://www.spies.com/Sox/`. Besides a Unix source, there's also a DOS executable available.

5.1.5 TREC

TREC stands for the Text REtrieval Conference, which is a somewhat misleading name for a software package. This C-language package computes a standardized measures of retrieval performance, and was developed so that different retrieval strategies could be meaningfully compared. The TREC package was written by Gerald Salton at Cornell University. It may be obtained at `ftp://ftp.cs.cornell.edu/pub/smart`.

5.1.6 LVQ2

This is the Leaning Vector Quantizer package developed at the Helsinki University of Technology[4]. It is available at `http://www.cis.hut.fi/nnrc/nnrc-programs.html`. I mention it here because it is a supervised vector quantizer similiar in function to TreeQ, but very different in philosophy and implementation. It also has some useful utilities such as a Sammon mapper, which produces a non-linear 2-d mapping of high-dimensional data useful for visualization. An interesting project might be to compare the LVQ and TreeQ approaches in terms of efficiency and accuracy.

---

[3]http://htk.eng.cam.ac.uk/

[4]http://www.cis.hut.fi/nnrc/

Appendix A

Technical Stuff

Bibliography