Chapter 22

Classifying

Many of the most important analytic tasks involve classifying or categorizing various things. In this chapter we will discuss two general approaches to classifying: *parametric* classifying and *non-parametric* classifying. In the first instance, we will see how numerical data can be categorized according to arithmetic ranges. We will then revisit the **humsed** command and learn how it can be used to classify different types of non-numeric data tokens.

The *recode* Command

Suppose that we have a Humdrum spine that contains numerical information representing the moment-to-moment heart-rate of a listener. Heart rate is related to arousal level and so we might use our data to identify passages that tend to arouse listeners. Since the average heart-rate of listeners differs, we are interested primarily in the rate-of-change. We can use the **xdelta** command to calculate the differences in heart-rate between successive values.

```
xdelta -s = heart.dat > changes
```

The example below displays the input (left) spine and the corresponding output (right) spine for the above command:

```
**heart   *Xheart
=133      =133
55        0
56        1
55        -1
=134      =134
58        3
56        -2
55        -1
=135      =135
57        2
55        -2
56        1
```

```
=136      =136
55        -1
60        5
62        2
=137      =137
61        -1
59        -2
59        0
=138      =138
*-        *-
```

A certain amount of heart-rate variation is to be expected because of monitoring equipment and other variables. So we are primarily interested in large changes of heart-rate, such as the change occurring in measure 136. The **recode** command allows us to classify numerical data according to value or range. In the above case, we may be interested in identifying acceleration or decelerations that exceed some threshold. The **recode** command requires that the user supply a *reassignment file* that specifies how numerical values are to be reassigned. In our heart-rate application, we might create the following reassignment file, named `reassign`. Reassignment files obey the following syntax: for each line, *conditions* are given on the left followed by a single tab, followed by a *reassignment string.*

```
>3       +event
<-3      -event
else     .
```

The above reassignment file may be interpreted as follows: if the numerical value is greater than 3, then output the string "`+event`"; if the numerical value is less than −3, then output the string "`-event`"; otherwise output a string consisting of an isolated period (`.`). We can invoke an appropriate command as follows:

```
recode -f reassign -i '**Xheart' -s ^= changes
```

The **-f** option is required, and is used to identify the file containing the reassignment information. The **-i** option is also required, and is used to identify the exclusive interpretation for the data to be processed. The **-s** option tells **recode** to skip records matching some specified regular expression — in this case, to skip barlines. Finally, "`changes`" is the name of our input file.

The result of applying this process to the right-most spine in the above example is given below:

```
*Xheart
=133
.
.
.
=134
.
.
.
```

```
=135
.
.
.
=136
.
+event
.
=137
.
.
.
=138
*-
```

Notice that we have used **recode** to drastically reduce the volume of data by transforming the input into a set of more basic cateogires.

Having constructed our new output spine, we can further process this information in various ways. For example, we might assemble this spine to our original musical score. Then we might then use **grep -n** to located any points in the score where a heart-rate related event has occurred.

Permissible relational operators used by **recode** include the following:

==	equals
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
else	default relation

Conditions are tested in the order given in the reassignment file. Thus if a numerical value satisfies more than one condition, only the first string replacement is made. Consider the following reassignment file:

```
<=0        LOW
>100       HIGH
>0         MEDIUM
```

The order of specification is important here. If the `MEDIUM` condition was specified prior to the `HIGH` condition, then all values greater than one hundred would be categorized as `MEDIUM` rather than as `HIGH`. Only a single **else** condition is allowed in a reassignment file; when it is present, the else statement should appear as the last reassignment.

Classifying Intervals

The **recode** command has innumerable applications. Suppose we wanted to determine how frequently ascending melodic leaps are followed by a descending step. Let's consider two different ways of distinguishing steps and leaps: a "semitone" method and a "diatonic" method. In the first method, we might define a step interval as either one or two semitones. Our reassignment file (dubbed "`reassign`") might appear as follows:

```
>=3      up-leap
>0       up-step
==0      unison
>=-2     down-step
<=-3     down-leap
```

Given this reassignment file, we can now begin our processing. In the first method, we translate to semitone data using **semits**, translate to semitone-differences using **xdelta**, and then classify into five interval types using **recode**. The **context -n 2** command will create pairs of interval types, then **rid**, **sort** and **uniq -c** are used to generate an inventory. Finally, we use **grep** to identify what happens following ascending leaps:

```
semits melody | xdelta -s = | recode -f reassign \
   -i '**Xsemits' -s = | context -n 2 | rid -GLId | sort \
   | uniq -c | grep 'up-leap .*$'
```

An alternative way of distinguishing steps from leaps is by diatonic interval. For example, we might consider a diminished third to be a leap, while an augmented second may be considered a step. In this case, we can use the **mint** command to determine the melodic interval size; the **-d** option limits the output to diatonic intervals and excludes the interval quality (perfect, major, minor, etc.). The appropriate reassignment file would be:

```
>=3      up-leap
==2      up-step
==1      unison
==-2     down-step
<=-3     down-leap
```

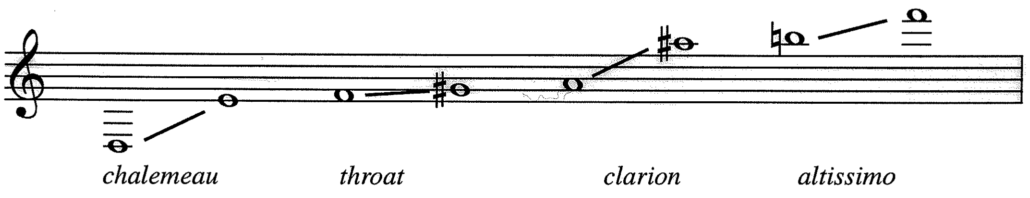
The appropriate command pipe would be:

```
mint melody | xdelta -s = | recode -f reassign -i '**mint' \
   -s = | context -n 2 | rid -GLId | sort | uniq -c \
   | grep 'up-leap .*$'
```

Clarinet Registers

Consider another use of the **recode** command. Imagine that we wanted to arrange Claude Debussy's *Syrinx* for soprano clarinet instead of flute. Our principle concern as arranger is determining what key would be especially well suited to the clarinet. Tone color is particularly important for this piece. The clarinet has four fairly distinctive tessituras as shown in Example 21.1. These are the *chalemeau* register (dark and rich), the *clarion* register (bright and clear), the *altissimo* register (very high and piercing), and the *throat* register (weak and breathy).

Example 21.1. Clarinet registers (notated at concert pitch).



Suppose we wanted to pick a key that satisfies two conditions: (1) it is not out of range for the clarinet, and (2) it minimizes the number of notes played in the throat register. We can use **recode** to classify all pitches according to the following reassignments:

```
>=30   too-high
>=23   altissimo
>=8    clarion
>=5    throat-register
>=-10  chalemeau
else   too-low
```

Now we simply explore various transpositions using **trans** and create an inventory of pitch types. For Debussy's *Syrinx*, the minimum number of throat tones (without exceeding the clarinet's range) occurs when we transpose down a major sixth:

```
trans -d -5 -c -9 syrinx | semits | recode -f reassign \
   -i '**semits' -s = | rid -GLId | sort | uniq -c
```

Open and Close Position Chords

Inputs to the **recode** command can be quite sophisticated. Consider, for example, the task of classifying chords as "open" or "close" position. According to one definition, a chord is said to be in "open" position when the the interval separating the soprano and tenor voices is an octave or greater. One music theorist has claimed that close position chords are more common than open position. How might we test this?

In determining an appropriate sequence of Humdrum commands, it is often helpful to work backwards from our goal. We'd like to end up with a spine that simply encodes the words "open" or "close" for each sonority. This classification will be based on the distance separating the soprano and tenor voices. Our reassignment file might be as follows:

```
<=12      close
>12       open
```

We will need to extract the soprano and tenor voices, translate the pitch representation to `**semits` and use **ydelta** to calculate the semitone distance between the two voices. In the following set of commands, we have also added the **ditto** command to ensure that there are semitone values for each sonority.

```
extract -i '*Itenor,*Isopran' inputfile | semits -x | ditto \
   | ydelta -s = -i '**semits' | recode -f reassign \
   -i '**Ysemits' -s = > tempfile
grep -c 'open' tempfile
grep -c 'close' tempfile
```

The **grep -c** commands tell us whether open position sonorities are more common than close position sonorities.

Flute Fingering Transitions

There is no fixed limit to the length of a reassignment file. Consider for example, the following file named `map`. Each `**semits` value from C4 (0) to C7 (36) has been assigned to a schematic representation of flute fingerings. The letter 'X' indicates a closed key, whereas the letter 'O' indicates an open key. The first letter pertains to the left thumb; the next group of four letters pertain to the ensuing fingers of the left hand; the final group of letters pertain to the right-hand fingers. The little finger of the right hand is able to play three keys (labelled X, Y, and Z). Fingerings are shown only for the first octave (from C4 to C5):

```
<0     out-of-range
==0    X-XXXO-XXXZ
==1    X-XXXO-XXXY
==2    X-XXXO-XXXO
==3    X-XXXO-XXXX
==4    X-XXXO-XXOX
==5    X-XXXO-XOOX
==6    X-XXXO-OOXX
==7    X-XXXO-OOOX
==8    X-XXXX-OOOX
==9    X-XXOO-OOOX
==10   X-XOOO-XOOX
==11   X-XOOO-OOOX
==12   O-XOOO-OOOX

etc.

else   rest
```

Suppose we wanted to determine what kinds of fingering *transitions* occur in Joachim Quantz's flute concertos. Since instrument fingerings are insensitive to enharmonic spelling, an appropriate input representation would be `**semits`. Having used **recode** to translate the pitches to fingerings, we can then use **context -n 2** to generate diads of successive finger combinations.

```
semits con* | recode -f map -s = | context -n 2 -o = > fingers
```

For example, if our input contains the pitch G5 followed by B4, the appropriate data record in the `fingers` file would be the following Humdrum double-stop:

```
X-XXXO-OOOX X-XOOO-OOOX
```

We could create an inventory of finger transitions by continuing the processing:

```
rid -GLI fingers | sort | uniq -c | sort -n
```

We could create a similar reassignment file containing fingers pertaining to the pre-Boehm flute. Suppose the revised reassignment file was called `premodern`. We could determine how the finger transitions differ between the pre-Boehm traverse flute and the modern flute. In Chapter 29 we will see how the **diff** command can be used to identify differences between two spines. This will allow us to identify specific places in the score where Baroque and modern fingerings differ.

The **recode** command can be used for innumerable other kinds of classifications. For example, `**kern` durations might be expressed in seconds (using the **dur** command), and the elapsed times then classified as *long*, *short* and *medium* (say). Sound pressure levels (in decibels) might be classified as dynamic markings (***ff***, ***mf***, ***mp***, ***pp***, etc.), and so on.

Classifying with *humsed*

The **recode** command is restricted to classifying numerical data only. For many applications, it is useful to be able to classify data according to non-numerical criteria. As we saw in Chapter 14, stream editors such as **sed** and **humsed** provide automated substitution operations. Such string substitutions can be used for non-parametric classifying. We can illustrate this with **humsed.**

Suppose we wanted to classify various flute finger-transitions as either *easy*, *moderate* or *difficult*. For example, F4 to G4 is an easy fingering, E5 to A5 is a moderate fingering, whereas C5 to D5 is difficult. As before, it is best to use a semitone representation so we don't need to consider differences in enharmonic pitch spelling. We can use the **semits** command to transform all pitches. Then we can use **context -n 2** to generate pairs of successive pitches as double-stops. We can then create a **humsed** script file (let's call it `difficulty`) containing substitutions such as the following:

```
s/5 7/easy/             [i.e. F4 to G4]
s/16 21/moderate/       [i.e. E5 to A5]
s/12 14/difficult/      [i.e. C5 to D5]
etc.
```

We can apply the script as follows:

```
humsed -f difficulty sonata*
```

Since there are a large number of possible pitch transitions, our script file is apt to be especially large. However, notes an octave apart have a high likelihood of having identical fingerings on the modern flute. A more succinct **humsed** script would deal with fingering transitions rather than pitch transitions.

```
s/X-XXXO-XOOX X-XXXO-OOOX/easy/
s/X-XXXO-XXOX X-XXOO-OOOX/moderate/
s/O-XOOO-OOOX X-OXXO-XXXO/difficult/
etc.
```

The three substitutions shown above apply to many more pitch transitions than the original transitions F4-G4, E5-A5, and C5-D5. The above three substitutions apply also to F5-G5, F5-G4, F4-G5, E4-A4, E4-A5, and E5-A4.

Having created a file classifying all fingering transitions as "easy," "moderate" or "difficult," we can characterize our Quantz flute concertos using the following pipeline:

```
semits Quantz* | recode -f map -s = | context -n 2 -o = \
    | humsed -f difficulty
```

The output will be a single spine that classifies the difficulty of all fingering transitions.

Classifying Cadences

Consider another application where we use **humsed** to classify cadences. Suppose we have Roman-numeral harmonic data (as provided by the `**harm` representation). In the case of Bach's chorale harmonizations, for example, cadences are clearly evident by the presence of pauses (designated by the semicolon). We can easily create a spine that identifies only cadences. Consider a suitable reassignment file (dubbed `cadences`):

```
s/V I;/authentic/
s/V7 I;/authentic/
s/V i;/authentic/
s/V7 i;/authentic/
s/IV I;/plagal/
s/iv i;/plagal/
s/iv I;/plagal/
s/V vi;/deceptive/
s/V VI;/deceptive/

etc.

s/^[IiVv].*$/./
```

(The precise file will depend on your preferred way of labeling cadences.) Remember that, unlike the **recode** command, all of the substitutions in a **humsed** or **sed** script are applied to every input line. The final substitution causes any record beginning with either an *i*, *i*, *v* or *V* to be changed to a null data token. In effect, any progression that is not deemed to be an authentic, plagal or deceptive cadence is transformed to a null data record. Using the above reassignment file, we could create a cadence spine using the following pipeline:

```
extract -i '**harm' chorales | context -o = -n 2 \
   | humsed -f cadences | sed 's/\*\*harm/**cadences/'
```

We first extract the `**harm` spine using **extract**. We then generate a sequence of two-chord progressions using **context** — taking care to omit barlines (`-o =`). We then use **humsed** to run the script of cadence-name substitutions. Finally, we use the **sed** command to change the name of the exclusive interpretation from `**harm` to something more suitable — `**cadences`.

Many more sophisticated variants of this sort of procedure may be used. For example, one could first classify harmonies more broadly. In so-called "functional" harmony, for example, supertonic chords in first inversion are normally considered to be subdominant functions. One could construct a whole series of re-write rules that classify harmonies in a variety of ways.

Orchestration

One of the simplest classifications in a musical score is whether or not an instrument is sounding or resting. Suppose we extracted the viola part from Beethoven's Symphony No. 1. We might use the **ditto** command to ensure that each data record encodes either a note, rest, or barline:

```
extract -i '*Iviola' symphony1 | ditto -s =
```

Let's append to this pipeline a **humsed** command that makes two string substitutions. The first substitution replaces all data records containing the lower-case letter `r` (i.e., rests) with the string `-viola`. The second substitution changes any record that does not begin with either a minus sign or an equals sign to the string `+viola`. In effect, we've transformed the viola part so that all data tokens encode either `+viola`, `-viola` or are barlines.

```
extract -i '*Iviola' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-viola/; /s/^[^-=].*$/+viola/' > viola
```

Now imagine that we repeat this process for every instrument in Beethoven's Symphony No. 1. In each case, we substitute the name of the instrument (preceded by a plus-sign or minus-sign) for the various note or rest tokens.

```
extract -i '*Iflt' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-flt/; /s/^[^-=].*$/+flt/' > flt
extract -i '*Ioboe' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-oboe/; /s/^[^-=].*$/+oboe/' > oboe
extract -i '*Iclars' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-clars/; /s/^[^-=].*$/+clars/' > clars
extract -i '*Ifagot' symphony1 | ditto -s = \
   | humsed 's/.*r.*/-fagot/; /s/^[^-=].*$/+fagot/' > fagot
```

etc.

When we are finished, we reassemble all of the transformed parts into a complete score.

```
    assemble cbass cello viola violn2 violn1 tromb tromp fagot \
       clars oboe flt > orchestra
```

We now have a file that contains data records that look something like the following excerpt:

```
+cbass   +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass   +cello -viola -violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass   +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass   +cello -viola -violn +violn -tromb -tromp +fagot -clars+oboe  +flt
-cbass   -cello +viola +violn +violn -tromb -tromp -fagot -clars+oboe  +flt
-cbass   -cello -viola -violn +violn -tromb -tromp -fagot -clars+oboe  +flt
=131     =131   =131   =131   =131   =131   =131   =131   =131  =131   =131
+cbass   +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass   +cello -viola -violn +violn -tromb -tromp +fagot -clars+oboe  +flt
-cbass   -cello +viola +violn +violn -tromb -tromp -fagot -clars+oboe  +flt
-cbass   -cello -viola -violn +violn -tromb -tromp -fagot -clars+oboe  +flt
+cbass   +cello +viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
+cbass   +cello -viola +violn +violn -tromb -tromp +fagot -clars+oboe  +flt
etc.
```

The first sonority indicates that all of the string instruments are playing, that the brass are inactive, and that all of the woodwinds are sounding with the exception of the clarinet.

A representation such as the above provides an opportunity to study instrumental combinations in Beethoven's orchestration. For example, the following command will count the number of sonorities where the oboe and bassoon sound concurrently:

```
    grep -c '+fagot.*+oboe' orchestra
```

It is better to express this count as a proportion of the total work. We can count the total number of sonorities in the work by omitting any leading plus or minus sign:

```
    grep -c 'fagot.*oboe' orchestra
```

How often are the oboe and bassoon resting at the same time?

```
    grep -c '-fagot.*-oboe' orchestra
```

Excluding *tutti* sections, do the trumpet and flute tend to "repell" each others' presence?

```
    grep -v '\-' orchestra | grep -c '+tromp.*-flt' orchestra
    grep -v '\-' orchestra | grep -c '+tromp.*+flt' orchestra
    grep -v '\-' orchestra | grep -c '-tromp.*-flt' orchestra
    grep -v '\-' orchestra | grep -c '-tromp.*+flt' orchestra
```

When all of the woodwinds are playing, which of the remaining instruments is Beethoven most likely to omit from the texture?

```
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-cbass'
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-cello'
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-viola'
grep '+fagot.*+clars.*+oboe.*+flt' orchestra | grep -c '-violn'
etc.
```

Many refinements can be added to this basic approach. For example, instead of classifying instruments as simply being "present" or "absent," we might distinguish various registers for each instrument — as we did with the clarinet when describing **recode**. We could then determine whether Beethoven tends to link, say, activity in the chalemeau register of the clarinet with low register activity in the strings.

Further refinements might include relating orchestration to structural aspects of the music. For example, we might use **yank** to extract sections of movements; we could then compare possible differences of orchestration between the first and second themes, for example. Similarly, we could reduce instruments to instrument classes, and examine how brass, woodwinds, strings, and percussion in general are related.

Reprise

A large number of analytic tasks simply involve classifying things. In general, two sorts of classifying methods can be distinguished: (1) a numerical or *parametric* classification can be used to reassign various ranges of numerical values into a finite set of classes or categories; (2) a *non-parametric* classification maps one set of words or terms into a second (usually smaller) set of words (used to label various classes or categories). In this chapter, we have seen that, for any Humdrum representation, parametric classification can be done using the **recode** command and non-parametric classification can be achieved using the *substitution* operation provided by the **humsed** command.