Class Notes for I546/N546

Christopher Raphael

December 8, 2009

Chapter 1

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a catch-phrase from statistics that describes a number of inventive ways to understand data by visualizing (or sonifying) them. Given our working motto of "Music as Data," music is also amenable to such techniques.

The class web page: http://www.music.informatics.indiana.edu/courses/I546/ contains a variety of materials we will use this semester. In particular there is a directory containing a collection of MIDI (=Musical Instrument Digital Interface) files. Please take a look at this directory and play some of the midi files using any generic midi player (usually your web browser will know how to find the appropriate player). Each midi file in this collection has an associated data file suffixed by ".dat", so a file named "xxx" would have both

1. xxx.mid (the midi file)
2. xxx.dat (the data file)

Generally speaking, the midi files are only for listening since they represent their music data in a rather complicated manner. See http://253.ccarh.org/handout/smf/, for instance, if you would like a detailed description of the midi file format. We will instead use the data (.dat) files for our analyses. They are derived, by program, directly from the midi files though they contain less information.

Each data file has two columns of numbers containing the onset times of notes, in midi "ticks," and the midi pitches:

times in midi ticks	midi pitches
t_1	p_1
t_2	p_2
t_3	p_3
$\vdots$	$\vdots$

For instance, "Mary had a Little Lamb" would begin something like:

0	64
256	62
512	60
768	62
1024	64
1280	64
1536	64
$\vdots$	$\vdots$

Midi ticks are a time unit defined in the midi file — usually the midi file tells how many midi ticks exist in each quarter (above we have assumed 256 ticks per quarter). We will generally just be interested

in the relative values (eg. t_1 twice (half) as large as t_2). Usually midi files represent changes of tempo in a separate tempo "track." Thus, even if there are change in tempo, any particular musical note value (e.g. quarter note) will be constant in midi ticks. This is useful for analysis of symbolic music since interpretive information is separated from score information.

The midi pitches are in the range of 0 – 127, with 60 representing "Middle C", 61 representing "Middle C#" etc. http://www.informatics.indiana.edu/donbyrd/Teach/MusicalPitchesTable.htm gives a listing of the midi pitches with their associated note names and frequencies in cycles per second (Hz).

1.1 Pitch Analysis (pitch_dist.r)

Most of the scenarios we will discuss in the class have associated R programs. I will indicate this as above: pitch_dist.r is the program that demonstrates the ideas of this section. These programs are all available on the class web page http://www.music.informatics.indiana.edu/courses/l546/.

We are interested in examining the pitch content of a particular piece of music. We will do this by counting the number of each type of pitch that appear in a piece. We can do this by either counting the instances of each pitch, or of each *pitch class*. We will call all instances of the note C pitch class 0, regardless of what octave they occur in. Similarly, we will call all instances of C# or D♭, pitch class 1, etc. giving 12 pitches class 0, . . . , 11 as in the following table.

C,B#	C#,D♭	D	D#,E♭	E,F♭	F,E#	F#,G♭	G	G#,A♭	A	A#,B♭	B,C♭
0	1	2	3	4	5	6	7	8	9	10	11

Note that we can compute the pitch class, c of a midi pitch, m by

$$c = m \bmod 12$$

where $a \bmod b$ gives the remainder when a is divided by b. For instance, note that all of the C's: $\ldots, 36, 48, 60, 72, 84 \ldots$ have remainder 0 when divided by 12, so all are in pitch class 0.

1.1.1 Tonic and Mode

Tonal music (music that is "in a key") has a main pitch that serves as the "resting place" or "home base" known as the *tonic*. Often we recognize the tonic without knowing exactly how we arrive at this conclusion, though one can make this determination by statistical or algorithmic means, as well. As we consider examples of pitch distributions, think about algorithmic means or "formulas" for identifying the tonic and mode.

Often the notion of *scale* plays into our understanding. We will deal with two kinds or *modes* of scales: major and minor. Each of these scales is formed of an arrangement of half steps (difference of 1 midi pitch) and whole steps (difference of 2 midi pitches) as indicated in the following tables where W=whole step and H=half step.

Major Scale:		W		W		H		W		W		W		H	
	C		D		E		F		G		A		B		C

Minor Scale:		W		H		W		W		?		?		?	
	A		B		C		D		E		F		G		A
											(F#)		(G#)		

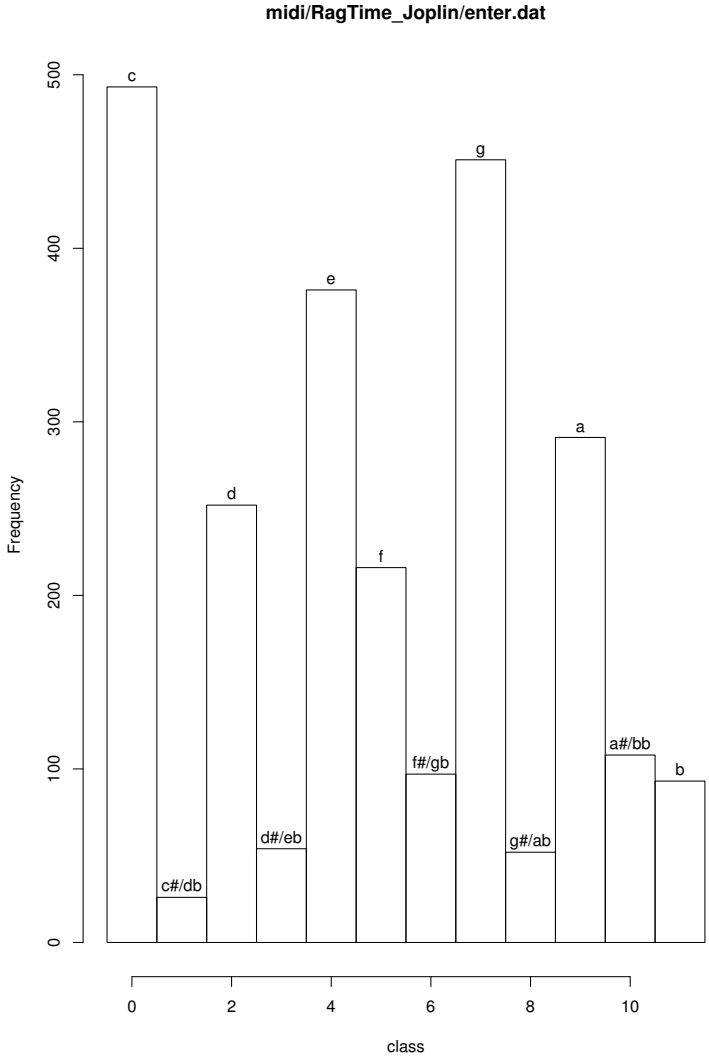


Figure 1.1: Pitch Histogram for the "Entertainer"

Observe the pitch distribution for the "Entertainer" in Figure 1.1. See how the notes in the C major scale occur in greater proportion than the other notes, with the greatest preference for the pitch class of C. This suggests the piece is in C Major, (which it is).

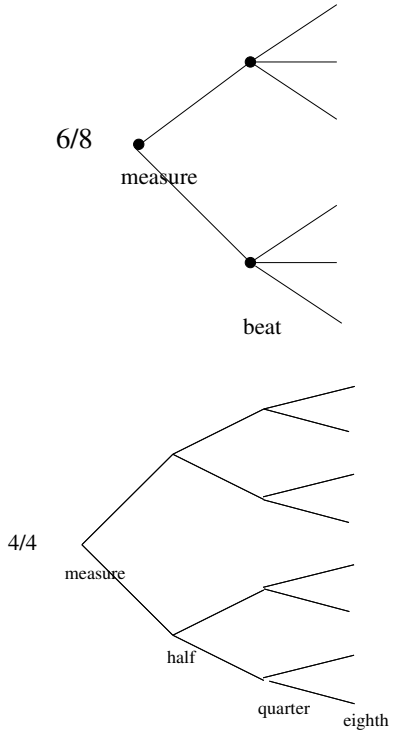
1.2 Rhythm (time_dist.r)

Much music either explicitly or implicitly has a meter. Western music notation captures some aspects of meter with a *time signature*. For instance

4/4 = "four counts to a measure; a quarter note gets 1 count"

6/8 = "six counts to a measure; eighth note gets 1 count"

Historically, time signatures are usually associated with subdivision tendencies. For example, 6/8 time has "duple" subdivision of measure with triple subdivision of beat, while 4/4 typically has duple subdivision of measure, half measure, and beat. These subdivision tendencies can be represented by tree diagrams:

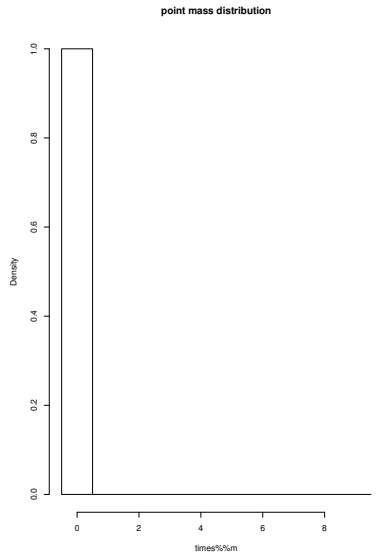


1.2.1 Finding Rhythmic Structure in Music Data

Consider the following simple experiment. Suppose we have a collection of regularly spaced onset times, say 0, 10, 20, 30, 40, ... Suppose we "mod out" (examine the remainder) using different numbers as our divisor:

$$
\begin{aligned}
&\text{mod}\mathbf{10} \\
0 \bmod 10 &= 0 \\
10 \bmod 10 &= 0 \\
20 \bmod 10 &= 0 \\
30 \bmod 10 &= 0 \\
40 \bmod 10 &= 0 \\
50 \bmod 10 &= 0 \\
&\ldots
\end{aligned}
$$

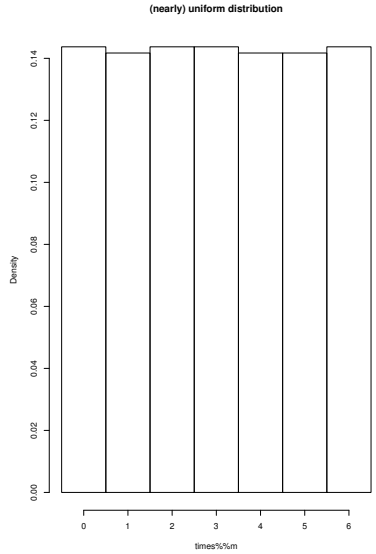
So the histogram of the onset times "mod 10" would be



Similarly if we mod out by 7, then

$$
\begin{aligned}
&\text{mod}\mathbf{7}\\
0 \bmod 7 &= 0\\
10 \bmod 7 &= 3\\
20 \bmod 7 &= 6\\
30 \bmod 7 &= 2\\
40 \bmod 7 &= 5\\
50 \bmod 7 &= 1\\
60 \bmod 7 &= 4\\
70 \bmod 7 &= 0\\
&\ldots
\end{aligned}
$$

the histogram would be



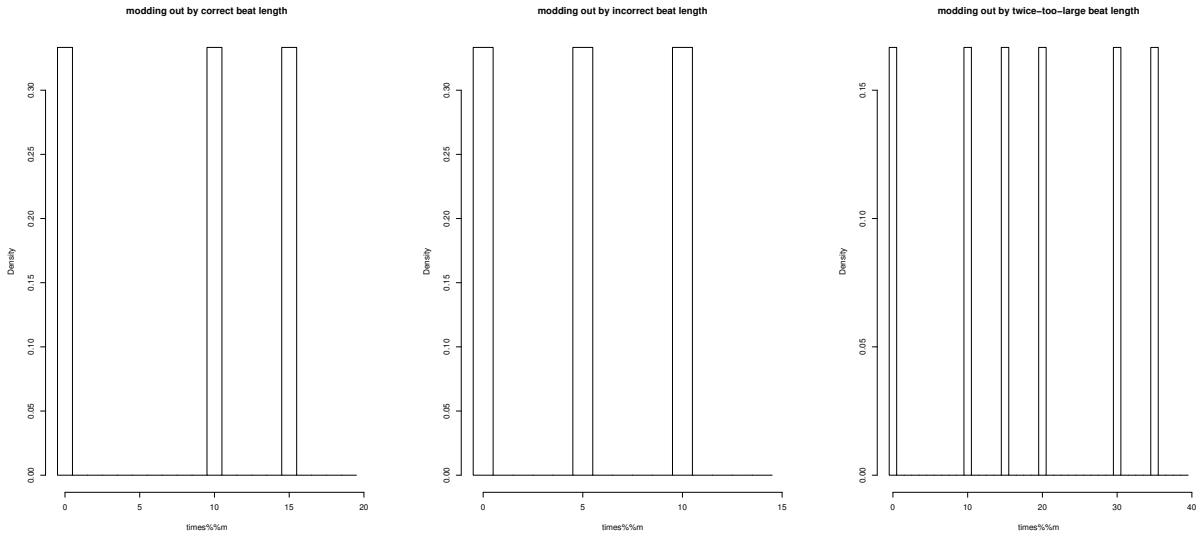
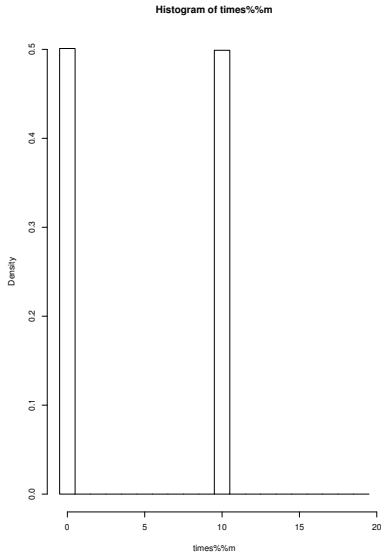


Figure 1.2: onset times of repeated eighth and two sixteenths when modded out by (left) the correct beat length, (middle) a incorrect beat length, and (right) a beat length that is twice too long.

Finally, if we mod out by 20, then

$$\begin{aligned} \text{mod}\mathbf{20} & \\ 0 \bmod 20 &= 0 \\ 10 \bmod 20 &= 10 \\ 20 \bmod 20 &= 0 \\ 30 \bmod 20 &= 10 \\ \ldots & \end{aligned}$$

the histogram is



For another example, consider the rhythm of an eighth and two sixteenths, repeated over and and as in the "William Tell Overture" or as Dmitri Shostakovitch is so fond of. In terms of onset times in midi

ticks, this would look like: 0, 10, 15, 20, 30, 35, 40, 50, 55, ... Figure 1.2 shows the histograms when we mod out by the "correct" beat length (20) with the incorrect choices of 15 and 40. Note the general pattern in which the correct beat length shows the beat pattern clearly, while an incorrect choice of beat length tends to produce a more uniform distribution. Note the special case of a beat length that is twice too long. In this situation we see the same beat pattern repeated exactly (or nearly so) twice in a row. In looking at these figures, remember the "circular" nature of the histograms, so that the right edge is really "right next to" the left edge.

1.2.2 Real Examples

The real world is messier, but the same ideas apply. In particular, plausible choices of important musical length (beats, measures, etc.) lead to more concentrated (less uniform) histograms that clearly depict rhythmic structure. For instance, see the example of the "Carolan's Welcome" which can be found in the "Irish" subdirectory of the midi files for the class. Figure 1.3 shows the histogram of onset times when we mod out by both the correct beat length and an incorrect beat length. Notice how one can see quite a bit of detail about the typical beat patterns by examining the correct beat histogram.

However, be cautioned that in real-world examples:

1. Don't have exact repetition of beat and measure patterns, but rather patterns that occur frequently.
2. Periodicity occurs at several levels, such as beat and measure, or maybe pairs of notated measures.

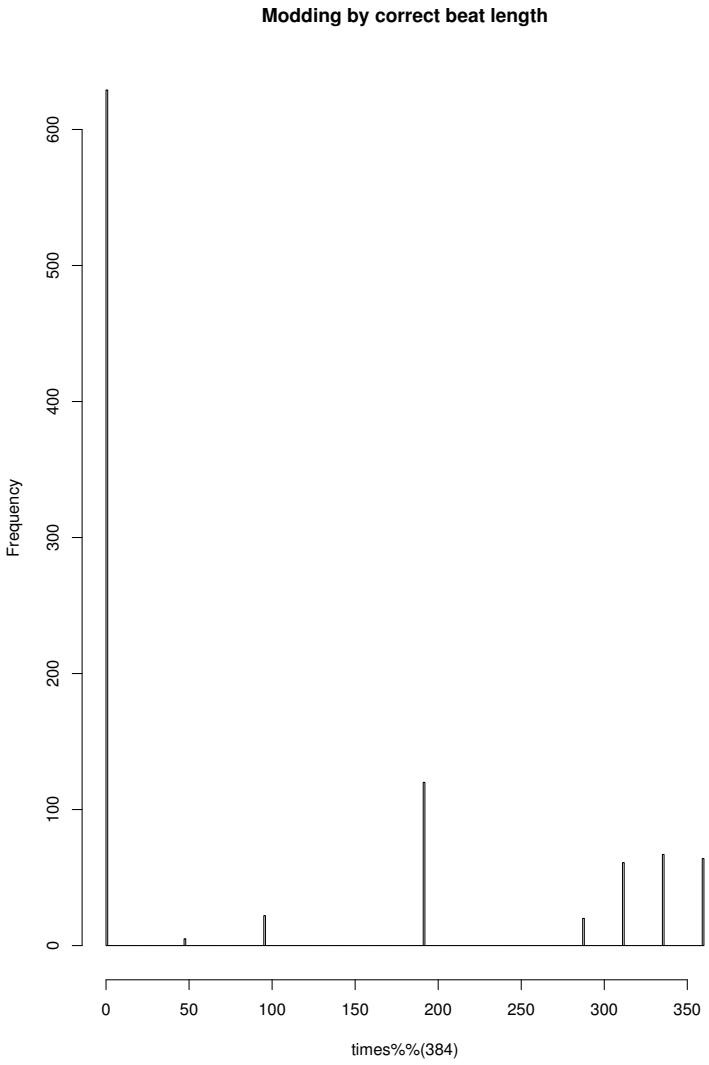
Figure 1.4 shows the onset times modded out by the correct measure length in the "Carolan's Welcome." In this example it is clear that the piece is in 3/4 time since given the triple subdivision of the measure and the duple subdivision of the beat.

1.2.3 Semi-Automatic Characterization of Rhythmic Structure

Here is an *algorithm* or "recipe" for finding the appropriate beat lengths and measure lengths while deducing rhythmic structure of the measure. Unlike a "real" algorithm, the steps must be executed with some reflection and subjective comparison to get the desired results.

1. Find the shortest unit of musical time having the property that nearly all note durations are multiples of this time unit. Sometimes this unit is referred to as the *tatum* (= temporal atom) and we will denote it by t. This will be a unit such that, when the durations are modded out by the unit, we get a highly concentrated histogram. Note that we do this with the *note durations* a.k.a. the inter-onset times (IOIs) and not the actual onset times.
2. Assuming only groupings of 2 or 3, mod out by $2 \times t$ and $3 \times t$. Determine the "better" grouping as the more "patterned" and less "uniform" histogram. Let $f_1 = 2$ or 3, whichever is better. To find the next grouping, repeat modding out now with $2 \times f_1 \times t$ and $3 \times f_1 \times t$ and let f_2 be the best of these choices.
3. Continue in this manner until additional grouping factors of 2 and 3 give nearly identical copies of the same histogram. At this point stop, since we are no longer finding meaningful rhythmic structure.

To interpret the results, suppose we end up with the collection of factors: $2 \times 2 \times 3 \times t$. This would be characteristic of a time signature such as 12/8 that has two levels of duple grouping followed by a level of triple grouping. Of course, given the way some music can be reasonably described by several choices of time signature, we cannot say for sure what would be the notated choice. However, we do have an understanding of the rhythmic structure of the piece.



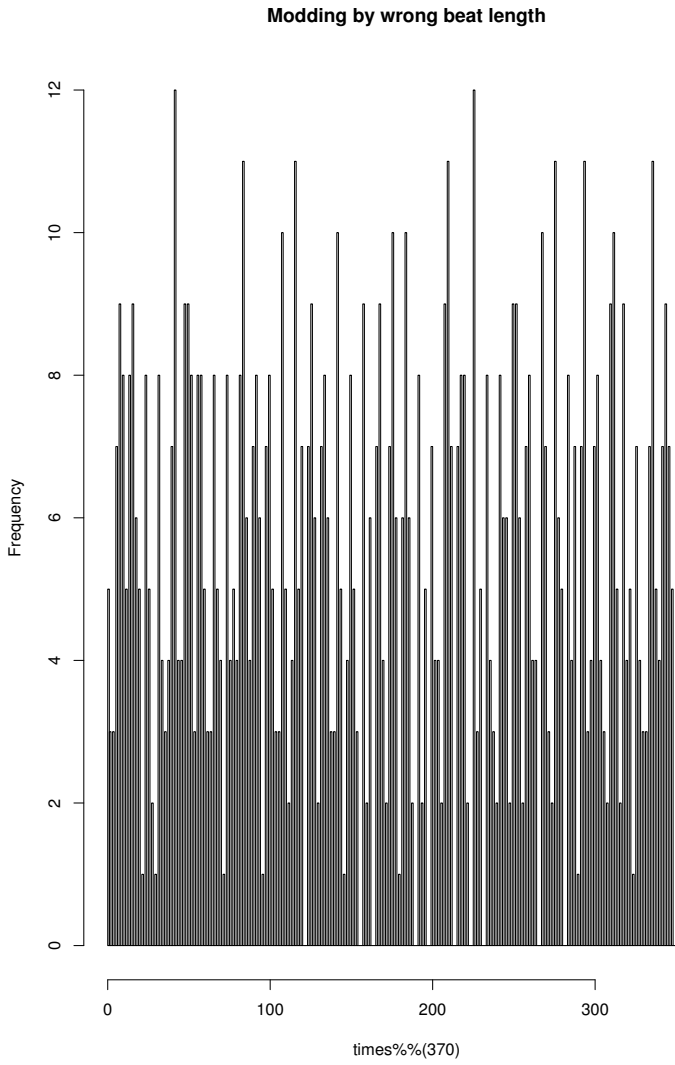


Figure 1.3: onset times when modded out by correct and incorrect beat lengths. The correct beat length shows that most onsets lie on the beat, while there is occasional subdivision into eights, and less frequent subdivision into 16ths. Also visible is the arpeggiated pickup to the beat

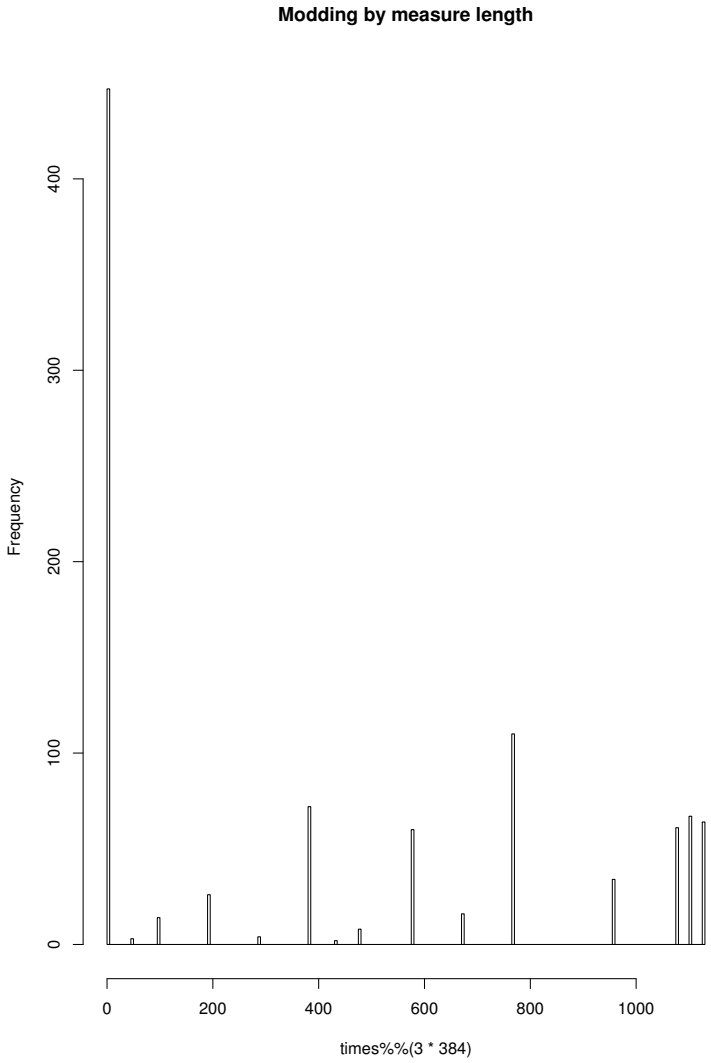


Figure 1.4: Onset times of "Carolan's Welcome" when modded out by correct measure length

1.3 Expressive Timing (onset.r)

The class web site has directory called **ost** containing pairs of files with the names xxx.wav and xxx.ost where "xxx" identifies the piece. The .wav files are, of course, actual recordings that you can play with essentially any media player. The .ost files (which stands for *onset time*) give information about the actual timing of the performance. In particular, each note of the piece is expressed as a row in the data file where the first three elements describe the starting time of the note in *musical units* while the last entry gives the onset time of the note in *real time* units. Real time is expressed in terms of "frames" where a frame lasts for 256/8000 seconds. Thus there are about 31, or more precisely 8000/256, frames per second. The first three entries of a row give

1. the measure number
2. the numerator of the measure position
3. the denominator of the measure position

So, for instance, if an .ost file begins with

```
1   7   8   131
2   0   1   202
.   .   .    .
.   .   .    .
.   .   .    .
```

this would mean the first note of the piece begins 7 eighth notes into the measure (the eighth "pickup" to the measure) and it was played at 131*256/8000 seconds. Similarly, the 2nd note begins at the start of the 2nd measure and was played at 202*256/8000 seconds.

You should be careful *not* to interpret the numerator and denominator given by the middle two columns as representing the fraction of the measure that has elapsed. For instance, in 3/4 time steady eighth notes would produce measure positions: 0/1, 1/8, 1/4, 3/8, 1/2, 5/8. Clearly at the 1/8 position 1/6 of the measure has elapsed. However, in 4/4 time (or any n/n time) the measure position is the same as the fraction of the measure that has elapsed.

Figure 1.5 shows the onset times for a performance of the opening of the 2nd movement of the famous Rachmaninov 2nd piano concerto, plotted in various ways. The upper left panel shows the onset times (in seconds) for the first 70-or-so measures, plotted against musical time (in measures). The slope of the line is an indication of the local tempo, showing that there is a significant overall tempo change around bar 50. The upper right panel shows only the first 10 measures of the piano entrance with each measure position given a different color. From this image we can see slight deviations from the basic tempo showing the way in which expressive timing is used. The bottom left panel shows the same thing, but with only the downbeats (starts of each measure) given a different color. The bottom right panel shows the inter onset times (IOIs = times between consecutive onsets) with a different color given to each of the 12 different possible measure positions (each quarter note of 4/4 is subdivided at the triplet level). Note the somewhat regular stretching of musical time that happens leading up to the beginning of each measure. In this figure we show the trajectory of several measures with line segments. It is interesting to see if the timing shows a tendency for the player to group into 3 groups of 4 (as the pitches suggest), or 4 groups of 3 (as the time signature suggests).

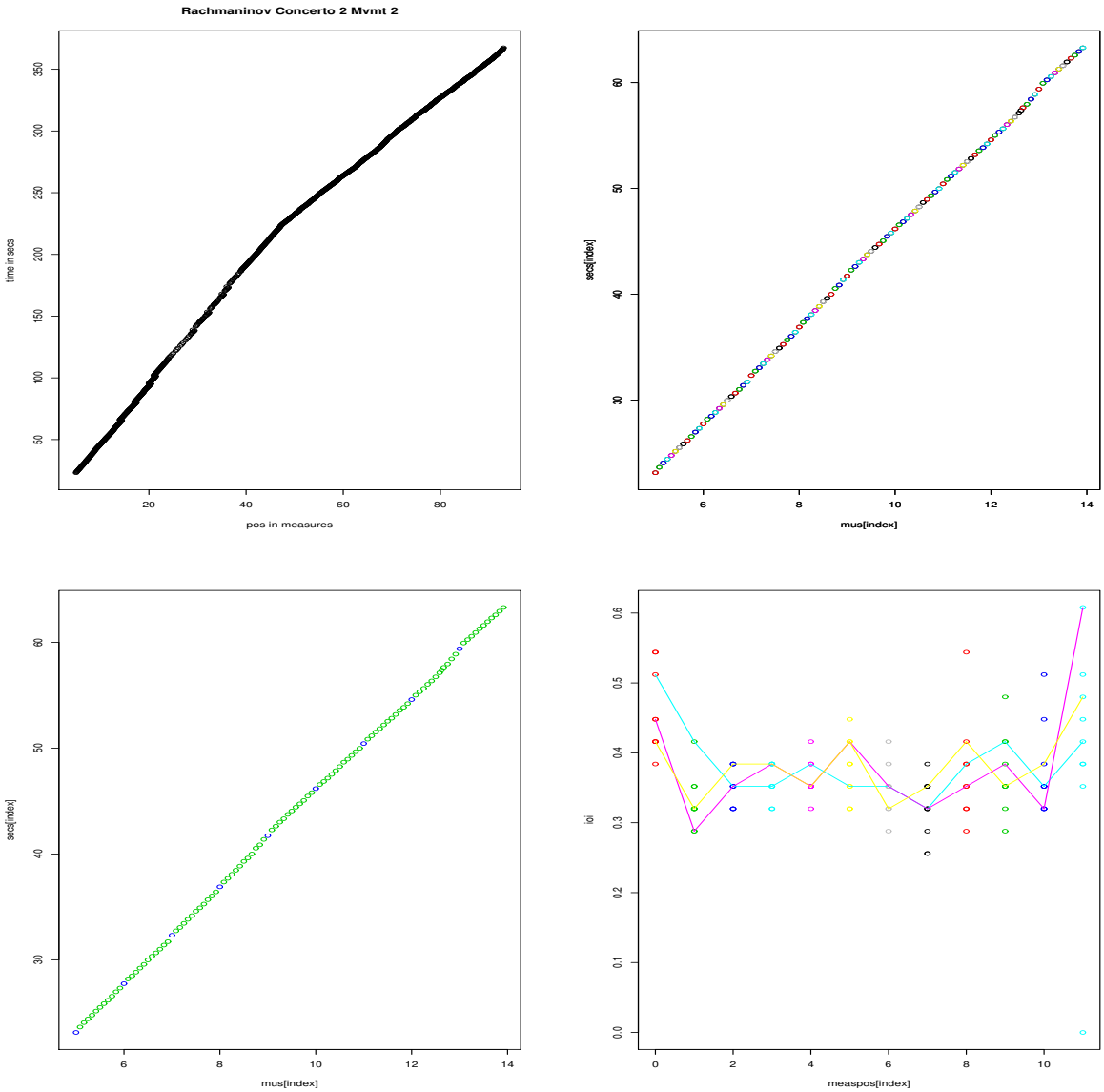


Figure 1.5: Various ways of plotting the onset times of the piano in the Rachmaninov 2nd Piano Concerto, Mvmt 2

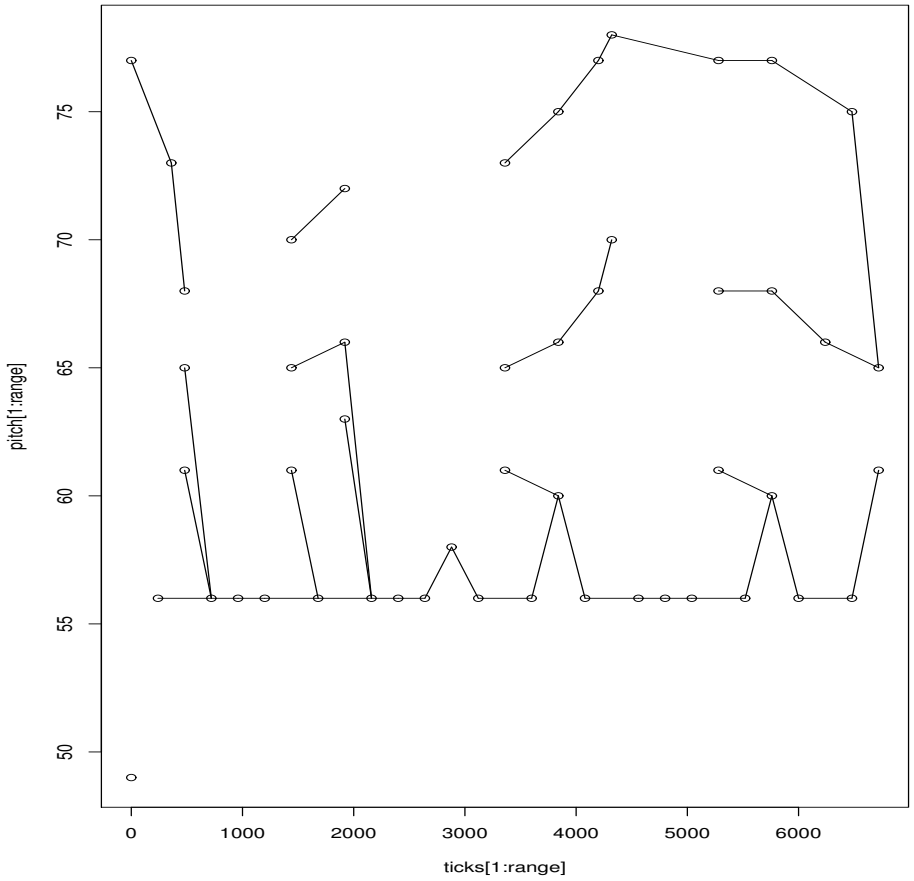


Figure 1.6: Piano Roll representation of Chopin "Raindrop" Prelude with an attempt at finding the voices automatically

1.4 Piano Roll Representation (piano_roll.r)

A "piano roll" representation plots each note of a piece of music with the *time* of the note in musical units (say beats) on the x axis and the pitch of the note (say in midi pitch) on the y axis. This kind of representation is straightforward and easy to interpret.

A *voice* is a monophonic (one-note-at-a-time) musical part. This notion is nearly always meaningful when we have music played by a collection of monophonic instruments or vocal parts. In fact, the notion of a voice is often meaningful in piano music as well. In the homework you are asked to produce a piano roll representation in which you (do your best to) group the notes into voices by drawing line segments. Figure 1.6 shows a piano roll representation of the Chopin "Raindrop" prelude. In this figure I have attempted to compute the voices automatically, as your homework asks, and have drawn them with line segments. Voices are useful for a number of music informatics applications. The main reason for this is it is easier to model and "process" a one-dimensional sequence, making individual voices easier to handle than polyphonic music.

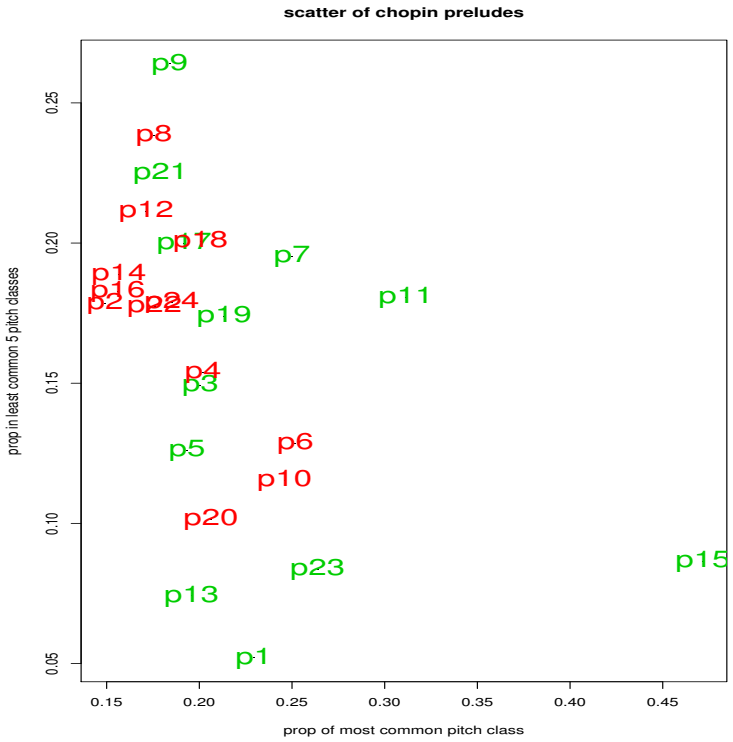


Figure 1.7: The Chopin Preludes with the frequency of the most common pitch class plotted against the frequency of the 5 least common pitch classes.

1.5 Various Scatter Plots

The previous plots are all ways of viewing single pieces of music where the objects we plot are notes. It can also be interesting to develop plots where the plotted objects are something other than notes. In the following scatter plots, each plotted point will represent a *piece* of music, though one can imagine plots where points represent composers, genres, pianos, orchestras, reviews, or many other aspects of music.

Figure 1.7 is computed using the **chopin_scatter.r** program on the class web page. This example computes two numbers from each of the 24 Chopin preludes, measuring aspects of the distribution of pitch classes. These two numbers are used as the x and y coordinates of a plotted point. The horizontal axis measures the proportion devoted to the most common pitch class — often the tonic. It is difficult to describe precisely what this means in musical terms, but pieces that modulate would likely have lower values for this "feature." The vertical feature measures the proportion devoted to the 5 least common pitch classes. For a piece that remains in a single key this would measure something like the proportion of out-of-scale notes. Informally we can view this is a measure of chromaticism.

"Outliers" are points that are "far away" from the majority. It is interesting to listen to the outliers of Figure 1.7 to see if the plots genuinely capture aspects that may be of musical interest. Note the "Raindrop" prelude shows up as an outlier, though not for the reason we had anticipated. Rather, the most frequent pitch class is the dominant note that represents the *ostinato* raindrops. However, the "highest" members of the plot do seem to represent the more chromatic preludes.

Figure 1.8 shows a collection of Joplin rags, Beatles songs, and Abba songs computed from the R file **rhythm_scatter.r**. For each song we have computed both the proportion of notes that lie *on the beat* as well as the proportion that lie exactly between two beats. Our goal in doing this was to differentiate

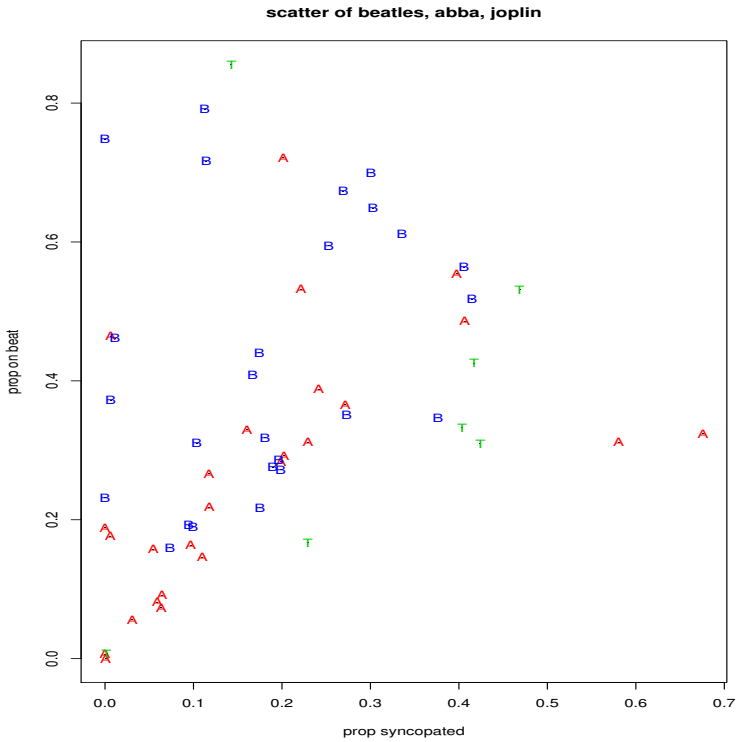


Figure 1.8: A scatter plot of a collection of Joplin Rags (T), Beatles songs (B), and Abba songs (A). For each song we have computed the proportion of notes that lie on the beat versus the proportion that are (exactly) between beats. Perhaps this latter attribute is a measure of the degree of syncopation.

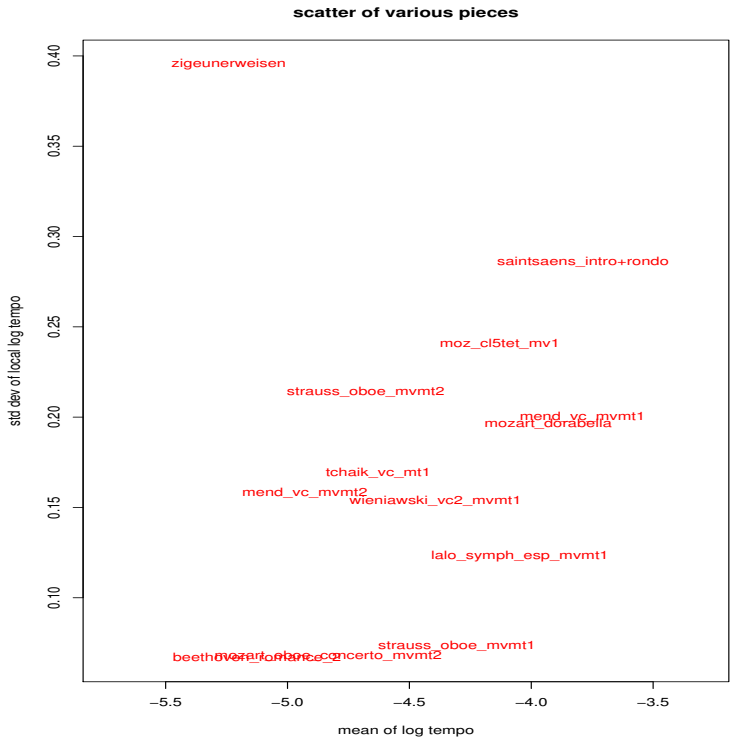


Figure 1.9: Plot showing the mean log tempo vs. the standard deviation of log tempo for a collection of actual performances.

between rather "square" pieces — which may be expected to have a large proportion of notes lying on the beat — with more syncopated pieces — which may be expected to have many offbeats. From the plot one can see that the Joplin rags do seem to be more syncopated than the Beatles songs, at least according to this rather informal measure. The prevalence of Abba songs near the origin (0,0) of the plot is due to inaccurate quantizing of some of the midi files, rather than any attribute of the music itself. The plot is quite effective at identifying the midi files that have these quantizing errors, since such a small fraction of notes lie either on the beat or on the after-beat. This is typical of exploratory data analysis, in that you often finds things other than what you thought you were looking for.

Figure 1.9 shows a collection of actual performances of various movements of concerti and other solo pieces using the **rubato_scatter.r** program. For each piece I have computed the tempo for each measure in *measures per second*. One can then take these tempi and, for each piece, plot the *mean* or average tempo versus the *standard deviation* of tempo. We will come back to the standard deviation more formally later in the course, but for now, this is simply a measurement of variation in data. You can think of the standard deviation informally as exactly what the name suggests — the average deviation or difference of a data point from the center. Thus the tempi would have a high standard deviation when the tempo changes a fair amount from measure to measure, while the standard deviation would be 0 of the tempo were completely constant.

When one plots the tempo standard deviation versus the mean tempo, the standard deviation grows with increasing tempo. This is *not* because faster pieces are more rubato than slower ones, but rather is an artifact (unintended side effect) of the way we have measured tempo. For instance, if we looked at a number of measurements of tempo that were, say 120 bpm plus or minus 5 bpm we would see the same standard deviation as if we looked at measurements of 60 bpm plus or minus 5 bpm. But, the latter case

represents much greater variation in tempo, since 5 bpm represents a bigger *proportional* change of 60 than 120. Of course, the proportional change is the relevant quantity when measuring tempo change — this is why the measurements on the old fashioned "Seth Thomas" metronome are closer together for low tempi than for higher tempi.

Many measuring scenarios occur where the importance or relevance of a change is measured by the proportional increase or decrease. For instance, consider the price of a stock where the proportional change tells us how much money we made or lost. When plotting such data it is better to measure the *logarithm* of the actual values, since the same proportional change always gives the same difference in the log. That is, since $\log(a/b) = \log(a) - \log(b)$, then

$$\log(px) - \log(x) = \log(p) = \log(py) - \log(y)$$

That is the same proportional change gives the same difference in logarithms. Thus pieces with the same amount of *proportional* tempo variation would see about the same standard deviation of the *log* tempo.

In Figure 1.9 it is easy to find the fast and slow pieces by looking at the horizontal axis, while the vertical axis measures our proxy for rubato — std dev of log tempo. The relative rankings of rubato are not surprising considering the composers and the tempi. That is, with some exceptions, the classical era movements are less rubato and the slower movements are more so.

Chapter 2

Probability and Statistics Introduction and the Bag of Notes Model

As a companion to this presentation I recommend you read the first 6 chapters of the Grinstead and Snell text that is on the web page. I recommend reading the material gradually along with the presentation of the material in class.

2.1 Basic Notions of Probability and Statistics

Some experiments have random outcomes. We denote the possibles outcomes of an experiment as set $\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$. Ω is often called the *sample space*. We will only deal with experiments that have a finite number of outcomes. Examples are the following:

1. Flip 3 coins: $\Omega = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\}$ All of these outcomes are equally likely to occur.

2. Throw pair of dice. Possible dice totals are $\Omega = \{2, 3, \ldots, 12\}$. Of course, the possible outcomes are not equally likely.

3. Examine the pitch class of a "randomly" chosen note. $\Omega = \{c, c\sharp \ldots, b\}$

A *probability distribution*, $p(\omega)$ assigns probabilities to the possible outcomes $\omega \in \Omega$, such that

- $p(\omega) = 0 \Rightarrow \omega$cannot occur.
- $p(\omega) = 1 \Rightarrow \omega$is certain to occur.

If $0 < p(\omega) < 1$ then $p(\omega)$ is the fraction of times ω would occur if experiment is replicated over and over.
A probability distribution must have

1. $0 \leq p(\omega) \leq 1$
2. $\sum_\omega p(\omega) = 1$

An *event* is a subset of the sample space, $A \subseteq \Omega$. For events, A, we have

$$P(A) = \text{the probability of } A = \sum_{\omega \in A} p(\omega)$$

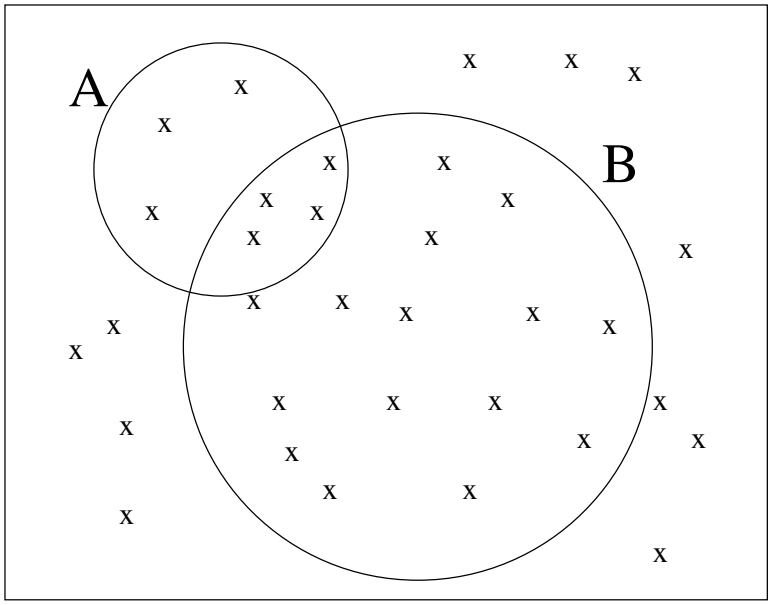


Figure 2.1: Pictorial description of conditional probability.

Example: Coin Flipping

Flip a coin 3 times and let A be the event that the *first* flip is H. Then

$$P(A) = \underbrace{p(HHH)}_{1/8} + \underbrace{p(HHT)}_{1/8} + \underbrace{p(HTH)}_{1/8} + \underbrace{p(HTT)}_{1/8} = 1/2$$

Example: Ω = Sum of Dice

Let the event $B = \text{sum} \leq 3$. Then $P(B) = p(2) + p(3) = 1/36 + 2/36 = 3/36$

Conditional Probability

If A, B are events $P(A|B)$ is the *conditional* probability of A occurring *given* that B has occurred. The definition of conditional probability is

$$P(A|B) = \frac{P(A \overbrace{\cap}^{\text{and}} B)}{P(B)} = \frac{P(A,B)}{P(B)}$$

The meaning of this is described in figure 2.1. If we know that B has occurred then our "entire world" is restricted to B. Knowing this, the probabilities in B must be recalibrated or scaled to sum to 1, since we *know* that B must occur. To do this, we simply divide the probabilities of the elements of B by $\sum_{\omega \in B} p(\omega)$. Thus $P(A|B)$ must be the sum of the probabilities that are in *both* A and B, but now with each probability divided by $\sum_{\omega \in B} p(\omega)$. This is exactly what the definition says.

Example

Let $A = \text{Dice sum is odd}$ and $B = \text{Dice sum} \leq 3$. Then

$$P(A|B) = \frac{P(A,B)}{P(B)} = \frac{P(\text{Dice} = 3)}{P(\text{Dice} \leq 3)} = \frac{2/36}{3/36} = 2/3$$

Independence

Two events, A, B, are *independent* if one occurring has no influence on the other occurring. Said more precisely, that is

$$P(A|B) = P(A)$$

As a consequence, if A, B being independent then

$$P(A) = P(A|B) = \frac{P(A,B)}{P(B)} \Longrightarrow P(A,B) = P(A)P(B)$$

This latter relation, $P(A,B) = P(A)P(B)$ is usually taken as the definition of independence. In practice, independence is usually an *assumption* one makes, rather than something that is verified from probabilities.

Example

Suppose A is the event that the dog gets a walk and B is the event that the dog steals food. Suppose that $P(A) = .95$ and $P(B) = .05$ and that the two events are *independent*. Then

$$P(A,B) = P(A)P(B) = (.95)(.05) = .0475.$$

Random Sample

A *random sample* is a collection of independent *identically distributed* (having same probability distribution) random experiments.

Example

100 Bloomington residents are chosen from the entire population of Bloomington residents by putting their names in a jar, selecting one at random, replacing it and mixing thorougly, and continuting this 100 times. We only observe the political affiliation of individual (D = Democrat, R = Republican).

This is a random sample since the trials have no affect on one another and hence independent. Also, each time we have the sample probability of drawing a R or D, so the trials are identically distributed.

Suppose the result is DDRRR .... Suppose that $P(D)$ is the proportion of Democrats and $P(R)$ the proportion of Republicans. Then

$$P(DDRRR\ldots) = P(D)P(D)P(R)P(R)P(R)\ldots = P(D)^{\#D}P(R)^{\#R}$$

Maximum Likelihood Estimation

Say want to estimate the proportion or probability of Democrats or Republicans from a random sample. (Assume these are the only possibilities). The *maximum likelihood estimate* (MLE) chooses the probability giving the highest likelihood to the data — that is, the best explanation of the data.

Let $p = P(D)$ (so $1-p = P(R)$) and suppose we sample N voters. The usual statistical notation writes the *estimate* of some quantity as that same quantity with a "circumflex" or "hat" over it. So we will write $\hat{p}$ for our estimate of p and refer to this as "p hat." The obvious estimate of p using our data would be

$$\hat{p} = \frac{\#D}{N}$$

What about the MLE for p?

We know that $P(\text{data}) = p^{\#D}(1-p)^{\#R}$. The MLE maximizes $P(\text{Data})$ as a function of p. That is

$$\begin{aligned}\hat{p} &= \arg\max_p p^{\#D}(1-p)^{\#R} \\ &= \arg\max_p \log(p^{\#D}(1-p)^{\#R}) \\ &= \arg\max_p \#D\log(p) + \#R\log(1-p)\end{aligned}$$

To maximize this, we set the derivative equal to 0 and solve for p to get $\hat{p}$. That is,

$$\frac{\#D}{\hat{p}} - \frac{\#R}{1-\hat{p}} = 0$$

Substituting $N - \#D$ for $\#R$ and solving for $\hat{p}$ gives

$$\hat{p} = \frac{\#D}{N}$$

This is good news since the MLE does the obvious thing in a simple case where it is easy to see what the obvious thing is. This result holds more generally, as follows.

Suppose we have an experiment taking values in $\Omega = \{\omega_1, \ldots, \omega_K\}$. Perform the experiment N times *independently*. By definition, the MLE estimate, $\hat{p}(\omega_1), \ldots \hat{p}(\omega_K)$ chooses the $p(\omega_1)\ldots p(\omega_K)$ such that

$$P(\text{data}) = p(\omega_1)^{\#\omega_1} p(\omega_2)^{\#\omega_2} \ldots p(\omega_K)^{\#\omega_K}$$

is maximized. While the calculations to maximize this quantity are somewhat more complex, they yield the same result:

$$\begin{aligned}\hat{p}(\omega_1) &= \frac{\#\omega_1}{N} \\ \hat{p}(\omega_2) &= \frac{\#\omega_2}{N} \\ \vdots &\quad \vdots \\ \hat{p}(\omega_K) &= \frac{\#\omega_K}{N}\end{aligned}$$

The R examples **mle.r** demonstrates the simple computation of the MLE in some simple examples.

Music Example

We have observed the pitch class histogram (the # of each pitch class) for a number of different pieces already. We write the histogram as $h(\omega)$ where $h(\omega)$ is the number of notes of pitch class ω with $\omega \in \{c, c\#, \ldots, b\}$. If N is the number of notes in the piece, then $h(\omega)/N$ is the *proportion* of notes of pitch class ω. That is $\hat{p}(\omega) = h(w)/N$ is our maximum likelihood estimate of the true probability of ω, $p(\omega)$.

Classification

A classifier takes data, x, and tries to categorize it according to "classes" $C_1, C_2, \ldots, C_L$. For example,

1. An email message can be viewed as a string of characters $x = x_1, x_2, \ldots, x_N$ The email could be classified into two classes: **Spam** and **Not Spam**.

2. The image of a handwritten letter or number could be represented as an array of grey-level values:

$$x = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{NN} \end{pmatrix}$$

x could be classified according to classes **0,1,. . . ,9,A,B,. . . ,Z**.

There are a great many techniques for performing classification, though statistical classifiers are among the most successful approaches for many problems.

Maximum Likelihood Classification

Suppose we have observed data $x = (x_1, \dots, x_N)$ which we would like to classify according to categories $C_1, C_2, \dots C_L$. Imagine that we have a probability model for the data under each class: $p_{C_1}(x), p_{C_2}(x), \dots, p_{C_L}(x)$. Usually this would be *learned* (estimated) using "training" data in which the classes of each training example are *known*. The MLE estimate chooses the class that gives that data the greatest possible likelihood, as it is the *best explanation* of the data, x. That is,

$$\underbrace{\hat{c}(x)}_{\text{Estimated Class}} = \arg\max_l p_{C_l}(x)$$

Example: Key Estimation (key_est.r)

Suppose the possible keys are C, C#, . . . , B (the major keys) and c, c#, . . . , b (the minor keys). For each key, suppose we have learned a pitch class distribution simply by observing the proportions of the various pitch classes for music *known* to be in the key. For example, our C distribution may look something like Figure 2.2. If $x = (x_1, \dots, x_N)$ is a vector of pitch classes taken from the notes of a piece of music, we can compute the probability of x, $p(x)$, under the assumption that the piece was in C Major by assuming that x is a random sample from the C major distribution. In that case:

$$\begin{aligned} p_C(x) &= p_C(x_1)p_C(x_2)\dots p_C(x_N) \\ &= p_C(0)^{\#x=0}p_C(1)^{\#x=1}\dots p_C(11)^{\#x=11} \end{aligned}$$

This sort of model is called a "bag of notes" model since the order of the notes is not considered important in the probability model. The MLE approach would then proceed by estimating a pitch class distribution for each of the 24 possible keys and then choosing the key giving the greatest probability to the data, under our random sample model.

Some comments on the implementation of this idea:

1. Do we really need to estimate the key distributions, p_k, separately for each possible key? It is reasonable to assume that the probability of the *tonic* is the same for all of the major keys. A similar assumption would apply to all of the 11 remaining chromatic pitches, for that matter. Thus we could estimate a single major and minor distribution and obtain all 24 models by *translating* or *transposing* the appropriate minor or major model to the desired key. To be more specific, if p_C is the C Major distribution, then the *D Major* distribution would be given by $p_D(i) = p_C((i-2) \bmod 12)$.

2. Maximizing $p_k(x)$ over the possible keys, k, is the same as maximizing $\log(p_k(x))$ over the possible keys. This comes from the fact that the log function is increasing as shown if Figure 2.3. When the

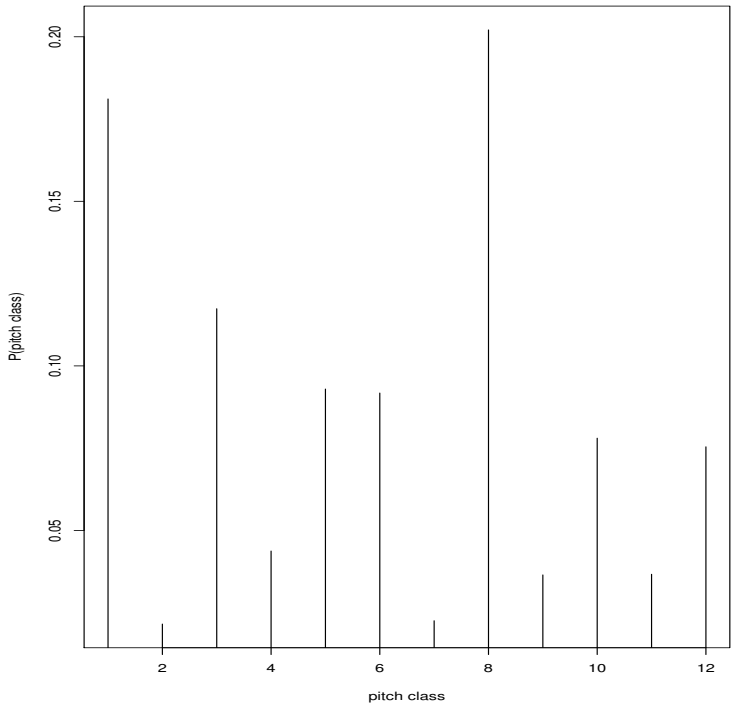


Figure 2.2: The empirical probability distribution for the key of C major

object we seek to maximize is a product of many factors, often it is easier to maximize the log of the object, which becomes a sum. So, in our case,

$$\begin{aligned} \log(p_k(x)) &= \log(p_k(0)^{\#x=0} p_k(1)^{\#x=1} \ldots, p_k(11)^{\#x=11}) \\ &= (\#x = 0)\log(p_k(0)) + (\#x = 1)\log(p_k(1)) + \ldots, (\#x = 11)\log(p_k(11)) \end{aligned}$$

The R program **key_est.r** implements this idea for key estimation assuming we are only dealing with *major* keys. In this example we

1. Train a C major model by transposing several pieces to C major and observing the proportion of each pitch class over the entire collection of pieces.

2. Estimate the key of a particular piece in major mode by computing all 12 translations of the C major distribution and finding which one gives the maximum log likelihood to the data.

3. While there may be better ways to do this, we can perform a simple-minded harmonic analysis by applying the approach to estimate the key of each *measure* of a piece. The key estimates could be taken to be *chord* labels for the different measures.

Random Variables

A *random variable* is a quantity measured from a random experiment. The *distribution*, $p(x)$, of a random variable gives the probability of the possible outcomes, x. Typical notation uses capital letters for random variables and small outcomes for their outcomes. So $X = 5$ denotes the event that the random variable, X, takes on the value 5, and $P(X = 5)$ is the probability of this event. Thus $p(x) = P(X = x)$.

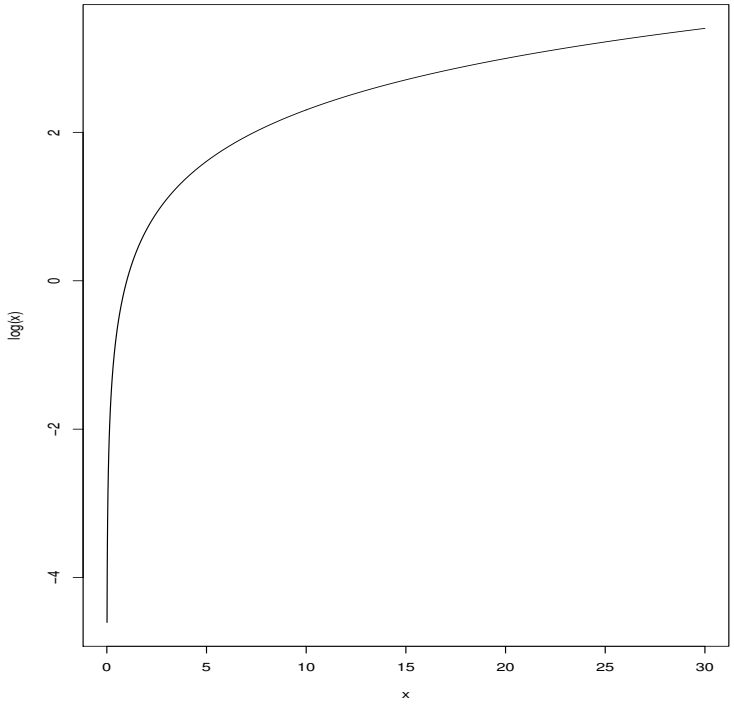


Figure 2.3: The log function is increasing. This means that $x > y \Leftrightarrow \log(x) > \log(y)$. So one can always maximize the log of a function rather than the function itself to find the maximizing value.

For example, flip a coin 3 times and let $X = \#$ H's. Then X can be tabulated as follows:

Ω	HHH	HHT	HTH	HTT	THH	THT	TTH	TTT
X	3	2	2	1	2	1	1	0

In this case we have

$$\begin{aligned} P(X=3) &= p(3) = 1/8 \\ P(X=2) &= p(2) = 3/8 \\ P(X=1) &= p(1) = 3/8 \\ P(X=0) &= p(0) = 1/8 \end{aligned}$$

Expectation

A random variable, X, has *expectation*, $E(X)$, defined as

$$E(X) = \sum_x xp(x) = \text{the "average" value of } X$$

In the above example we have $E(X) = 0(1/8) + 1(3/8) + 2(3/8) = 3(1/8) = 1.5$. This is, on average, the number of heads we would get for each trial of the experiment, since each of the 3 single flips will yield .5 H's, on average.

Variance

Let X be a random variable with expectation $E(X) = \mu$ (μ is for *mean* which is another name for expectation). The *variance* of X, $V(X)$, is defined as

$$V(X) = \sum_x (x-\mu)^2 p(x) = E(X-\mu)^2$$

In the above example $V(X) = (0-1.5)^2(1/8) + (1-1.5)^2(3/8) + (2-1.5)^2(3/8) + (3-1.5)^2(1/8) = 3/4$ Since the variance is the average squared difference from the mean, μ, variance is a measure of the "spread" of the distribution. Typically one writes σ^2 for the variance of a random variable.

The *standard deviation* is the (positive) square root of the variance. Since the variance measures the average squared difference from the mean, then the standard deviation can be thought of informally as the average difference from the mean. Normally one writes σ for the standard deviation of a random variable. The standard deviation is useful as the basic unit of variation for a random variable. Thus, there are many results that describe probabilities of events such as the random lying within within so many standard deviations of the mean. For instance, for "normal" random variables (a term to be defined later) $P(|X-\mu| < 2\sigma) \approx .95$ and for *any* random variable $P(|X-\mu| < k\sigma) \geq 1 - 1/k^2$. Results such as these clarify what the standard deviation measures and make its meaning more intuitive.

Entropy

Entropy is a number that measures the amount of "surprise" or uncertainty in a distribution. For instance, consider the two probability distributions described by Figure 2.4. In the left panel, the distribution

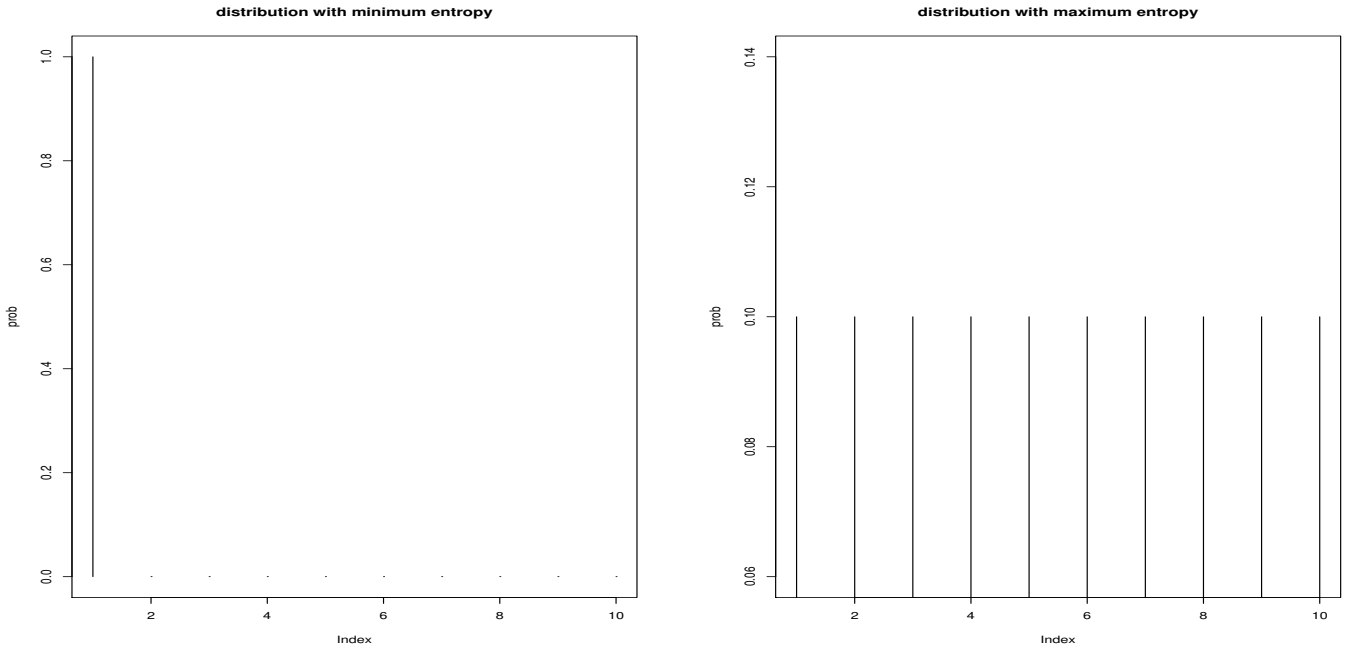


Figure 2.4: For the two distributions picture, the left panel is the "point mass" distribution which is the one that gives the minimal entropy. In contrast, the "uniform" distribution while the right gives the maximal entropy over all other distributions on the same number of values. Explicitly, if X has a point mass distribution then $H(X) = \frac{\log_2 1}{1} = 0$. If X has a uniform distribution on n values, then $H(X) = \frac{n(\log_2 n)}{n} = \log_2(n)$.

concentrates all of its weight on the first outcome. Sometimes such a distribution is called a "point mass" since all of the probability is concentrated on a single outcome. If a random experiment with such a distribution were performed, there would be no uncertainty as to the outcome before the experiment. In this case the entropy turns out to be 0. In the case of the right panel we have a "uniform" distribution in which all outcomes are equally likely. Before an experiment with such a distribution, we would be completely uncertain about the outcomes, since all possibilities are equally likely. This case turns out to have maximal entropy. Other distributions fall between these two extremes.

For a random variable, X, with distribution $p(x)$, the *entropy* of X, $H(X)$, is defined by

$$H(X) = \sum_x \log_2(\frac{1}{p(x)})p(x)$$

In this definition, $\log_2(y)$ is the number we need to raise 2 to, to get y — that is, $\log_2(2^q) = q$. If we think of $p(X)$ as the random variable that reports the *probability* of the outcome X, rather than the value of X itself, then the entropy can be written as $H(X) = E(\log_2(\frac{1}{p(X)}))$. This interpretation becomes more intuitive after we examine the quantity $1/p(x)$.

Consider the probabilities 1, 1/2, 1/4, 1/8, ... These can be tabulated as

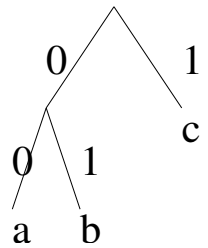
p	1	1/2	1/4	1/8	1/16
$p = 2^q$	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
$\log_2(1/p)$	0	1	2	3	4

We can see from this table that $\log_2(1/p(x))$ is a measure of the rareness or surprise of outcome x, so $H(X) = E(\log_2(1/p(X)))$ is the average surprise.

Entropy and Huffman Coding

Suppose the symbols $\{a, b, c, \ldots\}$ appear as a random sample from some probability distribution. We want to develop a code in which we represent each symbol as a string of 0's and 1's. Such a code is called a *binary code* since we use *two* symbols. We imagine that we are going to send the binary codes for our symbols to someone else, and require that our code can be *decoded* by the receiver. For the code to be decodable, the symbol codes must be the *leaves* of a *binary tree.*

For instance, suppose we have the symbols $\{a, b, c\}$. We can represent these symbols as $a = 00, b = 01, c = 1$ corresponding to the tree:



Note that binary string 01100101 can be decoded into "bcacb" by beginning at the *root* of the tree and descending according to the binary symbols. During the process, we print out the associated letter every time we reach a *leaf* node of the tree and return to the root.

Now we imagine that the symbols $\{a, b, c, \ldots\}$ from our alphabet are generated according to some probability distribution and we want to choose our binary coding of the symbols to give the minimum

number of binary digits (*bits*) sent, on average. Thus, it would make sense to use short codes for frequent symbols and longer codes for rarer symbols. In terms of the language of probability, we wish to choose the code that minimizes $E(\#$ bits per symbol) For instance, suppose we use the coding of $\{a, b, c\}$ given above with $p(a) = 1/2$ and $p(b) = p(c) = 1/4$ as tabulated as

p	1/2	1/4	1/4
symbol	a	b	c
code	00	01	1

In this case, $E(\#$ bits per symbol$) = 2 \times 1/2 + 2 \times 1/4 + 1 \times 1/4 = 7/4$.

An interesting result from the coding theory tells us that no matter what code we choose,

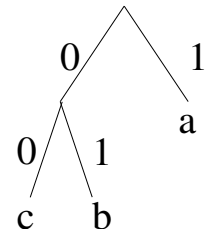
$$H \leq E(\#\text{ bits per symbol})$$

where H is the entropy of the distribution. This is useful to know since we know we have achieved the best possible coding if the expected bits per symbol *equals* the entropy.

For the example above, the entropy of the distribution is

$$\begin{aligned} H &= 1/2 \log_2 2 + 1/4 \log_2 4 + 1/4 \log_2 4 \\ &= 1/2(1) + 1/4(2) + 1/4(2) = 3/2. \end{aligned}$$

Since the entropy is less than the expected bits per symbol, we have not found the ideal coding yet. However, it is easy to see that

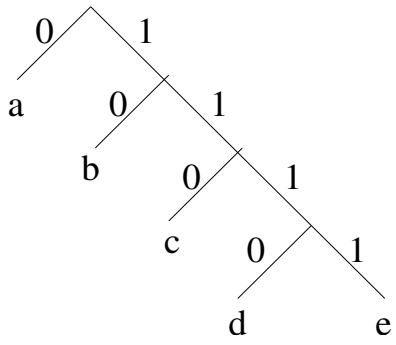


gives $E(\#$ bits per symbol$) = 2(1/4) + 2(1/4) + 1(1/2) = 3/2 = H$ so this must be the optimal coding.

Example

Consider the following probability distribution on the symbols $\{a, b, c, d, e\}$ with the proposed binary coding of the symbols. The associated tree for the coding is also shown.

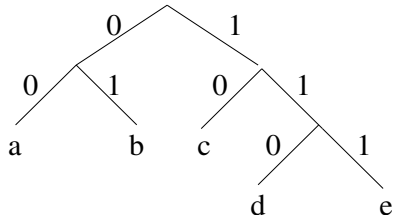
symbol	a	b	c	d	e
p	1/2	1/4	1/8	1/16	1/16
code	0	10	110	1110	1111
length	1	2	3	4	4
$\log_2(1/p)$	1	2	3	4	4



For this case it is easy to see from the table that the proposed coding is optimal. This is because E(bits per symbol) is the average of the **length** row (weighted by the p row) while the entropy, H is the average of the $\log_2(1/p)$ row (again, weighted by the p row). Since the last two lines of the table are identical, the two averages: E(bits per symbol) and H are equal, hence the coding is optimal. That is

$$\begin{aligned} H &= E(\#\text{ bits per symbol}) \\ &= 1(1/2) + 2(1/4) + 3(1/8) + 4(1/16) + 4(1/16) = 1 + 7/8 \end{aligned}$$

Just for illustrative purposes, let's try a more evenly balanced code that aims for codes of more even length. Such a code is depicted below.



For this coding

$$E(\#\text{ bits per symbol}) = 2(1/2) + 2(1/4) + 2(1/8) + 3(1/16) + 3(1/16) = 2 + 1/8$$

which is clearly worse than the one proposed before.

Entropy and Rhythm (entropy.r)

Have observed an interesting property of rhythmic units such as *beat* and *measure*:

1. Modding out onset times (observing the remainder when dividing onset times by) a meaningful musical time unit gives "concentrated" distribution (low entropy).
2. Modding out my meaningless time unit leads to more uniform distribution (high entropy).

Idea: Find beat (or other musical time unit) by looking for time unit that gives low entropy remainder distribution. The R program **entropy.r** plots the entropy when we mod out by all possibles choices ranging from $b/3 \ldots 3b$ where b is the true beat length.

In interpreting the plots generated by applying the **entropy.r** program to different pieces, we see a generally increasing trend as the time unit increases in length. This is easy to explain. While we won't actually *prove* this, the entropy, H, has the property:

$$H(X \bmod a) \leq H(X \bmod ab)$$

where $a, b = 1, 2, 3 \ldots,$. In words this says that if we have two time units, u_1 and u_2 with u_2 a *multiple* of u_1, then u_2 always leads to an entropy that is at least as big as u_1 when modding out the values X. This explains the increasing trend in entropy that the R program produces. We can identify the "correct" (ie musical meaningful) times units as units that give low entropies when compared to their neighbors. Visually, these correct time units tend to have entropies that lie "below" the curve" generated by the entire collection of relatively low entropies. While this is not a foolproof rule, it does provide a general guideline that we will exploit in the coming discussion. I strongly doubt there is any foolproof method of determining rhythmic structure automatically.

Bayes' Rule

Suppose we have "states of nature" (Classes) $C_1, \ldots, C_L$. We have a "prior" model, $p(C_l)$, $l = 1, \ldots, L$ giving the *a priori* (before any data observed) probability of the various classes. For instance, if we are trying to identify an instrument that plays in a piece heard on the local radio station, before we listen to any sound we believe the digeridoo is less likely than electric guitar. Thus, our *a priori* probability for the digeridoo should be less than the guitar. Of course, this all may change after we listen to some audio data, but the prior information should figure into any classifications we make. For instance, if both the digeridoo and the electric guitar constitute comparable explanations of the audio, our bias should be with the more likely instrument *a priori*. Bayes' rule formalizes this idea.

Suppose, in addition to the prior model, $p(C_l)$, we also have a "data model" giving the probability of our observations, x, *given the class*, $p(x|C_l)$ $l = 1, \ldots, L$. Having observed the data, x, what are the *posterior* probabilities of the classes $C_1, \ldots, C_L$? Bayes' rule is the following calculation:

$$\begin{aligned} p(C_l|x) &= \frac{p(C_l, x)}{p(x)} \quad \text{defn. of conditional prob.} \\ &= \frac{p(x|C_l)p(C_l)}{\sum_k p(x|C_k)p(C_k)} \quad \text{writing out numer. and denom.} \end{aligned}$$

where we have used the intuitively plausible *law of total probability*:

$$p(x) = p(x|C_1)p(C_1) + p(x|C_2)p(C_2) + \ldots + p(x|C_L)p(C_L)$$

Example (Rare Disease)

A disease affects .1% of a population. A test is devised such that the probability of both *false positive* and *false negative* is .01. If a randomly chosen person tests positive for the disease, what is the probability the person actually has the disease?

Let D be the event that the person has the disease and write $\bar{D}$ for the complementary event of not having the disease. Let $+$ and $-$ be the events of testing positive and negative. We have:

$$\begin{aligned} p(D) &= .001 \\ p(\bar{D}) &= .999 \\ p(+|\bar{D}) &= .01 \\ p(-|D) &= .01 \end{aligned}$$

Using Bayes' rule we can calculate the probability the person has the disease:

$$\begin{aligned} p(D|+) &= \frac{p(+|D)p(D)}{p(+|D)p(D) + p(+|\bar{D})p(\bar{D})} \\ &= \frac{(.99)(.001)}{(.99)(.001) + .(01)(.999)} \\ &\approx \frac{x}{x + 10x} \\ &= 1/11 \end{aligned}$$

It is quite unlikely the person who tested positive actually has the disease. The intuition for this result is given in the calculation that shows there are two ways of explaining the observed test result of +. Of these, the one in which the person doesn't have the disease is 10 times as likely as the one in which the person *does* have the disease.

Bayes Classifier

The *Bayes Classifier* chooses the class that maximizes the probability of correct classification. Suppose we are given $p(C_l)$ for $l = 1, \ldots, L$ and $p(x|C_l)$ where the possible classes are $\{C_l\}$ and the observable data is x. Bayes' rule tells us that

$$p(C_l|x) = \frac{p(C_l)p(x|C_l)}{\sum_k p(C_k)p(x|C_k)}$$

Thus the Bayes classifier choose the class l that maximizes this probability:

$$\hat{l} = \arg\max_l p(C_l|x) = \arg\max_l p(C_l)p(x|C_l)$$

Note that this is a minor (but important) variation on the maximum likelihood classifier which was $\hat{l} = \arg\max_l p(x|C_l)$. In the Bayes classifier we take into account both the degree that the classes explain the data, x, and the *a priori* likelihood of the classes. The **Rare Disease** example shows us the importance of doing this.

Continuous Probability Distributions

So far we have considered "discrete" random variables having only a finite (or countable) number of outcomes: $p(x)$ $x \in \{x_1, x_2 \ldots\}$. What if X can take *any* value? In this case we represent probability with a *density* function, $f(x)$. Now probabilities are represented as *areas* under the curve generated by $f(x)$. This is described in Figure 2.5 in which we have drawn a density function. For the interval (a, b) the probability that the random variable X, with density $f(x)$, lies in (a, b) is described by the *area* under the curve $f(x)$ between a and b. Or, symbolically,

$$p(a < X < b) = \int_a^b f(x)dx$$

In many cases one needs calculus to compute such probabilities, however, probabilities for many common distributions are built into the R language.

The Gaussian Distribution (normal.r)

A random variable X is *normal* (a.k.a. *Gaussian*)) with mean $E(X) = \mu$ and variance $E(X - \mu)^2 = \sigma^2$ if it has the density function

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

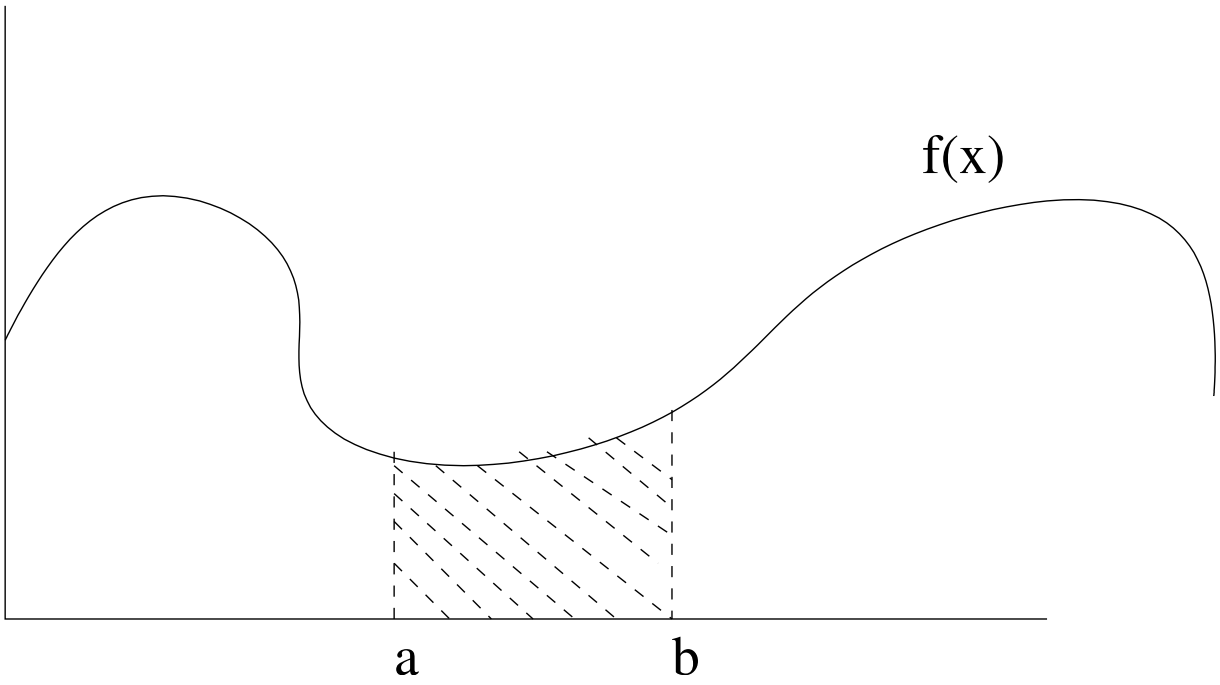
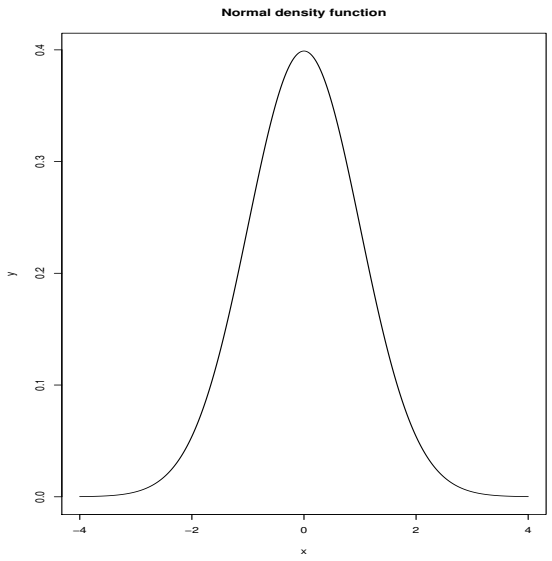


Figure 2.5: The density function $f(x)$ gives probabilities of intervals as areas (integrals).

This is the famous "bell-shaped" curve drawn below:



The highest value of the the density is at the mean μ and it is symmetric around this value. Roughly speaking, one can think of the standard deviation σ as the half width at half height.

Normal distributions are good for fitting many data distributions found in practice. Suppose we have a sample $x_1, \ldots x_n$ which we wish to model as a normal random sample. How do we get the *parameters* (μ and σ) for the distribution? One can show that the MLE estimates μ and σ^2 as the *empirical* mean and variance. That is:

1. Since μ is the mean or expected or average value of the distribution, take $\hat{\mu}$ to be the average value

of the *sample*:

$$\hat{\mu} = \frac{1}{n}\sum_i x_i$$

2. Since σ^2 is the average or expected squared distance from the mean μ, take $\hat{\sigma}^2$ to be the average squared distance from $\hat{\mu}$ over the sample:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_i (x_i - \hat{\mu})^2$$

Then we can estimate $\hat{\sigma} = \sqrt{\hat{\sigma}^2}$.

Building a Gaussian Classifier (simple_irish_class.r)

We have seen that a Bayes Classifier is composed of

1. $p(C_l)$ for $l = 1, \ldots, L$ (the class probabilities)
2. $p(x|C_l)$ for $l = 1, \ldots, L$ (the class-conditional data distributions)

Suppose we have training data $x_1, \ldots, x_n$ with *labels* $t_1, \ldots, t_n$ for the data points. The label t_i tell which class x_i comes from so $t_i \in \{1, \ldots, L\}$. To estimate the Gaussian classifier

1. Estimate $p(c_l)$ as the fraction of type l in the training data:

$$\hat{p}(C_l) = \frac{\#\{t_i = l\}}{n}$$

2. Estimate μ_l as the empirical mean of the l-labeled examples:

$$\hat{\mu}_l = \frac{\sum_{t_i=l} x_i}{\#\{t_i = l\}}$$

3. Estimate σ_l^2 as the empirical mean of the l-labeled examples:

$$\hat{\sigma}_l^2 = \frac{\sum_{t_i=l}(x_i - \hat{\mu}_l)^2}{\#\{t_i = l\}}$$

The Bayes classifier classifies a *new* point x as the class with the highest posterior probability. That is, x is classified as $\hat{C}(x)$ where

$$\begin{aligned}
\hat{C}(x) &= \arg\max_{C_l} p(C_l|x) \\
&= \arg\max_{C_l} \frac{p(x|C_l)p(C_l)}{p(x)} \\
&= \arg\max_{C_l} p(x|C_l)p(C_l) \\
&\approx \arg\max_{C_l} \hat{p}(C_l)\frac{1}{\sqrt{2\pi\hat{\sigma}_l^2}} e^{-\frac{1}{2}(\frac{x-\hat{\mu}_l}{\hat{\sigma}_l})^2} \\
&= \arg\max_{C_l} \log(\hat{p}(C_l)) - \frac{1}{2}(\frac{x-\hat{\mu}_l}{\hat{\sigma}_l})^2 - \frac{1}{2}\log(\hat{\sigma}_l^2)
\end{aligned}$$

Extension to Several Features (time_sig_feat.r)

Suppose our observable data, x, now measures *several* attributes, $x = (x_1, \ldots, x_J)$. The components of x are often referred to as *features*. It is often convenient to assume that, under each class, the features are *independent*. In this case, the usual terminology is that the features are *conditionally independent*. The conditional independence of the features allows us to write the data distributions $p(x|C_l)$ for $l = 1, \ldots, L$ as a product:

$$p(x|C_l) = p(x_1|C_l)p(x_2|C_l)\ldots p(x_J|C_l)$$

If, in addition, we also assume that the individual feature distributions, $p(x_j|C_l)$, are Gaussian with mean values μ_{jl} and variances σ^2_{jl}, then we have

$$p(x|C_l) = \prod_{j=1}^{J} \frac{1}{\sqrt{2\pi\sigma^2_{jl}}} e^{-\frac{1}{2}(\frac{x-\mu_{jl}}{\sigma_{jl}})^2}$$

Suppose that $\hat{\mu}_{jl}$ and $\hat{\sigma}^2_{jl}$ are the class conditional estimates of the Gaussian parameters obtained, exactly as in the preceding section, by looking at empirical averages for each feature under each class. That is, suppose that our sample vectors are $x^1, \ldots, x^n$ with class labels $t_1, \ldots, t_n$ (we use the *superscript* to index the different observations of the feature vector because the subscript is used to index the different feature values). Then we have

$$\begin{aligned} \hat{\mu}_{jl} &= \frac{\sum_{t_i=l} x^i_j}{\#\{t_i = l\}} \\ \hat{\sigma}^2_{jl} &= \frac{\sum_{t_i=l}(x^i_j - \hat{\mu}_{jl})^2}{\#\{t_i = l\}} \end{aligned}$$

Our Gaussian classifier then reduces, by taking logs, to

$$\begin{aligned} \hat{C}(x) &= \arg\max_l p(C_l) \prod_j p(x_j|C_l) \\ &= \arg\max_l \log(p(C_l)) - \frac{1}{2}\sum_{j=1}^{n}\left[\log(\sigma^2_{jl}) + \frac{(x_j - \mu_{jl})^2}{\sigma^2_{jl}}\right] \end{aligned}$$

Chapter 3

Music as Sequence

Up until now we have regarded a piece of music as a "bag of notes" in which the order of the pitches, rhythms, or whatever aspect we considered, had no importance. While this is certainly not a *true* assumption, it led to reasonably simple statistical models that proved useful in our applications. However, since the order of pitches, rhythms, or whatever musical attribute we treat, is fundamental to the nature of music, the depth and usefulness of the applications we can model as bags of notes is limited. In this section of the course we take a view of music as a *sequence*, rather than an orderless random sample. Of course, this view is still limited, since not all aspects of music are captured by the notion of a sequence. For example, much music is composed of a collection of separate voices. While each of these might be viewed as a sequence, the overall music is composed of a collection of *intertwining* sequences that have rather complicated dependence upon on another. However, there *are* aspects of music, such as functional harmony, that genuinely are sequences. Often the usefulness of a model does not depend only on how *correct* the model is. Rather, there is a trade off between simplicity and correctness. As a broad generalization, the most useful models tend to be the ones that navigate this trade off well — that is, models that are reasonably simple yet also capture important aspects of the data being modeled.

3.1 Dynamic Programming

A graph is a construction from computer science and discrete mathematics composed of a collection of *nodes* and *edges*. The nodes are places we might *visit* in the graph, while we can only travel between nodes that are connected by edges. A particular kind of graph is called a *trellis* — while more general definitions are possible, we call a graph a trellis if each of the nodes lives at a level: $0, 1, 2, \ldots, n$ and all edges connect nodes at successive levels. Such a trellis graph is depicted in Figure 3.1.

The graph in the figure is known as a *weighted* graph since each of the edges has an associated cost. A common problem one considers with such a graph is to find the minimal cost path from the *start* (leftmost node) to the *end* (rightmost node), where the cost of the path is the sum of the arc costs traversed along the path. This optimal path is computed using a technique known as *dynamic programming*, which relies on a single simple, yet powerful, observation as follows:

Key Observation and Dynamic Programming Algorithm

For each node, n, in the trellis, the best scoring path to the node is a best scoring path to a node, m, at the previous level, plus the arc (m, n).

This is clearly true since if we were to consider a suboptimal path from the start to m plus the arc (m, n), this must achieve a worse score than the optimal path to m plus (m, n). The observation leads directly to an algorithm for computing the optimal path.

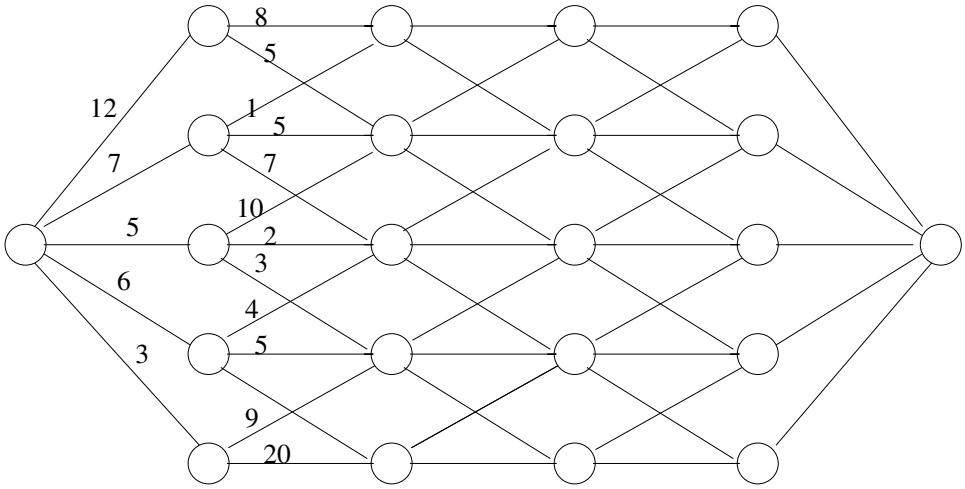


Figure 3.1: A "weighted" trellis graph in which each arc of the graph has a cost. We wish to find the least costly path beginning at the leftmost trellis node and ending at the rightmost trellis node.

Let $c(i, j)$ be the cost realized in going from node i to node j. Let $m(i)$ be the cost of the optimal path from the start to node i and set $m(\text{start}) = 0$. Then, reasoning from our "key observation," we see that the score of the optimal path to node j must be the *best* of the optimal paths to the predecessors, i, of j with (i, j) concatenated onto these optimal paths. Put algorithmically, we can compute the score of the optimal path from start to end by letting

$$m(j) = \min_{i \in \text{pred}(j)} m(i) + c(i, j)$$

Then when these optimal costs are all computed, $m(\text{end})$ will be the optimal cost from start to end.

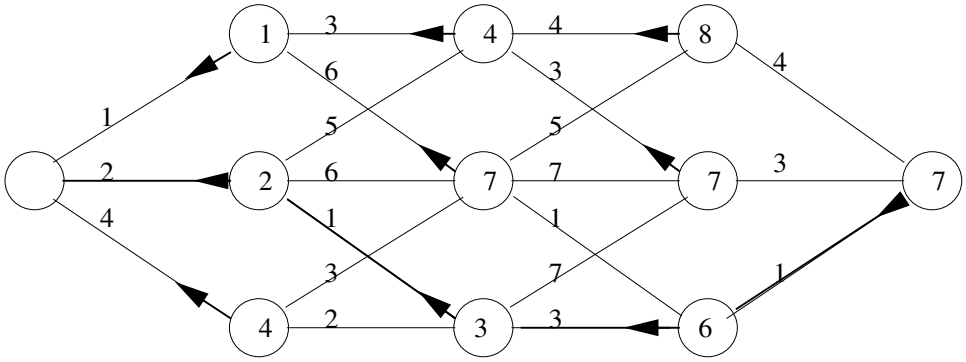
Of course, we really wanted to find the *optimal path*, not just its cost. A simple addition to our algorithm solves this problem. Let

$$a(j) = \arg \min_{i \in \text{pred}(j)} m(i) + c(i, j)$$

so that $a(j)$ is the optimal predecessor of node j. When we have computed both of these quantities for all trellis nodes, we can then "trace back" the optimal path by following the optimal predecessors back from the end node. That is, the optimal path (in reverse order) is given by:

$$\text{end}, a(\text{end}), a(a(\text{end})), a(a(a(\text{end}))), \ldots, \text{start}$$

In the following simple example we have filled in the circles of each node j with the optimal cost to j, $m(j)$, and have indicated the optimal predecessors of each node with an arrow that points backward along the appropriate arc.



3.1.1 Dynamic Programming Examples

While at first blush, dynamic programming (DP), seems like a rather abstract notion, the technique finds application in a very large range of practical domains such as robot navigation, DNA sequencing, computer vision, error correcting codes, optimal planning problems, speech processing, and really many more. This section shows three examples of DP applied to musical problems: piano fingering, voice leading and finding hypermetric structure.

Piano Fingering (piano_fingering.r)

DP can be applied to a whole range of instrument fingering problems to find natural ways to finger difficult passages. Among the most interesting of these are the piano and the guitar, since these constitute particularly difficult challenges. It would be fair to say that both problems are still "open" from a technical point of view, meaning there is considerable room for improvement over what any algorithmic approach can offer. We look here at a simplification of the problem to the barest essentials, in an effort to illustrate how DP can contribute.

Suppose, rather than looking at the entire range of piano fingering challenges, we limit our view to the following.

- We will only consider a single hand.
- We look only at monophonic (one-note-at-a-time) music.
- We do not consider the way in which the *damper* and *sostenuto* pedals can be used to solve fingering issues.
- We do not allow finger "substitutions" (changing the finger on a held note).
- We do not consider any rhythmic component, thus our view of the music is simply a sequence of pitches.
- We only consider the white keys.

DP can be employed even if many, or perhaps all, of these restrictions are dropped, however the problem becomes more complex. Our goal here is to pose a plausible and *transparent* application of DP to a musical problem.

In keeping with our assumptions, we view the music as a sequence of (white) pitches using the numbering $\ldots$, -2 = a, -1 = b,0 = c', 1 = d', 2 = e', $\ldots$, with no regard for rhythm: $m_1, m_2, \ldots$. A possible fingering assigns one finger from

$$F = \{\underbrace{\text{thumb}}_{1}, \underbrace{\text{index}}_{2}, \underbrace{\text{middle}}_{3}, \underbrace{\text{ring}}_{4}, \underbrace{\text{pinkie}}_{5}\}$$

to each note in the sequence. We will then write our fingering of the music as:

$$(m_1, f_1), (m_2, f_2), \ldots$$

where $f_i \in F$.

Example (Mary had a Little Lamb)

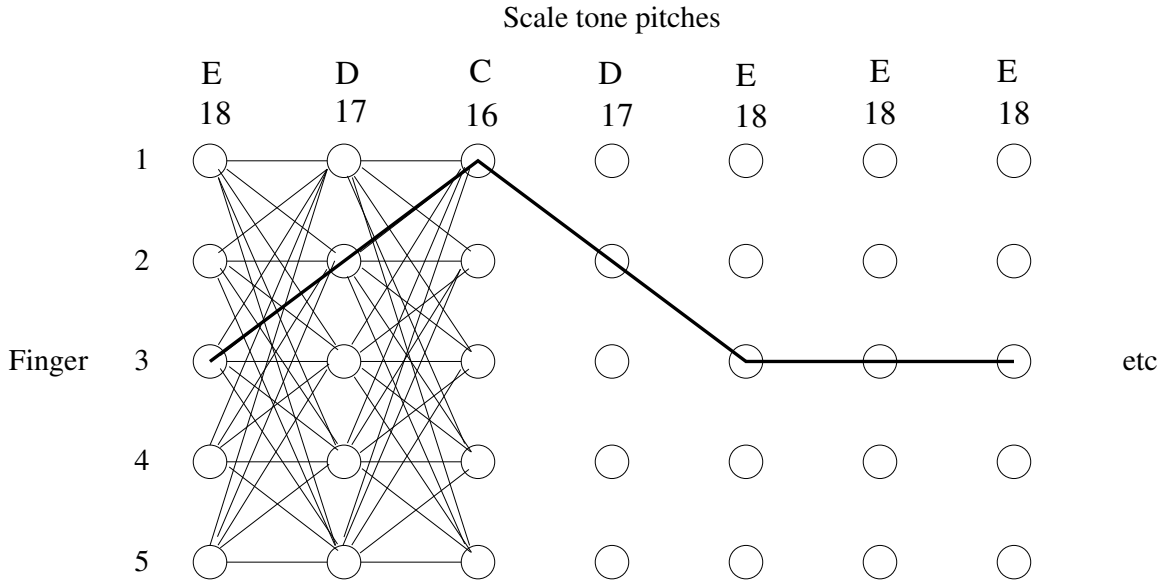
The music here, in the key of C Major, is given by $18, 17, 16, 17, 18, 18, 18, \ldots$ and a possible fingering would be

$$(18, 2), (17, 2), (16, 2), (17, 2), (18, 2), (18, 2), (18, 2) \ldots$$

This 5-year-old-style fingering suggests that we play each note with the index finger. Can we do better?

Idea: Fingering = Path Through Trellis

We view the *state* of the performance as a (note,finger) pair and view the performance as a sequence of states. We define a *cost* for going through each possible state transition and express the total cost of a performance as the sum of all transitions costs along the performance (=path). We find the optimal performance through dynamic programming. This is illustrated in the following figure.



Note that in this state space we don't have an explicit start state and end state. We could add a start state by putting a single state at the left edge of this array and connecting it to each state in the 1st column. Similarly, we could add an end state by connecting it to the last column. However, there is really no need to do this. Rather, we modify our conception of DP as now looking for the least expensive path from *any* state in the 1st column to *any* state in the last column. There is almost no modification of the basic DP algorithm needed to treat this case.

How to express the transition costs?

1. There is a symmetry in fingering that simplifies our situation somewhat. For instance, the difficulty of starting with the thumb on a note and stretching to the octave above with the ring finger is about the same as starting with the ring finger on the octave above and stretching downward one octave with the thumb. We will assume this symmetry holds in general. That is, $C((m_1, f_1), (m_2, f_2)) = C((m_2, f_2), (m_1, f_1))$. The essential idea here is that the difficulty of both of these situations is really about the same as the difficulty of playing both notes at once with the two fingers. **With this symmetry assumption we can assume that $m_2 \geq m_1$ in all that follows, since if this is not true, then we can compute the cost by switching (m_1, f_1) and (m_2, f_2).**

2. When no finger crossing occurs (i.e. $f_2 > f_1$) would like to have the difference of fingers about equal to the difference in pitches $m_2 - m_1 \approx f_2 - f_1$. We model this as

$$C((m_1, f_1), (m_2, f_2)) = |(m_2 - m_1) - (f_2 - f_1)|$$

3. We want to discourage using the same finger for different consecutive notes. So when $f_1 = f_2$

$$C((m_1, f_1), (m_2, f_2)) = \begin{cases} 0 & \text{if } m_1 = m_2 \\ 100 & \text{otherwise} \end{cases}$$

4. Finger crossings are more difficult to execute than non-crossings, so we want a mild penalty to discourage finger crossings when unnecessary (think of trilling with the right-hand middle finger below the thumb). When crossover occurs, $(f_2 < f_1)$, we penalize this greatly unless the notes are neighbors and we cross the thumb under the middle or ring finger.

$$C((m_1, f_1), (m_2, f_2)) = \begin{cases} 3 & \text{if } f_2 = 1 \text{ and } f_1 \in \{3, 4\} \text{ and } |m_2 - m_1| = 2 \\ 100 & \text{otherwise} \end{cases}$$

The implementation of this on the computer requires some thought, and since it is our first dynamic programming example we do it in some detail. We can think of the states of our trellis as a two-dimensional array, with 5 rows (one for each finger) and N columns, one for each note. We will use i,j to designate the state in the ith row and jth column, which corresponds to the choice of using finger i for note j. We first discuss computing the function m[i,j], giving the optimal score to node i,j. Note that we need to compute m for the (j-1)th column before we compute it for the j column, so our initial code must begin with

```
for (i in 1:5) m[i,1] = 0       # no cost to choosing any finger for the first note
for (j in 2:N) {
  # compute m[,j] using m[,j-1]
}
```

We will now compute the jth column of m, m[,j] in another loop that goes *inside* the loop we already have. This inner loop will be over the various possible finger choices for the jth note.

```
for (i in 1:5) m[i,1] = 0
for (j in 2:N) {
  for (i in 1:5) {
    # compute the score of m[i,j]
  }
}
```

To compute m[i,j], we need to use the basic idea of dynamic programming. We will loop over all possible "predecessors" of node (i,j), which for this problem are (1,j-1) (2, j-1) ..., (5,j-1), since we can play the previous note with any finger choice. In the code below we index these previous finger choices with the variable ii and compute the score we get coming to (i,j) *through* (ii,j-1), for each finger ii. Any time we find a score better than the "best-so-far" score, we remember this by resetting m[i,j]. We will denote the cost of going from state (ii,j-1) to state (i,j) as c(ii,j-1,i,j) here, though we will need to flesh this out in the final version of the program.

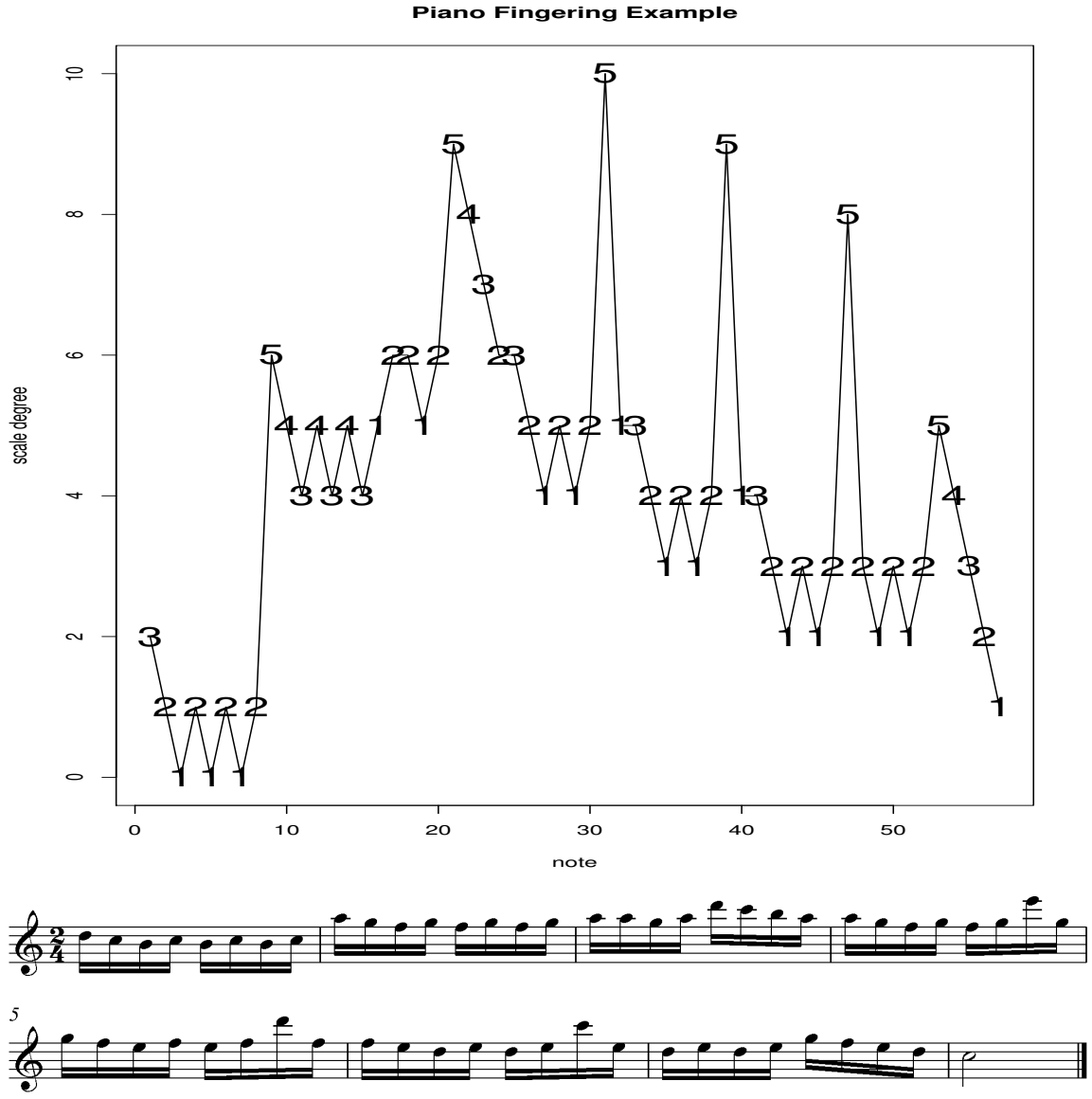


Figure 3.2: Piano fingering example for the 16th note variation of the Mozart "Twinkle, Twinkle, ...," variations. For simplity, some minor changes have been made so that notes all lie on the white keys. The finger, 1-5, used for each note is indicated in the plot while the scale degree is on the vertical axis.

```
for (i in 1:5) m[i,1] = 0
for (j in 2:N) {
  for (i in 1:5) {
    m[i,j] = Inf  # Inf = infinity --- we know we can do better than this score!
    for (ii in 1:5) {
      t = m[ii,j-1] + c(ii,j-1,i,j)
      if (t < m[i,j]) m[i,j] = t  # always true 1st time through loop
                                  # but may continue to improve as we iterate
    }
  }
}
```

At the end of this section of code we will have computed the array m correctly, giving the optimal score to each node in our graph. Remember that we are interested in finding the best possible *path* (i.e. the fingering) and not the best possible score. We accomplished this by adding a "pointer" to the best possible predecessor of each node in our drawings. From a computational perspective we will represent these pointers also as a two-dimensional array. That is, a[i,j] will be a number in 1 to 5, giving the finger of the best possible predecessor of node i,j. In other words, if a[i,j] = ii, then ii,j-1 is the best precessor of i,j. To accomplish this we need only make a very minor modification to the program:

```
for (i in 1:5) m[i,1] = 0
for (j in 2:N) {
  for (i in 1:5) {
    m[i,j] = Inf
    for (ii in 1:5) {
      t = m[ii,j-1] + c(ii,j-1,i,j)
      if (t < m[i,j]) {
        m[i,j] = t
        a[i,j] = ii         # now a[i,j] is the best-so-far predecessor
      }
    }
  }
}
```

Finally, we must actually compute the best possible path. First we find the best scoring state at level N — the last note in the excerpt. This state will tell us the finger to use on the last note. Then we follow the pointers (the a[i,j]) bacward until we get to level 1. We will hold the optimal fingering in the array f[]. After the above code we need to add the following:

```
f[N] = which.min(m[,N])  # f[N] is the finger of the best scoring node at level N
for (j=N:2) f[j-1] = a[f[j],j]
```

This last line requires some thought to understand. Having found the optimal last finger, f[N], we go through the remaining notes backwards just as we followed the pointers backwards in our graph. We would choose the optimal finger at note N-1 as the optimal predecessor of node f[N],N — a[f[N],N]. This becomes our optimal finger for note N-1: f[N-1] = a[f[N],N]. Similarly, we choose the optimal predecessor of finger f[N-1] at level N-1 to be our finger choice for level N-2. That is: f[N-2] = a[f[N-1],N-1]. The loop makes quick work of these assignments.

The R program **piano_fingering.r** shows a fleshed out implementation of DP for this problem, but really looks much like what we have above. In the program, we represent both the optimal scores and the optimal predecessors as the two-dimensional arrays **opt_score** and **opt_pred[i,j]**. Dynamic programming is used to find the fingering that minimizes our cost function. Figure 3.2 shows an example of the fingering generated by our program for a simplification of the Mozart *Twinkle, Twinkle* 16th note variation.

Voice Leading Using Dynamic Programming

Suppose we have a sequence of chords, $T_1, T_2, \ldots, T_N$ where, for simplicity, each chord is a triad (here meaning a collection of 3 of pitches consisting of alternating scale tones). While we assume that each chord has three notes, it is not difficult to generalize this arbitrary chords. We will write $T_{n,1}, T_{n,2}, T_{n,3}$ for the 3 midi pitches of the nth triad, moving from bottom to top. We will use the usual terminology of "root," "3rd," and "fifth" to refer to the three pitches of the chord.

For a triad, T, write $I(T)$ for the "inversion" operation that moves the lowest note one octave up. Thus if T is in "root position" (with the root as the lowest member), then $I(T)$ is the "1st inversion" version of the chord which has the 3rd of the chord as the lowest member. Similarly, $I(I(T))$ is the "2nd inversion" of the chord with the 5th as the lowest member. We will write $I^k(T)$ for the operation $\underbrace{I(I(\ldots I}_{k \text{ times}}(T)\ldots))$ which iterates the inversion operation k times. Clearly after 3 of these operations, the chord has been moved one octave up, though we can continue the inversions as long as we like. Similarly, we write $I^{-1}(T)$ for the "downward" inversion that moves the highest note one octave down, and $I^{-k}(T)$ the operation that does this k times.

Suppose that for each chord T we let the possible inversions of the chord be $I^{-K}(T)\ldots, I^0(T), \ldots I^K(T)$ for some fixed number K. The dynamic programming state space of the problem corresponds to an $L \times N$ array of states where $L = 2K + 1$ is the number of inversions. In this array, the element in the nth column and kth row describes the possibility that we use the kth inversion for the nth triad. This is described by the array

$$\begin{array}{rrcr}
I^K(T_1) & I^K(T_2) & \ldots & I^K(T_N) \\
I^{K-1}(T_1) & I^{K-1}(T_2) & \ldots & I^{K-1}(T_N) \\
\vdots & \vdots & \vdots & \vdots \\
I^1(T_1) & I^1(T_2) & \ldots & I^1(T_N) \\
I^0(T_1) & I^0(T_2) & \ldots & I^0(T_N) \\
I^{-1}(T_1) & I^{-1}(T_2) & \ldots & I^{-1}(T_N) \\
\vdots & \vdots & \vdots & \vdots \\
I^{1-K}(T_1) & I^{1-K}(T_2) & \ldots & I^{1-K}(T_N) \\
I^{-K}(T_1) & I^{-K}(T_2) & \ldots & I^{-K}(T_N)
\end{array}$$

With the array described above, each state in the graph represents a particular voicing of a particular chord. Thus, any *path* through this graph from the left edge to the right edge constitutes a possible *voice leading*. We will describe a cost for each pair of chords in adjacent columns. Suppose that we consider going from configuration A to configuration B where both A and B are collections of 3 pitches notes using the MIDI numbering. In this case the lowest voice undergoes the transition A_1 to B_1, the middle voice goes from A_2 to B_2, etc. We want to minimize the total amount of movement encountered between voices while making this transition. Thus our cost, C will be

$$C(A, B) = \sum_{j=1}^{3} |A_j - B_j|$$

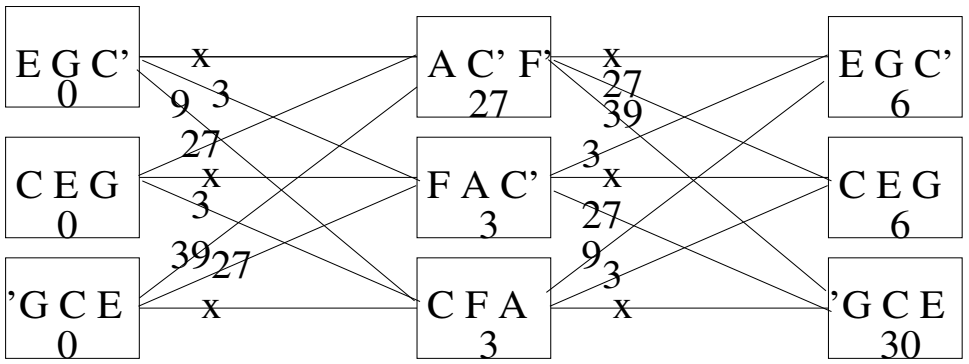


Figure 3.3: The voice leading example of C Major, F Major, C Major

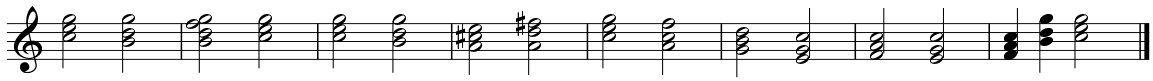


Figure 3.4: The voice leading produced by the DP algorithm. The chords that formed the input to the program are labeled above the chords

If we let the cost of a path be the sum of the transition costs encountered along the path, then we are trying to minimize the total movement in half steps traversed by the voices — this is we seek a "parsimonious" voice leading. This is something we know how to minimize with DP.

It is possible to choose voice leadings that do not allow parallel fifths, octaves, or to impose other constraints on our voice leading. For instance, if we wish to disallow parallel fifths we would modify the transition cost to be

$$C(A,B) = \begin{cases} \sum_{j=1}^{3} |A_j - B_j| & \text{if } A \to B \text{ contains no parallel fifths or fourths} \\ \infty & \text{otherwise} \end{cases}$$

It is a simple matter to check for parallel intervals by exhaustively looking at all 9 pairs of diads from A and B.

As a simple example, suppose our chord progression is C Major, F Major, C Major. We choose only 3 inversions for each triad so our picture does not become too messy — it is easy to handle any reasonable number of possibilities on the computer. Thus our states will be (C,E,G), (E,G,C'), ('G,C,E) for the first and last chords and (F,A,C'), (A,C',F'), (C,F,A) for the middle chord, where C denotes middle C, C' is one octave above, and 'C is one octave below. Figure 3.3 shows the costs between each chord as the sum of half steps traversed by each voice. Note that the we give infinite cost between the horizontal arcs which each involve parallel fifths. The states contain a number in each box which is the optimal cost to that box. It comes as no surprise that the state sequences (C,E,G), (C,F,A), (C,E,G) as well as (E,G,C'), (F,A,C'), (E,G,C') are both optimal sequences traversing only 6 half steps in total.

Figure 3.4 shows an example of the chord progression in "America the Beautiful" with the voice leading given by dynamic programming. In this example we have added an additional penalty to keep the voices from drifting too far from a reference pitch.

Finding Phrase Structure

The last example is, technically speaking, not really an example of dynamic programming, though it is included in this section on DP examples since it is so close in spirit.

Consider the following table that lists a number of familiar simple folks songs, Christmas carols, and popular music songs. For each piece of music the table includes the total number of measures, along with a decomposition of the total number of measures into groups. Many of the pieces consist of a number of measures that is a power of two: $2^3 = 8$, $2^4 = 16$, $2^5 = 32$ .... This is typical when all groupings are diadic. That is, if we have a two measure group, two of these groups form a phrase, two of these phrases form the piece, then we have 2x2x2=8 measures. Clearly this happens frequently in this collection of simple pieces of music, and in *many* kinds of music. This kind of grouping structure is common to all of the pieces in the table with 2^n measures.

Notice that some of the pieces have a number of measures that contains a factor of 3, such as "Down in the Valley" and "The First Noel." This can happen either because the piece is composed of groups of 3 measures, as in "Down in the Valley," or it can happen if the piece is composed of 3 phrases (or 3 *somethings*) as in "The First Noel." Not all pieces have such regular phrase structure in which the measure groupings are given by a product of simple factors. Consider, for example, "God Save the Queen." This piece begins with a phrase that is composed of 3 groups of 2 measures (2x3), then followed by a squarer 8 measures with diadic structure, 2x2x2. We will write the phrase structure for such a situation as 2x3 + 2x2x2 where the '+' denotes the concatenation of not-necessarily-related pieces. "O Come, O Come Emanuel" is a particularly interesting example of this, composed mostly of 3-bar groupings, but having a contrasting diadic grouping structure for the text, "rejoice, rejoice, Emanuel," which helps to highlight the way this passage expresses a more declarative feeling than does the rest of the carol. Look through the list and think about the phrase structure in some of the more unusual tunes such as "Scarborough Fair," "Shenandoah," and "Yesterday."

measures	name	"true" structure	recognized structure
16	amazing grace	4x2x2	2x4x2
16	america the beautiful	4x2x2	4x4
8	auld lang syne	4x2	4x2
32	home on the range	4x2x2x2	4x4x2
16	away in a manger1	4x2x2	4x4
16	away in a manger2	4x2x2	4x4
16	danny boy	2x2x2x2	2x4x2
24	down in the valley	6x2x2	3x4x2
16	early one morning	4x2x2	8x2
16	edelweiss	4x2x2	4x4
24	first noel	2x2x2x3	4x6
14	god save the queen	2x3 + 2x2x2x2	2x7
16	greensleeves	4x2x2	8x2
16	in the bleak midwinter	2x2x2x2	2x4x2
16	it came upon a midnight clear	4x2x2	4x4
8	loch lomond	4x2	4x2
19	o come o come emanuel	3x2x2 + 2x2 + 3	3x4 + 2x2 + 3
16	o little town of bethlehem	4x2x2	4x4
8	red river valley	2x2x2	2x4
8	riddle song	2x2x2	2x4
16	rock a bye baby	2x2x2x2	4x4
14	sakura	2x2 + 2 + 2x2x2	2x7
18	scarbarough fair	(5+7)x2	(5+4)x2
20	shenandoah	(2x2 + 2 + 2x2)x2	2x5x2
16	simple gifts	2x2x2x2	82x
8	somewhere over the rainbow	2x2x2	4x2
24	sound of music	2x2x2x3	2x4x3
8	streets of laredo	2x2x2	8
8	swing low	2x2x2	2x4
18	today	2x2x2x2 + 2	6x3
16	vilja merry widow	2x2x2x2	4x4
12	waters of babylon	2x2x3	4x3
8	water is wide	2x2x2	2x4
22	yesterday	7+2x2x2+7	2+5x4

The problem we wish to consider now is how to compute this kind of deeper structure from the music itself. In this problem we are given data files for each piece that represent the music as a list of notes with some other annotations. In our representation, we are also given measure boundaries. The following is an example of "Auld Lang Syne" as represented in our data set. For now we focus only on the first two columns, which contain the note lengths and the pitches.

```
mode: major
time: 4/4
tempo: 1/4 = 100
trans: -8
1/4 g1 > . .
-------------------------------------
3/8 c2 * . I
1/8 b1 > .
1/4 c2 > .
1/4 e2 > .
-------------------------------------
3/8 d2 * . V
1/8 c2 > .
1/4 d2 > .
1/8 e2 > .
1/8 d2 > .
-------------------------------------
3/8 c2 * . I
1/8 c2 > .
1/4 e2 > .
1/4 g2 > .
-------------------------------------
3/4 a2 + . IV
1/4 a2 > #
-------------------------------------
3/8 g2 * . I
1/8 e2 > .
1/4 e2 > .
1/4 c2 > .
-------------------------------------
3/8 d2 * . V
1/8 c2 > .
1/4 d2 > .
1/8 e2 > .
1/8 d2 > .
-------------------------------------
3/8 c2 * . vi
1/8 a1 > .
1/4 a1 > . IV
1/4 g1 > .
-------------------------------------
4/4 c2 ) . I
-------------------------------------
```

We will describe a DP-like method for finding the best grouping structure for a piece based on this data. However, before we can do this we need to think about the possible labellings for a collection of measures. Understanding the way to generate the possible labeling will be the key to finding the optimal labeling. For instance, imagine all the possible ways we could label a section of 4 measures. If we enumerate them

all we will get:

$$\{4, 3+1, 2+2, 1+3, 2\times 2, (1+1)\times 2, 1+1+2, 1+2+1, 2+1+1, 1+1+1+1\} \tag{3.1}$$

Clearly there are quite a few of these and it will be difficult to find, say, all the possible ways of labeling 16 bars without a systematic approach. We will proceed by *induction* — if this term is not familiar, it is best illustrated by example, rather than definition, so read on.

Imagine first that we have found all the ways of labeling a single bar. This is easy, since there is only 1 possible way: $\{1\}$. This is the *basis* for our induction. Now imagine we have found all ways of labeling $n-1$ bars (we *have* for $n=1$). We describe how to construct all the possible labellings of n bars out of what we already have. To do this there are only three types of possible labels:

1. We can label the n bars as a single unit: n.
2. We could try to label the bars as a product. So for each factor f of n (other than n and 1) we compute labels of the form $A\times f = \underbrace{A, A, \ldots, A}_{f \text{ times}}$ where A is a possible labeling of n/f bars.
3. For each $j \in 1, \ldots, n-1$ we could label the bars as $A+B$ where A is a possible label of j bars and B is a possible label of $n-j$ bars.

For instance, suppose we have the possible labellings of

1 bar $\{1\}$

2 bars $\{2, 1+1\}$

3 bars $\{3, 1+2, 2+1, 1+1+1\}$

and from these we want to create the possible labellings of 4 bars. As described above, we have three types of labellings: those coming from single labellings, products, and sums. So listing all possibilities when $n=4$ gives

1. One "whole" labeling: $\{4\}$
2. The one nontrivial factor of 4 is 2 so we get product labellings $\{2, 1+1\}\otimes\{2\} = \{2\times 2, (1+1)\times 2\}$.
3. The sum labellings
   (a) for $j=1$ we get $\{1\}\oplus\{3, 1+2, 2+1, 1+1+1\} = \{1+3, 1+1+2, 1+2+1, 1+1+1+1\}$
   (b) for $j=2$ we get $\{2, 1+1\}\oplus\{2, 1+1\} = \{2+2, 2+1+1, 1+1+2, 1+1+1+1\}$
   (c) for $j=3$ we get $\{3, 1+2, 2+1, 1+1+1\}\oplus\{1\} = \{3+1, 1+2+1, 2+1+1, 1+1+1+1\}$

It is easy to see that any labeling with more than 2 +'s can appear in multiple ways, but it is easy to thin out duplicate labellings as long as we have a way to generate them all. Observe that the labels of Eqn. 3.1 are exactly the same labellings as those generated by our 3-step procedure.

Now that we know how to generate all the possible labellings, let's consider a method of *scoring* them as to their musical plausibility for a *particular* piece of music. There is no single correct way to do this, but I will propose something that is at least plausible. Suppose we denote the segment of measures $m, m+1, \ldots, n-1$ by (m, n). Note that this doesn't actually include the nth measure, but goes up to the barline of the nth measure. Now imagine we have already managed to score all shorter subsequences of (m, n).

1. If we consider labeling (m, n) as a single phrase, S, we want score the plausibility that this sequence of measures is a phrase. We expect to have long notes at the end of phrases so we will impose an increasing penalty on the labeling as the longest note as the last measure becomes shorter. Thus every time we label a segment (m, n) as a single phrase, we will impose the cost
$$C(S, m, n) = H_{\text{long note}}(m, n)$$

2. We have a preference for labellings that do not involve lots of +'s. This is because it is unusual for neighboring sections of music to be unrelated metrically, which is what is implied by the + operation. So we will assign a fixed cost every time two sections are concatenated with +, as well as impose a limit of at most 2 +'s for musical excerpt. This also will limit the rapid growth of possible labellings as we consider longer and longer units. Thus if we create a label on (m, n) by concatenating labels A on (m, j) and B on (j, n) where $m < j < n$, the score of this label will be
$$C(A + B, m, n) = \begin{cases} C(A, m, j) + C(B, j, n) + H_{\text{plus cost}} & \text{if } A + B \text{ has less than 2 +'s} \\ \infty & \text{otherwise} \end{cases}$$
We wont bother adding labels with infinite cost to the list of possibilities.

3. If a label on (m, n) is composed by replicating the k-measure label A f times where $f \times k = n - m$, then
$$C(A \times f, m, n) = C(A, m, m + k) + C(A, m + k, m + 2k) + \ldots, C(A, n - k, n) + H_{\text{similarity}}$$
where $H_{\text{similarity}}$ is a penalty that is larger the more the corresponding measures of music differ from one another. The simplest approach might leave out this penalty all together, but a more sophisticated view expects parallel elements of the same product to be have similar structures, either in pitch, rhythm, or both.

Note that this construction of the cost follows closely the construction of the possible labels in an inductive manner, building larger labels *and* costs out of shorter ones. That is, every time we create a new label we do this by either labeling a section of measures as a single phrase, composing two neighboring sections with '+', or composing a collection of of neighboring sections with $\times$. In each of these cases we have described how to construct the cost of the new label, using the cost of the smaller labels.

Though we will not discuss the coding of this experiment, which is more complicated than our previous R programs, we implemented this algorithm and ran it on the musical examples of the table given above. The final column of this table is the *recognized* rhythmic structure.

In comparing this recognized structure with the neighboring column, one should consider the objective of the algorithm more carefully. If no "dissimilarity" penalty is added when we consider the label $(A, A \ldots, A)$ then we have no way of choosing between various factorizations, such as 2×4 and $2 \times 2 \times 2$ that consider similar representations of structure. When adding the dissimilarity penalty we encourage explanations where the various examples of A in the product label $(A, A, A \ldots, A)$ are similar to one another. One can argue if this is a good way to distinguish between such hypotheses, but it is the way we have proposed here. Note that the *ground truth* structures supplied in the table don't make such distinctions, since all examples of 2^n measures are expressed as $2 \times 2 \ldots \times 2$.

3.2 Hidden Markov Models

In the beginning of this course we used the "bag of notes" model to reach our conclusions about the music under study. While clearly simplistic, we did find some useful applications of the approach. Last chapter we saw the value of taking a sequence-based view of music. Here we continue with this idea, but now in a probabilistic context.

3.2.1 Markov Chains

A *Markov Chain* is a sequence of random variables $x_1, x_2, \ldots$ taking values in some finite state space $\Omega = \{\omega_1, \ldots, \omega_L\}$ such that

$$p(x_n|x_1, \ldots, x_{n-1}) = p(x_n|x_{n-1})$$

That is, each state depends only on the previous state. To make this definition more clear, consider the case of four random variables x_1, x_2, x_3, x_4. No matter what kind of dependence these variables have, it is always true that

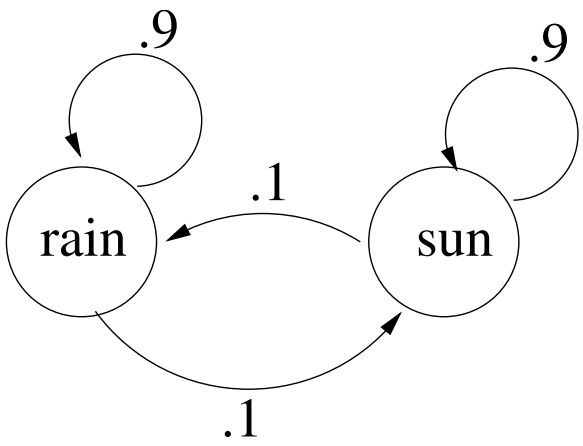
$$p(x_1, x_2, x_3, x_4) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)p(x_4|x_1, x_2, x_3)$$

We will show this later, however it is intuitively plausible as an extension of our definition for conditional probability. If the variables are the first four variables of a Markov chain, then the third and fourth factors simplify giving

$$p(x_1, x_2, x_3, x_4) = p(x_1)p(x_2|x_1)p(x_3|x_2)p(x_4|x_3)$$

State Graph

Usually a Markov chain is depicted with a *State Graph* showing the states as nodes in a graph with directed (with arrow) edges showing transition probabilities. The sum of the probabilities of all edges leaving a state must be 1. If we denote sun by S and rain by R, a sequence generated by this Markov chain would look like $SSSSSRRRRRRRRRRSSSSSSSRRRR$



Musical Examples

It is possible to create Markov chains that capture, to some extent, the tendencies of certain musical sequences. For instance if think of the key label for each measure of a piece of music we might might model this as a Markov chain with the following state graph. Here we only show the transitions from C, with the expectation that the transitions out of the other keys would simply be "rotations" of the $P(\text{key}|C)$ distribution. In Figure 3.5 figure the darker lines indicate more likely transitions.

Another example is that of musical rhythm. We can represent musical rhythm by enumerating the states as the possible places in the measure where a note can begin. For instance, we may do this in 3/4 time as $\Omega = \{0/1, 1/8, 1/4, 3/8, 2/4, 5/8\}$ if we believe the only possible note onset positions are at the

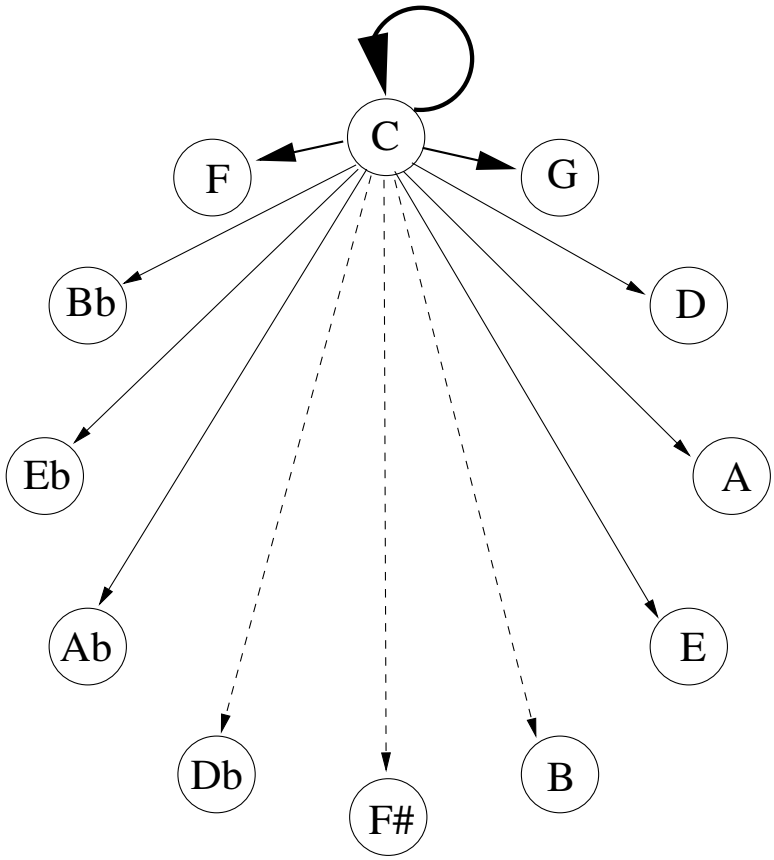
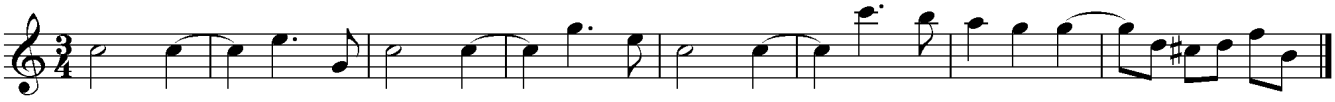


Figure 3.5: Typical key transitions from C major. Transitions from other keys could be modeled as "translations" of this model.

eighth note divisions. This creates a little ambiguity, however, since by the transition $1/4 \rightarrow 2/4$ we could mean that the note lasted for $1, 4, 7, 10, \ldots$ quarter notes. We can resolve this by introducing the tie symbol every time a note is tied over the bar line. Thus the rhythm of the fragment



would be notated unambiguously as

0/1	2/4	tie	1/4	5/8	0/1	2/4	tie	1/4	5/8	0/1	2/4	tie	1/4	5/8
1/4	2/4	3/4	tie	1/8	1/4	3/8	2/4	5/8						

What would the transition probabilities be in this case? One might make generalizations such as

1. The "odd positions (1/8, 3/8, 5/8) tend to be followed by the next quarter note position
2. Ties are relatively rare
3. ...

However, one of the virtues of the Markov model is we can assign transitions probabilities through parameter estimation. It will come as no surprise, given what we have seen that this amounts to *counting*. For instance:

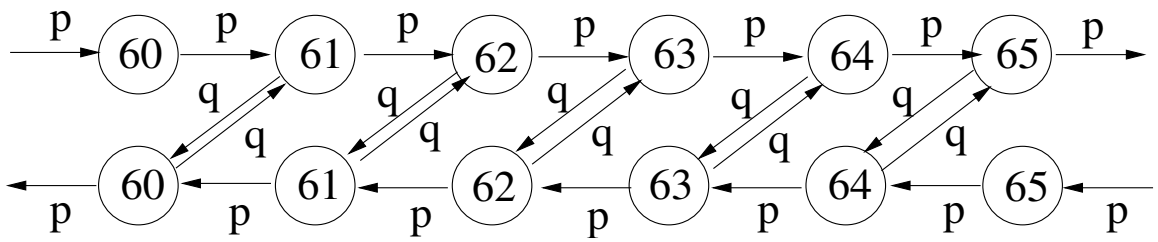
$$\hat{p}(\text{tie}|2/4) = \frac{\#\{i : x_i = 2/4, x_{i+1} = \text{tie}\}}{\#\{i : x_i = 2/4\}}$$

If we were to use the above sequence as our training data we see that the state 2/4 appears 5 times and in 4 of these cases it is followed by a tie. Thus $\hat{p}(\text{tie}|2/4) = 4/5$.

Sound in R (chromatic_wanderings.r)

There is a package in R that allows one to import and create sound files. The package works at the *sample* level, meaning that R's view of these sound files is *not* symbolic, but rather as a sequence of SR numbers per second where typical sampling rates are $SR = 8000, 16000, \ldots$ This view of music is not the perspective we take in this class, for the most part, however, it is useful for getting simple sound examples that convey the results of our experiments. The tutorial on the web page shows how to install the **TuneR** sound package, and how to get simple sounds out of it.

Using this package, we can create sound files that describe the results of Markov chains in action. The following figure describes a simple Markov chain for creating a sequence of pitches. The **chromatic_wanderings.r** program samples from this Markov chain and plays the sound file the results.



Here the program is listed with comments

```
sr = 8000; # sampling rate used in playing function
tatum = .1;  # the length of time (secs) of the smallest musical time unit

playit = function(f) {  # this function plays a melody.  the melody is expressed
  # using a midi pitch for each evenly spaced tatum value
# no need to worry about how the function works.
  h = NULL;
  library(tuneR) # need to link with the sound library
  bits = 16 # bit depth
  f[f<1] = f[f<1] + 8
  for (i in 1:length(f)) {
    fr = 440*2^((f[i]-69)/12);
    h = c(h,rep(fr,sr*tatum))
  }
  y = sin(2*pi*cumsum(h/sr))
  u = Wave(2^(bits-4)*y, samp.rate = sr, bit=bits)
  play(u,"play")        # how to play it on Linux --- this will be different on different platform
}

lo = 80;    # the lowest possible midi pitch
hi = 100;   # the highest possible midi pitch
p = .8      # prob of keeping same direction
notes = 50  # how many notes the model will generate
s = rep(0,notes);
s[1] = lo;  # start at bottom of pitch range
up = 1      # start with upward chromatic direction
for (i in 2:notes) {
  if (up) { s[i] = min(s[i-1]+1,hi); }  # if up move upward if possible
  else { s[i] = max(lo,s[i-1]-1); }
  if (runif(1) > p) up = 1-up;  # 1-p is prob of switching directions
}
playit(s)  # play the notes
```

Factorization of Markov Chain (mc_sim.r, markov_train_synth.r)

We denote the joint probability function for the 1st n states of the Markov chain to be $p(x_1, x_2, \ldots, x_N)$. We are interested in breaking this down to a simpler form. By our definition of conditional probability we know that

$$p(A, B) = p(A)p(B|A)$$

We apply this rule twice to get to get

$$p(\underbrace{x_1}_{A}, \underbrace{x_2, x_3}_{B}) = p(\underbrace{x_1}_{A})p(\underbrace{x_2, x_3}_{B} | \underbrace{x_1}_{A}) = p(x_1)p(x_2|x_1), p(x_3|x_1, x_2)$$

where in the last equation we applied the conditional probability rule to $p(x_2, x_3|x_1)$. Note that this is simply a probability for both x_2 and x_3 so the usual rule for of $p(x_2, x_3) = p(x_2), p(x_3|x_2)$ becomes, $p(x_2, x_3|x_1) = p(x_2|x_1)p(x_3|x_1, x_2)$. If we apply this idea to longer strings of state variables we get, more generally,

$$p(x_1, x_2, \ldots, x_N) = p(x_1)p(x_2|x_1), p(x_3|x_1, x_2) \ldots p(x_N|x_1, x_2, \ldots, x_{N-1})$$

This is always true number what kind of distribution we have on the variables.

For a Markov chain, we have seen that only the most recent past is relevant so the above equation simplifies to

$$p(x_1, x_2, \ldots, x_N) = p(x_1)p(x_2|x_1), p(x_3|x_2) \ldots p(x_N|x_{N-1})$$

In a *time homogeneous* Markov chain, which we will study, one assumes that the same transition distribution, $p(x_{n+1}|x_n)$ is used at all times n.

As a result of this, we see that in order to define a Markov chain we need

1. The *initial* distribution $p(x_1)$
2. The *transition probability matrix* Q where $Q_{ij} = p(x_{n+1} = \omega_j | x_n = \omega_i)$

For instance, in the "Rain and Sun" Markov chain introduced above, we would have two states $\Omega = \{\omega_1, \omega_2\} = \{\text{Rain}, \text{Sun}\}$ where

$$Q = \begin{pmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{pmatrix} = \begin{pmatrix} p(\omega_1|\omega_1) & p(\omega_2|\omega_1) \\ p(\omega_1|\omega_2) & p(\omega_2|\omega_2) \end{pmatrix} = \begin{pmatrix} .9 & .1 \\ .1 & .9 \end{pmatrix}$$

The R example **mc_sim.r** shows how to synthesize a chain from using a transition probability matrix. With this level of abstraction the synthesis is completely generic and could be used for any transition probability matrix, Q. The example **markov_train_synth.r** shows the training of Markov model for rhythm from actual music data. The program then synthesis a random rhythm and plays it back using the pitch of the note to denote different measure positions.

Higher Order Markov Models (markov_2nd_order_trans.r)

In many situations one can predict more about future states given knowledge of *several* of the most recent states. For instance, consider the situation where we have data that looks like:

$$AABBAABBAABBAABB \ldots$$

Imagine we learn transition probabilities from the training data. We would get, for instance,

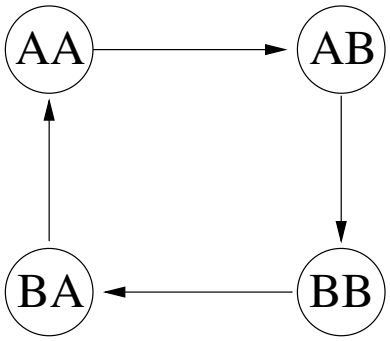
$$\hat{p}(A|A) = \frac{\#\text{ times we observe AA}}{\#\text{ times we observe A}} = 1/2$$

with the identical probabilities estimated for the other transitions so : $\hat{p}(A|A) = \hat{p}(B|A) = \hat{p}(A|B) = \hat{p}(B|B) = 1/2$ This learned model tells us that $\hat{p}(A|B) = \hat{p}(A|A)$ and that $\hat{p}(B|A) = \hat{p}(B|B)$ so that the knowledge of the current state tells us *nothing* about the next state. Thus, this Markov model believes that we can best represent this sequence data as random flips of a coin. Clearly this doesn't capture the obvious cyclical patterns generated by the observed data.

Alternatively, we could model the state as *two* successive observations from the sequence. If we do this we see that every time we observe AA it is followed by B and every time we see BA it is followed by B. If we estimate the transition probabilities in this case we would view our sequence data as pairs of *overlapping* observations: $AA, AB, BB, BA, AA, AB, BB, BA, \ldots$ Thus we would get

$$\begin{aligned} \hat{p}(AB|AA) &= 1 \\ \hat{p}(BB|AB) &= 1 \\ \hat{p}(BA|BB) &= 1 \\ \hat{p}(AA|BA) &= 1 \end{aligned}$$

This model could be represented by the state graph



Clearly, this model captures the behavior of the data more accurately.

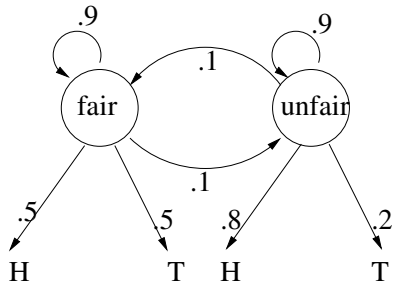
This idea can be extended to viewing the state as any number of consecutive observations. The number of consecutive observations used is called the *order* of the model. It is interesting to note that, *if* we have an unlimited supply of data, the model's ability to capture the behavior of the data always improves as we look at larger order models, as measured by nearly any reasonable measure of model performance. However, when dealing with finite supplies of training data, as we always do in practice, this trade off is considerably more complex. In a nutshell, the more parameters, or *tweakable* numbers of a model there are, the more damage is done by the inevitable inaccuracies in learning these parameters. This often leads to *poorer* model performance if the learning inaccuracy proves more important than the increased modeling power.

The R example **markov_2nd_order_trans.r** continues the rhythm synthesis, this time using a 2nd order Markov model. In a statistical sense the rhythms should be "more like" those demonstrated in the training, though we may or may not perceive this in a musical sense. But, for instance, the model should be capable of predicting future values with more accuracy.

Hidden Markov Models

Suppose we have a Markov chain, $x = x_1, x_2, x_3, \ldots$, but *do not* observe x directly. Instead, each time we visit a state, the state "outputs" a random observation from some distribution associated with the state.

As an example, consider the following figure that describes an observed sequence of coin flips. In this example there are two coins, a "fair" coin and an "unfair" coin. The fair coin gives equal probability to H (heads) and T (tails), but the unfair coin gives probability .8 to H (and .2 to T). The coin flipper flips the current coin and then makes a random choice of which coin to use next. In doing this, she stays with the current coin with probability .9 and switches to the other coin with probability .1.



An observed sequence from the coin may looking something like the following.

$$\underbrace{THHTHHTTHHTT}_{\text{fair?}}\underbrace{HHHTHHTHHHHHHH}_{\text{unfair?}}\underbrace{THTHHTH\ldots}_{\text{fair?}}$$

In hidden Markov model problems (HMMs), the inference problem is usually to give meaning to the observations by uncovering the sequence of hidden states. This meaning (here which coin is operating) is annotated in the equation above for a particular sequence of flips — of course, we can't know for sure which coin was operating at any time, since the observed sequence is possible under *any* sequence of coins.

More formally, we assume that each observation, y_n, depends only on the current state, x_n. We will write x and y for the entire sequences of states and observables: $x = (x_1, x_2, \ldots, x_N)$, $y = (y_1, y_2, \ldots, y_N)$. Thus

$$\begin{aligned} p(y|x) &= p(y_1, \ldots, y_N | x_1, \ldots, x_N) \\ &= p(y_1|x_1, \ldots, x_N) p(y_2|y_1, x_1, \ldots, x_N) \ldots p(y_N|y_1, \ldots, y_{N-1}, x_1, \ldots, x_N) \\ &= p(y_1|x_1) p(y_2|x_2) \ldots p(y_N|x_N) \end{aligned}$$

Putting this together with the factorization of the Markov chain, x, gives

$$\begin{aligned} p(x,y) &= p(x)p(y|x) \\ &= p(x_1)p(x_2|x_1)p(x_3|x_2)\ldots p(x_N|x_{N-1}) \\ &\quad \times p(y_1|x_1)p(y_2|x_2)p(y_3|x_3)\ldots p(y_N|x_N) \\ &= \underbrace{p(x_1)p(y_1|x_1)}_{f(x_1)}\underbrace{p(x_2|x_1)p(y_2|x_2)}_{f(x_1,x_2)}\underbrace{p(x_3|x_2)p(y_3|x_3)}_{f(x_2,x_3)}\ldots\underbrace{p(x_N|x_{N-1})p(y_N|x_N)}_{f(x_{N-1},x_N)} \end{aligned}$$

Usually in an HMM the y's are *observed* and hence are known to us. On the other hand, the x's are *unobserved*, and hence unknown. Thus, we can think of this factorization as a product of functions where

the first, $f(x_1)$, just depends on the first state variable, the 2nd, $f(x_1, x_2)$, depends on the first two state variables, the third, $f(x_2, x_3)$, depends on the next two state variables, etc. This view of the factorization will be important to us in what follows.

HMMs for Recognition (coin_hmm.r)

Suppose we have an HMM, (x, y), with $p(x, y)$ as before. We *observe* $y = (y_1, y_2, \ldots, y_N)$ and want to find the *most likely* state sequence given our observations. That is, we seek $\hat{x} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ where

$$\hat{x} = \arg\max_x p(x|y)$$

Note that since $p(x|y) = \frac{p(x,y)}{p(y)}$ and y is *fixed*, we have

$$\begin{aligned}
\hat{x} &= \arg\max_x p(x|y) \\
&= \arg\max_x p(x, y) \\
&= \arg\max_{x_1,\ldots,x_N} [p(x_1)p(y_1|x_1)][p(x_2|x_1)p(y_2|x_2)] \ldots [p(x_N|x_{N-1})p(y_N|x_N)]
\end{aligned}$$

The important thing to notice about this factorization is that we express the quantity we want to optimize, $p(x, y)$, as a product of factors depending on *consecutive* variables from the chain (x_n, x_{n+1}). This is ideal for maximizing using dynamic programming, as follows.

Suppose we observe an N-long sequence $y_1, \ldots, y_N$ and our state variables x_n can take values in a state space $\Omega = \{\omega_1, \ldots, \omega_L\}$ having L different states. We create a trellis graph having N "levels" with L possible states at each level — thus we can think of the trellis as an $L \times N$ array of states. We define the initial score for the state l in layer $n = 1$ as $p(x_1 = \omega_l)p(y_1|x_1 = \omega_l)$. Then we define the transition score for moving from state l at stage $n - 1$ to state l' at stage n as $p(x_n = \omega_{l'}|x_{n-1} = \omega_l)p(y_n|x_n = \omega_{l'})$

Suppose now that we take the score of an entire path through the lattice as the *product* of the scores traversed along the arcs. (In the past we took the sum of arc scores, but this is just a minor difference). Then for any path through the lattice, (that is, any N-long sequence of states), $x_1, x_2, \ldots, x_N$, the score of this path is

$$p(x, y) = [p(x_1)p(y_1|x_1)][p(x_2|x_1)p(y_2|x_2)] \ldots [p(x_N|x_{N-1})p(y_N|x_N)]$$

where each factor inside the brackets represents a transition score at the associated level of the trellis graph. Thus we have constructed a weighted graph where, for each path $x = x_1, \ldots, x_N$, the score of this path is $p(x, y)$. Since we can use dynamic programming to find the best scoring path through the trellis, we will have also found the state sequence that maximizes $p(x, y)$.

This is best illustrated in terms of an example, so let's return to the simple HMM describing the fair and biased coins. Suppose that we observe the sequence $y = HTH \ldots$ and consider Figure 3.6.

Note that the first flip is H, so the score for beginning in the fair state, F (moving from the start state to F), will be $p(F)p(H|F)$ while the score for beginning in the biased state (moving from the start state to B) will be $p(B)p(H|B)$. This reflects our prior probabilities for beginning in the two states, $p(F)$ and $p(B)$, as well as the likelihood for observing our first flip, which was H, in the two states, $p(H|F)$ and $p(H|B)$.

Then in the second level of the trellis there are four possible transitions to consider, $F \to F$, $F \to B$, $B \to F$,$B \to F$, each with a prior probability, $p(F|F)$, $p(B|F)$, $p(F|B)$, $p(B|B)$. Since our second observed flip was T, we must also factor in the observation probability associated with the transitions. Thus, if we move to state F (from either preceding state) the observation probability will be $p(T|F)$ while if we move to state B the probability will be $p(T|B)$. These are exactly the transition scores that are indicated in the Figure 3.6.

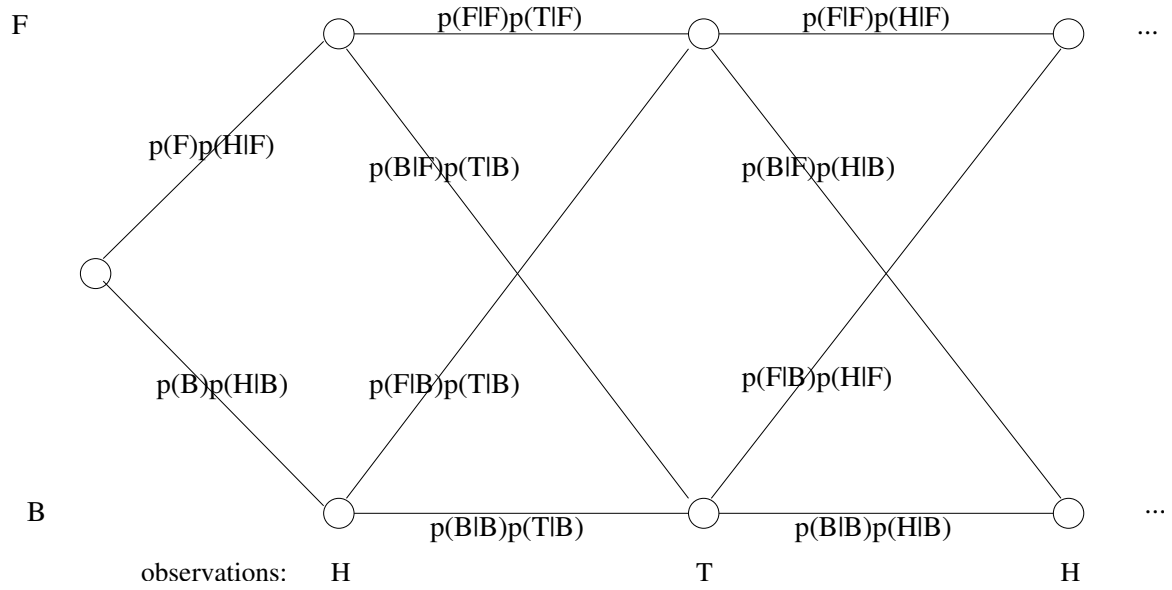


Figure 3.6: The trellis for the coin-flipping HMM with fair and biased coins.

Finally, there is one more trellis level sketched for the 3rd observation, H. This follows exactly the same pattern as we observed before except that our observation probabilities will be for H rather than T, as in the previous trellis level. Note that the product of all of the scores traversed along any path such as $x = BBBBBFFFFFBBBB\ldots$ will be $p(x, y)$. Thus, we can find the most likely sequence given our observations using dynamic programming.

The dynamic programming is nearly identical to the way we originally introduced the idea. Here we will use notation similar to when the concept of DP was first introduced in this chapter. The scores for the initial state, F, B will be $s_1(F) = p(F)p(H|F)$ and $s_1(B) = p(B)p(H|B)$ as indicated in the figure. We have also described the way we can construct transition scores, $s_n(F, F), s_n(F, B), s_n(B, F), s_n(B, B)$ for moving through the various state pairs on the nth level. Since there is only one possible path leading to the states at the first level, we will define the optimal scores for these as $m_1(F) = s_1(F)$ and $m_1(B) = s_1(B)$. But for subsequent levels we got the scores of the optimal paths as

$$\begin{aligned} m_n(F) &= \max(m_{n-1}(F)s_n(F, F), m_{n-1}(B)s_n(B, F)) \\ m_n(B) &= \max(m_{n-1}(F)s_n(F, B), m_{n-1}(B)s_n(B, B)) \end{aligned}$$

Thus, $m_N(F)$ and $m_N(B)$ will be the scores of the optimal paths ending in F and B. If one of these is larger than the other, it will be the optimal score through the graph which is the same as $\max_x p(x, y)$. Of course, we are interested in finding the best sequence, not the probability it creates, so we need to remember how we constructed the optimal score. To this end we simply need to remember which which predecessor state creates the optimal score at level n. This is done exactly as before by remembering the optimal predecessor to each trellis state.

The R program, **coin_hmm.r** illustrates the algorithm for finding this optimal path (as well as simulating data from an HMM). In this program our sequence of observed coin flips is $y[1 : N]$, here represented as 0 for H and 1 for T. The program constructs two $N \times 2$ arrays called **optscore** and **optpred** where **optscore[n,l]** is the best score to state l at stage n and **optpred[n,l]** is the best predecessor state of state (n, l). (Here the trellis state of (n, l) means we are in state $l \in \{0, 1\}$ in the nth stage). For both of these arrays we take l=0 to mean the fair coin and l=1 to be the biased coin. The first line of the following fragment initializes the scores at the first level of the trellis using the matrix R[s,y] as the probability

of observing value y while in state s. The collection of three nested loops then, for each level, looks at each state. For this state we examine all possible predecessor and compute the score using each one. We remember both the optimizing score as well as which predecessor gives the optimizing score. This sort of fragment needs to be studied for a bit to understand, but it is straightforward implementation of the DP algorithm presented above.

```
for (s in 1:L) optscore[s,1] = p[s]*R[s,y[1]];  # p(x_1 = s)p(y_1 | x_1= s)
for (n in 2:N) {     # for each observation
  for (s in 1:L) {   # for each possible state
    for (r in 1:L) { # for each possible predecessor state
      z = optscore[r,n-1]*Q[r,s]*R[s,y[n]];  # optimal score if we come to s from r
      if (z > optscore[s,n]) {   # if best so far
        optscore[s,n] = z;
        optpred[s,n] = r;
      }
    }
  }
  optscore[,n] = optscore[,n] / sum(optscore[,n])  # rescale to avoid underflow
}
```

The traceback of the optimal path is identical to what we have seen before:

```
xhat[N] = which.max(optscore[,N])  # best final state
for (n in (N-1):1) xhat[n] = optpred[xhat[n+1],n+1];  # follow pointers back
```

Key Analysis with HMMs (key_hmm.r)

A harmonic analysis, as we define it here, labels each unit of musical time, say measure, with a key or (key,chord) pair. When we only try to label keys, then the idea is essentially to track key modulations throughout the music under consideration. Labeling with (key,chord) pairs in which, say, the chords are the seven basic triads built on the seven scale degrees, is a version of *functional* harmonic analysis, as practiced by music theorists. Either of these harmonic analyses can be performed using any musical unit (measure, beat, note etc.) as the unit to be labeled. In what follows we will give key labellings at the measure level, though a modification in the homework you will perform chord analysis for music that remains in a single key.

In discussing bag-of-notes models, we saw the difficulty in assigning a key *independently* to each measure in a piece of music. In essence, what appear to be different keys when looking at isolated measures can often be explained in terms of a single key when considering a larger context. For instance, imagine a sequence of measures in which the notes of the odd measures are all notes of the C Major triad, while the notes of the even measures all belong to the G Major triad. Labeling such music as an alternation between keys would not be reasonable, though any reasonable isolated measure analysis would recognize the music in this way. The idea we introduce here is to look for a reasonable *sequence* of keys, one for each measure, that also explains the data well. Thus, as we have done elsewhere, we look for a key sequence that is both plausible *a priori* and is also consistent with the observed pitch data.

How should a key sequence behave? Without training a transition model directly from data, one might assume

1. Most often, keys tend to persist for long sequences of measures
2. When we modulate (change keys), it is often either up or down a perfect 5th

3. Some modulations are quite rare, such as to neighboring half step keys, or by augmented 4th.

4. Other keys are somewhere in between in terms of likelihood

These are the assumptions that were expressed in Figure 3.5 in which the key sequence is modeled as a Markov chain. This model only considers major keys, though it is simple to extend it to include minor keys also. We will continue with the Markov-chain-on-major-keys assumption here, while acknowledging that it has certain weaknesses, such as not representing the high likelihood of ending up on the key where one begins.

The Markov model on key tells us how to represent *a priori* key sequence probability:

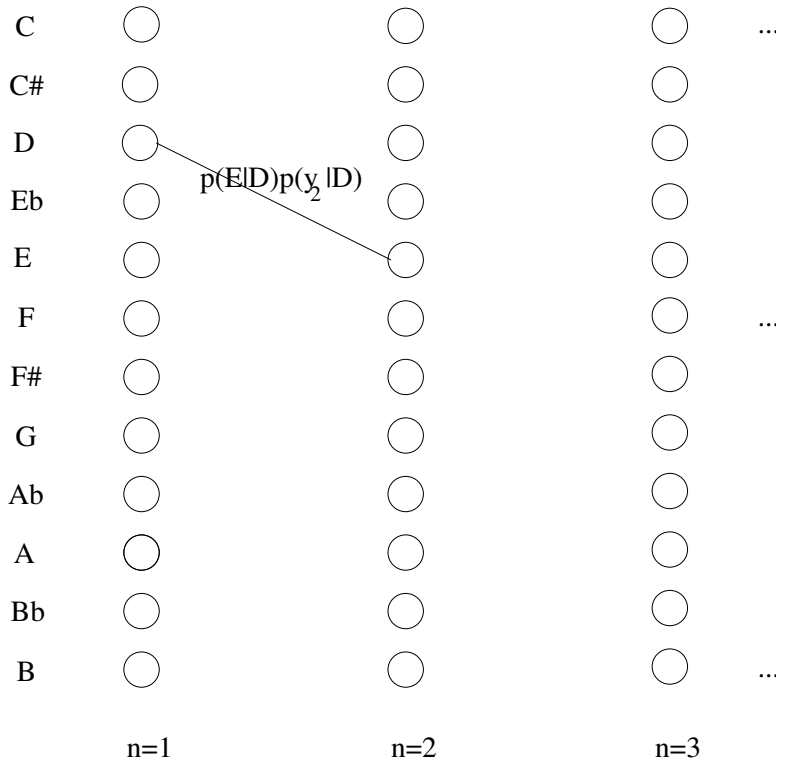
$$p(x) = p(x_1, \ldots, x_N) = p(x_1)p(x_2|x_1)\ldots p(x_N|x_{N-1})$$

using the transition probability matrix implicit in Figure 3.5. Our next task is to represent the data model, which describes the likelihood of the pitch data given a key sequence, x. To do this we assume that we have a bag-of-notes model for the pitches in each key, k, with probabilities $p(c|k) = p_0(c-k)$ where c is the pitch class and p_0 is the C major distribution and the subtraction is taken modulo 12 ($-3 \bmod 12 = 9$). Thus, if we let y_n be the collection of pitch classes in the nth measure, we model

$$p(y_n|x_n) = \prod_{c=0}^{11} p_0(c-k)^{\#\{y_n=c\}}$$

where by $\#\{y_n = c\}$ we mean the number of pitches in the nth measure belonging to pitch class c. This is exactly the model we used before in performing *piece-level* key estimation written out more compactly.

We now can construct the trellis graph in which the cost of each path $x_1 \ldots, x_N$ in the trellis is the product of arc scores it traverses. We have seen previously that if we let the score of the arc going from key x_{n-1} in measure $n-1$ to key x_n in measure n as $p(x_n|x_{n-1})p(y_n|x_n)$ then the product of the arcs traversed will be $p(x, y) = p(x_1, \ldots, x_N, y_1, \ldots, y_N)$. We have also seen that maximizing $p(x, y)$ when y is observed is the same as finding the most likely sequence, x, *given* y. To do this, we must create 12 states for the 12 possible keys at each of the N possible measures — a $12 \times N$ array. Then we make the score for going from key k in measure $n-1$ to key k' in measure n equal to $p(k'|k)p(y_n|k')$. We have indicated this in the following figure, though we have not drawn all the arcs between successive levels of the trellis for clarity's sake.



The program of **key_hmm.r** is a straightforward implementation of dynamic programming for this situation. The one small difference is that rather than maximizing the product of the arc scores along a path, we use the fact that

$$\log(p_1 \times \ldots \times p_N) = \log(p_1) + \ldots + \log(p_N)$$

This means that we can equivalently maximize the sum of the logs of the arc costs. This latter calculation is better for computer arithmetic since there is no problem with *underflow* (the problem of representing very small numbers), as there is with the straight product.

One can see from the examples in the **key_hmm.r** that the program tends to recognize long sequences of measures as single keys. Often this is appropriate and is a big improvement over our first attempt at labeling key movement. The example of Figure 3.7 shows that this kind analysis has significant weaknesses too. In this example we use the Chopin 9th Prelude in E Major, which is quite chromatic. The piece uses a wide variety of chords, and many of the unusual chords are seen as different keys in the analysis. This serves as motivation for the following section.

Functional Harmonic Analysis with HMMs

Functional Harmonic Analysis assigns a (key,chord) pair to every unit of music (measure, beat, note, etc.). For this the key is defined as a tonic in $\{C, C\sharp, \ldots, B\}$ together with a mode in $\{\text{Major}, \text{Minor}\}$. We write this as

$$\text{Key} \in \{C, C\sharp, \ldots, B\} \times \{\text{Major}, \text{Minor}\}$$

meaning that we take one from each of the two sets above. Of course it would be possible to allow a richer collection of keys that distinguishes between enharmonic equivalents since as $C\sharp$ and $D\flat$ — (a picky look our favorite Raindrop Prelude might say that, since the piece is in $D\flat$ Major, we should call the parallel

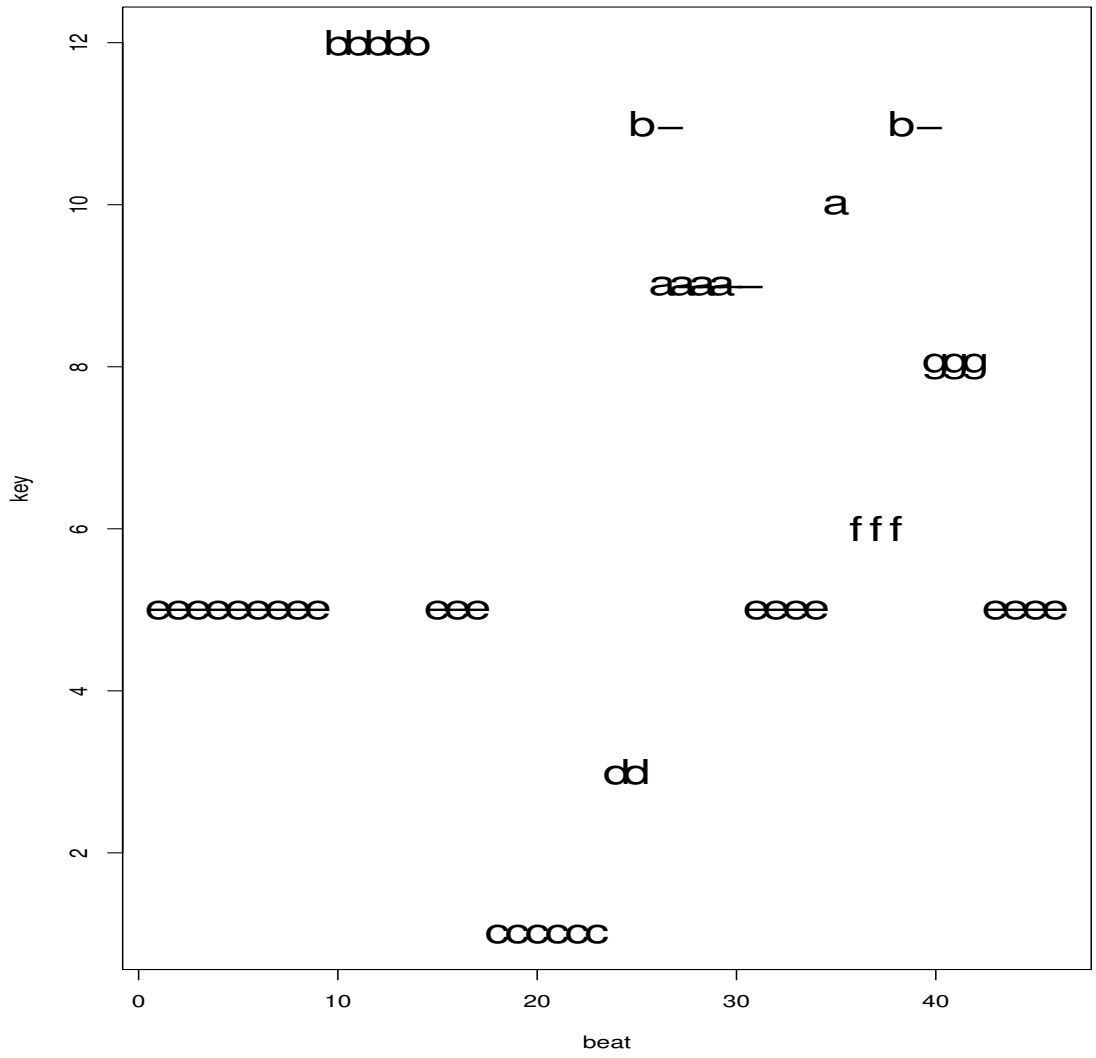
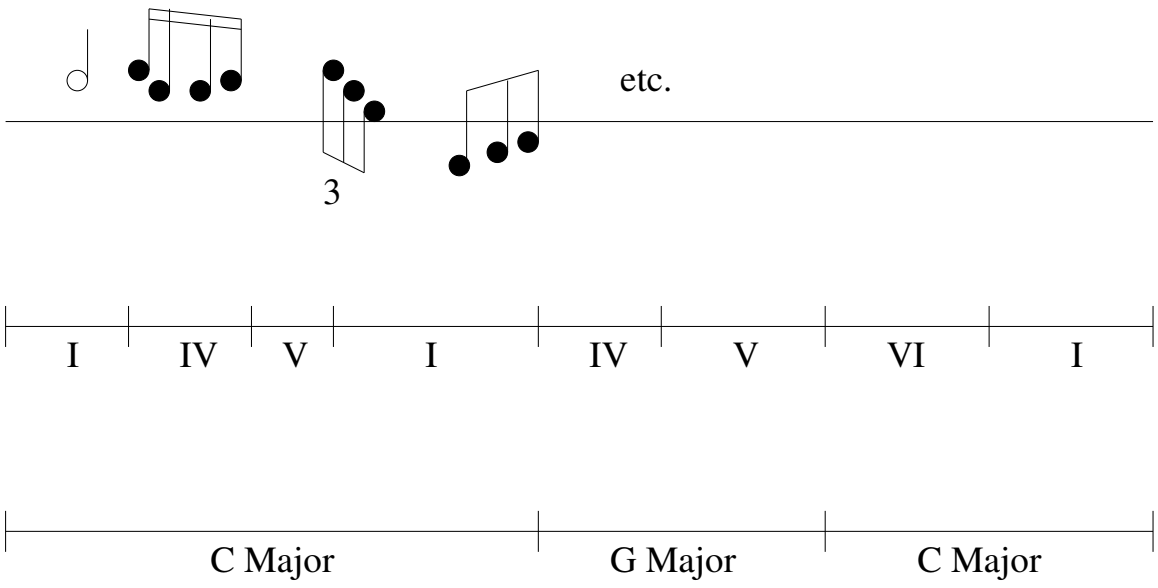


Figure 3.7: Key analysis of Chopin Prelude #9 at the *beat* level. "e-" means the key of E flat

minor of the middle section $D\flat$ Minor which has $B\flat\flat$ in the key signature). We may also allow a larger collection of modes, though we will not do so here.

Regardless of the mode under consideration, we will take $\{I, II, III, IV, V, VI, VII\}$ for the possible chords meaning the triads built on the first, second, etc. scale degrees. In our notation we will use all capital Roman numerals even for a minor triad. So, for instance, the I chord in C Major would be the triad C, E, G, while the V chord in D Minor would be $A, C\sharp, E$ (as a default we will use the "harmonic" minor scale in constructing our minor key triads with the flat 6th and raised 7th scale degrees). It is possible to create a much richer vocabulary or chord including dominant 7ths, Augmented 6ths, Neapolitan, and many others. One could even regard certain common suspensions as chords (such as V^{4-3}) though a theorist may not think of these as chords at all.

The goal of the analysis is to create a labeling of each musical unit with a (chord,key) pair. For example, the labeling might look like:



There are some necessary issues we still must address to flesh out this model into a state where it can actually be implemented. Since we will only show the *results* of this implementation (rather than its *coding*), we will keep this discussion at a rather high level, focusing on ideas rather than details.

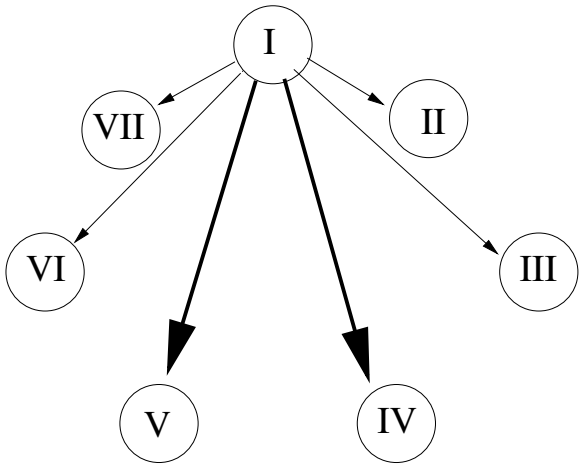
The data model, which describes the likelihood of the MIDI pitches in a measure, given the (key,chord) pair, can be accomplished rather simply. A (key,chord) pair specifies both a scale and a triad from the scale. For instance, (D Major, IV) specifies the D major scale: $D, E, F\sharp, G, A, B, C\sharp$ and the triad G, B, D. Every note that appears in the measure is now either

1. The tonic note
2. In the triad, but not the tonic
3. In the scale but not the triad
4. Outside the scale

We will assign four different probabilities to these four cases and represent the data model in terms of these probabilities. That is, if y_n is the nth measure,

$$p(y_n|x_n = (\text{key}, \text{chord})) = p_{\text{tonic}}^{\#\{y_n=\text{tonic}\}} p_{\text{triad}}^{\#\{y_n\in\text{triad -tonic}\}} p_{\text{scale}}^{\#\{y_n\in\text{scale-triad}\}} p_{\text{else}}^{\#\{y_n\notin\text{scale}\}}$$

To generate the Markov model on (key,chord) sequences, we assume that the key sequence is a simple Markov chain just as before. As long as the key remains constant we can assume the the chord also follows a Markov chain that is *independent* of the key we are in. This could be represented as something like:



In this figure we have represented the likelihood of the transitions out of the I chord by the thickness of the lines. Transitions would also need to be represented for the other chords, of course.

When we change from one key to another, we could choose the original chord from a fixed distribution that would favor the more basic chords such as I and V.

While we have chosen to represent only a few chords for every key, the model has the ability to capture notions such as secondary dominants, though perhaps not in the way a theorist might conceptualize them. For instance, if we have the sequence of chords: C Major, G Major, A Minor, D Major, G Major, C Major, this could be captured through the (key,chord) sequence: (C Major, I), (C Major, V), (C Major, VI), (G Major, V), (G Major, I), (C Major, I).

The course web page has a link to several harmonic analysis examples that were computed using this framework, represented as MIDI files. The midi files play the piano for the original music along with a "droning" chord for each (key,chord) that is recognized. In addition, if you have a MIDI player that supports text, such as a karaoke MIDI player, then the (key,chord) pairs will be written out when the appear in the music. We will demonstrate this in class.

Rhythm Recognition with HMMs (rhythm_hmm.r)

Score-writing programs allow a user to enter music by typing in notes, rhythms, and various other music symbols, much like in a word processor. Some programs also allow the user to enter the music by *playing* it on a MIDI keyboard attached to the computer. For users with good keyboard skills this can be a more efficient way of entering music data — sometimes much more so. The MIDI data conveys pitch in an

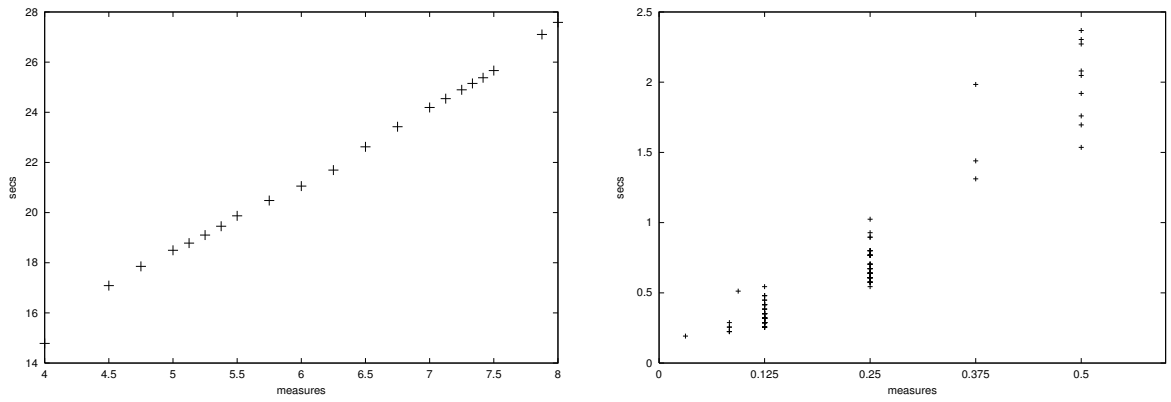


Figure 3.8: The **left** panel shows musical time vs. real time for a performance of the opening of R. Schumann's 3rd Romance for oboe and piano. The **right** panel shows the notated musical length of each note plotted vs. the note inter-onset intervals (IOIs). As the the vertical overlap of these categories demonstrates, it can be quite difficult to "quantize" the actual timing to recover musical timing.

unambiguous manner (except for the spelling of notes), however the rhythmic interpretation of the MIDI data is often highly ambiguous here.

We model the problem as follows. Suppose we are given a sequence of times at which note onsets occur: $t_1, t_2, \ldots, t_N$, given in seconds. We wish to transcribe these data into rhythm, as one would notate in a music score. In addition to the score-writing program scenario, this rhythm recognition problem also comes up when one wishes to transcribe *sampled audio* into music notation. This "audio signal to score" problem, though still in an early stage, could be useful when one wants to notate music that exists *only* in audio form, or for which one doesn't have a score. While this problem also requires us to identify note events (pitches and times) from the audio, it involves rhythm recognition as well.

Why is this problem difficult? As with all music performance data, one shouldn't expect strict adherence to the formula presented in the musical score. Such metronomic data would neither be possible nor desirable for a human to create. If the tempo of the piece is known, then each rhythmic interval (e.g. half note) corresponds to an actual time between onsets. In this case one may try to *quantize* the data by rounding to the nearest musical unit. One problem with this quantization approach is that we can always find a musical unit that is as close as we like to the observed interval. For instance, if the tempo is 60 quarters per minute and the inter-onset interval (IOI) is .531, a natural choice would be to call this IOI an eighth note. However, if we were to classify the IOI as an eighth note tied to a 1/128th note this would correspond to .53125 secs. — in better agreement to the observed time. Furthermore, if we consider progressively more exotic musical units we could come as close as we like to the observed IOI. Clearly this would be undesirable, so the moral is that we care about *more* than representing the observed timing accurately. Labeling the note as an eighth note is a *simple* explanation that describes the data reasonably well. It makes musical sense to prefer simpler explanations over more complex ones, since more complicated rhythms involving 1/128th notes and other such oddities are quite rare. This is the idea of *Ockham's Razor* which states that simple hypotheses are, in general, preferable to complex ones, so our analysis should be biased *toward* these simple explanations. One could formulate this idea from a statistical point of view in which a *prior distribution* over musical time intervals, estimated from score data, would find simpler intervals more likely than complex ones.

It is worth noting the the quantization problem may not be solvable by restricting one's attention to the simple musical note lengths. Figure 3.8 shows the musical timing for an excerpt from R. Schumann's *3rd Romance for Oboe and Piano*. In the figure, the left panel shows the musical time plotted vs. real

time, as we have seen in previous examples. The right panel of the figure plots the musical length of each note, in measures, on the x-axis with the actual IOI, in seconds, on the y-axis. Thus, since the piece is in 4/4 time, the collection of notes corresponding to the musical time of .125 are all eighth notes. It is clear that there is considerable variation to the IOIs for each length category and, for example, that some eighths are longer than some quarter notes. This example is not unusual for *rubato* playing and further demonstrates the weakness of trying to assign rhythm by quantizing.

Before taking this problem further, we note some weaknesses with the formulation. Musical rhythm is, of course, not just a function of timing, though timing is certainly central. Pitch figures into rhythm perception in several ways. For instance, we are inclined to perceived certain kinds of pitch-related events on strong metric positions, such as chord changes and simultaneities. In addition, we often use pitch in assigning groupings in music. Thus, a similar pitch configuration heard at different places in a musical excerpt is often perceived as occurring in the same position within the metric hierarchy. For instance, in a sequence of rising 8 note arpeggios (think Bach C Major Prelude) we are strongly inclined to hear the lowest notes of each grouping as in lying in the same metric place. We will not take advantage of this pitch information in looking at rhythm recognition, though a more sophisticated analysis might do so. We will also ignore the dynamic level of each onset, which may be part of perceived accent, also figuring into perceived rhythm.

Still, there is quite a bit we can do with the musical element we *do* model — timing. Most importantly, since we look at the entire sequence of note onset times, rather than isolated IOIs, we can model the likelihood of certain rhythmic *sequences*. This is more powerful than isolated IOI recognition since context is so important in expressing the plausibility of a rhythmic sequence.

We will model rhythm in the context of a known meter. Within this meter we will define a collection of possible musical onset positions within the measure. For instance, if we believe we are in 4/4 time and the only possible metric position lie on the eighth note positions then our *state space* would be $\Omega = \{\frac{0}{1}, \frac{1}{8}, \frac{1}{4}, \frac{3}{8}, \frac{1}{2}, \frac{5}{8}, \frac{3}{4}, \frac{7}{8}\}$. We will model the sequence of metric positions, $x_1, \ldots, x_N$ corresponding to our observed times $t_1, \ldots, t_N$ as a Markov chain with state space Ω. This allows us to capture many short- and long-range notions of musical rhythm, as follows.

We may believe that, in the music under consideration, syncopations are rare. Thus weak metric positions such as 3/8 will tend to be followed by neighboring stronger positions. We could capture this notion by requiring the transition probabilities from 3/8 to favor 1/2 as the next metric position. Similarly we may wish to discourage tying over the bar line and could reflect this in our choice of transition probabilities by assuming a small probability for any any transition that moves over the barline while not "landing on" the downbeat. Finally, we may have a state space Ω reflecting various possible subdivisions of the beat. For instance, if we allow both triple and duple subdivision of the beat in 2/4 time, then our state space would be $\Omega = \{\frac{0}{1}, \frac{1}{12}, \frac{1}{8}, \frac{1}{6}, \frac{1}{4}, \frac{1}{3}, \frac{3}{8}, \frac{5}{12}\}$ In such a situation it would be likely for the 2nd note of the first quarter note triplet, 1/12, to be followedby another triplet position such as $\{\frac{1}{12}, \frac{1}{6}, \frac{1}{4}, \frac{1}{3}, \frac{5}{12}, \frac{0}{1}\}$. The preceding ideas could be used to further refine the likelihood of the next measure position with 1/12 quite unlikely, as a tie over the bar line, and 1/6 more likely as the next triplet pulse.

Our knowledge about metric structure, as captured by a Markov model on Ω, also captures longer-term musical information. For instance, we expect that long notes are more likely to occur at downbeats, which is partly captured by discouraging ties across the bar line. Thus we are more likely to recognize the occasional long note as on a downbeat or other strong metric position, when other interpretations are consistent with the data. This embeds further musical knowledge into our model.

How do we set the transition probabilities for this model? Of course, we may wish to *train* the model by observing actual transition counts from music data that is similar to the type we try to recognize. We assume that, either by training or hand-specifying, we have a Markov model represented by the $L \times L$ probability transition matrix, Q, when our state space, Ω, has L elements. We suppose we also have an initial probability distribution $p(\omega)$ telling us the distribution from which we select our first metric position

x_1.

Now we consider the problem of relating our *observed* times $t_1, \ldots, t_N$ to our *unobserved* state sequence $x_1, \ldots, x_N$. Suppose that our tempo, τ, is given in quarters per minute. If we have a transition from x_n to x_{n+1} with $x_n, x_{n+1} \in \Omega$, the elapsed musical time, $E(x_n, x_{n+1})$ would be

$$E(x_n, x_{n+1}) = \begin{cases} x_{n+1} - x_n & \text{if } x_{n+1} > x_n \\ B/4 + x_{n+1} - x_n & \text{otherwise} \end{cases}$$

where B is the number of quarters per measure. Thus, in 3/4 time, $E(1/4, 1/2) = 1/4$ while $E(1/2, 1/4) = 3/4 + 1/2 - 1/4 = 2/4$. Now we would expect the transition x_n, x_{n+1} to have an IOI, in seconds, of

$$\mu(x_n, x_{n+1}, \tau) = \frac{4 \times 60 \times E(x_n, x_{n+1})}{\tau}$$

We don't expect the associated IOI, $t_{n+1} - t_n$, to be *exactly* $\mu(x_n, x_{n+1}, T)$ seconds, of course, so we model it as

$$t_{n+1} - t_n \sim N(\mu(x_n, x_{n+1}, \tau), \sigma^2)$$

Thus we assume that each IOI is *normally* distributed with mean as predicted from the note length and tempo, but with some random variation. We choose the normal distribution simply because it has been already introduced and is familiar at this point. Better modeling choices might reflect asymmetries in note length distribution (note stretches may be more common than "compressions"), We might also model the notion that the observation variance, σ^2, increases with note length rather than being constant.

What we have now is a hidden Markov model that relates the hidden Markov chain describing the rhythmic interpretation we seek to the observed IOIs, $y_n = t_n - t_{n-1}$. The R program **rhythm_hmm.r** carries out the usual dynamic programming recognition strategy on a time sequence from an actual performance. In this example the transition probability matrix, Q, was assigned by hand for 4/4 music using the guidelines discussed above.

As with our other HMM examples, we interpret the data as the most likely path through a trellis graph, given our data. For the 4/4 case discussed above in which only notes lying on eighth note boundaries are possible, the graph would look like that in Figure 3.9. The graph allows all possible connections between adjacent trellis levels, meaning that any note position can be followed by any other. The score for moving from state x_n at onset n to state x_{n+1} at onset $n+1$ would be $Q(x_n, x_{n+1}) \times N(t_{n+1} - t_n; \mu(x_n, x_{n+1}, \tau), \sigma^2)$, as discussed above. Most importantly, the arc score tries to balance the faithfulness to the observed IOIs with finding a reasonable rhythmic interpretation of what is played. In the implementation in the program we have allowed σ^2 to increase with the expected note length. Also, we estimate the overall tempo as the ratio of the *known* number of beats to the overall time of the excerpt in minutes.

To get the performance data, we have used a program that allows a person to tap in a sequence of times corresponding to a musical rhythm. An example of the output of the program using a sequence of times for "Once in Royal David's City" is given below. To make the raw data more intelligible, I have printed out the IOIs (in seconds) divided by the tempo (in beats/sec) giving the IOIs in beats. In an ideal scenario, these numbers would all be integers or "half integers."

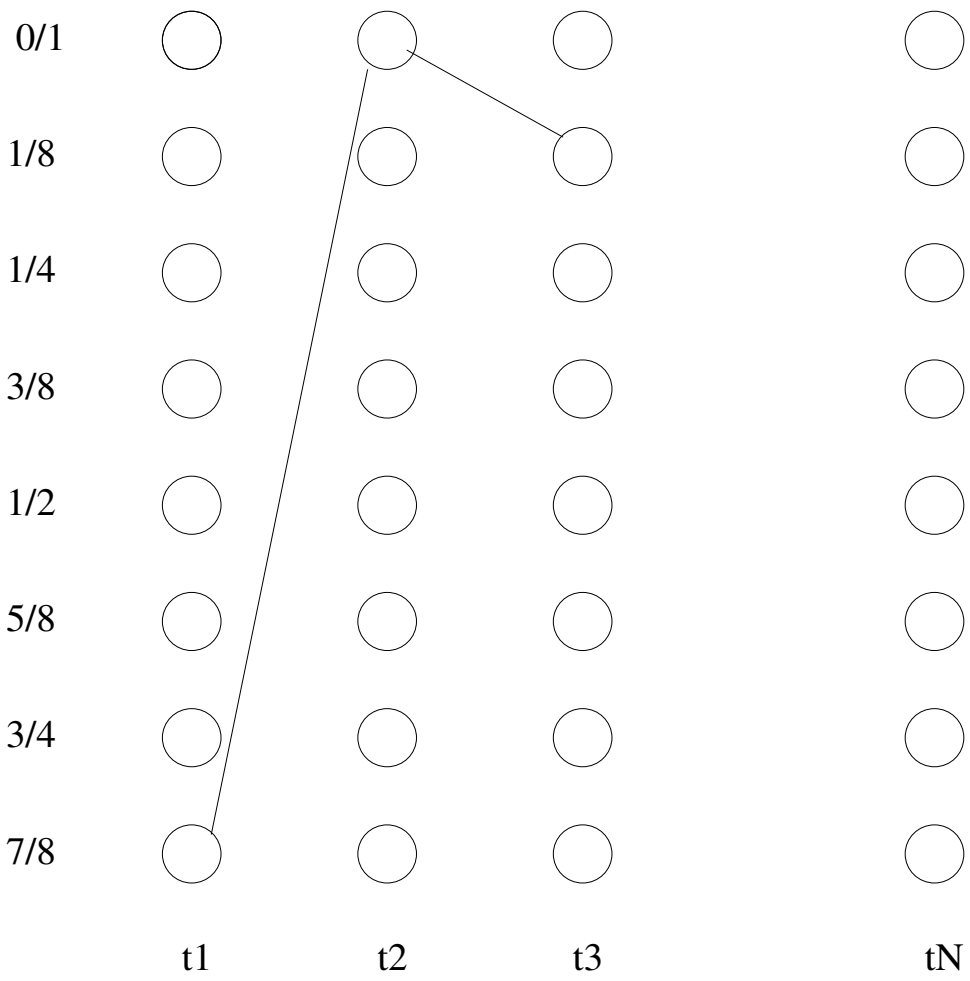


Figure 3.9: The trellis graph for rhythm recognition. The possible states of the graph correspond to rhythmic position within a measure and are represented by *rows* of the figure. The first column gives the states we may find the performance in for the first onset time, the 2nd column gives the states we may find the performance in for the 2nd onset time, etc. We use dynamic programming to find the most likely path through this trellis, with the arc scores as described.

2.1982301	2.1931416	2.9818584	0.9286504	0.8752212	0.8777655	0.9693584
1.0253319	1.8751106	1.9717920	2.0659292	2.1091814	3.1548673	0.8625000
0.9133850	0.8574115	0.9540929	1.0609513	4.2056416	2.0252212	2.1066372
3.0480088	0.9362832	0.8777680	0.9184735	0.9362832	0.9464602	2.1040904
1.9463496	1.9488938	2.1448034	2.8877162	0.9413742	0.8955727	0.8599608
0.9057471	1.0558654	4.3506612	1.8674830	2.1626055	3.0556416	1.0049804
1.9717895	2.0939210	4.3532054	1.9539848	1.9768831	2.9106118	0.8879501
0.9566346	0.8981118	0.8574166	1.1321903			

The recognized sequence of measure positions is given by the our program as follows. In each pair of rows the first is the numerators for a measure while the second is the denominators.

1	3
2	4

0	3	1	5	3	7
1	8	2	8	4	8

0	1	1	3
1	4	2	4

0	3	1	5	3	7
1	8	2	8	4	8

0	1	3
1	2	4

0	3	1	5	3	7
1	8	2	8	4	8

0	1	1	3
1	4	2	4

0	3	1	5	3	7
1	8	2	8	4	8

0	1	3
1	2	4

0	3	1	3
1	8	2	4

0	1	3
1	2	4

0	3	1	5	3	7
1	8	2	8	4	8

0
1

While this example was recognized completely correctly, it is perhaps not too surprising due to the way in which the obvious quantization of the data is correct nearly all of the time. Many of the examples seem to contain mistakes, however.

An interesting attribute of the program is its ability to put rhythms in reasonable places. For instance, the song "Oh Hannukah, O Hannukah" begins with an eighth note upbeat. One could conceivably transcribe the rhythm with this upbeat occurring at the start of the measure, effectively sliding the whole transcription forward one eighth note. Our data (output) model makes no distinction between these two interpretations, since it is based only on the musical note lengths and not where they appear in the measure. On the other hand, the prior model embodied by Q sees the syncopation-filled rhythm corresponding

to this incorrect interpretation as unlikely. This example is also shown below.

0.9955426	0.9518785	0.9737105	0.8820158	0.9955426	0.9256800	0.9213136
0.9082143	0.9999090	2.0085509	0.9518785	1.0654053	2.0347494	2.1308105
1.9255890	0.8732830	1.0217411	2.0303830	0.8863822	1.0086419	1.9037569
0.9562449	1.0173747	2.0129173	2.0522150	0.8907487	0.8820158	0.9475121
0.9999090	1.8469935	0.9300464	0.9911762	2.0216501	0.8252524	1.1396343
4.0913308						

3	1	5	3	7		
8	2	8	4	8		

0	1	1	3	1	3	7
1	8	4	8	2	4	8

0	1	1	3	7	
1	4	2	4	8	

0	1	3	1	3	7
1	4	8	2	4	8

0	1	1	5	3	7
1	4	2	8	4	8

0	1	3	1	3	7
1	4	8	2	4	8

0	1
1	2

Pitch Spelling

Many have had the experience of reading a MIDI file into a score-writing program and finding that the spellings of the notes chosen by the program are not the same as those chosen by the composer. This is not terribly surprising since MIDI gives no way of distinguishing between enharmonic spellings, representing, for instance, both middle C♯ and D♭ as MIDI pitch 61. This seems like a reasonable choice on the part of the MIDI designers since MIDI was never meant to serve as a music score representation. However, MIDI has become a *de facto* standard, since it is, by far, the most common electronic score format. Thus one occasionally faces the problem of constructing a musical score from a MIDI file.

Pitch spelling is the problem of making the appropriate choice of sharp, flat, natural, double sharp or double flat when this "spelling" information is not available. One obvious use of the problem was mentioned above, though, there is a pitch spelling component to *any* transcription problem including the

audio-to-score version. A pitch spelling algorithm may also be used like a regular text-based spell checker, and could alert a user to possible wrong notes or questionable spelling choices.

A possible approach to pitch spelling problem is to cast it as a *classification* problem. For example, given a MIDI 67 use the collection of neighboring pitches to decide if the note is G, F♯♯ or A♭♭. There is a big weakness with this kind of approach, however. There is a relatively simple logic that governs the spelling pitches. If we understand how this logic works in the key of C Major, we should be equally equipped in D♭ Major. Treating the problem one of simple classification blurs this commonality. A better approach builds some simple musical knowledge into the strategy.

The key to performing pitch spelling is to recognize a hidden tonal assumption that explains what we see on the page. According to this assumption, each note of tonal music can be thought of as a scale degree with possible modifications of ♯ and ♭ for some of the pitches, in the context of a key. For instance, in the case of major mode, the possible scale degrees would be

$$\Delta_{\text{major}} = \{\hat{1}, \sharp\hat{1}, \flat\hat{2}, \hat{2}, \sharp\hat{2}, \flat\hat{3}, \hat{3}, \hat{4}, \sharp\hat{4}, \flat\hat{5}, \hat{5}, \sharp\hat{5}, \flat\hat{6}, \hat{6}, \sharp\hat{6}, \flat\hat{7}, \hat{7}\}$$

Observe that notes in the scale can *only* be represented as the corresponding scale degree in $\{\hat{1}, \ldots, \hat{7}\}$. Also observe that the out-of-scale notes each have only two possible enharmonic representations, such as $\sharp\hat{1}$ and $\flat\hat{2}$, corresponding, for example, to C♯ or D♭ in the key of C Major.

When the key is known, there is a simple way to relate this scale degree to the actual pitch spelling. For instance, suppose we are in E Major which has the scale: E, F♯, G♯, A, B, C♯, D♯. If scale degree $\hat{2}$ occurs, we would write the 2th note of the E major scale which is F♯. If, instead, the scale degree is $\flat\hat{2}$, then the ♯ of the 2nd scale tone, F♯, is "undone" by the ♭ of $\flat\hat{2}$, leaving us with simply F. On the other hand, if the scale degree is $\sharp\hat{2}$, we write *F*♯♯ since the 2nd scale degree already has a sharp. (We write the double sharp as ♯♯ because we have no way to get the usual double sharp symbol in our document). Note that pitches in the diatonic scale are only allowed a single spelling, so, for instance, C♭ simply cannot exist in the key of C Major since we do not have a scale degree $\flat\hat{1}$. Whether or not $\flat\hat{1}$ *actually* exists may be a subject of debate, though we have chosen to model this as an impossibility.

The actual accidentals that appear in the page will depend, of course, on the *notated* key signature. For instance, if our algorithm gives a note spelling as F♯, we only write the sharp if the key signature does not already have F♯. Similarly, we would use the ♮, ♯♯ or ♭♭ signs when we need to *overrule* something in the key signature. Thus $\flat\hat{2}$ with 4 sharps in the key signature would be written as *F*♮.

The minor scale works in a similar fashion. Here we take the *natural minor* as the basic scale, which uses the same scale tones as the relative major. For instance, A minor is the relative minor of C major and both use the white keys of the keyboard. For the minor key we allow the scale degrees

$$\Delta_{\text{minor}} = \{\hat{1}, \sharp\hat{1}, \flat\hat{2}, \hat{2}, \hat{3}, \sharp\hat{3}, \flat\hat{4}, \hat{4}, \sharp\hat{4}, \flat\hat{5}, \hat{5}, \hat{6}, \sharp\hat{6}, \hat{7}, \sharp\hat{7}\}$$

In the natural minor the half steps occur at different positions in the scale, leading to a different collection of scale degrees involving raised and lowered notes. However, in minor we have chosen *not* to allow two situations that may be considered to "make sense," accounting for the fact that $|\Delta_{\text{minor}}| = 15$ $|\Delta_{\text{major}}| = 17$ and In particular, we do not allow $\flat\hat{7}$, since the 7th scale degree is already lowered in the natural minor, nor do we allow $\flat\hat{1}$, since the usual notational convention would be to avoid the flatted first scale degree in either mode.

Our state space for each note, now including the key, can now be represented as

$$\begin{aligned} \Omega &= \Delta_{\text{major}} \times \{\text{C}, \text{D}\flat, \text{D}, \text{E}\flat, \text{E}, \text{F}, \text{F}\sharp, \text{G}, \text{A}\flat, \text{A}, \text{B}\flat, \text{B}\} \\ &\bigcup \Delta_{\text{minor}} \times \{\text{A}, \text{B}\flat, \text{B}, \text{C}, \text{C}\sharp, \text{D}, \text{E}\flat, \text{E}, \text{F}, \text{F}\sharp, \text{G}, \text{G}\sharp\} \end{aligned}$$

where by this notation we mean that a state can be *either* one of the major scale degrees with one of the listed major keys, *or* one of the minor scale degrees with one of the listed minor keys. In listing the major and minor keys we have chosen the tonic names that lead to the key signatures with the fewest number of sharps or flats in the key signature. It would be fine, in principle, to consider C♯ Minor and D♭ minor as two different keys, and enlarge the possible keys this way according the the "*line* of fifths" (as opposed to the circle of fifths). However, our favorite example of the Chopin *Raindrop Prelude* is one of the many cases in which the notationally-simpler key signature is preferred over the more logical and complex one. That is, the middle section of the *Raindrop Prelude* is in the parallel minor of the basic key of *D*♭ Major, which might logically be called D♭ minor, though it is notated instead as C♯ minor.

We consider now the situation in which we try to spell a single voice of music with MIDI pitches $m_1, \ldots, m_N$. Our possible labellings are of the form $x_1, \ldots, x_N$ where $x_n \in \Omega$. We assign several simple rules for computing the cost of a label sequence.

1. We only allow labelings that makes sense, implemented in the following way: If a label x_n and its corresponding MIDI pitch are inconsistent, the label gets a cost of ∞. For instance $x = \hat{2}$ is D in C Minor and would only be consistent with m_n if $m_n \bmod 12 = 2$

2. If a label is consistent with the MIDI pitch we penalize the label as C_1, C_2, C_3 or C_4 as it is

   (a) C_1: in the tonic chord

   (b) C_2: not in the tonic chord but in the tonic scale

   (c) C_3: one of the "likely" accidentals of $\sharp\hat{4}$ or $\flat\hat{7}$ in major or $\sharp\hat{7}$, $\sharp\hat{6}$, or $\flat\hat{2}$ in minor. For the major key, these are the new accidentals that appear in the neighboring keys on the circle of fifths. The choice of the favored minor accidentals is a little harder to justify in a principled way. We chose $\sharp\hat{7}$ from the usually major dominant chord in a minor key, $\sharp\hat{6}$ since it appears frequency in minor due to the ambiguity of the scale, and $\flat\hat{2}$ since it is more likely than $\sharp\hat{1}$ from the Neapolitan chord.

   (d) C_4: the remaining out-of-scale scale degrees.

   These penalties will have $C_1 < C_2 < C_3 < C_4$. Thus we "like" hypotheses that see notes as in the tonic chord and don't like ones that see notes as out-of-scale, especially those from distant keys.

3. Changes between keys are only allowed at measure boundaries. Changes of key always assume a cost C_5.

4. We will reward transitions that move from scale degree $\sharp\hat{n}$ to $\hat{n}+1$ and similarly from $\flat\hat{n}$ to $\hat{n}-1$ with the obvious "wrap-around" interpretation of addition and subtraction. This reward will will be $-C_6$.

Observe that the cost of a label sequence $x_1, \ldots, x_N$ depends only on the adjacent label pairs

$$(x_1, x_2), (x_2, x_3), (x_3, x_4), \ldots, (x_{N-1}, x_N)$$

thus the problem lends itself naturally to dynamic programming. We build the usual DP trellis by enumerating all of the possible states of Ω in a column with one column for each of the N midi pitches in the excerpt. We find the best scoring path according to the usual DP recursion and trace back to find the best state sequence.

The following examples make a couple of simple modifications on this basic algorithm. First, it is wasteful to create states that are inconsistent with the current MIDI pitch since these states receive a cost of ∞ and cannot possibly be chosen by the DP algorithm. We did not create such states. Also, in

the examples we treated the spelling of a single monophonic line *with knowledge* of the "accompaniment" notes in the other parts. To incorporate this knowledge, we also scored each of the accompaniment notes according to its lowest cost spelling in the key indicated by our state. That is, if we are in the key of C Major and MIDI 61 appears, we would choose the lower of the two scores given by $\sharp\hat{1}$ and $\flat\hat{2}$.

The first example treats the first 50-or-so measures of the Mendelssohn violin concerto in E minor (though the movement is in C Major). The spellings given by our algorithm are indicated in Figure 3.10 and are in complete agreement with our score. The second example is an excerpt from the first movement of the same piece shown in Figure 3.11. This example shows a few mistakes marked as diamonds over the incorrect spellings. While the mistakes are all flats chosen where sharps belong, not all of the flatted scale degrees recognized by our algorithm are mistaken. The notes marked as $F\natural$ are, of course, mostly recognized as $\flat\hat{2}$ in E minor. While we don't include the recognized keys here, it is worth mentioning that they are not particularly accurate. Luckily, reasonable spellings from one key are often reasonable from a nearby key, so the algorithm is not particularly sensitive to key mistakes.

Rhythm Recognition with Unknown Tempo (rhythm_rec_sim.r)

Our last version of rhythm recognition assumed that the tempo was known and constant. While this may be a reasonable assumption in some cases, such as when the player's goal is to communicate the rhythm accurately and not to play expressively, it is certainly not reasonable in many musical scenarios. When music data are sampled *in vivo* (a real musical setting) the tempo is generally unknown *a priori* and may vary through the performance. This leads to what might be called a "chicken and egg" problem. That is

1. Without knowing the rhythmic values assigned to the notes, we will not know the number of beats that elapse over any segment of the music. Thus, we cannot estimate the tempo without knowing the rhythm.

2. Without knowing the tempo we cannot say how long any note (e.g. quarter, eighth, etc.) should last. Thus, we cannot estimate the rhythm without knowing the tempo.

It seems that it is nearly impossible to separate rhythm and tempo estimation into two separate problems. This doesn't mean there is nothing we can do, however: clearly the human manages this problem without a great deal of difficulty. The key here is *not* to look at rhythm and tempo as two separate problems, but rather we estimate them *simultaneously*. To do this, we must first create a model that represents them simultaneously.

We have already discussed the modeling of rhythmic sequences in our earlier attempt at rhythm recognition. We will treat rhythm exactly as before, by assuming that our measure positions, $x_1, \ldots, x_N$ live in a known collection of possible onset positions (in musical units), $\Omega = \{\omega_1, \ldots, \omega_L\}$. We model this sequence as a Markov chain.

How to model the tempo? We will use a *random walk* model for tempo, which assumes that the sequence of tempi, $t_1, t_2, \ldots, t_N$ are modeled by

$$t_{n+1} = t_n + e_n$$

where the *increments* $\{e_n\}$ are assumed to be independent, small, and centered around 0. Essentially, this model says the next tempo differs from the current one by something that is small. Thus, if let $P(e_n = 1) = P(e_n = -1) = 1/2$ we would have a discrete random walk. However, we could also model the increments by a continuous distribution such as $e_n \sim N(0, \sigma^2)$. A random walk for tempo captures the notation that the tempo could change at any time, but allows an unrealistic amount of tempo flexibility. We use the model because it is simple to implement. To be specific, will assume there are a finite collection of possible tempi represented in seconds/beat. One could think of these are corresponding to the markings on an old-style metronome, listed here as $\{a_1, a_2, \ldots, a_K\}$. For instance these possible tempi could be

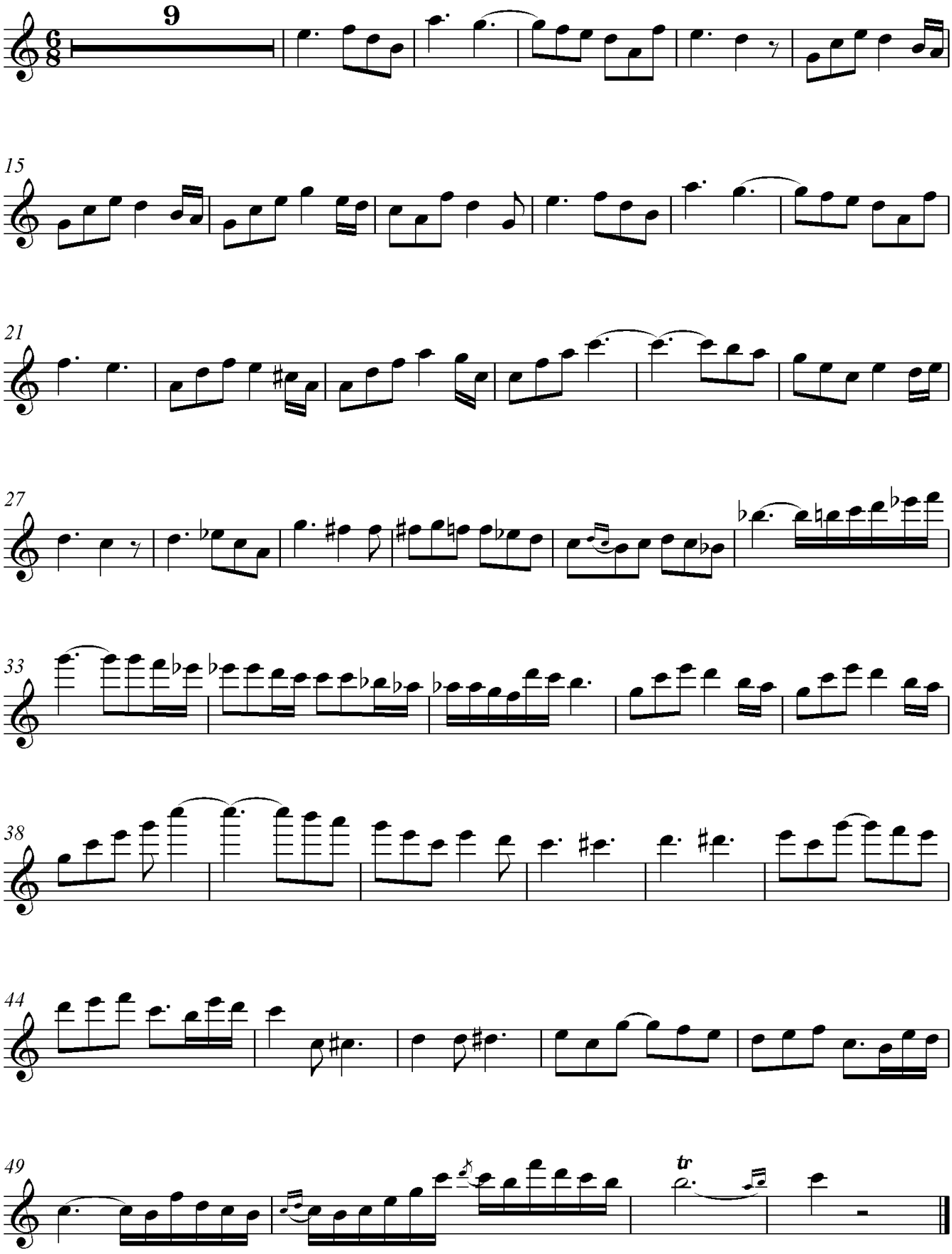


Figure 3.10: Pitch spelling for the opening of the 2nd movement of the Mendelssohn violin concerto.

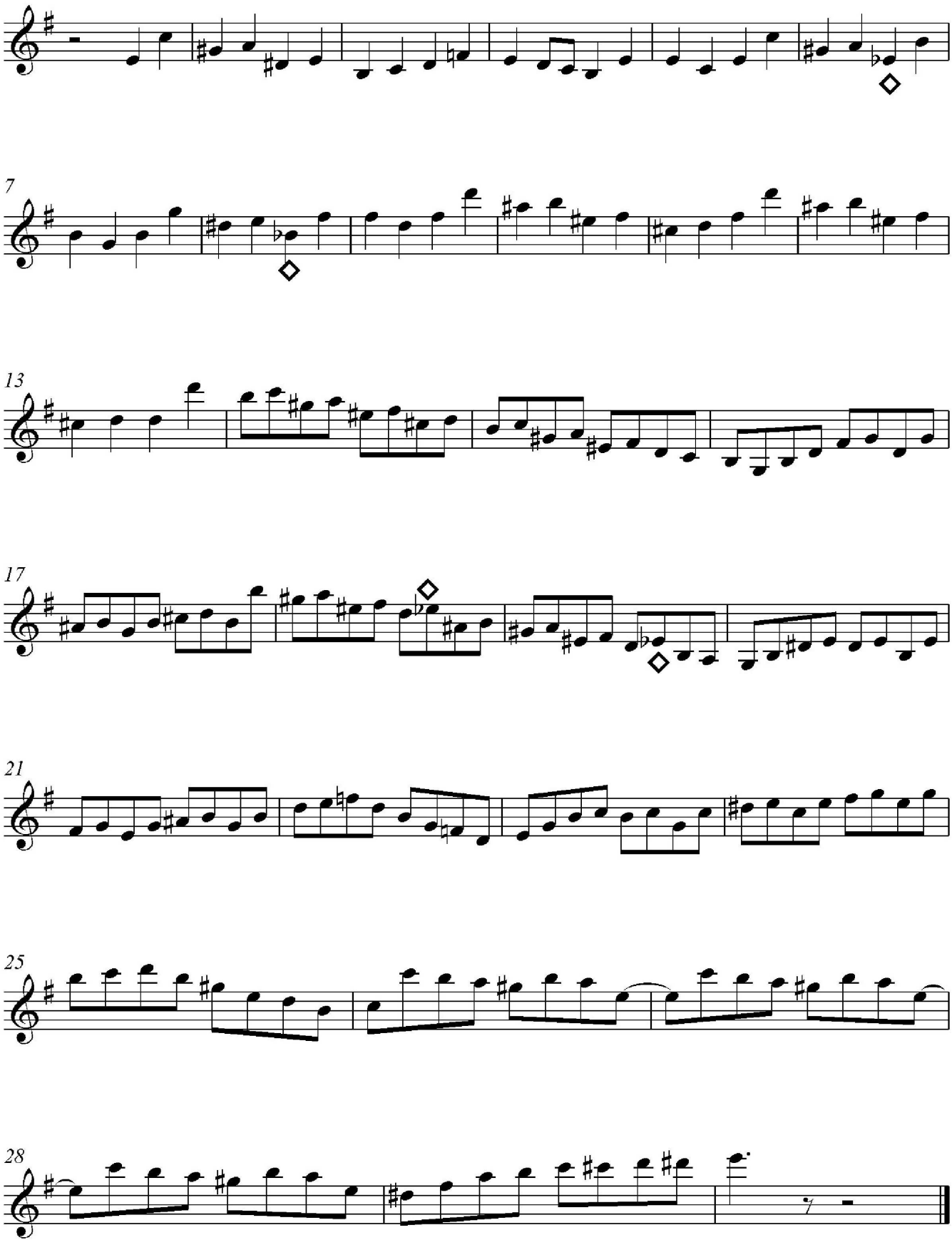


Figure 3.11: Pitch spelling for an excerpt from the 1st movement of the Mendelssohn violin concerto.

$\{\underbrace{.30}_{\text{fast}}, .31, .32 \ldots, \underbrace{2.}_{\text{slow}}\}$. Note that since the units are seconds per beat, the smaller numbers correspond to the faster tempi. This range covers a wider range of tempi than one would likely face in any realistic setting. We will model the tempo sequence $\{t_n\}$ by assuming that the tempo of the next note can be either the most recent tempo or either of its neighbors (if they exist), all with equal probability. That is

$$P(t_{n+1} = a_{k'} | t_n = a_k) = \begin{cases} 1/3 & |k - k'| \leq 1 \text{ and } k \notin \{1, K\} \\ 1/2 & |k - k'| \leq 1 \text{ and } k \in \{1, K\} \\ 0 & \text{otherwise} \end{cases}$$

Since we won't observe the tempo process directly, we don't know what the values of $t_1, \ldots, t_N$ are. But we can still relate these to the observed onset times as if they were known. We follow ideas from our previous rhythm recognition experiments. Let $E(x_n, x_{n+1})$ be the elapse musical time in going from measure position x_n to x_{n+1}. We have, from the previous discussion,

$$E(x_n, x_{n+1}) = \begin{cases} x_{n+1} - x_n & \text{if } x_{n+1} > x_n \\ B/4 + x_{n+1} - x_n & \text{if } x_{n+1} < x_n \end{cases}$$

This assumes, as before, that we do not allow notes to last longer than one measure, so that the note lengths can be computed from the measure positions. (Of course, the basic unit we treat could be longer than a measure if this assumption is not true).

For this model to be useful in recognition, we must relate hidden tempo and rhythm sequences to the data we observe. We will denote the inter-onset times (IOIs) as $y_1, \ldots, y_N$ and model these as

$$y_n \sim N(\mu = t_n E(x_{n-1}, x_n), \sigma^2)$$

Thus the mean or average value of an IOI is the time that would be predicted by knowing the tempo and note length plus some some 0-mean normal random "error."

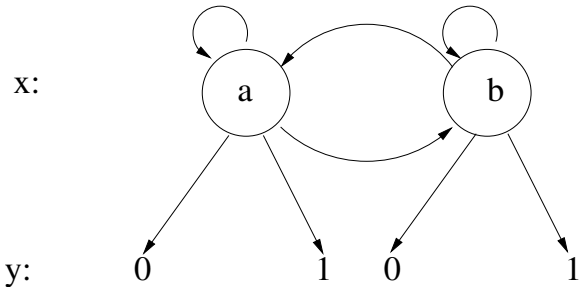
Now we look at the problem of trying to recover the hidden sequence of states (t_n, x_n) pairs, from our observed IOIs $y_1, \ldots, y_N$. At first glance, it might appear that, due to the two-dimensional nature of the hidden state, this model is fundamentally different than previous HMMs we have studied and not amenable to the same types of recognition strategies. That the state is really made of two numbers need not bother us. We simply enumerate the possible states as $\{\omega_1, \ldots, \omega_L\} \times \{a_1, \ldots, a_K\}$ and define the appropriate transition probability matrix for these states. The discussion above implicitly assumes the tempo and measure position processes are independent of one another, thus

$$P((t_{n+1}, x_{n+1}) | (t_n, x_n)) = P(t_{n+1} | t_n) P(x_{n+1} | x_n)$$

The program **rhythm_rec_sim.r** performs rhythm recognition for this model. In substance, this program is very close to ones we have studied earlier that compute most likely paths using dynamic programming. The one wrinkle here is the two-dimensional state. We handle this here with a function that convert converts a state in $\{1, \ldots, K \times L\}$ to its two dimensional (t_n, r_n) form, and another that moves in the opposite direction. The program accomplishes the dynamic programming one the 1-d representation and computes transition scores using the 2-d representation. In its current form, the program recognizes data that is *synthesized* from the model, however it could easily be tested on real timing data.

Training HMMs with the Forward-Backward Algorithm

Imagine we have a simple HMM such as:



and we wish to learn the transition probabilities and the output probabilities for the model. In the usual HMM scenario we only observe the "output" of the model, y. However, for now assume that we observe the hidden process, x, as well. One particular instance might look like:

x	a	a	a	a	b	b	b	a	a	a	a	b	b	b
y	1	1	0	1	0	0	0	0	1	1	1	0	0	1

In this case, it is natural to estimate the probabilities we want by simply *counting*. For instance,

$$\begin{aligned} \hat{p}(a|b) &= \frac{\#\{X_n = b, X_{n+1} = a\}}{\#\{X_n = b\}} \\ \hat{p}(0|a) &= \frac{\#\{X_n = a, Y_n = 1\}}{\#\{X_n = a\}} \end{aligned}$$

or more generally,

$$\begin{aligned} \hat{Q}_{ij} &= \frac{\#\{X_n = i, X_{n+1} = j\}}{\#\{X_n = i\}} \\ \hat{R}_{ik} &= \frac{\#\{X_n = i, Y_n = k\}}{\#\{X_n = i\}} \end{aligned}$$

Of course, we often don't observe the X's in practice, and, in fact, the hidden process is often fictional, thus impossible to observe under any circumstances. A very simple way to deal with this situation is the following simple algorithm:

1. Assign initial parameters to the model (transition and output probabilities)
2. Find the most likely state sequence, $\hat{x}$, using dynamic programming
3. Use $\hat{x}$ to *re*estimate the parameters by counting.
4. Go to 2.

Such an approach may work well if our initial guess is close to the "truth," however, the algorithm usually does not perform so well in practice. A better way is the *Forward-Backward* or *Baum-Welch* algorithm. In the Forward-Backward algorithm, we estimate a *soft* labeling of the data, which gives the

probability of a particular hidden variable being in a certain state. The algorithm then iterates back and forth between the estimation of model parameters and the soft labeling of the sequence.

More specifically, the algorithm replaces the actual counts in the above equations for $\hat{Q}$ and $\hat{R}$ with the *expected* counts. That is

$$\begin{aligned}\hat{Q}_{ij} &= \hat{p}(X_{n+1} = j | X_n = i) \\ &= \frac{E\#\{X_n = i, X_{n+1} = j\}}{E\#\{X_n = i\}} \\ &= \frac{\sum_n P(X_{n+1} = j, X_n = i | Y = y)}{\sum_n P(X_n = i | Y = y)}\end{aligned}$$

and

$$\begin{aligned}\hat{R}_{ik} &= \hat{p}(Y_n = k | X_n = i) \\ &= \frac{E\#\{X_n = i, Y_n = k\}}{E\#\{X_n = i\}} \\ &= \frac{\sum_{n:y_n=k} P(X_n = i | Y = y)}{\sum_n P(X_n = i | Y = y)}\end{aligned}$$

Thus, if we can compute the probabilities state occupancy probabilities, $P(X_n = i | Y = y)$ and the as well as the probabilities $P(X_n = i, X_{n+1} = j | Y = y)$ we can implement the Forward-Backward algorithm as

1. Initialize the transition probabilities and observations probabilities
2. Using the current model, compute $P(X_n = i | y_1, \ldots y_n)$ and $P(X_n = i, X_{n+1} = j | y_1, \ldots y_n)$ with the forward-backward probabilities. This will use our current estimates of Q and R.
3. Substitute these probabilities into the above formulas to reestimate Q and R.
4. Go to 2 and repeat until the nothing much changes.

All that is missing from the algorithm is the way to compute these probabilities. These come from the following recursions. We define $\alpha_n(i) = P(X_n = i | y_1, \ldots, y_n)$, which can be computed by

$$\alpha_1(i) = \frac{P(X_1 = i) R(i, y_1)}{\sum_j P(X_1 = j) R(j, y_1)}$$

and

$$\alpha_n(i) = \frac{\sum_j \alpha_{n-1}(j) Q(j, i) R(i, y_n)}{\sum_{i'j} \alpha_{n-1}(j) Q(j, i') R(i' y_n)}$$

Similarly, $\beta_n(i) = P(y_{n+1}, \ldots, y_N | X_n = i)$ can be computed by

$$\beta_N(i) = 1$$

and

$$\beta_n(i) = \frac{\sum_j \beta_{n+1}(j) Q(i, j) R(j, y_{n+1})}{\sum_{i'j} \beta_{n+1}(j) Q(i', j) R(j, y_{n+1})}$$

Using this quantities we can get

$$P(X_n = i | y_1, \ldots, y_N) = \frac{\alpha_n(i) \beta_n(i)}{\sum_j \alpha_n(j) \beta_n(j)}$$

and

$$P(X_n = i, X_{n+1} = j | y_1, \ldots, y_N) = \frac{\alpha_n(i) Q(i, j) \beta_{n+1}(j)}{\sum_{i'j'} \alpha_n(i') Q(i', j') \beta_{n+1}(j')}$$

3.2.2 Expressive Melody Synthesis

Musical expression is a deep subject, and one risks sounding rather foolish trying to address it in generalities. However, if one wants to attack the expressive synthesis problem, this is exactly what must be done. My personal belief is that, while it may not be obvious how to codify the patterns of expression into a mathematical framework, there is quite a bit of logic in expression that remains to be discovered. The approach presented here is an attempt to do just that.

There is no official taxonomy of expression, though one can identify at least three different aspects,

Conveying musical structure concerns showing the higher level structure of the piece, including phrase boundaries. Often tempo changes are used to mark structural divisions.

Musical prosody is about showing note-level grouping and direction, analogous to prosody in speech.

Musical affect is the possibly changing mood of a piece: happy, sad, calm, agitated, etc.

Expressive synthesis is a subject that has a *comparatively* long history in music science, including several decades of research. Most of this work is focused on piano music, since it is comparatively easy to represent a piano performance. A common view of a piano performance corresponds the MIDI representation — the onset time, offset time, and initial "velocity" of each note. Sometimes pedaling is also added to the representation. Doubtless this representation misses some things — for instance, the resonating strings of the piano interact with one another. Perhaps, more importantly, much of what the player does is in response to that actual sound that is produced. Thus a MIDI performance on one piano may sound quite different from that "same" performance on another piano. Thus one wonders how much is actually captured by a MIDI representation. Still, the view is attractively simple, and allows one to get started with the problem.

In contrast, I'll talk here about expressive synthesis of *melody*. The most interesting context for melody synthesis is with a "continuously controlled" instrument such as the voice. A continuously controlled instrument allows one to vary many different "parameters" of the sound over time, such as loudness, tone color, pitch, and noise content. Other aspects of expression can be derived from these more basic parameters to produce timing, vibrato, articulation, glissando, and many other aspects.

A realistic look at a real instrument allows for so many parameters, the problem becomes needlessly complex. Likely one doesn't need all of these to create convincing musical expression. We will look at the simplest possible view of a continuously controlled instrument — one that varies only pitch and intensity over time. This is analogous to the *theremin* instrument discussed eariler in class, which modulates the amplitude and frequency of a sine wave. This can be described mathematically as

$$s(t) = a(t)\sin(\int_0^t 2\pi f(t)\tau d\tau)$$

The important thing to understand here is that the sound is specified entirely by the $f(t)$ and $a(t)$ functions. We have seen that this instrument already is capable of considerable musical expression in the hands of a theremin master. To make this model a little more interesting we will change the tone color of the sound, making it brighter as the sound becomes louder. The physical way in which this is done is beyond the scope of this class, though this is a natural thing to do since most acoustic instruments share this same coupling of loudness and tone color.

What can our instrument do? Simply by using the changing frequency function, $f(t)$, we are able to get notes and rhythm. By varying the pitch over the duration of a single note we can get vibrato. Changing the frequency quickly between pitches produces glissando. The intensity function can be used to create the usual kinds of dynamics, but also articulations and other effects. See Figure 3.12 for examples.

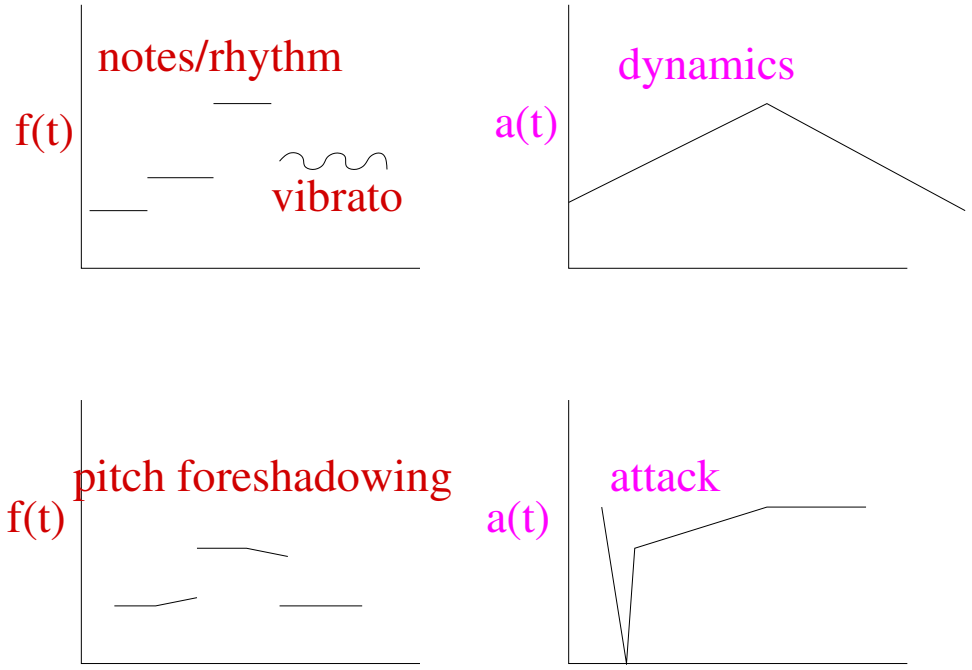


Figure 3.12: Examples of what can be accomplished with the theremin representation.

We will demonstrate the range of music inflection in class that can be acheived with our theremin-like instrument. One thing that becomes obvious when one tries to create musical sounding performances by maniplulating $f(t)$ and $a(t)$ is that not all notes can be treated the same. Any technique that handles the notes in a uniform manner is bound to sound monotonous. To address this issue we introduce here af *representation* of prosody that makes clear interpretive choices by labeling each melody note with a symbol from a small alphabet,

$$A = \{l^-, l^\times, l^+, l^\rightarrow, l^\leftarrow, l^*\}$$

describing the role the note plays in the larger context. These labels, to some extent, borrow from the familiar vocabulary of symbols musicians use to notate phrasing in printed music. The symbols $\{l^-, l^\times, l^+\}$ all denote stresses or points of "arrival." The variety of stress symbols allows for some distinction among the kinds of arrivals we can represent: l^- is the most direct and assertive stress; $l^\times$ is the "soft landing" stress in which we relax into repose; l^+ denotes a stress that continues *forward* in anticipation of future unfolding, as with some phrases that end in the dominant chord. Examples of the use of these stresses, as well as the other symbols are given in Figure 3.13. The symbols $\{l^\rightarrow, l^*\}$ are used to represent notes that move *forward* towards a future goal (stress). Thus these are usually shorter notes we "pass through" without significant event, perhaps focussing the listener's attention on what is coming. Of these, $l^\rightarrow$ denotes the "garden-variety" passing note, while l^* is reserved for the passing stress, as in a brief dissonance, or to highlight a recurring beat-level emphasis, still within the context of forward motion. Finally, the $l^\leftarrow$ symbol denotes receding movement, as when a note is connected to the stress that precedes it. This commonly occurs when relaxing out of a strong-beat dissonance *en route* to harmonic stability.

Examples of these labels for actual musical examples are given in *Amazing Grace* and *Danny Boy* in Figure 3.13. We will show how, when these symbols are expressed using the $f(t)$ and $a(t)$ functions, we can capture some of the desired prosody of the music.

Having found a way to represent one aspect of musical expression, we are on much more solid footing

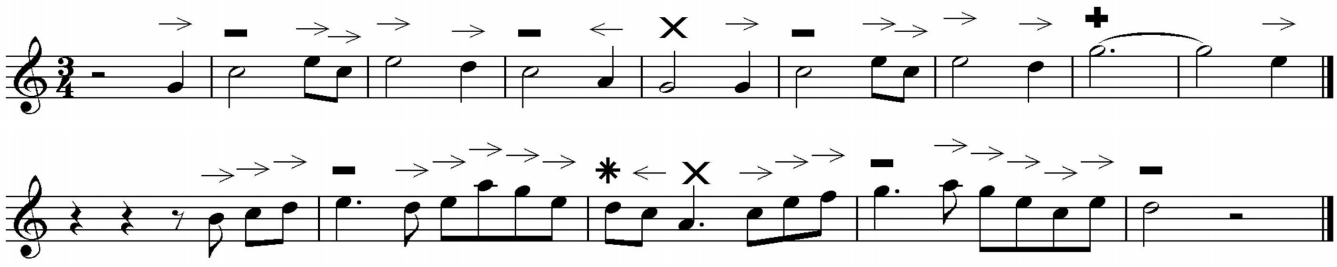


Figure 3.13: *Amazing Grace* (**top**) and *Danny Boy* (**bot**) showing the note-level labeling of the music using symbols from our alphabet.

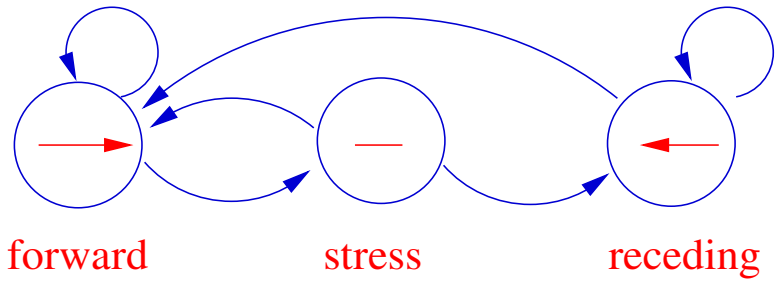


Figure 3.14: coming

for trying to synthesize the expression for a new piece of music. We cast the problem as one of *estimating* the prosodic labeling — one symbol for each note. We will write this sequence of labels as $x_1, x_2, \ldots$ where $x_n \in A$. The important thing to notice here is that there is a pattern that governs the evolution of the symbols. We tend to see sequences of forward moving notes, followed by a single stressed note, perhaps followed by several receding notes, with this pattern repeating throughout the melody. This behavior is illustrated in Figure 3.14. The state-like description of this patterns suggests modeling the sequence x as a Markov chain, and this is exactly what we have done. But somehow the movement between the states must *depend* on the musical score. We have modeled this by letting the probabilistic state transitions of x depend on local measurements about the musical score, y_n. These include aspects such as length of note, strength of metric position, contour of pitchs, etc. We then model the conditional dependence of x on y as

$$p(x|y) = p(x_1|y_1) \prod_{n=2}^{N} p(x_n|x_{n-1}, y_n, y_{n-1})$$

Without going into details, these transition probabilities are learned from a corpus of hand-labeled examples. We then compute the most likely state sequence, given the score features, using the usual ideas of dynamic programming.

We will present a number of examples in class that show the results of this process.