Notes for Biological Sequence Analysis

Anders Krogh
The Bioinformatics Centre, University of Copenhagen, Denmark

September 10, 2019

Contents

Chapter 1

Some background on math notation for BSA

From previous experience, the math notation is a problem for many students in the course Biological Sequence Analysis. These notes is an attempt to help you better understand the course and the book used (Durbin et al).

Please suggest additions and changes, so this perhaps can develop into a more complete chapter.

1.1 Variables, indices, sums, etc

A variable is a mathematical abstraction that you remember from equations in school, like $3x+1=10$, where the aim is to find the correct value for variable x ($x=3$ in this case). A variable can also mean a letter, such as a nucleotide or an amino acid, which may for instance be denoted by a. This variable can take on values `A`, `C`, `G`, or `T`, if we are working with DNA.

We also talk about a sequence x (or y), which is a shorthand, which can be written in more detail as $x_1, x_2, \ldots, x_l$, where the length of the sequence is called l. This is a general representation of a sequence, and each element of x can take on values from the alphabet we are working with. An instance of such a sequence may be `CCAGGATTCAGAGAT` (again assuming DNA). For this particular sequence $l = 15$ and x_1 =`C`, x_2 =`C`, x_3 =`A`, x_4 =`G`, ..., x_l =`T`.

An index is a type of variable that can take on positive integer values (sometimes including 0). Let us assume that we have the weight (in kilograms) of N different subjects. The subjects are numbered from 1 to N. When I say that "the weight of subject i is denoted x_i", I use i as an index, which in this case is a number between 1 and N. It is a short way of saying that x_1 is the weight of subject 1, x_2 is the weight of subject 2, etc. To calculate the mean weight of the subjects, we have to calculate

$$m = \frac{1}{N}(x_1 + x_2 + x_3 + \ldots + x_N).$$

Using indices, we can instead write this with a sum symbol:

$$m = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

Note that the index could also have been called j or t or whatever, at that we could just as well have written $m = \frac{1}{N}\sum_{t=1}^{N} x_t$. There is a similar product symbol, so multiplication of all the xs can be written as

$$\prod_{i=1}^{N} x_i = x_1 x_2 x_3 \cdots x_N.$$

Sometimes we are lazy and write just $\sum_i x_i$, which means that the limits for i are implicit – we know that i can take values from 1 to N.

Sums and products can also be over sets other than subsets of integer numbers. We may write for instance

$$\sum_a q_a = 1, \tag{1.1}$$

for the probabilities of the letters we are considering. Here a denotes a letter (e.g. one of the four nucleotides) and q_a the probability of the letter. The sum is (implicitly) over all possible letters, so it means $q_{\mathtt{A}} + q_{\mathtt{C}} + q_{\mathtt{G}} + q_{\mathtt{T}}$. A more precise way of writing this would be using set notation

$$\sum_{a \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}} q_a = 1.$$

Instead of q_a, one can also call it $q(a)$, which is just another way of writing the same.

1.2 The logarithm

Logarithms are used a lot with probabilities. To compare two probabilities, we often look at the log of the ratio between them, which is more convenient that the ratio itself. In some models many probabilities are multiplied making the result very small, but when taking the logarithm, the number becomes managable. If a probability is e.g. 10^{-400}, the negative logarithm is 400, if log base 10 is used.

A logarithm has a *base* number. The "normal" logatithmic functions have base 10 or base e, and we will often encounter logarithm base 2. The logarithm base A, $\log_A$, is the inverse of exponentiation

$$\log_A(A^x) = x \quad \text{and} \quad A^{\log_A(x)} = x.$$

The *natural logarithm* (often denoted 'ln') is the inverse of the exponential function with base $e \simeq 2.718$.

The logarithm (to any base) of a product is the sum of the logarithms:

$$\log(abcd) = \log(a) + \log(b) + \log(c) + \log(d)$$

$$\log(a^b) = b\log(a).$$

It is useful to remember that

$$\log_A(1) = 0 \quad \text{and} \quad \log_A(A) = 1$$

Logarithms are proportional

$$\log_A(x) = \log_B(x)/\log_B(A)$$

To calculate the logarithm base 2 from the natural logarithm, just do

$$\log_2(x) = \frac{1}{\ln(2)}\ln(x) \simeq 1.4427\ln(x).$$

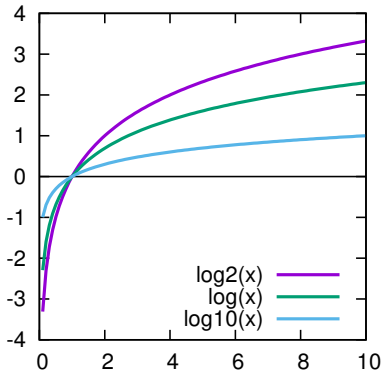


Figure 1.1: Plot of the three most commonly used logarithms with base 2, e, and 10.

The result of taking $\log_2$ is sometimes called bits. It has to do with the fact that it takes n bits to specify 2^n numbers in a computer. For instance, the 256 numbers from 0 to 255 may be represented in an 8 bit binary format (00000000, 00000001,00000010,00000011,...,11111111), and since $2^8 = 256$, $\log_2(256) = 8$.

The most common logarithms are the natural logarithm, the logarithm base 10, and the logarithm base 2. Since they are proportional, theoretical results are usually independent of which logarithm is used, so we may be a bit sloppy specifying which one is used. In Figure 1.1 you see graphs of these functions.

1.3 Probabilities

Probabilities are difficult to undestand for many people. Just keep in mind that a lot of the rules are really very simple and intuitive even if they sound very complex.

Here are some important properties of probabilities:

Probabilities are non-negative, less than or equal to 1, and sum to one. A probability is always between 0 and 1 (both included). The probability of mutually exclusive events sum to one. Letter probabilities for instance always sum to one when you sum over all letters in the alphabet as in Equation 1.1 above.

The probability of two independent events is the product of individual probabilities. If you play dice, the probability of rolling a 6 is $1/6$. The probability of rolling TWO sixes in two rools is $\frac{1}{6}\frac{1}{6} = 1/36$. The two rolls are independent. Similarly, if DNA sequences are constructed by picking each nucleotide randomly with probability q_a for letter a, the probability of obtaining a sequence x, would be

$$\prod_i q_{x_i}.$$

Marginal probabilities are obtained from joint probabilities by summing. If one random variable B has mutually exclusive outcomes, we can obtain $P(A)$ by summing $P(A,B)$ over all possible values of B.

Let us say that A is hair color (dark, blond, red) and B is the sex of a person (male or female). If there is 1.1% chance that a person is male and has red hair and 1.2% chance that a person is female and has red hair, then it really is quite obvious that the chance that a person has red hair is 1.1+1.2=2.3% (P(red,male)+P(red,female)). That is all there is to it.

We often would write this with sums. Assume that B can take values $b_1, b_2, \ldots, b_n$, then

$$P(A) = P(A,b_1) + P(A,b_2) + \ldots + P(A,b_n) = \sum_i P(A,b_i)$$

I believe that P(A) is called the marginal probability, because in a table like the one below it appears in one of the margins. P(B) is the other Marginal probability in this example (in which I made up the numbers).

	A (Hair color)			
B (Sex)	dark	blond	red	P(B)
Male	0.271	0.215	0.011	0.497
Female	0.266	0.225	0.012	0.503
P(A)	0.537	0.440	0.023	

The joint and conditional probability of two events A and B are related as

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

This is an extremely important property that will be used over and over again.

Sometimes the summing rule for joint probabilities above is rewritten using this. Since $P(A,b_i) = P(A|b_i)P(b_i)$, it becomes

$$P(A) = \sum_i P(A|b_i)P(b_i)$$

Example

Assume that a disease is correlated with a mutation. If you have this mutation, there is a risk of 60% that you will develop the disease. Only one in 1000 people carry this mutation. What is the probability that a random individual carry the mutation **and** develops the disease?

Answer: The probability og the mutation is $P(M) = 0.001$ and the probability of developing the disease given that you have the mutation is $P(D|M) = 0.6$. Therefore the joint probability of having the mutation **and** developing the disease is

$$P(M,D) = P(D|M)P(M) = 0.6 * 0.001 = 0.0006.$$

If we assume that one in 100 develops the disease when they do not have the mutation ($M-$), we can also calculate the overall probability of developing the disease, because

$$\begin{aligned} P(D) &= P(D,M) + P(D,M-), \text{ and} \\ P(D,M-) &= P(D|M-)P(M-) = 0.01 * (1 - 0.001) = 0.00999, \text{ and thus} \\ P(D) &= 0.00999 + 0.0006 = 0.01059. \end{aligned}$$

Bayes theorem

The above identity between joint and conditional probability can be used both ways ($P(A,B) = P(A|B)P(B)$ and $P(A,B) = P(B|A)P(A)$). From this we can derive the famous Bayes theorem:

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

This is used a lot in many different settings and is the basis for "Bayesian statistics".

Example

This can be used in the above example to calculate the probability of having the mutation, given that you develop the disease:

$$P(M|D) = P(D|M)P(M)/P(D).$$

In order to calculate $P(D)$, we can use $P(D) = P(D,M) + P(D,M-)$, where $M-$ means that you do not have the mutation. We already know the first of these terms $P(D,M) = 0.0006$. The second term is $P(D,M-) = P(D|M-)P(M-) = 0.01(1-0.001) = 0,00999$, because $P(M-) = 1-P(M)$. So,

$$P(M|D) = 0.6 * 0.001/(0.0006 + 0.00999) = 0,057.$$

So less than 6% of those having the disease actually carry the mutation.

1.4 Algorithmic complexity and computer jargon

In a computer's memory (or hard drive) all information is converted to sequences of 0s and 1s called bits. The size of a storage device is measured in bytes. A byte is 8 bits. A kilobyte, denoted kB is 1000 bytes, and similarly MB, GB and TB are 10^6, 10^9, and 10^{12} bytes, repectively (although sometimes it means $2^{10} = 1024$, $2^{20} = 1024^2$ and $2^{30} = 1024^3$ bytes).

When we talk about computer *memory*, we mean the RAM. This is the storage that can be accessed very rapidly when we run a program. The memory is normally smaller than the hard-disk on a computer. Typical size of memory in personal computers, phones, etc is between 1 and 16 GB, whereas computer hard-disks are typically 1-8 TB.

In bioinformatics we are often concerned with how much computer memory and run-time a program or algorithm is using. With large quantities of data, this may well determine whether a bioinformatic analysis is feasible or not. The actual run-time of an algorithm obviously depends heavily on the data, how the algorithm is implemented, and what hardware is running it. However, how it scales with the input size, can often be deduced.

As you will see in the course, the number of calculations it takes to align two proteins of length l_1 and l_2 is *proportional* to $l_1 * l_2$. Therefore we expect that the number of seconds it takes on any computer with any sensible implementation is a constant times this. This is essentially what we mean by the time complexity of an algorithm and it is sometimes written as $O(l_1 l_2)$, which means that the runtime is "order l_1 times l_2". In a strict mathematical sense it should be the limiting behaviour of the time as a function of l_1 and l_2.

The same goes for memory usage, although the constant of proportionality is often easier to work out. The standard pairwise alignment by dynamic programming has memory complexity equal to the time complexity, $O(l_1 l_2)$, because with each operation, you have to store a number in a $l_1 \times l_2$ table. The number you have to store is a small integer and can easily fit in one byte, so it means that in terms of memory usage, you can align two sequence if $l_1 l_2$ is less than the memory of your computer (e.g. $l_1 = l_2 \simeq 30000$ letters if you have one GB memory).

Chapter 2

A short introduction to dotplots

2.1 Sequence comparison

Protein, DNA and RNA sequences are conserved in evolution. Not such that homologous proteins from human and round worm are identical, but they may have almost the same structure and some similarities in the amino acid sequences. Proteins often have strongly conserved functional parts (like active sites) mixed with much less conserved regions where the exact sequence of amino acids is not so important for the structure and function of the proteins. These regions often vary in size between species. The comparison of sequences is difficult because we usually see insertions and deletions (indels). These indels mean that we cannot trivially compare two sequences side by side. This sequence comparison problem is one of the most fundamental problems in bioinformatics.

Perhaps the simplest way to compare two sequences with indels is to make a *dotplot*, which we will discuss in this chapter. To be honest, dotplottting is probably not going to be the most important tool for you, but it is a good way to start thinking about sequence comparisons. Having said this, a dotplot is also a very good tool in some circumstances. In situations where the sequences are not co-linear, e.g. when comparing chromosomes with re-arrangements, a dotplot is very useful. It is also worth noting that dotplots can be used for identifying repeated and palindromic structures, which may not be seen when using other pairwise comparison methods (pairwise alignment).

2.2 What is a dotplot?

A dotplot is a sort of scatter plot of one sequence against another, where a dot is put at places where the two sequences are identical or similar. In the simplest case, a dot is put in positions where the letter is the same in the two sequences. This is illustrated with a protein against itself in Figure 2.1A. We clearly see the diagonal line, which of course must be there. We also see a lot of dots scattered in the remaining plot, which perhaps shows some patterns, but are very hard to interpret.

In a dotplot of two random protein sequences approximately every 20th point would contain a dot, and this is part of the reason for the off-diagonal dots in Figure 2.1A. This of course is one dot for every four in DNA or RNA sequences, which would make them even harder to interpret. Therefore it is customary to average over a small window. For instance, one can show a dot if

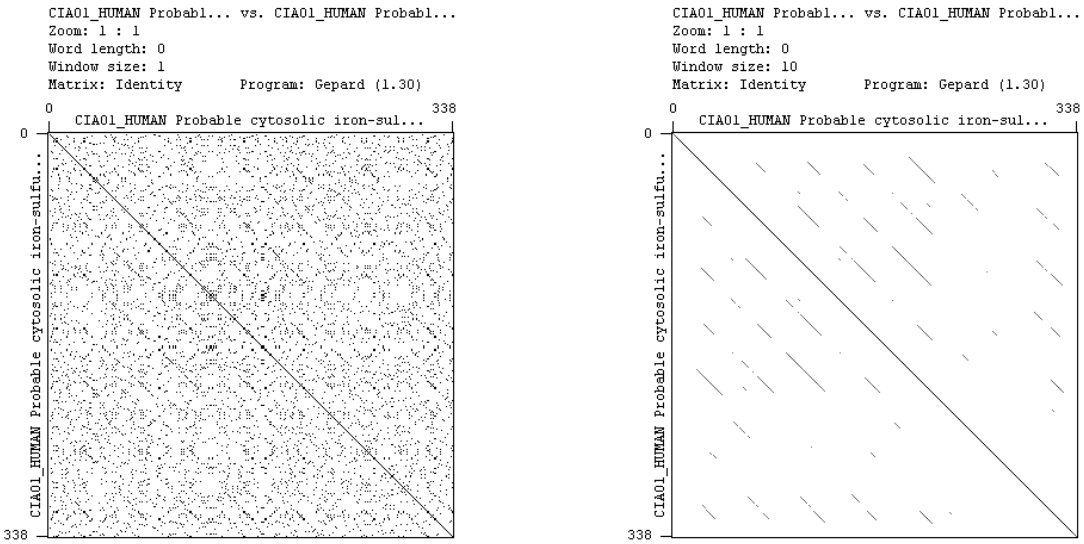


Figure 2.1: Dotplots made with the Gepard program described later. The protein has the Swiss-Prot identifier CIAO1_HUMAN, "Probable cytosolic iron-sulfur protein assembly protein CIAO1". A. A window size of one gives a result, which is difficult to interpret. B. With a window size of 10, one can see that the there are similarities within the protein. This protein has 7 copies of a domain called the WD domain.

```
....WSPCGNYLASASFDATTCI.....
...WSPCGNYDLGSADFDDTTIWK....
```

Figure 2.2: Two windows of size 5 shown. The first window have 0, and the second has 3 of 5 identical letters.

3 out of five neighboring letters are identical or more generally if t out of w letters are identical, where w is the window size and t is the threshold, see Figure 2.2. This is illustrated in Figure 2.1B for the same protein as in Figure 2.1A and using a window size of 10. It is now much more clear that there are regions in the protein that are very similar (see caption for more details).

In Figure 2.3A a dot plot of the protein used in Figure 2.1 against another protein is shown. Again it is clearly seen that these proteins have regions of similar sequences.

An even more sophisticated approach is to use gray scales showing a light gray dot if few letters in the window are the same, and a dark gray one if many are identical. Furthermore, instead of using a count of identical letters, one may use a scoring matrix (such as Blosum or PAM matrices) to calculate a score of the two sequences against each other. In several available dotplot programs, one can set the window size, choose how to score, and vary the display by setting the score limits for the gray scale. This is illustrated in Figure 2.3B using a Blosum62 score matrix and gray scale.

2.3 Hairpins and inverted repeats

In DNA sequences, repeated sequences are very common and can occur on both strands. Repeats on opposite strands means that a window of DNA is identical (or very similar) to another window if it is reverse complemented. A similar phenomena can be observed in RNA, which often form

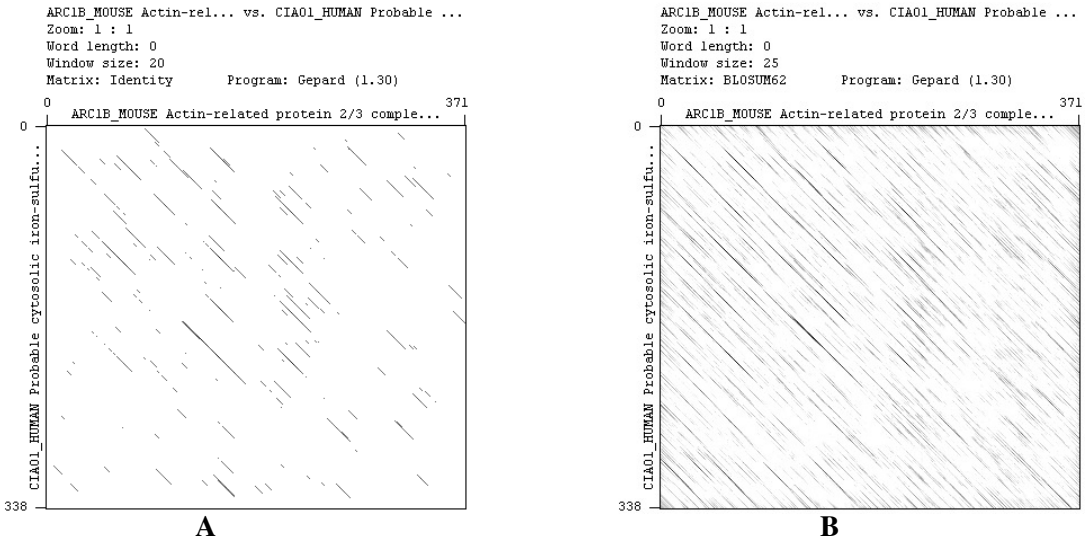


Figure 2.3: Dotplot of actin-related protein from mouse (Swiss-Prot identifier ARC1B_MOUSE) against the protein used in Figure 2.1. The mouse protein has 4 wd domains, which can be seen in the dot-plots. A. Using 0/1 score for matching letters and a window size of 20. B. Using BLOSUM62 scoring matrix, a window size of 25, and a more extended gray-scale.

structures by base-paring to itself. Because base-pairing occurs between (almost) complementary strands, the two sides of the strand appears as repeats on opposite strands.

Many programs for dotplots can also show this type of reverse complement matches. The similarity between two windows is calculated on both strands, and the maximum score is displayed. This is illustrated in Figure 2.4 for an RNA molecule, where the base-paring regions can be seen in the dotplot.

As another example, the dot-plot was made of an E. coli (K12) bacterial genome against the Salmonella Typhi genome Figure 2.5. These two genomes are fairly closely related, which is quite clear from the plot, because of the strong diagonal parts. However, one can also see that some regions are reversed in the two genomes. This phenomena is often observed when comparing genomes from different species.

2.4 Calculating dotplots

It is very simple to make a dotplot as a scatter plot in a graphics program. You just need to calculate the matrix of scores to use in each point. If the sequences are long, however, it is quite a lot of calculations that need to be done. For sequences of length N and M there are $N \times M$ points to calculate. In the plot shown in Figure 2.5 we have $N = 4.6$ mill. and $M = 4.8$ mill., which means that there are about 2×10^{13} points. That is a large number even for modern computers. To be able to do that, a clever method has to be used.

One of the most widely used programs for making dotplots is called *dotter* (http://sonnhammer.sbc.su.se/Dotter.html). It uses several smart tricks for speeding up calculations and saving memory and can be used for sequences of several thousand letters. It also have nice interactive tools for adjusting parameters, zooming, etc.

In this chapter, I have used another program called *gepard* (http://cube.univie.ac.at/gepard), which

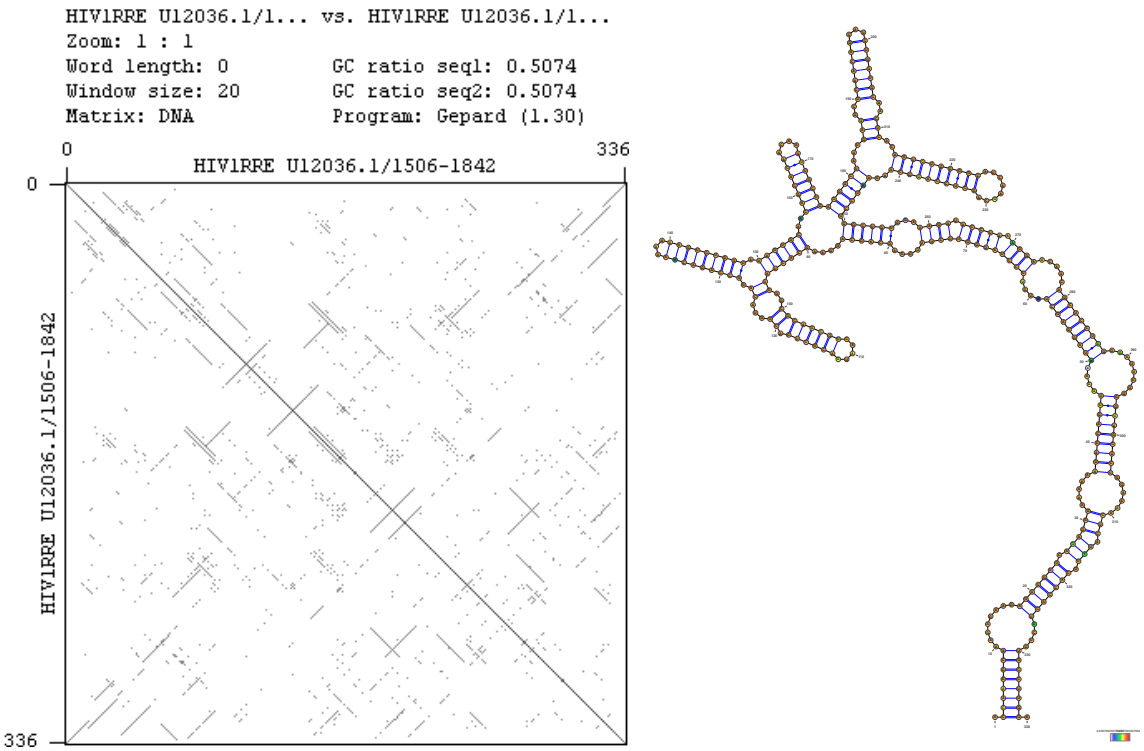


Figure 2.4: The Rev response element (RRE), which is a region in the RNA molecule of the HIV env gene (see Rfam RF00036). To the left a dot plot is shown against the sequence itself using a window size of 20. To the right a predicted secondary structure of the RNA molecule is shown. In the dotplot you can see anti-diagonal lines corresponding to RNA strands, where the molecule is complementary to itself because of base-pairing.

is written in Java and therefore runs on almost any box. This can be used for standard dotplot of small sequences. It can also be used for chromosome-scale dotplots, because it can "cheat". Instead of calculating a standard dotplot as described above, it looks for identical sequences of length n (these are often called words of size n). For the plot in Figure 2.5 a word size of 10 was used. So it displays a dot if two words are identical. The identification of identical words can be done by a clever indexing of the sequences called a suffix array. We will later discuss such sequence indexing techniques.

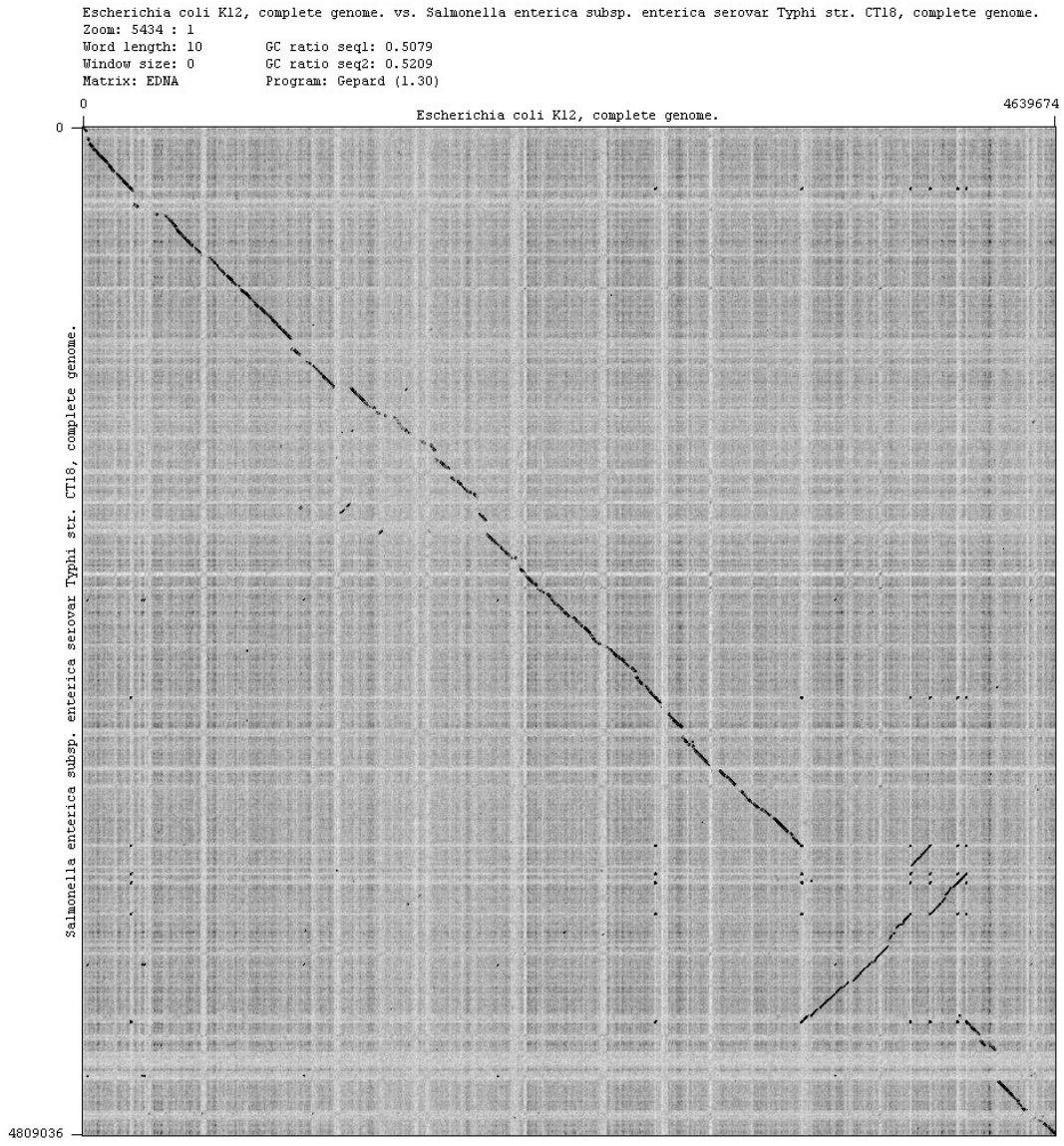


Figure 2.5: Dotplot of E. coli versus Salmonella Typhi. The anti-diagonal lines corresponds to regions that are inverted in one genome relative to the other.

Chapter 3

Statistics of Alignment Scores: an Experimental Investigation

3.1 Introduction

When searching a database with pairwise alignment, using e.g. the Smith-Waterman algorithm, you will get the best match between the query sequence and all the sequences in the database. Even if two sequences are completely unrelated, you will normally be able to find a short local match with positive score. Therefore, we need to try to figure out how large a score should be, for a match to be relevant for further analysis.

As we cannot relate the match score to biological relevance, we ask instead if a given score is statistically significant. To do this, we need to compare to a null hypothesis, which in this case is that the sequences are unrelated. To model unrelated sequences, we use random sequences. Then we can calculate a p-value [1]:

> Given a score S, the p-value is the *probability* to obtain the same score or higher if we searched a database of random sequences with the same size as the actual database with a random query sequence of the same length as the actual query.[1]

The more used E-value is closely related to this, as explained in [1]:

> Given a score S, the E-value is the *expected number* of matches with the same score or higher if we searched a database of random sequences with the same size as the actual database with a random query sequence of the same length as the actual query.

One of the most common errors you hear is that people call the E-value a probability. It is *not*. It can take values bigger than one. It can also take non-integer values, because it is an *expectation*, meaning that it is an average.

The idea of this note is to run some simple experiments to see how the theoretical result relates to reality. Hopefully these experiments will also help you to understand what the the term E-value actually means.

[1] It is assumed that the composition of the random sequences match the scoring model as discussed later.

Tools used

You can redo these experiments on your own on a linux or Mac computer – and with a bit more effort under Windows[2]. We will use some simple tools.

water is part of the Emboss package (http://emboss.sourceforge.net). It implements the Smith-Watermann algorithm (there is a similar one called needle for the Needleman-Wunch algorithm).

shuffleseq is also from the Emboss package. It is used to shuffle the letters in a sequence in random order.

R is an open source statistics program, which can be installed on any type of computer. Here we will use it for plotting. It is very useful for many other (statistics) tasks in bioinformatics and if you do not know it already, it is worth learning.

Awk is an old scripting language that uses regular expressions and preceded (and inspired) more advanced languages like perl and python. It is available in all unix versions and today it is typically used in little one-line programs on the command line. You need not learn awk.

Unix commands We will be using a few standard programs in unix. To scroll through a text file, you can use `more` or `less`. To see the end of a text file (last 10 lines by default), you can use `tail`. To sort lines in a file, use `sort` and to do numerical sort, use `sort -n`. To printthe name of your current diretory, use `pwd`.

3.2 Local alignment scores

In this section we will perform a database search with the Smith-Waterman local alignment method and plot the score distribution.

Here is an arbitrary protein from yeast in fasta format that we will use in the experiment.

```
>GRPE_YEAST
MRAFSAATVRATTRKSFIPMAPRTPFVTPSFTKNVGSMRRMRFYSDEAKSEESKENNEDL
TEEQSEIKKLESQLSAKTKEASELKDRLLRSVADFRNLQQVTKKDIQKAKDFALQKFAKD
LLESVDNFGHALNAFKEEDLQKSKEISDLYTGVRMTRDVFENTLRKHGIEKLDPLGEPFD
PNKHEATFELPQPDKEPGTVFHVQQLGFTLNDRVIRPAKVGIVKGEEN
```

Try to run water against the database of about 4200 E. coli proteins that we will use for our experiment:

```
water GRPE_YEAST.fa Ecoli.prot.fasta
```

You can press return when it prompts for gap penalties etc. to use default values. The program uses the Blosum62 substitution matrix as default. The output by default ends up in a file with the extension "water" – like `GRPE_YEAST.water`. Take a look at the output using `more` or `less`.

Now, to study the score distribution, you need to get the scores. You see there are lines in the output like `# Score: 80.5` for each of the proteins in the database. We just need to extract those numbers. Here we use `awk` to do this, but you can also use python or perl if you prefer. If the water output is in a file called `GRPE_YEAST.water` this command will print all the scores:

[2]In Windows version 10, you can install a 'bash shell', but I have no experience using it. You can also install a package called cygwin that emulates unix.

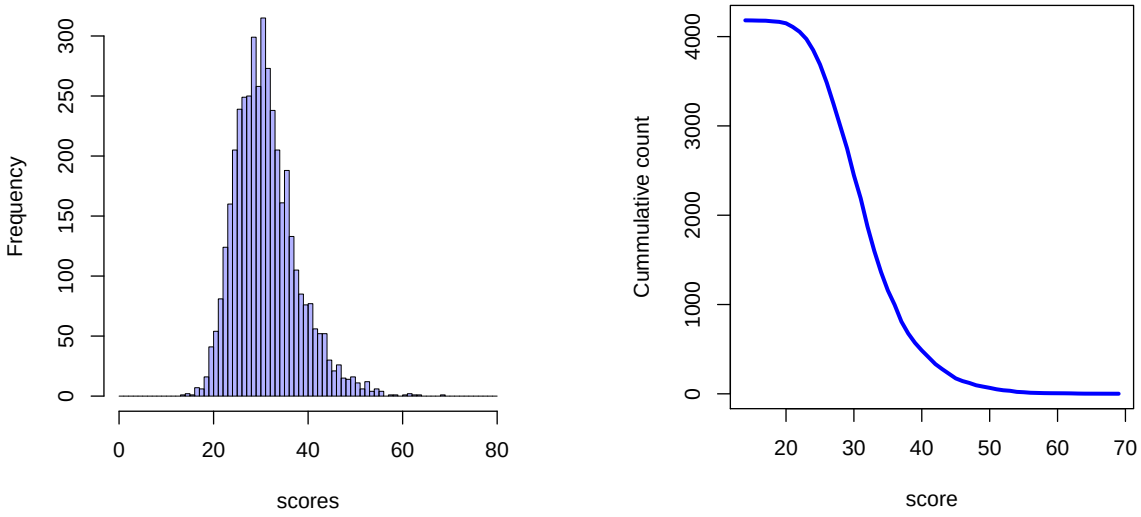


Figure 3.1: Left: Histogram of alignment scores when searching the database of E. coli proteins with the GRPE_YEAST protein. Right: Same, but plotted reverse cumulative

```
awk '/^# Score:/ {print $3;}' GRPE_YEAST.water
awk '/^# Score:/ {print $3;}' GRPE_YEAST.water | sort -n > GRPE_YEAST.score
```

The second line above shows how you can sort the scores numerically (`-n`) and direct the output to a file (>).

If you look at the end of the sorted file (use `tail GRPE_YEAST.score`) you see that there is one score of more than 250, but all other scores are much lower. The single large score is likely to be a match to a biologically related protein, because there is such a large gap in scores to all the other matches.

To visualize the scores, you can use R:

```
setwd("/Users/.../yourdirectory")
tmp <- unlist(read.table("GRPE_YEAST.score", head = F))
scores <- tmp[-length(tmp)]
hist(scores, breaks = c(0:80), col=rgb(0,0,1,0.3))
```

You have to insert the full directory name in the first command (in unix `pwd` will print your current directory). When you read the scores with read.table, you obtain a data frame. The unlist command turns it into a standard numerical vector. The third line of magic removes the last data point, which is the single last score – all other scores are below 80. The col argument to hist puts color on the bars and can be omitted. The resulting histogram is shown in Figure 3.1.

Since we are interested in the total number of matches at or above a certain score we might as well show that. So, for every score value in the file we plot the total number of occurrences of this score or higher, see Figure 3.1. Here are the R commands:

```
# table() counts number of occurences, rev() reverses
# cumsum() makes cumulative sum
tmp<-table(scores)
counts <- as.vector(tmp)
Cwt <- rev(cumsum(rev(counts)))
Swt <- as.numeric(names(tmp))
plot(Swt, Cwt, main="Cumulative Distribution", xlab="score", ylab="Cummulative
    ⟶ count", type="l", col="blue")
```

Note the line break shown with a blue arrow in the last line. If you copy paste, concatenate the parts without the blue arrow.

3.3 Comparing to the E-value

In this section, we will compare the score distribution obtained above to the theoretical model for E-values (while in the next section we'll get back to discuss what the E-value actually is).

As mentioned above there is a single large score, which is likely to be a real match between the yeast protein and a homologous protein in E. coli. The reason we are interested in score statistics is of course to find out how large a score should be before we will consider the match interesting.

From the book we know that the E value is given by $E = Kmne^{-\lambda S}$, where K is a constant close to 1, m and n are the length of the query and the length of the database, and λ is a constant, which scales the scores to the correct exponential function. (If the scores reported by the program are in e.g. bits, we need to scale them before applying the exponential function with base e.)

We need to calculate λ, so we need to find out the log-scale of the BLOSUM62 matrix. If EMBOSS is installed you can type `locate BLOSUM` to get a list of different Blosum matrices that comes with the package in a directory ending with `emboss/6.6.0/share/EMBOSS/data/` (locate may not be working on your computer). The beginning of the file called `EBLOSUM62` looks like this:

```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
 ... some left-out lines ...
X  0 -1 -1 -1 -2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2  0  0 -2 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

In the third line you see that it is in half bits, meaning that it is $2\log_2(p_{ab}/(q_a q_b))$, where p_{ab} is a probability according to the match model and $q_a q_b$ is the probability according to the random model (see Chapter 2 in [1]). To scale it to the natural logarithm, which is used in the theory, we use that $\log_2(x) = \log(x)/\log(2)$. Therefore $\log(p/q) = \log(2)\log_2(p/q) = (\log(2)/2)(2\log_2(p/q))$ and thus $\lambda = \log(2)/2$.

Having λ, we can now plot the theoretical $E = Kmne^{-\lambda S}$ in the cumulative plot we did above. The theory is assuming high scores, because we are interested in the high random scores – those are the ones that may be confused with real matches. In the cumulative plot we cannot see the details too well for large scores and therefore we plot the data logarithmically instead. The additional R commands are

```
lambda <- log(2)/2.
evalue = function(x){1.0*228*1335035*exp(-lambda*x)}
plot(Swt, Cwt, main="Cumulative Distribution", xlab="score", ylab="Cummulative
    ⟶ count", type="l", col="blue", log="y")
curve(evalue, from=30, to=80, type="l",col="red", log="y", add=T)
```

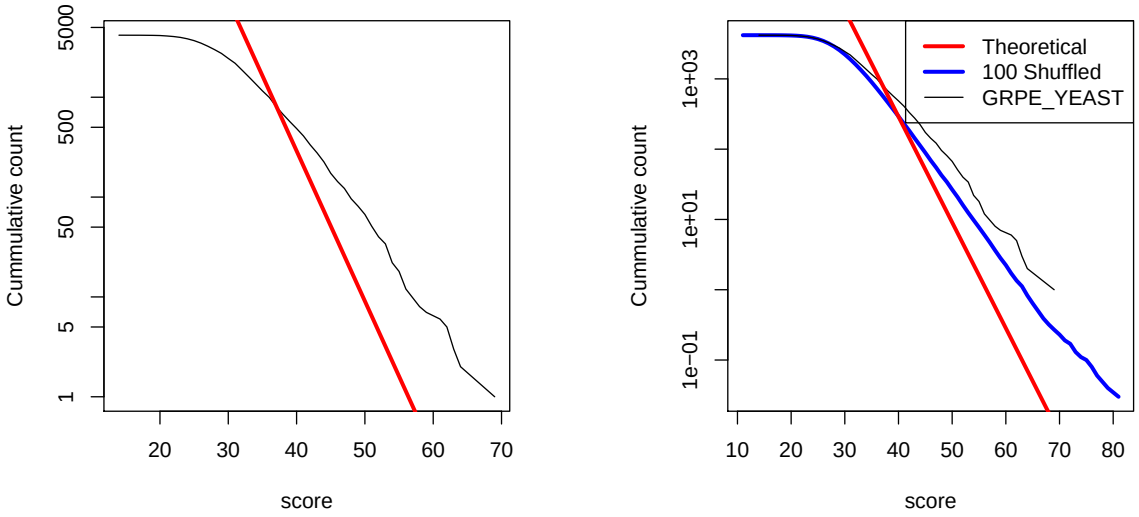


Figure 3.2: Left: Logarithmic plot of the number of matches above a certain score (same data as in Figure 3.1 left) with the red line showing the theoretical model. Right: The same when averaging over 100 shufflings of the query sequence shown in blue. The other lines are identical to the left plot. Notice the y-axis is changed, because the minimum number of matches is 1 when searching with one sequence and it is 1/100 when averaging over 100 runs.

Here we used the length of the query sequence $m = 228$, the total number of amino acids in the database is $n = 1335035$, and we assumed $K = 1$. The resulting plot is shown in Figure 3.2

Exercise 1 *Discuss which factors could influence the fairly poor correspondence between the theory and the actual observed scores. Here are some keywords that may guide you: Homology, composition, randomness, single query, violation of assumptions.*

3.4 The expected number of random matches

In this section, we try to find out what an E-value really is and use randomized query sequences instead of a real protein.

What does it really mean when we for instance say that there is an expected number of random matches with a score greater than or equal to some number S (the E value)? It means something like this:

If we run many (e.g. hundreds of) searches with random sequences of the same length and composition against a database, the E-value is the average number of matches with a score of S or greater in these searches.

Random sequences

The theory talks about random sequences, which means sequences obtained by picking random letters from the background distribution q_x both for query and target sequences. However, the important point is probably that the sequences are unrelated. Therefore, we will do an experiment where the database contains real biological sequences, and the query is a shuffled sequence.

Shuffling a sequence means that you reorder the letters in the sequence in random order just like mixing a deck of cards. Here we use a program called shuffleseq from the Emboss package. This setup is a bit closer to the real situation than if we used completely randomly generated sequences. You can try to run shuffleseq on the yeast protein we used above – just give the filename as argument to shuffleseq.

You can try to do a search as above using a single randomly shuffled sequence by redirecting the output to a file (`shuffleseq GRPE_YEAST.fa > shuffled.fa`) and then run water as above. It is also possible to do everything in one go: shuffle the sequence, send it to water and send it on to awk to extract the scores. This will be useful when we want to do this many times in order to collect statistics. Then it looks like this:

```
shuffleseq -sequence GRPE_YEAST.fa -outseq stdout | water -brief -asequence
    ⟶ stdin -bsequence Ecoli.prot.fasta -outfile stdout -gapopen 11. -
    ⟶ gapextend 1. | awk '/^# Score:/ {print $3;}' > shuffled.scores
```

All the options to water are there to bypass the interactive mode of the program. You can see all the options to the program by running `water -help`.

You can now plot the scores just like we did above for the scores obtained with the original protein.

However, we want to go a step further and try many different shufflings, so we can calculate averages. We can run it any number of times using a "for loop". Using the same command as above, it looks like this, if we do it 100 times

```
for i in {1..100}
do
shuffleseq -sequence GRPE_YEAST.fa -outseq stdout | water -brief -asequence
    ⟶ stdin -bsequence Ecoli.prot.fasta -outfile stdout -gapopen 11. -
    ⟶ gapextend 1. | awk '/^# Score:/ {print $3;}' >> 100shuffles
done
```

Each time the loop runs, a new shuffling is made of the sequence. The symbol >> means that the output is appended to the file. If you used only one > the file would be over-written each time. After running this (which takes something like 15 minutes depending on the computer) we plot the result as before (re-plotting everything to get the axes right)

```
s100 <- unlist(read.table("100shuffles",head=F))
tmp<-table(s100)
counts <- as.vector(tmp)
C100 <- rev(cumsum(rev(counts)))/100
S100 <- as.numeric(names(tmp))
plot(S100, C100, main = NULL, xlab="score", ylab="Cummulative count", type="l",
    ⟶  col="blue", log="y")
points(Swt, Cwt,type="l")
curve(evalue, from=30, to=80, type="l",lwd=3, col="red", log="y", add=T)
```

The resulting plot is shown in Figure 3.2 (right). The fit of the theoretical model is not perfect, but the curve from the shuffled sequences is at least closer to the theoretical line.

Exercise 2 *Discuss again which factors you think are most important for the poor correspondence between the theory and the actual observed scores. In the table below, you can see the composition of the sequence and the database. Do you think that plays a role?*

AA	GRPE_YEAST	E. coli proteins	AA	GRPE_YEAST	E. coli proteins
A	0.070175	0.095070	M	0.021930	0.028008
C	0.000000	0.011560	N	0.043860	0.039309
D	0.065789	0.051634	P	0.048246	0.044342
E	0.100877	0.057745	Q	0.043860	0.044409
F	0.065789	0.039081	R	0.065789	0.055342
G	0.039474	0.073548	S	0.065789	0.058097
H	0.017544	0.022629	T	0.065789	0.053810
I	0.030702	0.060123	V	0.057018	0.070647
K	0.105263	0.043981	W	0.000000	0.015378
L	0.083333	0.106769	Y	0.008772	0.028498

(There happens to be 15 of R, S, and T in the protein, so it is not an error that they have exactly the same frequency $15/228 = 0.065789$.)

Exercise 3 *Redo the last experiment with 100 shufflings. Do it twice using two different proteins. You can pick the proteins from the E. coli database itself. Plot the average of the shuffled and the theoretical line. Finally run an experiment where you also shuffle the proteins in the database.*

What differences do you see between the experiments? How does the theory fit experiment?

Exercise 4 *Write a program that generates random sequences from a distribution over amino acids. Use the above frequencies for the E. coli database. Generate a database of random sequences of the same size as the E. coli database. You can make the same number of sequences (4183) each with a length equal to the average length (1335035/4183=319). Now generate 100 random sequences of fixed length and do the search of the random database as above. Plot the average (as above) over the 100 runs and compare to the theoretical line.*

How does the theory fit this experiment?

Chapter 4

String Matching in Biological Sequence Analysis[1]

4.1 Introduction

We have seen that biological "spelling" is very sloppy in the sense that sequences (DNA, RNA, and proteins) can vary and still do the same. Therefore it is often advantageous to use probabilistic models such as weight matrices and hidden Markov models. However, in many situations it is still of interest to look for exact sequence matches. First of all there are biological signals that are constant, such as restriction sites for restriction enzymes, and secondly, searching for short exact sequences, which we call "words" or "k-mers", can often be used as a way to speed up non-exact matching algorithms as is done in Blast. Searches for words are fast, even if it is done in the straight-forward way, but there are fascinating data structures that can make them even faster.

First, we need to establish what is meant by a word. In the context of this chapter, a word is defined as a sequence of letters of length k, which is usually short. Words are also referred to as k-mers. The kind of searches we are addressing in this chapter are of the form: locate all occurrences of a certain word (or a list of words) in a database of sequences.

New DNA sequencing techniques have dramatically increased the amount of data produced, and new computational techniques for dealing with this data are required - for instance to align the sequencing reads to a reference genome. A single experiment will often generate several Gb of DNA. The reads from some of these platforms are short, typically 100 bases, and therefore one often has to map millions of such short sequences on to a reference genome. Doing this using Blast or similar techniques is basically hopeless, and therefore many new methods have been developed that use exact string matching of various sorts.

The string matching techniques discussed in this chapter are highly developed in computer science, and the aim of this chapter is to give the reader an introduction to the field and an intuitive understanding of the techniques – we will just be scratching the surface.

[1] Parts of this chapter was used as the basis for the chapter "Algorithms for Mapping High-Throughput DNA Sequences", in a volume of "Comprehensive Biomedical Physics", Volume 6: Bioinformatics, Ed. Bengt Persson, Elsevier (ISBN: 978-0-444-53633-4). Therefore parts of the text may be identical.

Example: Blast

The initial stage in Blast [2, 3] is a search for words. First all words of size k in the query sequence are extracted, i.e. all $l-k+1$ overlapping subsequences of length k, if the sequence has length l. If it is a protein query, neighbor words are added to the list – those words that score above a certain threshold with the substitution matrix used. Then all occurrences of the words in the word list are located in the database. These matches are then extended and alignments are constructed (as discussed elsewhere). A similar procedure is used in the Fasta program [4] and many other programs.

Sequence analysis based on k-mers

Due to the data explosion from next-generation sequencing many methods have been developed that rely on k-mers instead of old-fashioned alignment.

In metagenomics, for instance, one would often like to identify which bacterium each DNA read originates from, which is called binning. One could align the reads to all known bacterial genomes to find the closes matches, but this will be a computationally demanding approach. Instead, many methods nowadays would find which genomes k-mers from a read are found in, and do some analysis based on this to identify the most likely origin of the read. Looking up exact matching k-mers is very fast compared to alignment (as we shall see).

Apart from mapping and metagenomic binning, there are many k-mer based methods for diverse applications, such as RNA-seq analysis, genotyping, genome assembly, and genome comparison.

4.2 Hash tables

In programs like Blast, we have a set of words we need to look for in the database (e.g. the set of words in the query sequence). To find out if they occur in a long sequence, one can compare them one by one to every position in the target sequence. This will take time proportional to the number of words in the set multiplied by the length of the target sequence. This naive algorithm is too slow, and there exists a number of ways to do it faster. Many programs use so-called hash tables.[2] The idea in a hash table is to map words (or text) to numbers.

4.2.1 Word enumeration

Enumeration of words is a simple way to assign numbers to small words of fixed length. You can assign a unique number to every possible word in several ways, but the most common is simply to enumerate words alphabetically starting at 0. It is just like the construction of numbers in the 10 digit system, except that the base of the numbers is the size of the alphabet instead of 10. In mathematical terms, if the letters are numbered from 0 to $M-1$, the word number is $x_k + x_{k-1} * M + x_{k-2} * M^2 + \ldots x_1 * M^{k-1}$ where and x_i is the number corresponding to the ith letter in the word. It is illustrated in Figure 4.1 for DNA.

Assume you have a set of words of a given length k, which we call the query set, and you want to see if any of these words occur in a database or target sequence. In applications like Blast the query

[2] In Blast – at least the original version – a finite state machine is used, which we will not cover in detail.

Words	Enumeration		
AAAGT	256*0+64*0+16*0+4*2+3	=	11
AACAG	256*0+64*0+16*1+4*0+2	=	18
AGTTC	256*0+64*2+16*3+4*3+1	=	189
CCAAG	256*1+64*1+16*0+4*0+2	=	332
:	:	=	:
GCCTA	256*2+64*1+16*1+4*3+1	=	605
:	:	=	:
TTGAT	256*3+64*3+16*2+4*0+3	=	995

Figure 4.1: Enumeration of words. The four letters are numbered: A is 0, C is 1, and so forth.

Sequence	GTTCTAAGACATGT														
Words ($k=2$)	GT	TT	TC	CT	TA	AA	AG	GA	AC	CA	AT	TG	GT		
All 2-mers	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
Word number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Occurrence	1	1	1	1	1	0	0	1	1	0	0	2	1	1	1	1

Figure 4.2: Simple example of enumeration of 2-mers. In the top you have a short DNA sequence (length 14) and below the 13 words ($k = 2$) it contains. In the table below, you see the 16 possible 2 mers, their word numbers (second row) and how often they occur in the sequence (last row).

set would be the set of words from a query sequence and the target sequence would be sequences in a data base. You create an array as long as the total number of all possible words of the given length (M^k) and initialize all entries to 0. For all words in your query set, you set the array value to something diffent from 0, i.e. you calculate word numbers as above and for instance add a 1 in the corresponding entry in the array, as illustrated in Figure 4.2. When searching a target sequence for words in the set, you compute the unique number for each word in the target sequence and look up in the array if it is present in the query set. The time to do this is proportional to the target sequence length only – independent of the size of your query set.

The array of word occurrences can of course be replaced by an array with pointers to lists with the exact word locations.

Word enumeration is used in a wide range of programs in bioinformatics. In Exercise 5 below, the limitations of this approach is addressed.

Exercise 5 *Here we investigate whether word enumeration is always applicable. What is the possible number of words you can construct with an alphabet size of M? How much computer memory would be needed for enumeration of DNA words of size 10 if each integer takes 8 bytes of memory? What is the maximum realistic word size for enumeration of DNA words in a typical personal computer?*

Exercise 6 *Let us say you want to lookup words of length 5 in a long DNA sequence. Using the formula above, for every position i, you would have to make the calculation* $q_i = 256x_{i-4} + 64x_{i-3} + 16x_{i-2} + 4x_{i-1} + x_i$. *How could you reduce the number of calculations if you go through the sequence letter by letter? In other words, if the word number* q_i *is calculated, how can you calculate* q_{i+1} *with fewer mathematical operations?*

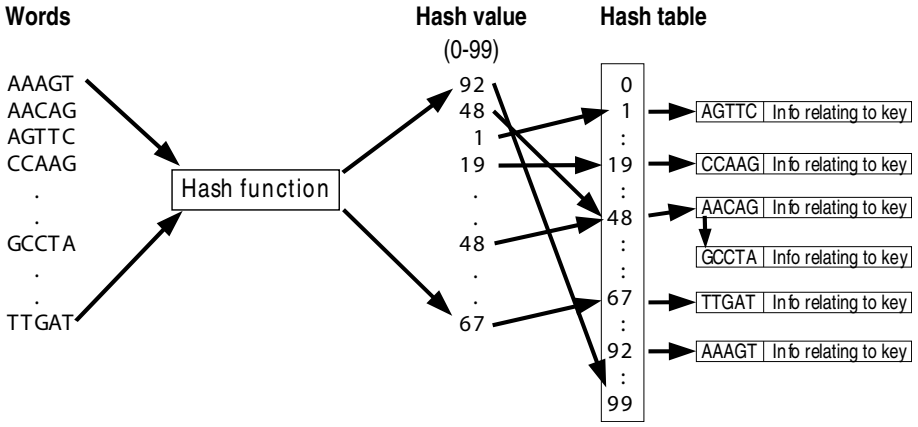


Figure 4.3: To the left is shown a list of words (size 5). A hash table of size 100 is constructed by obtaining a key (hash value) between 0 and 99 from the hash function. The word and any information regarding the word is stored in the table entry corresponding to the key. Note that the single collision shown is resolved by creating a linked list of records with all the words with the same hash value.

4.2.2 General Hash Functions

As we saw above (Exercise 5) enumeration requires too much memory to be feasible for long words. A hash table is an alternative to enumeration. It is not quite as fast, but much more memory efficient and, even more importantly, it is also applicable to words (or sequences) of varying lengths.

The basic idea of a hash table is the same as in enumeration: to map a word (or som other entity) to a number between 0 and some number N_{hash}. To do this, a *hash function* is required. A hash function must map the input, which we call the *key*, to a unique number Information relating to that key can be stored in a table under that number (for instance as a pointer to a data structure that holds the word along with the relevant information), see Figure 4.3. So for each item we want to store, a unique key is needed, and for this key, the hash function maps the key to the number of the bucket (table entry) in which it is stored.

Enumeration as discussed above is an example of a hash function where a word of size k is the key. Since all words yield different numbers, it is an example of a so-called perfect hash. For more general cases, it is impossible in practice to construct a function for which no two keys are mapped to the same number. When two words are mapped to the same number, it is called a *collision*. Therefore there are two important issues in hashing: construction of a good hash function and a method to resolve collisions.

The ideal hash function must minimize the chance that two different words are mapped to the same number, i.e., it should minimize the chance of collisions. This can be achieved if the hash function generates numbers that are evenly distributed in the interval from 0 to $N_{\text{hash}} - 1$ and ensures that similar keys have uncorrelated hash values.

Collisions are normally dealt with by storing a linked list of keys that map to the same number. Then, when you look up a key, you search by normal string comparison through the linked list to find the right one. Clearly the size of a hash table compared to the number of records it stores is very important. If the hash table has a fraction f of the N_{hash} entries in use, the probability of a collision is f when the next item is entered (assuming a good hash function). Therefore the number of records stored in the table should normally be less than the size of the table. If a hash

table is over-full, you may almost loose the advantage of a hash, and therefore schemes have been developed for dynamically enlarging tables.

Hash tables are built-in data structures in many high level programming languages such as perl and python (called dictionaries in python), and they can be used for most purposes without the user having to worry about the underlying algorithms.

Example: A simple hash function

It is easy to construct some number from a word (or any key). In computers, letters are represented as numbers between 0 and 128 in a byte. This is the ASCII character encoding, where the capital letters start at number 65 and lower case letters at 97. In programming languages you can easily get the numerical value of characters (in Python, the function "ord" returns the number, so ord('a') returns 97). You can for instance add up all the letters in the key to get a number and take modulo N_{hash} (the modulo gives the remainder after integer division, so e.g. 37 modulo 5 is 2). This will give a value between 0 and $N_{\text{hash}} - 1$. The word "ABcd" would give the number 65+66+99+100=330 and if the hash table has size 100, we would end up with the number 30.

This hash function is far from perfect. For instance, any permutation of letters in the key will give the same value. The numbers resulting from such a simple function may also be far from uniformly distributed, which increases the probability of collisions. Therefore real hash functions use more elaborate operations on the keys to make them more "random" (but don't forget that the same key must always give the same value!).

Exercise 7 *Assume we store N different items in a hash table of size N_{hash}. We define the fill factor $\lambda = N/N_{hash}$. This is the average number of keys mapping to each bucket. Assume that the hash function is ideal and assigns a uniform random number to the key. You can use the Poisson approximation of the binomial distribution in the following.*

a) What is the probability $P(n = 0)$ that there are 0 keys mapping to a given bucket as a function of λ? What is the probability $P(n > 1)$ that there is more than 1 key mapping to a given bucket?

b) What is the average fraction of buckets that have collisions if the fill factor is 1?

c) Make a plot of the probability $P(n > 1)$ as a function of λ. At approximately what fill factor do we have collisions in half the buckets?

4.3 BLAT: Indexing the database

Blast and Fasta search the database for a list of words from the query sequence. If you turn this around and instead make a word index of the database and search the query for occurrences of database words, the search can in principle be done in time proportional to the length of the query sequence. This may speed up the search by several orders of magnitude, but is not as easy as it sounds, because one has to store all the locations of all words in the (potentially very large) database. All this has to be stored in computer memory (RAM) in order to benefit from the gain in speed. Some programs like BLAT do exactly that [5].

BLAT is a fast tool to locate sequence matches of high similarity. It is perfectly suited for identifying the genomic location of EST, cDNA or similar sequences, which you expect match almost perfectly over a reasonable length.

In BLAT only the *non-overlapping* k-mers are indexed to save memory, and all k-mers of the query sequence are compared against the index. Some simple calculations are helpful for understanding the power of this approach.

Assume that on average there is a fraction q of exactly identical letters between a query and a match. Then the probability that a particular k-mer matches perfectly is q^k. If there are w indexed words in the match, the probability that *no* words match is

$$P_{\text{miss 1}}(q,w,k) = (1-q^k)^w. \tag{4.1}$$

We can also estimate the number of words that match a query at random. If we assume that all M letters are equally likely, the probability that a random word is identical to any specific word in the index is $(1/M)^k$. There are L/k non-overlapping words in the database index and we are testing around l words in the query ($l-n+1$ to be precise), so the expected number of random matches is

$$N_{\text{random}}(k,l,L,M) = l(L/k)(1/M)^k \tag{4.2}$$

Values for various word sizes and some reasonable parameters are shown in Table 4.1. Note that for short words, we would have many random matches, and for long words, the possibility of missing a match increases. To overcome this problem, BLAT uses the same trick as Fasta and other programs and requires two or more matching words in a matching region. The probability that *no* two words out of w match is the probability of having no match ($(1-q^k)^w$ as above) plus the probability of having exactly one match. The probability of having exactly one match is $wq^k(1-q^k)^{w-1}$, so the probability of *not* having two or more matches is

$$P_{\text{miss 2}}(q,w,k) = (1-q^k)^w + wq^k(1-q^k)^{w-1}. \tag{4.3}$$

(The distribution of matches is binomial). The corresponding values are also shown in Table 4.1. The probability of a random match with two identical words is a bit more difficult to calculate, but it turns out to be roughly squared relative to the one word case, so requiring two word matches, will bring the number of random matches significantly down.

By default, BLAT uses two 11-mer matches for DNA, which seems like a good compromise (see Table 4.1). The two matches have to be on the "same diagonal", which means that they have to be close to each other and in the right order. It is implemented by sorting word-matches on the diagonal number, which is the position of the match in the database minus the position in the query (see Figure 4.4). Two matches that are co-linear (two words in a match with no insertions and deletions) give the same value of this number. Say the first word matches at position i in the genome and j in the query and the other word matches at position $i+56$ in the genome and $j+56$ in the query. Then the difference is $i-j$ for both the word matches. So if the diagonal numbers

word size	7	8	9	10	11	12	13	14
$P_{\text{miss 1}}$	5.2e-8	2.1e-6	1.8e-5	0.00011	0.00052	0.002	0.0065	0.0093
$P_{\text{miss 2}}$	8.9e-07	3.5e-05	0.00029	0.0017	0.008	0.03	0.095	0.13
N_{random}	2.6e+06	5.7e+05	1.3e+05	2.9e+04	6.5e+03	1.5e+03	3.4e+02	80
$N_{\text{random 2}}$	1.6e+02	8.7	0.49	0.027	0.0016	8.9e-05	5.1e-06	3e-07

Table 4.1: The probability that a matching region is missed with one ($P_{\text{miss 1}}$) or two ($P_{\text{miss 2}}$) matching words for $q = 0.95$ and a matching region of length 100. Last rows show the expected number of random one-word matches and two word matches for a DNA with a database size of 3 billion bases and a query of length 100. The expected number of two word matches is approximated by $l(L/k)(1/M)^{2*k}$, which a bit too low.

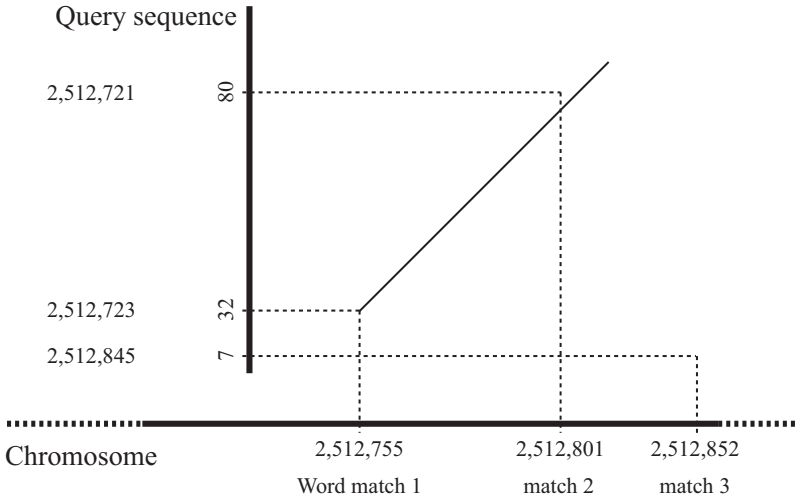


Figure 4.4: The query have three word matches close to each other in a chromosome. When calculating the diagonal number (chromosome position minus query position), those that are close to the same diagonal give almost the same. This is the case for word match 1 and 2, but match 3 does not fit any other others. The diagonal number is shown to the left of the positions in the query sequence.

are similar, the two matches "makes sense" and the match is analyzed further to see if it can be extended beyond the two matching words.

4.4 Trees

Strings can also be organized in tree data structures for fast searching. As we saw with Blat, indexing of the database can make searching faster, and using tree data structures may – in theory at least – be even better. Methods for constructing and searching trees of various types is a big field in computer science, and we will only scratch the surface here and cover some of the most relevant topics.

4.4.1 Tree terminology

Computer scientists don't get out much and therefore computer trees are drawn upside down. The root is the dot at the top. The branches are called edges and they are put together at the nodes. This terminology is borrowed from graphs – formally, a tree is a graph with no cycles (acyclic) and connected (no unconnected islands). Figure 4.5 shows an example of a tree. A node has zero or more child nodes, which are the nodes below directly connected. Nodes immediately above are called parent nodes and leaf nodes are the nodes without children. Internal nodes are those that are not leafs and root. From the root you can go to any other node by a unique path.

4.4.2 Prefix trees

If you have a number of words over an alphabet of size M they can easily be organized in a tree in which the root and all internal nodes have up to M children, one child per letter in the alphabet. The tree is constructed such that each path from the root to a leaf spells out a word in the list.

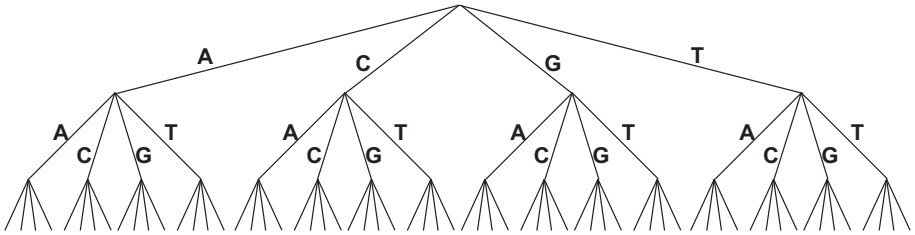


Figure 4.5: A simple tree representing all DNA words of length 3.

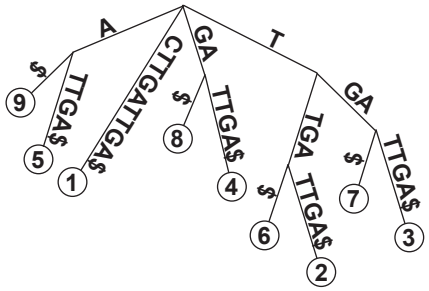


Figure 4.6: A suffix tree for the sequence CTTGATTGA$. The $ is the termination symbol.

In this tree all words descending from a common node has the same beginning or prefix, and therefore it is called a prefix tree. It is also called a *trie*.

Now it is easy to check if a word is in the list by descending the tree letter by letter until you can't come any further (the word is not there) or until the word is exhausted and you are at a leaf – in which case you have a match. The internal nodes may also correspond to words in the list (say that both "bicycle" and "bicycles" are in the list, then the first correspond to the internal node).

The time it takes to search the tree is proportional to the length of the word and independent of the number of words in the original list, just like word enumeration. Thus the tree is an alternative to enumeration, if your words are too long to enumerate, and the number of words you want to index is small enough that the tree fits in the (RAM) memory of the computer.

As we shall see later, it is also possible to search for inexact matches to words in a tree, and this is a major advantage over enumeration and hash tables.

4.5 Suffix trees

A suffix of a string is a substring that extends all the way to the end of the string. A suffix tree represents all possible suffixes of a sequence and turn out to a be very convenient data structure for searching. Suffix trees are covered in depth in [6].

You can construct a tree with all possible suffixes of a string (such as a chromosome), just like constructing a prefix tree. In this tree all internal nodes correspond to sub-strings and leaf nodes correspond to the suffixes. All suffixes are represented by a path from the root to a leaf.

In a suffix tree nodes with only one parent are eliminated, and several letters are put on the resulting edges, see Figure 4.6 for an example of a suffix tree.

For convenience we will add a termination symbol $ to the end of the sequence (letter $l+1$). This termination letter must be different from other letters in the sequence and is needed for suffixes

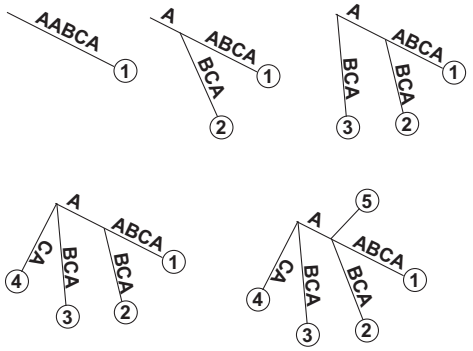


Figure 4.7: Construction of the suffix tree for the sequence AABCA begins with one branch for the complete sequence (suffix 1) and then adds suffixes one by one, from left to right in the figure. Here the termination symbol is not shown

that are prefixes of another suffix. A bit of terminology: suffix i of a sequence $x_1x_2\ldots x_l$ is the subsequence $x_ix_{i+1}\ldots x_l$ from letter i to the end of the sequence and it is called s_i. The first suffix s_1 is identical to the sequence.

It is straight-forward to construct a suffix tree by the following naive algorithm: Make a branch corresponding to the whole sequence, s_1. For suffix $i = 2\ldots l$ add suffixes one after the other by matching the suffix to the tree letter by letter until impossible, at which point the rest of the suffix is added as a new branch. Figure 4.7 shows an example.

Exercise 8 *What is the worst-case run-time for this naive algorithm? Hint: Think of a sequence of length l consisting of just one letter repeating. What do you think would be a more realistic run time estimate?*

Algorithms for suffix tree construction in linear time exists, see [6] for details.

4.5.1 Generalized suffix trees

So far we have considered suffix trees for single sequences. A generalization to many sequences is easy. One possibility is to concatenate all sequences with a termination symbol in between each pair. The suffix tree built from the concatenated sequence can then be used for searching just as above, but the retrieved suffix number of course has to be translated back to the original sequence. A query sequence can not match across sequence boundaries, because of the termination symbol. Another solution is to add the sequences one by one, each with a specific terminating character and label each leaf with the sequence id and the suffix number in that sequence.

The advantage of a (generalized) suffix tree over the other trees described is that we need not decide on a maximum word length – one can match a sequence of any length to a suffix tree. Thus we have a data structure, which can be used to search large genomes for exact matches much faster than e.g. a Blast search. However, this comes at a price: for this to be efficient the whole tree needs to fit in the computer memory, and for large genomes quite a lot of memory is needed. Secondly, it takes time to build the tree, so it is only worth-while if the genome is to be searched many times.

Exercise 9 *Can you estimate the memory usage for a suffix tree for a sequence of length l? Assume that both pointers and integers are four bytes. Hint: Think of a sequence consisting of just one*

repeated letter and assume that each internal node needs a pointer to each child and information about the sequence on each edge.

4.5.2 Inexact matching and seed matching

Very often we need to do inexact matching in biology, and then things are less straight-forward. However, algorithms exists that can search suffix trees with up to a fixed number of mismatches or insertions/deletions, both exact and approximate. We will not go into detail with this, but just show a very simple approach.

Imagine that you want to find all matches with at most one mismatch. You first divide the sequence in two halves and match them individually exactly. One of the two have to match exactly – otherwise you would need at least two mismatches for the complete sequence to match. So if none of them match you already know that there is no match. If one half match, you can check at all the matching locations in the target to see if the extension to the second half match with only one mismatch.

In many programs an exact seed match is required. For instance, you may require that the first 20 bases of a DNA sequence match, because then you have a limited number of possible match positions to check the extension, and you can also continue search in the suffix tree testing all possible mismatches after the first 20 bases.

4.5.3 Position specific scoring matrices

Remember that a position specific scoring matrix (PSSM or weight matrix for short) contains a score $s_i(a)$ for seeing letter a in position i of the pattern the matrix represents. The score of any subsequence of the same length as the matrix can easily be obtained by systematically descending all possible paths of the suffix tree to the depth equal to this length and adding matrix scores along the way. This is called a depth-first traversal, where you start by taking e.g. the right-most branches first, then when the length of the matrix is exhausted, you have the first subsequence and its score. Then you back up until the first node, from where you descent along the right-most unvisited branch or back up further, if all branches have been visited. This you continue until you have been through the whole tree. If many subsequences occur multiple times in the genome, the advantage as compared to a naive search is that you only score each unique sequence once.

Usually you are not interested in matches below a certain cut-off (c), because those that score below are unlikely to be the binding sites or whatever you are looking for. By calculating the best possible score of any sequence from matrix position i to the end of the matrix, you can see if it is possible to reach a score above c if you continue. If this is not possible, you can stop the search and thereby skip a whole sub-tree. If the width of the matrix is k, it corresponds to calculating a local threshold for each position, which is:

$$c_i = c - \max_a s_{i+1}(a) - \max_a s_{i+2}(a) - \cdots - \max_a s_k(a) = c_{i+1} - \max_a s_{i+1}(a). \tag{4.4}$$

This is done recursively from the end of the matrix and setting $c_k = c$. Whenever the score accumulated up to position i does not exceed c_i, one can back up to the nearest parent node. If the cut-of is restrictive, this can save a lot of computing. By the way, this type of look-ahead scoring can easily be used also to speed up a naive search with a weight matrix.

If you have a query sequence $x_1, x_2, \ldots, x_k$, you can construct a weight matrix of length k with a score of 0 for the correct letter at each position ($s_i(x_i) = 0$) and a score of -1 for all other letters

($s_i(a) = -1$ if $a \neq x_i$). A total score of -1 with such a matrix corresponds to a match with one mismatch to the original sequence, a score of -2 corresponds to a match with two mismatches, etc.. So to find matches with up to n mismatches, you just have to do the matrix search with a cut-off of $-n - 0.5$. This is one way of searching for sequences with up to a fixed number of mismatches.

4.5.4 Suffix arrays

Suffix arrays are related to suffix trees, but are generally easier to implement efficiently. A suffix array is a lexicographically sorted list of suffixes each represented with the suffix number. A suffix array can be constructed in time similar to the time it takes to build a suffix tree.

Here you see the suffix array for the sequence `CTTGATTGA$`. The terminating $ character is sorting first, so the very last suffix in the sequence is the first in the suffix array.

10	`$`
9	`A$`
5	`ATTGA$`
1	`CTTGATTGA$`
8	`GA$`
4	`GATTGA$`
7	`TGA$`
3	`TGATTGA$`
6	`TTGA$`
2	`TTGATTGA$`

Searching for exact matches in the suffix array is like looking up a word in a dictionary. You first identify the suffixes starting with the first letter of the query (this will be an interval of suffixes). Next the sub-interval of suffixes that match the first two letters is identified, etc. You will end up with an interval of matching suffixes, which may of course be empty. Because the suffixes are sorted, you can use binary search to find the maches. In binary search you half the search interval in each step. If your initial interval is $[1, L]$, you check suffix $L/2$ (rounded to integer) in the suffix array to see if your query letter is lower or higher. If it is lower, you next check the midpoint of the interval $[1, L/2]$, if it is higher, check the midpoint of $[L/2, L]$. You continue like this until you have a match.

You can precalculate some things to make the search faster. This is the idea in a data structure called an enhanced suffix array (ESA), in which two other lists are used, so you can emulate the depth-first search of a suffix tree. One list is called lcp (longest common prefix) and stores the length of the longest common prefix with the previous suffix, and it tells you at what depth of the tree the suffix splits from the previous. Another is called skip, and it stores the number of suffixes to skip in the list to find a sorter lcp – that corresponds to backing up the tree to the nearest node.

It turns out that an ESA is more efficient than a suffix tree, both in terms of memory access and memory usage. However it still needs to store an integer for the suffix number, an integer for skip, a small integer for lcp (which can often be limited to a byte) and a character for the base for each base in the genome. That is, the memory usage is 10 times the genome size, if we assume four bytes per integer, or roughly 30 GB for the human genome (which is approaching the limit of 4 byte integers).

Read more about suffix arrays in [7] and [8].

4.5.5 Applications of indexing techniques

Two examples of applications of suffix trees in bioinformatics are as the first step for alignment of complete genomes [9] and for searching for common motifs in DNA sequences, such as transcription factor binding sites in promoters [10].

Suffix arrays are used in Last [11], which is a quite general search tool that can replace Blast for many applications and is a lot faster. A suffix array is built of the database and seed matches are found using binary search in the suffix array and then extended as in Blast. Contrary to Blast, it uses seeds of varying lengths, which can limit greatly the number of seeds that needs extension.

Enhanced suffix arrays are used in Vmatch, which is a versatile sequence matching program developed by Stefan Kurtz (http://www.vmatch.de/). For weight matrix searching, an open source package, PSSMsearcher, is made by three students from the Bioinformatics Centre [12] and also PoSSuMsearch does essentially the same [13].

The new sequencing technologies produces so much data that standard techniques are stretched to their limit, and therefore many new methods are currently being developed, which builds on various indexing methods. Often the sequences needs to be mapped to a known genome (often called the reference genome). The early methods for matching used a simple scheme to index the reference genome [14, 15]. Later methods used an index structure called the Burrows-Wheeler transform. It is similar to an ESA, but much more compressed, so these programs can run on standard hardware, whereas an ESA based approach may easily require huge amounts of RAM in the computer. These programs are the state-of-the-art and include BWA [15] and Bowtie [16].

Chapter 5

Weight Matrices

Proteins and DNA often contain binding sites and other signals of various kinds which are conserved to some degree. Such sites are rarely completely conserved. The eukaryotic doner splice site (see Box 1) is an example in which part of the pattern is an almost 100% conserved dinucleotide GT. It is so conserved that we will completely disregard those splice sites without the GT consensus. It would be easy to search a genome for all occurences of this dinucleotide. However, it occurs about every 16 base pairs in most genomes, and very few of those sites are actually splice sites.

Table 5.1 shows the occurence of bases around the donor splice site in a set of about 2000 known splice sites from vertebrate genes (data from [18]). Apart from the conservation of the GT dinucleotide, it is clear that no other bases are completely conserved. The two bases before the donor are 59% and 82% conserved, respectively, and there is conservation between 51% and 84% in the four bases following the GT, see the frequencies in Table 5.2. Pulling out sequences from this most conserved region, two bases upstream and 6 bases downstream from the splice site, does not give a single consensus sequence that accounts for a significant fraction of donor sites. The 40 most common sequences are shown in Table 5.3. The most frequent covers less than 0.5% of the sites. Therefore, searching for a single sequence cannot help in the identification of donor sites.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	516	597	713	1230	162	0	0	1062	1469	135	318	548	391	443	421
C	495	596	725	261	49	0	0	59	167	106	322	469	623	593	516
G	523	542	399	291	1692	2068	0	895	250	1731	362	645	487	543	626
T	534	333	231	286	165	0	2068	52	182	96	1066	406	567	489	505

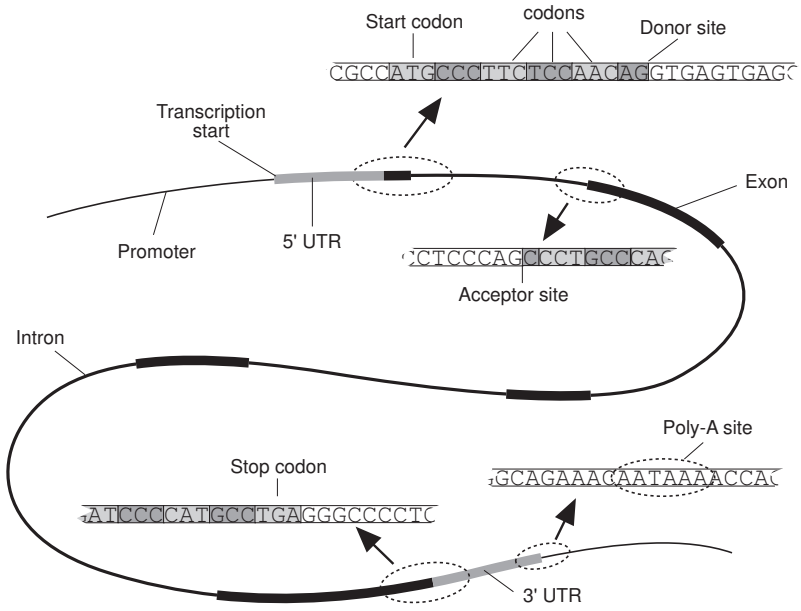
Table 5.1: The base counts for 2068 vertebrate donor sites. They are aligned such that splicing happens between position 5 and 6 with 5 bases before and 10 bases after the donor site. Notice the complete conservation of GT at position 6 and 7.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	0.25	0.29	0.34	0.59	0.08	0.00	0.00	0.51	0.71	0.07	0.15	0.26	0.19	0.21	0.20
C	0.24	0.29	0.35	0.13	0.02	0.00	0.00	0.03	0.08	0.05	0.16	0.23	0.30	0.29	0.25
G	0.25	0.26	0.19	0.14	0.82	1.00	0.00	0.43	0.12	0.84	0.18	0.31	0.24	0.26	0.30
T	0.26	0.16	0.11	0.14	0.08	0.00	1.00	0.03	0.09	0.05	0.52	0.20	0.27	0.24	0.24

Table 5.2: The base frequencies for the donor sites. These numbers are just the ones in Table 5.1 divided by the total number of sites (=2068).

Eukaryotic gene structure

Eukaryotic genes come in pieces called *exons* which are connected by *introns*. The drawing shows a single strand of DNA with a gene.



Exons are shown as thick lines. After assembly of the transciption complex, transcription of DNA to RNA starts at the transcription start site and ends around the poly-A site. The resulting pre-messenger RNA is *spliced*, meaning that the introns are cut out and the exons glued together into one continuous *messenger RNA* (mRNA).

The coding region of the mRNA, which is translated into protein, starts with a start codon (ATG) and ends with a stop codon (TAA, TGA, or TAG). Thus, there are untranslated regions (UTR) both at the 5' (left) and 3' (right) end.

The first two bases of an intron are almost always GT and the last two almost always AG. Note that introns can occur in the middle of a codon.

For more details, see for instance [17] or any other textbook.

Box 1: Eukaryotic gene structure.

5.1 The log-odds weight matrix

The most common method to search for a pattern like the donor site is to use a *weight matrix*, which is also called (more precisely) a position specific weight matrix or a position specific score matrix (PSSM).

It is assumed that each position in a site varies independently of other positions in the site. The probability that a sequence is a donor site can then be written as a product of probabilities. If $p_i(a)$ is the probability of letter a at position i in the site, the total probability of a sequence $x = x_1, \ldots, x_l$ is

$$P(x) = p_1(x_1)p_2(x_2)\cdots p_l(x_l). \tag{5.1}$$

AGGTGAGT	92	AGGTAAGA	43	AAGTGAGT	20	GGGTAAGT	16
GGGTGAGT	86	AGGTGGGT	40	CAGTGAGT	20	AGGTATGG	15
AGGTAAGC	67	ATGTGAGT	38	AGGTAGGC	19	GGGTGAGC	14
TGGTGAGT	62	TGGTAAGT	33	ATGTAAGT	19	TGGTATGT	14
AGGTGAGA	57	AAGTAAGT	26	AGGTCAGT	18	CTGTGAGT	14
AGGTGAGG	55	AGGTTGGT	26	TGGTAAGG	18	GGGTAAGA	14
AGGTAAGT	49	AGGTATGT	25	CTGTAAGT	18	GAGTAAGT	13
CGGTGAGT	48	AGGTAAAT	24	CAGTAAGT	18	AGGTAATT	13
AGGTGAGC	47	AGGTAGGT	24	TGGTGAGC	17	CGGTAAGT	13
AGGTAAGG	45	TGGTGAGA	21	TGGTAAGA	16	GAGTGAGT	13

Table 5.3: The most frequent donor sites with their number in the data set of 2068 donors. Two bases upstream and 6 bases downstream are included.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	0.00	0.21	0.46	1.25	-1.67	$-\infty$	$-\infty$	1.04	1.51	-1.94	-0.70	0.08	-0.40	-0.22	-0.30
C	-0.06	0.21	0.49	-0.99	-3.40	$-\infty$	$-\infty$	-3.13	-1.63	-2.29	-0.68	-0.14	0.27	0.20	0.00
G	0.02	0.07	-0.37	-0.83	1.71	2.00	$-\infty$	0.79	-1.05	1.74	-0.51	0.32	-0.09	0.07	0.28
T	0.05	-0.63	-1.16	-0.85	-1.65	$-\infty$	2.00	-3.31	-1.51	-2.43	1.04	-0.35	0.13	-0.08	-0.03

Table 5.4: The log-odds matrix corresponding to Table 5.1.

Here l denotes the length of the pattern (e.g. 15 for the donor sites above). Usually the frequencies of the letters are used as estimates of the probabilities, so for the donor sites one could use the numbers in Table 5.2. For instance, the probability of a sequence ACCAG... would be $0.25 \times 0.29 \times 0.35 \times 0.59 \times 0.82 \times \ldots$.

Some genomes have a very biassed base composition with e.g. a very high fraction of the bases A and T. In such a case, a position in a pattern with, say, high frequency of A is not as significant as if it occurred in a genome with very high G+C content. It is the same situation with protein sequences, where some amino acids are generally more common than others. To correct for this it makes sense to compare the above probability to a null model $Q(x)$. Therefore the *log-odds score*

$$\text{log-odds} = \log \frac{P(x)}{Q(x)} \tag{5.2}$$

is used to score patterns. If the probability $P(x)$ according to the model is greater than the probability $Q(x)$ according to the null model, then the log-odds is positive. Therefore, the higher the log-odds, the more likely is it that the sequence is the pattern described by the model.

Usually a very simple null model is used, in which each base is considered random and independent with the probability $q(a)$ for letter a. Then we can write the log-odds as

$$\text{log-odds} = \log \frac{p_1(x_1)}{q(x_1)} + \log \frac{p_2(x_2)}{q(x_2)} + \ldots + \log \frac{p_l(x_l)}{q(x_l)}. \tag{5.3}$$

Thus, it is a sum of terms $s_i(a) = \log \frac{p_i(a)}{q(a)}$, where there is one number per letter at each position. This is the weight matrix.

Table 5.4 shows the weight matrix for the donor sites, where the null model assumes same probability $(1/4)$ for each base. The logarithm base 2 was used, in which case the numbers are in *bits*, but a change of base for the logarithm just changes the scale of the numbers, so it is not important.

Note that the values directly show which bases are preferred at each position. The scores are $-\infty$ for bases that do not occur at a certain position in the data set, as seen at position 6 and 7

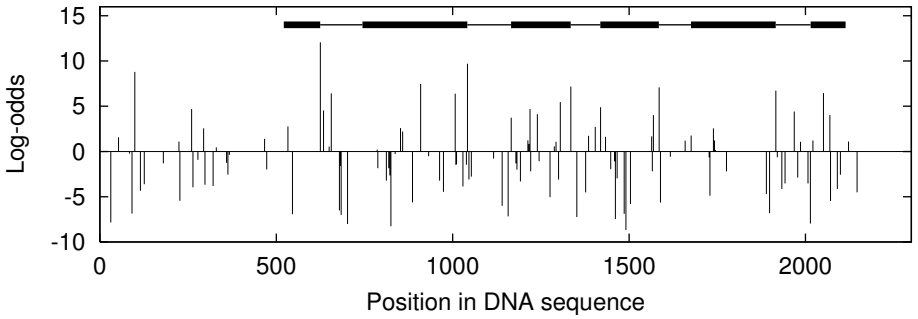


Figure 5.1: The donor weight matrix was used to scan a genomic sequence of a gene. The plot shows the log-odds score for all the windows with a GT in position 6 and 7 (all other have a score of $-\infty$). In the top of the plot the actual gene structure is shown with heavy black lines showing coding regions and thin lines showing introns. Most of the highest scoring sites are actual donor sites. Some 'false' donor sites have as high a score as the lowest scoring real donors. One difficulty in judging the result is that there might be alternative donor sites (alternative splicing) and there could be donors upstream and downstream from the gene shown, because non-translated regions are not shown.

where only one base is allowed. This essentially means that sequences like that are 'illegal' and all together get a score of $-\infty$.

5.2 Searching

The idea, of course, was to identify *new* patterns. It is easy to scan a sequence with the weight matrix by simply calculating the score for each window of length l in the sequence, and those with a high score are likely to be such a pattern. A scan with the donor site matrix is shown in Figure 5.1. The actual donors are among the highest scoring sites, but many sites have scores larger than 0 without being real donors.

The main interest would often be to search a genome for donor sites. Judging from Figure 5.1, it will not be trivial to distinguish the true donor sites from other sites. How often do we expect to see a falsely predicted doner site with a score larger than some threshold? One way to get a quantitative answer to this question is to use the score distributions in actual sequences (rather than trying to estimate it analytically). All the coding regions longer than 30 bases were cut out of the set of vertebrate genes, and so were all the introns longer than 30 bases. This yields 4615 sequences where we would not expect to find complete donor sites (apart from an occasional alternative splice site and a few due to errors in the data). A histogram of scores is shown in Figure 5.2.

The two histograms overlap quite a lot, so it is not possible to perfectly predict donor sites with this weight matrix (or any other currently available method, for that matter). As always we are faced with a trade-off between the number of falsely predicted sites (false positives, FP) and the number of true sites missed (false negatives, FN) when selecting a threshold for the score. We can count how large a fraction is above a certain cut-off value[1]. Multiplying this fraction by 1000 yields the expected number of false predictions in 1 kb. This is shown in Figure 5.3 along with the fraction of true donor sites that are missed.

[1]This is the fraction of *all* possible sites in the sequence, including those without GT at position 6 and 7.

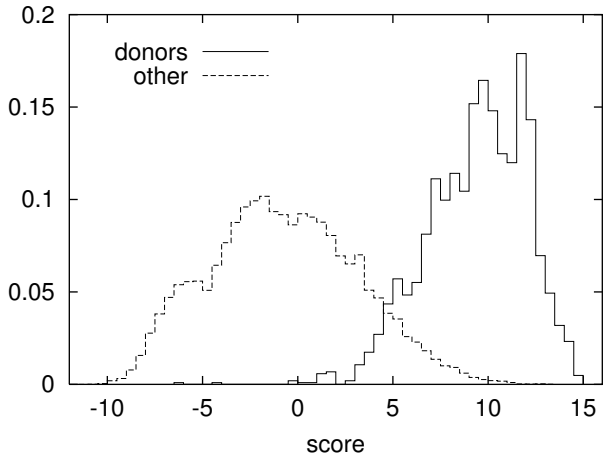


Figure 5.2: A histogram of scores for the intron/exon set of sequences to the left, and for the set of donor sites to the right. Scores of $-\infty$ for sites without GT at position 6 and 7 are disregarded for the intron/exon set.

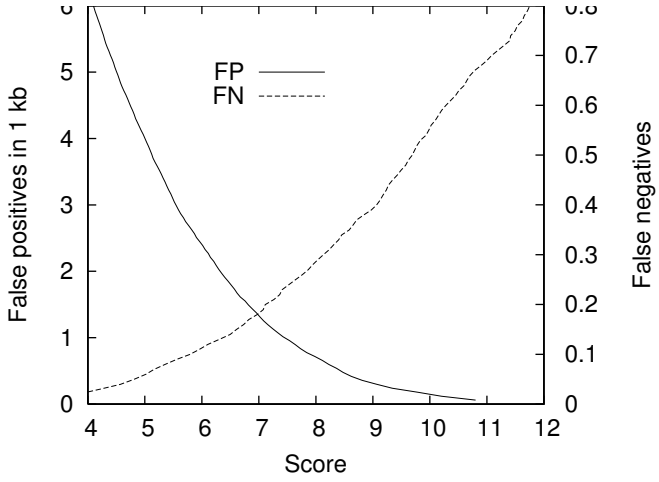


Figure 5.3: The decreasing curve shows the expected number of falsely predicted donor sites in 1000 bases (=1 kb) as a function of the score cut-off. The increasing curve shows the fraction of true donor sites that are missed at the same cut-off (right scale).

For an actual application of the weight matrix, one would have to figure out how many donors it is acceptable to miss and how many false predictions one can live with. It is not an easy situation, because with distances between donors of tens or hundreds kb in some human genes, one would have to select a very conservative threshold in order to avoid a flod of falsely predicted sites, and then a lot of true donors would be missed. At an expected rate of about 1 false prediction per 10 kb, as much as 70% of the true sites are missed.

5.3 A weight matrix for signal peptides

So far we have been concerned with patterns in DNA. Now, we will turn to a protein example, where weight matrices can also be very useful. Some proteins have a signal peptide in the N-terminus that tells the cell that they are to be exported. After export, the signal peptide is cleaved off, see Box 2.

Signal peptides

Picture of a signal peptide??

Proteins that are secreted from bacteria or eukaryotic cells have a *signal peptide* in the N-terminal. The signal peptide is the first 20–40 (typically) amino acids of the translated protein. It helps guide the protein to the membrane where the translocation aparatus moves it across the membrane and then *cleaves* off the signal peptide leaving the rest of the protein on the other side of the membrane. The cleavage happens at the *cleavage site*.

Some of the main characteristics of signal peptides are an initial region (the n-region) which is positively charged and a hydrophobic region (the h-region) before the cleavage site. Signal peptides differs in the details between eukaryotes and prokaryotes and even within these groups.

For more details, see for instance [17] or any other textbook.

Box 2: Signal peptides.

From a set of 1011 eukaryotic proteins with known signal peptides, a region around the cleavage site was extracted. Rather arbitrarily 10 amino acids before the cleavage site and 5 after was chosen. A weight matrix was contructed as described above with a uniform background distribution and used to predict the location of cleavage sites.

To do a more proper evaluation of the performance, cross-validation was used. The original set was randomly divided into 10 sets. For each set a weight matrix was contructed from the *other* nine sets, and then the sequences in the set were scanned with the matrix. In this way, no test sequence was part of the matrix estimation, but the method was tested on all the 1011 sequences. See Box 3.

The sequences consists of the signal peptide and 30 amino acids after the cleavage site, so they are not complete proteins. For each sequence the highest scoring position according to the weight matrix was found and compared to the position of the known cleavage site. A histogram of the differences is shown in Figure 5.4. A total of 393 of the 1011 cleavage sites were predicted correctly, but 967 are within ± 10 from the correct site.

To get an idea of the specificity and sensitivity, a histogram was made of the maximum scores for the signal peptides (although a large fraction correspond to the wrong location), see Figure 5.5. For comparison a set of 645 well studied sequences *without* signal peptides were scanned with the weight matrix, and a histogram of scores for *all* possible positions plotted. This is essentially the 'background score distribution'. The weight matrix is quite good at distinguishing between signal peptides and non-signal peptides.

When constructing the weight matrix it was assumed that the amino acid distribution is uniform. This is nto true. For instance tryptophan (W) and cystein (C) are quite rare with frequencies of about 1.5% in an average protein and alanine (A), leucine (L), and glycine (G) each account for more than 7-8% of amino acids. From the set of 645 proteins without signal peptides the frequency of each amino acid was found. The whole procedure was repeated using this as the background distribution for constructing the weight matrix. Now 470 of the 1011 cleavage sites were predicted correctly compared to 393 before, and 972 were within ± 10 from the correct site (967 before). The score histograms (Figure 5.6) improves quite a bit as well, even if it is hard to see. With a

Cross validation

When making predictions, one would normally want to estimate how good the predictions would be on *new data*, for instance, proteins or DNA for which there is no experimental knowledge. Often prediction methods builds upon known examples. It is a common mistake to estimate the performance of the method on the same data as it was derived from (the *training set*). However, it is easy to construct methods that can 'predict' the training data. Just think of fitting a curve to a noisy experimental time series: by choosing curves with enough degrees of freedom, it is easy to fit a the observations exactly. However, if the experiment is repeated, it is very unlikely that the new points will fall exactly on the same curve.

Therefore it is important to test a prediction method on examples which has not been used to construct the prediction method. One way to do this, is to divide the available data into two sets: one for constructing the method ('training') and one for testing. When only limited number of data are available it is often better to do *cross-validation*.

In cross-validation the data set is split into N subsets of roughly equal size (N is typically 5 or 10). Then N versions of the prediction method is constructed, each time leaving out a different one of the subsets for testing. In the end the test results are averaged.

Leave-one-out cross validation is the special case where each subset only contains one data item (e.g., a protein).

Box 3: Cross validation.

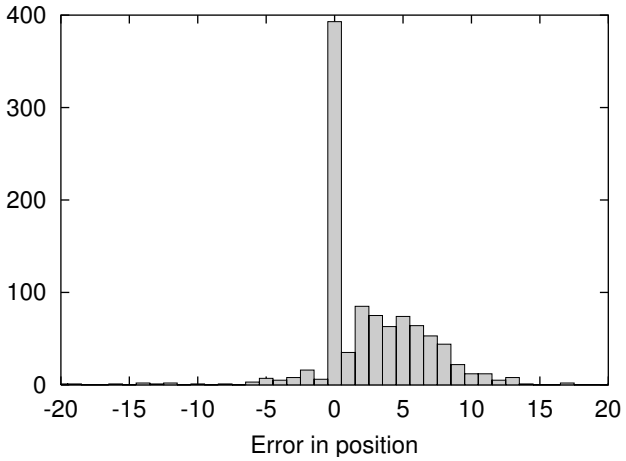


Figure 5.4: A histogram showing the position of the *actual* cleavage site minus the position of cleavage site predicted by the weight matrix. Note the assymmetry – the prediction is often before the true position.

score cutoff of 10, 364 sites are falsely predicted as signal peptide cleavage sites with the uniform background, whereas there are only 18 false positives with the non-uniform background.

This example clearly shows that it sometimes pays to take the background distribution into account.

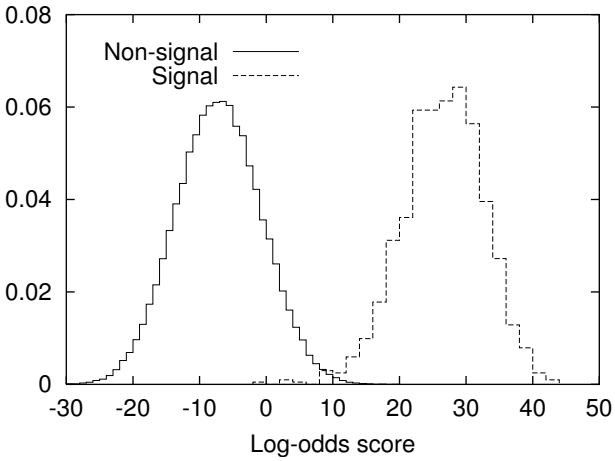


Figure 5.5: Histograms showing the distribution of the maximum score for signal peptides (to the right) and the distribution of all scores for a set of proteins with no signal peptides (left).

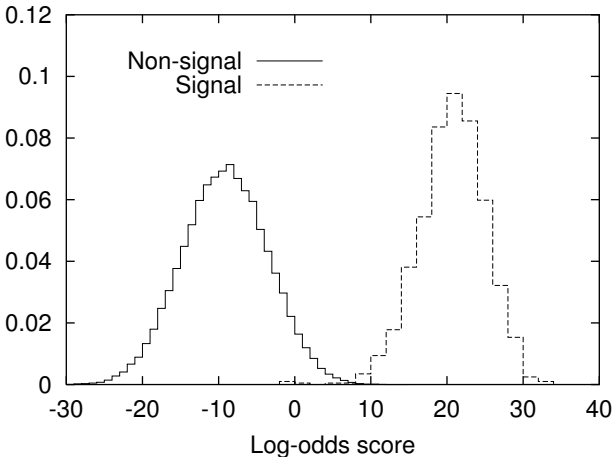


Figure 5.6: Histograms as in Figure 5.5, but using a realistic background distribution of amino acids, rather than a uniform distribution. The improvement is more significant than it might seem, because the small tail from a score of 10 of non-signal peptides in Figure 5.5 actually contains 364 sites, whereas it contains only 18 sites in this histogram.

5.4 Information in a site

It is possible to quantify the information in a certain pattern, e.g., a donor site. We could say that a pattern consisting of l totally conserved bases contains 'l bases of information'. For now assume that the base composition is uniform, so each base occur with the same frequency. Let us consider the *average* score of a pattern described by a weight matrix:

$$\text{Average score} = \sum_a p_1(a) \log \frac{p_1(a)}{q(a)} + \sum_a p_2(a) \log \frac{p_2(a)}{q(a)} + \ldots + \sum_a p_l(a) \log \frac{p_l(a)}{q(a)} \qquad (5.4)$$

where the sums are over the possible letters. If the pattern was completely conserved in each position, one p_i would be one and the rest zero. Therefore the average score is equal to $-l \log(1/N)$, where N is the size of the alphabet. If it is DNA, $N = 4$, and if we use the logarithm base 4, the average score would be l. So this number correspond to the number of bases.

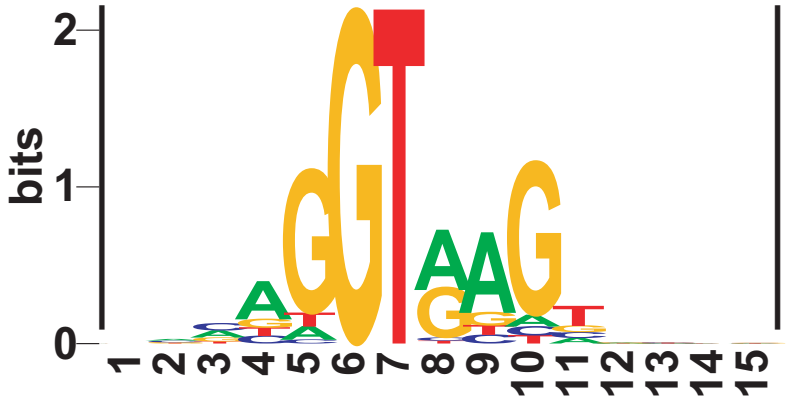


Figure 5.7: A 'logo' of the information content of each position in the donor sites. The total height show the information content at a specific position. The height of the individual letters is proportional to their frequency at the position. Splicing happens between position 5 and 6 (before the GT).

It turns out that it makes sense to use the average score as a measure of information in the pattern, also if the pattern is not totally conserved or if the base composition is not uniform. If we use the logarithm base 4, the information in the pattern tells how many completely conserved bases it corresponds to. Ususally the logarithm base 2 is used, and the results is called 'bits of information', and it is just twice the number of 'bases of information'.

Each term in (5.4) gives the contribution from one site to the average score. It is called the relative entropy and is defined

$$H(p||q) = \sum_a p(a) \log \frac{p(a)}{q(a)} \tag{5.5}$$

It is zero only if $p(a) = q(a)$ for all a, and otherwise it is positive. It can be thought of as a distance between probability distributions.

This enables us to quantify how much each position contributes to the score (on average). It is sometimes plotted in a so-called logo [19], with the additional assumption that q is uniform[2] using the logarithm base 2. A logo of the donor site is shown in Figure 5.7.

The same can be done for protein sequences, of course. In this case a completely conserved position would contain $\log(20) \simeq 4.32$ bits. A logo of the signal peptide cleavage sites is shown in Figure 5.8.

5.5 Interesting questions

- How to determine the matrix width?
- What's the score distribution in random sequences? Significance.
- What do you do when making a matrix from a small sample of sequences? (Regularisation.)
- Higher order matrices / non-independent positions.

[2] If q is uniform, $q(a) = 1/N$, the relative entropy can be written as $\log(N) + \sum_a p(a) \log p(a)$ where the last term (with a minus in front) is called the entropy.

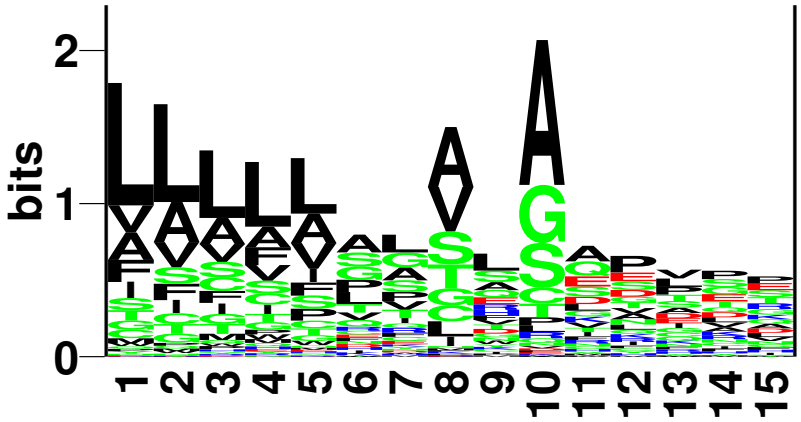


Figure 5.8: A logo of the eukaryotic signal peptide cleavage sites. The cleavage happen between position 10 and 11. Notice the hydrophobic stretch in the first few positions and the high occurrence of A just before the cleavage site.

Chapter 6

Identification of protein coding genes in genomic DNA

6.1 Introduction

Since the first draft of the human genome was published in 2001 it has been common to read in the newspapers and even in the scientific litterature that we now know all the human genes. That is (now, a couple of years later) a pretty crude approximation of reality: many genes are wrongly or incompletely annotated, some genes are not found, some of the annotated genes are probably not even genes and a lot of genes have splice forms that are not known. Therefore, scientists are still debating how many genes are present in the human genome. When speaking about genes, most people think of protein coding genes. If we turn our attention to other genes, such as those encoding regulatory RNAs, the situation is far worse. In this chapter, however, we are going to focus on protein coding genes entirely and leave the fascinating world of non-coding RNAs for another time.

Even in organisms much simpler than the human the situation is similar: a lot of genes in the data bases are wrong in various ways, and these errors propagate to the protein databases and sometimes to other genomes. This is why it is important to know how the gene identification is done, even if you're not in the business of sequencing or annotating genomes, in order to judge the quality of the gene identification for a genome.

Identification of genes is done by a combination of methods:

1. From the DNA sequence alone ("de novo").
2. By similarity to known proteins from other organisms.
3. By genome comparisons.

The two last points are of course related, but differ in the methods used. We start with the de novo prediction of genes, because it gives a good background for understanding the rest, and it is often the first task performed.

We will first discuss how to identify prokaryotic genes in genomic DNA. By many people this is considered a solved problem, but although it is certainly an easier problem than identifying eukaryotic genes, this view is wrong; it is far from trivial to identify correctly all the protein coding genes in a prokaryotic genome. Partly because of this misconception, the databases are full of wrongly assigned prokaryotic genes and the corresponding proteins.

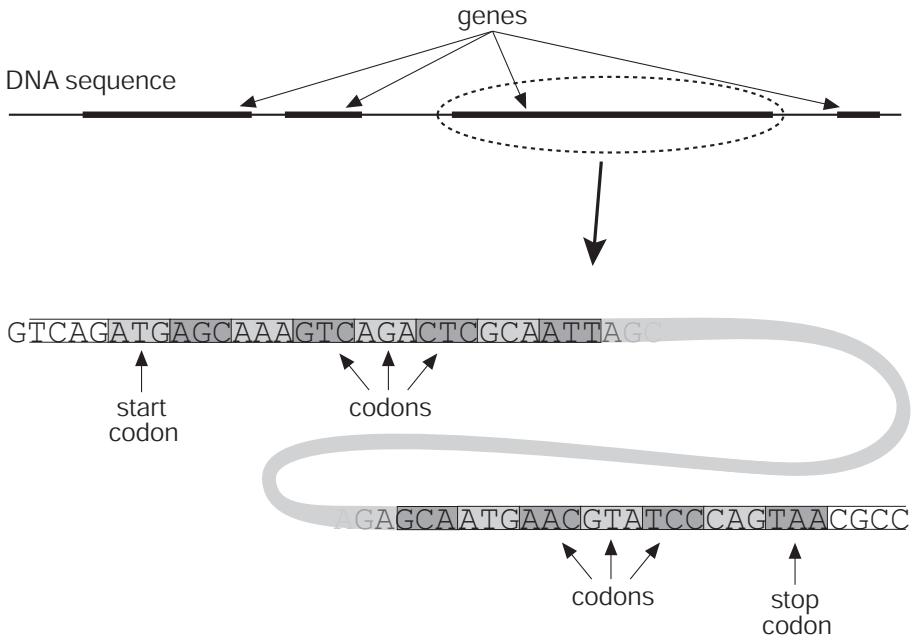


Figure 6.1: The drawing shows a single strand of DNA with several genes and a zoom below showing the structure of the coding region of one of the genes.

Genes are transcribed from DNA into RNA by the RNA polymerase. The polymerase and various other factors bind upstream of the transcribed region (the transcript) and initiates the transcription at the transcription start site. This upstream region is called the promoter and usually contains various binding sites for molecules that regulates the gene expression; these are called transcription factors. In prokaryotes, which we are focussing on first, the transcription can be terminated in two different ways. In Rho independent termination a hairpin structure is formed in the transcript, which causes the polymerase to fall off. In Rho dependent termination the Rho protein causes the termination in a poorly understood manner. A transcript can contain more than one gene. In this case it is called an operon, because genes on the same transcript are often functionally related, and it makes sense to transcribe them in exactly the same amounts.

A protein is coded for by a coding region on a transcript. It is translated from RNA to protein by the ribosome, which binds just before the coding region starts at the ribosome binding site and initiates translation. In prokaryotes translation can start while the gene is being transcribed and several ribosomes can translate the same gene simultaneously. The coding region contains a number of codons that each code for an amino acid in the protein. It starts with a start codon, which is often ATG, that codes for Methionine. In prokaryotes there are other possible start codons, but they are all translated to Met. The coding region ends at the a stop codon, where the ribosome stops translation. Usually the stop codons are TAA, TGA and TAG. Exactly which start codons are used depends on the organisms, and there are also some organisms that utilize a non-standard genetic code and slightly different stop codons.

The ”text-book protein coding gene” as discussed above and illustrated in Figure 6.1 is slightly simplified, because there are various effects that can alter the standard codon structure. One such example is a programmed frame shift, in which the ribosome jumps from one reading frame to another (see explanation of reading frame below).

Figure 6.2: A DNA sequence can be coding in three different reading frames on each strand. The figure shows a DNA sequence (one strand) and the translation to amino acid single letter codes below in the three frames such that the amino acid letter is aligned with the first base of the codon. The blue methionine (M) is the start of a gene and the start codon (ATG) is also colored blue. The frame with the blue start codon is open for the remainder of the sequence shown, whereas the other two frames contain stop codons. All stop codons are colored red and are shown with a red star in the amino acid translation.

6.2 Open reading frames

The most important signature of a protein coding gene is the so-called open reading frame or ORF. An ORF is a genomic region with a triplet (codon) structure that does not contain stop codons, although the precise definition varies a bit. An ORF can appear in six different reading frames; three on each strand, see Figure 6.2. Here we will define an ORF as a region that starts with a a start codon and ends with a stop codon, where the start and stop codons are those used by the prokaryote in question. By a maximal ORF we will mean the ORF with the most upstream start codon (i.e., the ORF cannot be extended without including a stop codon).

The ORFs are the gene candidates, but it is far from all ORFs that code for proteins. Essentially all DNA sequences contain ORFs, even a random sequence. This means that ORFs are everywhere – in promoter regions, on the complementary strand of protein coding genes, in RNA genes and so on. The question is then: how do we discriminate real coding ORFs from those ORFs that just appear randomly anywhere in the DNA sequence?

First of all, most of the ORFs that are not coding are relatively short compared to real genes. In fact, for a random sequence of letters the probability of finding an ORF of a certain length decreases exponentially with the length of the ORF, which we will see below. Therefore very long ORFs are usually genes.

Secondly, it turns out that genes have a different base composition in the ORF than one would expect if the DNA was random. This can be used to construct various measures for how "gene-like" an ORF is. Table 6.1 shows the frequency of observed codons in *Aeropyrum pernix*, an aerobic archaeon, which lives in almost boiling water (90-95 degrees Celcius). These codon frequencies differ from organism to organism. For these frequencies to be useful, we need something to compare to (a "null model").

Codon	Amino Acid	Frequency (%)	Expectation (%)	Log odds (bits)	Codon	Amino Acid	Frequency (%)	Expectation (%)	Log odds (bits)
AAA	K (Lys)	0.78	1.06	-0.4497	CAA	Q (Gln)	0.23	1.36	-2.5675
AAG	K (Lys)	3.14	1.36	1.2106	CAG	Q (Gln)	1.55	1.72	-0.1580
AAC	N (Asn)	1.63	1.36	0.2637	CAC	H (His)	1.16	1.72	-0.5783
AAT	N (Asn)	0.39	1.06	-1.4375	CAT	H (His)	0.44	1.36	-1.6170
AGA	R (Arg)	1.08	1.36	-0.3296	CGA	R (Arg)	0.14	1.72	-3.6543
AGG	R (Arg)	5.05	1.72	1.5499	CGG	R (Arg)	0.52	2.20	-2.0707
AGC	S (Ser)	2.51	1.72	0.5427	CGC	R (Arg)	0.44	2.20	-2.3307
AGT	S (Ser)	0.59	1.36	-1.2093	CGT	R (Arg)	0.25	1.72	-2.7841
ACA	T (Thr)	1.01	1.36	-0.4301	CCA	P (Pro)	0.77	1.72	-1.1727
ACG	T (Thr)	1.07	1.72	-0.6865	CCG	P (Pro)	1.25	2.20	-0.8126
ACC	T (Thr)	1.43	1.72	-0.2660	CCC	P (Pro)	2.18	2.20	-0.0090
ACT	T (Thr)	0.85	1.36	-0.6733	CCT	P (Pro)	1.18	1.72	-0.5447
ATA	I (Ile)	3.69	1.06	1.7928	CTA	L (Leu)	1.96	1.36	0.5316
ATG	M (Met)	1.98	1.36	0.5482	CTG	L (Leu)	2.98	1.72	0.7897
ATC	I (Ile)	1.07	1.36	-0.3361	CTC	L (Leu)	3.57	1.72	1.0511
ATT	I (Ile)	0.80	1.06	-0.4040	CTT	L (Leu)	1.51	1.36	0.1597
GAA	E (Glu)	1.09	1.36	-0.3203	TAA	Stop	–	–	–
GAG	E (Glu)	6.35	1.72	1.8805	TAG	Stop	–	–	–
GAC	D (Asp)	3.07	1.72	0.8337	TAC	Y (Tyr)	2.52	1.36	0.8942
GAT	D (Asp)	1.38	1.36	0.0289	TAT	Y (Tyr)	1.10	1.06	0.0503
GGA	G (Gly)	1.24	1.72	-0.4720	TGA	Stop	–	–	–
GGG	G (Gly)	2.66	2.20	0.2754	TGG	W (Trp)	1.38	1.72	-0.3171
GGC	G (Gly)	3.50	2.20	0.6729	TGC	C (Cys)	0.43	1.72	-2.0121
GGT	G (Gly)	1.57	1.72	-0.1328	TGT	C (Cys)	0.16	1.36	-3.0476
GCA	A (Ala)	1.45	1.72	-0.2494	TCA	S (Ser)	0.59	1.36	-1.1982
GCG	A (Ala)	2.12	2.20	-0.0483	TCG	S (Ser)	0.85	1.72	-1.0249
GCC	A (Ala)	3.75	2.20	0.7717	TCC	S (Ser)	1.18	1.72	-0.5521
GCT	A (Ala)	2.46	1.72	0.5107	TCT	S (Ser)	0.69	1.36	-0.9819
GTA	V (Val)	1.44	1.36	0.0873	TTA	L (Leu)	0.41	1.06	-1.3916
GTG	V (Val)	3.08	1.72	0.8360	TTG	L (Leu)	0.67	1.36	-1.0218
GTC	V (Val)	2.36	1.72	0.4531	TTC	F (Phe)	2.45	1.36	0.8520
GTT	V (Val)	2.30	1.36	0.7608	TTT	F (Phe)	0.56	1.06	-0.9238

Table 6.1: Codon usage table for *Aeropyrum pernix*, which has 44% A/T content.

6.3 Comparing with ORFs in random sequences

Although genomes are far from being random sequences of DNA, it is often a good idea to compare to random sequences, just like we do when calculating the significance of a database hit in Blast.

6.3.1 Length distribution of ORFs

What is the length distribution of ORFs in a random sequence?

First, let us consider a different problem: We want to find the probability of *not* roling a six in n rolls of a die. In each attempt the probability of a six is $1/6$, so the probability of not getting one is $5/6$. Since the rolls are independent, the probability of not getting one in n rolls must be $5/6$ to the power of n. This is an exponentially decaying function.

If a sequence is random with equal probability for the four letters A, C, G, and T, the probability of any given triplet being a stop codon is $3/64$, because there are 64 possible triplets and three

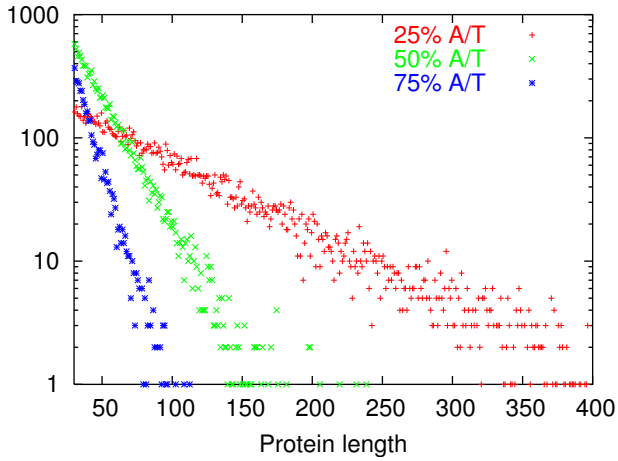


Figure 6.3: The plot shows the number of ORFs for each length in three random sequences of one million bases each. The sequences differ in their A/T content: one has 25% A and T, one 50% and one 75%. Lengths are measured in triplets (= number of amino acids).

possible stop codons codon (TAA, TGA or TAG). Then the probability of finding n or more codons is $61/64$ to the power of n in analogy with the dice example.

In a genome, the ORF can start in many different positions, so the expected number of ORFs longer than n triplets is proportional to the genome size. Because of dependence between the ORFs, the calculation of the constant of proportionality is difficult, but it can easily be found emperically from a semi-logarithmic plot like the one in Figure 6.3 (an exponential function is a line in a semi-logarithmic plot).

If the four bases occur with uneven frequencies, the length distribution changes, although it is still exponential, see Exercise 11. It is instructive to compare a sequence with a high A/T content and one with a low A/T content as in Figure 6.3. The exponential length distribution holds also for more complex random models (such as n-th order Markov chains), at least approximatively and for long ORFs.

Exercise 10 *The probability that an ORF is n codons or longer is* $(\frac{61}{64})^n$ *as calculated above. What is the probability that it is exactly n codons long, that is, the following triplet is a stop codon?*

Exercise 11 *Explain why ORFs tend to be longer in sequences with low A/T content as compared to those with high A/T content. Calculate the probability that an ORF is n codons or longer for a sequence in which A and T each occur with probability p and G and C each with probability* $0.5 - p$.

Exercise 12 *Why is it not wise to consider all ORFs longer than 100 codons almost certain genes, no matter which genome you're looking at. (It is actually a "method" you occasionally run into.)*

Is all this relevant? Yes it is. Take a look at Figure 6.4, which shows the length distribution of known coding ORFs and all ORFs in the *E. coli* genome. It is clear that the length distribution of ORFs is a blend of a distribution similar to the one in random sequences (linear part for short ORFs) and another one encompassing the ORFs that are actually coding for proteins.

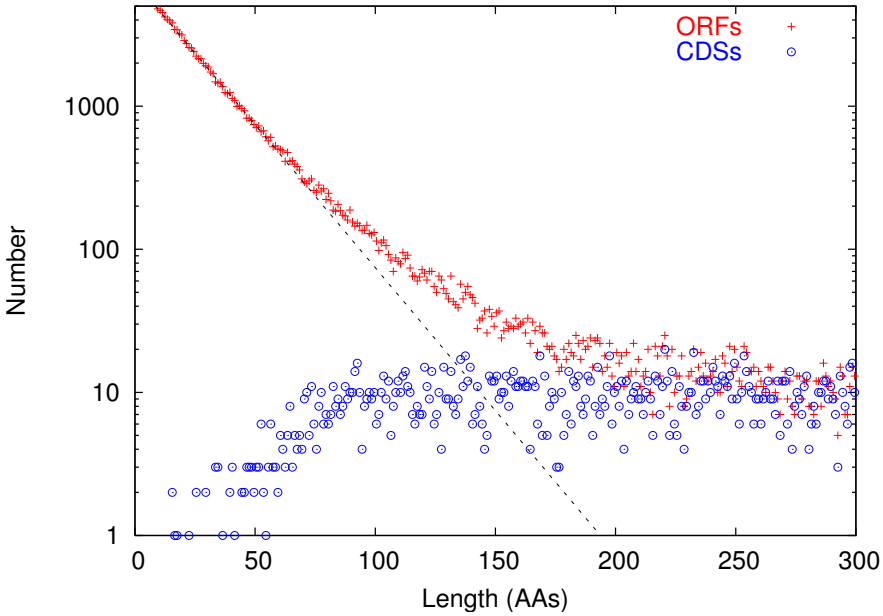


Figure 6.4: All open reading frames were extracted from *E. coli* and here their length histogram is shown in a semi logarithmic plot. The close fit to a line means that the length distribution is exponentially decreasing for small lenghts. The length histogram of genes that are known in *E. coli* is also shown (CDS=coding sequence).

6.3.2 Scoring ORFs

Let us get back to the problem of scoring ORFs. Rather than uniformly random sequences with equal probabilities for the four bases, it is generally better to take the base composition of the organisms genome into account. *A. pernix* is a G/C rich organism with an overall A/T content of 44%. (Because of the symmetry of the two strands in a genome, it will have the same number of As and Ts and the same number of Gs and Ts, and therefore one usually use A/T content or G/C content to describe the base composition). The frequencies of codons in a random sequence with the same A/T content is easily calculated. These frequencies are also shown in Table 6.1. The frequency of the CGA codon in genes is only 0.14%, so this particular codon for Arg is very much lower than expected. On the other hand the AGG codon also for Arg has a frequency of 5.05% although one would expect only 1.72% in a random sequence.

Exercise 13 *Verify that the triplet CGA appears with a frequency of* 1.72% *in a random sequence with 44% A/T as given in Table 6.1.*

To turn these numbers into a score for a given ORF, we do the usual trick of converting to log-odds (c.f. weight matrices). That is, we take the observed frequency, divide with the expected and take the log (base 2). Then we get a bit score for each codon, which is also shown in Table 6.1. To score an ORF we can now add these numbers, so for instance the score for the sequence AGGGAGCGA would be 1.5499+1.8805-3.6543. This score would generally be higher, the more 'gene-like' the ORF is.

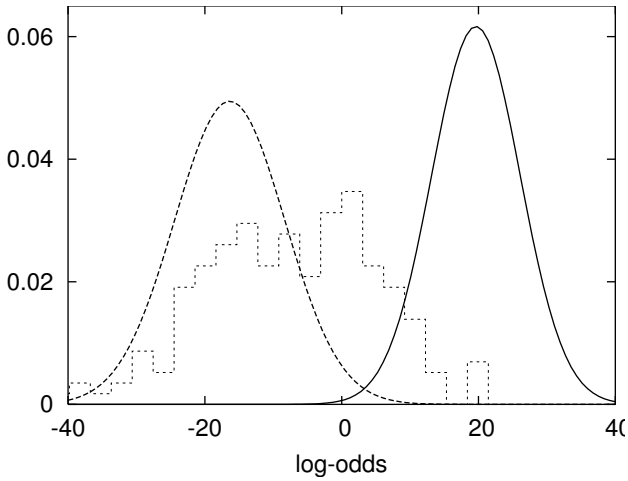


Figure 6.5: The theoretical score distributions for coding ORFs (full line) and ORFs in random sequences (dashed) for *A. pernix* using the approximation of a normal distribution (as described in the text) and assuming a length of 50 codons. The thin dashed line shows the score histogram for actual ORFs of length 50 found in the genome (188 in total).

6.3.3 When is a score significant?

When adding numbers like that, the result becomes normally distributed, if codons are independent. Even if they aren't, the normal distribution is often a pretty good approximation. In Figure 6.5 the theoretical normal distributions are shown for ORFs that code for proteins and for random ORFs both of a length of 50 codons. For the coding ORFs it was assumed that codons are random and independent with the frequencies from *A. pernix* in Table 6.1. For random ORFs it was assumed that the ORF was from a random sequence with the base composition of *A. pernix*. The two distributions overlap a bit, but there is a fairly good separation; if the log-odds is larger than about 10, it is unlikely that it is a random ORF. The overlap is decreasing with increasing lengths (because the averages of the normal distributions are proportional to the length, and the standard deviations are proportional to the square root of the length).

Although the overlap between the two distributions might seem small, one always has to put it in a genomic perspective. The more random ORFs of a certain length there are in the genome, the more likely is it that some number of ORFs have scores in the high tail of the distribution. So to calculate the significance of a score, one has to essentially multiply the score distribution with the expected number of random ORFs of a certain length. Both terms favor long ORFs, so they are usually no problem, but short ORFs (typically below 100-200 codons) can still be difficult to classify.

The separation between coding and random ORFs in Figure 6.5 is deceptive, because the random model is too simple for most genomes. If the null model is not good, bad discrimination results. In the plot the actual score histogram of ORFs of length 50 found in the genome is also shown, which does not look very similar to the theoretical. Usually, a higher order Markov chain will give more realistic result, which unfortunately usually means a poorer separation.

6.4 Gene finding programs for prokaryotes

The above method for scoring ORFs is based on codon usage, but there are many other ways of doing it. We will not go through them all, because they all try to capture essentially the same information in more (or less) sophisticated ways. One important extension is to use hexamer statistics, that is to use the frequencies of all $61 * 61 = 3721$ dicodons (and perhaps also "out of frame" hexamers).

One of the earliest gene finders is called GeneMark, and despite its age, it is still in use and remarkably good even compared to some of the newest alternatives. It is based on something called an inhomogeneous Markov chain for scoring reading frames. We will not go into the mathematical details, but conceptually it is actually not very different from the codon usage procedure we just went through. The model can calculate the probability that a certain sequence is coding in a given reading frame or whether it is non-coding. It then works by calculating these probabilities in windows of 96 nucleotides (96 is the default window size, but it can be changed by the user). If the average probability of all in-frame windows in an ORF is higher than some cut-off (default is 0.5) a proper start and stop codon is found and a gene is predicted.

6.4.1 Ribosome binding sites

Both GeneMark and other similar methods (like the codon usage sketched above) have trouble assigning the correct start codon. It is a quite common practice to always select the most upstream start of an ORF as the start codon. This is probably correct in more than half of the genes in most organisms, but it is often wrong. GeneMark and most other gene finders tries to predict the ribosome binding site in order to get a better prediction of the correct start.

A gene is translated to a protein by the ribosome. The ribosome binds to the ribosome binding site (RBS) just upstream of the beginning of the coding region on the mRNA. A part of the 16S ribosomal RNA (CCUCCU) binds to a little region which is approximately complementary (AGGAGG) on the mRNA. However, it is far from perfectly complementary. In Figure 6.6 (left) a logo of a set of upstream regions from *E. coli* is shown. Here you see the RBS in a region of low conservation about 10 bp upstream of the start codon. The sequences were aligned at the start codon (as you can see in the logo), and because the distance between the RBS and the start wary, the RBS's are not correctly aligned. After alignment of the RBSs, the logo looks like the right logo in the figure.

To identify the RBS, GeneMark uses a position specific scoring matrix as we learned about earlier. It then combines coding score and RBS score to find a start codon with a high scoring RBS about 10 bases upstream.

Newer gene finding methods like GeneMark.hmm and EasyGene *combine* the same type of model for the coding region and the RBS weight matrix into one hidden Markov model. With such a model, one can find the most probable gene (including the start with the most convincing RBS), and these gene finders are easier to use for automating gene finding, because the output is easier to interpret. They are also generally slightly more accurate.

In principle signals other than the RBS could be included in a gene model, such as promoter signals and transcription termination signals. Although there is work done in promoter prediction, it is rarely used in gene finding. This will probably change in the future, where methods are expected for prediction of whole transcripts.

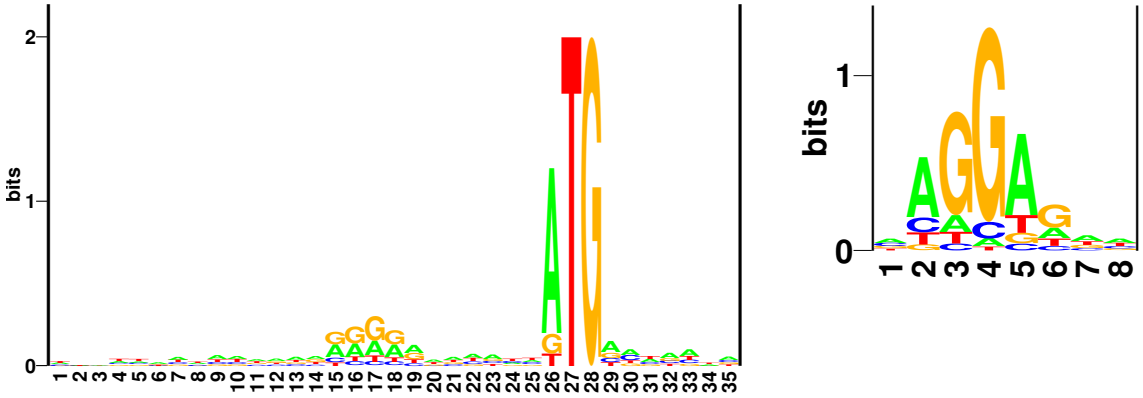


Figure 6.6: Left: Logo of DNA sequences from around the start codon of some *E. coli* genes. 25 bases upstream of the start and 10 bases of the gene was included in the sequences.
Right:After alignment of the *E. coli* RBSs the logo shows much more conservation.

6.4.2 Training data

It turns out that codon usage (and other features) can differ significantly from organism to organism. Therefore it is necessary to estimate codon usage statistics, or other statistics, for each individual prokaryote (although it is possible to group closely evolutionary related ones). To do this, one needs a reliable set of known genes to estimate the numbers from (often called a training set). The first gene finding methods used hand-curated sets of genes, that is, genes that specialists had faith in. This becomes inpractical when dealing with newly sequenced genomes.

There are at least two ways to overcome this problem and automatically construct sets of fairly reliable genes. The first is to consider all ORFs longer than a certain cut-off real genes. The cut-off should be at least 200 amino acids or should ideally depend on the G/C content of the organism. A draw-back of this method is that the correct start codon is not known.

Another, and probably slightly better, method is to take all ORFs and search a database of reliable proteins (e.g. Swiss-prot). ORFs with a significant match to a protein *from a different organism* are unlikely to be random ORFs, because genes are generally much more evolutionarily conserved than the rest of a genome. This set of ORFs can be used as training data. There is even a method to obtain a subset of these genes with certain start codons. This is illustrated in Figure 6.7.

These methods makes it possible to construct a gene finder for a given prokaryote completely automaticly, without having to have specialists manually look at all the genes. These methods are used by several modern gene finding programs, and although there is a risk of including a few wrong genes this way, they have proven sufficiently reliable.

6.4.3 Common errors

At the time of this writing about 200 prokaryotic genomes have been fully sequenced. Most of these are annotated by the group doing the sequencing, and each group typically use a different method. Therefore the annotation standard varies a lot, and there are many errors. These errors carry over to other resources, such as protein data bases and other genomes, where genes are annotated by similarity to wrongly annotated genes. Generally there are more errors in genomes that are rich in G and C. Here is a list of some of the most frequent errors.

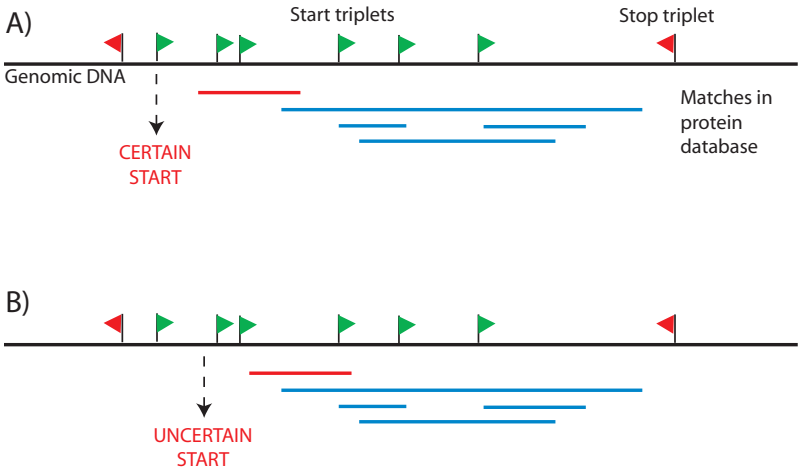


Figure 6.7: A: The black vertical line shows the genomic DNA with stop codons marked with red flags and start codons as green flags (in only one frame). The horizontal lines below (blue and red) show some matches to proteins in a database such as SwissProt. The most upstream match is coloured in red and it leaves only one probable start codon. B: The same as A except that the most upstream match is further downstream leaving three possible start codons.

Wrong start codons. As already mentioned, genes are often annotated with the most upstream start codon.

Lots of short ORFs are annotated as genes. As shown above, the real difficulty in gene finding is to distinguish the short random ORFs from the real protein coding genes. Some gene finders (like Glimmer and Orpheus) are particularly bad at this, and some annotaters insist that ORFs longer than 100 base pairs are genes.

Missing genes. Some genes have not been discovered. This is also mainly a problem with very short genes, because those are the ones that are difficult to discriminate from the random ORFs.

Sequencing errors and frameshifts. Almost all gene finding methods are assuming that genes are of the "textbook type" with an uninterupted ORF. Errors occur when sequencing DNA, which can break an ORF into two (overlapping) ORFs, if for instance a base is missed. Similarly frameshifts occur, which will also break the ORF. In most of these cases the automated gene finding methods will predict a single gene as two different genes or they will only predict one part of the gene.

The EasyGene gene finder (which the author is involved with) makes a serious attempt to deal with the first two problems with very good results. It is fully automated and is the first gene finder that use a measure of statistical significance for predicting genes. In summary it works as follows

1. Extract a data set with certain starts as described above.
2. Estimate a model (HMM, but similar to a weight matrix) for the RBS.
3. Estimate a complete model (an HMM) of the genes with coding region and including the previously found RBS model.
4. Calculate the significance of all the ORFs in the genome. The significance measure is essentially the expected number of genes predicted with the same log-odds score or higher in one Mb of random DNA.

5. Report all predictions below a certain cut-off in the significance measure (default 2).

6.5 Eukaryotes

In eukaryotes it is more difficult to find protein coding genes, because the coding regions come in pieces – the gene is split into exons and introns. The introns can be very long (frequently longer than 10 thousand bases in the human genome) and the exons very short (often less than 100 bases), so it is a bit like finding needles in a hay stack. Therefore the methods used for prokaryotes are not directly applicable in eukaryotes.

Any isolated signal of a gene is hard to predict. Current methods for promoter prediction, for instance, will have either a very low specificity or a very bad sensitivity, such that they will either predict a huge number of false positives (fake promoters) or a very small number of true promoters. The same is essentially true for splice site prediction: if looked at in isolation, splice sites are very hard to recognize with good accuracy. Therefore, to predict something like a splice site, you also need to predict coding exons and vice versa (disregarding the splice sites of introns in untranslated regions). In a long DNA sequence, you probably would not expect to see a coding exon with two associated splice sites unless there are other exons with which it can combine. In this way predictions of the various parts of a gene should influence each other, and prediction of the entire gene structure will also improve on the predictions of the individual signals. Therefore, in the last few years, gene prediction has moved more and more towards prediction of whole gene structures, and these methods typically use modules for recognition of coding regions, splice sites, translation initiation and termination sites, and some even use statistics of the 5' and 3' untranslated regions (UTRs), promoters, etc.. This combination of predictions has indeed improved the accuracy of gene prediction considerably, and as more knowledge is gained about transcription and translation, it is likely that the integration of other signals can improve it even further.

The most straight-forward way to combine recognition of splice sites and other signals with the "content" of coding regions and UTRs is through hidden Markov models.

6.6 HMMs for gene finding

One ability of an HMM is to model *grammar*. Many problems in biological sequence analysis have a grammatical structure, and eukaryotic gene structure is one of them. If you consider exons and introns as the 'words' in a language, the sentences are of the form exon-intron-exon-intron...intron-exon. The 'sentences' can never end with an intron, at least if the genes are complete, and an exon can never follow an exon without an intron in between. Obviously this grammar is greatly simplified, because there are several other constraints on gene structure, such as the constraint that the exons have to fit together to give a valid coding region after splicing. In Figure 6.8 the structure of a gene is shown with some of the known signals marked.

I will here briefly outline a straight-forward application of HMMs to gene finding with the weight on the principles rather than on the details. Apart from being the basis for most modern gene finding programs, it is also a very nice example of an application of HMMs.

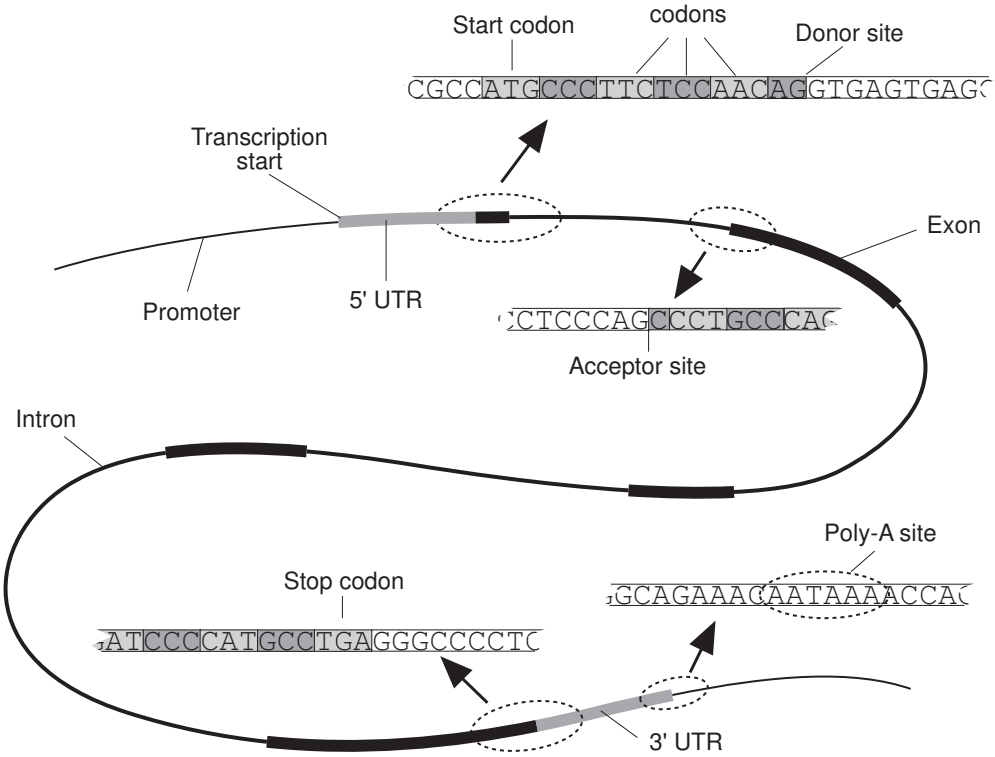


Figure 6.8: The structure of a gene with some of the important signals shown.

```
C T C C C T G T G T C C A C A G G C T
T A T T G T T T T C T T A C A G G G C
G T T C C T T T G T T T C T A G C A C
T G C C T C T C T T T T C A A G G G T
T C C T A T A T G T T T G A C A G G G T
T T C T G T T C C G A T G C A G G G C
T T G G G T T T C T T T G C A G A A C
C A C T T T G C T C C C A C A G C G T
C C C A T G T G A C C T G C A G G G T A
T A T T T A T T T A A C A T A G G G C
A T G T G C A T C C C C C C A G G A G
T T T T C C T T T T C T A C A G A A T
T C G T G T G T C T C C C C A G C C C
T T C C A T G T C C T G A C A G G G T G
A C G A C A T T T T C C A C A G G A G
G T G C C T C T C C C T C C A G A T T
```

Figure 6.9: Examples of human acceptor sites (the splice site 5' to the exon). Except in rare cases, the intron ends with AG, which has been highlighted. Included in these sequences are 16 bases upstream of the splice site and 3 bases downstream into the exon.

6.6.1 Signal sensors

One may apply HMMs directly to many of the signals in a gene structure in much the same way as one could build a position specific scoring matrix (weight matrix). In Figure 6.9 an alignment is shown of some sequences around acceptor sites from human DNA. It has 19 columns and an HMM with 19 states (no insert or delete states) can be made directly from it. Since the alignment is gap-less, the HMM is equivalent to a weight matrix.

There is one problem: in DNA there are fairly strong dinuclotide preferences. A model like the one described treats the nucleotides as independent, so dinucleotide preferences can not be captured. This is easily fixed by having 16 probability parameters in each state instead of 4. In column two we first count all occurrences of the four nucleotides given that there is an A in the first column

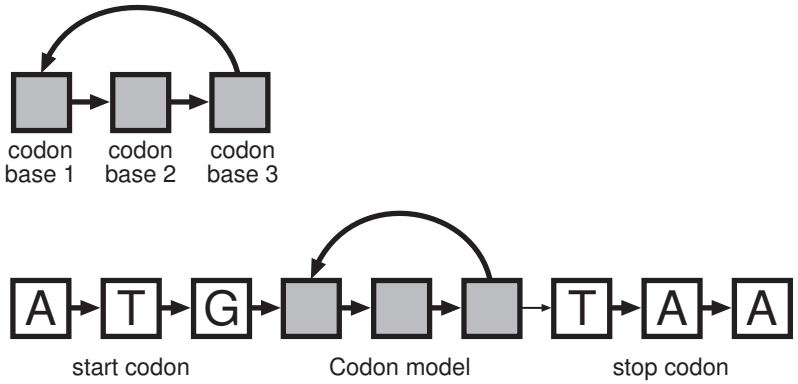


Figure 6.10: Top: A model of coding regions, where state one, two and three match the first, second and third codon positions respectively. A coding region of any length can match this model, because of the transition from state three back to state one. Bottom: a simple model for unspliced genes with the first three states matching a start codon, the next three of the form shown to the left, and the last three states matching a stop codon (only one of the three possible stop codons are shown).

and normalize these four counts, so they become probabilities. This is the *conditional probability* that a certain nucleotide appears in position two, given that the previous one was A. The same is done for all the instances of C in column 1 and similarly for G and T. This gives a total of 16 probabilities to be used in state two of the HMM. Similarly for all the other states. To calculate the probability of a sequence, say ACTGTC..., we just multiply the conditional probabilities

$$P(ACTGTC\ldots) = p_1(A) \times p_2(C|A) \times p_3(T|C) \times p_4(G|T) \times p_5(T|G) \times p_6(C|T) \times \ldots$$

Here p_1 is the probability of the four nucleotides in state 1, $p_2(x|y)$ is the conditional probability in state 2 of nucleotide x given that the previous nucleotide was y, and so forth.

A state with conditional probabilities is called a first order state, because it captures the first order correlations between neighboring nucleotides. It is easy to expand to higher order. A second order state has probabilities conditioned on the two previous nucleotides in the sequence, i.e., probabilities of the form $p(x|y,z)$. We will return to such higher order states below.

Small HMMs like this are constructed in exactly the same way for other signals: donor splice sites, the regions around the start codons, and the regions around the stop codons.

6.6.2 Coding regions

The codon structure is the most important feature of coding regions. Bases in triplets can be modeled with three states as shown in Figure 6.10. The figure also shows how this model of coding regions can be used in a simple model of an unspliced gene that starts with a start codon (ATG), then consists of some number of codons, and ends with a stop codon.

Since a codon is three bases long, the last state of the codon model must be at least of order two to correctly capture the codon statistics. The 64 probabilities in such a state are estimated by counting the number of each codon in a set of known coding regions. These numbers are then normalized properly. For example the probabilities derived from the counts of CAA, CAC, CAG, and CAT are

$$p(\mathrm{A}|\mathrm{CA}) \quad = \quad c(\mathrm{CAA})/[c(\mathrm{CAA}) + c(\mathrm{CAC}) + c(\mathrm{CAG}) + c(\mathrm{CAT})]$$

$$
\begin{aligned}
p(\mathrm{C}|\mathrm{CA}) &= c(\mathrm{CAC})/[c(\mathrm{CAA})+c(\mathrm{CAC})+c(\mathrm{CAG})+c(\mathrm{CAT})] \\
p(\mathrm{G}|\mathrm{CA}) &= c(\mathrm{CAG})/[c(\mathrm{CAA})+c(\mathrm{CAC})+c(\mathrm{CAG})+c(\mathrm{CAT})] \\
p(\mathrm{T}|\mathrm{CA}) &= c(\mathrm{CAT})/[c(\mathrm{CAA})+c(\mathrm{CAC})+c(\mathrm{CAG})+c(\mathrm{CAT})]
\end{aligned}
$$

where $c(xyz)$ is the count of codon xyz.

One of the characteristics of coding regions is the lack of stop codons. That is automatically taken care of, because $p(A|TA)$, $p(G|TA)$ and $p(A|TG)$, corresponding to the three stop codons TAA, TAG and TGA, will automatically become zero.

For modeling codon statistics it is natural to use an ordinary (zeroth order) state as the first state of the codon model and a first order state for the second. However, there are actually also dependencies between neighboring codons, and therefore one may want even higher order states. In my own gene finder, I use three fourth order states, which is inspired by GeneMark [20], in which such models were first introduced. Technically speaking, such a model is called an inhomogeneous Markov chain, which can be viewed as a sub-class of HMMs.

6.6.3 Combining the models

To be able to discover genes, we need to combine the models in a way that satisfies the grammar of genes. I restrict myself to coding regions, i.e. the 5' and 3' untranslated regions of the genes are not modeled and also promoters are disregarded.

First, let us see how to do it for unspliced genes. If we ignore genes that are very closely spaced or overlaps, a model could look like Figure 6.11. It consists of a state for intergenic regions (of order at least 1), a model for the region around the start codon, the model for the coding region, and a model for the region around the stop codon. The model for the start codon region is made just like the acceptor model described earlier. It models eight bases upstream of the start codon,[1] the ATG start codon itself, and the first codon after the start. Similarly for the stop codon region. The whole model is one big HMM, although it was put together from small independent HMMs.

Having such a model, how can we predict genes in a sequence of anonymous DNA? Any path of a DNA sequence through the model can be interpreted as a gene prediction – when the path goes through the ATG states, a start codon is predicted, when it goes through the codon states a codon is predicted, and so on. For each such path we can calculate the probability by simply multiplying the corresponding emission and transition probabilities (or adding the log-odds values), and thus we can find the *best* path, i.e., the one with the largest probability. Although there are an enormous number of possible paths through the model, it can be done efficiently by a dynamic programming algorithm, which is called the Viterbi algorithm. It is like an alignment of the DNA sequence to the HMM, and it resembles very much the pairwise alignment problem, where two sequences are aligned so that they are most similar, and indeed the Viterbi algorithm is very similar to pairwise sequence alignment..

That is quite simple: use the Viterbi algorithm to find the most probable path through the model. When this path goes through the ATG states, a start codon is predicted, when it goes through the codon states a codon is predicted, and so on.

This model might not always predict correct genes, but at least it will only predict sensible genes that obey the grammatical rules. A gene will always start with a start codon and end with a stop

[1] A similar model could be used for prokaryotic genes. In that case, however, one should model the Shine-Dalgarno sequence, which is often more than 8 bases upstream from the start. Also, one would probably need to allow for other start codons than ATG that are used in the organism studied (in some eukaryotes other start codons can also be used).

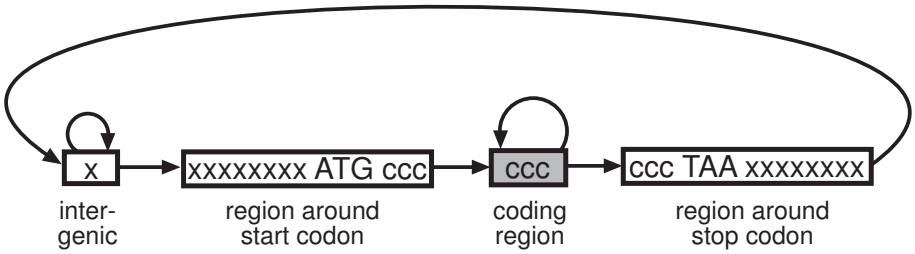


Figure 6.11: A hidden Markov model for unspliced genes. In this drawing an 'x' means a state for non-coding DNA, and a 'c' a state for coding DNA. Only one of the three possible stop codons are shown in the model of the region around the stop codon.

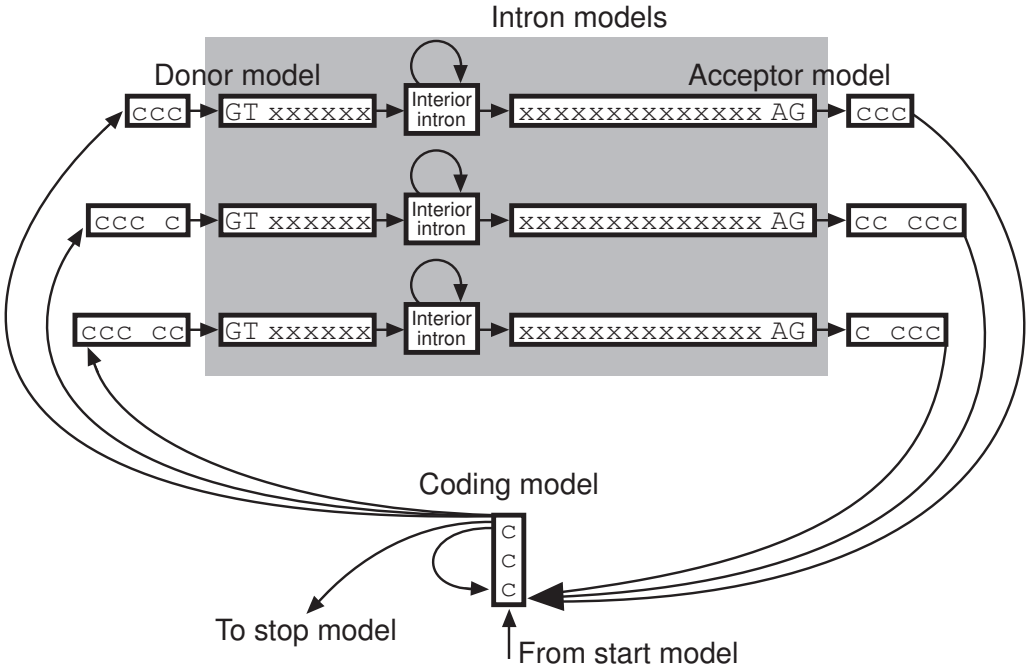


Figure 6.12: To allow for splicing in three different frames three intron models are needed. To get the frame correct 'spacer states' are added before and after the intron models.

codon, the length will always be divisible by 3, and it will never contain stop codons in the reading frame, which are the minimum requirements for unspliced gene candidates.

Making a model that conforms to the rules of splicing is a bit more difficult than it might seem at first. That is because splicing can happen in three different reading frames, and the reading frame in one exon has to fit the one in the next. It turns out that by using three different models of introns, one for each frame, this is possible. In Figure 6.12 it is shown how these models are added to the model of coding regions.

The top line in the model is for introns appearing between two codons. It has three states (labeled ccc) before the intron starts to match the last codon of the exon. The first two states of the intron model match GT, which is the consensus sequence at donor sites (it is occasionally another sequence, but such cases are ignored here). The next six states matches the six bases immediately after GT. The states just described model the donor site, and the probabilities are found as it was described earlier for acceptor sites. Then follows a single state to model the interior of the intron. I actually use the same probability parameters in this state as in the state modeling intergenic regions. Now follows the acceptor model, which includes three states to match the first codon of the

next exon.

The next line in the model is for introns appearing after the first base in a codon. The difference from the first is that there is one more state for a coding base before the intron and two more states after the intron. This ensures that the frames fit in two neighboring exons. Similarly in the third line from the top there are two extra coding states before the intron and one after, so that it can match introns appearing after the second base in a codon.

There are obviously many possible variations of the model. One can add more states to the signal sensors, include models of promoter elements and untranslated regions of the gene, and so forth.

The gene finder sketched above is called HMMgene. There are many details omitted, such as special methods for estimation and prediction described in [21]. GENSCAN is another early gene finder which is based on an HMM [22]. The main differences between the GENSCAN state structure and that of HMMgene is that GENSCAN models the sequence in both directions simultaneously. This was done to avoid prediction of overlapping genes on the two strands, which at the time were considered very rare in the human genome. Now more and more examples of genes inside introns are accumulating. The sensors in GENSCAN are very similar to those used in HMMgene. For instance the coding sensor is a 5th order inhomogeneous Markov chain. The signal sensors are essentially position dependent weight matrices, and thus are also very similar to those of HMMgene, but there are more advanced features in the splice site models. GENSCAN also model promoters and the 5' and 3' UTRs.

Another important feature of GENSCAN is that it uses a generalized HMM that explicitly model the length distribution of exons.

6.7 Comparative methods

Functional parts of genomes are conserved in evolution. Now that so many genome sequences are available, comparative methods are used more and more for dentification of functional regions in genomes, including protein coding genes. We will not go into this in detail, but just mention some of the important developments.

In order to get help to find the genes in a genome by comparing to another genome, the two genomes must be quite similar. Otherwise not enough details are conserved that it is actually useful to use conservation between the two. They should obviously not be too similar either, because then they might be so similar that everything is conserved, and conservation will not help in identifying functional regions.

Assume we have two genomes with an adequate evolutionary distance. Then the next problem is how to find the conserved regions and use it for gene finding. One way would be to first align the genomes and then use the alignment in e.g. an HMM-like model to find the genes. There are several problems with this approach, because the alignment programs are usually not aware of gene structure, so the coding regions might not always be correctly aligned, splice sites might be shifted, and so on. Therefore some gene finders both align and predict genes at the same time.

Seveal programs such as TWINSCAN [23] and SLAM [24] uses a so-called pair HMM [1, chap. 4], which models two DNA sequences at the same time. Thus, a pair HMM can align and predict genes at the same time and therefore need not rely on another alignment. Clearly, this process is rather computational intensive and is not realistic to use for more than two sequences.

Methods for previously aligned sequences are not limited to two sequences, and several have been developped that use a multiple alignment. By replacing the standard emission probabilities of an HMM state by a probability of a column of a multiple alignment according to a phylogenetic model, one can take into account the full evolutionary history of the genomes [25, 26].

6.8 Concluding remarks and further reading

When locating genes in eukaryotic genomes there is often additional evidence available for some genes, such as genomic matches to known proteins from other organisms or EST or cDNA sequences. Such evidence is obviously very valuable and should always be considered when looking for genes. Several of the gene finding programs mentioned above and below are capable of including such information.

One complication if gene finding is that a protein coding gene is not a static thing. Most genes have several different forms, because of alternative splicing and alternative transcription start sites. Therefore, to just predict the most likely gene (using the Viterbi algorithm for instance) is not necessary the best strategy. There are ways to predict suboptimal transcripts, which in principle could predict alternative splice forms. However, the models used are probably too simple to expect a one-to-one correspondence between the most probable gene structures and the most abundant splice forms, because alternative splicing seems to often be highly regulated and tissue specific.

Methods for automated gene finding go back a long time, see [27] for a review of early methods. The first HMM based gene finder is probably EcoParse developed for E. coli [28]. The problem of wrongly annotated genes in prokaryotes is discussed in [29]. The GeneMark programs are described in [30]. The strategy for extraction of training sets using database similarity was first introduced in [31]. EasyGene is described in [32].

Some of the early HMM-based gene finders for eukaryotes apart from GENSCAN and HMMgene are VEIL [33], Genie [34, 35, 36], which combines neural networks into an HMM-like model. More recent ones include Augustus [37], Genezilla [38], GenomeScan [39], GlimmerHMM [40], Agene [41] and SNAP [42]. A recent comparison of gene finding programs can be found in [43].

Chapter 7

Motif Discovery

7.1 Introduction

Let's say that 50 genes have been observed to be significantly up-regulated when a cell-culture is subjected to UV light. It is quite likely that there is some transcription factor involved – some signalling cascade results in this transcription factor being activated. Transcription factors may have a quite distinct binding site, often located in the core promoter, which is the region around the start of the gene on the chromosome. You can search the DNA sequence for binding sites of known transcription factors, which may be found in some specialized databases. Maybe, however, the binding site is unknown. In this case you'll have to look for putative binding sites that occur in the promoter sequences of the up-regulated genes. This is called pattern or motif discovery and is a classic discipline in bioinformatics.

Often in bioinformatics, we are faced with genes or proteins that we know have something in common as in the example above. There may be binding sites, structural domains, sorting signals and so on that are shared among proteins, which are otherwise unrelated. In mRNAs there are conserved binding sites for other RNAs (e.g. miRNA) and RNA binding proteins. DNA contains many signals apart from transcription factor binding sites, such as motifs relating to splicing and chromatin structure. If a set of sequences share an unknown motif, we can use motif discovery to identify it.

Figure 7.1: The binding site for the NF-kappa-B transcription factor. A few sequences are shown (left) along with the logo (right) made from a total of 38 sequences. Data from the JASPAR database [44], identifier MA0061.

7.1.1 Motifs

Binding sites and other sequence patterns are often called *motifs*. Motifs are sequence elements that are approximately conserved – short fragments that align well. Often motifs are represented with weight matrices and graphically displayed with logos, both described in the chapter about weight matrices. Figure 7.1 shows an example of a binding site for a transcription factor.

7.1.2 Motif discovery

In motif discovery, the task is to find over-represented motifs in a set of sequences. Any given set of biological sequences would normally contain an enormous amount of different signals and other features, but in the mathematical formulation of motif discovery, we have to simplify the problem. It is therefore assumed that one or a few motifs exists in a background of more or less random characters. There may be additional assumptions depending on the biological questions and the type of sequences.

Crudely speaking, there are two different approaches to the problem:

Word-based methods. These methods start from identification of over-represented words (short sequences of fixed length). The advantage of these methods is that you can easily do an exhaustive word analysis that identifies all over-represented words according to your favorite statistics. The disadvantages are that the focus on words without mismatches, can miss variable motifs and that some sort of clustering of the words into motifs is required afterwards to make sense of the results.

Weight matrix based methods. In these methods a statistical description of the motif is sought directly. Most of these methods are based on sampling or iterative optimization techniques, which are not exact, and which may return a new result in each run. The advantage is that you get a weight matrix for the motif directly, which scores well by your statistical measure. The disadvantages are that they are inexact and often has much longer run time than doing word statistics.

There are many variants including some that mix these two approaches. In this chapter, I will focus on principles instead of trying to give an exhaustive review of the literature.

7.2 Word-based methods

The idea of the word-based methods is to count words in the dataset and compare the counts either to a background set of sequences or a background model. So there are two tasks: that of counting the words, and that of the statistical evaluation of whether or not certain words are over-represented.

When talking about words in a sequence, we mean *overlapping* words. In `AAGCTAGC` these words of length 3 occur: `AAG`, `AGC`, `GCT`, `CTA`, `TAG`, `AGC`. In DNA we may or may not count words on both strands, depending on the application.

It is normally straight-forward to count all words of a given size n. If the size of the sequence alphabet is A there are a total of A^n possible words of size n. Thus for small n, one can easily make a program with an index of all possible words, initialize at 0 for all words, and grind through the sequences and add one for each word observed. For a large alphabet (such as the alphabet of 20 amino acids) and/or for long words, the number of possible words will soon become too big for

indexing, in which case one can use a hash. As the number of words is upper bounded by the total number of letters in the sequences (L), this bounds the size of the hash, and this strategy becomes interesting when $A^n > L$. However, for very large data sets even a hash will become slow, but in most applications of motif discovery the word counting part is a minor issue.

7.2.1 Word Statistics using a Markov Background

There are two types of word statistics: comparing word counts to a background model or comparing to word counts in a different sequence set. We will first look at the background model approach.

To assess the over- or under-representation of a word in a set of sequences, we can ask whether the word appears significantly more (or less) than expected in a random sequence of the same composition. More precisely: if we observe a certain word k times in the sequences, we want to calculate the probability of this happening if the sequences had been generated by the background model. The "background" sequences of the same composition would normally be modelled as sequences of independent, identically distributed letters or as a Markov chain of some order. The first model can be considered a special case of a Markov chain (of "order 0").

So, we want to calculate the probability that the word $x_1 \dots x_n$ appears a certain number of times in a sequence that has been randomly generated by an mth order Markov model with $m < n$. For simplicity, let us assume that we are studying a single sequence. A set of several sequences can be concatenated into one, and if the sequences are long, the end result will be essentially the same. Let us define $f(a_1 \dots a_m)$ as the frequency of the word $a_1 \dots a_m$ in the sequence. We can then estimate an mth order Markov model with transition probabilities $P_m(a_{m+1}|a_1 \dots a_m) = f(a_1 \dots a_m, a_{m+1}) / \sum_c f(a_1 \dots a_m, c)$. Then the probability of the word $x_1 \dots x_m$ would be

$$P(x_1 \dots x_n) = f(x_1 \dots x_m) \prod_{i=m+1}^{n} P_m(x_i|x_{i-1} \dots x_{i-m}).$$

Assuming that the word occurrences can be considered independent, the word counts can be well approximated a Poisson distribution. The probability of observing k words is

$$P(k) = \frac{\lambda^k}{k!} e^{-\lambda} \quad \text{with} \quad \lambda = pl,$$

where l is sequence length and p is the word probability from above. The rate λ is the expected number of occurrences of the word in the sequence. See Box 4 for more details.

From the Poisson probabilities you can calculate a p-value for a word occurring k times by summing the Poisson probabilities for k or more occurrences. That is, if you are looking for over-represented words – for under-represented, you should sum probabilities for k occurrences or less. This can easily be done for all words of a given size and the extreme ones can be identified as those having small p-values.

The Poisson approximation is often very good for random sequences (not always for real biological ones). In Figure 7.2A the actual distribution is compared to the Poisson approximation for an arbitrary word (`ACGTA`) in a test using 1000 different DNA sequences each of length 10000 nucleotides. The sequences were randomly generated from a first order Markov model, which was estimated from human promoters. The most problematic approximation is the assumption that the words are independent. If we make a similar plot for a "periodic" word, that is a word that can overlap itself like `CCCCCC`, it generally looks bad, see e.g. Figure 7.2B. It is because for a periodic

Probability of word occurrences

If we assume that words are independent, the probability of k occurrences of a word is given by a binomial distribution. This can be seen in this way: In a sequence of length l, k occurrences of the word can be placed in $l-n+1$ different positions. At each position there is a certain probability p that the word occur. The binomial distribution looks like this:

$$P(k) = \left(\begin{array}{c} l-n+1 \\ k \end{array} \right) p^k (1-p)^{l-n+1-k}. \tag{7.1}$$

Typically a word will occur very infrequently, so the probability is low. In DNA a word of length $n = 5$, for example, will have a probability of the order of $4^{-5} \simeq 0.001$. Additionally, biological sequences tend to be very long, and these two criteria means that the binomial distribution is very well approximated by a Poisson distribution.

This can be shown in the following way (making the further approximation of using l instead of $l-n+1$).

$$P(k) = \left(\begin{array}{c} l \\ k \end{array} \right) p^k (1-p)^{l-k} = \frac{l!}{k!(l-k)!} \left(\frac{p}{1-p} \right)^k (1-\lambda/l)^l,$$

where we defined $\lambda = pl$. The first term, the binomial coefficient, is well approximated by $l^k/k!$ when k is much smaller than l. The second term is essentially equal to p^k when p is very small. It is a classic result in calculus that $(1+x/n)^n$ converges to e^x when n goes to infinity for fixed x. Assuming that p is small and l very large, it means that the last term can be very well approximated by $e^{-\lambda}$. All together we end up with the Poisson distribution

$$P(k) \simeq \frac{l^k}{k!} p^k e^{-\lambda} = \frac{\lambda^k}{k!} e^{-\lambda}$$

Box 4: Probability of word occurrences.

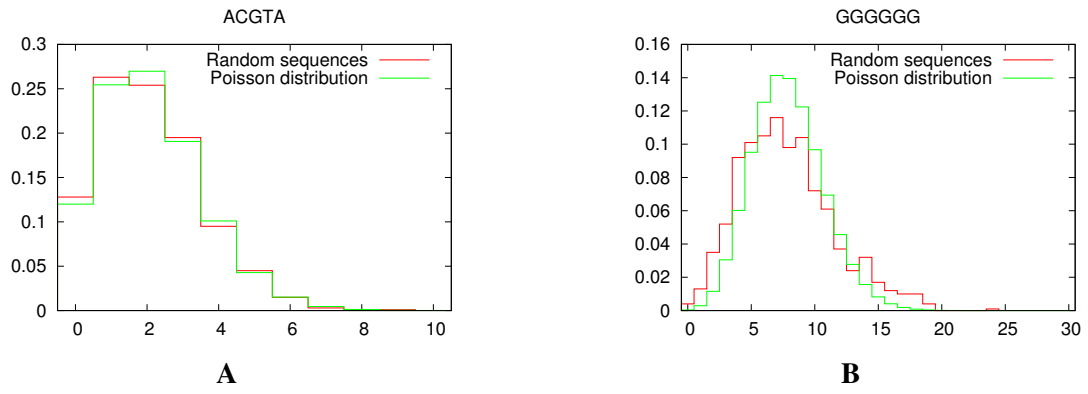


Figure 7.2: The number of words were counted in each of 1000 random DNA sequences of length 10000 and shown as a histogram along with the Poisson approximation of the distribution. See details in the text. A. For the word ACGTA we see a very good fit. B. For the periodic word GGGGGG the fit is less good.

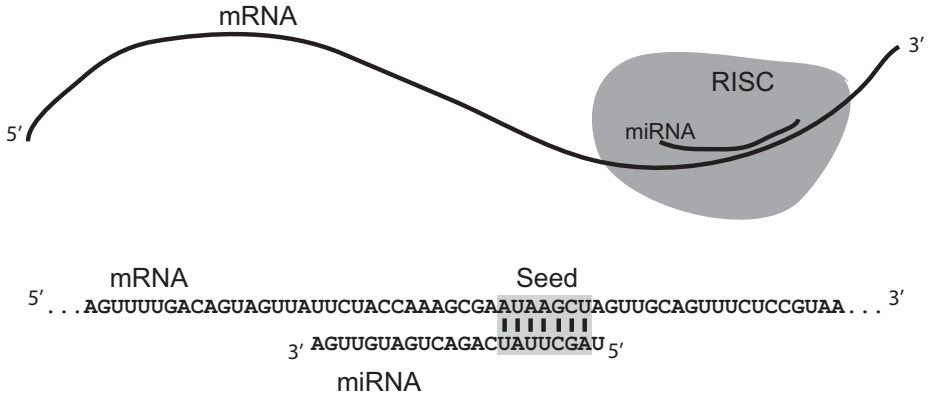
word, the probability that a word occurs is larger when it has already occurred, i.e., they are not independent events.

It is possible to work out some corrections to the Poisson model taking dependencies into account,

Gene regulation by MicroRNAs in animals

A microRNA (miRNA) is a small RNA molecule, around 22 bases long, which regulates gene expression by inhibiting the translation of mRNAs. A miRNA is incorporated into the RNA induced silencing complex (RISC) and mediates silencing through partial base-paring with the target mRNA. Most often the target site is complementary to position 2-7 or 2-8 of the miRNA. The complementary region in the target is called a seed site.

This cartoon shows it. Below the cartoon an example is shown with the human miRNA 21 (hsa-mir-21) base-pairing with a seed site in a sequence (this is not a real target sequence – just an illustration).



Normally the miRNA target sites are situated in the 3' untranslated part of the mRNA (the 3'UTR).

There is more to miRNA regulation than seed pairing. Sometimes a target site has no seed, but base-pairs with other parts of the miRNA, and sometimes the seed is incomplete and this is made up for by other base-pairings. Also other factors play important roles, such as base composition around the target site as well as RNA structure in the target and possibly other proteins binding to the target.

It turns out that mRNAs targeted by miRNAs are also gradually degraded. This effect is exploited when finding target genes experimentally. A miRNA can be introduced (transfected) into cells or it can be knocked down by a introducing complementary RNA. By comparing the gene expression in cells with and without the miRNA transfection or knock-down, one can identify genes that are targeted by the miRNA. In a transfection experiment, target genes will be down-regulated, and in knock-down experiments they will be up-regulated.

Box 5: Gene regulation by MicroRNAs in animals.

but it becomes too complicated to fit into this course. The French statistician Sophie Schbath and colleagues have studied these issues in detail and a search for her papers would be a good start if you want to learn more.

Apart from the mathematical intricacies, a weakness of this approach is that biological sequences often do not behave like Markov chains. Nonetheless this type of statistics has proven powerful in several applications.

Human miRNA-21 seed site

In an experiment a human miRNA (hsa-mir-21) was knocked down in a breast cancer cell line and gene expression was measured by DNA arrays [45] (See Box 5 to learn a bit about miRNAs). We would expect that the up-regulated genes have seed sites in their 3'UTRs, because the knock-down of the miRNA will remove the down-regulation of the target genes.

For this example, we analyze the UTRs of the most up-regulated genes (those with a log2 fold-change above 0.2). There are 123 such genes. For all the $4^6 = 4096$ words of size 6, we calculated the over-representation p-value using the Poisson model above. Additionally, the enrichment relative to the expected was calculated. The expected number is equal to λ above. We used a 2nd order Markov model as background.

It turns out that there is a very large number of significant words according to this analysis. Almost 1200 words have a p-value less than 0.01 and around 250 words have a p-value less than 10^{-10}. In this particular calculation, p-values less than 10^{-18} are reported as zero. When sorting those with a p-value of 0 according to their enrichment, the top-10 words are

Word	Number of occ.	Enrich-ment	p-value
GAAACC	73	3.40	0
CAGCCU	102	3.20	0
GGAGGC	83	3.18	0
CAUGGU	74	3.17	0
UUGGUU	123	3.13	0
UAUAUA	197	3.10	0
GAGGAA	85	3.05	0
UAAGCU	115	2.88	0
UUAUUU	370	2.87	0
UAUUUA	260	2.83	0

In the figure of Box 5 the mature sequence of mir-21 is shown, and one can see that the 7-mer seed site is AUAAGCU. We are lucky in this case that a 6-mer seed site for mir-21 (UAAGCU) is among the top scoring words.

7.2.2 Word Statistics with a background set of sequences

In the example above, we saw that the result is not totally clear. A major reason for this is that the human genome is not very well modeled by a Markov chain, because it is very inhomogeneous.

The other approach is to compare the occurrence of a word in the set of sequences to the occurrence in some other set of sequences (the background). The background set may for instance be a complete genome or the collection of all proteins. One could estimate the probability of a word from the number of occurrences in the background set instead of from the Markov chain, and then continue exactly as above. An alternative method is to do a proper statistical test and calculate a p-value for the null hypothesis: There is no difference between the foreground and background sets. Let's again assume that the background is one long sequence of length L and the word under study occurs K times in the background sequence. Then for the test we should consider the contingency table

	Word	Not word	Total
Foreground set	k	$l-k$	l
Background set	K	$L-K$	L
Total	$k+K$	$l+L-k-K$	$l+L$

For this the one-tailed Fisher's exact test is the appropriate one. It is based on fairly simple combinatorics: you ask how likely it is to select k words in the foreground set and K in the background set out of the total number of ways you can select $k+K$ words out of $L+l$. This is of course

$$\binom{l}{k}\binom{L}{K}\Big/\binom{L+l}{K+k},$$

which is known as the hypergeometric distribution. You get the final p-value by summing these probabilities for all cases more extreme than the observed one, i.e., the cases with k or more words in the foreground set if you are testing for over-representation. This in principle is straightforward, but the numerical implementation is not totally trivial. This is a standard test that can be performed in a statistics program, such as R. It is also called a hypergeometric test.

Human miRNA-21 seed site

In [45], from which we got data for the previous example, the Fisher test was used to analyze the 3'UTR sequences. Here a set of down-regulated genes were used as background. If we make the same table as before with the data from the paper, sorting the very significant words (p-value less than 10^{-18}) on enrichment, we get these top-10 words:

Word	Number of occ.	Expected number	Enrich-ment	p-value
UAAGCU	313	86.7	3.61	0
AUAAGC	240	66.9	3.59	0
AGUUAA	337	121.1	2.78	0
UAAAUU	646	318.0	2.03	0
AAAUUU	859	444.5	1.93	0
AUUUUA	1041	552.9	1.88	0
AUGUUU	734	401.1	1.83	0
AAUUUU	1021	567.3	1.80	0
AAAAAA	2388	1479.8	1.61	0
UUAAAA	1281	802.2	1.60	0

The expected number of words were calculated by scaling the number of occurrences in the down-regulated genes to a size identical to the up-regulated genes (that is Kl/L, where K is the number of words in the background, l is the total number of words in the up-regulated, and L is the total number of words in the background).

Note that two variations of the seed site AUAAGCU are at the very top, and thus this analysis is easier to interpret than the previous Poisson-based one. Note also that the sequence set used in the paper actually differ a bit from that used in the previous example, so although the conclusions hold, the numbers are not directly comparable.

7.2.3 Correlating words with a continuous score

Often we are working with experimental data, such as gene expression measurements. These are for instance fold-changes between two conditions, and we would like to find motifs associated

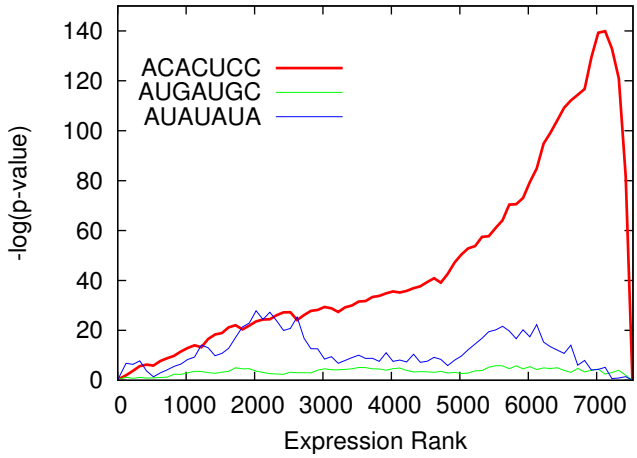


Figure 7.3: Word statistics in data from a micro-RNA transfection experiment. Human miRNA 122 was transfected in Hela cell culture and expression was compared to a control [46]. Fisher-test p-values were calculated as described in the text and their logs are plotted against the gene expression rank. The 7-mer seed site for mir-122, ACACUCC, is shown together with two arbitrary words.

with up- or down-regulated genes. One can choose a cut-off value that defines up-regulated or down-regulated genes and then use one of the methods described above. This cut-off, however, is arbitrary and it would seem more appropriate to find motifs that correlate with the fold-change or whatever measure available.

One approach to this is to calculate the Fisher p-value for any possible cut-off (or at least a large number of cut-offs), where genes above the cut-off are used as the foreground set and those below the cut-off are used as background. For each word, you can plot the p-value (or its negative log).

This is illustrated in the example of Figure 7.3, where human miRNA 122 was transfected in human Hela cells. In this experiment, you expect to see a down-regulation of targets of the miRNA. Indeed, we do see a very characteristic curve for the miRNA 122 seed site. Not only, do we see how it differs from other words (of which just a couple are shown), but the shape of the curve, and in particular its maximum, tells us something about which genes are most strongly affected.

This is the basis for a method called Sylamer [47], which has a web server you can try. Note however, that when we use the p-values in this way, we cannot say that the p-value for a word corresponds to the maximum value. When selecting the maximum p-value, it becomes biassed and can not be considered a p-value any more.

There are other methods similar to Sylamer, such as our own cWords, which actually combine the Poisson model with a rank correlation to identify significant words using a different statistic [48].

7.2.4 Summary of methods

What we have been through so far can be summarized in a table like this

Background	Statistics
Background model, e.g., Markov model	Poisson
Set of negative sequences	Fisher's exact test
Sorted list of sequences	Measure of over-representation for any cut-off

7.2.5 From Words to Motifs

One of the weaknesses of word-counting methods is that variable motifs show up as many different words. Therefore one often feels like drowning in interesting words. One solution is to cluster significant words as is done by cWords.

Another approach is taken by Weeder [49]. Weeder explicitly searches for words with mismatches. This is done by searching a *suffix tree* for over-represented words with up to a certain number of mismatches.

Another weakness of many of these methods is that the word size have to be decided in advance. Of course the statistics can be done for several lengths, but again this just adds to the already very large number of words that tend to come out of these methods.

7.3 Weight matrix based methods

Weight-matrix based methods use a probabilistic model of a sequence like a hidden Markov model. In the simplest model we assume that there is a single motif of length n occurring once in each sequence and all the remaining letters are random and independent with probability $q(b)$ for letter b. We further assume that the motif consists of independent letters with probability $p_i(b)$ for letter b at position i in the motif. These probabilities we also used to construct weight matrices, see Box 6. Our aim is to estimate these probabilities and the location of the pattern in each sequence.

If we know the position of the pattern, we can write down the total probability of a sequence $x = x_1, \ldots, x_L$ as the product of the motif and background probabilities,

$$P(x|W,a) = q(x_1)\ldots q(x_{a-1})p_1(x_a)p_2(x_{a+1})\ldots p_n(x_{a+n-1})q(x_{a+n})\ldots q(x_L), \tag{7.2}$$

where a denotes the position of the first letter of the motif and W denotes the weight matrix, or more precisely, the parameters (the pattern and background probabilities). The pattern position a is also called the alignment.

We now assume we have a set of sequences, in which the same motif occur exactly once in each. For sequence k the position of the first letter of the motif is called a_k. The probability of the whole set of sequences is assumed to be the product of the probabilities of the individual sequences (7.2),

$$P(x^1,\ldots,x^N|W,a_1,\ldots,a_N) = \prod_k P(x^k|W,a_k). \tag{7.3}$$

The question now is: How do we use this to estimate things like the pattern positions (the a's) and the weight matrix? There are essentially two ways to go from here:

Maximum likelihood. The weight matrix and motif positions are estimated such that the total likelihood (7.3) is maximized. This is similar to HMM estimation followed by Viterbi decoding. To maximize the likelihood, one can use expectation maximization, just as with HMMs.

The Bayesian approach. In Bayesian statistics the probability distribution over the parameters is sought. In this case, the probability of every motif position in a sequence should be estimated, as well as the probability distribution over weight matrix parameters. These probabilities can not be calculated exactly, but may be sampled in computer simulations as described below.

Many different methods have been developed, which often mix the two approaches. Because of the inherent fuzziness of the problem, with multiple weak binding sites, some sequences missing sites, etc., the Bayesian approach has much to offer. On the other hand, it is also computationally challenging and the presentation of results is more complex. Below we will go through the now classical Gibbs sampling approach [50], which use (sort of) maximum likelihood for the matrix parameters to simplify matters, but estimates probabilities for the positions.

7.3.1 Gibbs Sampling

In Bayesian statistics it is the distribution over the unknown parameters that are of interest. These probabilities can be calculated from

$$P(A,W|X) = \frac{P(X|A,W)P(A,W)}{P(X)}, \tag{7.6}$$

where Bayes' theorem is used (here X denotes all sequences and A the pattern positions/alignment). $P(A,W)$ is the prior probability over the pattern positions and parameters.

If this probability could be obtained, it would allow us to calculate many interesting quantities. For instance, by integrating over all weight matrix parameters, we would obtain a probability for all possible motif positions (the marginal distribution $P(A|X)$). Similarly, we could sum over all the positions to obtain a posterior distribution over the weight matrix parameters. The latter may be a bit hard to deal with, but one could find the matrix that maximizes this distribution – this is called the maximum a-posteriori estimate (MAP) and is the Bayesian replacement for maximum likelihood.

The probabilities of the Bayesian formulation cannot be calculated analytically, but one can use use computer simulations to obtain samples from the distributions (this is called sampling). Using a large number of samples, one can estimate averages, marginal distributions, etc. Gibbs sampling is a general method for sampling from a distribution over several random variables. In Gibbs sampling this is done by choosing one of the variables and sample that with the other variables fixed. The full Bayesian analysis and sampling become rather technical and a bit beyond the scope of this chapter, please see [51] for one example of a fully Bayesian approach.

One simplification, introduced in [50], is to combine Gibbs sampling and maximum likelihood estimation (or more precisely a maximum a-posteriori (MAP) estimate). Instead of sampling from $P(W|A,X)$, we choose the W that maximizes this probability. For given A this is easy: p_i is the frequency of letters in position i of the motifs and q is the frequency of all the letters outside the motifs. Usually, one would use a pseudo-count regularizer $\alpha(b)$ as well, so

$$p_i(b) = \frac{c_i(b)+\alpha(b)}{\sum_{b'}[c_i(b')+\alpha(b')]} \quad \text{and} \quad q(b) = \frac{C(b)+\alpha(b)}{\sum_{b'}[C(b')+\alpha(b')]} \tag{7.7}$$

Probabilities and weight matrices

We have met the multinomial probabilities, $q(b)$ and $p_i(b)$, before, when dealing with weight matrices. As you remember, a weight matrix was defined as

$$W(i,b) = \log(p_i(b)/q(b)).$$

There is a nice way of rewriting the sequence probability (7.2) using the weight matrix score. If $Q(x) = q(x_1)q(x_2)\dots q(x_L)$ we may write it as

$$P(x|W,a) = Q(x)\frac{p_1(x_a)\dots p_n(x_{a+n-1})}{q(x_a)\dots q(x_{a+n-1})}$$

The fraction is the relative probabilities, from which we obtain the weight matrix score by just taking the log. We call the weight matrix score for the pattern $S_w(x,a)$,then

$$S_w(x,a) = \log\frac{p_1(x_a)\dots p_n(x_{a+n-1})}{q(x_a)\dots q(x_{a+n-1})} = \sum_{i=1}^{n}\log\frac{p_i(x_{a+i-1})}{q(x_{a+i-1})} = \sum_{i=1}^{n} W(i,x_{a+i-1}).$$

Now the sequence probability can be written as

$$P(x|W,a) = Q(x)\exp(S_w(x,a)). \tag{7.4}$$

This can be used to calculated the probability of the pattern location a for a given weight matrix by "inverting" the probability using Bayes' theorem. The probability that the pattern occur at position a in the sequence is

$$P(a|W,x) = \frac{P(x|W,a)P(a)}{\sum_i P(x|W,i)P(i)} = \frac{\exp(S_w(x,a))}{\sum_i \exp(S_w(x,i))}. \tag{7.5}$$

$P(a)$ is the prior probability of the alignment. In the last equation, we assumed that a priori there is the same probability to have the pattern anywhere in the sequence – then $P(a)$ divides out. We are left with a very simple equation for the probabilities: for each position i in the sequence, we calculate the weight matrix score and take the exponential. Then the probability is this number divided by the sum over all the positions.

Box 6: Relation with weight matrices.

where $c_i(b)$ and $C(b)$ are the count of letter b at position i in the motif and outside motifs, respectively.

Now the Gibbs sampling goes like this: Initiate by picking random motif positions a_k for all sequences N. Then iterate these two steps:

1. Choose a random sequence (index r) and calculate the parameters of the background distribution and the weight matrix from the *remaining* $N-1$ sequences using equation (7.7).
2. Sample a new position a_r from $P(a_r|x^r,W)$ – it only depends on the sequence and weight matrix and not on the other sequences once W is estimated.

To use Gibbs sampling, we just have to calculate the conditional distribution of a_r in the second step. This is calculated from $P(x|a,W)$ (equation (7.2) or (7.4)) using Bayes' theorem

$$P(a_r|x^r,W) = \frac{P(x^r|a_r,W)P(a_r)}{\sum_a P(x^r|a,W)P(a)}. \tag{7.8}$$

If we assume that all positions in a sequence are equally likely a priori ($P(a)$ is uniform) these probabilities are easily calculated from the weight matrix scores as shown in Box 6 (or directly from the pattern probabilities).

Gibbs sampling example: Ribosome binding sites

To find the ribosome binding site (RBS), Gibbs sampling was done on around 1200 upstream regions of genes from E. coli. From each of the genes a sequence of 50 bases before the start codon was extracted. Gibbs sampling was done with a pattern size of 5 and a pseudo-count of 1.

In each round of Gibbs sampling, the alignment probabilities $P(a|x,W)$ were saved for one sequence, see Figure 7.4. Here one "round" means that all sequences have been updated ($\approx$ 1200 iterations per round). It is clear that the probabilities are almost uniform to begin with, where the alignments were initialized at random, and then becomes more and more peaked.

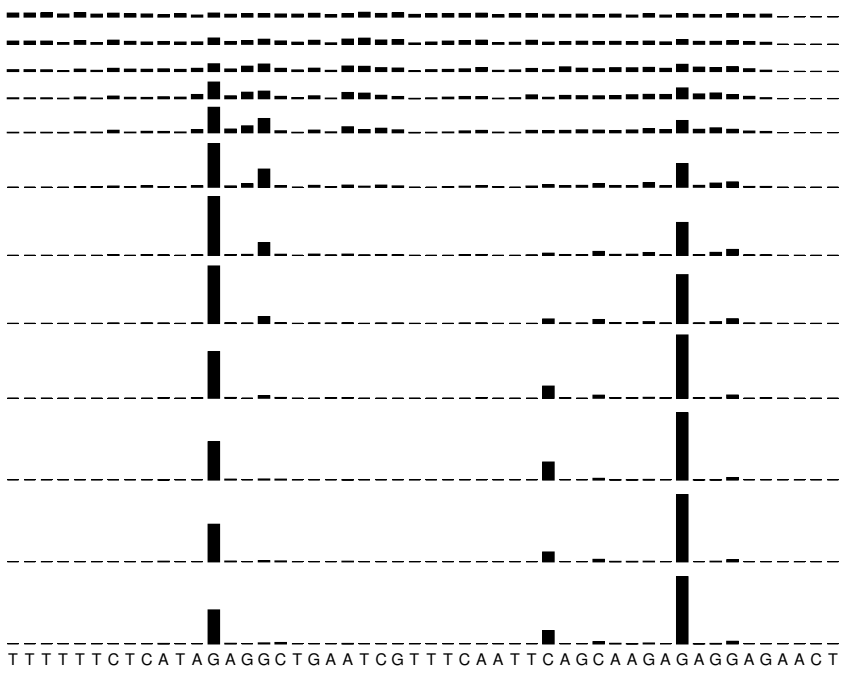


Figure 7.4: Alignment probabilities for a selected(but arbitrary) sequence in round 1, 3, 5,..., 23 are shown starting with an almost uniform distribution in the top. In the bottom one can see that an RBS-like sequence (AGGA) has high probability.

Logos of the sampled sites were made in each round as well, some of which are shown in Figure 7.5. A nice strong pattern is found already after around 20 rounds, but it is still being refined in subsequent rounds. The first glimpse of the "AGGA" RBS pattern is seen already in round 17.

The sampling continued for 1000 rounds, and in Figure 7.6 a few examples of alignment probabilities are shown calculated from an average of weight matrices. The average was calculated as a running exponentially decaying average over the last 500 rounds.

In this example, we actually needed to run Gibbs sampling 3 times to to find the AGGA pattern of the ribosome binding site, which is an indication of the fact that motif discovery is not always easy.

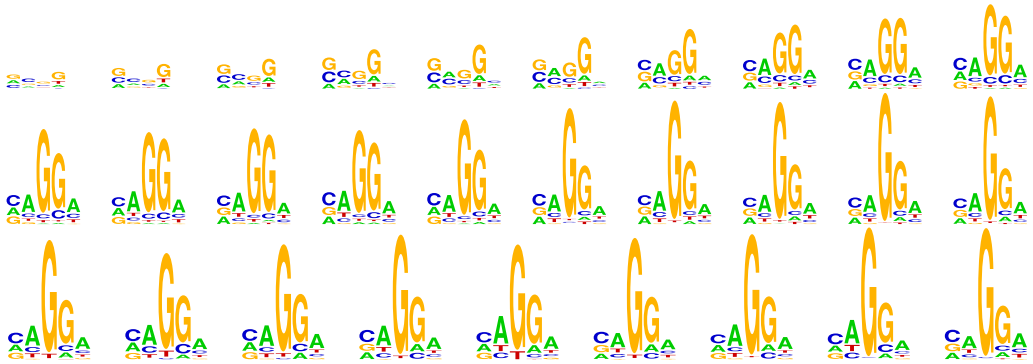


Figure 7.5: Logos of the sampled sites are shown for round 10 through 29 (the two top rows) and for every 10 round from 30 to 120 (last row). In the last panel you can observe the fluctuations that continue throughout the sampling. Logos are made with the weblogo program [52].

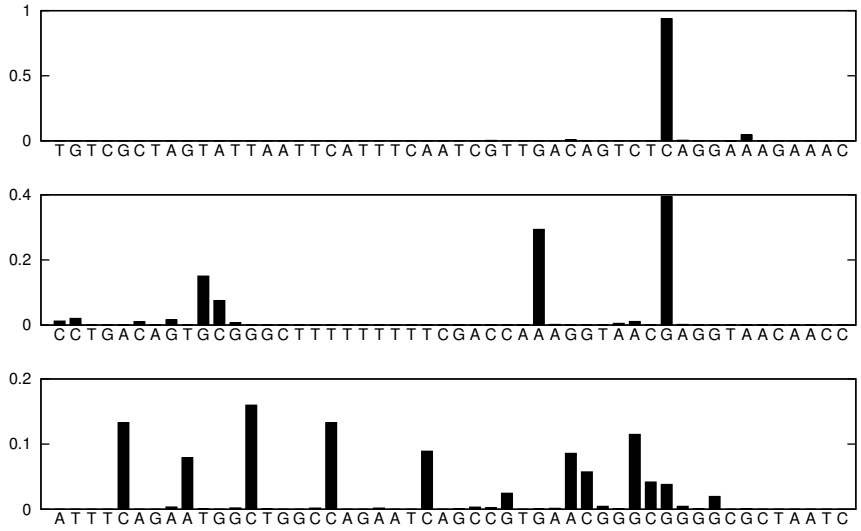


Figure 7.6: An average weight matrix was found and used to plot the pattern probability for three different sequences. In the top panel there is a clear signal around the RBS site as expected, in the second also a fairly clear signal, whereas the last is an example where there are no high-probability sites.

7.3.2 Extensions of Gibbs sampling

One of the difficulties in Gibbs sampling is that the space of weight matrices is very large, and it is not possible to sample it close to exhaustively. The theoretical guarantee that Gibbs sampling samples from the correct distribution only holds for infinitely long sampling times. In practice, the sampling will be restricted to one or a few peaks in the probability distribution.

One of the most common problems is that only a partial pattern is found. If for instance the correct pattern is AGGACT, it is quite likely that the sampling will find e.g. ACTXXX (where XXX just means 3 non-conserved bases), because in the beginning an essentially random position is chosen. Once this pattern is found, it is very difficult to escape, because in each new iteration, positions starting with ACT will have high probability, and it will be much less likely to choose a site starting with AGG. This particular problem is easy to correct, and it is done in most of the Gibbs sampling implementations: occasionally (e.g. every 50 rounds) it is checked, whether a shift of the pattern

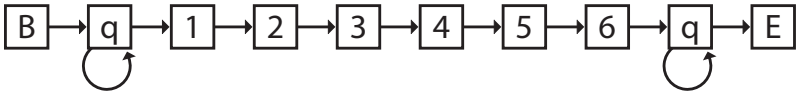


Figure 7.7: An HMM for pattern discovery. This model has begin and end states (B and E) and two background states (labeled q) which should have tied emission and transition probabilities. The states labeled 1 to 6 model a pattern of length 6.

will give a better one (higher probabilities). With the ACTXXX pattern, a shift in all the a's of -3 is likely to give a stronger pattern. This is illustrated here:

```
TTAAGGACTTACTGACACGGGCGAAGGGGA    TTAAGGACTTACTGACACGGGCGAAGGGGA
TGACAGTGCGGGCTTGTACAAGGACTATGC    TGACAGTGCGGGCTTGTACAAGGACTATGC
TATCGTAGTAGGACTGCAAAAATAAGGATA    TATCGTAGTAGGACTGCAAAAATAAGGATA
ACTTCAAGGACTCGTTATTAACTCCGCAAC    ACTTCAAGGACTCGTTATTAACTCCGCAAC
```

where initially the pattern to the left is found (ACTTAC, ACTATG, ACTGCA, ACTTTT) with no conservation in the last three bases, and after moving the pattern by -3, the correct pattern is found as illustrated to the right.

The width of the pattern can actually be adjusted in an analogous way, so one can find a good pattern width dynamically. This can be done by first expanding/shrinking the pattern to the right until it maximizes some criterion (such as information content per base) and then do it to the left. (This will make the above shifting of the pattern redundant.)

Another limitation of the simple Gibbs sampling presented above is the assumption that all sequences contains one instance of the pattern. To find more patterns one can run it multiple times. When a pattern is found, it is masked in the sequences (often by replacing the nucleotides in the pattern with Ns) and the program is run again. However, it is also possible to explicitly include multiple patterns in the model and to allow for the same pattern occurring multiple times in a sequence or not at all. Each implementation of motif discovery has its own way of expanding the scope.

7.3.3 Maximum Likelihood Methods

The model used above in equation (7.2) is identical to a hidden Markov model with a state for each position in the pattern, see Figure 7.7 apart from the transition probabilities in the two background states. If, however these transition probabilities are tied, they will be of no consequence. It is also required that the background states have tied emission probabilities. Such a model can be estimated from the sequences using the Baum-Welch algorithm and thereby a pattern can in principle be found. This is a maximum likelihood approach to the problem.

After estimating the model, we can use the Viterbi algorithm to find the the pattern positions. A path though this model is completely specified by the alignment of state 1 to the sequence, which corresponds to the alignment a in Gibbs sampling. (We could also calculate the posterior probability of being in state one for each position in the sequence. This would give us a probability of having the pattern in each position of the sequence.)

One drawback of this approach is that the estimation procedure often gets stuck in a local maximum of the likelihood. It is even less robust than the Gibbs sampling. For it to be useful, one would have to build in some heuristic to get out of the local maxima. Another drawback is the

whole idea of just selecting one final solution, which is often too simple (although for the user it is convenient).

The great advantage is that it is a very general and modular framework that is easy to expand to several patterns, allowing for many patterns in a sequence (having a looped model with just one q state) and so on. Also, if you already know something about the pattern (e.g. that it is likely to have GG at position 3 and 4), it is very easy to prime the model with this information before training it.

It is worth adding that there in principle is no problem in using sampling with HMMs, but this has not been heavily pursued in motif discovery.

MEME

One of the most used methods for pattern discovery is called MEME (Multiple EM for Motif Elicitation). It is similar to a hidden Markov model in that it uses Expectation Maximization (EM) as in Baum-Welch, but with several heuristic methods specialized to identification of motifs.

The most important heuristic is to seed the EM by sequences occurring in the data set [53]. For each word of a certain size a matrix is constructed by setting the letter probability for the letters in the word to some constant (say 0.9) and divide the rest of the probability among the remaining letters of the alphabet. Then a single step of EM is done and the best motifs are selected for further optimization. In HMM language, a step of EM consists of estimating the expected counts for each letter given a matrix, just like one step of the Baum-Welch procedure. Seeding the matrix with all words in the data set ensures that a large and *relevant* part of the weight matrix space is visited, and thereby increases the chance that a good maximum of the likelihood is found. The original MEME also allowed for more than one motif per sequence.

Later versions of MEME has added more heuristics, such as support for variable width. In fact there is a whole suite of programs derived from MEME [54], each of which are specialized for particular needs.

7.4 Concluding remarks

The literature on motif discovery is extensive, and many methods are available of which only a few are mentioned here with the aim of presenting the principles that are common to most of the methods.

In practice motif discovery is often very difficult. One of the main reasons is that biological sequences usually encodes many types of information at the same time. On top of that, the experimental results, from which you usually select the sequences, are usually noisy and contain a mixture of different effects. Examples:

- In promoter sequences you often have CpG islands, which means that the pattern discovery sometimes just find random CG rich “motifs”.
- If a miRNA is transfected, it may cause the down-regulation of a transcription factor (which is a direct target for the miRNA) and thus all the genes regulated by that transciption factor may be down-regulated. These genes will not be direct targets of the miRNA, and will not contain seed sites in their 3’UTR. This will be confusing for the pattern discovery.

After reading this chapter, you might think that pattern discovery is only relevant for DNA and RNA. This is certainly not the case, it has been applied for proteins as well, and many tools work for both proteins and nucleic acids. However, most of the applications are probably for the latter.

Bibliography

[1] R. M. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, Cambridge, UK, 1998.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[4] W. R. Pearson. Effective protein sequence comparison. *Methods Enzymol.*, 266:227–258, 1996.

[5] W. J. Kent. BLAT - the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[6] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[7] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1991.

[8] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the Second Workshop on Algorithms in Bioinformatics*, Lecture Notes in Computer Science 2452, page 449463. Springer-Verlag, 2002.

[9] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[10] G. Pavesi, G. Mauri, and G. Pesole. An algorithm for finding signals of unknown length in DNA sequences. *Bioinformatics*, 17 Suppl 1:S207–S214, 2001.

[11] S.M. Kielbasa, R. Wan, K. Sato, P. Horton, and M.C. Frith. Adaptive seeds tame genomic sequence comparison. *Genome Res*, 21:487–493, Mar 2011.

[12] T. T. Marstrand, J. Frellsen, I. Moltke, M. Thiim, E. Valen, D. Retelska, and A. Krogh. Asap: a framework for over-representation statistics for transcription factor binding sites. *Plos ONE*, 3(2):e1623, 2008.

[13] M Beckstette, D. Strothmann, R Homann, R. Giegerich, and Kurtz S. PoSSuMsearch: Fast and sensitive matching of position specic scoring matrices using enhanced sufx arrays. In *Proceedings of the German Conference on Bioinformatics*, 2004.

[14] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, Mar 2008.

[15] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.*, 18(11):1851–1858, Nov 2008.

[16] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, 10(3):R25, 2009.

[17] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. Watson. *Molecular Biology of the Cell*. Garland Publishing, Inc., 3rd edition, 1994.

[18] M. Burset and R. Guigo. Evaluation of gene structure prediction programs. *Genomics*, 34(3):353–367, 1996.

[19] T. D. Schneider and R. M. Stephens. Sequence logos: a new way to display consensus sequences. *Nucleic Acids Research*, 18(20):6097–6100, 1990.

[20] M. Borodovsky and J. McIninch. GENMARK: Parallel gene recognition for both DNA strands. *Computers and Chemistry*, 17(2):123–133, 1993.

[21] A. Krogh. Two methods for improving performance of a HMM and their application for gene finding. In T. Gaasterland, P. Karp, K. Karplus, C. Ouzounis, C. Sander, and A. Valencia, editors, *Proceedings of the Fifth International Conference on Intelligent Systems for Molecular Biology*, pages 179–186, Menlo Park, CA, 1997. AAAI Press.

[22] C. Burge and S. Karlin. Prediction of complete gene structures in human genomic DNA. *Journal of Molecular Biology*, 268(1):78–94, 1997.

[23] I. Korf, P. Flicek, D. Duan, and M. R. Brent. Integrating genomic homology into gene structure prediction. *Bioinformatics*, 17 Suppl 1, 2001.

[24] M. Alexandersson, S. Cawley, and L. Pachter. SLAM: cross-species gene finding and alignment with a generalized pair hidden Markov model. *Genome Res.*, 13(3), 2003.

[25] J. S. Pedersen and J. Hein. Gene finding with a hidden Markov model of genome structure and evolution. *Bioinformatics*, 19(2), 2003.

[26] A. Siepel and D. Haussler. Combining phylogenetic and hidden Markov models in biosequence analysis. *J. Comput. Biol.*, 11(2-3), 2004.

[27] J. W. Fickett. Finding genes by computer: the state of the art. *Trends in Genetics*, 12(8):316–320, 1996.

[28] A. Krogh, I. S. Mian, and D. Haussler. A hidden Markov model that finds genes in *E. coli* DNA. *Nucleic Acids Research*, 22:4768–4778, 1994.

[29] M. Skovgaard, L. J. Jensen, S. Brunak, D. Ussery, and A. Krogh. On the total number of genes and their length distribution in complete microbial genomes. *Trends in Genetics*, 17(8):425–428, August 2001.

[30] J. Besemer, A. Lomsadze, and M. Borodovsky. GeneMarkS: a self-training method for prediction of gene starts in microbial genomes. implications for finding sequence motifs in regulatory regions. *Nucleic Acids Research*, 29(12):2607–2618, 2001.

[31] D. Frishman, A. Mironov, H.-W. Mewes, and M. Gelfand. Combining diverse evidence for gene recognition in completely sequenced bacterial genomes. *Nucleic Acids Research*, 26(12):2941–2947, 1998.

[32] T. S. Larsen and A. Krogh. EasyGene - a prokaryotic gene finder that ranks ORFs by statistical significance. *BMC Bioinformatics*, 4(1):21, 2003.

[33] J. Henderson, S. Salzberg, and K. H. Fasman. Finding genes in DNA with a hidden Markov model. *Journal of Computational Biology*, 4(2):127–141, 1997.

[34] D. Kulp, D. Haussler, M. G. Reese, and F. H. Eeckman. A generalized hidden Markov model for the recognition of human genes in DNA. In D.J. States, P. Agarwal, T. Gaasterland, L. Hunter, and R.F. Smith, editors, *Proc. Conf. on Intelligent Systems in Molecular Biology*, pages 134–142, Menlo Park, CA, 1996. AAAI Press.

[35] M. G. Reese, F. H. Eeckman, D. Kulp, and D. Haussler. Improved splice site detection in Genie. *Journal of Computational Biology*, 4(3):311–323, 1997.

[36] D. Kulp, D. Haussler, M. G. Reese, and F. H. Eeckman. Integrating database homology in a probabilistic gene structure model. In R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klein, editors, *Proceedings of the Pacific Symposium on Biocomputing*, pages 232–244, New York, 1997. World Scientific.

[37] M. Stanke, O. Keller, I. Gunduz, A. Hayes, S. Waack, and B. Morgenstern. AUGUSTUS: ab initio prediction of alternative transcripts. *Nucleic Acids Res.*, 34(Web Server issue), 2006.

[38] J. E. Allen, W. H. Majoros, M. Pertea, and S. L. Salzberg. JIGSAW, GeneZilla, and GlimmerHMM: puzzling out the features of human genes in the ENCODE regions. *Genome Biol.*, 7 Suppl 1, 2006.

[39] R. F. Yeh, L. P. Lim, and C. B. Burge. Computational inference of homologous gene structures in the human genome. *Genome Res.*, 11(5), 2001.

[40] W. H. Majoros, M. Pertea, and S. L. Salzberg. TigrScan and GlimmerHMM: two open source ab initio eukaryotic gene-finders. *Bioinformatics*, 20(16), 2004.

[41] K Munch and A. Krogh. Automatic generation of gene finders for eukaryotic species. *BMC Bioinformatics*, 7(1):263, 2006.

[42] I. Korf. Gene finding in novel genomes. *BMC Bioinformatics*, 5, 2004.

[43] K. Knapp and Y. P. Chen. An evaluation of contemporary hidden Markov model genefinders with a predicted exon taxonomy. *Nucleic Acids Res.*, 35(1), 2007.

[44] E. Portales-Casamar, S. Thongjuea, A. T. Kwon, D. Arenillas, X. Zhao, E. Valen, D. Yusuf, B. Lenhard, W. W. Wasserman, and A. Sandelin. JASPAR 2010: the greatly expanded open-access database of transcription factor binding profiles. *Nucleic Acids Research*, 38(Database issue):D105–D110, Jan 2010.

[45] L. B. Frankel, N. R. Christoffersen, A. Jacobsen, M. Lindow, A. Krogh, and A. H. Lund. Programmed cell death 4 (PDCD4) is an important functional target of the microRNA miR-21 in breast cancer cells. *Journal Biol. Chem.*, 283(2):1026–1033, 2008.

[46] A. Grimson, K. K. Farh, W. K. Johnston, P. Garrett-Engele, L. P. Lim, and D. P. Bartel. MicroRNA targeting specificity in mammals: determinants beyond seed pairing. *Molecular Cell*, 27(1):91–105, Jul 2007.

[47] S. van Dongen, C. Abreu-Goodger, and A. J. Enright. Detecting microRNA binding and siRNA off-target effects from expression data. *Nature Methods*, 5(12):1023–1025, Dec 2008.

[48] A. Jacobsen, J. Wen, D. S. Marks, and A. Krogh. Signatures of RNA binding proteins globally coupled to effective microRNA target sites. *Genome Research*, 20(8):1010–1019, Aug 2010.

[49] G. Pavesi, P. Mereghetti, G. Mauri, and G. Pesole. Weeder web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes. *Nucleic Acids Research*, 32(Web Server issue):W199–W203, 2004.

[50] C.E. Lawrence, S.F. Altschul, M.S. Boguski, J.S. Liu, A.F. Neuwald, and J.C. Wootton. Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *Science*, 262(5131):208–214, Oct 8 1993.

[51] M. H. Tang, A. Krogh, and O. Winther. BayesMD: flexible biological modeling for motif discovery. *Journal of Computational Biology*, 15(10):1347–1363, 2008.

[52] G. E. Crooks, G. Hon, J. M. Chandonia, and S. E. Brenner. WebLogo: a sequence logo generator. *Genome Research*, 14(6):1188–1190, Jun 2004.

[53] T. L. Bailey and C Elkan. Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine Learning Journal*, 21:51–83, 1995.

[54] T. L. Bailey, M. Boden, F. A. Buske, M. Frith, C. E. Grant, L. Clementi, J. Ren, W. W. Li, and W. S. Noble. MEME SUITE: tools for motif discovery and searching. *Nucleic Acids Research*, 37(Web Server issue):W202–W208, Jul 2009.