1 Polynomials

We have three sets of data points generated by some polynomial with added noise. The goal of this task is to find a polynomial that fits the data points well, but that doesn’t model the noise. Naturally it’s impossible to decide precisely what the signal-noise ratio is when you don’t know the exact properties of the source that generated the data, but we’re going to try to use MDL to find an equilibrium between modeling the original data and the noise.

1.1 Theoretical background

Ideally we would take the smallest possible representation of a polynomial and deviations of data points from the polynomial, but in practice this is impossible, because we simply can’t know all the ways to represent a polynomial. The important thing to MDL is that the redundancy in the representation of the model and the data is of equal magnitude, because what we’re interested in is the ratio of the description length of the model and the description length of the data.

We chose to model the polynomials as its degree, \( g \), its parameters \( p_0, \ldots, p_g \), and the difference of all the data points to the polynomial \( \delta_0, \ldots, \delta_d \), where \( n \) is the number of data points, all in binary, prefix coded. The length of the prefix encoding of an integer \( n \) is about \( \log_2(n) \), so this takes us approximately

\[
\log_2(g) + \sum_{i=0}^{g} \log_2(p_i) + \sum_{i=0}^{n} \log_2(\delta_i)
\]

bits.

The number of bits you need to represent the polynomial grows linearly with its degree, because you need to represent the parameters of the polynomial. The number of data points stays the same for all degrees of course and because you can expect higher order polynomials to fit the data better (or equally good) the representation of the \( \delta \)‘s will decrease with increasing \( g \).

1.2 Implementation details

We know in this case that all the numbers we’re dealing with, the data points, the coefficients of the polynomial, and the deviations are integers that can be represented by 6 bits plus one for the sign. We could have enumerated all possible polynomials of 7-bit integers and used a smart search algorithm to find the best polynomial of all reasonable ones, but since the precision of existing \( \chi^2 \) fitters and the roughness of the numbers both are very high we decided to use Gnuplot’s \( \chi^2 \) fitter to fit the data.

The process is roughly as follows; for each degree \( g \in 1, \ldots, n - 1 \), where \( n - 1 \) \( \leq 9 \), because Gnuplot doesn’t work well for degree 0 or > 9, we let Gnuplot determine the polynomial \( P_g \) of that degree that best fits the data. We know that the coefficients of the original polynomial were 7-bit numbers, so we do all our arithmetic on \(-63, -62, \ldots, 62, 63\) scale, in order to minimize rounding errors, so we fit the data with a polynomial of the form \( P_g(x) = \sum_{i=0}^{g} a_i x^i \) and round \( a_i \) to the nearest integer value. Then for each data point \( n = (x, y) \) we calculate \( x - P_g(x) \) and round this to the nearest integer value to obtain \( \delta_n \). Then we compute the prefix code length of the degree, the parameters and the data and add this all together to come up with a description length of our favorite \( g \)-degree polynomial. Finally we choose the polynomial with the least description length. This polynomial is supposed to be close to the original polynomial.

1.3 Using poly.pl

The programs expects input of the form given on the website\(^1\). For every degree it shows the parameters, the \( y \)-value of the polynomial at every data point, the integer distances, and the description length of the degree, the parameters and the differences. Finally it prints the minimum description length polynomial and its degree in Gnuplot format.

\(^1\)http://www.cwi.nl/~tlee/final.html
1.4 Results

**Data 1** The first set of data points correspond to pieces of an urn that’s missing its middle part. The question was whether we could reconstruct the urn’s profile from this incomplete data. The resulting polynomial we found is \((-59x^3 + 10x^2 + 60x + 1)/63\). In figure one you can see that this simple polynomial fits the data very well. So the answer is probably yes.

![Figure 1](data1g2)

**Data 2** The second set of data points has more noise added to the data than the previous set. The data set has about the same shape as the first set, so we expected the polynomial to be of the same degree as for the first data set. This is indeed the case. The polynomial we found is \((-60x^3 + 16x^2 + 61x)/63\).

![Figure 2](data2g2)

**Data 3** We think that this is the most interesting of the three data sets, because the data fits a fourth degree polynomial quite a bit better than a third degree, yet it doesn’t need the distinct capability of a fourth degree polynomial, which is to model three extremes. Figure 3 shows a second, third and fourth degree polynomial fitted to the data. You can see that the second degree polynomial lacks the capability to represent the general shape of the data. The third and fourth degree polynomials however are almost of the same shape. Our program prefers the third degree polynomial \((59x^3 - 4x^2 - 36x)/63\), shown in figure 4.

![Figure 3](data3g2)

1.5 Things we thought about

**sin x** We fed \(\sin x\) with various parameters to our program to see what happened. For a limited number of waves our program is able to represent the data points with a polynomial, like with the previous data sets, a second degree for wave length \(\frac{1}{2}\), a third degree for wave length 1, etc. As you can see in figure 5, and 6.

![Figure 5](data4g2)

\[
\frac{(34x^5 - 138x^3 + 136x)}{63}
\]
become closer to the true model. That generally means that the more data points you get the simpler the model you find becomes. (not simpler than the true model of course)

In the case of functions with a shape that can’t be grasped by a polynomial of finite degree, line the sine function, odd artefacts can appear, like the huge coefficients. In the case of the sine function adding more data points only makes matters worse, because it increases the error. Every data point that can’t be modelled by the polynomial adds a lot to the total error.

2 Markov Chains

2.1 Theoretical background

This section describes the encoding of the markov chain that was derived from the input data. The encoding consists of $K$, the order of the chain, and $\theta$, the probabilities assigned to each of the $2^K$ possible histories. First of all, $K$ is bounded by the length of the input string, as a markov chain can say nothing about the first $K$ bits of a string (it needs at least a size $K$ history). The minimum amount of bits we would like to predict in the input string is at least 1, so $K \leq n - 1$. As we may assume the length of the input is known, we can use about $\lceil \log n - 1 \rceil$ bits.

To encode $\theta$ cleverly, we have to look at how it is established from the given input. To approximate $\theta$, for each history $h \in \{0, 1\}^K$ the total number of occurrences ($t_h$) in the input string is counted, and the number of times $h$ is proceeded by a 1 ($o_h$). The approximation of $\theta_h$ now equals $o_h/t_h$, and we have to store this approximation for each $h$.

If we observe the set of $o_h/t_h$ closely, we can notice some properties that will help us encode it more efficiently. First of all, $\sum_h t_h = n - K$. This is because we do frequency counts over the input data of length $n$, but have to exclude the first $K$ bits because they don’t have sufficient history. If we know the total number of distinct solutions to $\sum_h t_h = n - K$, with $n$ and $K$ fixed, we can code the entire array of $t_h$’s as the index into the lexicographical ordering of these solutions. This happens to be a known combinatorial problem, with a known solution: the number of solutions to $\sum_{i=1}^n x_i = r$ is $C(r+n-1, n-1)$, with $C(n, r) = n!/(n-r)!r!$. For our purposes, we can even improve on this by noticing that we can calculate one of $t_h$ (for instance $t_{1^K}$) from all the other $t_h$, $n$ and $K$. So, our encoding of all $t_h$ is now the index into the number of solutions to $\sum_{h-t_{1^K}} t_h = n - K - t_{1^K}$. This has length $\lceil \log C(x+y-1, y-1) \rceil$, with $x = n - K - t_{1^K}$ and $y = 2^K - 1$. To be able to do the decoding, we only need $n$ and $K$; $n$ is known and $K$ was decoded.
from the preceding bits.

At this stage we can assume knowledge of \( n, K \) and all \( t_h \). Now the observation that \( o_h \leq t_h \) will help us in our encoding. We can use a ‘counting’ encoding with variable maxima per digit. If we for instance count using the decimal system, each digit has 10 possibilities, and the total number of possibilities for 3 digits is \( 10 \times 10 \times 10 \). We will count using variable possibilities per \( o_h \), ranging from 0 to \( t_h \). Using this technique we can uniquely encode all \( o_h \) by specifying its index in the lexicographical ordering of all \( o_h \), which has a length of \( \lceil \log \prod_h (t_h + 1) \rceil \).

Now that we have encoded the entire markov chain, we can proceed to encoding the data. By definition, the markov chain gives us a probability distribution for a proceeding bit given its history \( h \). For the first \( K \) bits of data, this is impossible as they have no history of size \( K \) yet. We can of course make all kinds of assumptions to ‘make up’ a history for these bits, but as we have no information about how this data was produced these assumptions would be unfounded. Therefore, the first \( K \) bits are just copied into the encoding, adding \( K \) bits. For the following bits, we have a probability distribution for the occurrence of a 1 at that point. As was very explicitly stated in the lectures, every probability distribution is a code and vice versa, with the (average) code length of a bit being \( -\log p \) if the bit has probability \( p \) of occurring. We can calculate this for every bit with history \( h \) of length \( K \), as we have \( P(1|h) = 1 - P(0|h) \) from the markov chain. Summing these values for every bit, we get the average code length of the data given the markov chain.

The total length of the markov chain and data now amounts to:

\[
\lceil \log n - 1 \rceil + \lceil \log C(n - K - t_h K + 2K - 2, 2K - 2) \rceil + \\
\lceil \log \prod_h (t_h + 1) \rceil + \sum_{(h,i) \in \text{data}} -\log P(i|h)
\]

The order that minimizes this formula is the order that according to MDL should have produced the data.

There exists one problem with the encoding above: it does not account for the encoding of future sequences which haven’t occurred yet. This is because their estimated probability of occurring is 0, and therefore their code length is \( \infty \). We could circumvent this problem adding one ‘virtual’ sample of each non-occurring sequence into the frequency count. This is, however, mathematically unsound, and as we may assume that the same source produces the test and training sets we chose theoretical purity above practical use.

### 2.2 Implementation details

When implementing the calculations of the length of the encoding, we stumbled upon a few computational issues. Especially time and memory bounds, as well as bounds within the representation of integers within the language were a problem. A few tricks and approximations are used to avoid hitting these bounds are used. First of all, a hash is used to store the contents of \( \theta \). Storing these in an array would exponentially increase the memory usage and computation time with increasing \( K \). Now, computations are only performed on histories that actually occurred in the sequence.

In our encoding, we also need to calculate \( \lceil \log C(n, r) \rceil \). For increasing \( n \) or \( r \), the intermediate results explode out of bounds because of the factorial calculations. However, a little clever rewriting changes this to something computable:

\[
\lceil \log C(n, r) \rceil = \\
\lceil \log(n!/((n - r)!r!)) \rceil = \\
\lceil \log(n!) - \log((n - r)!r!) \rceil = \\
\lceil \sum_{i = 1}^{n} \log i - \sum_{i = 1}^{n - r} \log i - r \rceil = \\
r' = \max(n - r, r) \rightarrow \sum_{i = r' + 1}^{n} \log i - \sum_{i = 1}^{n - r'} \log i
\]

This last sum is computable as it only produces intermediate results that are much smaller than that of the original formula.

When encoding \( o_h \) we have to calculate \( \lceil \log \prod_h (t_h + 1) \rceil \). This produces again very big intermediate results, which the computer isn’t able to handle at some point. Here we just calculate \( \sum_h \log(t_h + 1) \).

At the moment, there are 2 stopping conditions. First of all, the program doesn’t look for larger order markov chains when the current chain totally predicts all data (when the data length drops to \( K \)). Longer markov chains will not result in smaller encodings. Furthermore, it stops at \( K = 52 \), for the practical reason that it otherwise will enter an infinite loop as some loop counter gets so big that decreasing it by 1 doesn’t decrease it as this falls outside of the variable’s precision.

### 2.3 Using markov.pl

The usage is simple: the program takes 2 arguments, the first argument the training sequence file and the second argument a test sequence. (There is a hidden 3rd argument, if it is set testing will never occur.). The files are read and every 0 and 1 is processed, the rest of the characters are filtered out. It’s output is for each \( K \) the length of the encoding of \( K \), the
length of the markov chain and the length of the data and the total sum. When it is done, it will dump the best markov chain it has found. It might occur that in a test sequence some subsequence is encountered which has probability of 0 occurring in the training sequence. The markov chain cannot encode this, so an error will be displayed.

2.4 Results

As the computation stops when we reach a point where the entire datasequence is reconstructible from only the markov chain (and the first $K$ bits), each graph can be thought of as continuing to the right as a growing function.

**Sequence 0** This sequence clearly shows the dip in description length when the ‘right’ order has been found. After this dip a mountain arises, indicating that the data is such that its predictability will not increase enough to compensate for the increase in the model size.

**Sequence 5** In this sequence a lot of runs of ‘01’ occur, indicating that a second order model will probably do quite good. For orders just higher than 2, the model size explodes faster than the data size implodes. However, this increase in model size falls when $K > 10$, whereas the data size keeps on decreasing until $K > 12$. This indicates that a high order ‘structure’ could be found, which is also visible by the relatively small difference between the description length at $K = 1$ and $K = 26$, only about 80 bits.

**Sequence 7** Looking at sequence 7, it can be seen that it primarily consists of 0’s, with an occasional 1. Humans would probably classify this as a run of 0’s, with some random noise of 1’s. Markov chains and MDL agree with this: increasing the order of the model only allows for encoding ‘noise’, which is very expensive in terms of model size. MDL therefore picks the model with $K = 0$, which is just a static description of the probability of noise occurring.

**Sequence 8** The interesting thing about this sequence is that at $K = 0$ and $K = 17$ the difference in description length is just 3 bits. At $K = 0$ the markov chain only has static information about the occurrence of 1’s, at $K = 17$ the entire data is contained in the markov chain. Just philosophizing in the wild, one could say that this string consist of random substrings of length 17, which occur in a predictable order. If there aren’t too many different substrings this would explain the relatively small model size at $K = 17$, and why it is able to compete with $K = 0$. 