Project in Information Theoretic Modeling: Report

Joan Verdaguer Olivella
Antoni Segura Puimedon

Table of Contents

Introduction

This report is the documentation of the compressors and decompressors that have been elaborated by the Spartans team that, as the reader can see in the header and on the cover, is composed by Joan Verdaguer Olivella and Antoni Segura Puimedon. The approach that was taken to compress and decompress the data, along with the side information that was supplied by the course was mainly a process of study of the nature of the data and then a process of trial for mechanisms that would help to achieve a reasonable compression.

In the effort of picking mechanisms to compress the data there were several methods that were tried and developed almost to completion that, unfortunately, were later fount not to be suitable for the purpose they had been chosen, and thus, discarded with the an obvious and quite considerable time impact. Due to this casualties, and as a safeguard zlib was picked to compress that for which we could not find a good compression system or didn't have time to fully implement and or debug.

In the following chapters, that will correspond to the data that was object to compression, we will present the evaluation we did on the data and the methods, final or discarded, that we tried on them.

This document will then contain an annex of tools that we used to complement and or assist on our compression/decompression effort and will be effectively closed by a conclusion chapter in which we will try to share some thoughts that have spawned on the whole process of taking and working on the assignments, as well as a little bit of feedback.

Info.xpm

The first data that we will set into discussion is info.xpm, that as the extension of the file hints, is an X Pixmap, id est, an ASCII based image that consists on an array of characters. It can be conceptually split in two parts: definition and execution. Both parts, are integrated in a standard character pointer array definition in C.

The definition part consists itself in two parts. On the first part, the declaration of the pixel width (150) and height (100), the amount of colors (5) and the number of characters per color (1). On the second part, the “table” that assigns a color to each of the characters.

The execution part is a sequence of pointers to characters, one for each line in the picture that is to be portrayed, that is filled with the characters defined in the last stage of the definition segment.

Acknowledging the nature of this data file, that is not mated with any kind of side information, the approach that was taken to compress it was the following. We decided to account in the compression output file only for the letters that are written over the otherwise regular and extremely compressible image (or rather definition of image). Thus, any reference to the colors, sizes and the non-dot characters is moved on to the decompressor in a hard-coded way.

The compressor, thus, centers it's functionality around the basic principle of splitting the execution part into a 150x100 matrix formed of the five different characters previously introduced in the definition part and looking for the dots that are in it. To further reduce the amount of data that needs to be stored for each of the columns, the dots are saved with start and end positions, allowing for greater chunks to be represented with less data. This last specialization is possible due to the fact that the letters that are represented by the dots are rather tall and overall not very round. This allows the output to save the eventual inefficiency of having to store information for each dot of a column individually.

The chunks for each column (the chunks for a new column are preceded by a bit set to '1', otherwise set to '0'. '1' One allows to ignore the rest of the column) are then encoded to a binary stream (7 bits per position) that is written to the standard output bit per bit to the

output by the bwriter[1] module.

In the other side, the decompressor opens the file that is given to it as a parameter and reads its binary content by using the breader[2] module.

In this moment, the decompressor rebuilds the information that was previously stated to be hardcoded generating a matrix with the background characters. Then, with the data that was read, from the input stream, the chunks of dots are written inside the matrix column by column until the end of the stream.

Finally, the definition part is written to the standard output, followed by the matrix that is handled row-wise.

1 Explained in the annex.
2 Explained in the annex.

Mushroom.dat

Mushroom.dat, unlike the previous data to encode, is provided along a side information that is composed by 22 variables or parameters of varied nature. This varied nature means that there are different sets of possible values for the parameters and one might even be null (denoted by a question mark).

Seeing that the side information is articulated into parameters which every row relating to each and every one of the p(oisonous) e(dible) of mushroom.dat, it was thought that maybe it would be possible to construct a machine that would compute the function that takes the 22 parameters as input and outputs p or e accordingly.

We realized that probably the function that the machine would have to compute would probably not be linear so we settled for a three layer neural network, hoping that the number of misses or exceptions would be small enough to be handled as a posteriori exceptions to the calculus that could be stored in a tuple array for correction.

The experimentation with the neural network was conducted with the mandatory 22 input neurons, different sizes of the hidden layer, namely five, eight and ten, and one output neuron that would ideally output one when poisonous and zero when edible. The neural network model picked for the computation was the standard feed forward and weight updating by backpropagation, and different activation functions were tested with the last pick being the sigmoid function. Each row of the sdat was fed to the neural network by a conversion process to reals from zero to one of each of the possible values of the discrete number of possible values of each parameter. As a side note, the neural network had one extra node in the input layer that would have its value constantly to one to offset the eventual case where a zero input was passed to each and everyone of the other neurons.

Once the neural network was completed, the objective would have been to store only the weight matrix on the output of the compressor, that would have been only the size of the bit length of the rounding to four or five decimals of the $(22+1)*8+8$[3] plus a small number of possible exceptions for eventual prediction errors.

3 Considering eight neurons in the hidden layer.

The result of the testing process however, gave unsatisfying results with a learning (compression) time that was close to two hours, which largely offset the time requirements for merely acceptable prediction success rate. With fewer iterations the convergence to the function was simply to vague and the number of exceptions did not support the pick of the method. It was tried upon the network several learning paces and modifiers depending on the speed without successful results and thus, after a lot of work into the machine, it was dropped and the search for patterns began.

The search for patterns was conducted on a parameter and set of parameters directly reflecting on the output basis, and it was eventually found that some values on certain parameters determined the output with almost no exception. Thus, it was decided to make a decompressor that just with the sdat would be able to produce the result without any previous compression work.

The decompressor works as following:

1.- Read one line of sdat into a list of the parameters.

2.- Output p unless the current row is not one of the exception rows and the following condition in not met:(s[0]=='c'or s[1]=='g'or s[4]in['c','f','m','p','s','y']or s[8]in['b','r']or s[13]in['c','b','y']or s[14]in['c','b','y']or s[16]=='y'or s[17]=='n'or s[18]in['l','n']or s[19]=='r'). When this happens output e.

3.- Return to 1 until no rows are left.

Curve1.dat

For curve1.dat, that is a set of points that comes with an equally large set of point in curve1.sdat. The MDL principle was applied and we started working on it by plotting curve1.dat respect curve1.sdat obtaining the following graphic.

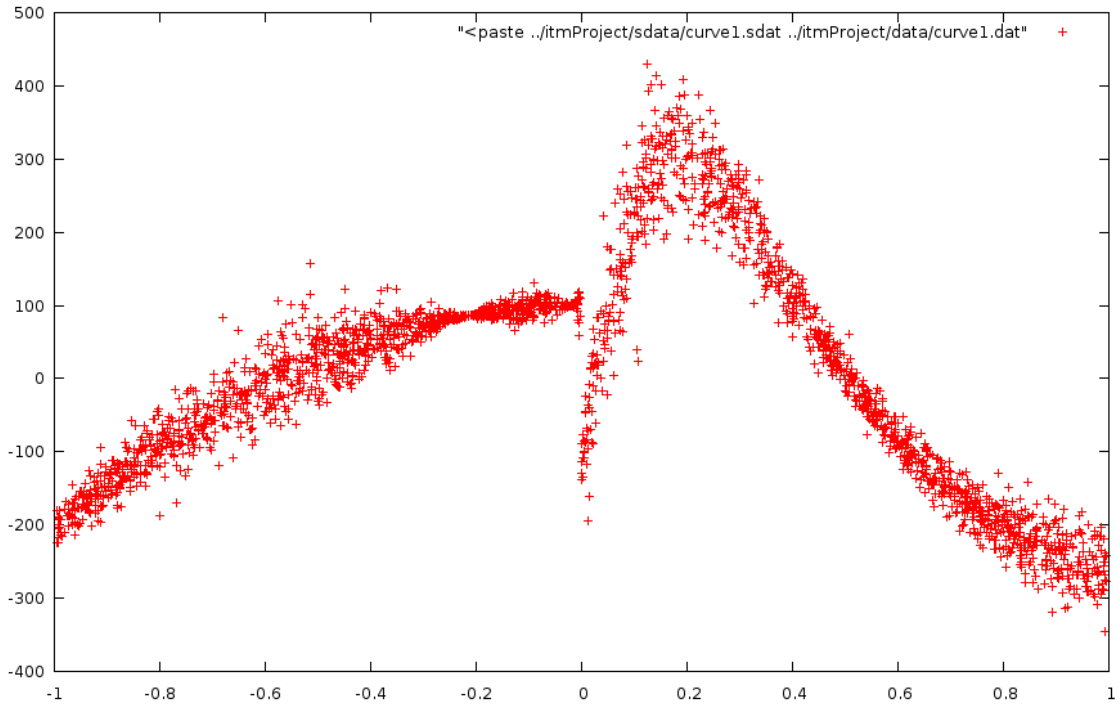

Illustration 1: Plotting of curve1.dat respect curve1.sdat.

It seemed pretty obvious to us, although maybe there is another way around that we failed to see, that the function defined by this graphic is in fact a composed function that would be f(x) when x is below zero and g(x) when x is above zero.

f(x), as depicted in illustration 2, is defined as a logarithmic function as follows: (180.28*math.log((91.31*x)+107.61)) – 723.83.

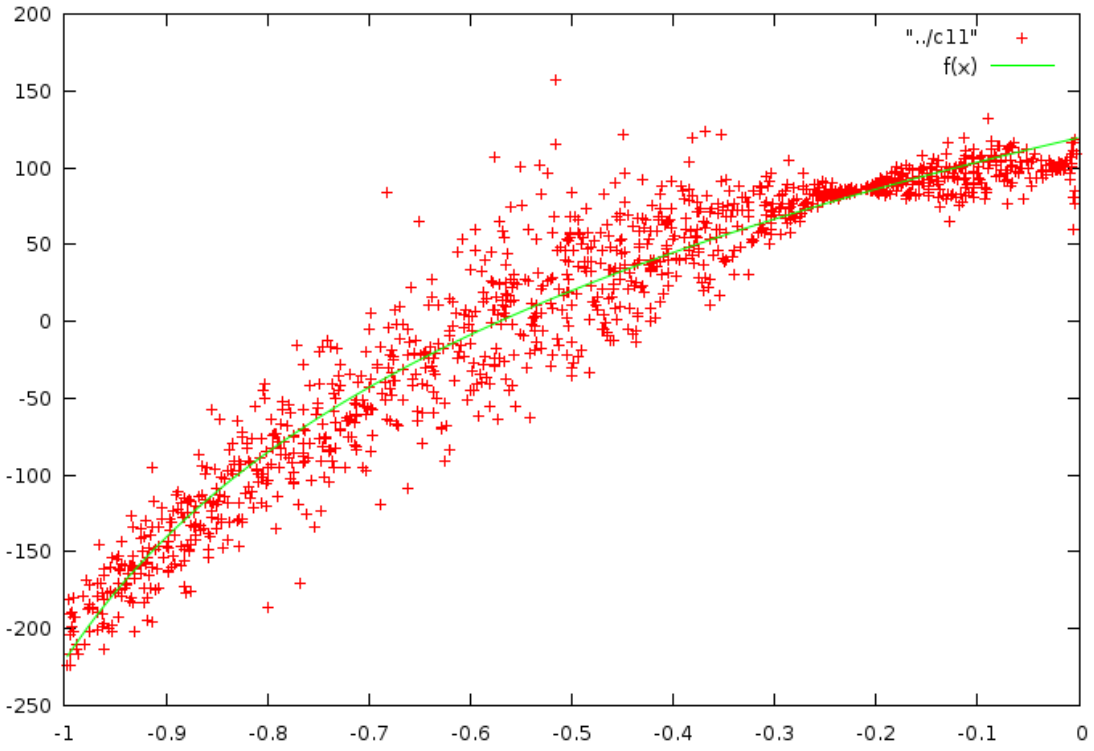

Illustration 2: Second part of the two-part function.

g(x) is, as depicted in illustration 3, defined as a polynomial function as follows: (-92.73 + 4504.47*x - 16529.19*x^2 + 19824.71*x^3 – 8002.51*x^4).

It is important to note that the floating point values that are included in those functions have been achieved by the fitting feature of gnuplot. Then, the compression is performed by storing the difference between each of the points in curve.dat to the point that would be ideally found inside the defined functions. Storing the difference instead of the points obviously requires less data, and thus, compression is achieved.

The differences, that are computed against f(x) when the point in sdat is below zero and against g(x) otherwise, are then converted to binary in an eight bit natural plus un sign bit, 9 bits per point (10 for the g(x) part). If the distance is extremely high it is encoded in

twelve bits, so one bit is added to all the distances to difference the low distance from the high distance points. Then, they are written to the standard output by the bwriter module.

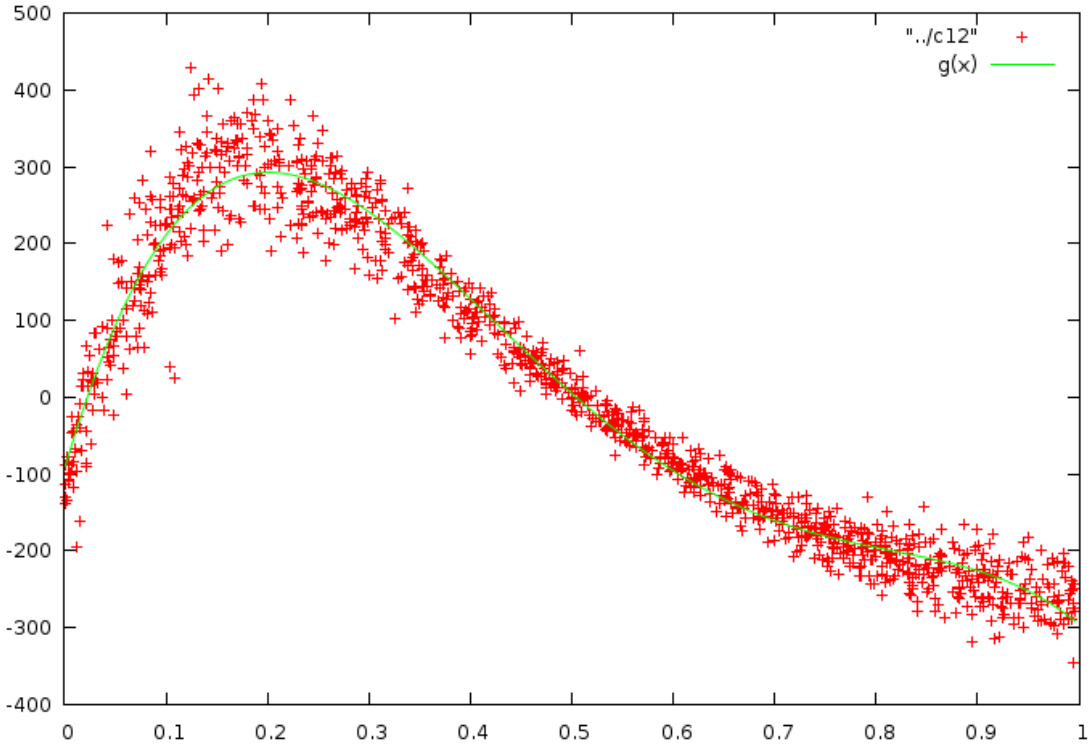

Illustration 3: Second part of the two-part function.

On the other side, the decompressor performs the symmetrical operation. It reads from the file that is given the binary data using breader, and then for each 9 bits it converts those in a floating point distance that will be added to the value resulting on computing f(x) or g(x) on the corresponding sdat entry, obtaining the original values that are written to the standard output.

Curve2.dat

Curve2.dat, along with its side information curve2.sdat, is a similar data to the one explained in the previous chapter (curve1.dat) in the sense that is a set of points that can be plotted as x(sdat), y(dat). This time though, although the same principle applies the function, that is also composed by two subfunctions is of a quite different nature and contains a rather high amount of distant noise or outliers.

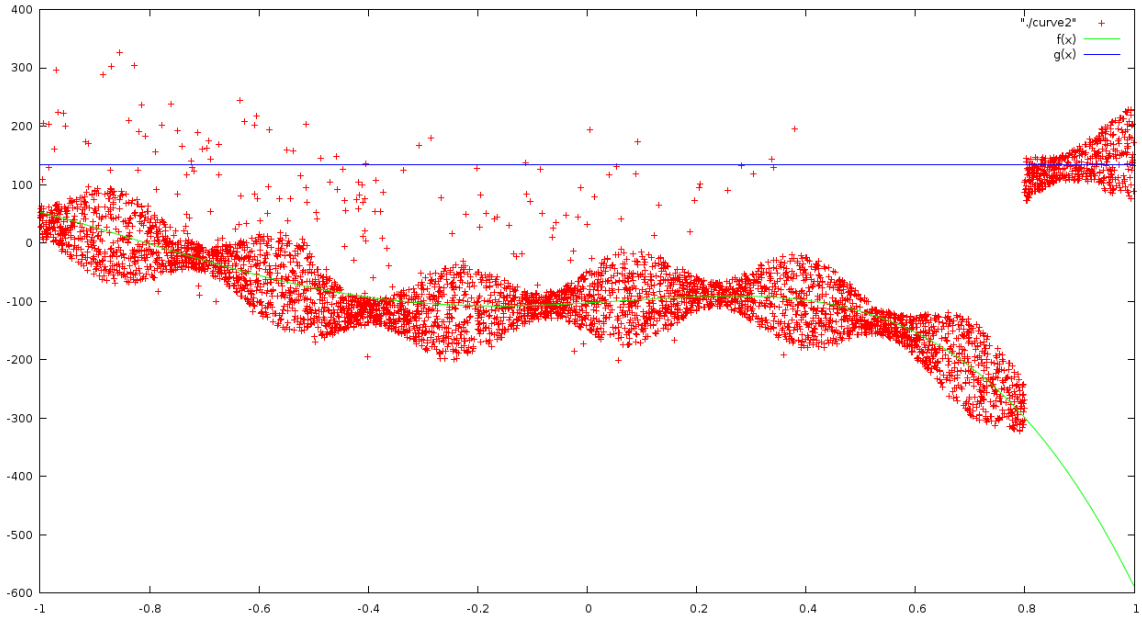

Illustration 5: Plotting of curve2.dat respect curve2.sdat.

In curve2, the subfunctions are defined as f(x) below 1 and g(x) above 1.

f(x) is a polynomial function defined as follows: -103.01 + 53.35*x + 86.01*x^2 - 374.07*x^3 – 251.26*x^4.

g(x) is a constant function defined as follows: 133.52

Then, the procedure used for compression is the same as explained in curve1 with the addition that for the high distance cases (>102.3) 14 (+1 per sign) bits encode the distance and otherwise it is 12 (+1 per sign) bits. An added bit is used to hint the

decompressor on the length of the following distance, one for long and zero for short.

After this, and as explained in the previous chapter, it is passed through bwriter and breader to the decompressor, that does the same procedure as curve1 does but reading 12 or 14 bits at a time as hinted by the extra bit per distance.

Letter.dat

This data set keeps some resemblance to the one explained in mushroom.dat but not the number of parameters in the sdat is decreased and composed of natural numbers and the output set is the uppercase letters of the alphabet. As at the point where the compression of this data was undertaken, the neural network had already been deemed quite unreliable for the purposes of compression in this settings, new models were tried that I will try to explain in the following paragraphs.

The first approach was to realize that the problem that would convert the information stored in sdat to that contained in dat was mainly a classification problem, and as a result, the k-nearest neighbor algorithm was picked as a fast and efficient method to draw inspiration from in the processof making the compressor and decompressor.

The K-Nearest Neighbor algorithm demands for a set of inputs to which all the other input is compared, so what would be needed in this case would be to convert sdat and dat content into a training set/knowledge base on which the hamming distances would be calculated. This method has essential problems in regards to compression because it requires quite a lot of data or, at least taking a quite large subset of the data to have as a comparison set (on which to calculate then the hamming distances of each of the sdat rows). As a result, it was thought of ways to reduced this unreasonable space requirements.

The first machine model that was tried was a simplification of the k-nearest neighbor that was based on the fact that for any of the rows, the most likely minimum distance row was to be another row belonging to the same letter. Taking that into consideration it was computed for each of the letters, which rows were more significative, that is, which rows where the ones that were picked as the minimum distance by the other rows in their class. Once this information was known, it was hardcoded on the machine the indexes of the best five rows for each of the letters (26*5) and, with that, the machine started an iterating

process over the rows were each row (that was not on the indexes) was pitted against the best candidates set and, if the distance to the any of the rows of a letter set was lower to that of the unclassified rows, that row was added to the letter set. This process would be repeated until only a few exception would be left, that would be encoded separately (they would constitute the other part of the output, apart from the key indexes).

The problem with the first machine, is that, even though the number of false additions was relatively ignorable, the convergence of the rows to the final clusters was too slow (several distance calculation methods and heuristics such as accepting rows with distances lower than a certain value even though there were better results outside the processed data) to meet the time requirements. It was also tried to increase the number of starting key elements, but it was seen that in the n-dimensional (where n is the number of parameters) space, the most relevant rows of each letter were likely to be together, so increasing the number among the most likely was not really helpful. An idea that we didn't have time to try was to perform some analysis of that space to pick as key indexes the rows that were more relevant to the relatively different subclusters that were in the space for each letter, as this would probably make diverse enough the comparing groups to speed up the clustering process.

The second machine that was designed, consists on applying statistics to the k-Nearest Neighbor by taking all the parameters for each row and letter and computing the mean and standard deviation for each of the parameters of a certain letter. With this information, that would constitute the first part of the two part code of the compression, each of the rows in sdat would be tried for distance and build a priority queue with the letters, and append to a list which of the elements of the priority queue was the valid one. Obviously, the valid element will likely be the first element, and the probability decreases quite fast, so applying huffman code to the list of valid elements reduces the length of the list significantly. The huffman tree was hardcoded on the decompressor and the list, converted to 11 digit binary per element, was added as the second part of the two part code.

The two parts were then written by bwriter to the standard output and recovered by

breader at the other side, where the priority queues would be rebuilt row per row with the first part of the code, and the element would be picked by removing the front element of the front of the list that constituted the second part of the code.

It is important to note that there was some study on distance functions that would maximize the concentration of valid element towards the first numbers to maximize the resulting compression and it was found that the best method was to set the distance to 70% of the hamming distance when the row fell inside the mean and set the distance to 100% of the standard deviation and add the squared distance to the closest +- range of the value.

Due to some problems of the precision that would make the process work if it was performed all in the same compressing machine (without violating the restrictions that would have to be applied in the decompressing part) but not when the encoding and decoding were in separate python executions probably due to a rather strange float precision issue, the method was not completed on time and was not submitted to avoid the penalty that would have ensued a late delivery. Zlib, thus, was the method picked for compression.

Forest.dat and rat.txt

These two sets of data were finally compressed by the zlib module that is bundled with python and that uses the DEFLATE compression algorithm. Forest.dat, resembles mushroom.dat and letter.dat but we were not able to find any direct relation (even with exceptions) like in mushroom.dat and we could not apply the machines designed for letter.dat for the same reasons they were not finally adopted in letter.dat. Thus, we started looking for more complicated patterns, but the search was unfortunately unsuccessful.

The DEFLATE algorithm is based on a list of blocks that of different nature, that is specified by adding two bits in front of each block that act as a header. An additional bit precedes the header indicating if the block is the last block (1) or there are still more blocks (2).

The first kind of block (00) in DEFLATE stream is the uncompressed, that just stores a literal with a maximum size of 64K without treating it. The other blocks are compressed blocks that can either use a preset huffman tree for symbol encoding (01) or a huffman tree that is inherent to the block content (10) and thus the tree is encoded at the begining of the block.

Additionally to the compression achieved by the symbol reduction that results from using the preset huffman block and specially with tailored huffman blocks (which are more common), a process of search for string (or byte sequence) repetition is undertaken within the stream to substitute the additional occurrences of one sequence by a mere pointer to the original place. This pointer consists on the number of bits one must travel back to find the beginning of the repeated sequence (at most 32KB) and the length of the sequence that is matched. This amount of time spent on this step has a significative impact on the compression results, and thus, zlib allows for a parameter to establish the maximum time allowed (1-9) to do this search.

Other groups data

The data submitted by other groups was analyzed for entropy, which was found to be maximum, which ruled out regular compression algorithms, and thus, a pattern search was started. Due to time constraints and a good ability by the other groups in picking quite obfuscated algorithms for generation of the data in the decompressor, we didn't find any successful compression decompression mechanisms. The final resolution adopted by the spartans was to just output the input data as not even zlib could give better results.

Conclusion

As stated in the introductory chapter of this report, here we will share some of the thoughts or what we have learned in the process of taking and working for this course, as well as try to provide some feedback for the following iterations of the course.

First of all, what we have learned is basically the traditional compression by the MDL principle and putting other common algorithms to real use. Another thing that we have learned, and probably the hard way is the convenience of settling for well known compression approaches when time is a constraint instead of generating new compression models because that can limit a lot the the results when an untried or new model turns out bad.

Second, I would like to state the convenience of taking the course of *Introduction to Machine Learning* before taking this course as the majority of data files that are provided contain enough side information to perform a good compression using the techniques described in that course. It would have been interesting perhaps to have half of the files without side information and half with (without counting the contributions by other groups).

Finally, we would have liked to try on more general and long files in which the entropy could not have been surpassed by side information.

Annex

The aim of the modules bwriter and breader is basically to encapsulate the functionalities of writing and reading from a file in a binary way. As this feature is used by various of our decompressors, we can save some space by sharing this module and just including it on the decompressors and using its method.

To write, bwriter accepts a string composed of zeroes and ones and then writes it to a file (which is also a parameter) grouping them by bytes. But, as the number of bits might not a multiple of eight, it uses the first byte of the file to write the amount of "extra" bits it will have to write on the end of the file to make it multiple of eight. This is, the first byte is the number of bits the string lacks to be multiple of eight.

To read, we simply do it the other way around, breader reads the first byte, transforms it to integer and then reads all the other bytes, transforming them to a string (formed of zeroes and ones) and skipping the lasts bits (depending on the amount indicated by the first integer we read).

In the bwriter module both functionalities are included (read and write) since the space it takes is not relevant. But in the breader module (in the decompressor part) only the functionality of reading is included.