Lab Work 3

André Pedrosa [85098], Filipe Pires [85122], João Alegria [85048]

Algorithmic Information Theory
Department of Electronics, Telecommunications and Informatics
University of Aveiro

December 31, 2019

1   Introduction

This report aims to describe the work developed for the third and final assignment of the course of 'Algorithmic Information Theory', explaining all scripts developed by us and presenting the results we considered most relevant regarding the quality of the solutions.

The programs implemented in Shell have the purpose of measuring similarity of images using a compression-based strategy and automatically identify human faces. Along with the description of the solutions, we also discuss the characteristics and the preprocessing done to the available dataset, the data compression algorithms used and the tests executed. All code developed is publicly accessible in our GitHub repository:

`https://github.com/joao-alegria/TAI.`

2 Image Classification

When dealing with image processing problems that require object identification, the usual environment calls for a classification approach, as the objects to be identified belong to specific classes with clear and separable characteristics. The challenge we are faced with (*1*) is a special approach for the image classification problem of facial recognition or identification. It is special in the sense that it does not aim to explore the usual strategies based on feature extraction, rather it is meant to rely on data compression methods and the notion of algorithmic information. Algorithmic information, also known as algorithmic entropy or Kolmogorov complexity, is the measure of information in a given data portion based on the theoretical algorithm that would be required to recreate that same data portion.

As this is an abstract definition, an example can clarify the concept: considering strings A="0101010101" and B="0010100110", for instance, one can easily reach the conclusion that string A is far easier to define than string B, since for A we can simply define it as the smaller string "01" repeated 5 times; string B, on the other hand, the process is not as simple and we probably will require the entire string to describe it. What this means is that A has less information than B. To achieve the so called algorithmic information of each one, the thought process is the same but now the string descriptions are achieved through algorithms; for our examples, the algorithms that define strings A and B are the following:

```
def stringA():                    def stringB():
    for i in range(5):                for letter in "0010100110":
        print("01", end="")               print(letter, end="")
```

By a quick analysis, it is obvious that *stringA()* is less complex than *stringB()*, iterating only 5 times, contrasting with the 10 iterations of the second; hence confirming that string B has more information than string A. This can be directly correlated with data compression, since compressors take advantage of the data repetition to reduce space usage, and thereby compress the information by storing the data that appears frequently and the indicators of the places where those repetitions occur.

With this in mind, one can proceed to understand how the image classification will be done. Using the Normalized Information Distance defined by equation 1, where *K(x)* and *K(x—y)* define the Kolmogorov complexity of the string x and of the string x given string y.

$$NID(x,y) = \frac{max\{K(x|y), K(y|x)\}}{max\{K(x), K(y)\}} \tag{1}$$

Unfortunately, because the Kolmogorov complexity is non-computable, this formula is only theoretical and we are sure there is no algorithm that can compute any string complexity. Nevertheless, compressors approximate the formula's results very closely, if used correctly. So we used two of the most frequently used approximations of `NID` to determine the compression distance between images and, with these distances, calculate the closest images, i.e. the most similar ones, and hopefully classify them as belonging to the group of images that represent the same face.

2.1 NCD Approximation

By using known general purpose compressors, such as gzip, lzma or tar, the Kolmogorov complexity can be estimated by the number of bits the resulting compressed file has. This approximation is called Normalized Compression Distance (NCD) and is expressed by equation 2, where *C(x)* and *C(x,y)* represent the number of bits the compression of x and of x and y concatenated contain, respectively. The xy concatenation can be performed in several ways, the most common being to simply perform an union on both; other more complex operations can also be done, such as interlacing the two data sources.

$$NCD(x,y) = \frac{C(x,y) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}} \tag{2}$$

Following the formula, results close to 0 indicate similarity and close to 1 dissimilarity. The only restriction to this approach is that the compressor used must follow the condition $C(x,x) \simeq C(x)$. If this is not verified, the results of NCD are not reliable.

2.2 NCCD Approximation

The second approximation is called Normalized Conditional Compression Distance, expressed by equation 3. The difference between NCD and NCCD is that the later uses conditional compressors to achieve its estimations. Conditional compressors are normally implemented with finite-context models and take in consideration the previous occurring terms to compress the new term. This is a more direct approximation of the NID and, in general, generates better results than the NCD.

$$NCCD(x,y) = \frac{max\{C(x|y), C(y|x)\}}{max\{C(x), C(y)\}} \tag{3}$$

C(x) has the same meaning as before and *C(x—y)* represents the number of bits the compression of x needs given that y occurred ($C(y|x)$ follows the same logic). Due to the conditional aspect of this approach, *C(x)* can be a special case of *C(x—y)* were y is equal to the empty string, making true the statement $C(x) = C(x|\epsilon)$.

3 Dataset

Like in all experimental work, specially in image processing, a considerably large dataset is required for any results to be significant and any conclusions to be solid. Also, when benchmarking an algorithm, it is recommended that standard test datasets are used for researchers to be able to directly compare the results. Taking this in consideration, we downloaded AT&T "The Database of Faces" (formerly "The ORL Database of Faces") and used it to test our code.

The image database contains 10 different images of each of the 40 distinct subjects. For some subjects the images were taken at different times, between April 1992 and April 1994, with varying lighting conditions and different facial expressions. Most faces are in an upright position in frontal view, with a slight left-right rotation. All 400 images were taken in black & white at the Olivetti Research Laboratory in Cambridge, UK.

Figure 1: Image taken from the assignment description (*1*). Examples from the ORL face dataset. In the first row, ten face images corresponding to the same subject (s01). In the second row, the first face image of subjects s02 to s11.

To speedup the processing of each image, we applied 2 transformations to each element of the dataset. The first consisted in a simple image reduction from 112×92 px to 56×46 px, ensuring that proportions remained the same. The second was a quantization process from 8 bits per pixel gray level to 4 bits only. These modifications were accomplished using ImageMagick (*2*), a command-line program to create, edit, compose, or convert bitmap images. The script developed to apply the transformations was `preprocessImages.sh`.

Figure 2: Example from subject s01 of the ORL face dataset, before and after preprocessing.

4 Experiment

Once the dataset was ready, we composed an experiment to determine which commonly used compressors provided the best results for our intents. The conducted study had 2 phases: the first where we tested several data compressors using NCD and compared the precisions of the face identification program using each of the compressors; the second where we tested a conditional compressor using NCCD and compared the best results of the first phase with those of this test.

4.1 Compressor Validation

Before conducting our experiment, we needed to make sure the compressor candidates were valid. This was done by verifying for each compressor C if condition $C(x, x) \simeq C(x)$ was true. The concatenation used in this verification uses the only valid operation for this situation, the "append" operation, which is explained in section 4.3.

The script checkCompressor.sh was a script developed to help us manually verifying each compressor. For each candidate, we compressed an example image and the concatenation of itself, and then ran this script. What it does is collect the outcomes of the compressions and print them to the console side by side for us to quickly detect which break the condition.

From the compressors put to test, the ones we considered invalid were pax and shar, with a difference between *C(x)* and *C(x,x)* of over 50% for an example image as x, while the others were less than 40%. Although 40% is still a very significant difference, we decided to keep most of the standard compressors as reaching a difference close to 0% was already known to be really difficult to find.

4.2 Tested Compressors

In this section we offer a small description of each of the validated standard compressors used in our experiment, chosen for their popularity and/or potential for our intentions.

Starting with **zip**, this command is a compression and file packaging utility for many OS, like Unix, VMS and MSDOS. It is analogous to a combination of the Unix commands tar and compress and it performs a lossless compression that can achieve 3:1/2:1 ratios. The standard compression method is deflate, which uses a combination of LZSS and Huffman coding.

Next we have **gzip**, a single-file/stream lossless data compression utility. This compression algorithm uses Lempel-Ziv coding (LZ77) to achieve its reduction factors. Gzip will only attempt to compress regular files and it will ignore symbolic links. Typically, text such as source code or English can be reduced by 60-70%. Compression is generally much better than that achieved by LZW, Huffman coding, or adaptive Huffman coding.

bzip2 is a free and open-source file compression program that uses the Burrows–Wheeler algorithm. It only compresses single files and is not a file archiver. This algorithm compresses most files more effectively than the older LZW and Deflate compression algorithms, but is considerably slower.

The Lempel–Ziv–Markov chain algorithm, aslo known as **lzma**, is another algorithm used to perform lossless data compression. Developed by Igor Pavlov, it was first used in the 7z format of the 7-Zip archiver. This algorithm uses a dictionary compression scheme somewhat similar to the LZ77 algorithm. It features a high compression ratio, generally higher than bzip2 but less space-efficient.

zpaq is an open source command line archiver for Windows and Linux. It uses an append-only format which can be rolled back to an earlier state to retrieve older versions. It supports fast incremental update by adding only files whose last-modified date has changed since the previous update. It compresses using deduplication and several algorithms (LZ77, BWT, and context mixing) depending on the data type and the selected compression level.

Finally we have **ppmd**, Prediction by Partial Matching, an adaptive statistical data compression technique based on context modeling and prediction. PPM models use a set of previous symbols in the uncompressed symbol stream to predict the next symbol in the stream. PPM algorithms can also be used to cluster data into predicted groupings in cluster analysis.

The conditional compressor used to test NCCD was given *a priori* for this assignment. It is based on finite context modeling and it is implemented in C++ and specialized to compress images with the help of OpenCV.

4.3   Study Pipeline

The experiment's pipeline is structured as an iterative process, built in testCompressor.sh. This script receives as its only parameter the operation to be used to concatenate images. As the concatenation can be done in more than one way, we executed the experiment for 3 different types of concatenations:

- Append - this operation unites the two images one above the other (duplicating the height of the resulting image).
- Interlace - interpolated horizontal stripes of both images are stacked together (in this operation half of the data of each image is not used).
- Average - for each pixel position, the average of the gray value of both images at that position is calculated and added to the new image in the same position.

These image manipulations were also done with the help of ImageMagick and are defined for ease of access in `mergers.sh`. Examples of these operations can be seen in Figure 3, where the first 2 faces correspond to the first images of subjects s01 and s02, and the 3 remaining images correspond to the concatenation using append, interlace and average respectively. Note that, to verify the condition discussed in section 4.1, the concatenation is done for the same image, so using either the average or the interlace operations will result in the exact same image as the original; this is why we used the append operation in that context.

Figure 3: Concatenation examples applied to subjects s01 and s02 using different operations.

Back to the experiment's script, its execution workflow follows the pseudo-code presented next, where `concat` corresponds to the chosen operation passed to the script and `compress` corresponds to the compression action executed by the current compressor of the first loop, defined in `compressors.sh`.

```
for each compressor, do:
  for each subject, do:
    minNCD = 1
    for each testImg, do:
      for each goldStdImg, do:
        tmp = concat(testImg,goldStdImg)
        cTmp = compress(tmp)
        cT = compress(testImg)
        cGS = compress(goldStdImg)
        maxS = getMaxSize(cTmp,cT,cGS)
        minS = getMinSize(cTmp,cT,cGS)
        NCD = (size(cTmp) - minS) / maxS
        if NCD < minNCD:
          minNCD = NCD
```

The output of each script execution was written to disk in different files. Then, to determine the performance of each compressor using each of the concatenation alternatives, we developed and executed a second script `calculateAccuracy.sh` that reads the output files, computes the accuracies and writes them on the console for analysis.

Along with the instructions for this assignment and the programs already developed, an additional script was provided to us to test NCCD, `TEST123`. To determine the performance of this solution, we developed `nccdAccuracy.sh` that does the same as `calculateAccuracy.sh` but for NCCD's output.

5   Results & Discussion

The results of the first phase were quite promising, with some compressors achieving accuracy percentages of over 70%. Table 1 presents the performance of NCD using each compressor and each of the concatenation techniques.

	zip	gzip	lzma	bzip2	zpaq	ppmd
Append	0.707	**0.725**	0.271	0.507	0.060	0.657
Interlace	0.100	0.560	**0.682**	0.578	0.021	0.025
Average	0.021	0.021	0.014	0.042	**0.089**	0.025

Table 1: Accuracy for each compressor using different concatenation operations.

The best compressor using the concatenation type "append" was `gzip`, using "interlace" was `lzma` and using "average" was `zpaq`. However, the average operation presented extremely low accuracies, failing to classify the faces over 90% of the time. Our understanding is that, due to the fact that this addition operation overlaps the two images, some information are loss. This phenomena can be better understood with an example. Assuming that we have a picture of person A and another from person B, as shown in 3, the average merging method overlaps the two pictures and in practical terms it can be considered that a picture of a new person is created, although for the human eye, a very inhuman person. The compression process takes advantages of existing pattern repetition in the image to compress it by removing those same repetitions. By creating a "new person", none of the patterns of the previous pictures maintain, since the "new person" with new patterns were created. This correlates directly with the fact that now the new image will never reach the compression size of any of the source images, since a picture of a person can only be compress so much. Even if the pictures are from the same person, but the angle by which the picture was taken change a few degrees, when overlapped may create a very different in relation to the original person. When comparing the compressed pictures aggregation with the original compressed photos, due to the high sensibility of this method, the

resulting compressed sizes should never be the same from a theoretical perspective, unless the images are exact copies.

Now, the reason why append presents better results than interlace is that append maintains the whole information of the original data sources, compared to interlace that discards some. This information loss is due to the limitations of the tools we used, were the best interlacing achievable was the intercalate the rows of the images, i.e., the first line of the resulting image was sourced from image 1, row 2 was sourced from image 2, row 3 from image 1, and so forth, as mentioned in 4.3. The interlacing method, by inserting only part of each source picture, the patterns that repeat in the original photo will most likely be inserted in the resulting photo, but not its repetition. This means that sometimes, although the original images being from the same person, a different photography angle may cause that the each row of the resulting aggregated image being unique. On the other hand, if the images are similar and many of the internal patterns repeat, this method will still capture that repetition and successfully compress efficiently that image. This method is still sensible, but not as much as the average method.

The append addition method is of course the most robust one, since it maintains the entirety of the the source images. This allows the compressor to have access of all the patterns and their repetitions. If the images are from the same person and in fact similar, there should only be the patterns of a person and a lot of repetition of those patterns, meaning that the compression will be very efficient and possibly achieve the compression size of one of the original photos.

But these results are very much dependent on the compression algorithms used, as one can see by the difference of accuracies amongst compressors using the same operation. The fact that `lzma` .... might have been very much beneficial when faced with an interlace-type concatenation. The inverse seems to have happened to `zip`, possibly because ....

Because "append" presented the highest performance on almost all compressors, we decided to discard the other two operators for the remaining of the experiment. Because `gzip` showed the best accuracy and presented itself as the most robust standard compressor (taking in consideration that its accuracy was also amongst the highest using "interlace"), we decided to proceed to the next phase with this compressor.

To calculate the performance of NCCD, we executed `nccdAccuracy.sh` after running `TEST123` for all dataset images. The accuracy of this solution was **0.617**, about 10% worst than the best alternative of NCD. Theoretically, NCCD should deliver better results than NCD, since the algorithm is a more direct approximation to NID, discussed in chapter 2, and a conditional compressor would benefit very much from processing faces, as they have many visual patterns to be explored. And this is verified when compared to NCD with most of the tested compressors. But, although surprising, it is not too hard to believe that NCD with `gzip` com-

pression surpassed NCCD when compared in terms of accuracy for face identification, as the difference is significant but not exaggerated.

6 Conclusions

After completing the assignment, we drew a few conclusions regarding the topics here explored and our endeavor to deliver work of quality.

The first aspect worth mentioning is how refreshing was the idea of tackling a classification problem through an unconventional strategy. The connection between the studied concepts and the project's goals were quite captivating.

The implementation was also interesting, since we decided to adopt Shell scripting and were able to work in close contact with the compression commands. Exploring the characteristics and capabilities of compressors we use on an almost daily bases will be very useful for future tasks requiring compression algorithms.

Finally, the levels of accuracy that the solutions achieved were quite impressive to us. Since we were used to developing machine learning approaches for problems relatively similar to this, we expected to see poor results from a solution that did not interpret the data in the conventional way. We were very much satisfied with the final results and the delivered work, as it accomplishes all objectives of the assignment.

In terms of code organization and readability, we made sure our repository was as well structured as possible and our code properly commented. The base folder contains a *README* file for basic instructions. The *src* and *examples* folders contain the code and other resources made available for the project (not developed by us), our scripts are on the base folder. The dataset is present in *orl_faces*, and the preprocessed images in *processedFaces*. The folder *experiments* contains the results of our studies and the scripts to calculate the performance of the solutions.

References

1. Armando J. Pinho, *AIT: Lab Work no.3*, University of Aveiro, 2019/20.

2. ImageMagick Studio LLC, *ImageMagick*, `https://imagemagick.org/index.php`, accessed in December 2019.