December, 2023

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/

Ce document contient les contributions des étudiants rédigées lors du cours "Information Algorithmique et IA" enseigné en octobre et novembre 2023.
Je suis très reconnaissant aux étudiants d'avoir été des participants actifs, et pour la qualité de leur travail.

This document contains students' contributions written during the course "Algorithmic Information and AI" taught in October-November 2023.
I am very thankful to students for having been active participants, and for the quality of their work.

Jean-Louis Dessalles

Content

December 2023

IA703

Algorithmic Information
& Artificial Intelligence
Micro-study

teaching.dessalles.fr/FCI

Name: Quentin AMIEL & Anatole REFFET

Complexity-based Text Similarity

Abstract

This project study proposes a methodology based on Gzip compression to assess textual similarity between languages, focusing specifically on the Universal Declaration of Human Rights. The normalized compression distance (NCD) is used to provide quantitative results on linguistic consistency between different versions. Partial imitation of a known linguistic database is shown by creating pairs of languages with the closest NCD values approaching Kolmogorov complexity.

Problem

We're looking to try and approximate a typical language similarity problem while using a compressor proxy. By switching up a usual similarity metric with a *NCD*, we tried to mimic what could be found in a baseline such as [1]. We do not aim to over-perform what we consider as a ground truth but to approximate it as much as possible.

Method

We've implemented a simple method based off [2], where we compress and compare the length of the compressed text to another in order to assess which are valid pairs by proximity. The information distance of two objects x and y is the sense of Kolmogorov Complexity is defined as the shortest programm that outputs x from y and vice-versa [3].

$$|p| = \max\{K(x \mid y), K(y \mid x)\}$$

The normalized information distance express the similarity between x and y.

$$NID(x, y) = \frac{\max\{K(x \mid y), K(y \mid x)\}}{\max\{K(x), K(y)\}}$$

However the initial described method [2] was used to classify text, we shift this method towards creating similar language pairs by using a compressor Z. In this paper, we'll use *gzip* as a compressor. As a matter of fact we'll also use therefore the *Normalized Compression Distance* (NCD) metric [4] as follows:

$$NCD_Z(x, y) = \frac{Z(xy) - \min\{Z(x), Z(y)\}}{\max\{Z(x), Z(y)\}}$$

We create pairs of languages, having the closest *NCD* value, to verify the relevance, we use [1] as ground-truth and we'll cherry-pick languages to verify that our method still verifies lexical similarity found within the literature.

To evaluate our results, we took as a reference the work carried out by the Universal Knowledge Core [1], which uses Levenshtein's distance to calculate a measure of similarity between 2 languages. In this way, we can check whether, for a tested language, the language predicted as being the closest by our method corresponds to the one with the greatest similarity according to [1]. For predictions that don't agree with the supposedly real targets, we've set up a score to measure how far one language is historically and linguistically from another. To do this, we refer to the Glottolog database, a tree structure of the world's languages, language families and dialects. The languoids are organized via a genealogical classification (the Glottolog tree) that is based on available historical-comparative research. Score calculation is based on the lowest common ancestor (LCA) between 2 nodes in the tree and their depths.

$$\text{Score}(L_1, L_2) = \text{depth}(L_1) + \text{depth}(L_2) - 2 * \text{depth}\left(\text{LCA}_{L_1, L_2}\right))$$

Results

While cherry-picking a few languages from dialects to main languages we found discrepancies such as when observing Russian we would find a strong proximity with *Gyliak (i.e. Nivkh)*, and *Ukrainian* on the other hand with CogNet v2. This can be explained as *Gyliak (Nivkh)* is simply not referenced in CogNet v2, however he is in Glottolog. This induces the need for another way of comparing the lexical similarity. As in that specific case, if we downgraded the NCD ranking to the next known language by CogNet v2, it would have been as well *Ukrainian*. We also seem to be able to find perfect or close to perfect matches such as for *Adyghe* or *Danish*, which sounds promising for the performance comparison of both methods.

If we sum up the issues we found were quite clear :
- Unable to handle language similarity if both languages are not in the same "family"
- Language availability discrepancies between both datasets CogNet V2 and UDHR.
- CogNet V2 seems to exclude as well languages or dialect that might be too similar to the source such as English and Scots

Languages	**CTS**	**CogNet v2 [1]**
Breton	Afrikaans	Cornish
French	Occitan (Languedocien)	Portuguese
Russian	Nivkh	Ukrainian
Japanese	Chinese	Korean
Icelandic	Faroese	Norwegian Nynorsk
Danish	Norwegian Bokmål	Norwegian Nynorsk
Abkhaz	Turkmen	Abaza
English	Scots	French
Adyghe	Kabardian	Kabardian
Dzonghka	Tibetan	Tibetan
Slovak	Czech	Czech

Table 1: Cherry-picked language similarity

Following these results, we delved deeper in creating a way of comparing these results while considering CogNet v2 as a ground truth. While using the metric described prior, in the case where both languages didn't belong to the same family, we directly assigned a score of 100 to ensure we could map them out further down the line. However, in the case where one of the ISO language could not be found, a score of NaN was assigned.

Score	**CTS**	**CogNet v2 [1]**
0 - Perfect prediction	8.28%	17.17%
1	5.86%	6.54%
2	4.85%	7.48%
100 - Different family	17.17%	20.25%

Table 2: Predictions at given score

If we check the distribution of these scores while excluding these scores standing at 100, we can however notice that CTS seems to be able to mimic partially what is achieved by CogNet V2. This goes on to show that the *CTS* method seems to perform quite weakly in comparison to the ground truth used in this paper.

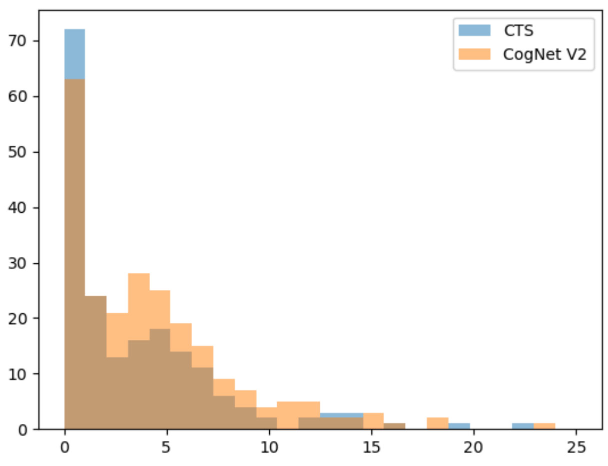


Figure 1: Distribution of scores

The findings we had earlier can be confirmed here as we can clearly notice that *CTS* under-performed in comparison to CogNet V2. However, if we go back to our main objective, which was to be able to link languages and find lexical similarity using compression, the results seems fitting.

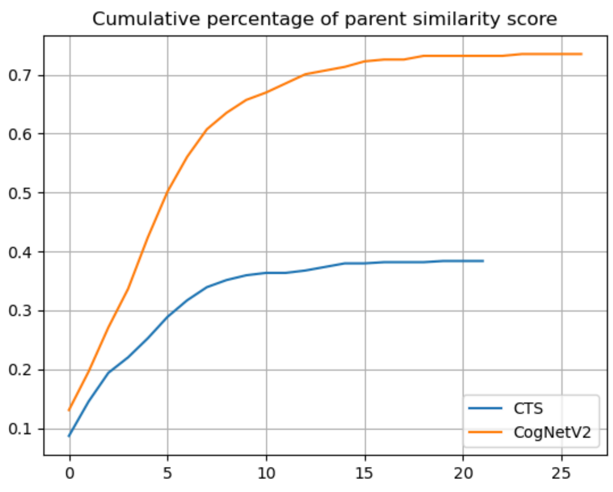


Figure 2: Comparison of CTS based method vs CogNet v2

Discussion

When starting this project, we hoped to be able to link languages and find similarity on a broad and known text such as the human's right declaration by "simply" using a compressor. This seems to have worked swiftly, however we were not able to match the performance of the ground truth we selected. This might be induced by the single text we had that was declined in 495 languages, as CogNet V2 uses multiple contemporary lexicons per language and might be more extensive. We noticed as well that the text we selected was quite short which might have impacted the compression, as the context was small and recurrence might not happen as often.

It could have been as well an interesting comparison to verify the compressors performance, as it was shown in [5], that a better *Z* compressor enables to bring closer the *NCD* to the *NID* and therefore the results would tend to be quite better. This would as well induce having more than one text at our disposal to give more context and common ground inbetween languages.

Bibliography

[1] G. Bella, K. Batsuren, and F. Giunchiglia, “A database and visualization of the similarity of contemporary lexicons”, in *International Conference on Text, Speech, and Dialogue*, 2021, pp. 95–104.

[2] Z. Jiang, M. Yang, M. Tsirlin, R. Tang, Y. Dai, and J. Lin, ““Low-Resource” Text Classification: A Parameter-Free Classification Method with Compressors”, in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, pp. 6810–6828.

[3] C. Bennett, P. Gacs, M. Li, P. Vitanyi, and W. Zurek, “Information distance”, *IEEE Transactions on Information Theory*, vol. 44, no. 4, pp. 1407–1423, 1998, doi: 10.1109/18.681318.

[4] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, “The similarity metric”, *IEEE Transactions on Information Theory*, vol. 50, no. 12, pp. 3250–3264, 2004, doi: 10.1109/TIT.2004.838101.

[5] R. Cilibrasi and P. Vitanyi, “Clustering by compression”, *IEEE Transactions on Information Theory*, vol. 51, no. 4, pp. 1523–1545, 2005, doi: 10.1109/TIT.2005.844059.

[6] H. Hammarström, R. Forkel, and M. Haspelmath, “Glottolog 3.0”. 2017.

[7] M. Voronov, “lingtypology: a Python tool for linguistic typology”. 2019. doi: 10.5281/zenodo.2669068.

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Paul Aristidou, Olivier Lapabe-Goastat

Fine-Tuning Regression Models for Peak Performance

Link to GitHub repository

Abstract

'Fine-Tuning Regression Models for Peak Performance' is a detailed investigation aimed at identifying the 'best model' in terms of complexity. This project systematically assesses various regression models, including polynomial, logarithmic, and exponential types, to determine the optimal balance between model complexity and predictive accuracy.

Problem

Our objective addresses the challenge of identifying the optimal regression model that achieves minimum description length (MDL), essentially a model that encapsulates the underlying data patterns with minimal complexity. Another interesting objective would be to observe whether a model with the smallest MDL also delivers the best performance in terms of mean squared error (MSE). This reflects the quest to find a model that not only succinctly captures the essence of the data but also predicts with high accuracy.

Method

Before delving into our methodology, we address the calculation of the Minimum Description Length (MDL), a crucial metric for model selection. In this framework, we calculate the MDL as a two-part code length:

- The first part, **the complexity of the model** $C(M_j)$, considers the number of parameters and their encoding lengths.

- The second part, **the complexity of the data given the model** $\sum_k C(x_k|M_j)$, is assessed using the Euclidean distance between the observed data points and the model predictions.

The model that minimizes the total description length, as shown below, is considered the most suitable:

$$M = \arg\min_j \left\{ C(M_j) + \sum_k C(x_k|M_j) \right\}$$

Now, to achieve our goal of finding the best regression model, we have established the following methodology :

- We begin by **scaling the feature variables**, a crucial step that normalizes the data and ensures that each variable contributes equally to the analysis, regardless of their original scale.
- Our next phase involves **constructing a variety of regression models**. Each model applies different transformations to the data: we explore polynomial, logarithmic, and exponential models to accommodate various underlying data relationships. Within each model type, we **experiment with a range of polynomial degrees**. This allows us to examine different model complexities, from simple linear relationships to higher-degree equations.
- The **training process** for each model follows, where the models are fitted to the training dataset. This fitting process is where the models 'learn' from the data, adjusting their parameters to best represent the data's structure.
- We then compute for each model the **minimum description length (MDL)**.
- We then compute the **Mean Square Error (MSE)** on a separate test dataset.
- Finally, we engage in a **comparative analysis** where we pit the models against each other to see which strikes the optimal balance. The 'best' model, in our context, is the one that not only predicts with high accuracy (as indicated by a low MSE) but also maintains a lower complexity (lower MDL).

N.B: Within each model type and for each degree of polynomial, we fine-tuned the coefficients to include multiple decimal places. This step ensured that we explored a comprehensive range of model specifications, enhancing the precision of our regression models and potentially improving their performance. What we observe is that it was best to take 0 decimal places.

Results

We will now create different datasets, and we will try different regression models. For each of these, we will calculate the Complexity and the Mean Squared Error. We will then see for which models the Complexity and the Mean Squared Error are minimized.

We will generate different datatsets to try different configurations. Each dataset will be generated around a function, to which some gaussian noise will be added. We will display only 3 datasets in this report, please feel free to see the 3 others in the annex.

Dataset 1: $y = 1 + 2x + 3x^2 - 4x^3 + noise$

In the figure below, you can see the formed dataset with 1000 points.

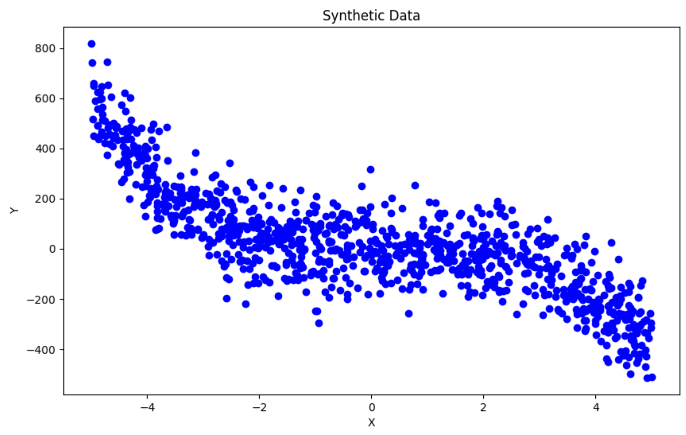


Figure 1: Dataset 1

For each regression model, we calculated the complexity with 0 to 4 decimals. In the figure below, you can see the results for a polynom of degree 2. The model where all parameters are integers is the most simple one, minimizing the complexity.

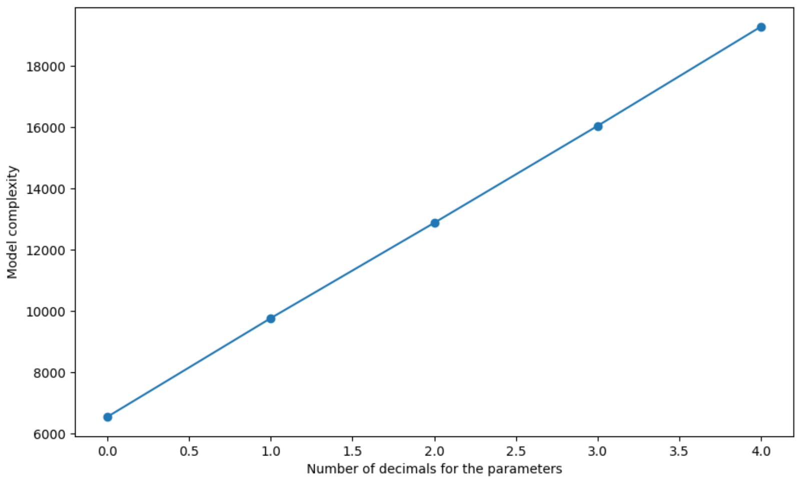


Figure 2: Dataset 1 - Complexity vs. number of decimals for the parameters

We see a similar result for all regression models, and for all other datasets. Therefore we show the results only for Dataset 1, and always keep parameters as integers.

We will now run fit the polynomial, exponential and logarithmic regressions, and let's see which ones minimizes the Complexity.

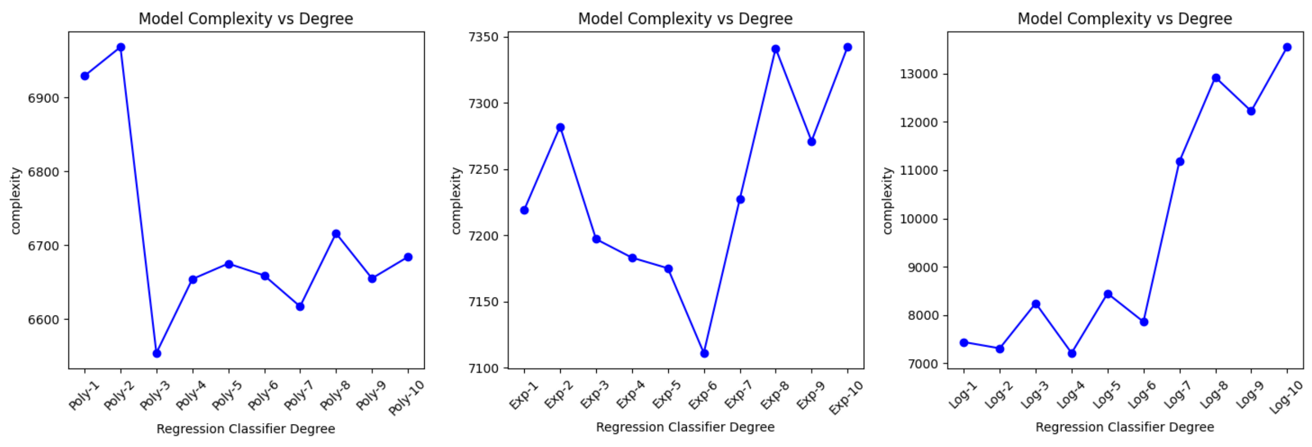


Figure 3: Dataset 1 - Complexity for different regression models

The model which minimizes the Complexity is the polynom of degree 3. This fits perfectly with the function that generated the dataset.

We will now look at which regression model minimizes the Mean Squared Error.

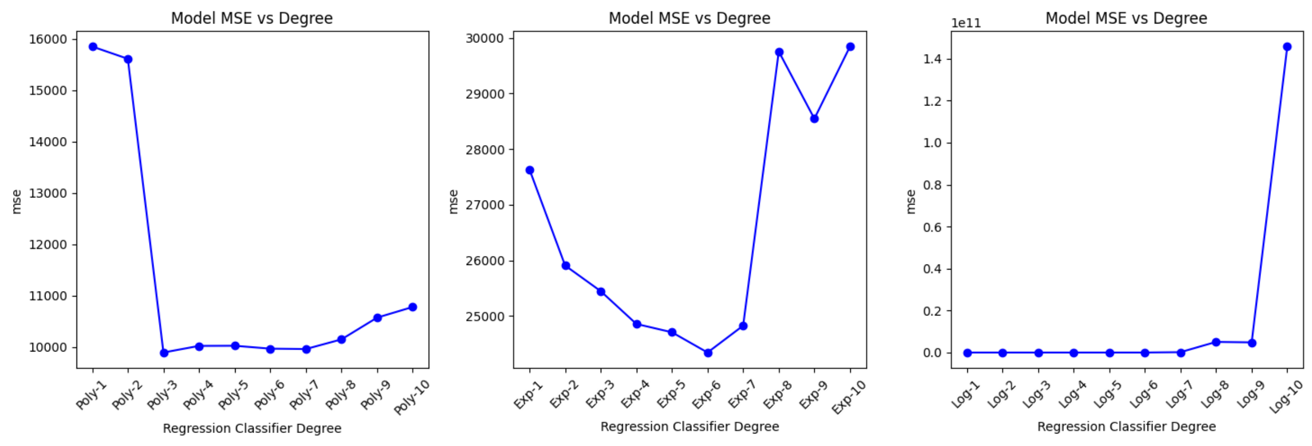


Figure 4: Dataset 1 - MSE for different regression models

The exact same regression model (polynom degree 3) minimizes as well the MSE.

We will finally plot the chosen polynom of degree 3 as "best" model.

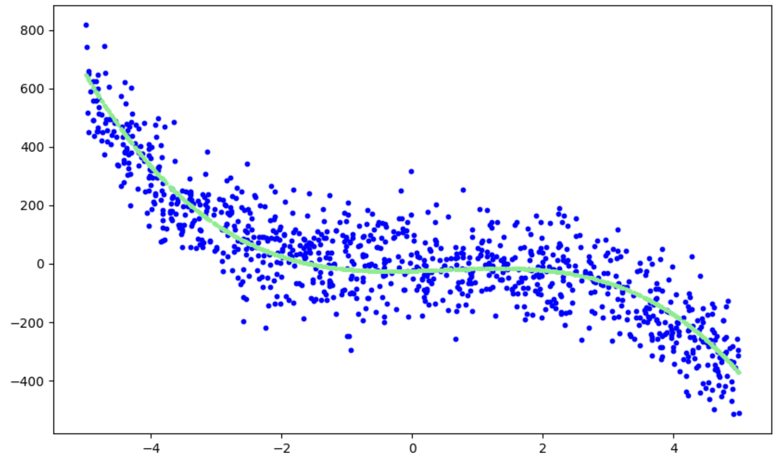


Figure 5: Dataset 1 - "Simplest" regression model minimizing Complexity

As conclusion, both Complexity and MSE approaches reach their minimum for the polynom of degree 3. The "simplest model" (minimizing the Complexity) is also the "closest" model (minimizing the MSE).

Let's now look at two other datasets.

Dataset 2: $y = 1 - exp(x) + 2exp(x)^2 - 4exp(x)^3 + noise$

As previously, the figures below show the formed dataset with 1000 points, as well as the Complexity and the MSE for each regression model.

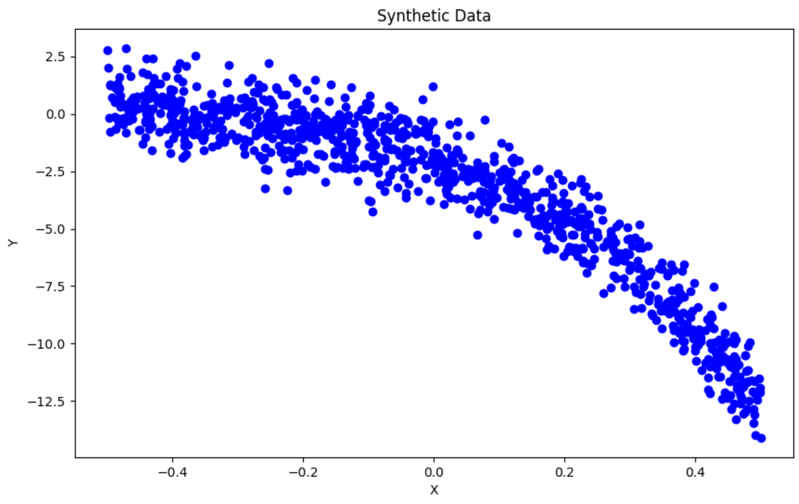


Figure 6: Dataset 2

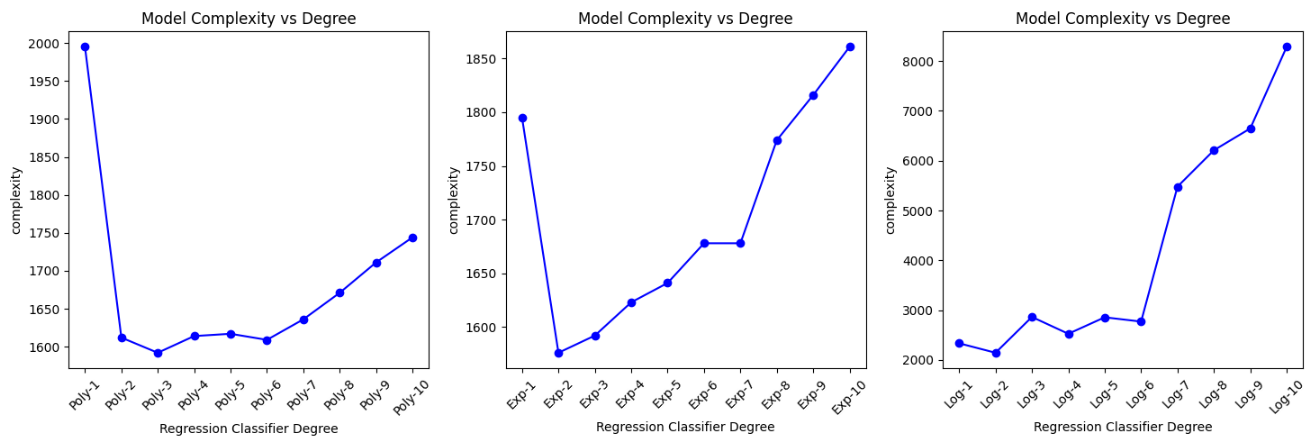


Figure 7: Dataset 2 - Complexity for different regression models

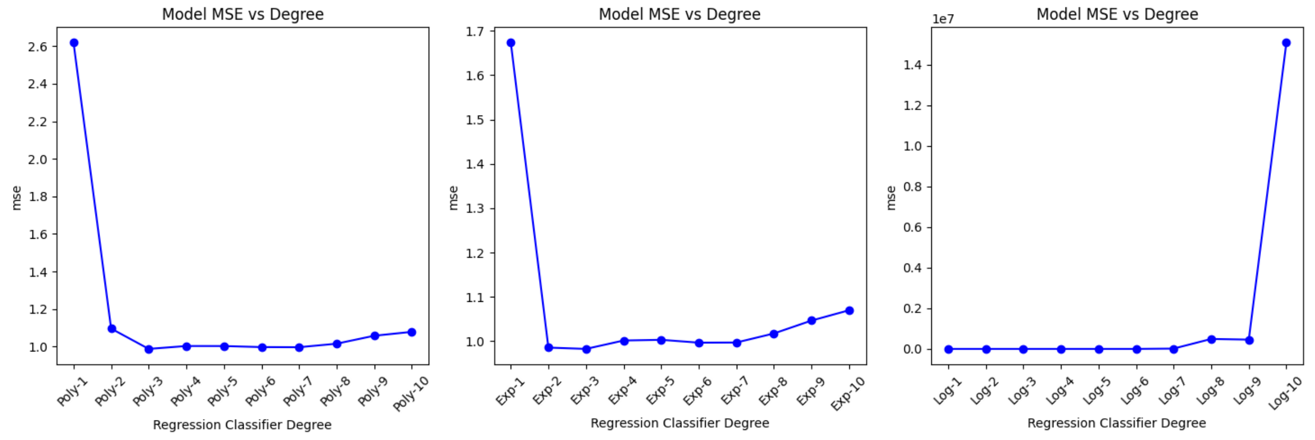


Figure 8: Dataset 2 - MSE for different regression models

For this dataset, the exponential regression model of degree 2 (Exp-2) is minimizing the complexity.
Looking at the MSE, Exp-2 is reaching 0.985 while Exp-3 is reaching 0.982. Both models are very close while the exponential regression model of degree 3 is minimizing the MSE.
This is an interesting result : eventhough Exp-3 is "closest" to the dataset, Exp-2 is the "simplest" one, and still very close to the dataset (almost as much as Exp-3). Therefore, in this case, it might make sense to choose Exp-2, as it simpler than Exp-3 and almost as close than Exp-3 to the dataset.

We will finally plot the chosen exponential polynom of degree 2 as "best" model.

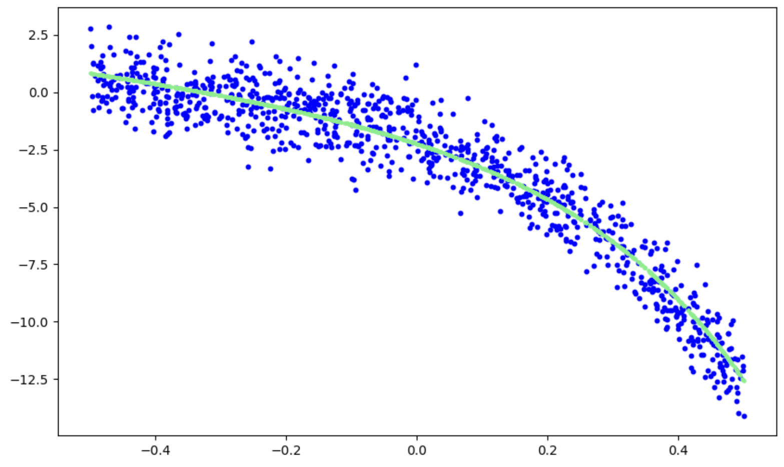


Figure 9: Dataset 2 - "Simplest" regression model minimizing Complexity

As conclusion, Dataset 2 is an interesting example where the "simplest" model (minimizing the Complexity) would be favored over the "closest" model (minimizing the MSE).

Let's now look at the third dataset.

Dataset 3: $y = 1 - 3x^2 - 3exp(x)^2 + noise$

As previously, the figures below show the formed dataset with 1000 points, as well as the Complexity and the MSE for each regression model.

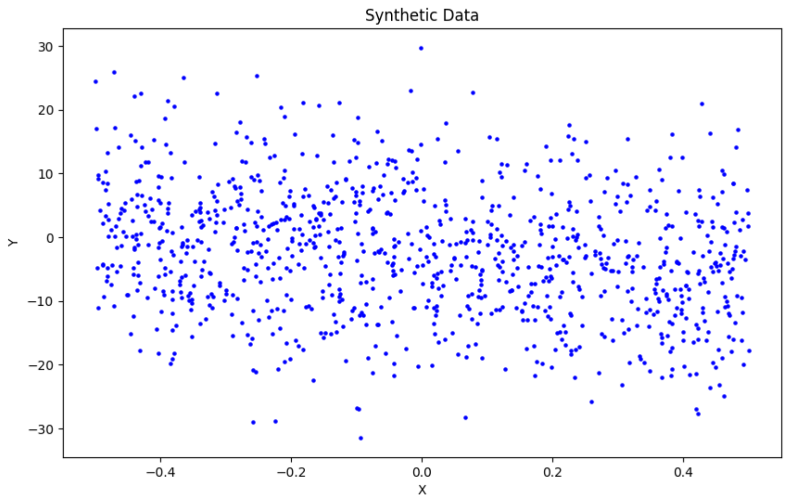


Figure 10: Dataset 3

This time, we have chosen a noise with high variance. As a consequence, the original function from which we generated the dataset is less visible. Let's see if this has an impact on the regression models minimizing the Complexity and the MSE.

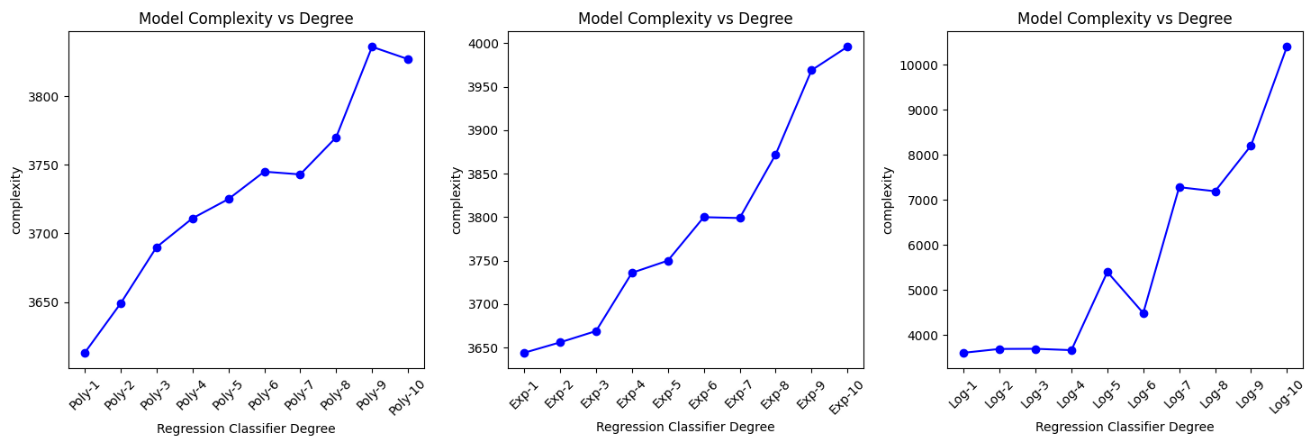


Figure 11: Dataset 3 - Complexity for different regression models

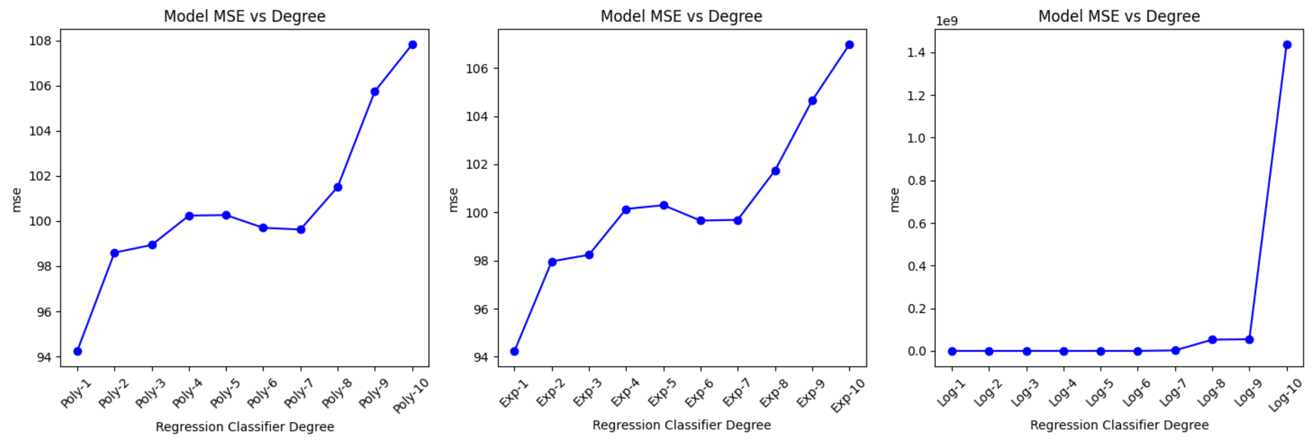


Figure 12: Dataset 3 - MSE for different regression models

We observe that the "best" regression model in the sense of the Complexity and in the sense of the MSE is the same : it's the lineear regression (Poly-1).

It might be a surprising result, given the function that generated the dataset, having a member in x^2 and a member in $exp(x)^2$. We would have expected that either the Poly-2 or the Exp-2 would have been a better fit.

This can be explained by the strong noise that we have applied. The noise is strong and the dataset is reduced between -0.5 and 0.5. As a consequence, the function is flatten by the noise and a simple Linear Regression (Poly-1) is the "best" fitting model (see figure below).

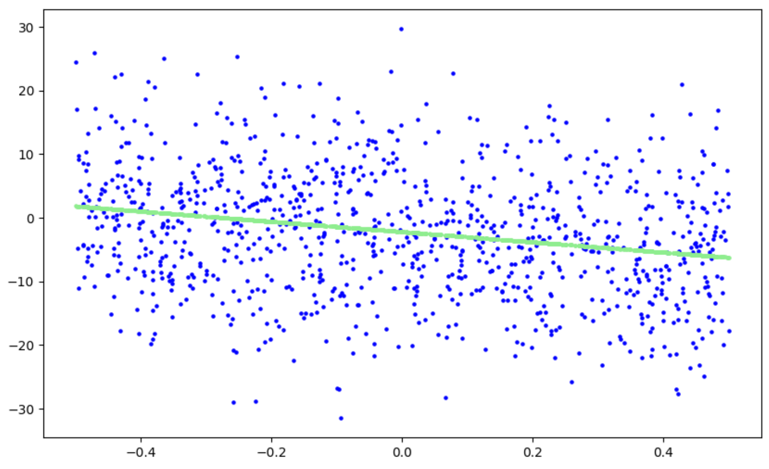


Figure 13: Dataset 3 - "Simplest" regression model minimizing Complexity

Discussion

Comparison with expectations

We have seen that the "best" Regression model in the terms of the MSE is often the same or very close to the "simplest" one in the terms of the Complexity. It confirms the initial hypothesis that minimizing the Description Length / the Complexity is a proper approach to choose the "best" regression model.

However, the MDL approach has a big advantage over the MSE approach. The MDL approach can be used without any test set, because the MDL shows an overfitting by training solely on a train set, without comparing it to a test set. On the contrary, the MSE approach on the train set with always improve with higher model complexity. Without any test set, the MSE approach cannot show an overfitting.

We can then conclude that, in the case when the amount of observations is limited and when we cannot afford to "lose" some data in splitting the dataset into train and test, the MDL is a powerful approach to choose the "best" Regression model without overfitting.

Limits

If the MDL approach has its advantages over the MSE approach, it has also some limits. The MDL approach might be less easy to be interpreted compared to the MSE one, which is more explainable. Also, calculating the Complexity of different models might involve more computational power than "simply" calculating the MSE for each Regression model.

Perspectives

One should consider the specific requirements of the application, the nature of the data, and the consequences of prediction errors. In some cases, the context of the problem may favor one approach over the other.

Also, mixing both approaches MDL and MSE might be something to investigate to get best of both worlds. Different mixing scenarios might be investigated:

- Weighing the 2 indicators and choosing the "best" regression model based on a this mixed weighted indicator.
- Without test set and having only a train set, one could have MSE as main criterion, but keeping a certain threshold of the MDL, stopping when a certain MDL threshold is reached. This would allow a model to train more extensively, but still keeping colmplexity below a certain limit.
- The MDL indicator could be compared to the BIC approach (Bayesian Indicator criterion), the BIC helping to prevent overfitting as well.

Bibliography

IA703 Course: Algorithmic Information and A.I., J-L. Dessalles
Model Selection and the Principle of Minimum Description Length, Mark H Hansen & Bin Yu, 2001
MDL Modeling — An Introduction, Jorma Rissanen, 1992

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Barberis Léo

Conway's Game of Life: An elementary Evaluation of Complexity

Abstract

Cellular automata are a family of algorithms that apply one or more simple rules to a set of cells, determining the state of each cell. The Game of Life, invented by Conway in 1970, applies the following rules to a grid of cells (see the original article in the appendix):

- Survivals: Every counter with two or three neighboring counters survives for the next generation.
- Deaths: Each counter with four or more neighbors dies (is removed) from overpopulation. Every counter with one neighbor or none dies from isolation.
- Births: Each empty cell adjacent to exactly three neighbors–no more, no fewer–is a birth cell. A counter is placed on it at the next move.

The original rules can be reformulated as follows:

> "A dead cell with exactly three living neighboring cells becomes alive (it is born); a living cell with two or three living neighboring cells remains alive, otherwise, it dies."

The goal of the project is to use some of the concepts learned in class, particularly to approximate the Kolmogorov complexity of the game, knowing that complexity is equivalent to incompressibility. Calculate the entropy and the Shannon information quantity.

Problem

How to evaluate complexity of the game of life?

Method

I used the Python programming language to implement the game of life. I partially reused the code from scienceetonnante available in github. see the appendix for the link.

Kolmogorov Complexity

I approached the Kolmogorov complexity of the grid by using the size of the grid compressed with Python's zlib function:

```python
def complexity(data: np.ndarray):
    ''' Returns the complexity of a binary array '''
    #transform the numpy array into a binary string
    s = data.tobytes()
    # Compress the array
    compressed = zlib.compress(s)

    return len(compressed)*8
```

Entropy and Shannon Information Quantity

In information theory, entropy corresponds to the expectation of the Shannon information quantity.

$$H = E[I(p_i)]$$

$$H = \sum_{i=1}^{n} p_i I(p_i)$$

The Shannon information quantity is defined as the inverse logarithm of the probability of an event:

$$I = \log_2\left(\frac{1}{p}\right) \tag{1}$$

Probability is nothing but the ratio between the frequency of an event of interest and the total number of events. Thus, I calculated the entropy as follows:

$$H = -\sum_{i=1}^{2} p_i \log_2(p_i)$$

$$pi = \frac{n_i}{N}$$

$$H = -\frac{n_1 \log_2\left(\frac{n_1}{N}\right) + n_2 \log_2\left(\frac{n_2}{N}\right)}{N}$$

$$H = \frac{n_1 \log_2(N)}{N} - \frac{n_1 \log_2(n_1)}{N} + \frac{n_2 \log_2(N)}{N} - \frac{n_2 \log_2(n_2)}{N} \tag{2}$$

This can be rewritten as:

$$\log_2(N) - \frac{1}{N}\sum_{i=1}^{2} n_i \log_2(n_i) \tag{3}$$

It should be noted that during the presentation, I had forgotten to divide by N, which skewed the results.
Finally, we can integrate the Shannon information into the entropy equation. (2):

$$H = \frac{n_1}{N}(\log_2(N) - \log_2(n_1)) + \frac{n_2}{N}(\log_2(N) - \log_2(n_2)) = \frac{n_1}{N}I_1 + \frac{n_2}{N}I_2 \tag{4}$$

The Shannon information quantities of living and dead cells are calculated with the following Python code:

```python
def entropy(data: np.ndarray):
    ''' Returns the entropy of a binary array '''
    #get all the values = True
    n1 = np.count_nonzero(data)
    #get all the values = False
    n0 = data.size - n1

    # Compute the entropy
    if n0 == 0 or n1 == 0:
        return 0
    else:
        return math.log(data.size,2) - (1/data.size)*((n0*math.log(n0,2) +
        ↪  n1*math.log(n1,2)))

```

Results

Is Randomness Complex?

In this experiment, we start with a grid of random cells. The first graph illustrates game of life at initial state and after 500 iterations. We observe that the curves representing entropy, complexity, and Shannon information quantity for the dead cells are identical, with only a slight constant variation.

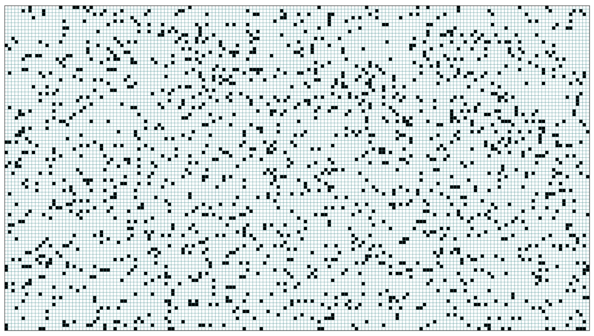


Figure 1: Random grid with p = 0.1

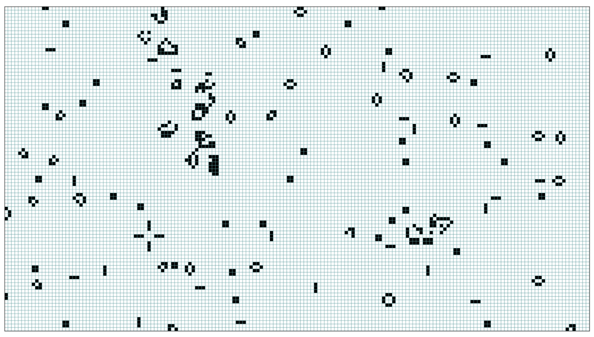


Figure 2: Random grid with p = 0.1 after 500 iterations

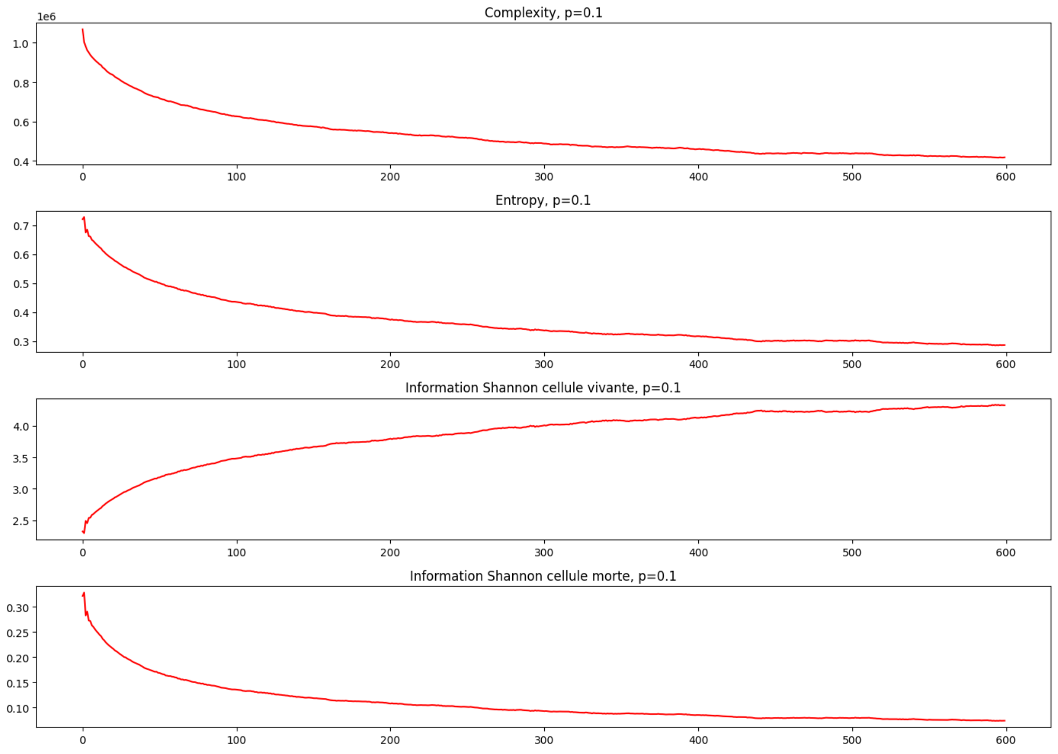


Figure 3: Calculation of different measures for living p = 0.1

From Simplicity to Complexity

The following experiment presents the R-pentomino, which from 5 cells produces a lot of complexity.

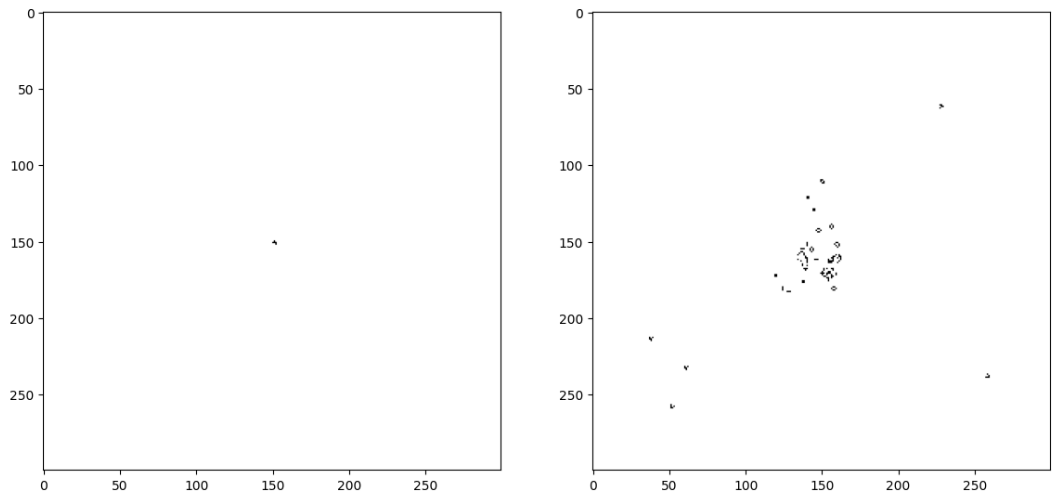


Figure 4: Initial result (left) and after 500 iterations (right)

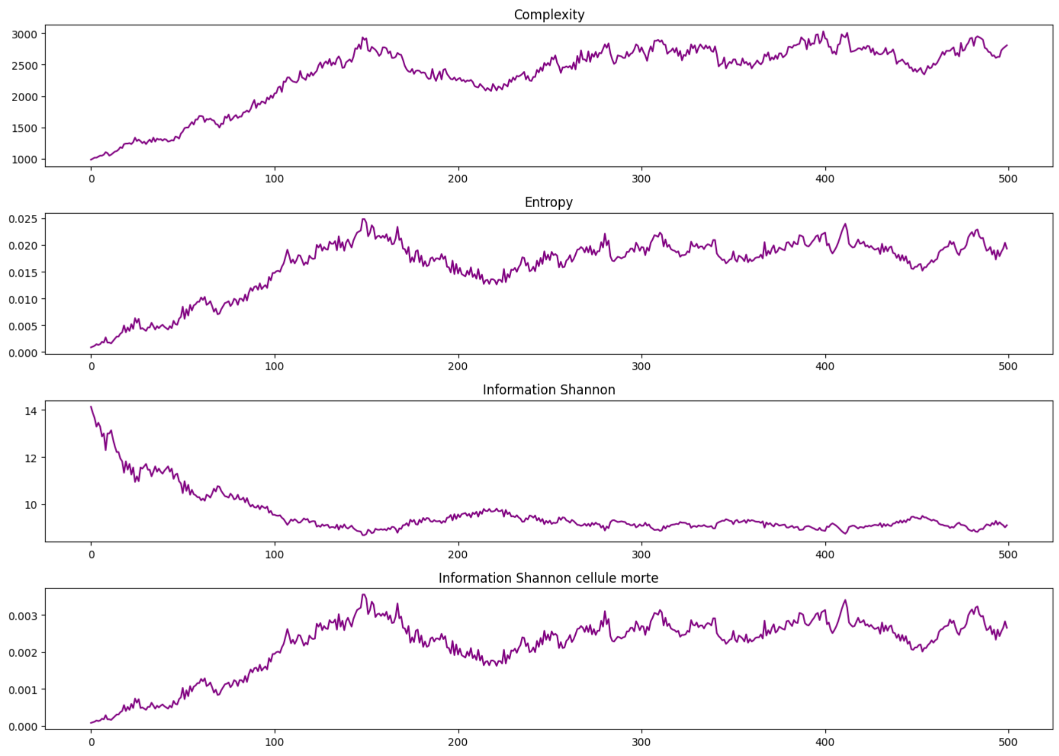


Figure 5: The measures increase over time

Is Algorithmic Information Theory Complex or Simple?

Finally, to conclude, I calculated the complexity of the string "Algorithmic Information Theory".

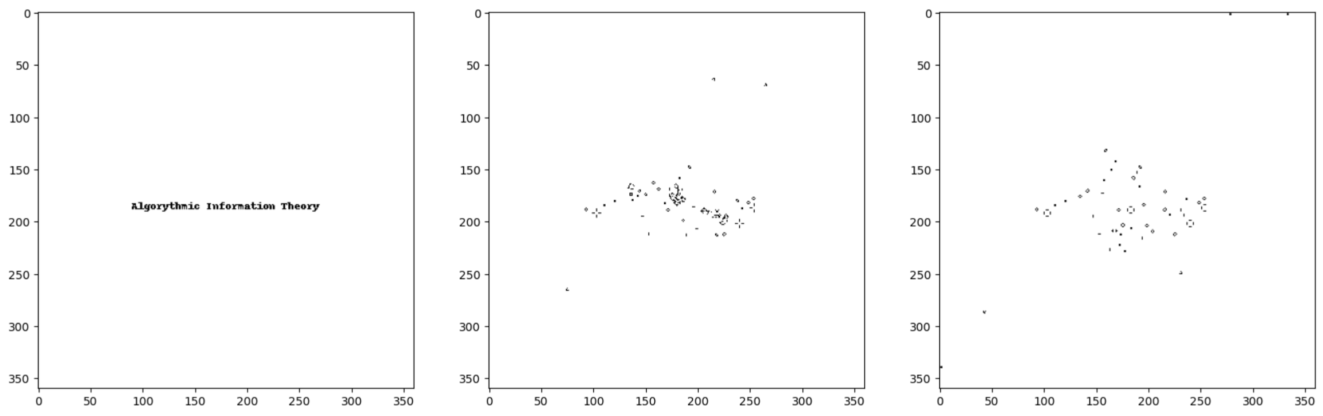


Figure 6: Initial result (left), after 500 iterations (center), after 1000 iterations (right)

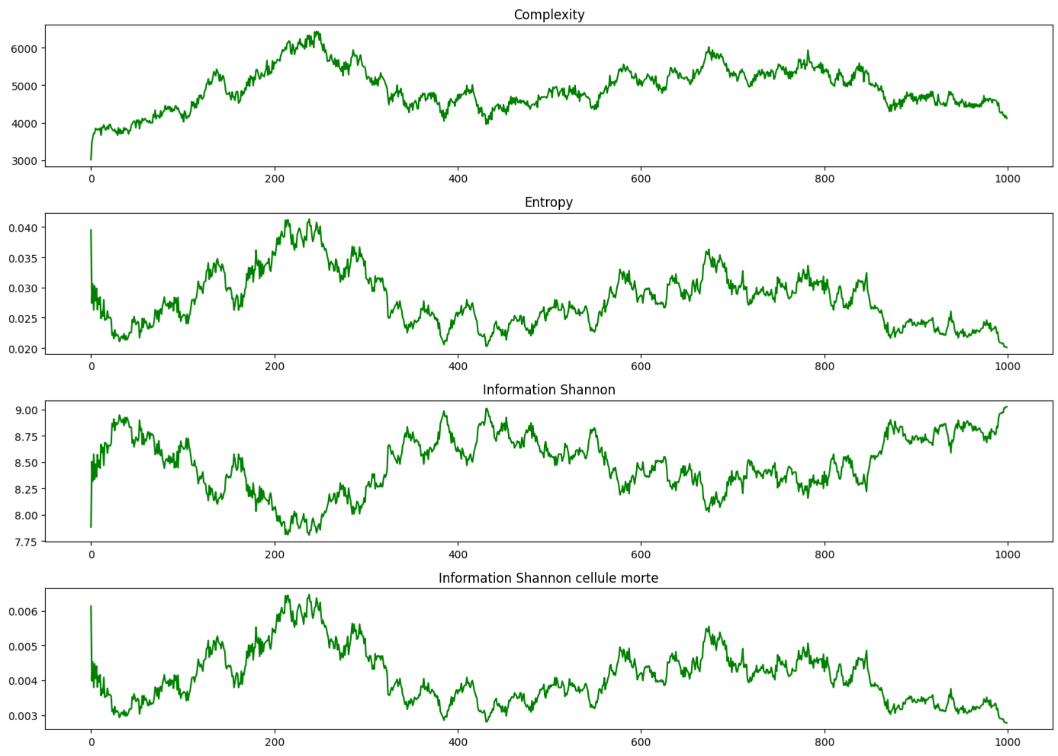


Figure 7: The curves grow and decrease without any particular trend

Discussion

The results demonstrate that the complexity curves, obtained by compression, are identical, within a constant margin, to the entropy curves and the Shannon information quantity of the dead cells. I calculated the cosine similarity measure between the complexity and entropy curves for both dead and living cells in the last experiment. This revealed a strong collinearity between these curves.

Table 1: Cosine Similarity

	Complexity	**Entropy**	**Information Dead**
Complexity	-	0.9964	0.9943
Entropy	0.9964	-	0.9996
Information Alive	0.9899	0.9817	-

This can be explained by the fact that entropy is the mathematical expectation of the Shannon information quantity (4). It is remarkable to note that complexity is extremely correlated with entropy. Consequently, to delve deeper, we could consider estimating entropy from complexity. Finally, to answer the problem does the game of life complex? Hard to tell we have seen that from a random grid the complexity is hight at start but then decrease. On the other hand, we have seen that from a simple pattern the complexity increases over time. So it depends on the initial grid.

Bibliography

References

[1] *Original article by Conway*:
The fantastic combinations of John Conway's new solitaire game "life". https://www.ibiblio.org/lifepatterns/october1970.html

[2] The game of life Wikipedia!
https://conwaylife.com/wiki/

[3] A popular video on the game of life
https://www.youtube.com/watch?v=S-W0NX97DB0

[4] Some source code used for the projecs
https://github.com/scienceetonnante/ConwayLife

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Names: Mohamed Reda Chenna & Quentin Barthélémy

Analyse de la complexité d'un texte dans la littérature par le biais d'algorithmes de compression

Abstract

L'étude compare la richesse lexicale de 4 langues (français, espagnol, italien, allemand) dans la poésie et le roman, en utilisant des algorithmes de compression.

Problem

L'objectif de ce projet est de comparer la richesse du vocabulaire de 4 langues différentes : le français, l'espagnol, l'italien et l'allemand.

Nous allons pour cela analyser le vocabulaire de chacune de ces langues au travers de deux grands genres littéraires : la poésie et le roman.

Nous tenterons ainsi de répondre à 2 questions :

- Quelle langue emploie le plus de vocabulaire ?
- Y a-t-il une différence de richesse lexicale entre les genres littéraires ?

Method

Constitution d'un corpus de données

Pour pouvoir réaliser ce projet, la première étape est de trouver un corpus de poèmes et de romans pour chaque langue que nous avons décidé d'étudier.

Après maintes et maintes recherches sur le Web, et en particulier sur des sites tels que Kaggle, contenant des bases de données sur de nombreux sujets, nous ne sommes pas parvenus à trouver les données qui nous intéressent.

Pour pallier ce problème, nous avons donc décidé de constituer notre propre jeu de données.

Attardons-nous d'abord sur le corpus de poèmes. Au cours de nos recherches, nous sommes tombé sur le site barapoemes.net/archives qui contient plusieurs centaines de poèmes en français, mais aussi une centaine de poèmes en allemand, en italien et en espagnol. C'est exactement ce dont on a besoin.

Nous avons ainsi commencé à copier-coller les poèmes en français un à un dans un fichier txt, mais, au bon de quelques manipulations, nous nous sommes vite rendus compte que récolter toutes ces données à la main nous prendrait au moins un dizaine d'heures : c'est impensable !

Il fallait donc trouver une solution pour accélérer le processus et nous avons donc logiquement pensé à construire un algorithme de scrapping pour récolter ces données automatiquement.

L'algorithme de scrapping

Passons en revue les grandes étapes de cet algorithme.

Nous avons besoin de deux librairies Python pour construire notre script : **requests** et **BeautifulSoup**.

```python
import requests
from bs4 import BeautifulSoup
```

La librairie **requests** est utilisée pour envoyer des requêtes HTTP de manière simple et intuitive. Elle nous permet d'intéragir facilement avec des sites web en envoyant des requêtes pour récupérer ou publier des données.

La librairie **BeautifulSoup**, quant à elle, est une librairie utilisée pour analyser / parser des documents HTML. Elle simplifie l'extraction d'informations spécifiques d'une page web, en nous permettant de naviguer dans le document HTML, de chercher des éléments spécifiques.

Au début du code, on spécifie le fichier où l'on va stocker les données, et on envoie une requête sur une page Internet (que l'on spécifie également) pour récupérer son contenu.

```python
# The file where data will be stored
filename = "poems/data/poems_fr.txt"

# URL of the website and get request
url = "http://www.barapoemes.net/archives/en_francais/index.html"
response = requests.get(url)
```

Une fois le contenu correctement récupéré, on utilise **BeautifulSoup** pour récupérer uniquement le titre et les liens vers les poèmes présents sur la page (une dizaine par page).

```python
# Check if the request was successful
if response.status_code == 200:

    # Parse the HTML content using BeautifulSoup
    soup = BeautifulSoup(response.content, 'html.parser')

    # Extract all <h2> tags within divs having class="item_div"
    h2_tags = soup.find_all('div', class_='item_div')

    # Extracting the text and link from each <h2> tag -> poems
    extracted_data = []
    for tag in h2_tags:
        h2 = tag.find('h2')
        if h2 and h2.a:
            title = h2.get_text(strip=True)
            link = h2.a.get('href')
            extracted_data.append({'title': title, 'link': link})
```

Enfin, on lance une requête sur chaque lien de poème récupéré pour en extraire le texte.

```python
for poem in extracted_data:
    url = poem['link']

    # Send a GET request to the website
    response = requests.get(url)

    if response.status_code == 200:
        # Parse the HTML content using BeautifulSoup
        soup = BeautifulSoup(response.content, 'html.parser')
        #print(soup)
        poem_lines = []
        paragraphs = soup.find_all('p')
        for paragraph in paragraphs:
            spans = paragraph.find_all('span', style='font-family: times new
   roman, times; font-size: 14pt;')
            for span in spans:
                poem_lines.append(span.get_text())

        poem_lines_str = ""
        for sentence in poem_lines[1:-2]:
            poem_lines_str += " ".join(sentence.split()) + '\n'

        # Write the text of each line of the poem in a file
        with open(filename, 'a', encoding='utf-8') as file:
            file.write(poem['title'] + "\n\n" + poem_lines_str + "\n\n")
```

Au cours de ce processus de collecte des données, nous avons fait attention à homogénéiser notre corpus, c'est-à-dire à collecter le même nombre de mots pour chaque ensemble de poèmes (un pour chaque langue étudiée) pour que notre analyse ultérieure soit équilibrée et pertinente. Nous avons ainsi des fichiers d'environ 12 500 mots chacun, ce qui correspond à près de 80 poèmes par langue.

Une fois les données sur les poèmes récupérées, nous avons réalisé un deuxième script Python dans le but de les nettoyer. En effet, ce qui nous intéresse est d'analyser la richesse de vocabulaire des langues : nous pouvons donc retirer les espaces surperflus entre les mots, retirer la ponctuation et tout mettre en minuscule pour faciliter notre analyse ultérieure. Nous avons également hésité à utiliser des techniques de lemmatisation ou de stemmatisation sur nos données, mais nous nous sommes finalement dit que les différentes formes que peut prendre un mot dans une langue (pluriel/singulier, masculin/féminin) participent à la richesse du vocabulaire de la langue.

L'algorithme de cleaning

Ce script prend en entrée le chemin de deux fichiers : le fichier à nettoyer et le fichier cible. Il appelle ensuite la fonction qui réalise le traitement.

```python
import string

original_file = "poems/data/poems_fr.txt"
new_file = "poems/cleaned/poems_fr_clean.txt"
formater_texte(original_file, new_file)
```

La fonction de cleaning réalise ensuite les différentes étapes que l'on a cité précédemment (retirer la ponctuation, retirer les espaces en trop et mettre le texte en minsucule) et stocke le résultat dans le nouveau fichier spécifié.

```python
def formatter_text(original_file, new_file):
    try:
        with open(original_file, 'r', encoding='utf-8') as file:
            content = file.read()

        content_without_ponctuation = content.translate(str.maketrans('', '',
    string.punctuation)).lower()
        words = content_without_ponctuation.split()

        # Rearrange text with 15 words per line
        format_text = ''
        for i, word in enumerate(words):
            format_text += word + ' '
            if (i + 1) % 15 == 0:
                format_text += '\n'

        # Write formatted text to a new file
        with open(new_file, 'w', encoding='utf-8') as output_file:
            output_file.write(format_text)

        return "Formatting successfully completed."
    except FileNotFoundError:
        return "The file was not found."
    except Exception as e:
        return f"An error has occurred: {e}"
```

En ce qui concerne maintenant le corpus de roman, nous avons décidé d'utiliser un extrait d'un unique roman, traduit dans chacune des langues étudiées : Alice au pays des merveilles.

On a veillé à conserver un extrait d'environ 12 500 mots pour chaque langue, afin de pouvoir en faire une comparaison pertinente avec le corpus du genre de la poésie par la suite. Comme pour la poésie, nous avons ensuite appliqué notre script de cleaning à ces données.

Les algorithmes de compression

Une fois les données collectées, la deuxième étape est maintenant de réfléchir aux algorithmes que l'on va pouvoir utiliser pour mener à bien notre étude.

Dans le cadre de la théorie de l'information algorithmique, l'information est définie comme ce qui persiste une fois la répétition éliminée. Notre objectif étant d'évaluer la diversité lexicale, nous avons entrepris une étude comparative des différentes langues et genres littéraires. Cette analyse s'appuie sur le taux de compression de chaque texte, permettant ainsi de mesurer et de comparer la richesse de leur vocabulaire.

Pour le faire, nous nous sommes basés sur 2 algorithmes qui sont: **Deflate** *via zlib* et **bzip2** *via bz2*.

DEFLATE (via zlib):

L'algorithme DEFLATE, utilisé pour la compression de données sans perte, combine efficacement la technique LZ77 avec le codage Huffman.

Dans LZ77, une partie cruciale du processus est l'utilisation d'une "sliding window", ou fenêtre glissante, qui parcourt les données à compresser. Cette fenêtre divise les données en sections déjà vues (le buffer de recherche) et les données à venir (le buffer prévisionnel). Lorsque DEFLATE repère une séquence répétitive dans le buffer prévisionnel qui correspond à une partie du buffer de recherche, il la remplace par une référence plus courte indiquant la distance à la répétition et sa longueur. Cette méthode élimine les redondances et réduit la taille des données. Ensuite, le codage Huffman est appliqué aux données issues de LZ77, attribuant des codes binaires de longueurs variables en fonction de la fréquence des éléments. Les éléments fréquents reçoivent des codes courts, ce qui optimise davantage la compression.

Cette combinaison de fenêtre glissante de LZ77 et du codage Huffman rend l'algorithme DEFLATE particulièrement efficace pour la compression de texte car il peut efficacement minimiser les redondances typiques dans les langues et représenter les données de manière très compacte, nous permettant donc d'obtenir des résultats efficaces.

On peut résumer le principe de l'algorithme selon le pseudo-code suivant :

Algorithm 1 Deflate

```
1: Input: A string of text or binary data
2: Output: Compressed data
3: Initialize a sliding window
4: Initialize an empty output stream
5: while there is more data in the input: do
6:     Use LZ77 to find the longest match in the sliding window for the current lookahead buffer
7:     if a match is found: then
8:         Output a pointer (distance and length) to the match in the sliding window
9:     else:
10:        Output the current symbol in the lookahead buffer
11:    end if
12:    Slide the window forward
13: end while
14: Apply Huffman coding to the output of the LZ77 step
15: Return the Huffman coded data as the final compressed output
```

bzip2 (via bz2):

De manière similaire à DEFLATE, l'algorithme Bzip2 est une méthode de compression de données sans perte qui se distingue par son efficacité, en particulier pour la compression de textes.

Il commence par appliquer la Transformation de Burrows-Wheeler (BWT) sur les données, une technique qui réarrange les caractères du texte de manière à regrouper les séquences similaires, facilitant ainsi leur compression.

Après la BWT, le Move-to-Front (MTF) est utilisé, un processus qui transforme les données en plaçant les éléments fréquemment utilisés au début d'une liste, ce qui rend la séquence encore plus compressible.

Ensuite, Bzip2 applique une codification de longueur de course (Run-Length Encoding, RLE), qui remplace les séquences répétitives par un seul caractère suivi du nombre de répétitions.

Finalement, le codage Huffman est utilisé pour compresser davantage les données en attribuant des codes courts aux éléments fréquents.

Cette combinaison de transformations rend Bzip2 extrêmement efficace pour compresser des textes, en réduisant significativement leur taille, tout en assurant une décompression fidèle pour restaurer les données originales.

L'algorithme suivant permet de résumer les différentes étapes suivi dans Bzip2 :

Algorithm 2 BZIP2

```
1: Input: A string of text
2: Output: Compressed data
3: bwtOutput <- Apply Burrows-Wheeler Transform to the input
4: mtfOutput <- Apply Move-to-Front Transform to bwtOutput
5: rleOutput <- Apply Run-Length Encoding to mtfOutput
6: compressedData <- Apply Huffman Coding to rleOutput
7: Return compressedData
```

Results

La méthode implémentée, on peut maintenant appliquer les algorithmes de compression sur notre corpus de données.

Pour le premier algorithme, DEFLATE, les résultats que nous obtenons sont les suivants :

	Français	Allemand	Italien	Espagnol
Poème	62.9%	60.1%	60.5%	62.1%
Roman	64.9%	65.2%	64.0%	65.3%

Et pour le deuxième algorithme, Bzip2, les résultats sont les suivants :

	Français	Allemand	Italien	Espagnol
Poème	68.7%	65.3%	66.1%	66.8%
Roman	70.3%	71.3%	69.2%	70.8%

On ne s'attardera pas sur la comparaison entre les deux algorithmes car ce n'est pas le sujet de notre discussion. Cependant, on peut tout de même remarquer que l'algorithme Bzip2 semble plus performant que DEFLATE (avec un taux de compression environ 6% au-dessus).

Nous ce qui nous intéresse c'est, dans un premier temps, de comparer le taux de compression des langues selon le genre littéraire. Le but est de répondre à la question initiale :

Quelle langue emploie le plus de vocabulaire ?

Pour les poèmes, on remarque que les taux de compression sont relativement proches d'une langue à l'autre (et ce pour les deux algorithmes). Il est ainsi difficile de conclure avec assurance sur la langue qui présente la plus grande richesse de vocabulaire. Néanmoins, l'allemand semble présenter le plus faible taux de compression, pour les deux algorithmes, ce qui se traduit par le fait qu'il contient plus d'informations que les autres langues et donc qu'il possède une plus grande richesse lexicale dans la poésie.

Dans le cas du roman, le raisonnement est similaire à celui pour le genre de la poésie. Cependant, les résultats diffèrent : c'est l'italien qui présente le plus faible taux de compression dans cet exercice, et donc la langue qui présente la plus grande richesse lexicale dans le roman.

Nous souhaitons maintenant nous attarder sur la deuxième question que nous nous sommes posés dans cette étude :

Y a-t-il une différence de richesse lexicale entre les genres littéraires ?

Si on reprend les tableaux comparatifs, on remarque que le roman est le genre qui est le plus compressible et ce pour chaque langue. On peut donc conclure que la poésie semble être le genre litttéraire qui possède la plus grande diversité lexicale.

Discussion

Nous sommes ainsi parvenus à apporter une réponse aux deux questions que nous nous sommes posés dans cette étude. Mais, comme dit dans la partie précédente, les conclusions faites ne sont pas forcément 100% fiables et nous allons donc ici nuancer ces réponses.

D'abord, dans la comparaison entre la poésie et le roman, nous avons conclu que la poésie est moins compressible et donc possède une plus grande richesse lexicale.

Cependant, la conclusion peut être biaisée par le fait que le roman choisi pour mener cette étude (Alice au pays des merveilles) est un roman tout public et donc destiné à être compris par tous : il ne serait pas surprenant que le vocabulaire employé soit réduit par rapport à d'autres romans.

De plus, nous n'avons pris qu'un extrait de ce roman afin de garder un nombre de mots équivalent par rapport au corpus de la poésie (12 500 mots) : cela a aussi pour conséquence de réduire le champs de vocabulaire possible.

Pour apporter une conclusion plus pertinente à cette question, il faudrait donc, sans doute, augmenter la taille de notre corpus et diversifié les auteurs / les types de romans.

Deuxième remarque, dans la comparaison de la richesse lexicale des langues, nous avons observé que les taux de compression entre les différentes langues, pour un même genre littéraire, sont très proches.

Encore une fois, le biais ici pourrait venir de la taille de nos données : seulement 80 poèmes par langue, 12 500 mots environ : La taille de ces données n'est pas suffisante comme pour pouvoir départager les langues au vu des taux de compression relativement proches entre celles-ci.

Bibliography

[1] http://www.barapoemes.net/archives/index.html
Référence des centaines de poèmes dans plusieurs langues.

[2] https://docs.python.org/fr/3/library/zlib.html
Documentation officielle de la bibliotheque zlib sur python.

[3] https://docs.python.org/fr/3/library/bz2.html
Documentation officielle de la bibliotheque bz2 sur python.

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Anaële BAUDANT-COJAN

Compression and correlation in financial time series

Abstract

Financial assets are more and more correlated. We will use compression as an estimator of shared information between different time series.

Problem

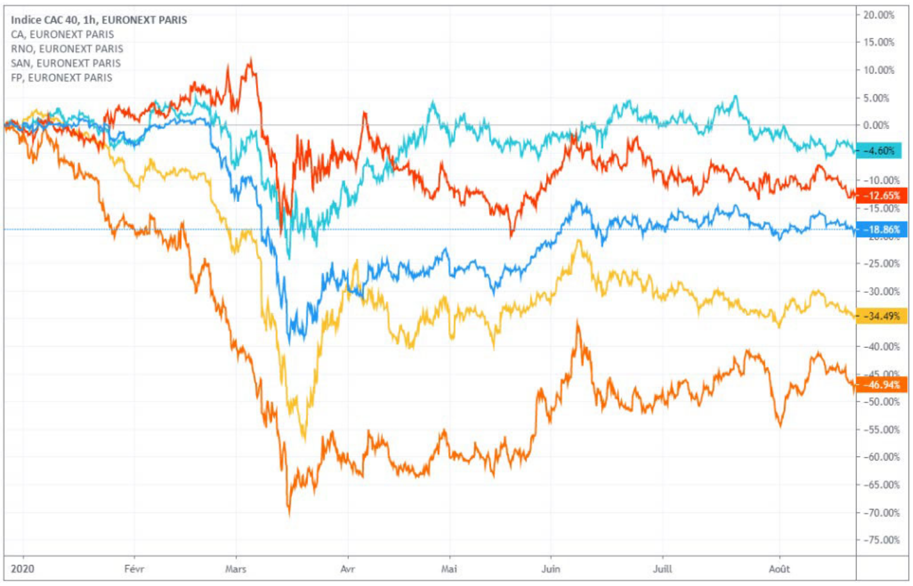
How can we use stock price's structure to compress data and define models for compression (no loss/lossy).

Method

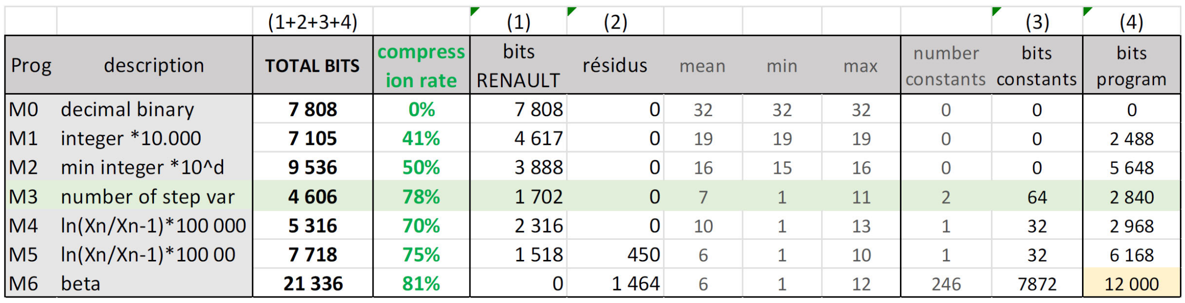
We compare the price evolution of different stock values (Renault, LVMH, Meta, Microsoft) to different benchmarks (dow jones, gold, CAC40, weather forecast, km of traffic jam in Paris Region, or in Manhattan, etc). We measure the length of the code required to code "Renault knowing CAC40" or "Renault knowing outside temperature" compared to the length of Renault on its own to measure similarities. We will finally try and explore how compression can be a proxy of different financial concepts such as diversification (on a "basket of values") or volatility.

Results

cours min	cours max	Plus de 9.000	granularité
0	0,1	0,0001	0,10%
0,1	0,2	0,0001	0,05%
0,2	0,5	0,0001	0,02%
0,5	1	0,0001	0,01%
1	2	0,0002	0,01%
2	5	0,0005	0,01%
5	10	0,001	0,01%
10	20	0,002	0,01%
20	50	0,005	0,01%
50	100	0,01	0,01%
100	200	0,02	0,01%
200	500	0,05	0,01%
500	1 000	0,1	0,01%
1 000	2 000	0,2	0,01%
2 000	5 000	0,5	0,01%
5 000	10 000	1	0,01%
10 000	20 000	2	0,01%
20 000	50 000	5	0,01%
50 000	100 000	10	0,01%



CAC40 versus Carrefour, Renault, Sanofi, Total Energies



		(1+2+3+4)		(1)	(2)					(3)	(4)
Prog	description	**TOTAL BITS**	compress ion rate	bits RENAULT	résidus	mean	min	max	number constants	bits constants	bits program
M0	decimal binary	**7 808**	**0%**	7 808	0	32	32	32	0	0	0
M1	integer *10.000	**7 105**	**41%**	4 617	0	19	19	19	0	0	2 488
M2	min integer *10^d	**9 536**	**50%**	3 888	0	16	15	16	0	0	5 648
M3	number of step var	**4 606**	**78%**	1 702	0	7	1	11	2	64	2 840
M4	ln(Xn/Xn-1)*100 000	**5 316**	**70%**	2 316	0	10	1	13	1	32	2 968
M5	ln(Xn/Xn-1)*100 00	**7 718**	**75%**	1 518	450	6	1	10	1	32	6 168
M6	beta	**21 336**	**81%**	0	1 464	6	1	12	246	7872	12 000

Remarques :
-Cout des programmes M0 à M5 très comparables (< 1ko)
-M4 : 75 valeurs fausses sur 244 valeurs entre –4,4% et +3,9%
-M6 : 446 constantes = X0 + beta + 244 valeurs du benchmark !
calcul des betas fait sur excel, estimation des bits du programme = fourchette très basse

BEST RESULT = variation in numberof steps
BETA (CORRELATION) = interestingmethod… but verycostly!

Discussion

Comparison with expectations, limits, perspectives.

Sources

- Financiales series : Yahoo Finance
- Temperature anomalies : National Centers for Environmental Information (NCEI) (noaa.gov)

  https://www.ncei.noaa.gov/access/monitoring/climate-at-a- glance/global/time-series/globe/land_ocean/all/11/2000-2023
- Températures à Paris : Meteociel - Climatologie mensuelle de Paris - Montsouris (75) www.meteociel.fr

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Names: Ahmed BELAAJ & Nour BEN REJEB

Anomaly detection using K-Nearest Neighbors and Huffman code

Abstract

The project focuses on anomaly detection in time series data, specifically daily temperature variations in Paris. It compares Huffman coding and K-nearest neighbors (KNN) distance averaging for outlier detection. Huffman coding compresses the temperature data based on the frequency of data points, facilitating the identification of outliers. Meanwhile, KNN calculates the average distances between data points to detect outliers based on predefined thresholds.

1 Problem

This work explores classification and compression techniques in the context of anomaly detection. We want to tackle the problem of outliers and evaluate which technique is more relevant and provides better results.
We can notice that traditional anomaly detection techniques face significant challenges regarding data handling and efficiency. Conventional techniques often require extensive computational resources for data compression, leading to high storage and memory usage.
This raises a critical need to explore alternative methods like Huffman coding for instance, which promises reduced data size and potentially faster processing due to its unique approach to data compression. We tried also to evaluate this method by comparing it to KNN investigating the combined effectiveness of Huffman coding and K-Nearest Neighbors for anomaly detection.
We raise the following question that we expect to answer throughout this report:
To what extent are K-Nearest Neighbors and Huffman coding relevant in anomaly detection? Are there particular aspects or particular characteristics that make them more or less effective?

2 Methods

2.1 Data Presentation

The dataset we have selected for our project describes temperature variations in the french capital, with 9,266 records spanning from 1995 to 2020. Sourced from Kaggle[1], this dataset allows us to trace daily temperature patterns in Paris. Each entry in the table corresponds to a single day, organized into eight columns as follow:
1: continent, **2:** country, **3:** city, **4:** day, **5:** month, **6:** year, **7:** temperature, and a full date column that consolidate the information from columns "4", "5", and "6" into a standard date format (day-month-year). Our study is based on this dataset, which provides insights into Paris's climate changes over 25 years.

2.1.1 Data Normality

In this section, we analyze the normality of our dataset and observe that the majority of temperature values are closely aligned with a normal distribution, centered around 52 Kelvin.
The bell-shaped curve in figure **??** illustrates this distribution, although there are notable outliers, particularly a cluster of values at -99 Kelvin, which represent missing or null records. To address these anomalies, we plan to use some advanced techniques such as K-nearest neighbors (KNN) and Huffman encoding for outlier detection.
These methods will be discussed further in the subsequent sections of the report.

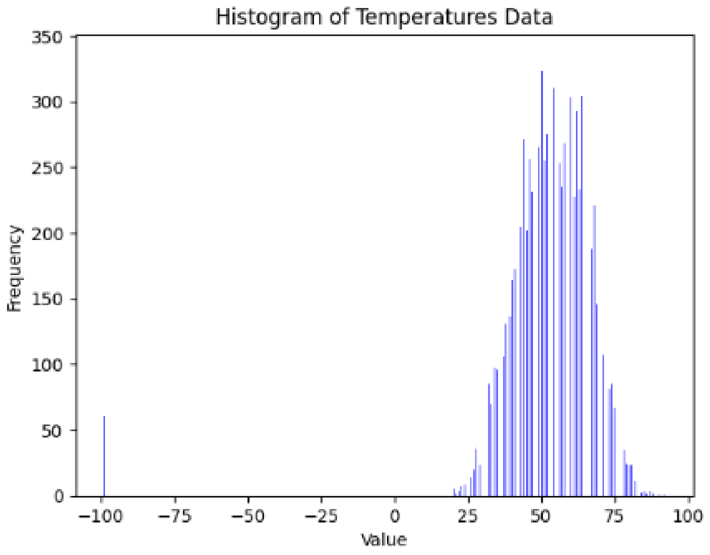


Figure 1: Temperature Distribution

2.1.2 Outliers Classification

Quartiles of a Normal Distribution
Finding outliers using quartiles involves calculating the first and third quartiles, Q1 and Q3, of a dataset, which represent the 25th and 75th percentiles, respectively. The IQR is then computed as the difference between Q3 and Q1. To identify outliers, boundaries are established using the IQR. This method is effective in highlighting values that are significantly different from the rest

[1]`https://www.kaggle.com/datasets/sudalairajkumar/daily-temperature-of-major-cities`

of the data, which could suggest anomalies or measurement errors.
In our context, Q2 represents the median (=54), and the temperature mean value is 52, which explains the normality of our data. The lower outlier threshold which indicates temperatures unusually low is 36. Similarly, the upper outlier threshold which indicates unusually high temperatures is 72.
Figure 3 below illustrates a box plot of Paris Temperature Data and highlights the presence of some outliers.

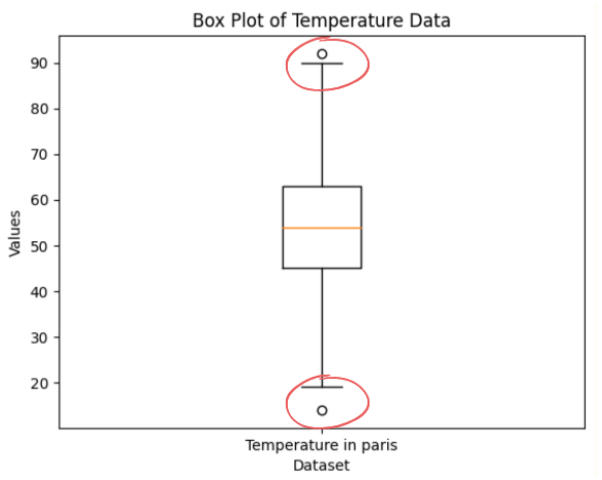


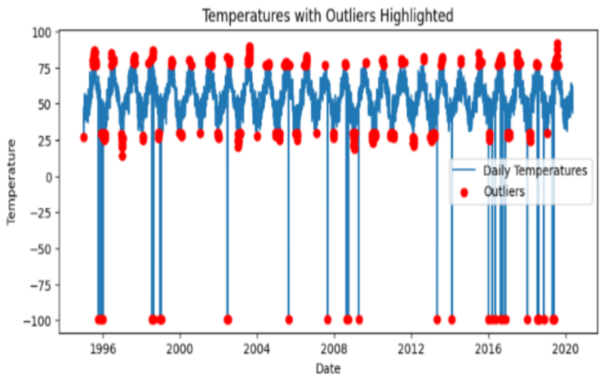
Figure 2: Box plot of Paris Temperature Data

2.2 K-Nearest Neighbors

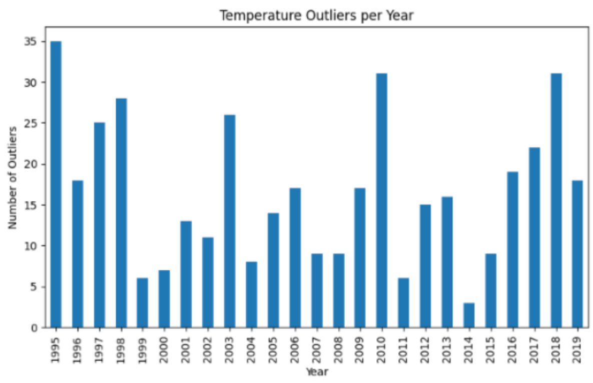
The K-Nearest Neighbors algorithm is a simple, yet effective method for detecting outliers in a dataset, such as temperature measurements. It identifies anomalies by comparing data points to their closest neighbors. Below is a table that breaks down the process into four main steps.

2.2.1 Steps

KNN Steps			
Step 1: Distance Calculation	Step 2: Sorting + 'K' Selection	Step 3: Average Distance Calculation	Step 4: Outlier Detection
The first step in KNN is to calculate the distances: The algorithm tries to measure the distance between each data point and every other point in our dataset. The chosen distance metric was the absolute distance.	Once distances are computed, they are sorted in ascending order. The algorithm then selects the 'k' smallest distances.	Computation of the average of the selected distances. This average tells how close or far a data point is from its nearest neighbors. This calculation helps in determining the similarity between data points.	The final step involves the detection of outliers by comparing the avg distance of each data point to a preset threshold. Data points that show a substantial deviation from this threshold are marked as potential outliers, that also represent extremely high/low temperatures.



(a) Temperature Variations in Paris



(b) Analysis of Temperature Outliers - per year

Figure 3: Figures for KNN algorithm

2.3 Huffman Coding

Huffman coding is widely used for lossless data compression. It constructs a binary tree to assign variable-length codes to data points based on their frequencies, minimizing the number of bits required to represent the data. We hypothesize that this approach is practical for identifying outliers in datasets, as rare data points are assigned longer codes, making them stand out.

2.3.1 Adaptation steps

In our methodology, we initially generate Huffman codes for the temperature dataset based on the occurrence frequency of each temperature all over the dataset. Subsequently, we determine the lengths of the generated codes in terms of the number of bits. We posit that extended Huffman codes indicate rare temperatures, which we consider outliers. The detection of these outliers is contingent on establishing a threshold through an iterative process of trial and error.

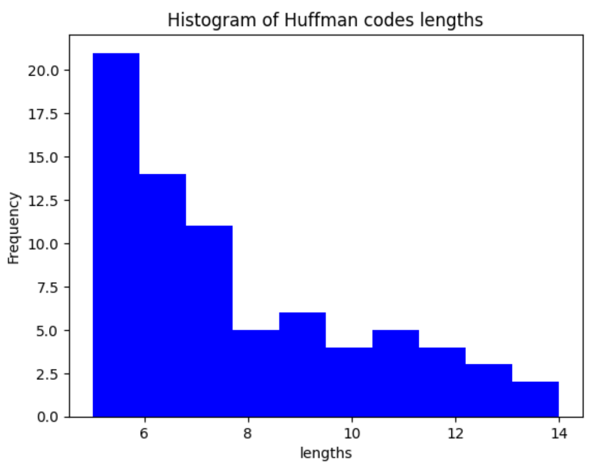
2.3.2 Huffman Code on sliding windows

We integrate a sliding window technique with Huffman coding. This method sequentially processes subsets of consecutive data points. We hypothesize that this enables more precise identification of outliers in localized time frames. Each data window is analyzed for unusual temperature patterns, with outliers determined by the length of their Huffman codes, as detailed in 2.3.1. This approach focuses on detecting variations within specific intervals, improving overall accuracy.

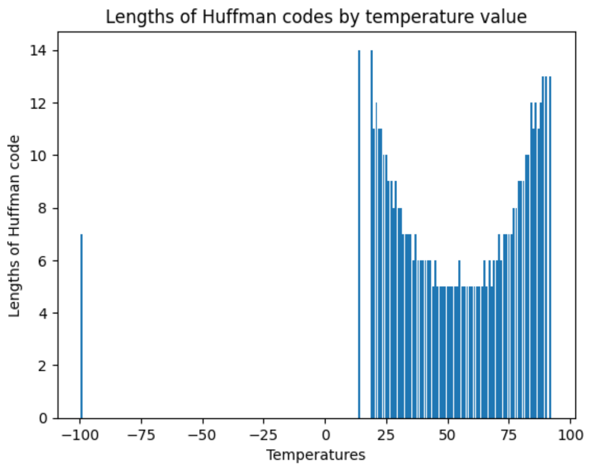
3 Results

3.1 KNN

- **Outlier Detection and Analysis:** The graph below illustrates the fluctuations in temperature data and marks the points that deviate from the expected range. The blue line captures the natural temperature variations in Paris and reflects seasonal changes. In contrast, we can see that the red dots represent extreme temperature values that are unusually higher or lower than usual.



(a) Histogram of codes lengths



(b) Huffman code lengths by temperature value

Figure 4: Analysis of codes lengths

- **Yearly Outlier Distribution:** This bar chart quantifies the outliers on a year-by-year basis. We observe that certain years, such as 1995, 2010, and 2018, show a significant increase in the number of outliers. These peaks suggest years of extreme temperatures than usual. Conversely, years like 1999, 2011, and 2014 are characterized by a relatively low count of outliers, indicating more stable temperature behavior.

3.2 Huffman

- **Huffman Codes length analysis :** According to figure 4a, most Huffman codes (without differentiating by temperature value) are in the 6 to 8 bits range. Fewer codes are beyond this interval. Longer codes are less common, indicating their association with outlier temperature readings. This experiment helped us define the threshold that provides the best capacity for detecting anomalies, which is 8.

- **Lengths of Huffman Codes by Temperature Value Analysis :** Figure 4b shows our hypothesis satisfaction: Huffman coding assigns longer codes to less frequent occurrences, leading to peaks at the extreme temperatures in the histogram, indicating their rarity. Conversely, mid-range temperatures have shorter, uniform code lengths, suggesting they occur more commonly, thus confirming that extreme temperatures could be considered outliers within the dataset.

- **Effect of sliding window size :** By evidence given by figure 5, there is a clear relationship between the sliding window size and the number of detected anomalies. Initially, the number of anomalies increases as the window size increases, suggesting that a small size can result in high false-positive rates. Starting from a window size bigger than 1000, the number of detected anomalies levels off and converges to an acceptable value, suggesting a balance between detecting true outliers and avoiding false positives.

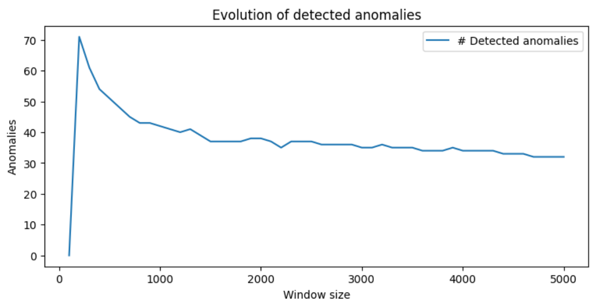


Figure 5: Relationship between window size and number of detected anomalies

4 Discussion

To summarize our findings, we performed a comparative analysis on outlier detection using KNN and Huffman coding techniques against a baseline IQR method that produced 75 unique temperature values and 38 expected outliers. Once we find the best parameters for each method, we observe that KNN identified 29 true outliers without any false positives and detected null values. In contrast, a simple Huffman approach flagged 40 potential outliers but included two false positives. The refined Huffman with sliding window approach pinpointed 36 outliers, matching KNN in avoiding false positives and correctly identifying nulls. KNN shows a conservative approach with no false alarms, while simple Huffman, detects more outliers that may include some false positives.

However, it's important to acknowledge the limitations of the anomaly detection task and the parametric nature of the methods, which may increase their Kolmogorov complexity. To mitigate this, it's recommended to wisely choose a default value of the parameter that minimizes the cost of changing it. In terms of perspectives, the methods have proven their efficiency in detecting anomalies in this type of data, but there are opportunities for improvement, such as integrating the use of adaptive Huffman codes and exploring the "Normalized Compression Distance" for KNN to spot outliers that are missed with the usual methods.

December, 2023

IA703

Algorithmic Information
& Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Yannick LETORT and Pierre BILLAUD

LSystem and Complexity

Abstract

Algorithmic information offers interesting approaches to evaluate complexity and to compare information using grouped compression. On the other hand, L-System is a generative grammar used to model virtual plants from rules and symbols. Our current work tries to find a way to establish a visual proximity between generated plants using complexity theory applied to L-System grammar.

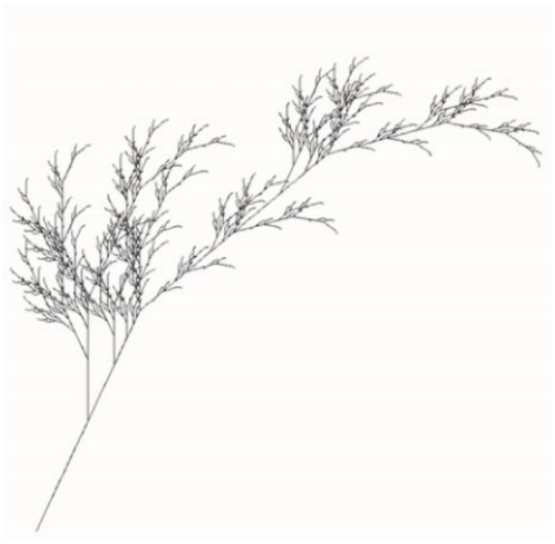


Figure 1 Generated plant using L-System

Problem

The automatic generation of L-System method raises a significant challenge: controlling the quality and relevance of the generated shapes. Traditionally, this task requires human supervision to visually evaluate each result produced by the L-System, a process which can be very time-consuming.

To overcome this challenge, our project focuses on the use of concepts and algorithms related to Kolmogorov complexity and information theory. Our aim is to develop an algorithmic method for evaluating the visual proximity between different plants from L-Systems by using their own generated formula. By quantifying this proximity with a value between [0, 1] (as Normalized Google Distance) we hope to significantly reduce the time needed to evaluate the L-Systems results, thus facilitating the process of selecting and adjusting plant forms in graphic design and computer graphics applications.

This project represents an attempt to apply algorithmic information principles to solve a visual problem.

Method

With the aim of assessing the visual proximity between different plants generated by L-Systems, we have selected four key algorithms, each offering a specific perspective on how generative strings can be compared and analyzed. These algorithms are: co-compression, entropy, Levenshtein distance and a combined algorithm. Here's why and how each was chosen:

1. Co-compression

Co-compression uses data compression techniques to evaluate the similarity between two strings. The underlying idea is that if two strings can be compressed together more efficiently than separately, they probably share common patterns. This method is particularly relevant to L-Systems, as it can detect repetitions and similar structures in the generated strings, which is a key indicator of visual similarity.

2. Entropy

Shannon's entropy, derived from information theory, measures the degree of disorder or in a string of characters. In the context of L-Systems, a high level of entropy could indicate a complex or diverse structure, while a lower entropy could mean a simpler or more uniform structure. Comparing the entropy of generative gives us an idea of their relative complexity.

3. Levenshtein distance

This measure, also known as edit distance, calculates the minimum number of modifications (insertions, deletions or character substitutions) required to transform one string into another. It's a valuable tool for assessing the structural similarity between two strings generated by L-Systems, as it quantifies how close or far are close or far apart in terms of character sequences.

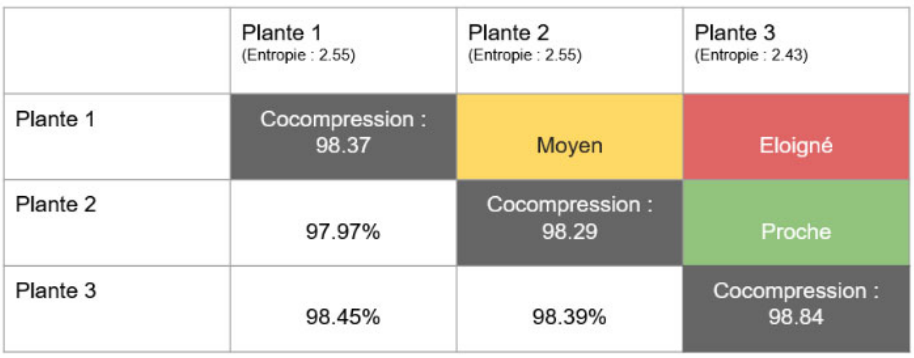
4. Combined algorithm

In addition to these methods, we have also developed a combined algorithm that integrates normalized Levenshtein distance, the difference in visual complexity based on specific symbols (such as '+', '-', '[', ']'), and the co-compression ratio. This algorithm aims to provide a more holistic assessment of similarity, taking into account both the structure and visual complexity. Each of these algorithms has been selected for its ability to provide different but complementary insights into how structures generated by L-Systems can be compared effectively. By using them in conjunction, we hope to develop a robust method for automatically assessing the visual proximity between plant shapes the need for manual supervision in the design process.

Results

Our exploration of different algorithmic methods for assessing the visual proximity structures generated by L-Systems led to a variety of results, each offering perspectives, but also with their own limitations.

Table 1 Results of co-compression and entropy algorithms taken independently



	Plante 1 (Entropie : 2.55)	Plante 2 (Entropie : 2.55)	Plante 3 (Entropie : 2.43)
Plante 1	▨ Cocompression : 98.37	🟨 Moyen	🟥 Eloigné
Plante 2	97.97%	▨ Cocompression : 98.29	🟩 Proche
Plante 3	98.45%	98.39%	▨ Cocompression : 98.84

Red means unsimilar plants | Yellow means medium similarity | Green means close similarity

1. Co-compression algorithm

Although the co-compression algorithm provided interesting clues to the structural similarity between L-System chains, we found that the results were often too close to each other to be conclusive. However, by combining this method with a ratio based on the size of the texts compressed separately and together, we found more useful results but still limited.

2. Levenshtein distance

Levenshtein distance has proved to be a good indicator for pattern recognition in L-System chains. However, it is extremely sensitive to chain length, which can distort results when comparing chains of very different lengths.

3. Entropy

The application of entropy as an individual measure has shown that most L-System chains had similar entropy levels. This limited its usefulness as an visual proximity indicator, as results were almost identical across different L-Systems, reducing its ability to distinguish variations in their complexity or structure.

Table 2 Results of the 3 algorithms combined together

	Plante 1 (Entropie : 2.55)	Plante 2 (Entropie : 2.55)	Plante 3 (Entropie : 2.43)
Plante 1	0.06	🟨	🟥
Plante 2	0.19	0.06	🟩
Plante 3	0.43	0.42	0.06

Red means unsimilar plants | Yellow means medium similarity | Green means close similarity

4. Combined algorithm

The combined algorithm, incorporating normalized Levenshtein distance, differences in visual complexity, and the co-compression ratio, produced more nuanced results. In trying to normalize this algorithm to a distance between 0 and 1, we found that it that it offers a better balance in the assessment of resemblance. Although this method isn't perfect, it did provide a better distinction between the structures, allowing us to adjust the balance between structural density and pattern recognition.

We can also see below the formulas used to find these results. They correspond to our combined algorithms of which we seek to normalize the results in order to obtain a distance between 0 and 1, like Normalized Google Distance (NGD).

Equation 1 Arbitrary formula for determining the visual distance between two plants

$$D(chaine1, chaine2) = \frac{Lev\_Norm(chaine1,chaine2)+Comp\_Diff(chaine1,chaine2)+Cocomp(chaine1,chaine2)}{3}$$

$$Lev\_Norm(chaine1, chaine2) = \frac{Lev(chaine1,chaine2)}{max(len(chaine1),len(chaine2))}$$

$$Comp\_Diff(chaine1, chaine2) = \frac{|Complexite(chaine1)-Complexite(chaine2)|}{max(Complexite(chaine1),Complexite(chaine2))}$$

$$Cocomp(chaine1, chaine2) = \frac{TailleSeparee-TailleCombinee}{TailleSeparee}$$

Discussion

In our exploration to determine visual proximity between plants generated by L-Systems, we have found results that both align with and deviate from our initial expectations. Our theory, while offering insights, remains incomplete. We observed that some algorithms, such as the distance of Levenshtein, value pattern similarity, highlighting structural resemblances in the generated strings. Others, like the co-compression ratio, emphasize length similarity, which indirectly relates to the visual density of the plant. However, this approach is not sufficient to form a robust theory. One key limitation is the integration of diverse values, which do not necessarily share the same nature or scale, leading to a complex and sometimes inconsistent understanding of visual proximity.

On the positive side, our results suggest a promising link between the generated formulas of L-Systems and their visual representations. This correlation, albeit not perfectly defined, opens a window into the understanding of algorithmically generated natural forms.

From another perspective, perhaps focusing on a more specific approach, such as solely on visual pattern similarity, could yield more relevant results. Narrowing our scope might provide a clearer, more consistent framework for analysis.

Ultimately, this exploration proved to be quite interesting, not just in the discovery of new algorithms or the attempt to theorize a formula, but also in how it allows us to consider our own human perception. How do we recognize a plant? What visual clues are most significant to us? These questions, while not fully answered in our project, highlight the intriguing intersection of computational theory and human cognitive perception.

Bibliography

All useful references with links :

L-system (Wikipedia)
https://en.wikipedia.org/wiki/L-system

Levenshtein distance (Wikipedia)
https://en.wikipedia.org/wiki/Levenshtein_distance

Scientific article - Normalized Information Distance :
https://arxiv.org/pdf/0809.2553.pdf

Graphical modeling using L-systems
http://algorithmicbotany.org/papers/abop/abop-ch1.pdf

Data Compression and Archiving
https://docs.python.org/3/library/archiving.html

L-System User Notes
https://paulbourke.net/fractals/lsys/

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Florent BRIAND

Fractals and Complexity in Computer Graphics

Abstract

In the field of computer graphics, fractals can be used to produce 3D models with interesting properties and clear advantages over regular 3D models. We will explore several methods to produce fractals and what is their relationship with the Kolmogorov complexity.

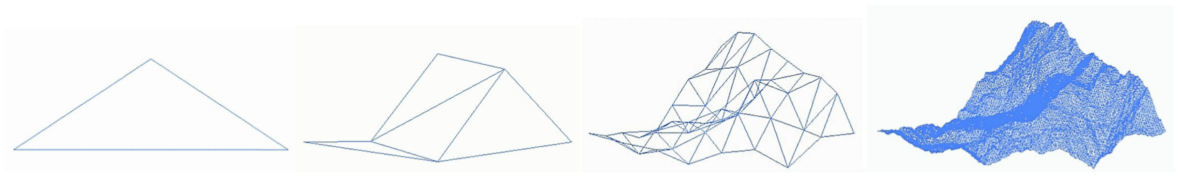
Problem

How can the special properties of fractals be used for computer graphics? We will compare the complexity of fractals algorithms to the resulting apparent complexity by measuring the size of the python files against the size of the images.

Method

Fractals are geometric objects exhibiting similar patterns at different scales. They can be produced by a simple algorithm containing a recursive function.

One of the main use of fractals in computer graphics is the production of terrain and landscapes. Below is an example of a landscape produced from a triangular fractal that replicates itself following a predefined mathematic function. After each iteration (going from left to right), the triangles divide and produce a shape that can be predefined by parameters.



The shape can also be tuned with the introduction of a noise function to break any visible regular pattern and create a more naturally looking scene in this context. Figures below are two different landscapes created by fractals. The models have then been assigned other textures and lighting effects to have a more realistic rendering.

The models generated by fractals can be extremely complex. The level of detail is defined by the number of iterations the algorithm is given. It is then possible to zoom in or come closer to the model without losing quality. This property allows for details to be stored and rendered at all possible scales in the same way. The size of the model can also be extended as required to be.

Fractals can also be very diverse as the result can be greatly modified by tweaking the initial parameters of the mathematical function. Small variations in the initial conditions can generate vastly different results any time the fractal is generated (chaos theory).

A very simple mathematical function can then generate extremely complex results. The program used to generate the model can be very short and its Kolmogorov complexity very low compared to the complexity of a regular function to generate the same model at the same level of details.

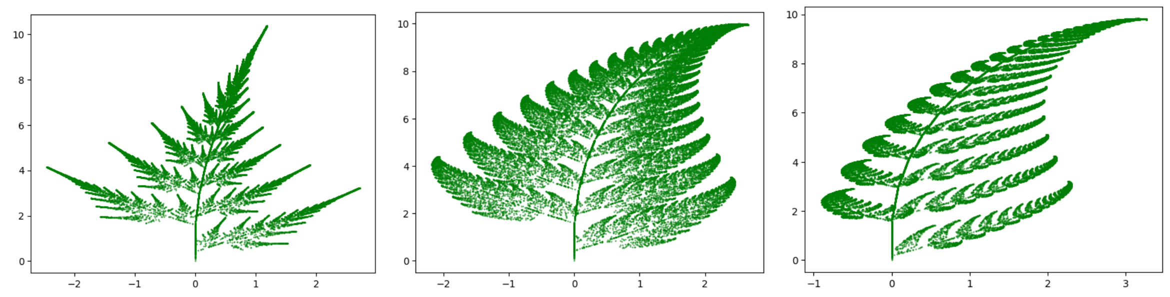
The direct consequence is reduction of required disk space. A short program can replace the full-size model with the benefit of being easily modified. This can be very beneficial for video games for example to render vast and complex landscapes with dense vegetations.

A famous example of fractal used to render vegetation is Barnsley's fern. It uses a recursive function applying four affine transformations. At each iteration, the previous point or pattern is moved in a translation and a rotation defined by the affine equations. This leads to a seemingly infinite repetition pattern that has a striking resemblance with a real fern.

$$f\left(x_{n+1}, y_{n+1}\right) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

Results

The parameters of the affine equations can be modified to make big variations in the size, shape, orientation of the leaf of the fern. Such simple transformations can then generate extremely complex results.



The models can be turned into 3D models with various technics to be used for computer graphics in video games for example. The number of plants and their shape variety in the scenery can be very beneficial to the realism of the video game. The resolution can be improved by increasing the number of iterations in the algorithm.

The above PNG image examples of size 533Ko can then be compressed into a small python file of 2Ko. The complexity can be even improved if only the matrix of coefficients is considered to generate the required shapes on a machine that already has the algorithm saved in memory.

Fractals can then effectively be considered as an extremely efficient compression method.

Discussion

A direct application of the interesting properties of fractal is the fractal compression method for images. The basic concept is identical to classic compression method, but instead of only translating similar looking blocs across the file, the fractal compression method also allows for rotations of the blocs.

This further increase the number of possibilities a bloc can replicate across the image, then drastically improves the compression ratio. At a similar level of quality, the fractal compression method leads to a smaller size. On the other hand, the encoding part is more time consuming as the number of possibilities to check for additional rotations is more important, making it unsuitable for real time video transmission. This method is then useful for specific cases of compression requirements such as disk storage or file downloads.

An interesting work to do would be to compare the compression ratio of fractal compression and regular compression versus the computation time required.

Bibliography

Barnsley, M. F. (1988), “Fractals Everywhere,” Academic Press, San Diego.

Barnsley, M. F., and Sloan, A. D. (1988), A better way to compress images, Byte Mag., 215-223.

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Pierre de La Ville Montbazon

Attempting to build a music genre tree

Abstract

In the article "Language Trees and Zipping" it was demonstrated that you can use compression of string sequences as a proxy for complexity. Then using the chain rule, you can determine a distance between two texts which can be used for authorship attribution and classification. When talking about music, we understand that a song is a complex object, however it tends to have recognizable patterns wether in melody or the types of instruments used or even the themes of the lyrics when they exist. This micro-study will attempt to apply a method similar to the previously mentioned study in order to classify music. It will test two methods, one compressing audio files, and one compressing a text sequence based on the notes contained in each songs.

Problem

Is it possible to efficiently measure distance between two songs using a similarly simple and elegant process ? Our goal with this study is to attempt to build a simple and easily replicable method for song classification and authorship attribution.

Method

0. Requirements

In order to execute the various tasks in this project, the following items were necessary.

- Python packages
  - pytube
  - music21
  - librosa
  - scipy
  - zlib
  - pydub
- ffmpeg
- wav2midi2wav project (should be used in a separate python environment using the 3.7.4 version)
- vamp plugin necessary to use the previous item

1. Data

The first step was to quickly build a small dataset that could be used to test our two methods. As a format for each audio file, I chose .wav format as it is uncompressed. In order to quickly obtain songs, I built four public youtube playlists (jazz,baroque,romantic,impressionist), which I then downloaded using the pytube package on python. Unfortunately, since the videos were downloaded from youtube, their format was mp4 (which is compressed) and had to be converted to .wav using ffmpeg. This means that despite using a .wav format, the songs have already gone through compression which will most likely influence the results of the experiment. Note that the links to the mentioned playlists are available in the notebooks sent with this report.

2. Detailing the two methods

In order to test the efficieny of our methods, we build a tree based on the compression distance between each file. In the first method, we build the distance matrix by simply compressing each pair of files and the file obtained by combining the two, we then obtain the smart distance between each file. For the second method, we use the opensource project "wav2midi2wav" in order to convert each song into the midi format. We then extract the note sequence of each song as a string, which we then compress similarly to the first method in order to get a distance. The goal of this method is to ignore some other factors like rhythm and instruments used, in order to focus on the content of the song, rather than its computer representation.

Results

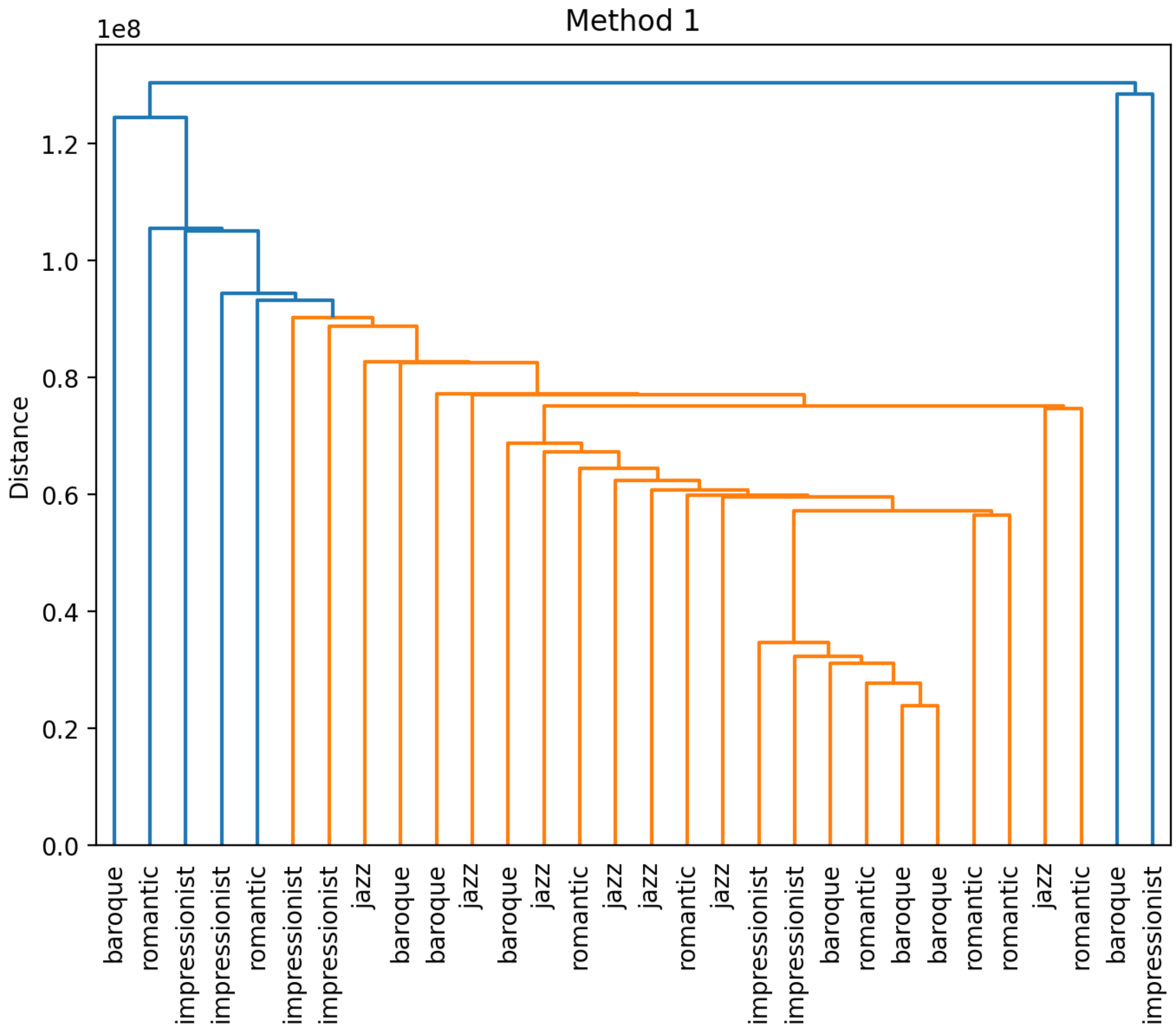


Figure 1: Method 1 tree

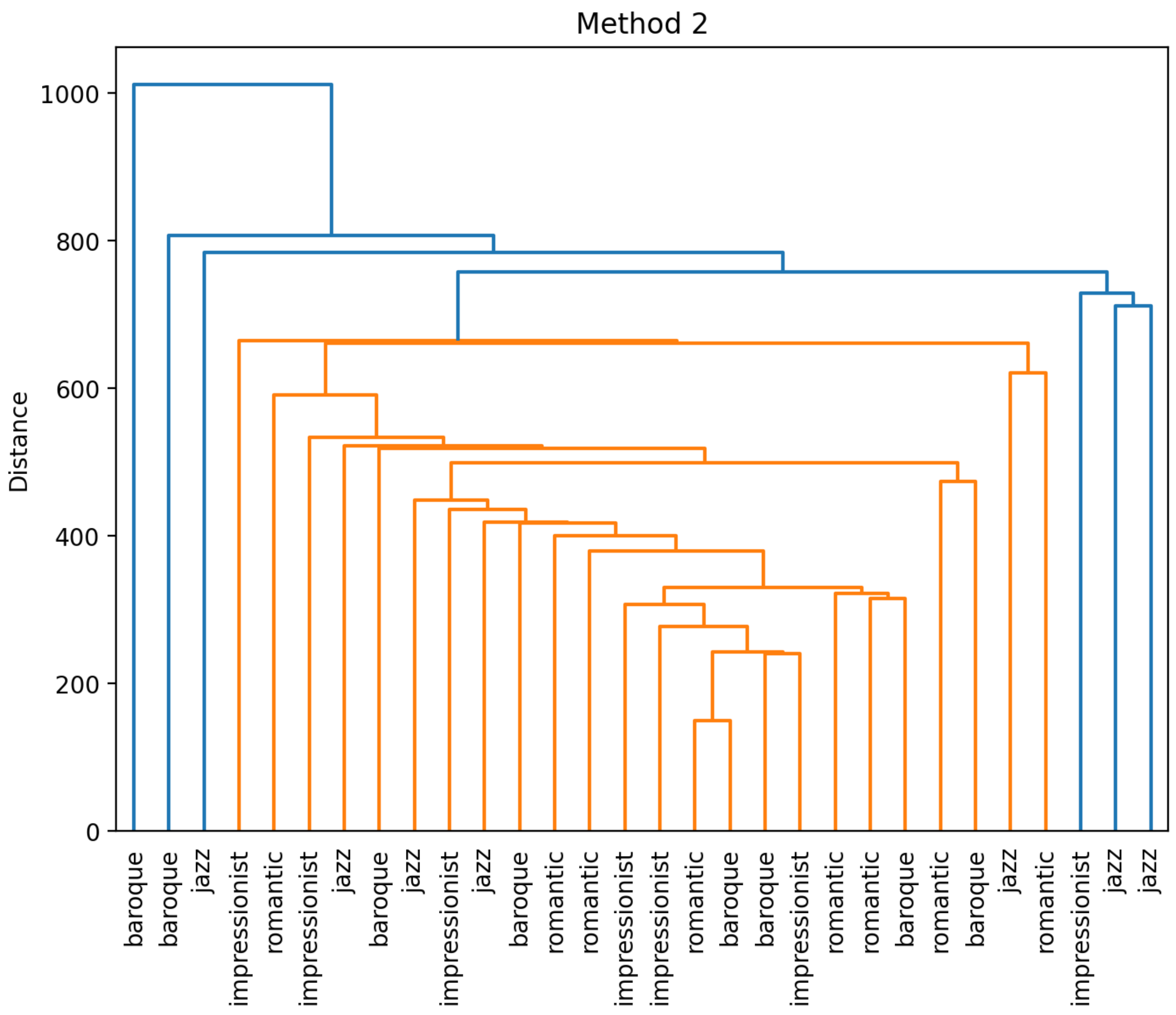


Figure 2: Method 2 tree

As we can see, neither of the two methods gives conclusive results, musics from two supposedly different genres are seen as more similar than two musics from the same genre. There are several ways to explain these results. Firstly, as mentioned in the previous section, the audio files used here had most likely already been compressed to some degree. Then the method used to convert the audio to .mid files although advanced remains imperfect, some notes are incorrect and for jazz songs in particular, a lot of instruments end up out of the final result. The following section will delve into possible improvements.

Discussion

In light of these considerations, I make the following recommendations for a possible second attempt :

- The dataset needs to be more varied with more musical genres
- The songs need to be in a lossless format from the start
- Each song should have a corresponding .mid file available, or a better method to convert .wav to .mid should be researched
- Finally some research into music theory or the consultation of an expert could be useful in order to better understand how our efforts should be focused.

Although this effort did not yield success, I believe that given enough time, a working method could be developed. However it might not be as simple and elegant as the method used in the article that inspired this micro-study. Where the method applied to text simply needed to use a character sequence, which is easy to obtain or even produce. Audio on the other hand, is more complicated from a computational point of view and requires treatment, which might defeat the purpose of using such a method.

Conclusion

To conclude this microstudy, the methods studied here have not shown to be efficient, that being said, it is clear that there is room for improvement in several areas like data preparation, which is why I remain optimistic about the potential of compression complexity as a tool for song analysis.

Bibliography

[1] Language Trees and Zipping : https://arxiv.org/pdf/cond-mat/0108530.pdf

[2] wav2midi2wav : https://github.com/gcunhase/wav2midi2wav

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Edouard DUCLOY

Prime numbers to reduce complexity

Abstract

Several approaches exist in order to reduce the complexity of integers. One of the most famous is the round number complexity, consisting in using round numbers as proxy to reduce its complexity and the ones of its neighbors.
The purpose of the present study is to use prime numbers, and check if it can be used to reduce as well the complexity of integers.
The method described hereafter proposes different ways to use prime numbers for complexity reduction purposes, either with prime decomposition of integers, or by using prime numbers as proxy. And the results obtained are worthwhile as they allow to reduce the complexity of more than 40% of the integers between 0 and 10,000.

Problem

Binary sequences of integers can be very long, which increases considerably the complexity of its information. And more complex information needs more computing resources to process it, which can be very inconvenient when dealing with big amount of data.
In the big data era we live in now, the need to compress data and reduce its complexity is then essential to meet today's challenges. This project aims at proposing solutions to address this issue.

Method

The method presented in this section uses the CompactIntegerCoding.py python code, from J.L. Dessalles course about description complexity of integers. CompactCoding0 and CompactDecoding0 functions are used to return binary coding of integers. And CompactCoding and RoundReferenceCoding are used to assess the performance of our new method.

In order to manipulate primes number, we first need to create some useful functions:

- **primes(N) :** a function which returns all prime numbers from 0 to N, and the next prime number superior to N.

```
>>> primes(8)
[2, 3, 5, 7, 11]
```

- **is_prime_number(N) :** a function which checks if N is a prime number.

```
>>> for i in range(31):
>>>     if is_prime_number(i) == True:
>>>         print(i, is_prime_number(i))
2 True
3 True
5 True
7 True
11 True
13 True
17 True
19 True
23 True
29 True
```

- **first_primes(N) :** a function which returns a list of the N first prime numbers.

```
>>> first_primes(15)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

- **prime_decompose(N) :** a function which performs prime decomposition of N, and returns a list of tuples with prime numbers and associated powers.

```
>>> prime_decompose(399)
[(3, 1), (7, 1), (19, 1)]
```

Then, we propose three different manners to use prime numbers to describe integers, and write the code of their functions. It is to be noted that, in order to enable '0' and '1' coding, the value 2 is summed to each integer. It is also to be noted that, in order to differentiate the different approaches presented below, a two bits heading is added in each case.

- **PrimeDecomposeCoding(N) :** The first approach uses prime decomposition of the integer. The principle is to use CompactCoding0 to code the powers of each prime number from 2 to the highest involved in the decomposition. Even the prime numbers which do not belong to the decomposition shall be coded with a '0' as long as they are lower than the highest one. If not, the function would return a same code for different integers, which would lead to inappropriate results.

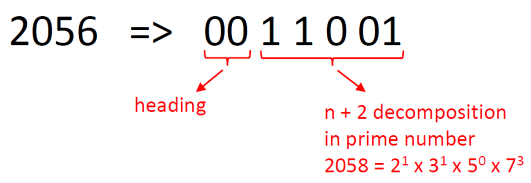
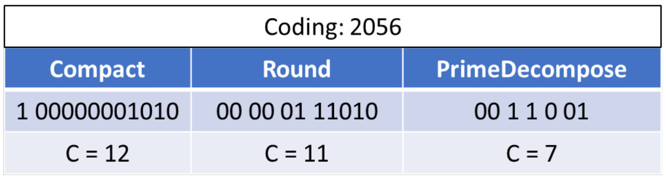


Figure 1: PrimeDecomposeCoding principle

The first ’00’ is the heading corresponding to PrimeDecomposeCoding approach.
The next sequences correspond to the prime powers descriptions of the prime decomposition.

```python
def PrimeDecomposeCoding(N):
    '''
    Code a number N by using prime number decomposition
    Each prime number with 0 power is coded 0
    '''
    N = N + 2
    list_primes = primes(N)
    factors = prime_decompose(N)
    code = ''
    k = 0

    for i,j in (factors):
        while list_primes[k] != i:
            code += '␣0'
            k += 1
        code += '␣' + CompactCoding0(j)
        k += 1

    return '00' + code
```

```
    >>> N = 2056
    >>> print ('Decomposition:␣', prime_decompose(N+2))
    Decomposition:  [(2, 1), (3, 1), (7, 3)]
    >>> print ('Primes␣list:␣', primes(prime_decompose(N+2)[-1][0]))
    Primes list:  [2, 3, 5, 7, 11]
    >>> print ('Compact:␣', CompactCoding(N), '=>', complexity(
        CompactCoding(N)))
    Compact:  1 00000001010 => 12
    >>> print ('Round:␣', RoundReferenceCoding(N), '=>', complexity(
        RoundReferenceCoding(N)))
    Round:  00 00 01 11010 => 11
    >>> print ('PrimeDecompose:␣', PrimeDecomposeCoding(N), '=>',
        complexity(PrimeDecomposeCoding(N)
    PrimeDecompose:  00 1 1 0 01 => 7
```



Coding: 2056		
Compact	Round	PrimeDecompose
1 00000001010	00 00 01 11010	00 1 1 0 01
C = 12	C = 11	C = 7

Figure 2: PrimeDecomposeCoding complexity comparison with CompactCoding and RoundReferenceCoding

We notice in this case that PrimeDecomposeCoding approach allows a significant complexity reduction, with 4 bits less than with RoundReferenceCoding. Nevertheless, the constraint to code each intermediate 0 power can lead to very long sequences, as shown in the example below.

```
    >>> N = 197
    >>> print ('Decomposition:␣', prime_decompose(N+2))
    Decomposition:  [(199, 1)]
    >>> print ('Primes␣list:␣', primes(prime_decompose(N+2)[-1][0]))
    Primes list:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
        53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
       127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
       193, 197, 199, 211]
    >>> print(PrimeDecomposeCoding(N), '=>', complexity(
       PrimeDecomposeCoding(N)))
    PrimeDecompose:  00 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 => 48
```

- **PrimeSkipCoding(N) :** The second approach uses prime decomposition as well, but differs in the way to code the intermediate 0 power. Instead of adding as many '0' as prime number with 0 power, the principle here is to add an extra '0' bit to indicate that the following sequence describes the quantity of prime numbers to skip in the decomposition. This approach can be very useful in case of several consecutive 0 power.

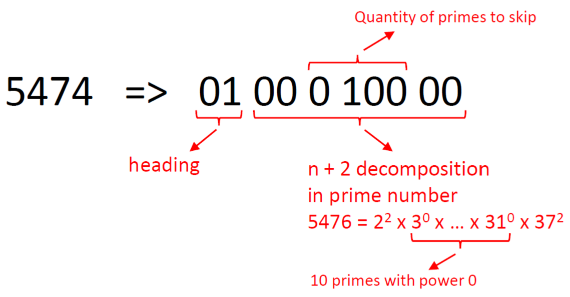


Figure 3: PrimeSkipCoding principle

The first '01' is the heading corresponding to PrimeSkipCoding approach.
The '0' in the middle of the sequence indicates that the following '100' corresponds to the quantity 10 of intermediate prime numbers to skip in the decomposition (from 3 to 31).

```
def PrimeSkipCoding(N):
    '''
    Code a number N by using prime number decomposition
    Quantity of prime number with 0 power in a raw is coded with a heading
        0
    '''
    N = N + 2
    list_primes = primes(N)
    factors = prime_decompose(N)
    code = ''
    k = 0

    for i,j in (factors):
        m = 0
        while list_primes[k] != i:
            k += 1
            m += 1
        if m == 0:
            code += '␣' + CompactCoding0(j)
        else:
            code += '␣0␣' + CompactCoding0(m) + "␣" + CompactCoding0(j)
        k += 1

    return '01' + code
```

```
>>> N = 5474
>>> print ('Decomposition:␣', prime_decompose(N+2))
Decomposition:  [(2, 2), (37, 2)]
>>> print ('Primes␣list:␣', primes(prime_decompose(N+2)[-1][0]))
Primes list:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
>>> print ('Compact:␣', CompactCoding(N), '=>', complexity(
   CompactCoding(N)))
Compact:  1 010101100100 => 13
>>> print ('Round:␣', RoundReferenceCoding(N), '=>', complexity(
   RoundReferenceCoding(N)))
Round:  01 11001 00 1100 => 13
>>> print ('PrimeSkip:␣', PrimeSkipCoding(N), '=>', complexity(
   PrimeSkipCoding(N)
PrimeSkip:  01 00 0 100 00 => 10
```

Coding: 5474		
Compact	**Round**	**PrimeSkip**
1 010101100100	01 11001 00 1100	01 00 0 100 00
C = 13	C = 13	C = 10

Figure 4: PrimeSkipCoding complexity comparison with CompactCoding and RoundReferenceCoding

- **PrimeProxyCoding(N) :** The third approach consists in using prime numbers as proxy to describe integers. To do so, the integer is described as the position of its closest prime number in the prime numbers lists and the distance which separates them.

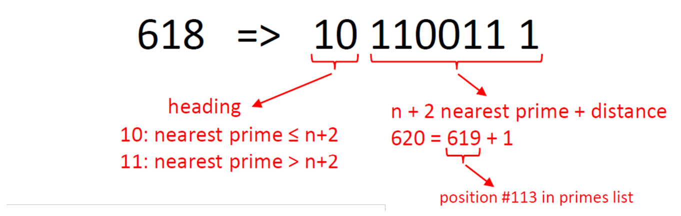


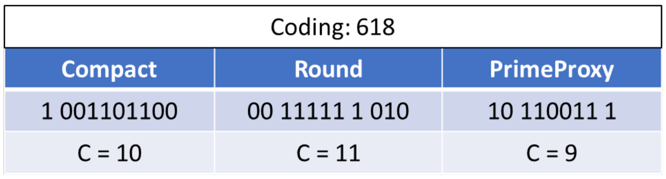
Figure 5: PrimeProxyCoding principle

The first '10' is the heading corresponding to PrimeProxyCoding approach, when the closest prime number is inferior to the integer. if the prime number is superior to the integer, the heading is '11'.
We notice that the description of the integer is composed of three binary sequences, the heading '10', the closest prime number description '110011' (corresponding to the position 113 of the prime number 619 in the prime numbers list) , and the distance to the closest

prime number '1'. It will be always the case, except when the closest prime is the integer itself in which case the distance description is omitted.

```
def PrimeProxyCoding(N):
    '''
    Code a number N by using prime number as proxy
    '''
    N = N + 2
    list_primes = primes(N)
    indice = np.argmin(np.abs(np.array(list_primes)-N))

    if indice == len(list_primes) - 2 and CompactCoding0(N-list_primes[
       indice]) == '0':
        return '10␣' + CompactCoding0(indice)
    elif indice == len(list_primes) - 2 and CompactCoding0(N-list_primes[
       indice]) != '0':
        return '10␣' + CompactCoding0(indice) + '␣' + CompactCoding0(N-
           list_primes[indice])
    elif indice == len(list_primes) - 1:
        return '11␣' + CompactCoding0(indice) + '␣' + CompactCoding0(
           list_primes[indice]-N)
```

```
    >>> N = 618
    >>> print ('Primes␣list:␣', primes(N+2)[-2:])
    Primes list:  [619, 631]
    >>> print ('Compact:␣', CompactCoding(N), '=>', complexity(
       CompactCoding(N)))
    Compact:  1 001101100 => 10
    >>> print ('Round:␣', RoundReferenceCoding(N), '=>', complexity(
       RoundReferenceCoding(N)))
    Round:  00 11111 1 010 => 11
    >>> print ('PrimeProxy:␣', PrimeProxyCoding(N), '=>', complexity(
       PrimeProxyCoding(N)
    PrimeProxy:  10 110011 1 => 9
```



Coding: 618		
Compact	Round	PrimeProxy
1 001101100	00 11111 1 010	10 110011 1
C = 10	C = 11	C = 9

Figure 6: PrimeProxyCoding complexity comparison with CompactCoding and RoundReference-Coding

Finally, in order to maximize the complexity reduction, we create a last function that takes the best of all three approaches, so the one who returns the shortest binary sequence. This function is called PrimeCoding.

```
def PrimeCoding(N):
    '''
    return the shortest binary sequence between PrimeDecomposeCoding,
       PrimeSkipCoding, and PrimeProxyCoding
    '''
    PDC = PrimeDecomposeCoding(N)
    PSC = PrimeSkipCoding(N)
    PPC = PrimeProxyCoding(N)
    if min(complexity(PDC), complexity(PSC), complexity(PPC)) == complexity(PDC
        ):
        return PDC
    if min(complexity(PDC), complexity(PSC), complexity(PPC)) == complexity(PSC
        ):
        return PSC
    if min(complexity(PDC), complexity(PSC), complexity(PPC)) == complexity(PPC
        ):
        return PPC
```

We also create the PrimeDecoding function which decodes binary sequences obtained by the PrimeCoding approach. This allows to check the bijectivity of PrimeCoding.

```
def PrimeDecoding(code):
    '''
    Decode a number from its prime number coding
    '''
    exponents = code.strip().split()

    if exponents[0] == '00':
        exponents = exponents[1:]
        exponents = [CompactDecoding0(exp) for exp in exponents]
        primes = first_primes(len(exponents))
        N = 1
        for i, exp in enumerate(exponents):
            N *= primes[i] ** exp

    elif exponents[0] == '01':
        exponents = exponents[1:]
        indice = 0
        N = 1
        for i in range(len(exponents)):
            k = 0
            if exponents[i] == '0':
                k = CompactDecoding0(exponents[i+1])
                indice += k
            elif exponents[i-1] == '0':
                pass
            else:
                N *= first_primes(indice+1)[indice] ** CompactDecoding0(
                    exponents[i])
                indice += 1

    elif exponents[0] == '10':
        exponents = exponents[1:]
        if len(exponents) == 1:
            N = first_primes(CompactDecoding0(exponents[0])+1)[-1]
        else:
            N = first_primes(CompactDecoding0(exponents[0])+1)[-1] +
               CompactDecoding0(exponents[1])

    elif exponents[0] == '11':
        exponents = exponents[1:]
        N = first_primes(CompactDecoding0(exponents[0])+1)[-1] -
           CompactDecoding0(exponents[1])

    return N - 2
```

Results

The figure below shows the complexity of integers described by each of the prime numbers approach presented above. It also allows the comparison with already known CompactCoding and RoundReferenceCoding methods.

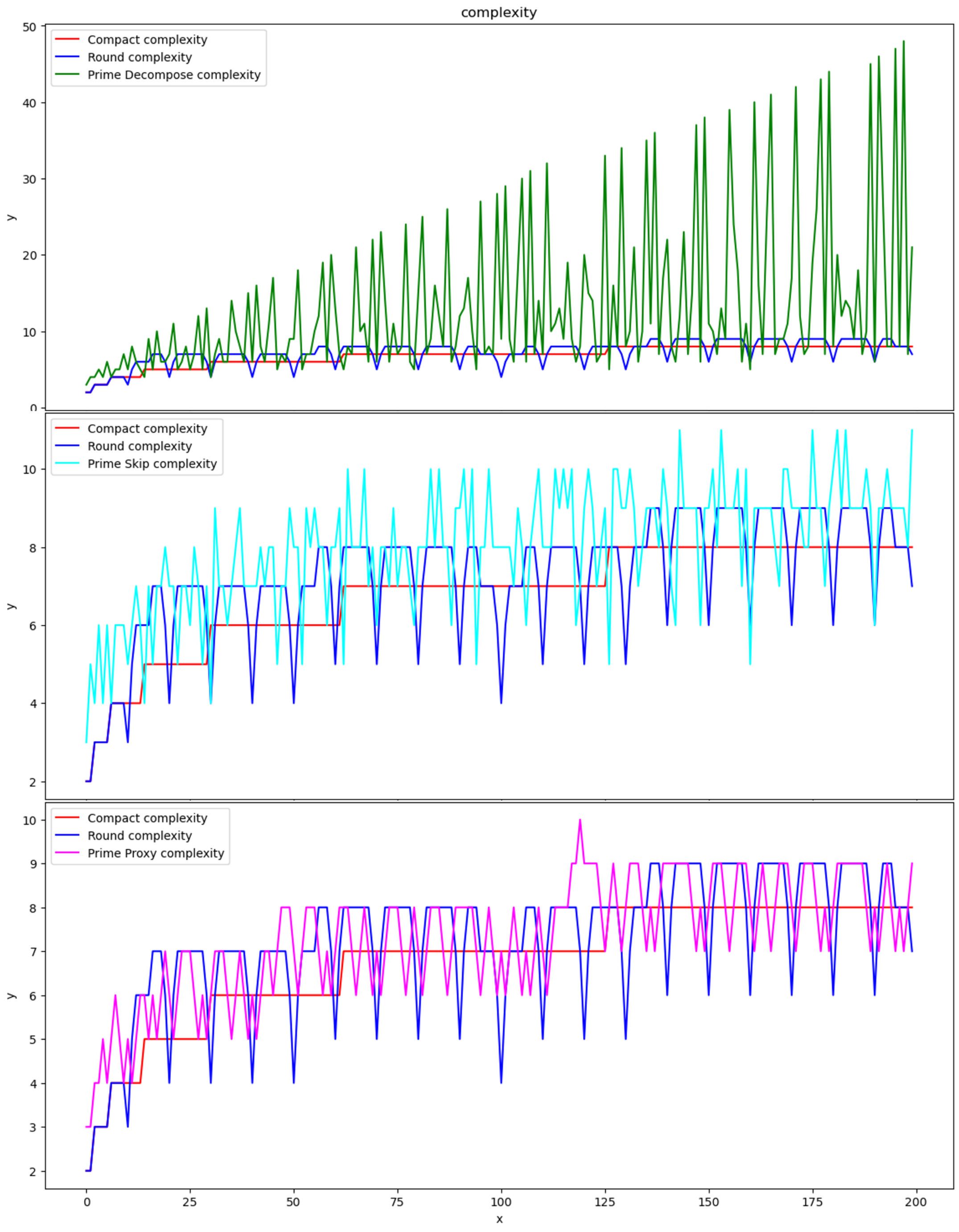


Figure 7: Complexity with the three different approaches

We can notice that each method allows to reduce complexity in some cases. For the PrimeDecomposeCoding approach, the drawback of adding an extra '0' for each intermediate prime number is here obvious.

The figure below shows the integers complexity obtained through the PrimeCoding approach, in comparison with the CompactCoding and RoundReferenceCoding ones.

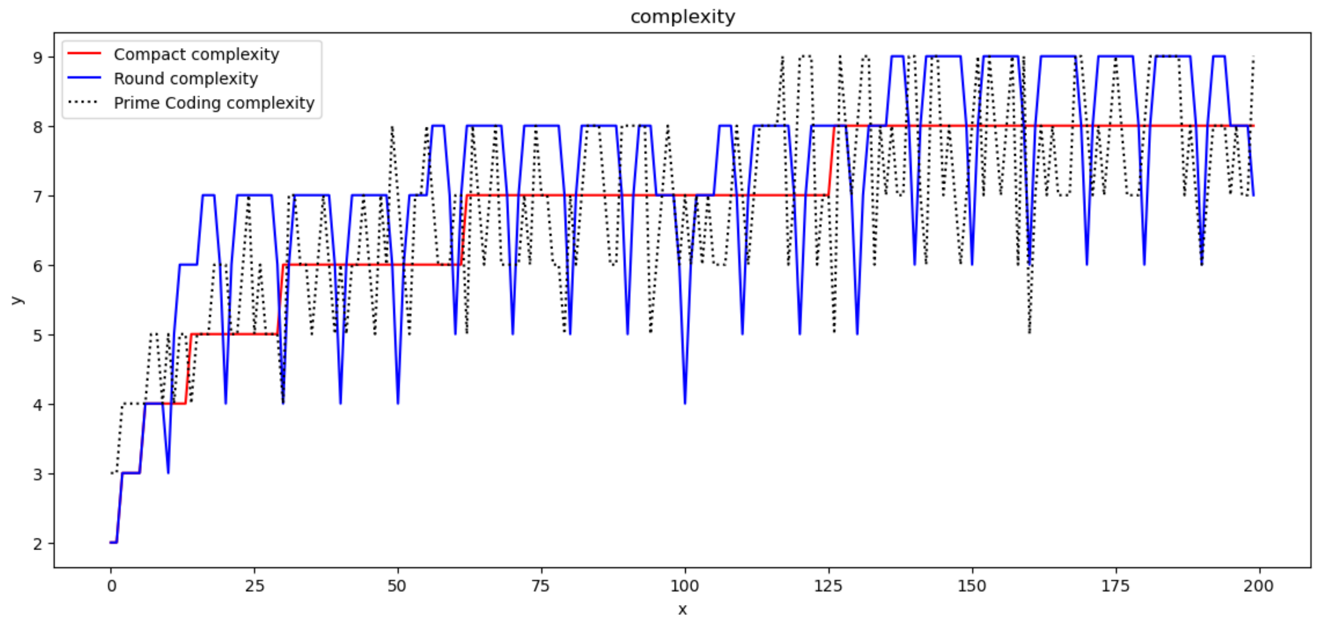
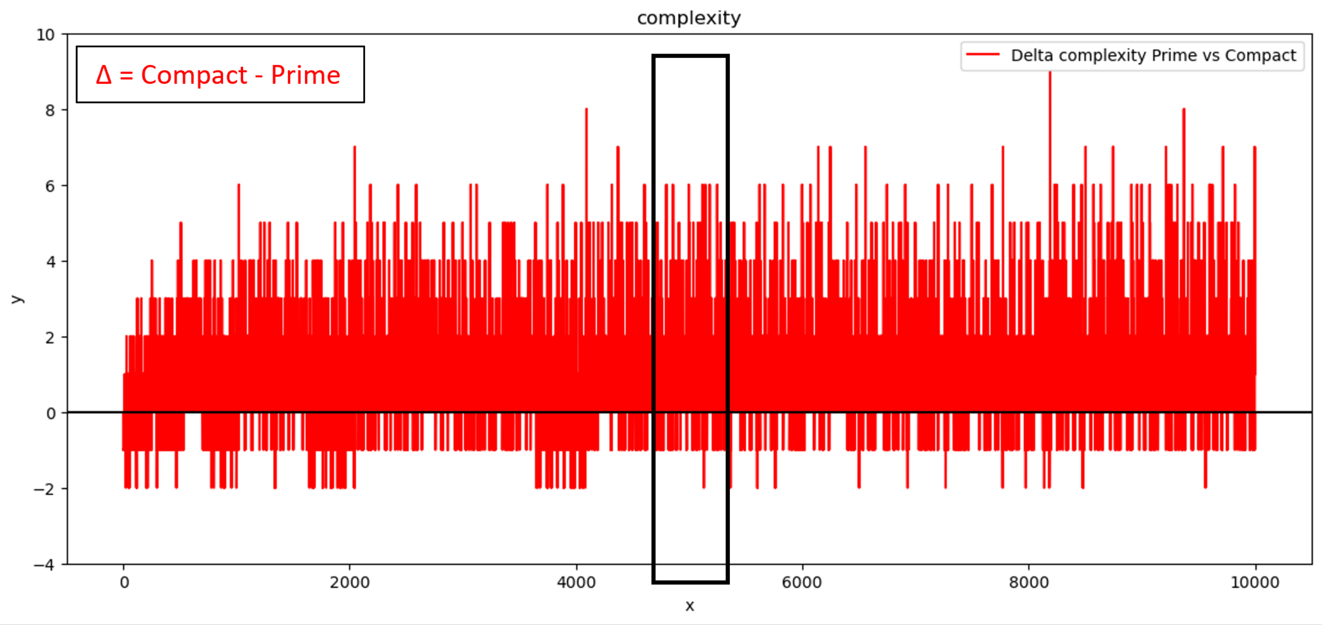


Figure 8: Complexity with PrimeCoding approach

We notice that this approach allows to reduce the integer complexity in several occurrences.

The figure below shows the complexity delta between PrimeCoding and CompactCoding, over the first 10,000 integers. The delta corresponds to the Compact complexity minus the Prime complexity. So a positive delta means that the prime approach allows a complexity reduction.



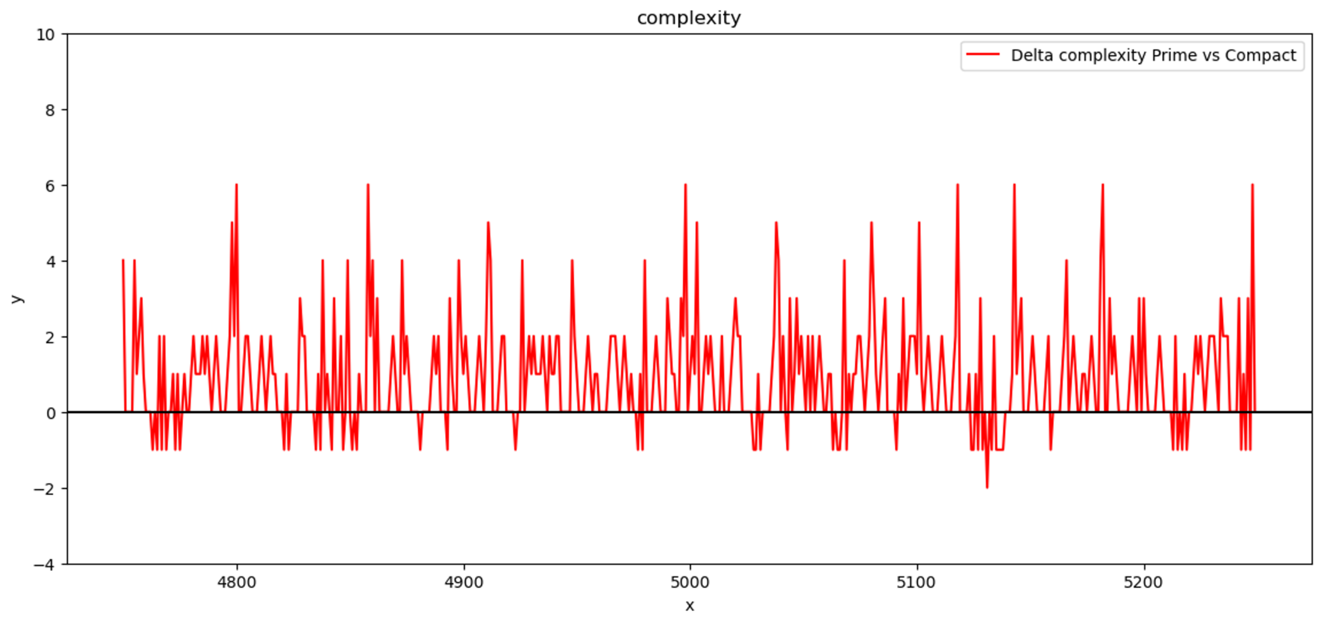
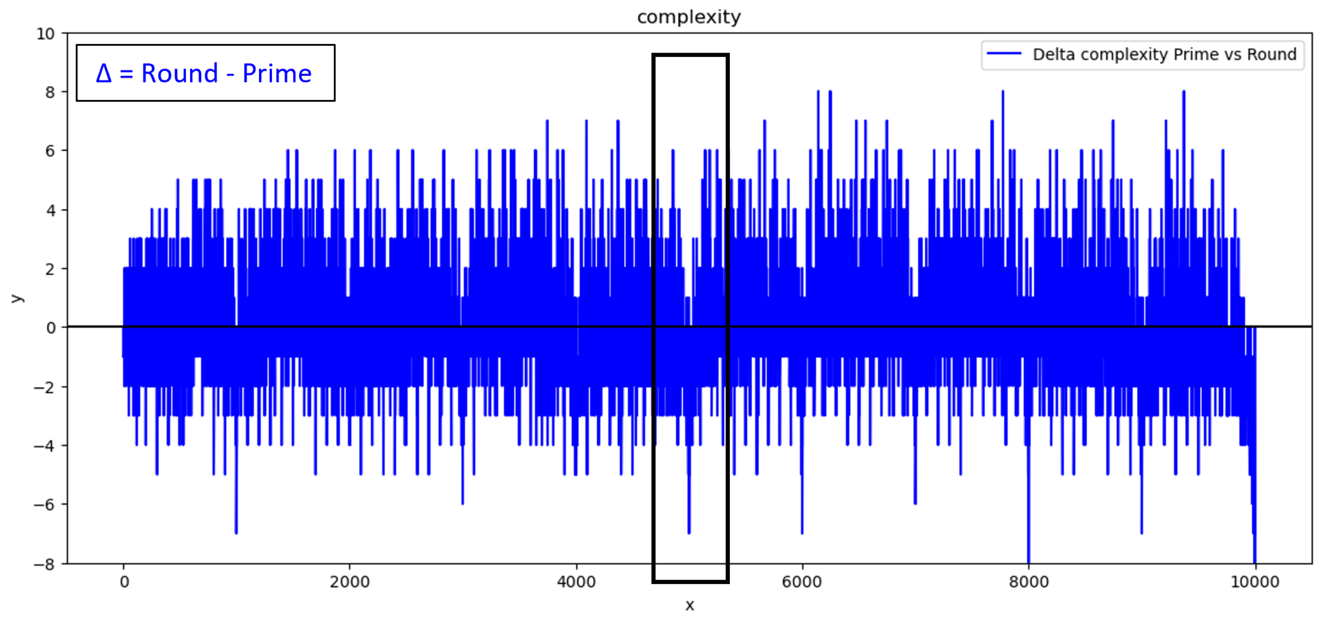


Figure 9: Complexity delta between PrimeCoding and CompactCoding

Over 10,000 integers, PrimeCoding allows a complexity reduction for 4,631 integers (46%), keeps the complexity unchanged for 3,849 integers (38%), and increases the complexity for 1,520 integers (15%). The mean reduction is 0.68.

The figure below shows the complexity delta between PrimeCoding and RoundReferenceCoding, over the first 10,000 integers. The delta corresponds to the Round complexity minus the Prime complexity. So a positive delta means that the prime approach allows a complexity reduction.



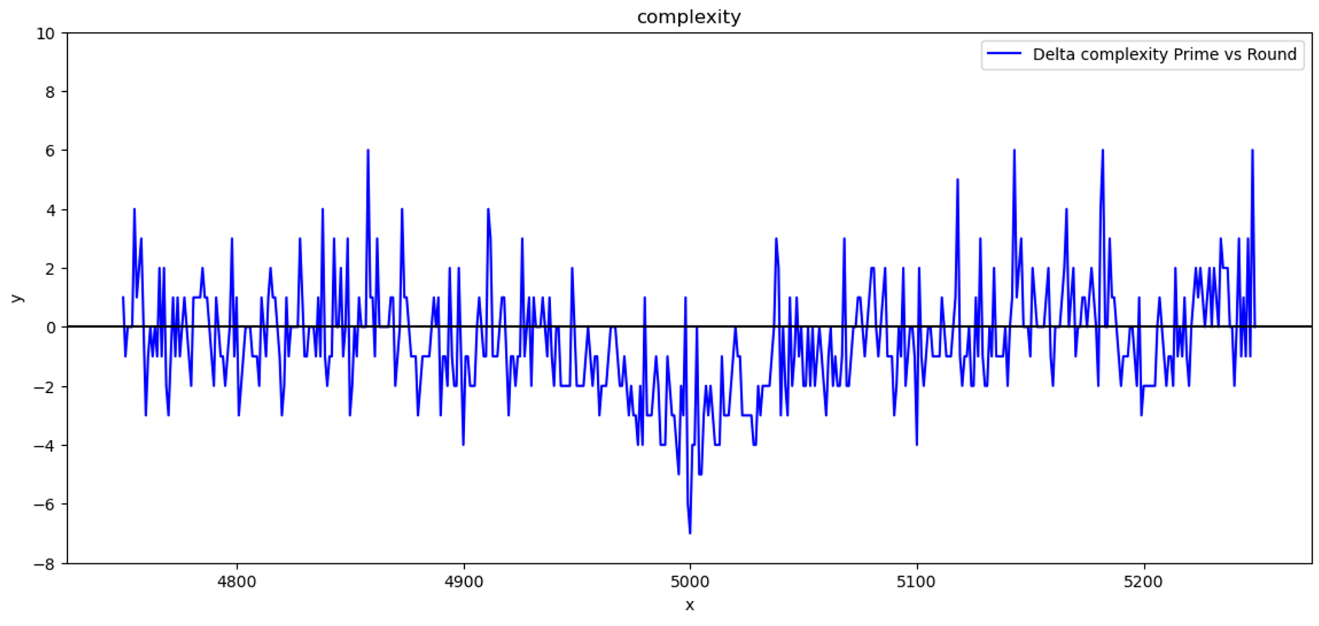


Figure 10: Complexity delta between PrimeCoding and RoundReferenceCoding

Over 10,000 integers, PrimeCoding allows a complexity reduction for 4,117 integers (41%), keeps the complexity unchanged for 2,281 integers (23%), and increases the complexity for 3,602 integers (36%). The mean reduction is 0.12.

Discussion

The results obtained through the use of prime numbers are worthwhile, as they allow complexity reduction in many cases. Nevertheless, the observations we could make do not allow to generalize about the performance of such method, because prime numbers are not uniformly distributed.
The RoundReferenceCoding seems to be more appropriate in more cases as the complexity drop is more significant for integers that seem more relevant (round numbers are more frequently used than other numbers).
This study showed that prime numbers can be good actors in the information compression. But, their usefulness shall be assessed depending on cases we have to deal with.

December, 2023

IA703

Algorithmic Information
& Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Marc EDOUARD

Kolmogorov complexity and entropy applied to time series

Abstract

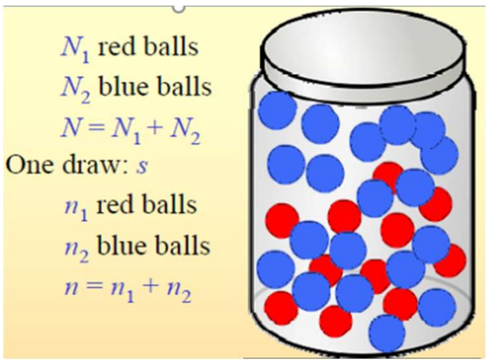
This micro study focus on how Kolmogorov complexity and entropy relate together, and how could they serve to monitor time series data.

Problem

Is metric from information theory could help to analyze time series and serve as process monitoring?

Method

First step will be to consider simple example and see how complexity and entropy relates using both theorical and empirical approach. In the case-study, we consider an urn that has a fixed number of blue / red balls (respectively N1 and N2). We then study Kolmogorov complexity and entropy metrics relative to a sampling of n balls, drawing n1 red balls and n2 blue balls. We can define surprise as difference between expected complexity and actual complexity:



$$U(s) = Cw - C = -n[\frac{n1}{n}\log(p1) + \frac{n2}{n}\log(p2)]$$

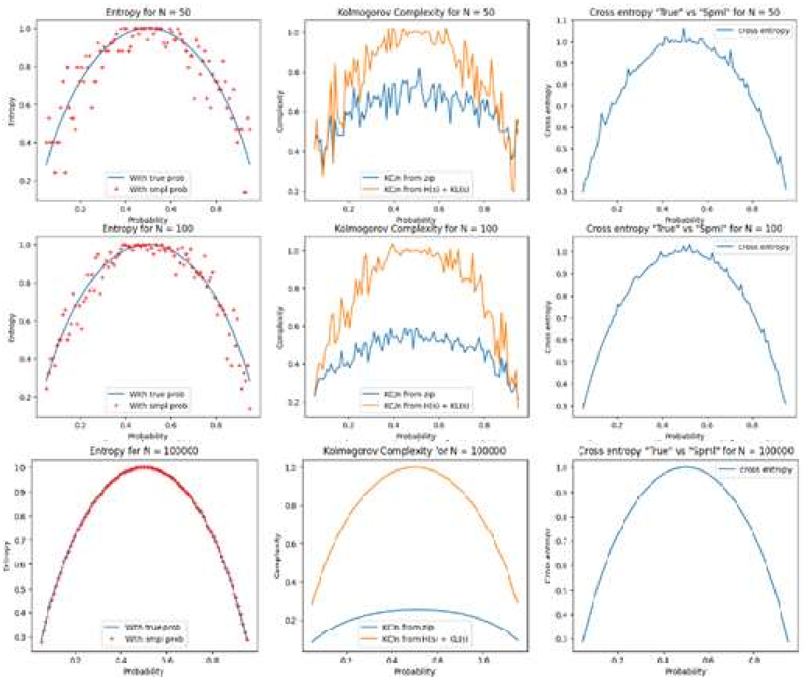
We can then review some usefull definition:

- Entropy, quantifying distribution information $H(P) = -\sum P(x) \cdot \log(P(x))$
- KL divergence, quantifying information lost when approximating one distribution by another $D_{KL}(P \parallel Q) = \sum P(x) \cdot \log\left(\frac{P(x)}{Q(x)}\right)$
- Cross-entropy, relative to encoding events from one distribution using optimal code from another distribution $H(P,Q) = H(P) + D_{KL}(P \parallel Q)$

Using sampling frequence for P and population frequence for Q, we can compute:

$$H(s) - \frac{U(s)}{n} = \sum \frac{n_i}{n}(log(p_i) - log(\frac{n_i}{n})) = -\sum \frac{n_i}{n}(log(\frac{\frac{n_i}{n}}{p_i})) = -KL(s) \implies \frac{U(s)}{n} = H(s) + KL(s)$$

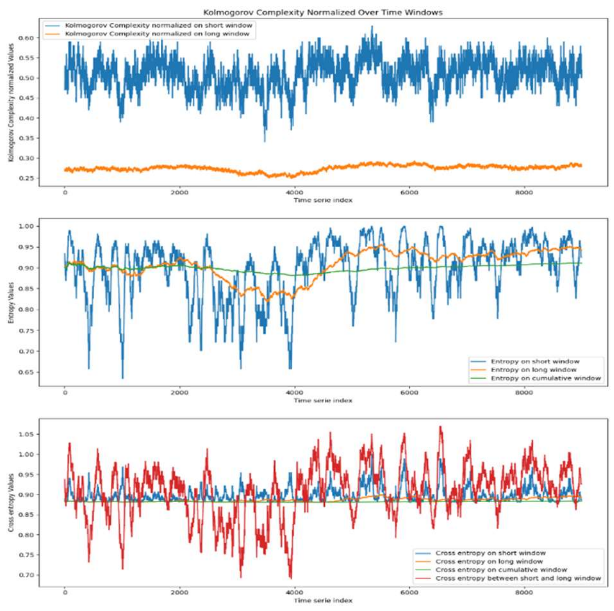
It implies that mean complexity per symbol is relative to cross entropy and will increased if there mismatch between sample and population distribution. Now that we have relation between complexity, entropy and KL divergence, we can perform some experiments by checking how these metrics behave when changing number of samples observed and the ratio of one class compared to another.



First, we can see that with N increasing, all metrics curves are smoothen and we found common curve on entropy.

Note that complexity has been approximated using theorical approach and using a Zip compressor (central graph). We can see that they have common behavior, with peak complexity with equal probabilities.

However, even if both are estimator, the difference between theorical and zip compressor is raising question. One possible answer is relative to the generation of the binary sequence using pseudo-random function.

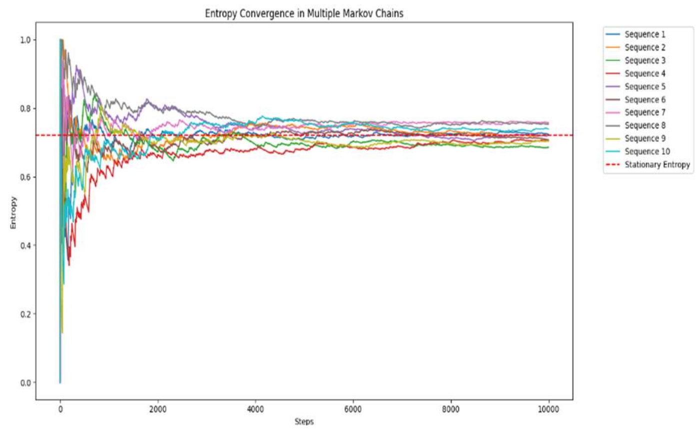


The second part of the study is to apply this method to time-series and see if it can help identify anomalies. At a specific time step, the approach is to define a time window and consider associated symbol as a draw with replacement to an urn with predefined probability (allow to have fixed probability). We define two time windows of different size and analyze Kolmogorov complexity using Zip compressor, entropy and cross-entropy of each time window. For explanatory purpose, we will also compute cross-entropy from sampled probability from short time window compared to long time window.

To check the behavior when there are some anomalies, noise is introduced in the second half of the sequence by flipping bit with predefined probability. On the first half, we can see that all metrics have their own dispersion but oscillate towards constant value. Knowing that we are using a stationary probability distribution, this is an expected and important property to confirm. On the second half of the sequence, we can see that all of these metrics are affected by the change of the sequence distribution. However, we can see that for some, for example the one involving short time windows, they have by essence have variance and it will be hard to distinguish "usual variance" and the one impacted by noise. Typical application would be fault detection by monitoring metrics variation when the process distribution is known to be stationary. But having seen variability of these metrics, it will be critical to identify the proper metrics and fine-tune related parameters to ensure good performances.

In previous part we have studied only binomial distribution, then it will be interesting to explore if such methods could be applied to another distribution. Note that concept of information source are defined within famous Shannon's paper as well as analysis markov chain. We will then explore a two-state markov chain in this section, we take a transition probability matrix A = [0.95, 0.05; 0.2, 0.8] and an initial probability p_init = [0.8, 0.2].

Markov chain source is known to be stationary, we can experiment it by drawing sequences and see that entropy of drawing a sample converge toward a constant value when n is large.



See on the left graph entropy of a symbol over time for different sequences generation.

To remind, we should also note that probability within markov chain converge to a stationary distribution, such that π A = π. Then it makes sense that associated entropy converge to a common value between all sequences : related value being H(π).

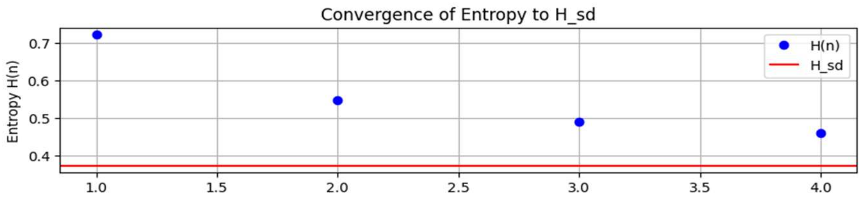
Notice here we check the entropy of a specific state at fixed time step, without influence of transition probabilities. It would be interesting to see this influence on the entropy of a sequence. Effectively, we could see in literature the notion of entropy rate of an information source: relative to the growth rate of the entropy of a sequence.

For a two-state markov chain with A = [1-α, α; β, 1-β]:

- Stationnary distribution $\pi = \left[\frac{\beta}{\alpha+\beta}, \frac{\alpha}{\alpha+\beta}\right]$ (we retrieve the result from our example),
- Entropy rate $H'(\mathcal{X}) = \lim_{n\to\infty} H(X_n|X_{n-1}, X_{n-2}, \ldots, X_1)$ converges to $H(X_2|X_1) = \frac{\beta}{\alpha+\beta}H(\alpha) + \frac{\alpha}{\alpha+\beta}H(\beta)$

Note that entropy rate can be computed using probabilities of each sequences, computing entropy and dividing by the length of the sequence. The first order entropy rate is then H(π).

See below graph with four first order entropy rates and its limits:



Looking at experimental results, we can retrieve the entropy of one state with n large. Regarding the entropy rate, we can see that the ratio between first order entropy and entropy rate limit is two. It means there exist non negligible structure from transition probabilities and the complexity of a large sequence grows smaller than its sequence length. Similarly to the previous section, one could use either entropy of a specific state or entropy rate to monitor stationary property of a sequence that is supposed to be drawn from a markov chain.

Results

On the first part, idea was merely to retrieve results from the course and complement with additional experiments to get a general understanding. Then the analysis on simple time series application showed that some stable properties could be used as a monitoring purpose. Depending on the application, metrics as Kolmogorov complexity, entropy, cross-entropy or KL divergence could be used to identify sudden noise in the information source. Finally, it was inspiring to discover steady-state regime of markov process and related convenient properties on entropy rate. Being used in a variety of application, finding metrics to monitor could help in specific tasks where we want to confirm stationary property (like fault insertion).

Discussion

Considering a first easy and stable process, it has been foreseen that related metrics should have similar properties, which could be used as monitoring purpose. However, it was interesting to see more clearly linked between Kolmogorov complexity, entropy and cross-entropy, as well as discovering that computing these metrics are manageable. Depending on specific constraints or objective of an application, there are variety of metric choice and parameters tuning.

There were no real expectations for the last part as there were no real prior background on markov process properties and links to entropy. Note it would be interesting to continue the work on investigation of entropy dynamic within a Hidden Markov model.

Besides, we can remind that both parties touch to very wide subject and then can give only a glimpse on related topics. Chosen experiments consider quite simple process with only two-states considered, which limits its application. However, provided approach should be scalable to larger distribution.

Bibliography

Shannon – A Mathematical theory of communication

Wiley – Elements of Information theory

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Majdi Ghorbel

Password Strength check using Algorithmic Information Theory

Abstract

This study introduces an innovative method for evaluating password strength based on Algorithmic Information Theory (AIT). By combining AIT principles with quantitative formulas, we present a holistic framework for assessing password robustness against cyber threats. This approach addresses current limitations in password strength assessment by emphasizing algorithmic complexity and information density.

Problem

When creating passwords for websites, users often follow common password requirements, such as:

- Minimum length,
- The inclusion of at least one uppercase letter
- At least one lowercase letter
- at least one digit

- and at least one special character.

However, these requirements do not guarantee strong passwords. For instance, the password "**Qwerty123***" meets these criteria but cracked by brute force method in less than One second!

Existing methods often overlook the importance of algorithmic complexity and information theory in evaluating password security. To address this, we propose the development of a program to assess password strength more ***accurately.***

To check passwords strength against these techniques, lets first list them and understand the vulnerabilities exploited in passwords:

Brute Force Attacks:

- Technique: Attackers systematically attempt every possible combination of characters until the correct password is found.
- Vulnerabilities Exploited: Brute force attacks exploit passwords that are short, lack complexity, and have a limited character set. They rely on the sheer computational power to crack passwords.

Dictionary Attacks:

- Technique: Attackers use a predefined list of common words, phrases, and patterns to guess passwords.
- Vulnerabilities Exploited: Dictionary attacks target passwords that are based on easily guessable words or patterns, such as "password123" or "admin."

Rainbow Table Attacks:

- Technique: Attackers use precomputed tables containing hashed passwords and their corresponding plaintext values to quickly crack hashed passwords.
- Vulnerabilities Exploited: Rainbow table attacks are effective against unsalted hashes and passwords with common or predictable character combinations.

Pattern-Based Attacks:

- Technique: Attackers exploit common patterns, such as "123456" or "qwerty," when guessing passwords.
- Vulnerabilities Exploited: Pattern-based attacks are successful when users create passwords that follow easily recognizable or predictable sequences.

Method

I developed a program in order to evaluate a password quality score, this score is based on different criteria from AIT, in fact we in order to get unpredictable password we need to ensure the password have high entropy (unpredictable) and high complexity (low compression).

The password quality is a score involving these factors:

1- Entropy Calculation:

Implementation: The system automatically calculates the entropy of each password. Entropy is determined based on the randomness and unpredictability of the password, considering factors like length and character diversity.

Outcome: Passwords are scored based on their entropy, with higher scores indicating more secure passwords.

2- Character Set Diversity:

Implementation: The system analyzes the diversity of characters in each password, checking for the presence of uppercase and lowercase letters, numbers, and special characters.

Outcome: Passwords with a greater variety of character types are rated higher for their increased resistance to certain types of attacks.

3- Pattern Detection:

Implementation: The system uses algorithms to identify common patterns and sequences (like repeated characters or sequential strings) within passwords.

Outcome: Passwords with less predictable patterns are rated higher, as they are more secure against pattern-based cracking methods.

4- Trigraph Complexity:

Implementation: The system analyzes the complexity of trigraphs (sequences of three characters) in passwords, assessing the randomness and uniqueness of these combinations.

Outcome: Passwords with more random and less common trigraphs are considered stronger and receive higher scores.

5- Common Passwords Check:

Implementation: The system cross-references passwords against databases of known common passwords and breaches.

Outcome: If a password matches a common or previously breached password

Results

We used the program to test different password, and the result is encouraging as we are able to evaluate password strength not only on length of password and the character used, but also we can identify patterns with means the password is less complex and so easy to crack.

I wanted to calculate a correlation between the score we get and the time necessary to break a password by a brute force method, unfortunately I didn't find a good program for brute force (for the moment)

```
  Password: password123
  Adjusted Entropy Score: 0.28
  Entropy Score: 
3.2776134368191157
  Trigraph Complexity Score: 1.0
  -------------------------------
  Password: P@ssw0rd!
  Adjusted Entropy Score: 2.95
  Entropy Score: 2.94770277922009
  Trigraph Complexity Score: 1.0
  -------------------------------
  Password: 1234567890
  Adjusted Entropy Score: 0
  Entropy Score: 
3.3219280948873626
  Trigraph Complexity Score: 1.0
  -------------------------------
  Password: 
CorrectHorseBatteryStaple
  Adjusted Entropy Score: 1.52
  Entropy Score: 
3.5238561897747247
  Trigraph Complexity Score: 1.0
  -------------------------------
  Password: y6_!2gF
  Adjusted Entropy Score: 2.31
  Entropy Score: 
2.8073549220576037
  Trigraph Complexity Score: 1.0
  -------------------------------
  Password: aaaaaa
  Adjusted Entropy Score: 0
  Entropy Score: -0.0
  Trigraph Complexity Score: 0.25
  -------------------------------
  Password: 156!@#qWe
  Adjusted Entropy Score: 3.17
  Entropy Score: 3.169925001442312
  Trigraph Complexity Score: 1.0
```

Bibliography

https://tests-always-included.github.io/password-strength/

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Frédéric Lauron

Zipf law on 'La Disparition' from Georges Perec

Abstract

We will compute the zipf law on 'La Disparition' from Georges Perec to see if this particlar constraint use of the french language has an influence ont the final curve.

Problem

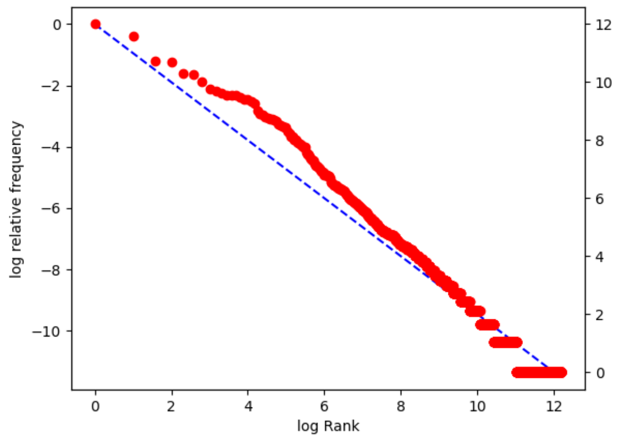
The zipf law can be verified on many different languages. Even if various languages follow this law, they do not draw exactly the same curve. How does the structure of a language influence the form of the zipf law curve ?

'La Disparition' from Georges Perec is a novel written without using the letters e, which is the most used letter in french. Does this distorsion of the use of the french language translate throw the zipf law?

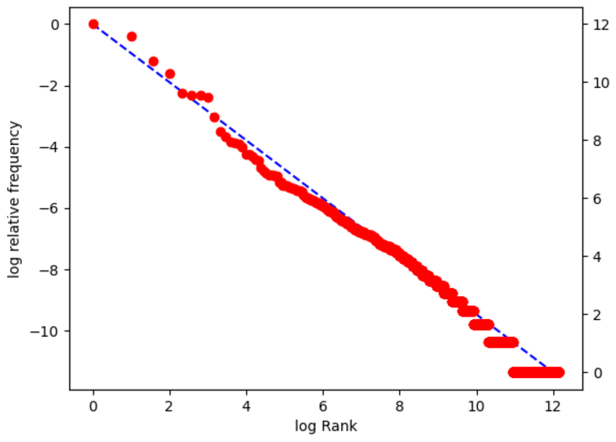
Method

We have first written a short python programme computing the relative frequency of words in a corpus. We then test it on 'Alice au pays des merveilles' (french version).Then we apply the previous code on 'La disparition' (french version).

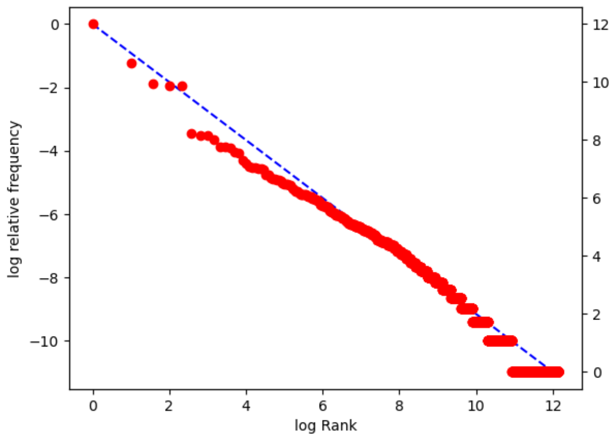
Results



(a) Alice au pays des merveilles no preprocessing

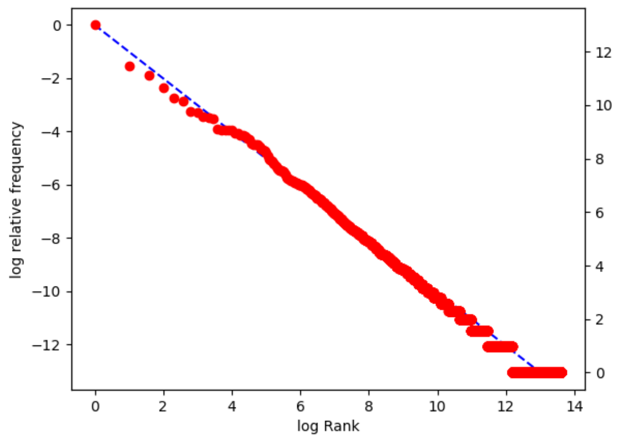


(b) Alice au pays des merveilles stopwords removed

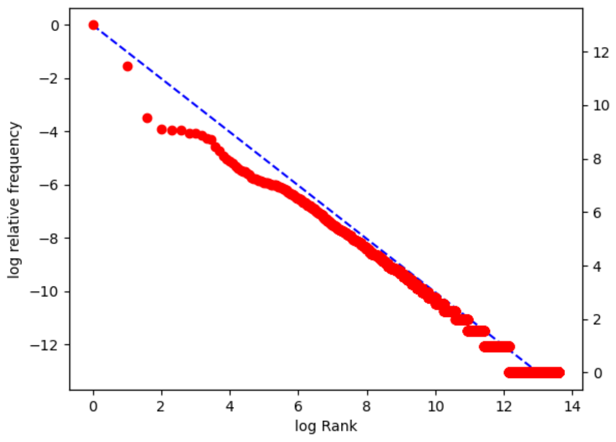


(c) Alice au pays des merveilles stopwords and punctuation removed

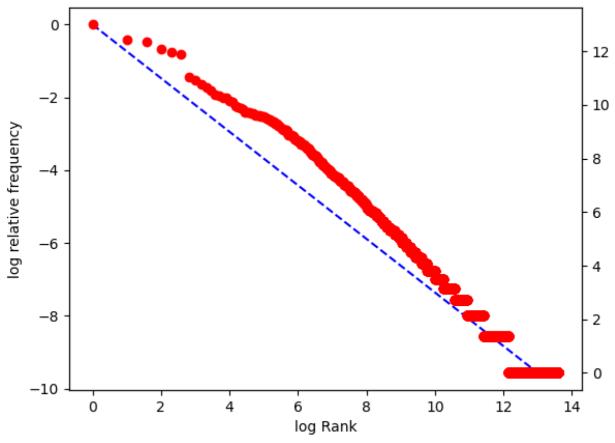
Figure 1: Zipf law Alice aux pays des merveilles



(a) La Disparition
no preprocessing



(b) La Disparition
stopwords removed



(c) La Disparition stopwords and punctuation removed

Figure 2: Zipf law La Disparition

Discussion

The zipf law can be verified on 'La Disparition' from Georges Perec. In the raw case (no preprocessing) the relative frequency of words aligned almost perfectly with the reference line. But when we remove stop words, contrary to alice 'Alice aux pays des merveilles, the curve slightly move away from the reference line. One possible interpretation is that Perec is using more stopwords than in 'Alice aux pays des Merveilles'. Futhermore when we removed punctuation, the curve slightly move aways again from the reference line. One hypothesis is that Perec is relying on using shorter sentences.

Futhermore it could be also interesting to see the zipf curve on 'Les revenentes' also from Georges Perec which is in a way the opposite of 'La Disparition' since it rely on an exlusive use of the letter e.

Bibliography

’La Disparition’
’Les Revenentes’

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Names: Maxime LEDIEU, Jaime MONTEALEGRE

Exploring Zipf's Law in Textual Diversity: Comparison between Automated Generation, Human Creativity and Statistical Models

Abstract

This report presents a comparative analysis of Zipf's Law across texts spanning five languages and various generation methods, including human authorship, AI models like ChatGPT, and statistical distributions. By examining word frequency patterns and readability, we assess how closely different sources mimic natural language. The findings aim to illuminate the linguistic alignment of machine-generated content with that of human language.

Problem

In an age where artificial intelligence (AI) is getting better at writing like humans, we face a new challenge: telling the difference between what machines write and what people write. Language models, like GPT-4, are so good now that they can create text that looks a lot like it was written by a person. This is a big deal for places like social media and news sites, where knowing who wrote something— a person or a computer— can matter a lot. When AI writes things that seems real, it could trick people, spread wrong information, or even help make fake news. That's why we need a good way to figure out if the words we read online are coming from a real person or a computer program. This report looks at how we might be able to use something called Zipf's Law, which talks about how often words show up in a language, to spot AI-written text. The idea is that the way words are used in AI text might be just a little off from how people naturally use them. If we can spot those differences, we might have a tool to help tell apart words written by people from those written by AI. Let's dive in and see what we find.

Method

This study conducts a comprehensive analysis of word frequency distributions across a variety of text corpora, examining their adherence to Zipf's Law and other linguistic measures. The research encompasses human-authored texts, AI-generated content, and a Gaussian-distributed corpus. Detailed methodologies for each analysis aspect are described below:

Text Corpus Preparation

- The selected text corpora include human-authored literature ("The Picture of Dorian Gray" in English and its translations in French, German, Russian, and Spanish, each ranging from 75,000 to 80,000 words) and AI-generated texts (a ChatGPT-generated corpus of approximately 50,000 words and AI-composed rap battle lyrics of around 600,000 words).
- A unique Gaussian-distributed text is created using a two-step process: First, 500 words are randomly chosen from an English dictionary and sorted alphabetically. Then, using a Gaussian distribution with a mean of 250 and a standard deviation of approximately 67, 50,000 indices are generated. These indices, ranging from 0 to 499, correspond to the position of words in the sorted list, forming the Gaussian-distributed text.
- Each text is preprocessed by removing stopwords and punctuation to standardize the analysis.

Tokenization

- The texts are tokenized into individual words, breaking them down into their fundamental linguistic units for frequency analysis.

Frequency Count and Ranking

- The frequency of each word in the corpus is counted. Subsequently, words are ranked in descending order based on their frequency counts.

Plotting and Theoretical Frameworks

- The frequency-rank distributions are plotted on a log-log scale. Zipf's Law, represented by the formula $f(r) = \frac{C}{r^a}$, where f is frequency, r is rank, C is a constant, and a is an exponent close to 1, is used as a benchmark. A linear pattern on this plot indicates a distribution adhering to Zipf's Law.
- The Gaussian distribution used in creating the synthetic text is expected to exhibit a bell-shaped curve in its frequency distribution, contrasting with Zipf's linear pattern.

Complexity Contrast and Readability Assessment

- To calculate the complexity contrast, we compare word frequencies in the English corpora with those in the Brown Corpus. The Brown Corpus is a comprehensive collection of American English texts, commonly used as a standard in linguistic studies. The formula used is:

$$\text{Complexity Contrast} = \left| \log_2 \left(\frac{\text{Corpus Frequency}}{\text{Brown Corpus Frequency}} \right) \right|$$

  This measure helps in understanding the variance of word usage in the studied texts compared to typical English usage.

- The readability of each text is evaluated using the Flesch Reading Ease (FRE) score. The FRE is calculated based on the average sentence length (ASL) and the average number of syllables per word (ASW), using the formula:

$$\text{FRE} = 206.835 - 1.015 \times \text{ASL} - 84.6 \times \text{ASW}$$

  This score provides insights into how accessible and understandable the text is for readers.

Analysis of the Same Corpus in Different Languages

- The study also involves analyzing "The Picture of Dorian Gray" in its various translated versions to observe the consistency of Zipf's Law across languages.

Through this method, the study aims to uncover insights into the linguistic characteristics of both human and AI-generated texts and their alignment with established linguistic principles.

Results

The Figure 1 shows the distribution of words in the Gaussian corpus. Words are organized in alphabetical order: at left there are words starting with "a", at the center words starting with "l" or "m" and at the right words starting with "z". The count of words is displayed at the y-axis. The final display is a Gaussian distribution.

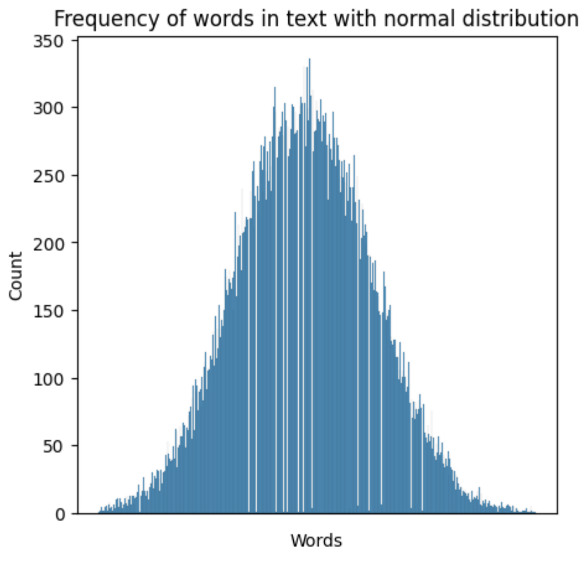


Figure 1: Distribution of words in the Gaussian corpus.

The Figure 2 displays the Zipf's law behaviour for all corpora: human-generated and AI-generated texts. The Gaussian text doesn't seem to follow this law. However, the corpus "The Portrait of Dorian Gray" and the text generated by ChatGPT seem to pass the Zipf's law, meaning they both follow this last empirical law. Indeed, they both follow a straight line with negative slope in the log-log graph. The text generated by the rap battles model, in contrast, doesn't seem to follow a straight line. It looks like there's a slight curve that describes better the behaviour of this text.

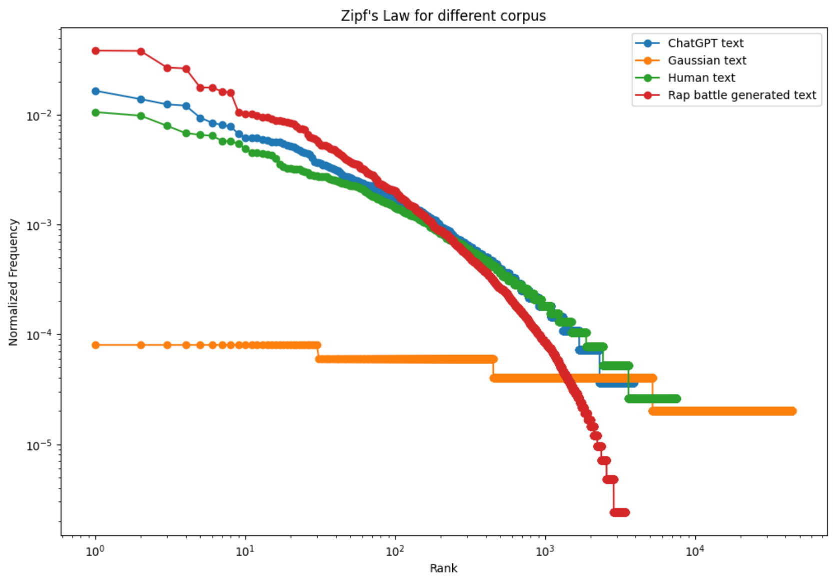


Figure 2: Zipf's law for human-generated and AI-generated corpus.

The Figure 3 shows the Zipf's law behaviour for the corpus "The Portrait of Dorian Gray" in several languages. As expected, they all seem to pass the Zipf's law: a negative slope looks to describe well the behaviour between the rank of words and their frequencies in a log-log graph. This result is congruent with the Zipf's law since it is stated that in many texts in human languages, word frequencies approximately follow a Zipf distribution.

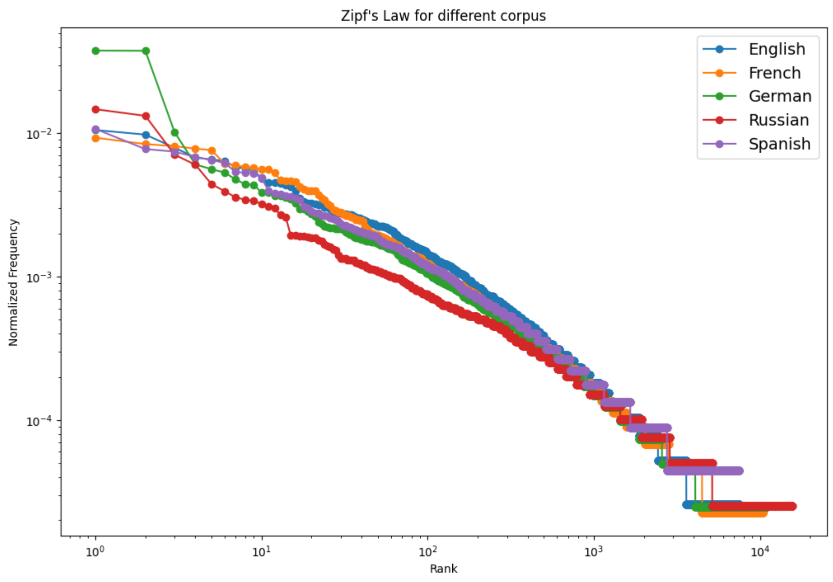


Figure 3: Zipf's law for several languages.

Complexity Contrast Analysis

The complexity contrast analysis, as shown in Table 1, effectively illustrates the thematic focus and lexical uniqueness of each corpus. This analysis reveals the underlying genre or subject matter through the specific vocabulary used in each text.

- In the ChatGPT text, words like "cosmos" and "odyssey" feature prominently with high complexity contrasts. This aligns with the fact that the ChatGPT-generated text was centered around a story set in space, thus reflecting its cosmic theme.
- The human-authored text ("The Picture of Dorian Gray") also displays words that are closely tied to its narrative, like "basil" and "picture." These words' high complexity contrasts underscore their significance within the context of the story.
- The rap battle generated text shows a different lexical pattern, with words such as "shit" and "bars" having high complexity contrasts. This highlights the unique and specific vocabulary that is characteristic of rap battles, indicating a starkly different genre and subject matter.
- Interestingly, the Gaussian text, constructed through a random distribution method, does not exhibit a clear thematic focus. The absence of a direct topic is evident from the varied and seemingly unrelated high-contrast words, which is a direct outcome of its random word selection process.

These findings from the complexity contrast analysis not only reveal the lexical diversity in each corpus but also provide insights into their respective themes and genres. The distinct vocabulary used in each text serves as a linguistic fingerprint, reflecting its specific narrative or thematic context.

Word	**Corpus**	**Complexity Contrast**
reality	ChatGPT text	26.65
knowledge	ChatGPT text	26.57
odyssey	ChatGPT text	26.50
cosmos	ChatGPT text	26.32
determinative	Gaussian text	22.15
slavish	Gaussian text	21.73
homogenization	Gaussian text	21.73
isothermally	Gaussian text	21.73
basil	Human text	26.59
harry	Human text	26.41
looked	Human text	26.08
picture	Human text	25.67
shit	Rap battle generated text	31.99
want	Rap battle generated text	31.88
mind	Rap battle generated text	31.47
bars	Rap battle generated text	31.40

Table 1: Complexity contrast in various text corpora.

Readability Assessment

The readability of the analyzed corpora, as assessed using the Flesch Reading Ease (FRE) score, reveals intriguing insights into the complexity of each text, as shown in Table 2.

- The Human text, "The Picture of Dorian Gray," achieved a readability score of 64.36, indicating a level of ease suitable for a broad audience, characteristic of conventional literary prose.
- Interestingly, the ChatGPT text scored 25.49 in readability, suggesting a higher level of complexity. This lower score indicates that when left to generate text freely, the AI tends to use more complex language structures and vocabulary. Notably, as the narrative produced by ChatGPT extended beyond 20,000 words, a tendency to repeat content was observed, necessitating intervention to encourage more creativity. This pattern underscores that unrestricted AI-generated text can become verbose and complex, deviating from simpler, more concise language usage.
- The Gaussian text's readability score of -133.33, significantly outside the normal FRE range, reflects its lack of coherent structure, which is a direct consequence of its random word selection process.
- The Rap battle generated text scored 80.40, indicating relatively easy readability. This score is likely influenced by the use of shorter words and fewer syllables common in rap lyrics, contributing to its straightforwardness despite specialized vocabulary.

The readability scores offer valuable insights into the narrative styles of different text generation sources. They highlight the challenges in AI-generated text, particularly in maintaining simplicity and avoiding repetitiveness in longer narratives.

Corpus	**Readability Score**
Human text	64.36
ChatGPT text	25.49
Gaussian text	-133.33
Rap battle generated text	80.40

Table 2: Flesch Reading Ease scores for the analyzed text corpora.

Discussion

This study's exploration of Zipf's Law in different text corpora, including human-authored, AI-generated, and statistically modeled texts, has yielded nuanced insights and raised several questions about conventional linguistic analysis methods.

Impact of Stopwords on Zipf's Law Analysis

Figures 4 and 5 reveal an intriguing aspect of linguistic analysis. Contrary to expectations, the human text and the ChatGPT text did not exhibit a clear linear trend consistent with Zipf's Law when stopwords and punctuation were removed. This deviation prompted a further investigation.

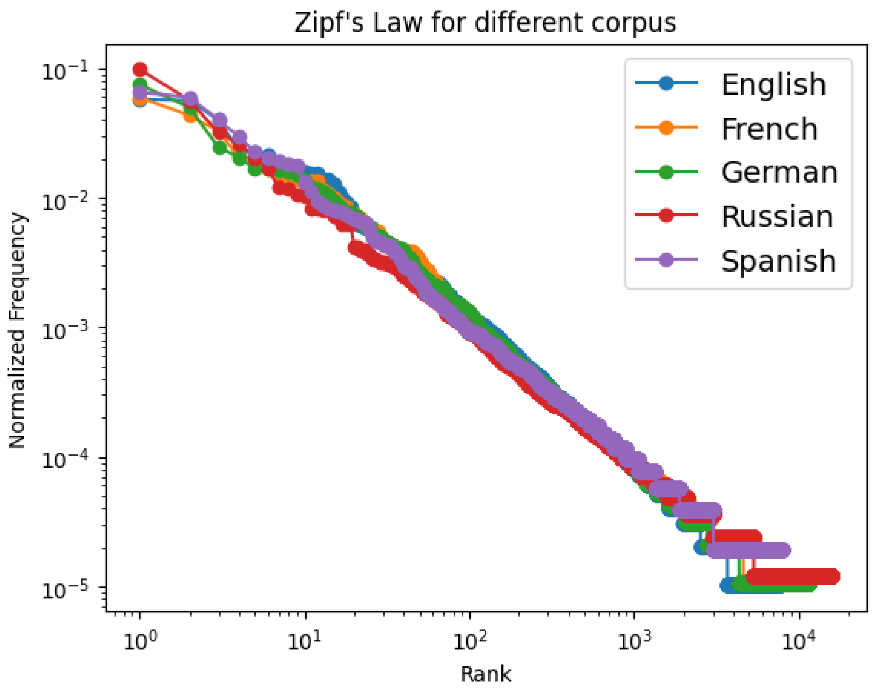


Figure 4: Zipf's law for several languages with stop words included.

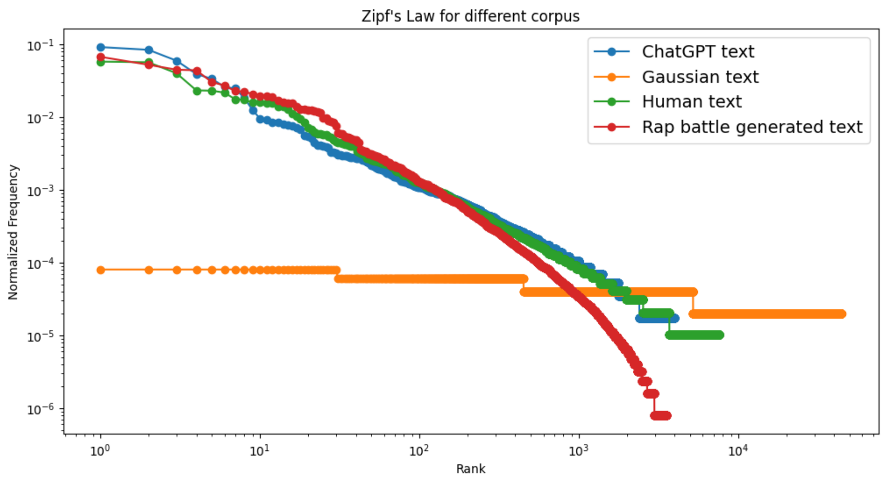


Figure 5: Zipf's law for corpora with stop words included.

Upon retaining stopwords, a significant improvement in the alignment with Zipf's Law was observed. This improvement challenges the conventional practice in linguistic analysis, which typically involves removing these elements. Stopwords, often comprising around 30% of words in a corpus, play a crucial role in maintaining the natural flow and structure of language. Their removal can distort the inherent linguistic patterns, impacting the analysis based on Zipf's Law.

Limitations and Future Perspectives

This study's exploration of linguistic patterns has brought to light several limitations and areas ripe for future research:

- A key limitation is the realization that Zipf's Law alone may not provide a definitive tool for distinguishing between human-authored and AI-generated texts. The complexity of AI language models, which can closely emulate human language patterns, calls for a more multifaceted approach to text analysis.
- The impact of different preprocessing methods on linguistic analysis, particularly the role of stopwords, emerged as a critical factor. Future studies should delve into how various text treatments influence the identification of AI-generated content.

Conclusion

This investigation into Zipf's Law across various text types illuminates the intricacies of linguistic patterns in an era where AI language models are increasingly mirroring human writing. While Zipf's Law provides a valuable lens for studying word frequency distributions, it alone does not offer a conclusive means to distinguish between human and AI-generated texts. The study highlights a pivotal challenge in linguistic analysis: the need for evolving analytical methodologies to keep pace with the sophistication of AI-generated texts. As we move forward, the field of linguistic analysis must embrace broader and more adaptable approaches, transcending traditional models to capture the nuances of both human and AI communication. This study not only reaffirms the relevance of Zipf's Law but also encourages further research into more comprehensive methods for text analysis, crucial for navigating the complexities of language in the digital age.

December, 2023

IA703

Algorithmic Information
& Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Pascal MAHÉ

Computing unexpectedness in chess

Abstract

This report presents a simple way to compute unexpectedness (surprise) in chess. The unexpectedness of an event, e, is expressed as $C_w(e) - C(e)$. We show that using the FE notation and a BFS, we can compute those two elements and deduce U(e).

Problem

In the real world, computing the unexpectedness of an event is a near-impossible task. The world complexity, $C_w(e)$, depicts the smallest causal changes made to the world to produce the event. It's almost always impossible to express meaningfully.

Similarly, the description complexity of an event, C(e), depends highly on what parameters need to be considered. Because the real world has near-infinite parameters, simply choosing the right ones is a complex task.

Reducing the problem-space to chess allows us to compute those values. The starting point of chess is known and immutable. From that we can compute $C_w(e)$. In the same way, the description of the game board gives description complexity.

In order to keep unexpectedness starting at 0, we'll add a variable, c, representing the unexpectedness of the starting position, the value of which was found empirically.

Method

We compute world complexity by counting moves to arrive at a state and description complexity with the Forsyth-Edwards Notation (FEN).

World complexity is a measure of how much the world must change to produce an event. In chess, world changes are discrete: the world can only change by one move at a time. Thus, counting the number of moves to go from the, immutable, starting position (given below) gives us directly world complexity: $C_w(e) = |moves(e)|$

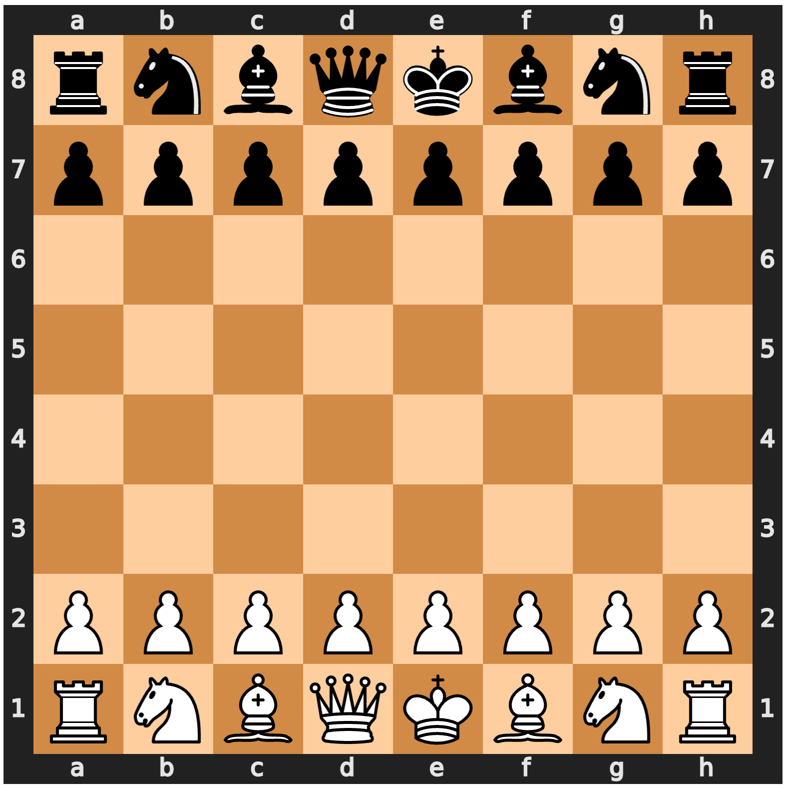


Figure 1: the starting position never changes

Now, computing the number of moves is the real challenge here. There are a finite number of possible boards but that number is in the range of $10^{46.5}$ (see: [1]). The algorithm implemented is a simple Breadth-First Search algorithm, with the pseudo-code given below:

```
def Node = (Board, state, parent)
root = Node(Board.starting_board(), unexplored, None)
bfs(root, target_board)
    queue = Queue()
    root.setAsExplored()
    queue.put(root)
    while not queue.empty():
        node = queue.get()
```

[1] ((https://math.stackexchange.com/users/21465/deedlit), 2015)

```
if boards_are_same(node.board, target_board):
    return node
new_nodes = generate_nodes(node.board)
order_nodes_by_proximity_to_target(new_nodes)
for child_node in new_nodes:
    if not nodeAlreadyExplored(child_node)
        child_node.setAsExplored()
        child_node.parent = node
        queue.put(child_node)
```

The code is directly inspired from the BFS page on Wikipedia. The basic idea is to build a queue filled with nodes (a node being a board with a parent and a move from the parent to the board). Looping on nodes in the queue, we simply generate the boards resulting from legal moves, check if those have already been explored and add those that haven't to the queue. We continue until the right node is found.

The only difference in the present algorithm is the ordering of the children in order to search those most likely to give the right answer first.

Once we have the list of moves to go from the starting position to that of the event, we simply count the moves, which gives $C_w(e)$

For description complexity, C(e), we use the Forsyth-Edwards Notation (FEN). It allows to describe a chess board state using letters (for pieces) and numbers (for empty squares). Each line of the board is described thus: if a square is occupied by a piece, it's written as that piece's letter, if it's empty, it's grouped with all adjacent empty squares and the number of empty squares is written. The following table gives which letter is used for which piece:

Black		White	
r	Rook ♜	R	Rook ♖
n	Knight ♞	N	Knight ♘
b	Bishop ♝	B	Bishop ♗
q	Queen ♛	Q	Queen ♕
k	King ♚	K	King ♔
p	Pawn ♟	P	Pawn ♙

For example, the line ‘1p1NP2P’ describes:

- 1 empty square
- A black pawn ♟ (the case of the letter gives the color: upper case for white, lower case for black)
- 1 empty square
- A white knight ♘
- A white pawn ♙
- 2 empty squares
- A white pawn ♙

Thus, the text r1bk3r/p2pBpNp/n4n2/1p1NP2P/6P1/3P4/P1P1K3/q5b1 gives the board:

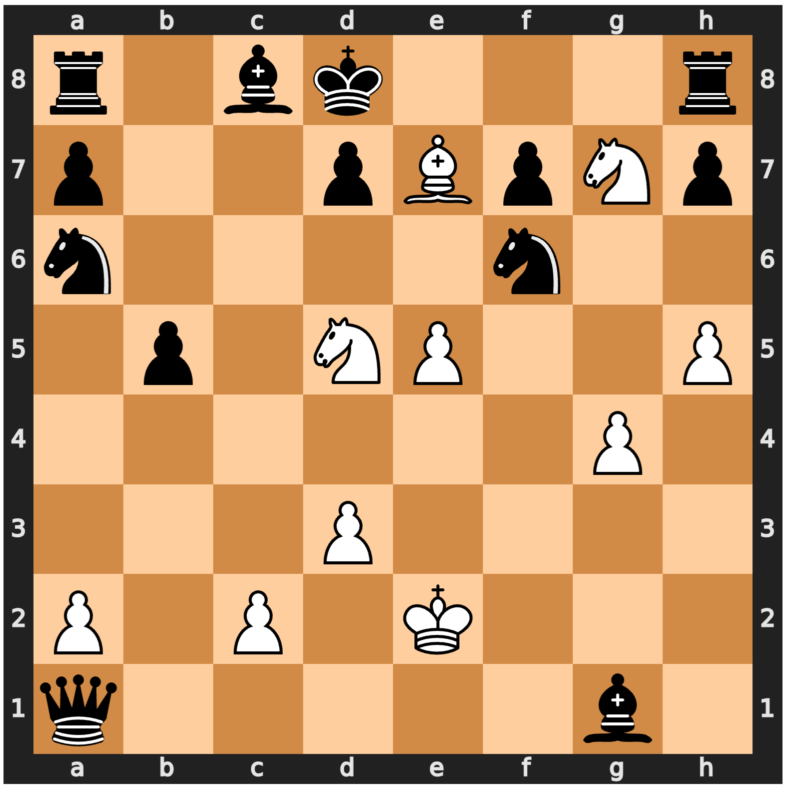


Figure 2: board for FEN 'r1bk3r/p2pBpNp/n4n2/1p1NP2P/6P1/3P4/P1P1K3/q5b1'

The FEN of a board is a complete description. It’s trivial to deduce description complexity from it: C(e) is simply C(FEN(e)), ie. the length of the FEN after compression divided by the length of the FEN: $\log_{2+}$(zip(FEN(e))) / $\log_{2+}$(FEN)

Results

Having implemented the calculation in Python, we can get the result by calling the python script. As mentioned, to “standardize” the result, the starting position is supposed to have an unexpectedness of 0. Using the method outlined here, it would have an unexpectedness of -1.192. So, we use a constant, c, with that value to ensure unexpectedness starts at 0.

For example, the starting position has an unexpectedness of:

$U(e_0) = C_w(e_0) - C(e_0) + c = 0 – 1.192 + 1.192 = 0$

While the following board, called $e_{5\text{-}2}$ because the 5th pawn has moved 2 squares up:

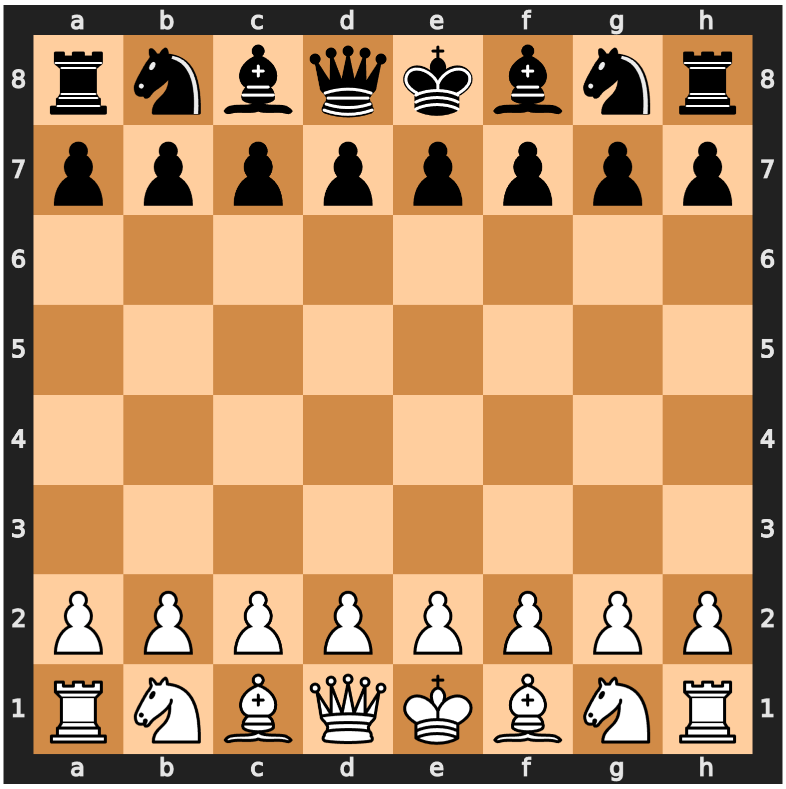


Figure 3: board with a white pawn having moved two squares up

has an unexpectedness of:

$U(e_{5\text{-}2}) = C_w(e_{5\text{-}2}) - C(e_{5\text{-}2}) + c = 1 - 1.0728 + 1.192 = 1.1191$

Discussion

Despite our best efforts, the implementation is quite slow. It cannot compute the move list for even a second move (the first move by black) in an acceptable time frame (ie. a few minutes). The logs produced by the code reveal that over 40 000 boards have been checked at that point, which is surprising: at this point in the game, each board can generate around 20 legal moves. Thus, all second moves should have been checked after 8 000 boards. It's possible that a bug prevents the second moves to be recognized as valid but since the first moves can be recognized, it's unlikely.

Furthermore, the algorithm used here, which uses extensively the python-chess to generate moves only considers legal moves. It would be impossible to find the unexpectedness of a board with 11 white pawns of 2 black kings.

Bibliography

Forsyth–Edwards Notation: https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation

Breadth-First-Search algorithm: https://en.wikipedia.org/wiki/Breadth-first_search

Python-chess: https://python-chess.readthedocs.io/en/latest/

How many legal states of chess exists?:
https://math.stackexchange.com/questions/1406919/how-many-legal-states-of-chess-exists

TELECOM Paris · IP PARIS	December, 2023 **IA703** **Algorithmic Information & Artificial Intelligence** Micro-study teaching.dessalles.fr/FCI

Name: Salimatou TRAORE

Kolmogorov complexity and programming languages

Abstract

This study explores the algorithmic complexity of various programs written in the Python, C and Java languages using the LZ78 compression method. The aim is to compare the Kolmogorov complexity and to visualize the results in order to better understand the intrinsic characteristics of each language.

Problem

In computer programming, the complexity of a code can be a crucial indicator of its performance and efficiency. However, measuring this complexity objectively and comparably between different programming languages remains a major challenge.

Method

1. Choice of languages: Python, C and Java have been selected for their popularity and their distinct programming paradigms. This is to cover a broad spectrum of programming styles and approaches.

2. Data collection: Programs performing the same operations of varying size and complexity were assembled in each language. These included standard algorithms and more advanced applications, reflecting a realistic range of uses:
   - Sum of the first n terms
   - Terminal tic-tac-toe
   - Bubble sort
   - Maze
   - Functions for Machine Learning

3. Compression Methodology: Several compression techniques (LZ77, LZ78, Huffman) have been tested to compute the Kolmogorov complexity:
   - Huffman coding: Creation of a tree structure composed of nodes
   - LZ77: Dictionary compression using a sliding window
   - LZ78: Dictionary compression using a global dictionary

After comparison, LZ78 was chosen for its relevance in assessing the information density of codes.

Here are the key stages in the process:

1. Tokenization:
   a. This function analyses the source code and breaks it down into tokens (keywords, identifiers, symbols, character strings, etc.).
   b. Uses regular expressions to identify the different elements of the source code.
   c. Returns a list of tokens, each represented by a tuple containing the token type and its value.
2. Compression:
   a. Implements the LZ78 compression algorithm.
   b. Builds a dictionary of character sequences encountered and associates an index with them.
   c. Transforms the source code into a series of indices and characters, representing the compression of the text.
3. Decompression:
   a. Reverse function of compress, using the LZ78 dictionary to reconstitute the original text.

b. Scans the compressed sequence and uses the dictionary to find the corresponding character sequences.

4. Analysis and visualization: The results obtained have been analyzed to determine the Kolmogorov complexity per language. Complexity, in this context, is evaluated by measuring the length of the compressed text in relation to the original length.

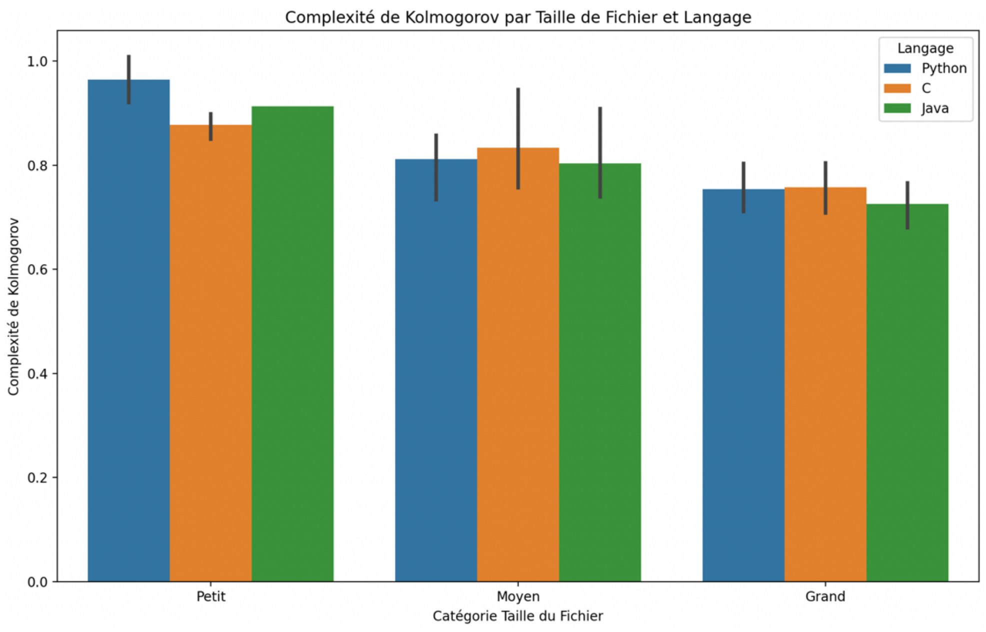
$$ratio = \frac{compressed\ length}{original\ lenght}$$

This ratio is an indicator of the redundancy and predictability of the data in the text. A higher ratio indicates higher complexity, meaning that the original text contains a significant amount of unique, non-compressible information.

Graphical visualizations have been created to illustrate the relationship between programmed size and complexity, and to compare average complexity between languages.

Results

<u>**Average complexity by language and programme size :**</u>



<u>Small files:</u>

Kolmogorov complexity is generally higher for small files, indicating information density and less redundancy.

Python and C stand out as having particularly high complexity, suggesting more concise implementations or densely used language features.

<u>Medium files:</u>

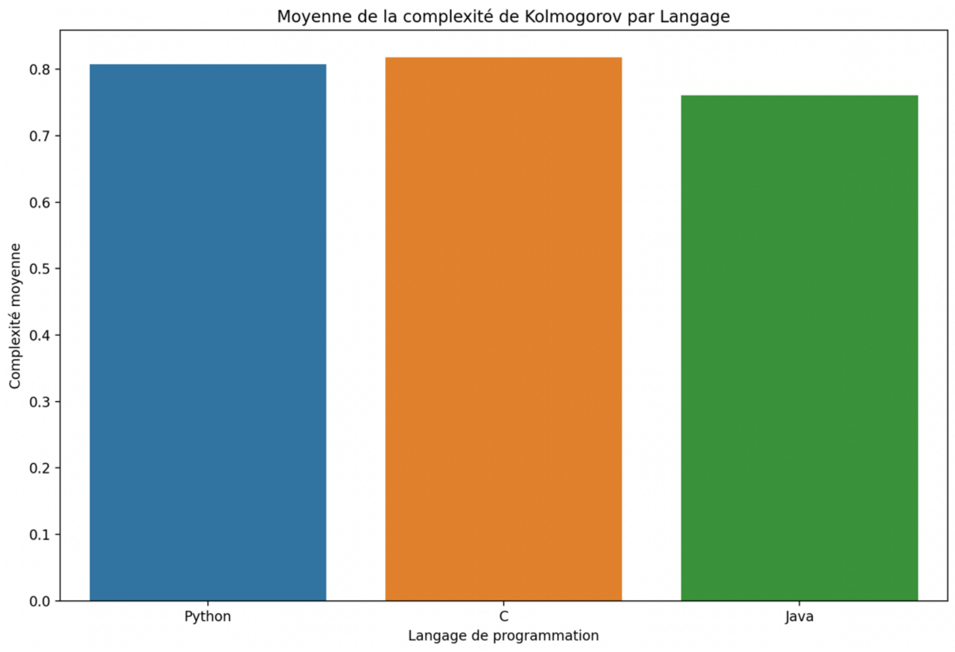
There has been a relative decrease in complexity, probably due to the introduction of more standardized code structures and the repetition of certain patterns.

Java shows a slight decrease in complexity compared to small files, which may be due to the increased use of standardized design patterns.

<u>Large files:</u>

Complexity tends to decrease slightly, reflecting the effectiveness of compression methods on longer repetitive sequences.

<u>**Average complexity per language :**</u>



<u>Java:</u>
Displays the lowest average complexity, which can be attributed to its intrinsic verbosity, resulting in redundancy that promotes better compressibility.

<u>Python:</u>
Demonstrates higher complexity due to its expressiveness, which allows more to be achieved with less code, resulting in reduced repeatability and therefore less efficient compression.

<u>C:</u>
Although less verbose than Java, the C language has an intermediate complexity due to less standardization in the structure of the code, as well as the need for more detailed instructions for low-level operations.

Discussion

Analysis of the results reveals notable aspects of Kolmogorov complexity in the different programming languages studied. The following discussion points emerge from this study:

- Compression efficiency in Java: The lowest average complexity observed in Java suggests that this language provides a more compact representation of information, despite its syntax often being perceived as verbose. This efficiency could be due to repetitive structures and patterns in the code that lend themselves well to compression. This challenges the view that conciseness is always synonymous with simplicity or efficiency.

- Python and C: Complexity and Clarity: Although Python is renowned for its clarity and simplicity, the higher average complexity compared to Java suggests that conciseness does not necessarily translate into lower complexity. In C, the even higher complexity may reflect the low-level nature of the language, requiring more detail to accomplish similar tasks.

- Influence of Size on Complexity: The relationship between programmed size and complexity highlights the importance of managing complexity in software development. For languages such as Java, where complexity remains relatively low even for larger volumes of code, this indicates an ability to maintain a certain efficiency despite increasing size.

- Study limitations and outlook: The results obtained are limited by the nature and size of the code samples analyzed. Future studies could include a wider range of programs to further generalise the findings.

In summary, this study highlights the variations in Kolmogorov complexity between different programming languages and raises important questions about the factors that influence this complexity. The results highlight the importance of choosing the right language for a given project, considering not only syntax and ease of use, but also informational efficiency and complexity management.

Bibliography

[1] Yusugomori, "DeepLearning," GitHub. Available on : https://github.com/yusugomori/DeepLearning.

[2] Lavauzelle, "Cours de théorie de l'information," Université Paris 13, 2020-21. Disponible sur : https://www.math.univ-paris13.fr/~lavauzelle/teaching/2020-21/docs/TI-poly-cours.pdf.

[3] GautierLePire, "Code source de différents algorithmes," Gist GitHub. Available on : https://gist.github.com/GautierLePire/ee104b066f184f9c62bc89e87e494c06.

[4] Wikipédia, "LZ77 et LZ78," Wikipédia. Available on : https://fr.wikipedia.org/wiki/LZ77_et_LZ78.

[5] AICourse, "Formation en intelligence artificielle," Télécom Paris. Available on : https://aicourse.r2.enst.fr/FCI.

December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: XIA Yuchen

Sentiment Analysis with Compressed Representations and Entropy-Based Weights

Abstract

The project explores the impact of compression methods, such as t-SNE and PCA, on the performance of sentiment analysis models. It also investigates how adjusting entropy-based weights during model training can optimize accuracy for labels that occupy a small portion of the dataset.

Problem

This study is based on the guided section of the sentiment analysis in the NLP course. First of all, I want to express special thanks to my teammates Salimatou, Paul, and Florent, who collaborated with me on sentiment analysis.
In the guided sentiment analysis section, we applied logistic regression to pre-processed feature vectors obtained from the GloVe library for the dataset. Although the global accuracy of the results was very high which values 89%, the model's performance on the other six labels, which constitute a smaller proportion of the dataset, was not satisfactory due to the imbalance in data distribution (data labeled as "no emotion" accounts for 89% of the dataset). Additionally, the F1 score was remarkably low, indicating a challenge in balancing precision and recall (which represents the case where the model has difficulty between correctly predicting positive examples and capturing all positive examples).
Hence, I aimed to address this issue from two perspectives: the complexity of the dataset and weight adjustment. The research investigates the impact of feature vectors compressed through dimensionality reduction on the performance of sentiment analysis models, examining whether a dataset with lower complexity is more amenable to machine learning. Furthermore, I explored how adjusting weights based on the entropy of each label could optimize the model's performance.

Method

Dimensionality reduction

The study employed two dimensionality reduction techniques, t-distributed Stochastic Neighbor Embedding (t-SNE) and Principal Component Analysis (PCA) to compress the feature vectors of datasets. The compressed feature vectors were then used for model training, and the performance changes were analyzed based on the model's classification report.
t-SNE, a non-linear technique, aimed to preserve the similarity relationships between data points and map high-dimensional data to two or three dimensions[1]. In this project, I chose to reduce the original dataset (50 dimensions) to three dimensions using t-SNE. PCA, a linear technique, mapped data to new coordinate axes by identifying the directions of maximum variance[2]. The choice to use t-SNE and PCA for comparison is mainly because they are commonly used techniques for dimensionality reduction. These two methods have different characteristics, especially in the target dimensions, which makes it possible to visualise more intuitively the impact of compression techniques on model performance. One advantage of PCA is its ability to adjust the size of the target dimension, making it convenient for studying the impact of dimensionality changes on the model. So, I also study the relationship between dimension and macro-average precision, information entropy and complexity when using PCA dimensionality reduction methods.
In the code, two functions *reduce_ dimensions_ with_ tsne* and *reduce_ dimensions_ with_ pca* are defined using TSNE and PCA functions from sklearn, which are used to downscale the database. The *n_components* parameter indicates the size of the target dimension.

```python
def reduce_dimensions_with_tsne(vectors, n_components=3):
    tsne = TSNE(n_components=n_components, random_state=42)
    return tsne.fit_transform(vectors)

def reduce_dimensions_with_pca(vectors, n_components):
    pca = PCA(n_components=n_components)
    return pca.fit_transform(vectors)
```

Entropy-Based Weights

Information entropy is chosen as a reference for weights because it is an effective way to measure data uncertainty or complexity[3]. Adjusting the weights by considering the information entropy of each label can make the model more flexible to adapt to the characteristics of different labels and improve the overall performance and generalisation ability[4]. Two directions have been explored in the research: on the one hand, as entropy increases, weights decrease; on the other hand, as entropy increases, weights increase.
The information entropy of each label is calculated based on the probability of that label appearing in the dataset. The probability p is calculated by $\frac{the\ count\ of\ label}{total\ count\ of\ all\ labels}$. The entropy of each label is then calculated using the entropy formula $-p * log_2(p)$.

Label	Entropy
Label 0	0.1493105616172063
Label 1	0.06160827568384072
Label 2	0.030770875179083883
Label 3	0.012734329047177657
Label 4	0.28688326296822564
Label 5	0.057557421630070986
Label 6	0.06249606100105652

Larger Entropy, Smaller Weight

When interpreting entropy as a measure of randomness or disorder, we tend to assign smaller weights to labels with higher entropy, since labels with higher entropy values indicate less predictability and it is desired to reduce their impact on the model.
Under this assumption, I set the weight formula as $\frac{1}{\alpha+entropy}$, where the hyperparameter α in the denominator serves as a hyperparameter. It is used not only to avoid division errors but also to adjust the impact of information entropy on weights. When α is much larger than entropy, the size of label information entropy has a small impact on weights. Conversely, when α is much smaller than entropy, the size of label information entropy has a significant impact on weights. In this study, values for α were set to 1, 0.5 and 0.001.
After calculating the weights for each label, a LogisticRegression model is created and weights are assigned to each class using the class_weight parameter. Then create a MultiOutputClassifier with Logistic Regression as the base classifier and fit it to the training data to train the model. code as follows:

```
# Create a LogisticRegression model with specified class weights
logreg_model = LogisticRegression(class_weight={i:
    weights_larger_for_larger_entropy[i] for i in range(len(
    weights_larger_for_larger_entropy))})
# Create a MultiOutputClassifier using the LogisticRegression model
clf4 = MultiOutputClassifier(logreg_model, n_jobs=-1)
# Fit the model to your training data
clf4.fit(X_train1, y_train)
```

Larger Entropy, Larger Weight

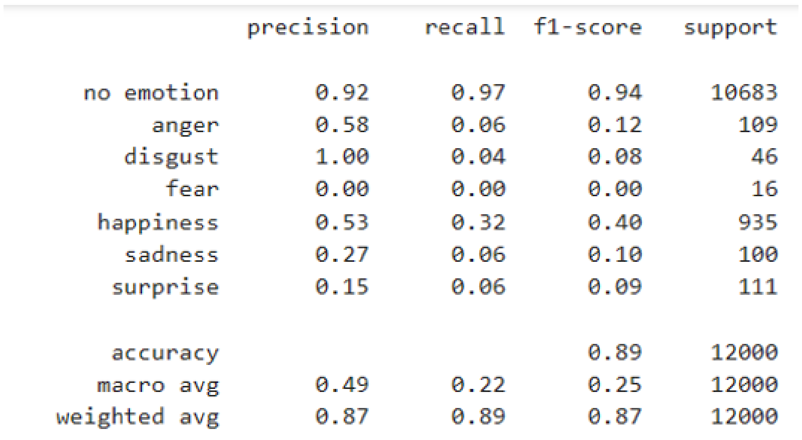
If entropy is considered as a measure of diversity or complexity, we tend to assign larger weights to labels with higher entropy because labels with higher entropy values carry more information and contribute more to the complexity of the problem. This weight adjustment method aims to better balance the model's predictions for different labels and investigates the effect of the magnitude of the entropy effect on the weights on the model performance.
Under this assumption, I set the weight formula as $\alpha + entropy$, where the hyperparameter α in the denominator serves as a hyperparameter. It is used not only to avoid division errors but also to adjust the impact of information entropy on weights. When α is much larger than entropy, the size of label information entropy has a small impact on weights. Conversely, when α is much smaller than entropy, the size of label information entropy has a significant impact on weights. In this study, values for α were set to 1, 0.5 and 0.001. After calculating the weights for each label, a LogisticRegression model is created and weights are assigned to each class using the class_weight parameter. Then create a MultiOutputClassifier with Logistic Regression as the base classifier

and fit it to the training data to train the model. code as follows:

```python
# Create a LogisticRegression model with specified class weights
logreg_model = LogisticRegression(class_weight={i:
    weights_smaller_for_larger_entropy[i] for i in range(len(
    weights_smaller_for_larger_entropy))})
# Create a MultiOutputClassifier using the LogisticRegression model
clf5 = MultiOutputClassifier(logreg_model , n_jobs=-1)
# Fit the model to your training data
clf5.fit(X_train1 , y_train)
```

Results

The results of the original dataset is presented as the graph below:



```
              precision    recall  f1-score   support

  no emotion       0.92      0.97      0.94     10683
       anger       0.58      0.06      0.12       109
     disgust       1.00      0.04      0.08        46
        fear       0.00      0.00      0.00        16
   happiness       0.53      0.32      0.40       935
     sadness       0.27      0.06      0.10       100
    surprise       0.15      0.06      0.09       111

    accuracy                           0.89     12000
   macro avg       0.49      0.22      0.25     12000
weighted avg       0.87      0.89      0.87     12000
```

Figure 1: The results of the original dataset

Dimensionality reduction

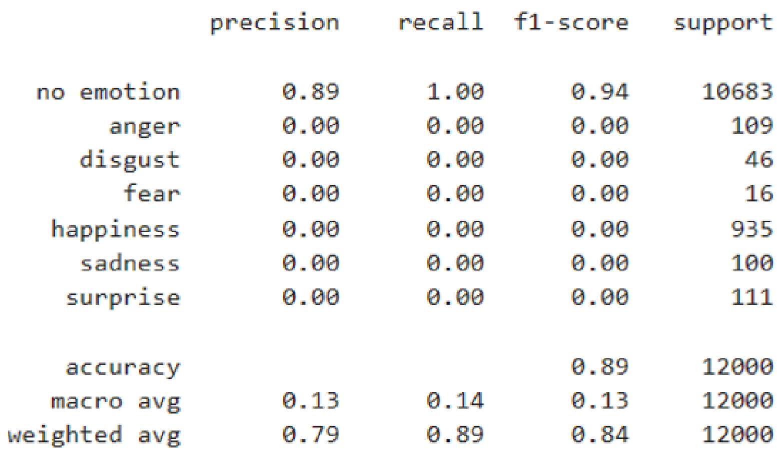
Since t-SNE compresses the vector to 3 dimensions, we first set the target dimension of PCA to 3. The information entropy of the compressed dataset is shown in the following table 1. We can see that the dimension reduction successfully reduces the total entropy of the dataset significantly, which means that the complexity and uncertainty of the dataset is significantly reduced.

Dataset	Entropy
Original Vector Entropy	83503.89042781954
t-SNE Compressed Vector Entropy	14200.566063789109
PCA Compressed Vector Entropy	13749.92270467712

Table 1: Table of overall entropy after different compression

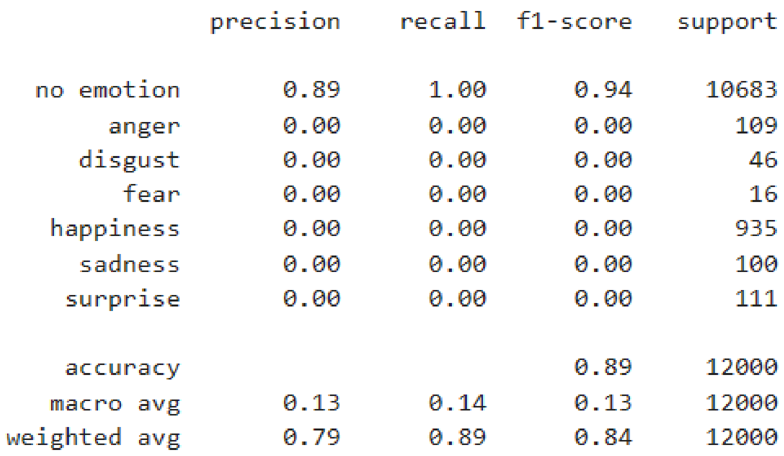
Based on the results of the model learning figure 2 and figure 3, we can observe that the reduction in dataset complexity did not have a positive impact on accuracy. The performance of the 'no emotion' label, which constitutes the vast majority of the database, remained unaffected.

However, there was a noticeable decline in the performance of labels with smaller proportions. I think the significant reduction from 50 dimensions to 3 dimensions might be causing a too extensive span, leading to the loss of crucial information.



```
              precision    recall  f1-score   support

  no emotion       0.89      1.00      0.94     10683
       anger       0.00      0.00      0.00       109
     disgust       0.00      0.00      0.00        46
        fear       0.00      0.00      0.00        16
   happiness       0.00      0.00      0.00       935
     sadness       0.00      0.00      0.00       100
    surprise       0.00      0.00      0.00       111

    accuracy                           0.89     12000
   macro avg       0.13      0.14      0.13     12000
weighted avg       0.79      0.89      0.84     12000
```

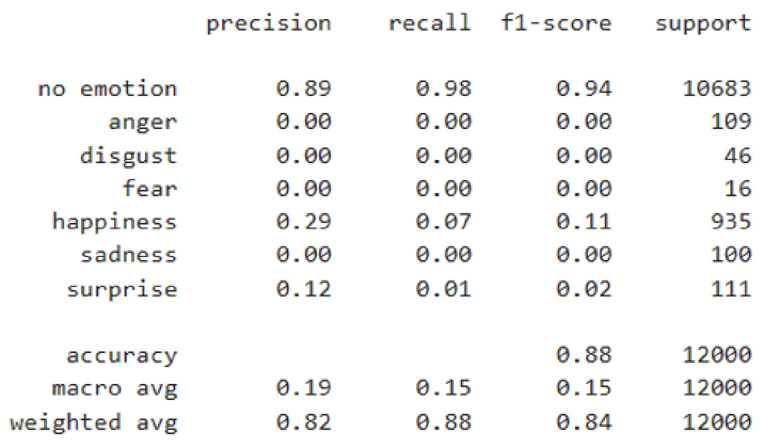
Figure 2: The results of the dataset compressed by t-SNE with aimed dimension 3



```
              precision    recall  f1-score   support

  no emotion       0.89      1.00      0.94     10683
       anger       0.00      0.00      0.00       109
     disgust       0.00      0.00      0.00        46
        fear       0.00      0.00      0.00        16
   happiness       0.00      0.00      0.00       935
     sadness       0.00      0.00      0.00       100
    surprise       0.00      0.00      0.00       111

    accuracy                           0.89     12000
   macro avg       0.13      0.14      0.13     12000
weighted avg       0.79      0.89      0.84     12000
```

Figure 3: The results of the dataset compressed by PCA with aimed dimension 3

Therefore, to avoid this issue, I conducted a second training with PCA, setting the target dimensions to 40. The entropy of the compressed dataset was 48179.307584280556. Although the reduction was not as significant as in the case of 3 dimensions, the information complexity still notably decreased. Based on the results figure 4, we observed that the overall impact on model performance did not change significantly compared to the 3-dimensional scenario, and the performance of minority labels still decreased noticeably. However, dimensionality reduction did accelerate the model training process; the original dataset took 14 seconds to train, while t-SNE and PCA took 0.3 seconds and 1.4 seconds, respectively.



```
              precision    recall  f1-score   support

  no emotion       0.89      0.98      0.94     10683
       anger       0.00      0.00      0.00       109
     disgust       0.00      0.00      0.00        46
        fear       0.00      0.00      0.00        16
   happiness       0.29      0.07      0.11       935
     sadness       0.00      0.00      0.00       100
    surprise       0.12      0.01      0.02       111

    accuracy                           0.88     12000
   macro avg       0.19      0.15      0.15     12000
weighted avg       0.82      0.88      0.84     12000
```

Figure 4: The results of the dataset compressed by PCA with aimed dimension 40

The graph 5 illustrates the relationship between dataset dimensions, macro average precision (which is a more adapted method to value the overall accuracy due to the imbalance in data distribution), entropy, and complexity (related to the length of the reduced vectors). The overall trend indicates a decrease in these values as the dataset dimensions reduce, demonstrating a consistent pattern.

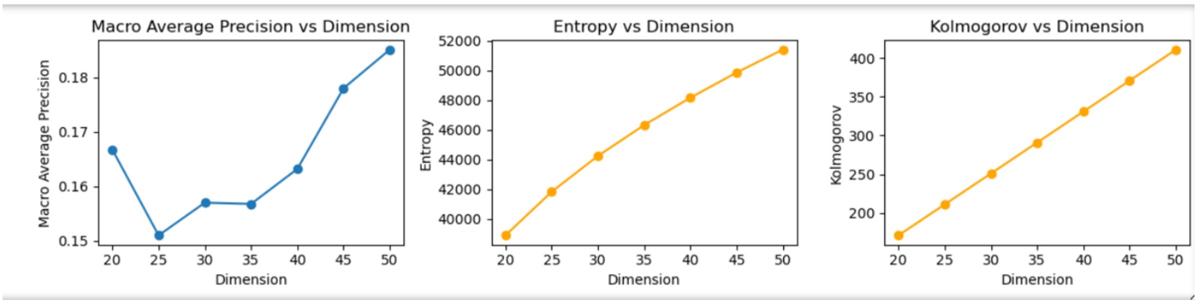


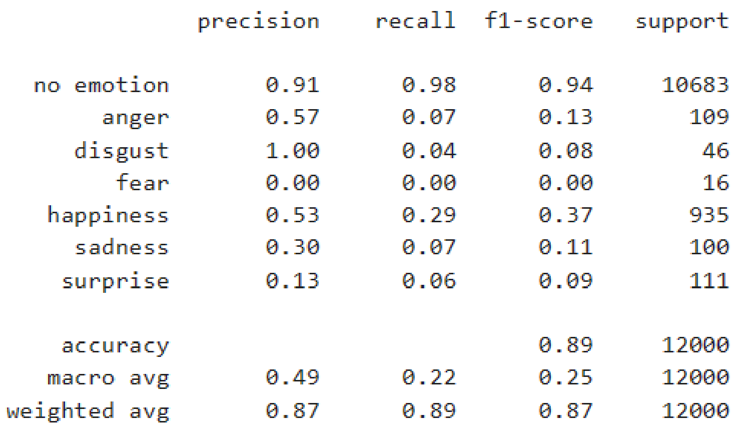
Figure 5: PCA reduction

Entropy-Based Weights

Larger Entropy, Smaller Weight

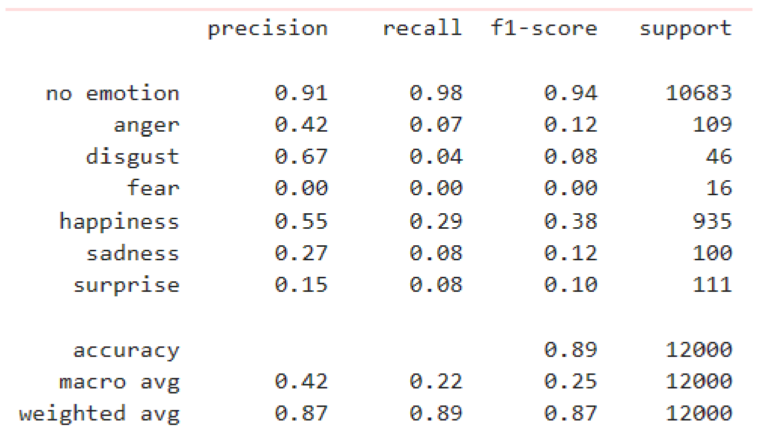
Compared to the results of the original dataset in figure 1, assigning smaller weights to labels with higher entropy has resulted in a modest optimization of the model. The model is designed to pay more attention to labels with lower entropy, thereby improving accuracy for these labels and enhancing the overall stability of the model.
Based on the overall accuracy, this weighted adjustment strategy did not lead to a significant improvement in the overall model performance. However, when examining the F1 scores for minority labels, there is a slight improvement in addressing the imbalance between precision and recall. Particularly, there is a noticeable increase in the recall for minority labels, indicating that the model more effectively captures positive samples for these categories. The model's recognition of minority categories has improved. However, the relatively insignificant improvement suggests that the model is influenced by other complex factors, and fine-tuning the weights does not bring substantial advantages. The selection of weights may require more nuanced adjustments to identify the optimal weight distribution. Taking the three values of α during the learning process

as an example, when α is set to 0.001, indicating a significant impact of entropy on weights, the optimization of F1 scores becomes more pronounced.



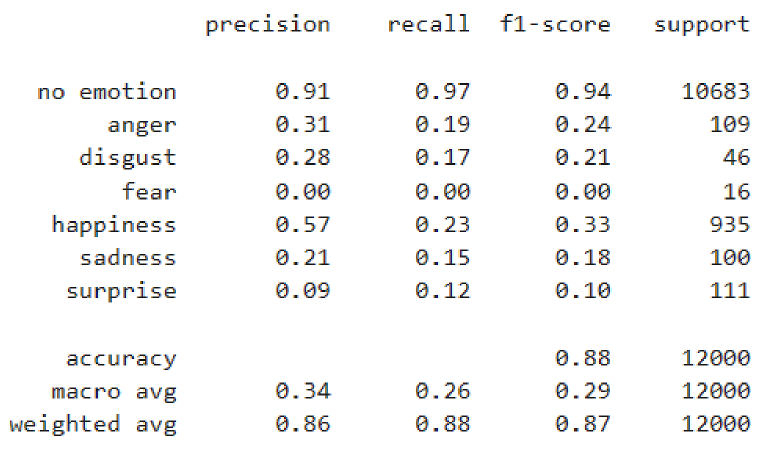
```
              precision    recall  f1-score   support

  no emotion       0.91      0.98      0.94     10683
       anger       0.57      0.07      0.13       109
     disgust       1.00      0.04      0.08        46
        fear       0.00      0.00      0.00        16
   happiness       0.53      0.29      0.37       935
     sadness       0.30      0.07      0.11       100
    surprise       0.13      0.06      0.09       111

    accuracy                           0.89     12000
   macro avg       0.49      0.22      0.25     12000
weighted avg       0.87      0.89      0.87     12000
```

Figure 6: The results with $\alpha = 1$



```
              precision    recall  f1-score   support

  no emotion       0.91      0.98      0.94     10683
       anger       0.42      0.07      0.12       109
     disgust       0.67      0.04      0.08        46
        fear       0.00      0.00      0.00        16
   happiness       0.55      0.29      0.38       935
     sadness       0.27      0.08      0.12       100
    surprise       0.15      0.08      0.10       111

    accuracy                           0.89     12000
   macro avg       0.42      0.22      0.25     12000
weighted avg       0.87      0.89      0.87     12000
```

Figure 7: The results with $\alpha = 0.5$

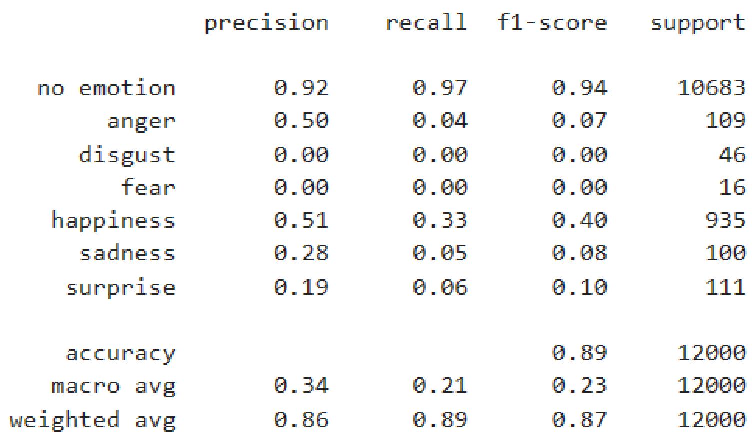


```
              precision    recall  f1-score   support

  no emotion       0.91      0.97      0.94     10683
       anger       0.31      0.19      0.24       109
     disgust       0.28      0.17      0.21        46
        fear       0.00      0.00      0.00        16
   happiness       0.57      0.23      0.33       935
     sadness       0.21      0.15      0.18       100
    surprise       0.09      0.12      0.10       111

    accuracy                           0.88     12000
   macro avg       0.34      0.26      0.29     12000
weighted avg       0.86      0.88      0.87     12000
```

Figure 8: The results with $\alpha = 0.001$

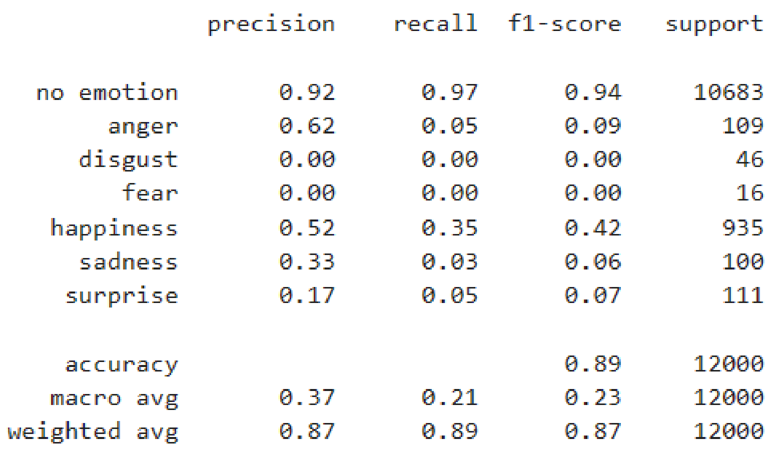
Larger Entropy, Larger Weight

Giving greater weight to labels possessing greater entropy does not seems to optimise the model compared to the results for the original dataset in figure 1. Contrary to the theoretical assumptions, the new model performs worse compared to the original model on the labels 'fear' and 'disgust', neither of which can correctly classify any sample. Even worse, the model's performance on the label 'anger' and label 'sadness' becomes 0% when $\alpha = 0.001$. Based on the classification report of the new model, it can be concluded that this hypothesis not only failed to improve the accuracy for minority labels but also made no contribution to optimizing the F1 score. This suggests that the greater the effect of entropy on the weights, the more pronounced the deterioration of the model is in that hypothetical case. This finding emphasises that adjusting the effect of label entropy on weights may require a finer balance to avoid adverse effects on label-specific performance.



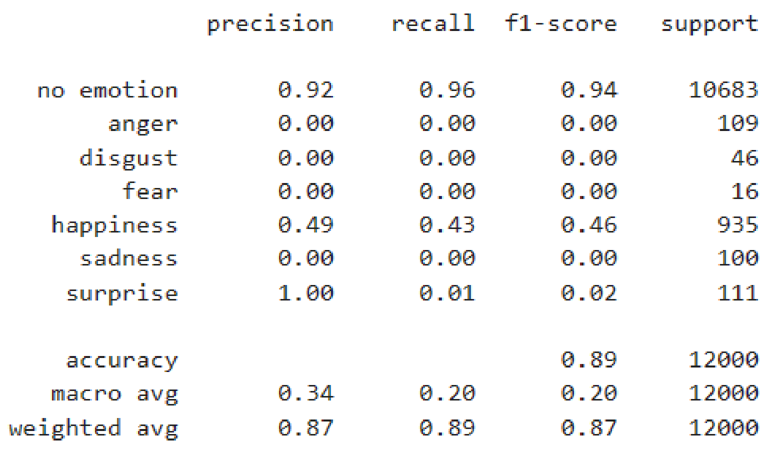
```
              precision    recall  f1-score   support

  no emotion       0.92      0.97      0.94     10683
       anger       0.50      0.04      0.07       109
     disgust       0.00      0.00      0.00        46
        fear       0.00      0.00      0.00        16
   happiness       0.51      0.33      0.40       935
     sadness       0.28      0.05      0.08       100
    surprise       0.19      0.06      0.10       111

    accuracy                           0.89     12000
   macro avg       0.34      0.21      0.23     12000
weighted avg       0.86      0.89      0.87     12000
```

Figure 9: The results with $\alpha = 1$



```
              precision    recall  f1-score   support

  no emotion       0.92      0.97      0.94     10683
       anger       0.62      0.05      0.09       109
     disgust       0.00      0.00      0.00        46
        fear       0.00      0.00      0.00        16
   happiness       0.52      0.35      0.42       935
     sadness       0.33      0.03      0.06       100
    surprise       0.17      0.05      0.07       111

    accuracy                           0.89     12000
   macro avg       0.37      0.21      0.23     12000
weighted avg       0.87      0.89      0.87     12000
```

Figure 10: The results with $\alpha = 0.5$



```
              precision    recall  f1-score   support

  no emotion       0.92      0.96      0.94     10683
       anger       0.00      0.00      0.00       109
     disgust       0.00      0.00      0.00        46
        fear       0.00      0.00      0.00        16
   happiness       0.49      0.43      0.46       935
     sadness       0.00      0.00      0.00       100
    surprise       1.00      0.01      0.02       111

    accuracy                           0.89     12000
   macro avg       0.34      0.20      0.20     12000
weighted avg       0.87      0.89      0.87     12000
```

Figure 11: The results with $\alpha = 0.001$

Discussion

The investigation delves into the impact of compression methods such as t-SNE and PCA on sentiment analysis model accuracy. The significant reduction in dataset complexity, as evidenced by a substantial decrease in entropy, contrasts with the unexpected finding that the diminished dataset complexity does not positively affect overall accuracy. Specifically, the performance of the majority label "no emotion" remains largely unaffected, while minority label performance experiences a noticeable decline. Nevertheless, both techniques accelerate the model training process compared to the original dataset.
Subsequently, the study explores the adjustment of weights based on label entropy during model training. The hypothesis that larger entropy leads to smaller weights optimizes the system but it seems like the model may have limited sensitivity to entropy as the optimization is not really obvious. Although there is no significant change in overall accuracy, there is an improvement in the recall of minority labels, resulting in a more balanced F1 score. Moreover, the magnitude of entropy's impact on weights influences model optimization, emphasizing the nuanced relationship between information entropy, weight adjustments, and overall model performance. The need to find a suitable hyperparameter α to modify the impact of entropy on weights underscores the complexity of this relationship for optimal model optimization.

In conclusion, the project suggests that dimensionality reduction through t-SNE and PCA does not enhance sentiment analysis model accuracy but does expedite the training process. However, adjusting weights based on label entropy proves to be beneficial, particularly for minority group labels. The study highlights the complexity of optimizing models in the context of imbalanced datasets and underscores the potential advantages of entropy-based weight adjustments.

References

[1] Wikipedia contributors, *T-distributed stochastic neighbor embedding — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=T-distributed_stochastic_neighbor_embedding&oldid=1187642267`, 2023.

[2] Wikipedia contributors, *Principal component analysis — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Principal_component_analysis&oldid=1189954365`, 2023.

[3] Wikipedia contributors, *Entropy (information theory) — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Entropy_(information_theory)&oldid=1190498462`, [Online; accessed 20-December-2023], 2023.

[4] M. Zhou, X.-B. Liu, J.-B. Yang, Y.-W. Chen, and J. Wu, “Evidential reasoning approach with multiple kinds of attributes and entropy-based weight assignment,” *Knowledge-Based Systems*, vol. 163, pp. 358–375, 2019.