Classification problem with data compression

Identify if a text was rewritten by ChatGPT

TAI - 2023/2024

Tiago Costa Nº 98601
João Mourão Nº 102578
João Rodrigues Nº 102487

Introduction

With the advancement of AI technology, the use of AI bots like ChatGPT to generate and rewrite text has become more prevalent. Identifying rewritten texts has become an important tool for various purposes, with the most important one being finding plagiarism.

In this report, we will document our solution to this problem. We created a program that uses data compression to figure out if a text was rewritten by ChatGPT or not. Since chatGPT is the most popular AI bot currently, it is the most likely to be used for rewriting text, which is why we are targeting it in particular.

Classification problem

The goal of our work is to be able to identify if a piece of text was rewritten by chatGPT or not, which is a classification problem. Normally, this would mean that we would have to extract features from our data and then carefully select these features to get the best possible results. However, we are taking a different approach for this problem, an information-theoretic approach. We will be using compression algorithms to measure the similarity between the text under analysis, files that were rewritten by chatGPT and files that weren't rewritten.

The compression algorithm that we will use is the Markov model, also known as a finite-context model, since it's an algorithm that takes advantage of the way language is written, identifying patterns in our language and it's also an algorithm that can be trained on other data, which lets us create a model for both rewritten text and not rewritten text, each one being better at compressing their respective type of text.

Finite-context model

A finite-context model uses the previous values of the sequence to predict what the next value will be. To do this, it stores all possible sequences and the amount of times each character occurred after that sequence. With this information, we can then predict what the next character will be. In our case, we will be simulating the finite-context model, so we calculate the probability of the next character occurring and use that probability to calculate the amount of data needed to store the character instead.

The types and amount of values it can use in the prediction can vary, however, high amounts of values will need exponentially more space, since the model needs to store data about all possible combinations, so it's better to limit it to a smaller amount of values. In our case, we will be using the alphabet as our set of values, so our model should be limited to using at most the 2 previous characters in its prediction.

To estimate the amount of bits needed to compress a character, we first calculate the probability of the character occurring using the following formula where N is the amount of times a character occurred after a sequence, e is the next character, c is the context (the

previous characters in the sequence that are used for the prediction), Σ is the set of all characters and alpha is a smoothing parameter that ensures that the probability never starts at 0:

$$P(e|c) = \frac{N(e|c) + \alpha}{\sum_{s \in \Sigma} N(s|c) + \alpha|\Sigma|}$$

After we know the probability of the character occurring, we use the following formula to calculate the amount of bits needed where P(s) is the probability calculated in the previous formula:

$$- \ log2\ P\ (s)$$

Using 100 Mb of text that wasn't rewritten by ChatGPT and 300 Mb that was rewritten, we created an estimation of a finite-context model for each type of text. This allows us to have 2 separate models that are more efficient at compressing the type of text that they were trained on. When we estimate the data needed to compress a file under analysis with both of them, we will get different results, since they are trained with different data.

The model that has the best compression rate for a file is the one that's trained on data that's similar to the file we are analyzing, which solves our classification problem. Files that have a better compression rate with the ChatGPT model were rewritten and files that have a better compression rate with the other model weren't rewritten.

Model implementation

In this chapter, we're going to explain how we applied those concepts in practice.

CleanData

Firstly, we had to get our data ready. We had a lot of data, but in order for the model to perform the best we created a python script to clean the data in the way we wanted. This script basically reads a csv file with many fields and writes the text and label fields into a new csv. Besides this, the script also modifies the text field so that only characters from 'a' to 'z', from 'A' to 'Z', and numbers from 0 to 9 remain in the file.

HashTable

Hashtable is one of the classes of our program. It contains a single attribute that is a hashmap with string as key and another hashmap as value with a char key and integer value. The first key is the context with k characters, the k will be defined later on, the second key is the char next to the context and the value is the number of times that the char

came after that context. Having this in memory allows us to calculate the probabilities any time we desire. The probability is also calculated in this class with the formula presented before and to calculate it we need to give the context and next char , if those are not available in the table, the count (value) is considered zero.

In this class we also have the option to serialize the hashtable, i.e., we can save the hashtable in disk, but also load the hashtable from a file. With these functionalities we don't need to train the model every time we want to predict one text.

MarkovModel

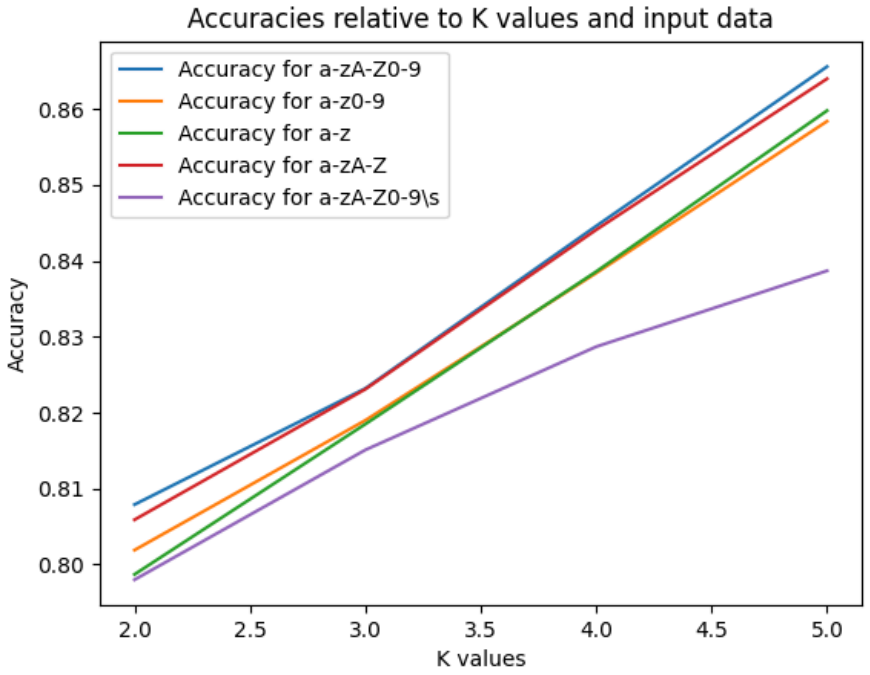
MarkovModel is another class and includes the HashTable referenced before, k (int) and alpha (int) as attributes. K is the length of context and alpha is the smoothing factor, avoiding probabilities equal to zero. The most important methods are "train" and "bitsToCompress". The method "train", will basically fill the hash table. It gets a string as input, and iterates it, getting k characters +1 (for the next char), and increments the entry in the hashtable with key equals to the k characters gotten in the iteration and second key the next char. The method "bitsToCompress" returns the quantity of bits needed to compress one certain document and it performs a similar iteration to the one did in "train", however, this time we will use context and next char not to fill the hashtable but to calculate the probability together with the alpha value. The final output will be the sum of -log2(probability) of every iteration in that document.

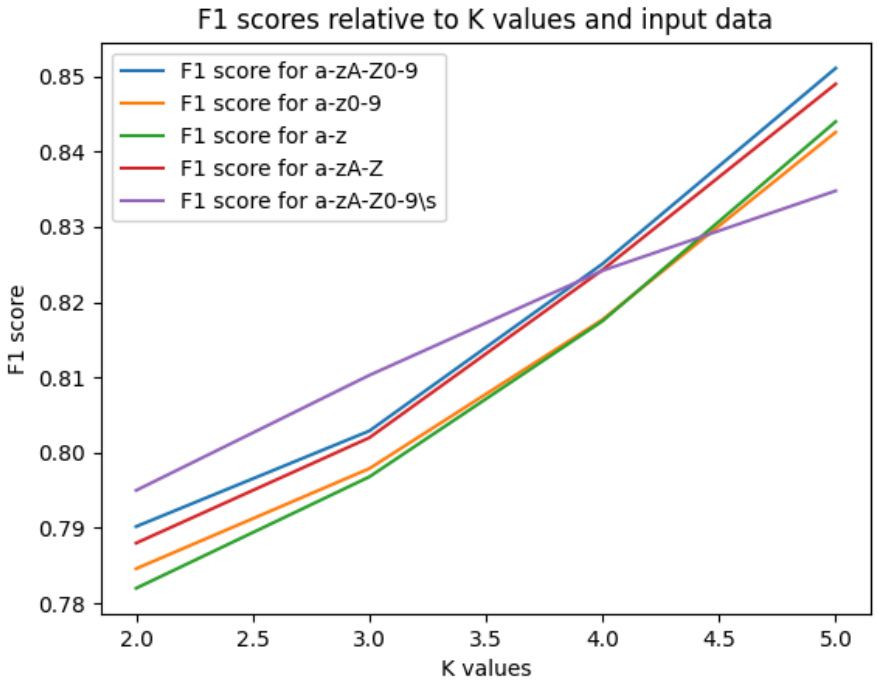
MainClass & Metrics

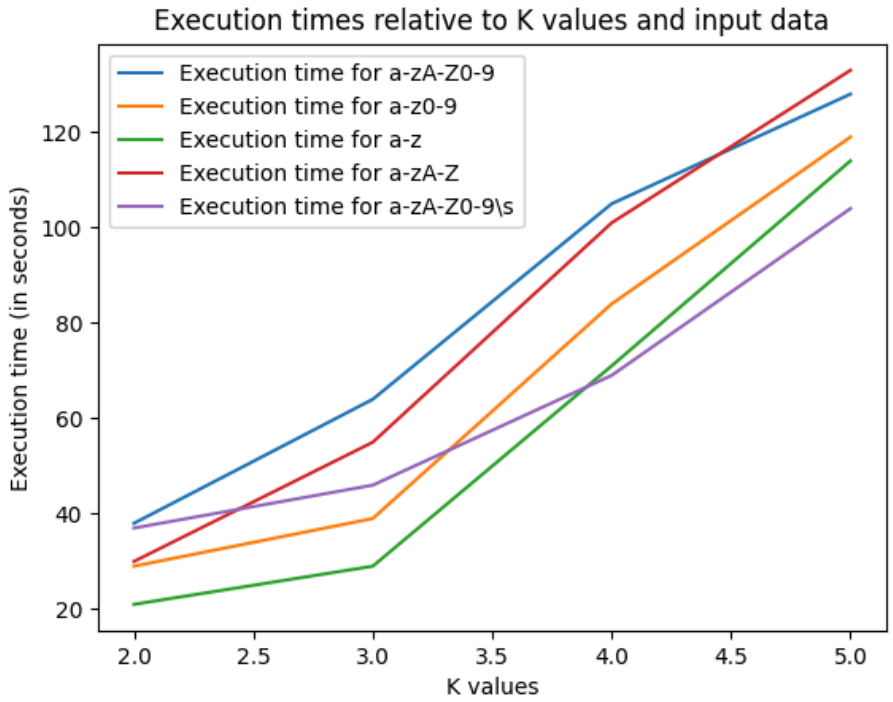
MainClass basically includes two MarkovModels one for human and another for AI and it accepts two files for human and two for ai, one is for train and the other for testing. This class has the possibility to train both models, automatically redirecting each one of the files for the correct model as well as the model testing. The test is done by passing the one specific text through the function "bitsToCompress" and the model producing the smaller value is labeling the text, i.e., if ai needs less bits to compress the text, we predict that the text was written by AI. Besides this, the prediction has also the possibility to fill the confusion matrix during the testing. Confusion Matrix is a structure present in our class Metrics that is extended by the MainClass, which is a hashtable with four possibilities of key: false positive, false negative, true positive, true negative. All the metrics like accuracy,precision,f1 score originate from the confusion matrix .

Experiments to improve results

In order to find the best model, we run some experiments varying the value of K and the input data format. For K we varied from 2 to 5 and for the input data we decided to create versions of it where we varied the existence of capital letters, numbers and spaces. In these tests, we supplied the models with nearly 300 MB of data: 150 MB of GPT text and 150 MB of human text.







Through this experiment, we could also conclude that for higher K values the execution time increases.

Results

After analyzing the results in the previous section, we concluded that the model performs the best with K=5 and with the data input with capital letters and numbers, but without spaces. Even though this was the best model we could create, it still has some problems as it misclassified texts generated by GPT-3.5 considerably more often than texts written by humans.

Actual \ Predicted	Positive	Negative
Positive	1980	637
Negative	56	2482

Recall: 0.7566
Precision: 0.9725
Accuracy: 0.8656
F1 score: 0.8511

Conclusion

AI is a powerful, but dangerous tool, that can be used to rewrite texts so that they are written in a more professional way, but that also can be used to commit plagiarism by rewriting other people's texts.

Plagiarism is a serious problem that is faced in a lot of professions, including professions related to education and science. AI makes it easier to plagiarize other people's work by easily rewriting said work, which can hide the fact that it's someone else's work.

We were able to create a program that helps alleviate this problem by identifying if a file was rewritten by chatGPT or not. Files that were rewritten need to be analyzed, because rewriting text could mean that plagiarism occurred, but it's not a guarantee. With our solution, the amount of text that needs to be analyzed is reduced significantly, making it easier to find the plagiarized work.

Contributions

João Mourão - 47.5%
João Rodrigues - 47.5%
Tiago Costa - 5%