Universidade de Aveiro

School Year 2021/2022

TAI - Practical Assignment 2

Diogo Bento 93391

Pedro Laranjinha 93179

Leandro Kiame 78177

Professors Armando J. Pinho & Diogo Pratas

Introduction

For this project it was requested that we develop programs that utilize **finite-context models** to calculate file sizes and identify or locate file languages based on compression according to a linguistic model.

Code Steps

lang.py

The program developed to return an estimation of a text's size according to a given model is ***lang.py***.

This program has 4 arguments:

- **--order:** sets the order of the model, default is **2**
- **--smoothing:** sets the model's smoothing parameter, default is **1**
- **--classsource:** the name of the file that will be read to generate the model, this value is sensitive to the current folder context
- **--input:** the name of the file whose cost will be analyzed

From inside the /bin/ folder, one can run the code by using *python3* as follows:

python3 lang.py <args>

This program functions as a pipeline of functions:

1. We use the order of the model and the input file to return a map that contains how many times a symbol occurs as follow-up to a context, and the alphabet of the text.
2. Then we use the symbol follow-up map, the alphabet and the smoothing parameter to return a cost table which contains the bit cost of any character in the alphabet occurring in each context.
3. Finally we use the cost table, alphabet, and text under analysis to return the estimation of the text's size in bits.

findlang.py

The program **findlang.py** allows the user to scan a file using several pre-compiled models and find which ones the text is most likely to be in based on file size when compressed according to each model.

This program has 2 arguments:

- **--classes:** folder the class files (models) are stored in, folder context sensitive
- **--input:** the name of the file whose cost will be analyzed

From inside the /bin/ folder, one can run the code in a similar way to ***lang.py***:

python3 findlang.py <args>

This program loads every model file in the provided folder, calculates file size for each, and displays all models ordered by associated cost.

locatelang.py

The program which was developed to find text language is **locatelang.*py***.

This program is different from **findlang** because it considers that a text can have several languages and as such tries to find intervals where the text might be in a given language, rather than simply select a global language.

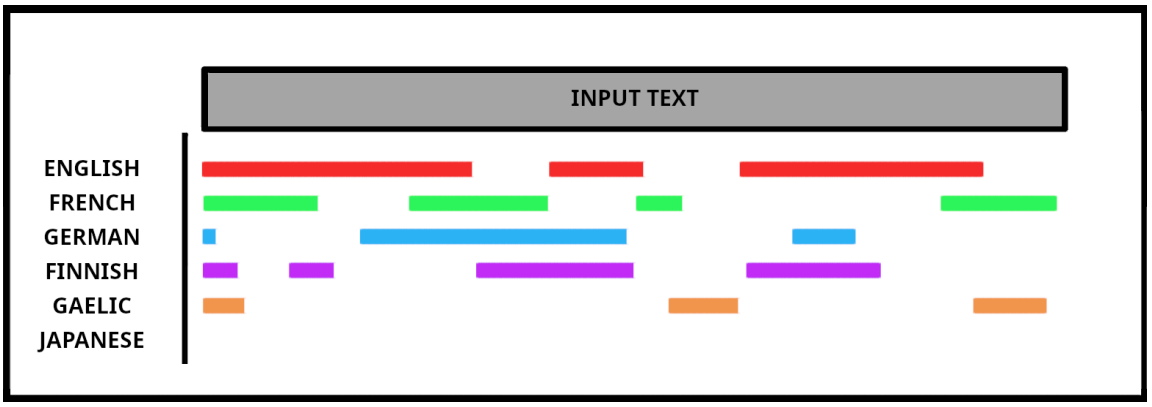

Fig 1: program output example

This program has 4 arguments:

- **--classes:** folder the class files (models) are stored in, folder context sensitive
- **--input:** the name of the file whose cost will be analyzed
- **--window-size:** the size of the window, default is **20**
- **--threshold:** the maximum average cost of a window to be considered a language, default is **3**

From inside the /bin/ folder, one can run the code in a similar way to **findlang.*py***:

python3 locatelang.py <args>

Given a folder with language models and a file to scan, this program returns what intervals of the file may belong to each language. This is done by iterating a window through the file and checking if the average of the costs of the symbols in it are below a threshold.

This window is a list of consecutive symbols in the file, and it iterates by removing its current first symbol and adding the next symbol in the file not in the window.

locatelanggroups.py

This program is a variation of **locatelang** where models are grouped under a certain label.

Each label is associated with the **best** result provided by their associated models, with the total interval span being our chosen successfulness metric.

While we scanned a folder in **locatelang** this program instead uses a JSON file that associates each label with a list of model file paths.

This program has 4 arguments:

- **--groups:** JSON file containing model groups, folder context sensitive
- **--input:** the name of the file whose cost will be analyzed
- **--window-size:** the size of the window, default is **20**
- **--threshold:** the maximum average cost of a window to be considered a language, default is **3**

Other than the differences mentioned above this program is functionally the same as **locatelang**.

locatelangalt.py

This program is an alternative to **locatelang** that focuses on creating a consecutive chain of languages rather than providing plausible intervals for each one.

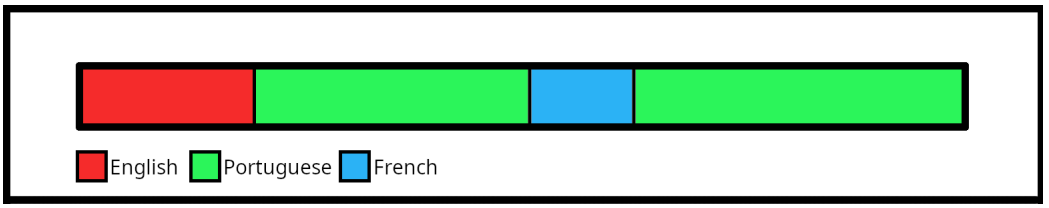

Fig 2: program split example

Due to this program being very computation intensive and accomplishing a different purpose from what was requested it was set aside and is not part of the core project.

This program has 5 arguments:

- **--classes:** folder the class files (models) are stored in, folder context sensitive
- **--input:** the name of the file whose cost will be analyzed
- **--intensive/low-memory:** toggles between keeping models in RAM or loading them whenever necessary, by default intensive mode is enabled.
- **--min-length:** minimum length of a language span, default is **50**
- **--max-default:** unknown character sequence length before exit, default is **7**

Given a text this program calculates cost according to all provided models, but stops doing so upon finding an unknown character sequence ***max-default*** times consecutively.

Once all values are calculated, models that didn't traverse ***min-length*** characters are filtered out and the model with the best cost/characters ratio is picked as the best.

The text is then offset by however many characters said model traversed and the process repeats until there is no text left.

It is possible for the min-length requirement to be waived if there are no qualifying models.

locatelanggroupsalt.py

This program is a variation of **locatelangalt** where models are grouped under a certain label.

Each label is associated with the **best** result provided by their associated models, with the cost ratio being our chosen successfulness metric.

While we scanned a folder in **locatelang** this program instead uses a JSON file that associates each label with a list of model file paths.

This program has 5 arguments:

- **--groups:** JSON file containing model groups, folder context sensitive
- **--input:** the name of the file whose cost will be analyzed
- **--intensive/low-memory:** toggles between keeping models in RAM or loading them whenever necessary, by default intensive mode is enabled.
- **--min-length:** minimum length of a language span, default is **50**
- **--max-default:** unknown character sequence length before exit, default is **7**

Other than the differences mentioned above this program is functionally the same as **locatelangalt**.

model_compiler.py

We preemptively compile files into models so that we do not need to process their respective text files multiple times and use this util program to do so.

This program iterates through all the files in a folder, compiles, and saves a model for each. These files can either be plain text or compressed using a XZ compression, and can be only partially compiled.

This program has 5 arguments:

- **--order:** order of the model, default is **3**
- **--smoothing:** smoothing parameter of the model, default is **0.1**
- **--folder:** folder containing the files to be compressed
- **--outputprefix:** the prefix of the compressed files
- **--fraction:** the fraction of the file to compress, default is **1**

Design Decisions

We read the input file in its entirety rather than in chunks and then analyze the extracted string as doing so simplifies the code.

For most purposes models are pre compiled into files, as this saves a lot of time.
This process also makes it easier to allow models with different smoothing and order values to be used together if the user so desires.

A text's final k characters are used as the start-up buffer for calculations.
We chose to do so because it makes the first k costs more consistent and is easy to use.

Results

To generate results we used the training and test files generated by this program**,** which was referenced by the teachers.

This program generates 1000+ training and test files, of which we used the first 100 training files and the first 20 test files (excluding the *Adyghe* language as that one generated faulty files).

The test files are further divided into:

- **full:** the whole file is used as the test, other types split this file to create their tests
- **tiny:** each file has a different 20-40 symbol test per line
- **short:** each file has a different 25-65 symbol test per line
- **medium:** each file has a different 80-120 symbol test per line
- **long:** each file has a different 120-200 symbol test per line

To help with the result generation we also made 3 programs:

- **mixer.py:** uses test files (besides full), to generate multi-language tests and save what language intervals are in each generated test
- **datagen.py:** gets results using all tests and stores them in a compressed JSON file
- **graph.py:** uses the file generated by **datagen** to create graphs for analysis

In the following graphs, when the values used are not specified, a model length fraction of 1.0, window size of 20, and threshold of 3.0 are used.

First we analyzed the outputs of **findlang** with varying model and target file lengths and confirmed that the smaller the file used for either the model or the target, the worse the accuracy is.

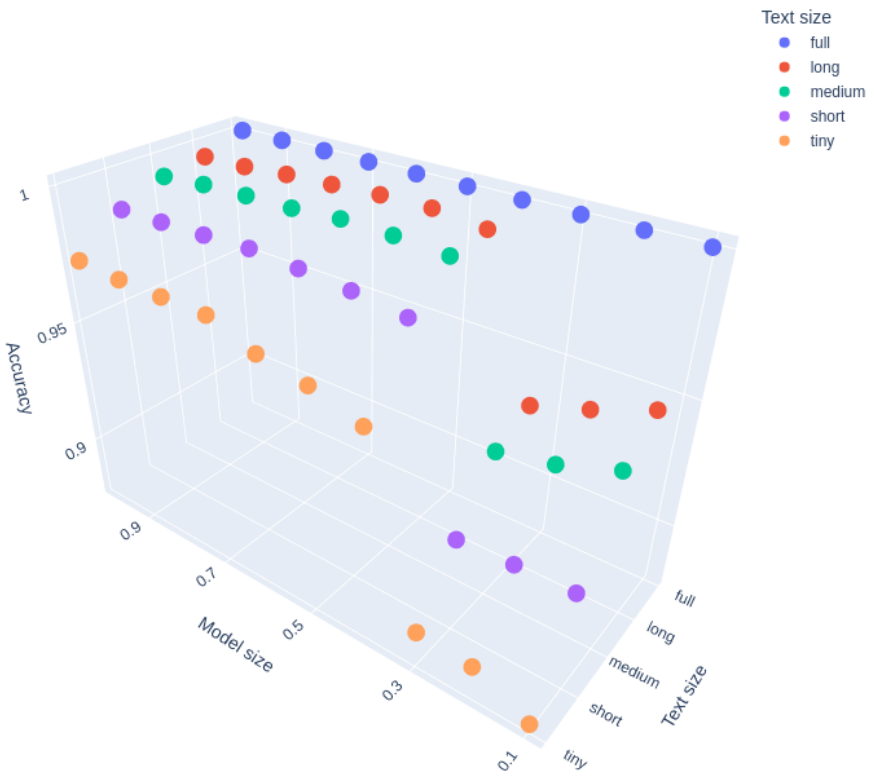

Fig 3: Accuracy per model and text size

As the accuracy isn't as simple to calculate for **locatelang**, we opted for visual analysis using varying window sizes, model sizes, and thresholds.

During this analysis we found out that the smaller the window size is, the more numerous and smaller the intervals found are. Also, as the intervals are from the start of the first to the start of the last valid window, too big of a window size can decrease the size of the intervals while still being accurate.

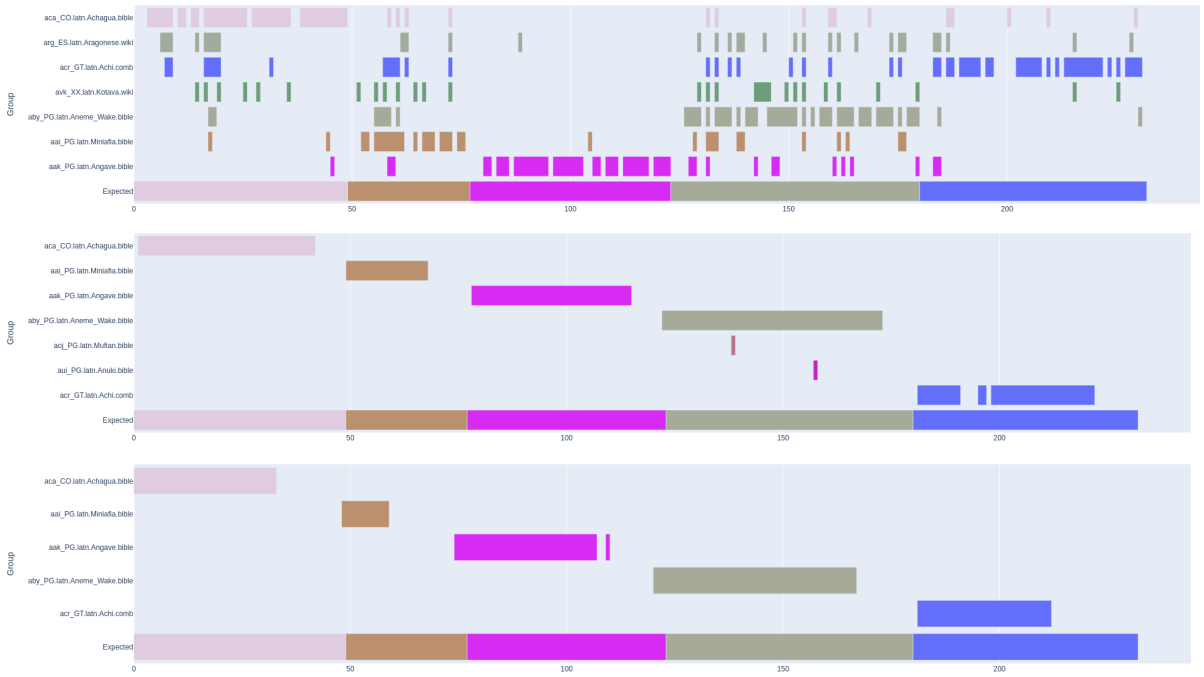

Fig 4: Language intervals with window sizes of 1, 10, and 20 respectively

We can also confirm what was found in the **findlang** analysis: the bigger the model size is, the more accurate the intervals found are.
This also causes the intervals to be bigger as to fit the expected ones better.

Fig 5: Language intervals with model length fractions of 0.1, 0.5, and 1.0 respectively

Lastly we find that the higher the threshold is, the bigger the intervals are, but unlike the model size increase, this doesn't always happen inside the expected intervals. Threshold has a sweet spot, when too high it decreases accuracy and when too low the correct intervals are smaller.

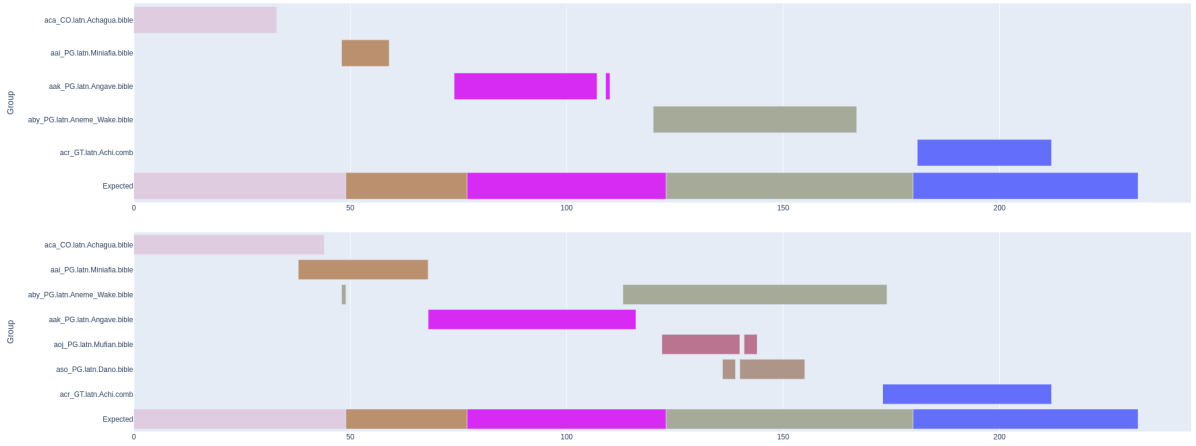

Fig 6: Language intervals with thresholds of 3.0 and 5.0 respectively

These plots and more can be found at **bin/report_tools/plots/**.