Moderation project - scansion algorithm

Aleksander Fedchin

December 4th, 2017

1 Introduction

In Latin poetry, hexameter can have different forms. Each of the first four feet can be either a dactyl or a spondee, which makes there be 16 primary types of hexameter (Table 1.1). These types can be further divided into subtypes based on how the breaks between words are distributed within the line, presence of elision, etc. Different poets preferred different versions of hexameter and the study of these preferences can be of value for determining the authorship or finding corruptions within the texts (e.g. [2][4]). A scholar attempting to conduct such a study by hand would be faced with a number of obstacles: (i) it might be necessary to scan tens of thousands of lines, (ii) it would be unfeasible to account for any possible subtype of hexameter, (ii) it would be difficult, though not impossible [2], to assess how the metric pattern of a line influences those of the following lines (iv) only one specific edition of the text could be chosen for the research. If there were a program that could scan latin meter automatically, all of these obstacles would be eliminated. In what follows, I propose an algorithm that can serve as a foundation for such a program. While developing the algorithm, I restricted myself from using any external databases. This restriction makes the problem more challenging and interesting from the Computer Science point of view, but is, of course, of little value for research purposes. The algorithm can be easily used with any meter whatsoever, although it was only tested on hexameter and elegiac couplet. The recall of the algorithm on elegiac verse was slightly worse than on hexameter (by around 5%), but, presumably, this can be fixed by taking into account the specifics of pentameter verse.

Dactyl =	long short short
Spondee =	long long
Trochee =	long short
Foot =	Spondee | Dactyl
LastFoot =	Spondee | Trochee
Hexameter =	Foot Foot Foot Foot Foot LastFoot

Table 1.1. The 32 versions of hexameter that my program takes into account. "long" means long syllable, and "short" means short syllable. The program does not specify, whether the last foot is a spondee or a trochee, even though it accepts both cases. The fifth foot is usually a dactyl, and this is why there are 16, not 32, "primary types".

2 Methods

Whether a certain syllable in a line is short or long depends on the length of the vowel, which can be long either by position (followed by two consonants with some exceptions) or by nature. Consequently, in some cases (around 57%), the program can decide on a specific type of hexameter (see Table 1.1) without any knowledge about the natural length of the vowels. From the lines that can be scanned this way, one can infer the natural length of the vowels in certain words and build a dictionary that can be later used to scan more lines. The chief obstacle is that of separation of the stem of a word from its ending that precedes placing the word into the dictionary.

Splitting a word into a stem and an ending is important, because, usually, the length of vowels in the stem of a word is preserved regardless of whatever postfix is attached to the end of that word. It is also important to distinguish the part of speech to which the word belongs, because the ending can have long or short vowels depending on the part of speech that it goes with. I refer to a part of speech with certain set of endings (such as 2^{nd} Declension or 3^{rd} Conjunction) as a "group". I did not account for every possible ending because the size of such a task would be overwhelming (in theory, one can use such tools as *Diogenes* for this purpose). I defined the following quantities to solve the problem of splitting a word into a stem and an ending:

$$
\begin{aligned}
P(\text{ending}|\text{group}) &= 1, \text{ if this ending is possible in this group,} && (2.1)\\
&\quad 0 \text{ - otherwise} && (2.2)\\
weight(\text{ending}) &= \frac{\text{number of groups}}{\sum_{g\in\text{Groups}}(P(\text{ending}|g)} && (2.3)
\end{aligned}
$$

Here, *weight* represents something like inverse document frequency of an ending (term) amidst the groups (documents). For each possible root, I store all words in the corpus that may represent inflections of this root (that are composed of this root and some legal ending). With $D_{r,g}$ being the set of all endings that are possible with a group g and have been seen with a root r, I define $S(\text{group}, \text{root})$ as:

$$\sum_{e \in D_{\text{group},\text{root}}} (\mathit{count}(e \text{ in root}) \times \mathit{weight}(e)) \tag{2.4}$$

And estimate $P(\text{group}|\text{root})$ as:

$$\frac{S(\text{group}, \text{root})}{\sum_{g \in \text{Groups}} (S(g, \text{root}))} \tag{2.5}$$

And then calculate the following probabilities:

$$P(\text{ending}|\text{root}) = \sum_{g \in Groups} P(\text{ending}|\text{group}) P(\text{group}|\text{root}) \tag{2.6}$$

$$P(\text{root}) = \frac{\text{number of possible inflections of this root in the corpus}}{\text{number of all possible inflections of all roots}} \tag{2.7}$$

$$P(\text{word}|\text{root}) = P(\text{ending}|\text{root}) P(\text{root}) \text{ where word} = \text{root} + \text{ending} \tag{2.8}$$

Finally, I split each word into an ending and a root by choosing the argmax of all possible $P(word|root)$:

$$\text{root of a certain word} = \ argmax_{root \in \text{ possible roots}}(P(word|root)) \tag{2.9}$$

This way I am able to find both the stem of the word and the part of speech to which this word belongs, and, consequently, the length of the vowels of its ending.

Another major problem for a scansion algorithm is to decide, where the letters i and u act as vowels, and where they should be considered to be consonants. I used Allen and Greenough's and Gildersleeve and Lodge's grammars [1][3] to tackle this and some other issues. However, most of the mistakes made by the algorithm (around 80%) are still due to the inaccurate separation of a line into syllables. For this reason, if I am to work on this project in the future, I would probably use a digitized Latin dictionary.

3 Runtime behavior

It would be difficult to evaluate the program's runtime behavior analytically. Nevertheless, it is clear that the runtime should increase with the number of words and with the average number of instances of a word in the corpus. For relatively small sample sizes (at least up to the size of the *Aeneid*), the runtime seems to be almost linearly dependent on the number of lines given as input. (Figure 3.1)

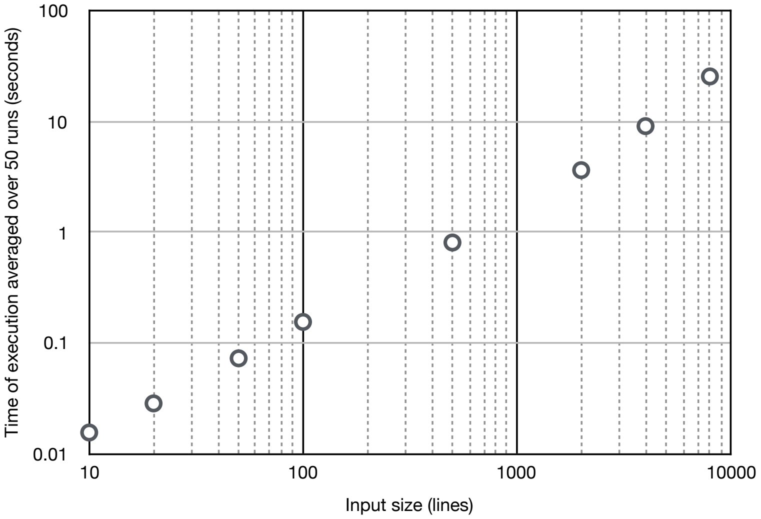
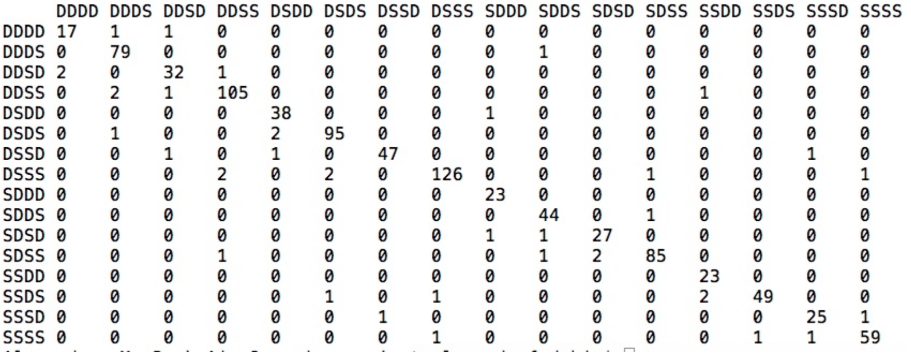


Figure 3.1. Relationship between the runtime of the algorithm and the size of the sample on which the algorithm was trained.

4 Performance evaluation

A testing set of 1000 scanned lines from the first four books of the *Aeneid* was kindly provided to me by Professor Ben Johnson (hexameter.co). The program was given those 1000 lines coupled with the rest of the *Aeneid* and the *Eclogues* and produced the result summarized in the following table:



```
     DDDD DDDS DDSD DDSS DSDD DSDS DSSD DSSS SDDD SDDS SDSD SDSS SSDD SSDS SSSD SSSS
DDDD 17   1    1    0    0    0    0    0    0    0    0    0    0    0    0    0
DDDS 0    79   0    0    0    0    0    0    0    1    0    0    0    0    0    0
DDSD 2    0    32   1    0    0    0    0    0    0    0    0    0    0    0    0
DDSS 0    2    1    105  0    0    0    0    0    0    0    0    1    0    0    0
DSDD 0    0    0    0    38   0    0    0    1    0    0    0    0    0    0    0
DSDS 0    1    0    0    2    95   0    0    0    0    0    0    0    0    0    0
DSSD 0    0    1    0    1    0    47   0    0    0    0    0    0    0    1    0
DSSS 0    0    0    2    0    2    0    126  0    0    0    1    0    0    0    1
SDDD 0    0    0    0    0    0    0    0    23   0    0    0    0    0    0    0
SDDS 0    0    0    0    0    0    0    0    0    44   0    1    0    0    0    0
SDSD 0    0    0    0    0    0    0    0    1    1    27   0    0    0    0    0
SDSS 0    0    0    1    0    0    0    0    0    1    2    85   0    0    0    0
SSDD 0    0    0    0    0    0    0    0    0    0    0    0    23   0    0    0
SSDS 0    0    0    0    0    1    0    1    0    0    0    0    2    49   0    0
SSSD 0    0    0    0    0    0    1    0    0    0    0    0    0    0    25   1
SSSS 0    0    0    0    0    0    0    1    0    0    0    0    0    1    1    59
```

Table 4.1. The confusion matrix for the scansion algorithm. The rows represent correct results and the columns - those given by the program

Identified correctly	874 (87.4%)
Identified incorrectly	39 (3.9%)
Multiple answers, among which there is a correct one	61 (6.1%)
Multiple answers, among which all are incorrect	3 (0.3%)
Failed to determine the meter	23 (2.3%)
Accuracy	$\frac{874}{874+39} \approx 96\%$

I also attach the corresponding confusion matrix (Table 4). It is important to note that the program can give multiple answers. It is better to have the program check all possible versions of hexameter, than to have it give the first version it found as some other scanning algorithms do [5]. My approach is both more straightforward to implement and more robust, because it ensures that no data is lost. The most likely answer among a few given can be chosen manually or by another (very simple) program.

As expected, the larger the input, the more lines can be ”identified” (i.e. the greater the recall is, Figure 4.1).

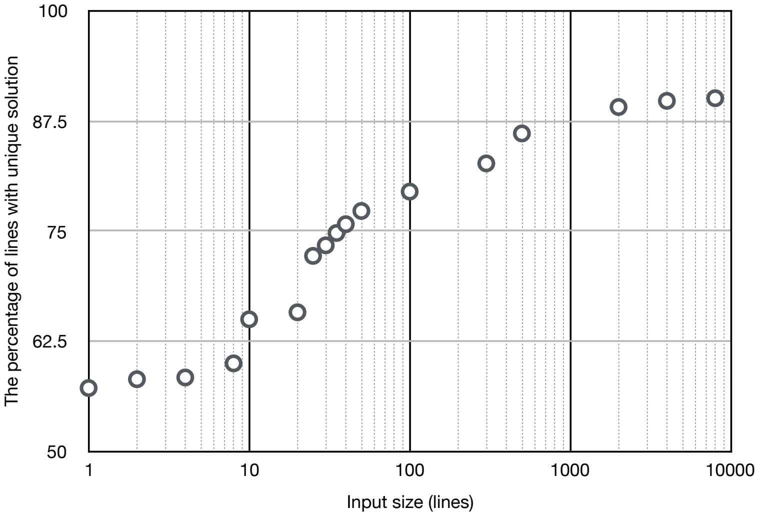


Figure 4.1. Relationship between the percentage of lines identified (averaged over 50 lines on different samples) and the size of the sample on which the algorithm was trained.

5 Applications and Conclusions

Based on the first five books of the *Aeneid* a language model was built (treating every line as a word and every identified sequence of lines as a sentence). Another language model was built from the first five books of the *Metamorhoses*. These training sets had approximately the same size (3852 and 3868 lines resp.). Comparing perplexities of the models on a sample, the program can decide on the authorship of the sample with some accuracy (Figure 5.1). Using bi- and trigrams did not give better results than using unigrams. Perhaps, the low precision of the algorithm p has too great an impact on the precision of constructed bigrams (p^2).

I plan to continue working on the algorithm by incorporating a digitized dictionary. This should lead to higher precision and better authorship detection. I am also going to test the algorithm on a variety of different meters.

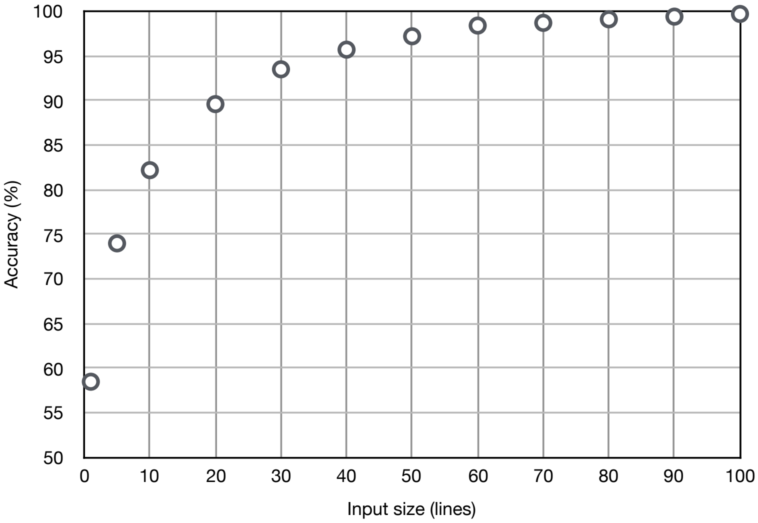


Figure 5.1. Relationship between the accuracy of the authorship classification algorithm and the size of the testing samples. Accuracy was averaged over 10^4 runs for each sample size.

References

[1] Allen, J.H., & Greenough, J. B. (1903). *Allen and Greenough's New Latin Grammar.* Boston. The Athenaeum Press.

[2] Duckworth, G. E. (1966). "Studies in Latin Hexameter Poetry". *Transactions and Proceedings of the American Philological Association.* 97:67-113

[3] Gildersleeve, B.L., & Lodge, G. (2005). *Latin Grammar.* London. Bristol Classical Press.

[4] Greenberg, N. A. (1987). "Metrics of the Elegiac Couplet". *The Classical World.* 80:4:233-241

[5] Papakitsos, E. C. (2010). "Computerized Scansion of Ancient Greek Hexameter". *Literary and Linguistic Computing.*