What's Wrong, Python? – A Visual Differ and Graph Library for NLP in Python

Balázs Indig[12], András Simonyi[1], Noémi Ligeti-Nagy[1]
[1]MTA-PPKE Hungarian Language Technology Research Group
[2]Pázmány Péter Catholic University, Faculty of Information Technology and Bionics
H-1083 Budapest, Práter street 50/A
{indig.balazs, simonyi.andras, ligeti-nagy.noemi}@itk.ppke.hu

Abstract

The correct analysis of the output of a program based on supervised learning is inevitable in order to be able to identify the errors it produced and characterise its error types. This task is fairly difficult without a proper tool, especially if one works with complex data structures such as parse trees or sentence alignments. In this paper, we present a library that allows the user to interactively visualise and compare the output of any program that yields a well-known data format. Our goal is to create a tool granting the total control of the visualisation to the user, including extensions, but also have the common primitives and data-formats implemented for typical cases. We describe the common features of the common NLP tasks from the viewpoint of visualisation in order to specify the essential primitive functions. We enumerate many popular off-the-shelf NLP visualisation programs to compare with our implementation, which unifies all of the profitable features of the existing programs adding extendibility as a crucial feature to them.

Keywords: Tools, Visualisation, Python

1. Introduction

In natural language processing (NLP), it is obviously fundamental to achieve good performance on the well-researched tasks; however, on a task currently under development, it is even more relevant to be aware of what have happened and where the program has gone wrong. The thorough analysis of the errors can lead to new ideas and methods, and finally, to better results. In most of the NLP tasks, this error inspection is carried out manually, while the number of tools supporting this work phase is low. The main reason of that is that for visual comparing there needs to be a good graphing library that displays output data correctly, which can be also hijacked to display visual differences in a form that helps human processing. Finally, a well-visualised good example can tell more of the inner-working of an algorithm than a thousand words or tables.

In scientific computing, especially in NLP, two programming languages serve as the source language of the majority of the new tools. JAVA is popular because of its maturity and cross-platform compatibility. In the job market, one can find many well-trained programmers developing in JAVA, the universities are still putting a great emphasis on teaching JAVA, and finally, the slowly evolving industrial applications are still using JAVA for new projects because of its adoption. However, Python is still gaining more and more popularity on the basis of its simplicity allowing fast prototyping and its ability to hide the performance of the critical parts – which are usually implemented in C++ – under a newbie-friendly Python API without sacrificing the overall running or development performance. The cheap work power of fast-trained Python programmers may turn the tide in the future. At the moment, main tools can be seen having both Python and JAVA APIs. Concerning the NLP visualisation task, there are mostly JAVA programs on the shelf, which are – strictly technically – not good as they can not be integrated well with Python programs. We stand with Python, therefore in this paper, we mostly sample Python programs along with a few others for comparison. We argue that the available solutions are not good enough for the future Python language NLP visualisation needs, therefore we created our own program in Python.

We describe the common NLP tasks of the past years to discover their common features, which are expected to be inherent in the methods and tasks of the following years. We gather the list of available tools to compare them and present our expectations for a new visual comparison program. Finally, we present our solution and our future plans.

2. NLP Tasks and Formats

The first *Conference on Computational Natural Language Learning (CoNLL)* was held in 1997 (Ellison, 1997). The first shared task, the *English noun-phrase chunking shared task* was announced in 2000 (Tjong Kim Sang and Buchholz, 2000) in a special format, the *CoNLL-2000 format*[1]. Later on, each year this format has evolved, extended and has become a de facto standard. The main idea behind the format is that each token is represented in a line of a text file and each of its features is represented as a field separated by a white-space – mostly a tabulator – character. The sentences are separated by a newline, and the order of the features has been standardised over the years. This so-called vertical format was a lightweight but machine-parseable alternative to XML by resembling / specialising the already existing *Tab Separated Values (TSV)* format[2]. The first field of each record (token) is always the actual token, the second is the lemma if any, and the third is the *Part of Speech*

[1]Ide et al. (2017) provides a broad overview of the state-of-the-art in standards development for language resources, including CoNLL and dependency annotations.

[2]A special hybrid version of TSV and XML format called *vertical file format* is used also in Sketch Engine (Kilgarriff et al., 2014) to represent corpora.

(POS) tag. The rest of the fields often differs and represents two kinds of structures.

One basic type of structure among tokens is *bracketing*. Many variants of bracketing task exist (e.g. NP-chunking, named-entity recognition) and many tasks can be interpreted as bracketing[3]. Basically, when a word or multiple subsequent words need to be labelled[4] – which implies bracketing – one can represent the task with the same data structure, and from our perspective, their visualisations are practically the same too.

Another basic type of structure being represented in various CoNLL formats comes from the further processing of the input sentence, where a one-level bracketing is insufficient. In *constituent parsing* immediate constituents are represented as brackets and it is needed to define higher-level relations between them to get the full parsing. These relations between the tokens (or groups) are represented by edges having a direction and a label, optionally. In *dependency parsing*, directed edges are always used and no artificial constituents are introduced to name the underlying groups of tokens. These two techniques can be represented in the data uniformly by using references to the token number to denote the edges. The resulting table cannot easily be read by humans but is practical for machines. This is the root cause why one would need a visual comparison tool.

There is a third type of commonly used structure which is not represented in the CoNLL format: the sentence alignment task. In the field of machine translation, tokens in a sentence should be reordered to get the correct translation. Sentence aligner tools also make mistakes, which, again, must usually be investigated by a human. In order to compare the alignments, one must see the two alignments with the matching and differing edges. Not to mention when one needs to see which part of the source sentence is translated to which part of the target sentence. This latter task involves bracketing and the alignment edges in conjunction.

We can see that most of the current NLP tasks involving supervised learning could profit from a visual comparison tool. A tool like this could help humans inspect the automatically generated output in view of the gold standard and increase the productivity of error analysis.

3. Traditional NLP centric visualisation and visual comparison tools

In this section we list the most important data visualisation libraries, focusing on their strength and weaknesses. There are a plethora of general data visualisation tools, that support many languages and are able to create interactive and publication-ready figures as well. In this paper, we sample only a few that suppose to shed light on the classes of the available programs regarding their good and bad sides.

In LaTeX there are many libraries that support general, linguistic and NLP focused graphing for vector-graphical publication-quality images, which use *PGF/TikZ* or other macros. Their common weakness is that they are optimised for printing and do not support interactive workloads; however, they have a nice output and are excellent for their specific task and easy to learn for LaTeX experts.

From the viewpoint of layout, one must consider *GraphViz* (Gansner and North, 2000) as it has advanced layout properties and can be customised with the standard DOT graph description language format for input, and it supports various output formats as well. It is, however, designed for non-interactive use and lacks the features needed for real interactive usage with a layout not easily customised.

As the de facto NLP framework for Python is *Natural Language Tool Kit (NLTK)* (Bird et al., 2009), it is a good starting point for searching for a proper base implementation. Its data structures and format handling capabilities are exceptional, but its interactive visualisation part is less extraordinary. It uses *GraphViz* for dependency graphs and a TKinter GUI for displaying the generated SVG figures in a cross-platform compatible way. As it lacks the layout code, only its input format and corpus handling capabilities are useful for our purpose.

In recent years, a new NLP framework in Python emerged from NLTK, which is called *Spacy* (Honnibal and Johnson, 2015) and it is gaining more and more popularity because of its speed and performance. It is a full-fledged NLP framework with a beautiful, modern, web-based visualisation module called *Displacy*. From our point of view, its main problem is that it does not support comparison and it has a heavy weight. Even if one would decouple the visualisation part and build a whole new comparison framework, the maintenance burden of the decoupled part would be too high because of the pace of Spacy's development.

Another novel emerging web-based visualisation library in Python is *Bokeh* (Bokeh Development Team, 2014), which is basically an interactive visualisation program designed for browsers around *D3.js* (Bostock et al., 2011) [5]. Its main goal is to release the users from the need to fiddle with JavaScript: it can be entirely customized from Python. The main problem of *Bokeh* is that it is not NLP centric and too general, and its API is rather abstract to be used directly. However, it can be used as a low-level graphing API for a custom layout engine as well as a basic WebUI.

We must mention the class of classic GUI-based or headless standard graphing APIs. The emerging *Matplotlib* (Hunter, 2007) is a great plotting API for Python, which is easier to interact with from Python than the other alternatives like *GNUPlot*[6]. These programs are able to create publication-ready figures in numerous supported exporting formats with the standard low-level primitives and also can be used as a base for interactive tasks as well. It is easy to write the NLP centric primitives with the standard ones and use the layout engine for our purposes.

For the specific task of NLP centric visual diffing we have found two off-the-shelf solutions, which are implemented in JAVA, and nowadays are not actively developed: one example is *MaltEval* (Nilsson and Nivre, 2008), which has

[3] For example, any kind of tokenisation task (e.g. paragraph, sentence or token splitting) can be interpreted as bracketing.

[4] Even POS-tagging can be interpreted as a labelled bracketing of each individual token.

[5] The goal is very similar to the *Computable Document Format* interactive demonstrations capability of the proprietary Wolfram's Mathematica, which can be tried at `http://demonstrations.wolfram.com/`

[6] `http://gnuplot.info/`

the typical anti-features we had faced with the aforementioned programs as well as they are too well-embedded into a framework: they cannot be hijacked for any purpose other than their well-defined role, which makes their usage for a slightly different task impossible, thus condemning them not to be used by a greater audience. It is hard to decouple them into modules to accommodate novel representations, thus forcing the user to reimplement much of the inner-workings, which is mostly a duplicate effort.

The other off-the-shelf example for the specified task is *What's Wrong With My NLP?*[7]. It has an unpleasant, old-style *graphical user interface (GUI)* (see Figure 5), which is considered clumsy and the rendered figure is a raster image, which can only be exported to EPS format. The internal representation is very promising from the aspect of extension to new formats of features. It is the best program currently available for our purpose.

As one can see, we checked multiple programs which could bring us closer to our goal: a lightweight library written with toolbox-philosophy in mind ready for versatile NLP-centric graphing and diffing in order to create a program, which could provide a human-friendly interactive comparison solution easily extendible to handle new formats and different layouts. We have found many well-defined or too abstract off-the-shelf implementations, that would have placed a large burden on us considering later development. Hence, we started to design a new implementation in Python with the followings in mind: (a) we tried not to constrain the edge drawing at all in our program, and (b) we tried to mix all the good features of the aforementioned programs in order to gain larger audience and satisfy the standard and non-standard comparison and visualisation needs as well. Some of the features were previously unavailable in any of the existing programs as their authors concentrated only on the standard NLP tasks. Our implementation is very close to *What's Wrong With My NLP?*. however, our goal was to use the resulting program in a pure Python environment and thus it only served as the base of our new Python implementation[8], which we will present in the next section in detail.

4. Our Solution: `LibWhatsWrong`

As described in the previous section, there is no real off-the-shelf solution in Python, which would help the interactive visual comparison of data and the visual, step-by-step debugging of a program's result with the possibility of exporting the resulting figures in a web-compliant format, that may also be embedded into scientific papers. The only similar solution implemented in JAVA has its own problems with the old-fashioned GUI and nowadays it is not actively developed.

Our goal is web-compliance in conjunction with the support of printable formats. To get started we have chosen the W3C standard *Scalable Vector Graphics (SVG)* as the final format, as there are many lightweight libraries to generate figures in this format in many languages. We selected *SVGWrite* [9] for the initial implementation. We designed an abstraction for the needed logical primitives as we wanted to be able to draw the standard drawing primitives with different drawing backends, such as *Matplotlib* or *Bokeh*.

We wanted to separate the input formats and corpus handling from the layout code, to encourage the potential users to implement even their really own format or utilise some other implementation of parsers of the widely-known data formats, for example, the ones in *NLTK* with our exposed API. This decision makes it possible in the future to quickly write a parser in Python for any specific format – based on our example implementations – and use it as a replacement. Between the input data – which is parsed into a unified form – and the description of the logical primitives for the drawing backend, lies the layout and the comparing code. Our implementation of layout generation and comparing is based on *What's Wrong With My NLP?*, with modifications to allow more control to customise the layout. One can create custom layouts easily by combining the layout primitives (which includes dependency, span and alignment layouts) or create new layouts from lower level logical or standard primitives.

The comparing process is fully separated from the layout computing. Numerous features of the primitives, which have a large impact on the complete laid out figure, can be set as parameters such as colour, font type, size, etc. This design of the core library allows one to use each step separately – even without the comparison – with his or her data in any supported or future format either from the graphical, command line or web UI or even from a Python program via an API. This fine-grained control was previously not available in *What's Wrong With My NLP?* or in any of the aforementioned solutions.

In our implementation it is also possible to have multiple edges from the same starting element to the same ending one or to form a loop by returning to the starting token, which is not a trivial requirement as it cannot be found in any of the aforementioned implementations[10] (see 5.. section for ideas and possible applications). The absence of this feature is caused by the fact that a well-formed dependency syntax tree cannot contain such edges, however, these types of errors are the ones which would be essential for a human corrector or may be required for a non-standard representation (see 5.. section for details).

The laid out figures can be exported to SVG, PDF and EPS formats currently and the range of the supported formats can be widened with the usage of *Matplotlib* as backend. To demonstrate the power of our implementation, we implemented a *GUI* around the library, that has all the features of *What's Wrong With My NLP?* – including interactive modification of the resulting figure –, but with a modern, sound GUI based on Qt 5 in PyQt (see Figure 6). There is also a simple demonstrative CLI for the basic drawing and comparing. Besides that, we plan to use *Bokeh* as UI and

[7] `https://code.google.com/archive/p/whatswrong/`

[8] We must note that with the proper knowledge of JAVA and the available libraries our main feature differences can be easily backported to *What's Wrong With My NLP?*to get a fully functioning future-proof NLP centric graph library and differ tool for JAVA.

[9] `https://github.com/mozman/svgwrite`

[10] The CoNLL formats do not support double edges and for example *Displacy* draws the two edges on each other

drawing backend to gain web UI functionality. In the next section, we present some useful ideas and possible applications for our library.

5. Possible applications

In this section, we would like to demonstrate the novelty and practicality of our application through four visualisation examples. We also enumerate some ideas regarding how to use our program, which might be useful for others in the future.

Although CoNLL format does not support double edges at all, and other applications simply draw two edges on each other if needed, correctly and visibly visualising two edges between the same two words is not only important because of our need to use the visualisation tool to improve the algorithm behind the given drawing, but also because there may be some parsing systems that specifically require and produce double edges. From a linguistic point of view, there are special relations between the constituents of a sentence that should be explored during the parsing: *focus*, for example, is a grammatical category that determines which part of the sentence contributes new, non-derivable, or contrastive information (Halliday, 1968). In Figure 1, we demonstrate how double edges – here an edge between the verb and its focus, and an edge between the verb and its object – are represented in our visualisation tool.

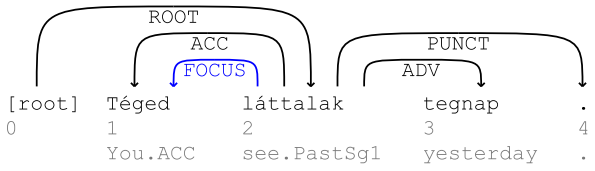

Figure 1: The visualisation of the dependency parsing of the sentence *Téged láttalak tegnap.* 'I saw YOU yesterday.'. The example demonstrates how our tool visualizes double dependency-like edges: one edge `ACC` starts from the verb *láttalak* and ends in its object, *téged*, while the other edge between the exact same tokens, `FOCUS`, means that the object here is the focus of the sentence (represented by the verb as the root). The two edges are drawn separately, and with different colours (blue and black), which indicates that the blue edge represents a connection on a different parsing level.

A loop – when an edge returns to the same token it started from –, is not part of the dependency formalism. However, some of the aforementioned tools support its visualisation, others do not. Drawing a loop is part of our solution, as we claim that it has an importance in parsing. For example, in some languages, like in German or in Hungarian, preverbal particles form one word with the verb. Therefore, the edge between a verb and its particle must return to the same token it started from (see Figure 2).

'The task of applying tags to each token in a sentence consecutively is called *sequential tagging*. In general, the tagger tries to assign labels to (neighbouring) tokens correctly. The well-known special cases of this task include Part-of-Speech tagging, Named-Entity Recognition (NER) and

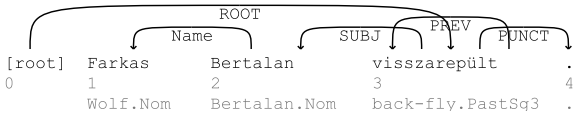

Figure 2: The visualisation of the dependency-like parsing of the sentence *Farkas Bertalan visszarepült.* 'Bertalan Farkas flew back.'. The example shows how our tool visualizes a loop. The edge `PREV` starts from the verb and ends in its verbal particle. In some languages, for example in Hungarian, the verbal particle and the verb often form one word, therefore the edge `PREV` must point to the same token it started from: the edge `PREV` connects the verb *repült* 'flew' and the verbal particle *vissza* 'back', but as they form one word the edge must form a loop.

chunking. In the latter, two *IOB tags* are used to determine a *well-formed one level bracketing* on the text.' (Indig and Endrédy, 2018)

Moreover, in languages, that do not have enough training data yet, or lack the specific programs to train a parser to do real syntactic parsing, chunking provides results which could substitute the parsing task in scenarios where only specific information is needed from the text. It is also essential for a linguist to see the specific examples and their frequencies in order to refine existing theories and recover annotation errors in the training set, which peculiarities have their own interesting aspects but cannot be harmonized with the automatic chunking process and its applications.

Figure 3 shows the visualisation of the chunking of a problematic Hungarian sentence. In Hungarian, noun phrases may end with an adjective even when followed by a noun that starts another noun phrase, despite the fact that the regular order of these categories within an NP is exactly `ADJ + NOUN`. Moreover, most of the adjectives can perfectly behave like a noun in the sentence. This is the reason why the chunking of the sentence in Figure 3 is not trivial: the word *cicaimádó* 'kitten-loving' is an adjective, but may bear the POS-tag `NOUN` because it often happens to behave like a noun in the sentences. However, in this case, this token is a true adjective being the modifier of the noun *író* 'writer'.

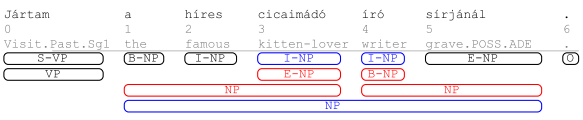

Figure 3: The visualisation of an NP-chunked sentence. The four lines under the sentence indicate two different chunking results in two forms: the first two lines show the differences in the `IOB` encoding and the last two lines show the differences in the chunks. The first and the last (blue) lines are the gold standards, and the middle ones (red) are the automatic results.

In machine translation, matching the corresponding words of the target and the source sentence is crucial for later steps. This task is called *alignment*. All of the source and target tokens must have their corresponding pair, but the order of the words can be different and multiple tokens can be

assigned to one and vice versa. The visual comparison of the gold standard and the automatic alignment (e.g. using different colours) is crucial to analyse errors in the input data and in the algorithm as well, in order to improve an aligning. In Figure 4, it can be seen how our tool visualises an alignment.

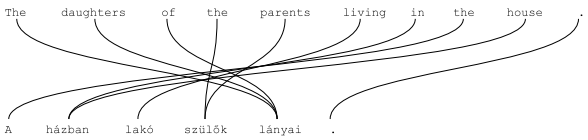

Figure 4: The visualisation of a sentence alignment. Some tokens have multiple corresponding pairs and vice versa.

Apart from the actual drawing and comparing, our program has an impressive feature set on the searching and filtering of the instances. One can filter by all the tokens and edge properties which can be a big help for the analyst to find, count, classify and correct errors. One can choose to create a visualized version on specific sentences, which matches some user-defined criteria in order to view a specific type of errors or sentences one by one or directly visualise the output of some parser like *Spacy* does.

6. Conclusion and Future Work

In this paper, we introduced a new NLP-centric visualisation and visual comparison library in Python, which adopts all the good features of the existing similar implementations while adding some novel key features so far not available in any of the current tools. Our idea was to keep the toolbox-philosophy in mind, write well-defined general modules and do not reimplement what is known to work. We also wanted to leave as much freedom as possible for the user to interact, (even dynamically) customise or extend the program for his or her needs even for non-standard formats or in-development purposes.

In the future, we want to use this program for multiple purposes, which involve interactive visual comparison and dynamic interactive visualisations. These applications will not only be interesting on their own but will also demonstrate the power of this library. We hope this program will be as useful for others as it is for us. The code is licensed under the LGPL 3.0 and is available at `https://github.com/ppke-nlpg/whats-wrong-python`.

7. References

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition.

Bokeh Development Team, (2014). *Bokeh: Python library for interactive visualization*.

Bostock, M., Ogievetsky, V., and Heer, J. (2011). D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December.

T. Mark Ellison, editor. (1997). *Proceedings of the 1997 Meeting of the ACL Special Interest Group in Natural Language Learning: Computational Natural Language Learning, CoNLL'97*. ACL.

Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233.

Halliday, M. A. K. (1968). Notes on Transitivity and Theme in English: Part 3. *Journal of Linguistics*, 4(2):179–215.

Honnibal, M. and Johnson, M. (2015). An Improved Non-monotonic Transition System for Dependency Parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378, Lisbon, Portugal, September. Association for Computational Linguistics.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

Ide, N., Calzolari, N., Eckle-Kohler, J., Gibbon, D., Hellmann, S., Lee, K., Nivre, J., and Romary, L. (2017). Community Standards for Linguistically-Annotated Resources. In Nancy Ide et al., editors, *Handbook of Linguistic Annotation*, pages 113–165. Springer.

Indig, B. and Endrédy, I., (2018). *Gut, Besser, Chunker - Selecting the best models for text chunking with voting*. Springer International Publishing, Cham. In press.

Kilgarriff, A., Baisa, V., Bušta, J., Jakubíček, M., Kovvář, V., Michelfeit, J., Rychlý, P., and Suchomel, V. (2014). The Sketch Engine: ten years on. *Lexicography*, pages 7–36.

Nilsson, J. and Nivre, J. (2008). MaltEval: an Evaluation and Visualization Tool for Dependency Parsing. In Nicoletta Calzolari (Conference Chair), et al., editors, *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, May. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2008/.

Tjong Kim Sang, E. F. and Buchholz, S. (2000). Introduction to the CoNLL-2000 Shared Task: Chunking. In *Proceedings of the 2nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning - Volume 7*, CoNLL '00, pages 127–132, Stroudsburg, PA, USA. Association for Computational Linguistics.

8. Appendix

To demonstrate the main – visible – differences between the JAVA version of *What's Wrong With My NLP?* and our Python implementation, we captured screenshots of the two applications during operation. As can be seen in Figure 5, the old version has an unpleasant, old-style GUI with numerous different windows often not being resizeable (for example the window named *Show Properties*). One of the biggest problems with this version is its clumsiness: the various windows are handled as separate units, therefore, for example, it is difficult to move them to the foreground one by one when they are covered by another window. Our implementation (see Figure 6), on the other hand, comes with a modern, clean GUI with easy to use sections grouped into tabs.

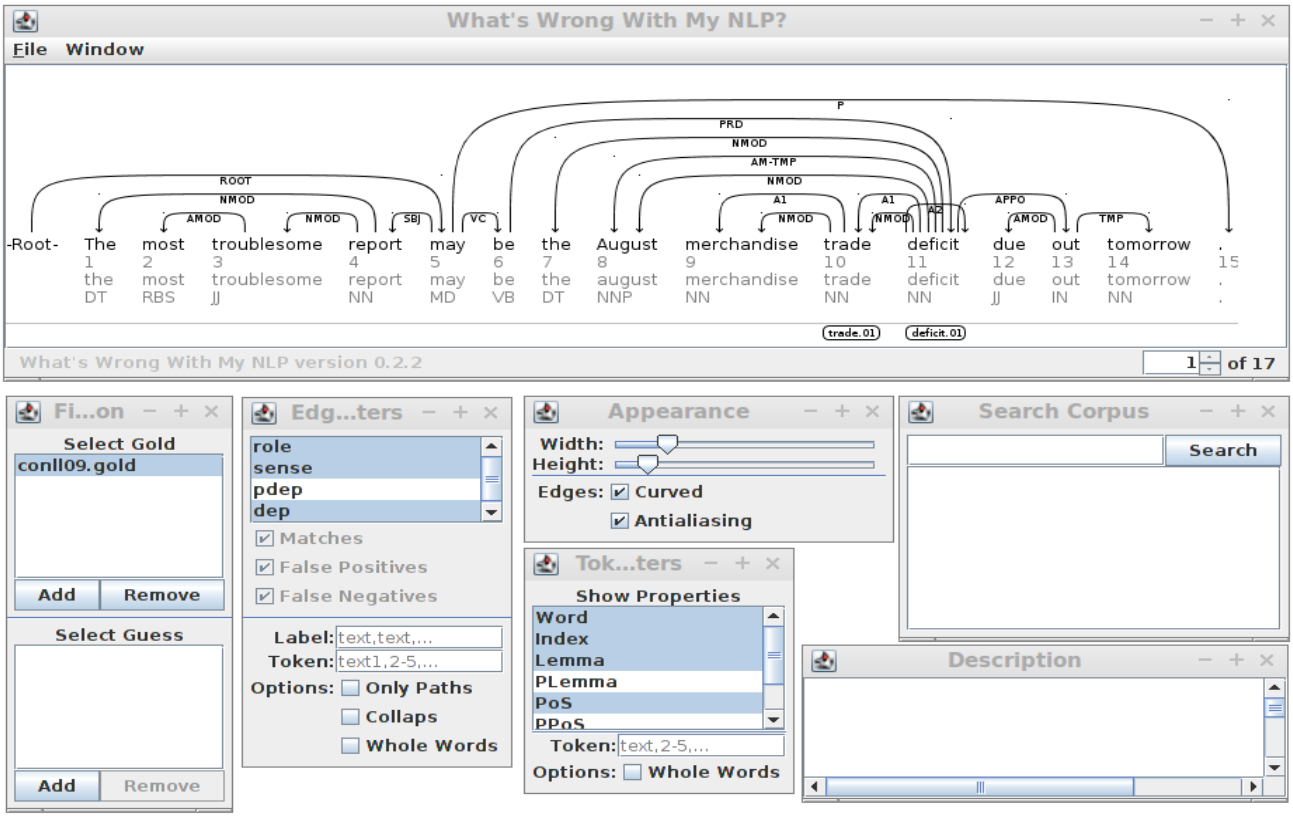

Figure 5: Screenshot of the JAVA version of *What's Wrong With My NLP?*

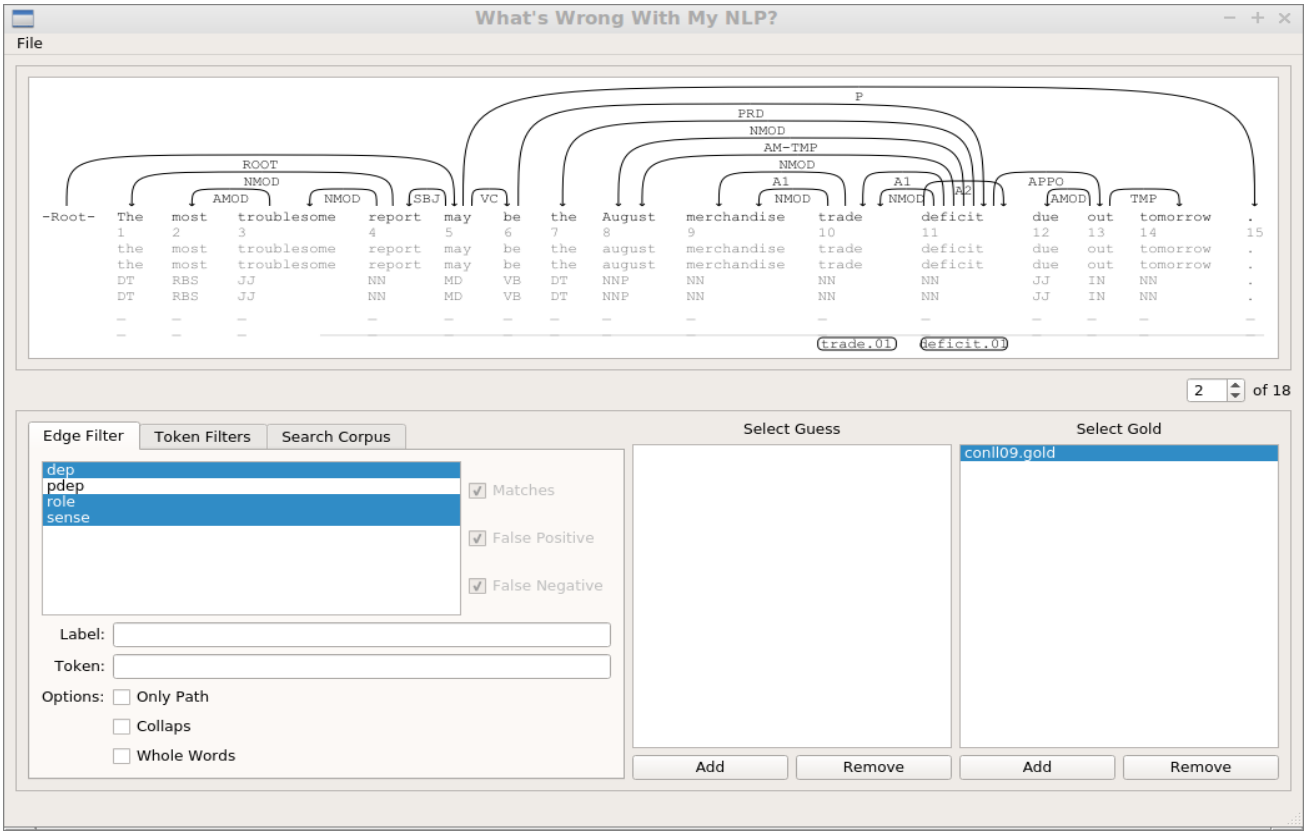

Figure 6: Screenshot of the Python version of *What's Wrong With My NLP?*