ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

Национальный исследовательский университет «Высшая школа экономики»

Факультет гуманитарных наук

Образовательная программа «Фундаментальная и компьютерная лингвистика»

КУРСОВАЯ РАБОТА

На тему «Интерпретация моделей искусственного интеллекта: мультиязычный пробинг выходящий из-под UD»

Тема на английском “Language Models Interpretation: Framework For Multilingual UD Probing”

Студент / студентка 2 курса группы № 203

Князькова Виктория Игоревна
(Ф.И.О.)

Научный руководитель
Сериков Олег Алексеевич
(Ф.И.О)
Старший преподаватель
(должность, звание)

Москва, 2022 г.

Table of context

1. Introduction

Modern language models show great results when it comes to text generation. They can complete sentences while maintaining correct grammar, style and meaning, generate whole coherent texts, understand questions and answer them like a human. For example, russian language model - YaLM, the descendant of Google's GPT-3 model, can do all of these: in Yandex.Referat [1] it generates a whole essay on selected topics, in Balaboba[2] it generates a really consistent text with a length of several paragraphs using only one or two lines of input and in Yandex search engine it sometimes generates answers to questions you ask. And if you read several texts generated by language models you can notice that most of them not only make sense but also are quite grammatical. That led to the question: if a language model can generate text considering grammar does it mean that model knows something about morphosyntactic properties of the language? If so, how to check it? One approach that can help in understanding what language properties are possibly encoded in language models is *probing.*

1.1. What is probing?

As soon as the model is pre-trained on some kind of task (e.g., predict the original value of the words replaced with a [MASK] token), we can use it to get word/sentence embeddings (i.e., vector representation of it), which are believed to capture important linguistic properties. In order to test this, we create so-called probing tasks, each of which highlights only one morphological or syntactic property of the language. After that we train an auxiliary classifier, which tries to predict for each sentence the answer to a probing task using only its embedding. If it succeeds, we have the right to say that this pre-trained model is storing readable information about the selected feature.

1.2 How to construct probing datasets?

If you want to apply a probing approach you need a sufficiently large dataset of annotated for target class sentences, so that is enough to train a classifier. Therefore, constructing sentences by yourself is not an option. One of the possible solutions is to extract sentences from some corpus, parse them and then somehow classify them with the help of linguists or some script. However, this solution is very language and task oriented and if you want to change something you have to start almost from scratch. That is why, I propose another way of constructing

[1] https://yandex.ru/referats/
[2] https://yandex.ru/lab/yalm

datasets, where almost all steps will be computer-aided, starting from sentence selection and their classification and ending with probing file creation and launching the classifier.

2. Related Works

In one of the most fundamental studies [Conneau et al., 2018], where the concept of probing tasks was first proposed, were presented 10 probing tasks that represent 3 different linguistics domains: surface information, syntactic and semantic information. The sentences were selected from Toronto Book Corpus, parsed by Stanford Parser and classified in some way.

However, my work is closely related to other probing studies [Miaschi et al., 2020] and [Mikhailov et al., 2021] in the way of creating datasets for probing tasks. In multilingual study about morphology [Mikhailov et al., 2021] authors investigate how morphosyntactic knowledge is distributed across the model for different languages. They create new probing tasks based on the detection of guided sentence perturbations and use Universal Dependencies treebanks for creating those tasks. In the Italian study [Miaschi et al., 2020] they also use Universal Dependencies for their experiments but have types of probing tasks closer to the original study [Conneau et al., 2018] even though the set of these tasks is more extended and covers more areas of linguistics.

3. Framework for creating probing tasks

To implement[3] the suggestion cited above, it is necessary to solve several tasks:

- Select some corpora with many languages presented and with the consistent annotation.
- Create a tool that transforms a set of sentences suitable for probing task into a file used by a classifier.
- Create a tool that filters sentences by probing task.

3.1. How about UD?

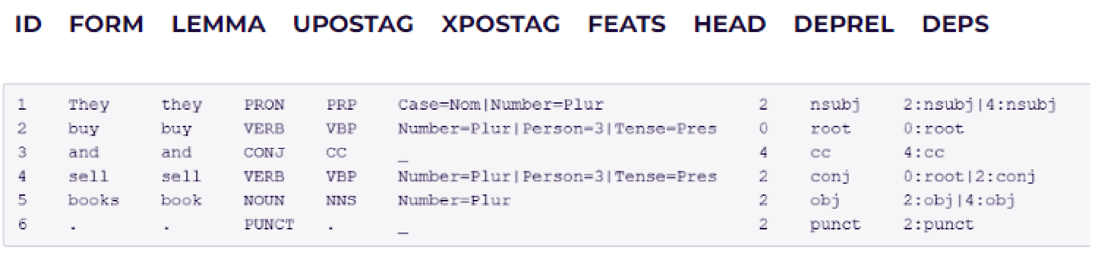
Universal Dependencies (UD) is a framework for consistent annotation of grammar (parts of speech, morphological features, and syntactic dependencies) across different human languages.

This project presents treebanks (datasets of parsed text with syntactic annotation) for more than 100 languages that are annotated with the help of fixed and universal tagsets. Hence this makes it possible to conduct probing of several languages at once and then even make cross-

[3] https://github.com/vknyazkova/UD_probing

linguistic comparison of the results. Moreover, these treebanks contain information about syntactic relations in the sentence while not all corpora do.

UD treebanks are encoded in plain text files with an “.conllu” extension. About conll-u format you can read more on UD website[4] but a very simple example is in the Fig.1 below:



```
ID   FORM    LEMMA   UPOSTAG   XPOSTAG   FEATS                              HEAD   DEPREL   DEPS

1    They    they    PRON      PRP       Case=Nom|Number=Plur               2      nsubj    2:nsubj|4:nsubj
2    buy     buy     VERB      VBP       Number=Plur|Person=3|Tense=Pres    0      root     0:root
3    and     and     CONJ      CC        _                                  4      cc       4:cc
4    sell    sell    VERB      VBP       Number=Plur|Person=3|Tense=Pres    2      conj     0:root|2:conj
5    books   book    NOUN      NNS       Number=Plur                        2      obj      2:obj|4:obj
6    .       .       PUNCT     .         _                                  2      punct    2:punct
```

Figure 1: The excerpt from a CoNLL-U formatted file

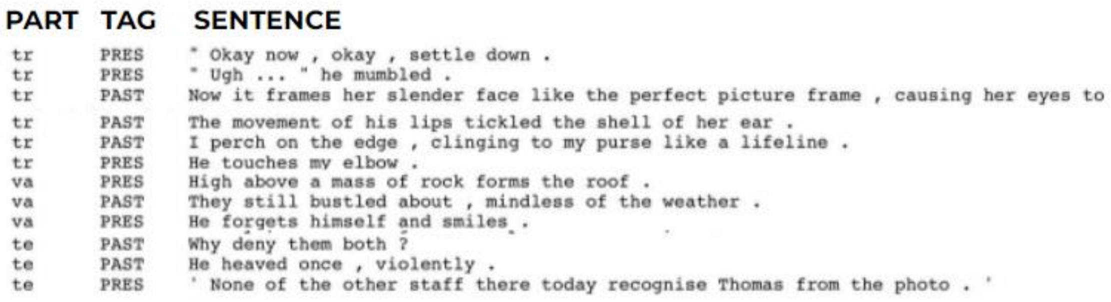
Here a sentence consists of several tokens. Each token is represented on one line, consisting of 10 fields. Fields are separated from each other by a TAB. The 10 fields are ID, FORM, LEMMA, UPOSTAG, XPOSTAG, FEATS, HEAD, DEPREL, DEPS (and MISC - any other annotation)

So according to the advantages described above and the fact that it is well-known among linguists and has a lot of ready-made tools to work with it, I decided to choose UD as a source of examples for probing tasks.

3.2 Probing file format and SentEval

So as soon as my goal is just to propose a different way of constructing probing datasets, I am not going to reinvent the wheel and plan to use already existing libraries that get sentence embeddings and train a classifier. But for them I need to turn suitable sentences from the CoNLL-U file into something that those libraries can take as an argument. Since almost all of those libraries are based on SentEval library[5] the format of an input file is the same: each line has 3 columns separated from each other by TAB. The fields are PART (train/validation/test), TAG (one of the probing task values) and the sentence itself (tokenized, tokens are separated by space).

[4] https://universaldependencies.org/format.html
[5] https://github.com/facebookresearch/SentEval/blob/main/data/probing/README.md



PART	TAG	SENTENCE
tr	PRES	" Okay now , okay , settle down .
tr	PRES	" Ugh ... " he mumbled .
tr	PAST	Now it frames her slender face like the perfect picture frame , causing her eyes to
tr	PAST	The movement of his lips tickled the shell of her ear .
tr	PAST	I perch on the edge , clinging to my purse like a lifeline .
tr	PRES	He touches my elbow .
va	PRES	High above a mass of rock forms the roof .
va	PAST	They still bustled about , mindless of the weather .
va	PRES	He forgets himself and smiles .
te	PAST	Why deny them both ?
te	PAST	He heaved once , violently .
te	PRES	' None of the other staff there today recognise Thomas from the photo . '

Figure 2: The example of SentEval probing file

3.3 Morphology probing tasks

First implementation of my idea was very simple sentence selection and classification based on feature's categories, that can be found in the column 6 (FEATS) of the conllu file. So, in this case probing tasks for some language can sound like "Does the model contain information about *Tense?" (you can substitute this grammar category for any represented in the language).*

However, putting the question this way could lead to ambiguity on deciding which class label should be assigned to the current sentence. For example, we are exploring *Person* category and we have a sentence, where there are multiple tokens with the same feature *Person* but with the different values of this category (like in *Fig.3*). So, in this case, we can either ignore all such sentences or make a decision which token should represent the class of the whole sentence. Choosing the first option we run the risk of selecting very few sentences from the conllu file, at least because of the redundancy when some features are expressed more than once (e.g. in agreement). With the second option we should somehow select the most central token in the sentence. One idea was to determine POS tags that represent each grammar category the best. For example, for *Tense* it could be *VERB*, and for *Gender* it could be *ADJ*. However, it won't necessarily work equally effectively in all languages, apart from the fact that the set of POS tags can vary from language to language. So, I decided to use information in the conllu file about syntactic relations and classify sentences by the token, which has a selected category and is the closest to the *root*. By doing this, we not only keep all the sentences with the selected grammar category but also annotate without relying on the peculiarities of the particular language.

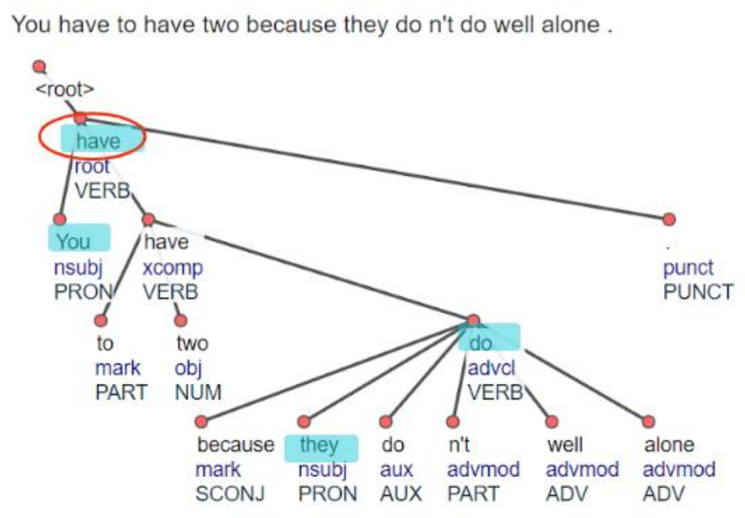


Figure 3: Probing task - Person. Tokens that have this category highlighted blue, the token that is selected to represent the sentence is circled red.

So, I wrote a small module that implements the algorithm described above and in order to get a file for probing (to use it in SentEval) you just need to import it and call the following function with the category name, path to your conllu file, and path to the output file as arguments:

```
def generate_probing_file(category, conllu_path, result_path,
                          partition=(0.8, 0.1, 0.1), shuffle=True,
                          random_seed=42):
```

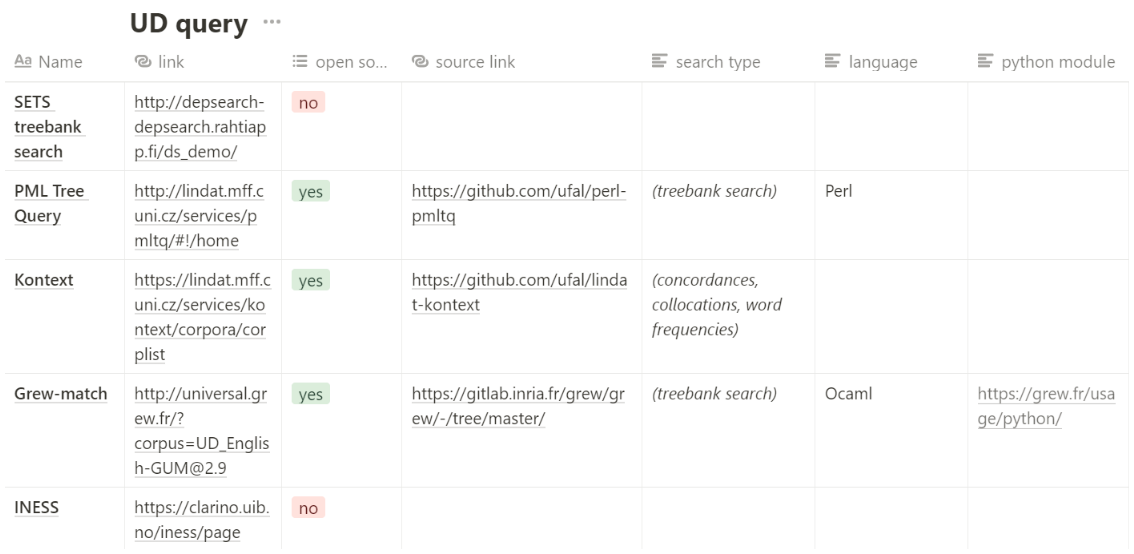
Figure 4: The parameters to function, that generates probing file

3.4 Customised probing tasks

The next step was to come with a tool that would allow you to construct datasets for more complex and customizable probing tasks. Because even very simple tasks from the original study [Conneau et al., 2018] like SubjNum or ObjNum need more than just classification by values of one single category. That is why our tool should be able to filter conllu file by syntax and morphology at the same time. But before developing an algorithm, it is necessary to decide on some metalanguage, with which you can create any custom probing task based on fields in the UD, in other words, come up with a query language for the UD treebanks.

But since the UD website[6] already has 5 different ready-made solutions I decided first to look at them and investigate if there are any possible ways to integrate any of them into my framework. For this reason, I was looking only for those that have an open source code (and it would be great if it was written in a programming language familiar to me) and those that make it possible to query syntax and morphology. After the comparison (see Table 1), there were only 2 options left: PML Tree Query and Grew-match. The first one, however, has such complicated query language that I couldn't figure it out. The second one turned out to be more intuitive and more similar to what linguists encounter when searching in corpora. Furthemore, it seemed more attractive due to its python module[7]. So, I decided to focus on Grew-match and explore its query language.

Table 1: Comparison table for UD queries



UD query ···

Name	link	open so...	source link	search type	language	python module
SETS treebank search	http://depsearch-depsearch.rahtiapp.fi/ds_demo/	no				
PML Tree Query	http://lindat.mff.cuni.cz/services/pmltq/#!/home	yes	https://github.com/ufal/perl-pmltq	*(treebank search)*	Perl	
Kontext	https://lindat.mff.cuni.cz/services/kontext/corpora/corplist	yes	https://github.com/ufal/lindat-kontext	*(concordances, collocations, word frequencies)*		
Grew-match	http://universal.grew.fr/?corpus=UD_English-GUM@2.9	yes	https://gitlab.inria.fr/grew/grew/-/tree/master/	*(treebank search)*	Ocaml	https://grew.fr/usage/python/
INESS	https://clarino.uib.no/iness/page	no				

3.4.1 Grew-match UD query

Overall, grew-match can search for a node, for a relation and add constraints on patterns. General ideas:

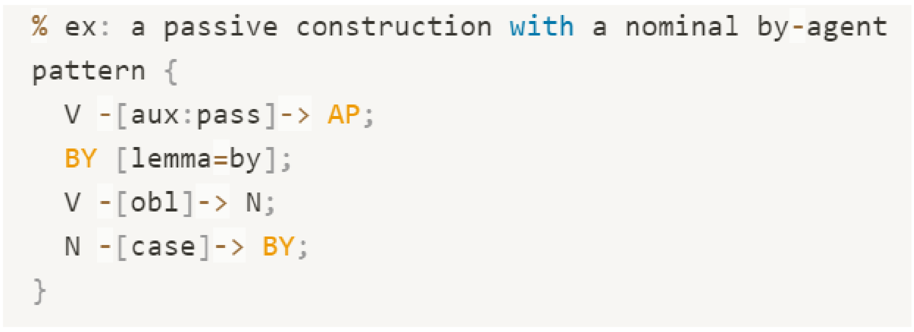
- Whole pattern is written inside curly braces {}
- Each node has its reference designation (for example *N*) and a pattern written inside square brackets []

[6] https://universaldependencies.org/#language-

[7] https://grew.fr/usage/python/

- The relations between nodes are written as the relations between nodes' reference designation.
- Relations have a direction.
- It supports disjunction of feature values and relations, negation and operator *in*
- It can check on equality or inequality of 2 features of 2 nodes.

For more details look for grew-match official tutorial [8]

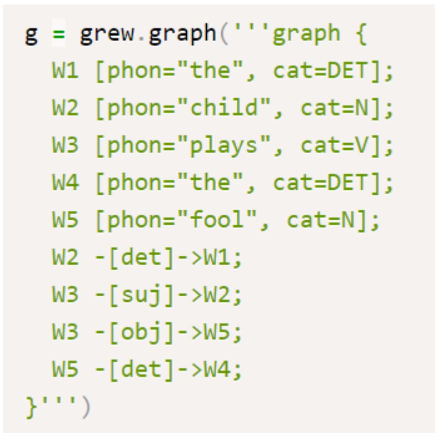


```
% ex: a passive construction with a nominal by-agent
pattern {
  V -[aux:pass]-> AP;
  BY [lemma=by];
  V -[obl]-> N;
  N -[case]-> BY;
}
```

Figure 5: An example of a grew-match query.

3.4.2 My implementation of grew-match queries

The initial plan was to integrate grew-match into my framework using its python module. However, this module turned out to be unsuitable for filtering the whole CoNLL-U file because its search function can search for a pattern in one graph only and this module doesn't have any built-in tool that can turn CoNLL-U file into grew-match graphs (like in Fig. 6)

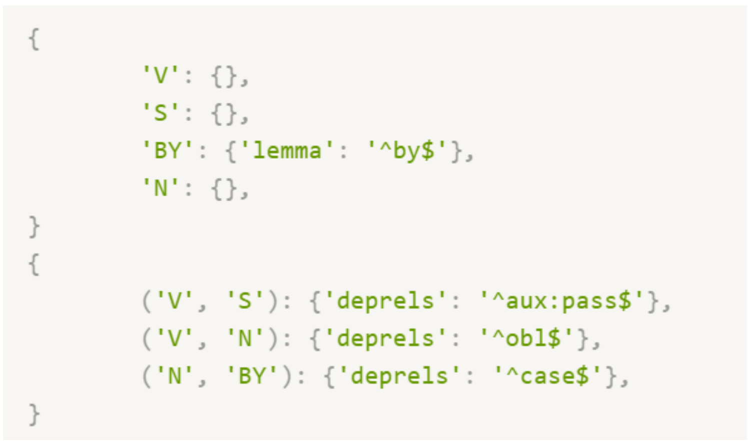


```
g = grew.graph('''graph {
  W1 [phon="the", cat=DET];
  W2 [phon="child", cat=N];
  W3 [phon="plays", cat=V];
  W4 [phon="the", cat=DET];
  W5 [phon="fool", cat=N];
  W2 -[det]->W1;
  W3 -[suj]->W2;
  W3 -[obj]->W5;
  W5 -[det]->W4;
}''')
```

Figure 6: An example of Grew syntax for graphs

[8] http://universal.grew.fr/?corpus=UD_English-ParTUT

That is why I decided to try to make my own function that filters CoNLL-U file by a given pattern. I also have to change the query language, because now query should be something that will be convenient for python to work with. I chose the format of python dictionaries and rewrote all grew-match queries in a new format (see Appendix). Below there is an example of the same query as in Fig. 5 rewritten in python dictionaries. The first dictionary is in charge of node pattern, the second contains all information about constraints between 2 nodes (relations, linear distance, equality/inequality of features).



```
{
        'V': {},
        'S': {},
        'BY': {'lemma': '^by$'},
        'N': {},
}
{

        ('V', 'S'): {'deprels': '^aux:pass$'},
        ('V', 'N'): {'deprels': '^obl$'},
        ('N', 'BY'): {'deprels': '^case$'},
}
```

Figure 7: Python dictionaries query (a passive construction with a nominal by-agent)

3.4.3 Filtering algorithm

The main idea is to check every sentence for compliance with all values in the pattern.

First, we are looking for all nodes in the sentence and if we can’t find at least one of them we skip that sentence. Next, we are checking for every constraint between every pair of nodes and again we skip the sentence if it does not obey at least one of the constraints. The final check is simultaneous compliance with all constraints performed by all node pairs.

That algorithm seems to be good (and relatively fast) at finding sentences that match the pattern, however it still should be thoroughly tested because at this point we can’t be sure that there are no FN (False Negative) sentences.

4. Results

In this section I will show what data can be obtained as a result of a probing tasks experiment, how it can be analyzed and what interesting findings I have got as the result of my morphology experiment.

4.1. The amount of effort required to conduct a large-scale study

First of all, I should pay attention to the convenience of my method of constructing datasets for probing tasks. In order to get data for all of this bar charts (Fig.8) you, as a researcher, only need to create a list of paths to 4 conllu files and a list of all available features in the language, then write a simple loop and inside this loop use generate_probing_file() function with elements from those lists as arguments. After that simple manipulations you will have a set of proing files to use in the SentEval library. There is no need to make preprocessing for every language (like parsing, extracting features etc.), we use the same steps for every language, which makes it easier to change the language of the probing experiment or the probing task itself.

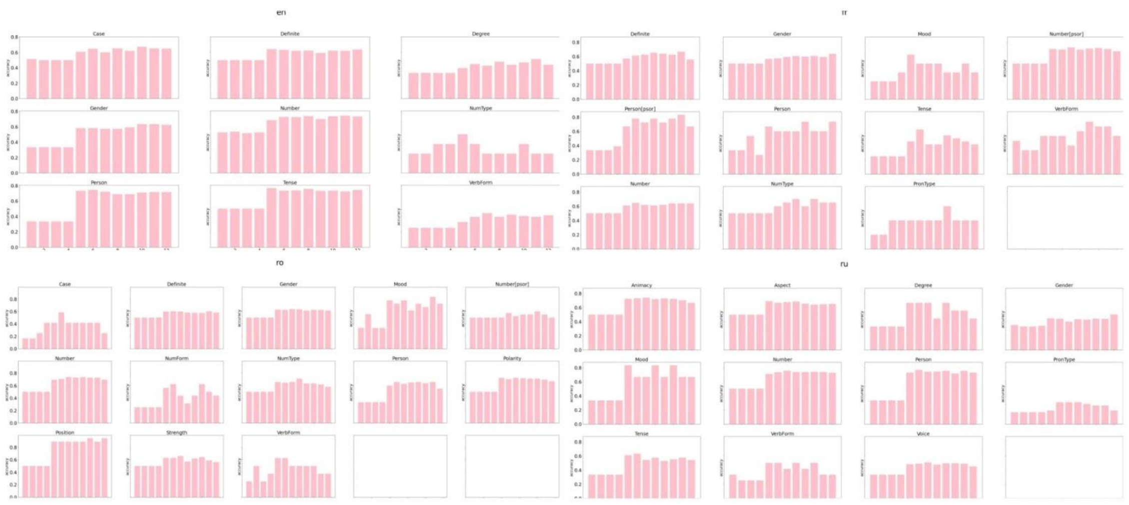


Figure 8: Examples of results of morphology probing experiments. Accuracy values on every layer of M-BERT model for every feature category available for every language (english, french, romanian and russian)

4.2 My probing results

Let’s take a closer look at my results. I used treebanks of 4 different languages: English, French, Romanian and Russian. From those conllu files I have generated probing files for every feature category presented in the language. Then I used the ruSentEval library to make probing experiments. I selected bert-base-multilingual-cased to evaluate its knowledge about these languages and “mlp”-classifier as a tool for these probing experiments. As a result of a probing experiment I got a json file with accuracy scores of every layer.

After that I made a very simple visualization for every language and feature (like in Fig.8) just in order to look at the whole picture and find some interesting patterns.

The first pattern that catches the eye is the fact that in most cases the first four layers have very low accuracy at a level of random guessing. While accuracy on higher layers is much better almost for every category.

The second one is that M-BERT ability to capture Person and Number information is almost the same for all of these 4 languages (see Fig. 9 and Fig. 10). However, french sentence embeddings from the 3d layer is an exception, for some reasons they seem to encode Person information on 3d layer better than other languages' embeddings (Fig. 9)

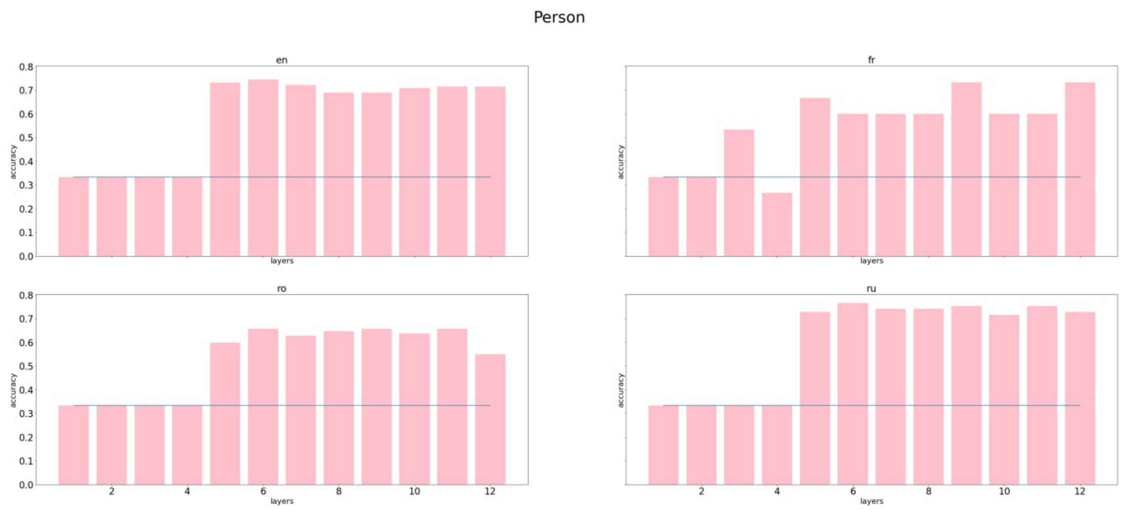


Figure 9: Accuracy scores obtained on every layer of MBERT with Person probing tasks. The blue line is the level of random guessing.

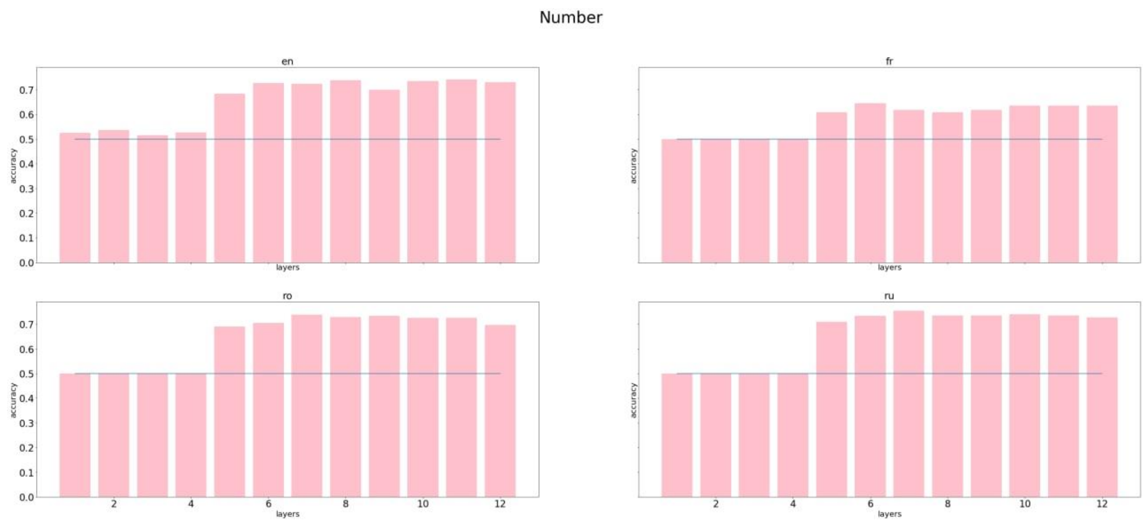


Figure 10: Accuracy scores obtained on every layer of MBERT with Number probing tasks. The blue line is the level of random guessing.

And another one interesting observation is that among all languages probing performance of Russian language is the lowest in relation to Gender.

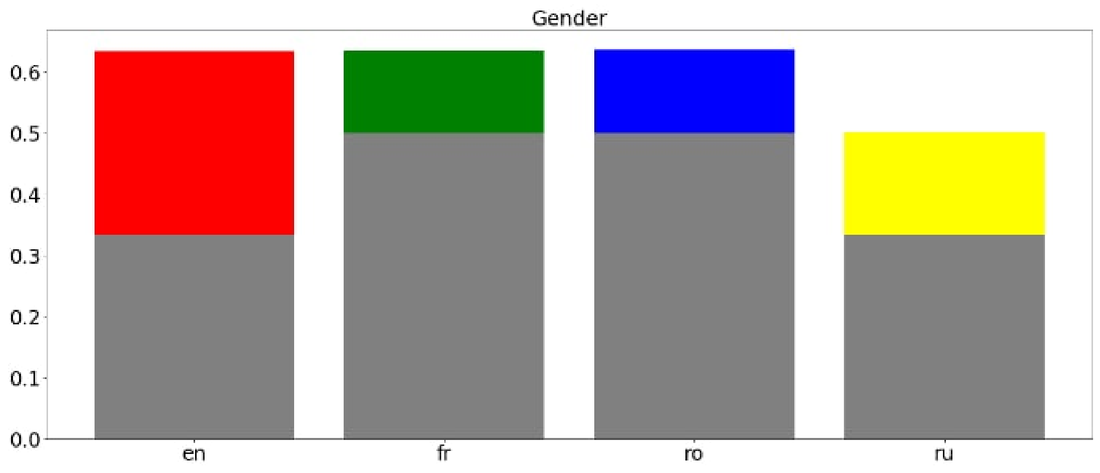


Figure 11: The best accuracy scores for Gender probing in English, French, Romanian and Russian. Grey columns on the chart represent the level of random guessing for every language.

5. Further work

First of all, I need to test my Filtering function for any False Negative results, because now it returns too few sentences even if a query is not very complicated.

And then, after I am sure that all my functions are working correctly, we are going to make some web application that will allow researchers to create probing files with minimal immersion in technical details. In order to do this, we have to come up with a GUI (graphical user interface), which should, for example, contain:

- Input file button (to load user's conllu file)
- Form template with node information (where user can select values from the list or enter his own regex) - like on Fig.8
- Some tool for drawing arrows between nodes and setting constraints on these arrows
- Button to add more patterns (each pattern is a single class in probing task) or to select option "this pattern vs the rest" (binary classification)
- Input element for every pattern form to set class label for the pattern

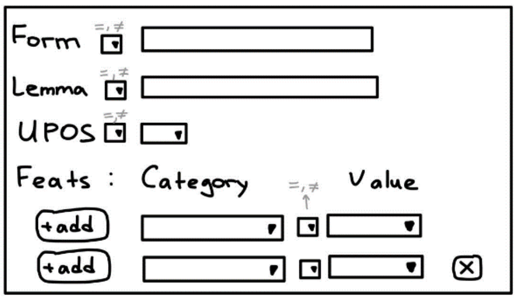


Figure 12: An example of node form for the web application.

6. Conclusion

In this course paper I proposed a new approach to constructing datasets for probing tasks experiments based on UD treebanks and applied one implementation of it to the set of 4 languages and made a comparison of the results obtained. Moreover, I started developing a framework for constructing datasets with customizable probing tasks, that can be useful in conducting experiments with more subtle probing tasks.

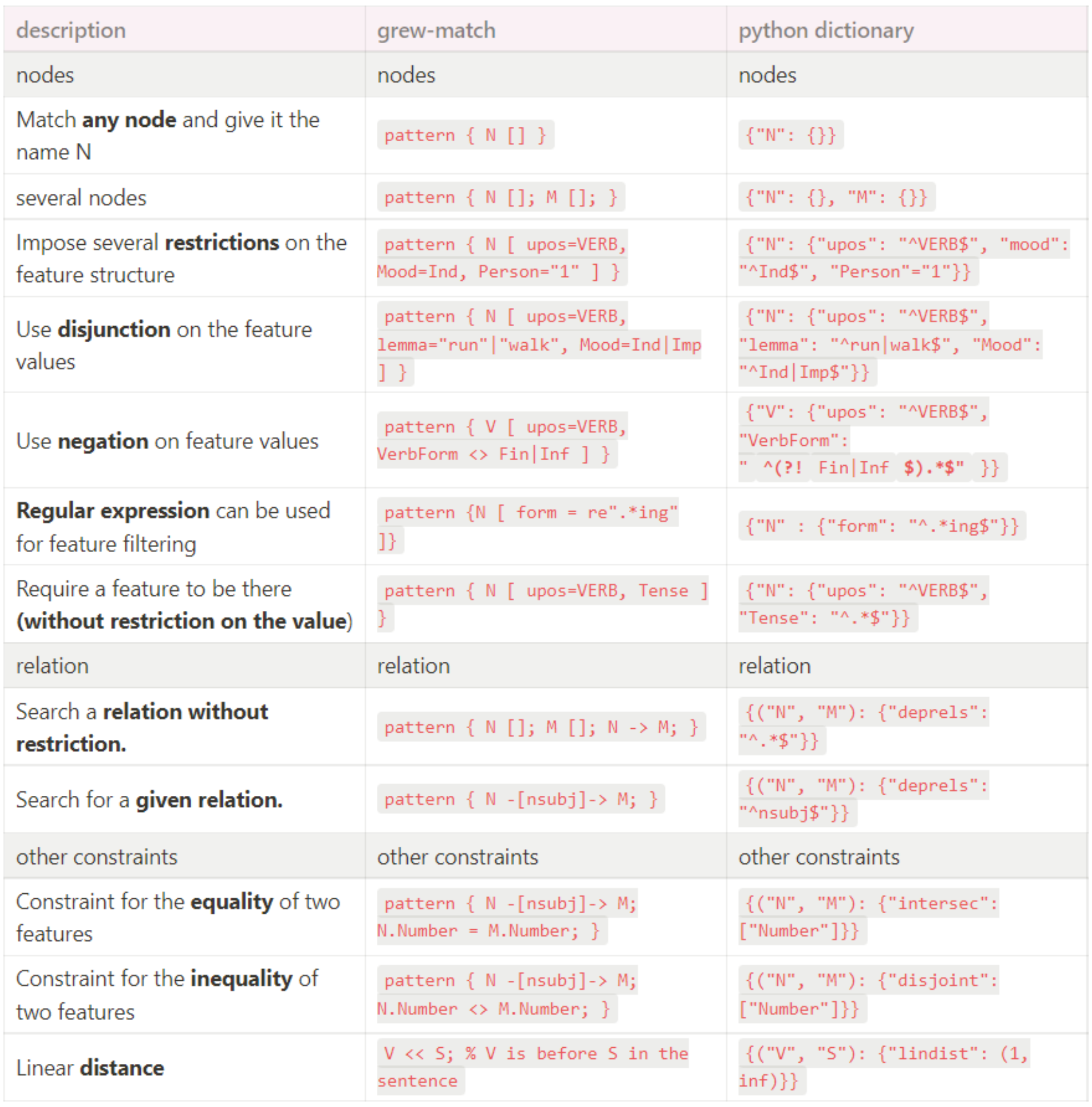
References

[1] - Conneau A. et al. What you can cram into a single vector: Probing sentence embeddings for linguistic properties //arXiv preprint arXiv:1805.01070. – 2018.

[2] - Miaschip A. et al. Italian transformers under the linguistic lens //Computational Linguistics CLiC-it 2020. – 2020. – C. 310.

[3] - Mikhailov V., Serikov O., Artemova E. Morph Call: Probing Morphosyntactic Content of Multilingual Transformers //arXiv preprint arXiv:2104.12847. – 2021.

Appendix



description	grew-match	python dictionary
nodes	nodes	nodes
Match **any node** and give it the name N	`pattern { N [] }`	`{"N": {}}`
several nodes	`pattern { N []; M []; }`	`{"N": {}, "M": {}}`
Impose several **restrictions** on the feature structure	`pattern { N [ upos=VERB, Mood=Ind, Person="1" ] }`	`{"N": {"upos": "^VERB$", "mood": "^Ind$", "Person"="1"}}`
Use **disjunction** on the feature values	`pattern { N [ upos=VERB, lemma="run"|"walk", Mood=Ind|Imp ] }`	`{"N": {"upos": "^VERB$", "lemma": "^run\|walk$", "Mood": "^Ind|Imp$"}}`
Use **negation** on feature values	`pattern { V [ upos=VERB, VerbForm <> Fin|Inf ] }`	`{"V": {"upos": "^VERB$", "VerbForm": " ^(?! Fin|Inf $).*$" }}`
Regular expression can be used for feature filtering	`pattern {N [ form = re".*ing" ]}`	`{"N" : {"form": "^.*ing$"}}`
Require a feature to be there **(without restriction on the value**)	`pattern { N [ upos=VERB, Tense ] }`	`{"N": {"upos": "^VERB$", "Tense": "^.*$"}}`
relation	relation	relation
Search a **relation without restriction.**	`pattern { N []; M []; N -> M; }`	`{("N", "M"): {"deprels": "^.*$"}}`
Search for a **given relation.**	`pattern { N -[nsubj]-> M; }`	`{("N", "M"): {"deprels": "^nsubj$"}}`
other constraints	other constraints	other constraints
Constraint for the **equality** of two features	`pattern { N -[nsubj]-> M; N.Number = M.Number; }`	`{("N", "M"): {"intersec": ["Number"]}}`
Constraint for the **inequality** of two features	`pattern { N -[nsubj]-> M; N.Number <> M.Number; }`	`{("N", "M"): {"disjoint": ["Number"]}}`
Linear **distance**	`V << S; % V is before S in the sentence`	`{("V", "S"): {"lindist": (1, inf)}}`