DrugBank Data Mining

Prediction of Drug Interactions using Inductive Logic Programming and Neural Networks

MIEIC
Programming Paradigms

Ângela Filipa Pereira Cardoso - 200204375 (angela.cardoso@fe.up.pt)

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, 4200-465 Porto, Portugal

July 18, 2017

Abstract

The aim of this work is to predict drug interactions in the Canadian drug database `DrugBank`. The fact that this database has both structured and unstructured information, compelled us to use two different learning tools. We use the Inductive Logic Programming tool `Aleph`, on the categorical fields. For the free text, we use the Neural Network `Word2Vec`. We will show that, although initial results where encouraging and showed a slight improvement when `Word2Vec` data was included, the final tests are extremely disappointing and there is no improvement from the `Word2Vec` data.

Contents

1 Introduction **4**

2 System Description **6**
2.1 Conceptual Description 6
2.1.1 Functionalities 6
2.1.2 Architecture 7
2.1.3 Programming Languages 7
2.1.4 Other Applications 9
2.2 Implementation 9
2.2.1 Implementation Details 9
2.2.2 Development Environment 10

3 Results **11**

4 Conclusion **13**

5 Improvements **14**

6 Resources **15**
6.1 Bibliography 15
6.1.1 Publications 15
6.1.2 URLs 15
6.2 Software 15

Appendices **17**

A User Manual **18**

1. Introduction

This work is part of an ongoing post doctoral research scholarship from the project NanoStima, within the research center CRACS from INESC TEC. It is joint work with Rui Camacho, Inês Dutra and Vítor Santos Costa.

`DrugBank` is an extensive Canadian drug database, containing semistructured information. The database format is `XML` and there are some categorical and some free text fields. As a part of the database, there is a classification of all the previously identified drug to drug interactions. For each drug there is a list of drugs with which there is some interaction, as well as a description of the type of interaction. Our main objective is to predict these interactions from the other information present in this dataset and maybe elsewhere. If successful we may extend our work to predict interactions by type, which should be extracted from the descriptive text of each interaction. Also, if we devise a method that does a good job of predicting drug interactions, any predicted interactions that are not in the database may be real interactions that are yet to be realized.

`Word2Vec` is a Neural Network, developed by Mikolov et al, that given a natural language text, transforms each word of the language into a vector in a high dimensional space, in such a way that words with similar contexts in the text are represented close to each other in the vector space. By context we understand the words surrounding (at a predefined distance) our objective word whenever it appears in the text. Our idea is that drugs with similar contexts should have some relation. In such a way that for example if drugs A and B are represented very close together in the vector space and drug A interacts with drug C, perhaps drug B also interacts with drug C.

`Aleph` (A Learning Engine for Proposing Hypotheses) is an Inductive Logic Programming system developed by Ashwin Srinivasan with input from several others. There are three files in an `Aleph` program: the background file, containing all the knowledge about our data; the positive examples file, which contains the examples we want to predict; and the negative examples file, which contains examples that are not supposed to be predicted by our theory. The basic `Aleph` procedure can be described in 4 steps:

- Select an example to be generalized.
- Construct the most specific clause that entails the example selected, and is within language restrictions provided, the "bottom clause".
- Find a clause more general than the bottom clause. This is done by searching for some subset of the literals in the bottom clause that has the "best" score.
- The clause with the best score is added to the current theory, and all examples made redundant are removed.

Our idea was to use `Aleph` for the structured part of `DrugBank`, while extracting information from the free text portion using `Word2Vec`.

2. System Description

2.1 Conceptual Description

2.1.1 Functionalities

The main purpose of the system is to train and test drug interactions using `Aleph` on the `DrugBank` database. For that there are several components and each has its own functionality:

- The `drugbank.py` module allows the user to prepare parts of the `XML` to be used with the `Word2Vec` software or with `Aleph`. It has several modes, namely:
  - `hyphens` - replaces the hyphens in the tags of an `XML` file with underscores, so that they will not interfere with `Prolog`'s interpretation of hyphens;
  - `names` - creates a file with the names of the selected drugs in `drugbank.xml` and another file with the same names concatenated, so that they can be seen as single words by `Word2Vec`;
  - `word2vec` - extracts the free text from the selected drugs in `drugbank.xml`, preparing it for `Word2Vec` to do its analysis;
  - `prolog` - extracts the parts of the selected drugs in `drugbank.xml` that are appropriate for introducing in `Aleph`, that is, the mostly categorical and structured tags, whose information is considered pertinent to the task.
- The `word2vec.py` module is used to turn the text obtained from `drugbank.py` into an array of vectors, each representing a word in the text. Several things can be configured in order to obtain the best results, such as the number of words that are considered part of the context of each word, the type of neural network to be used (`CBOW` or `Skip-Gram`), the number of steps of the network and the size of the vector space to use for the embeddings.
- The `filter.py` module filters `Word2Vec` data, selecting the words that belong to a provided list. We use it to consider only drug names in our final analysis.
- The `linkage.py` module is used to compute a hierarchical clustering of the word vectors produced by `Word2Vec`.
- The `cophenet.py` module is used to compute the cophenetic correlation coefficient of the clustering computed by `linkage.py`. This metric is used as a quality measure for the clustering process.

- The `word2vec-filter-500.ipynb Jupyter Notebook` was used to extract the actual clusters of words to be used in `Aleph`. They also allow us to visualize the Hierarchical Clustering.
- The `word2vec.pl` module transforms the clusters obtained from the `Jupyter Notebooks` into `Prolog` facts.
- The `drugbank.pl` module parses the `XML` file obtained from the `drugbank.py prolog` module into `Aleph` background, and positive and negative examples files.
- The `drugbank_test.pl` module is very similar to the previous one but was specifically designed to create sets of experiences in which one can train and test both with or without the `Word2Vec` clusters.

2.1.2 Architecture

Since the system modules and their functions have been described in the previous section, we present here, in Figure 2.1, the component diagram, with the interactions showing the intended order of use of each module. In other words, the arrows represent information flow from one component to the next.

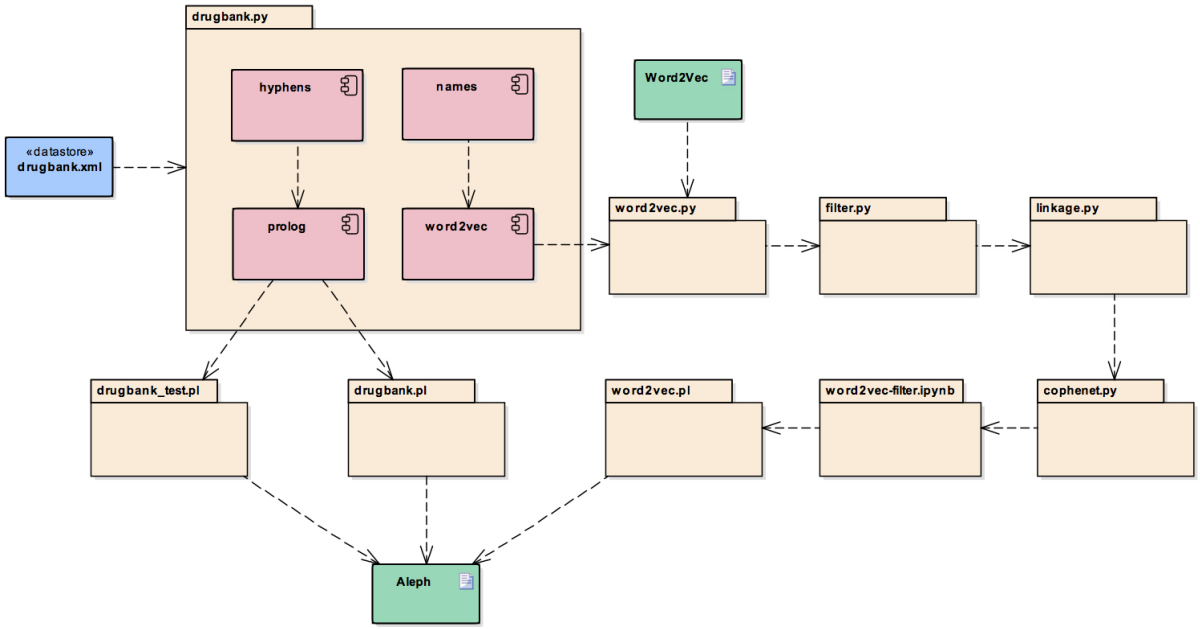

Figure 2.1: Component diagram with interactions

2.1.3 Programming Languages

Python

The first programming language chosen was `Python`. As seen in the description of the modules, we use it to work with the `XML` file, preparing it for `Prolog` and extracting text

portions for Word2Vec. It is also used to compute the word clusters that will be used in the final step as extra information for Aleph.

The main reason that lead to this choice was the fact that Word2Vec, as a part of the TensorFlow library, is available in Python. Also, Python is a language widely used by scientists, so it fits very well with this project, given its scientific nature. There are uncountable Python libraries and tutorials for all kinds of scientific purposes, which made this work much easier.

Python is a multi-paradigm programming language, depending on the use, it can be object-oriented, imperative, functional, procedural or reflective. As used in this application it was mostly procedural, because there was no real need for the structure of an object oriented approach. Also, the examples found online similar to our purpose where procedural.

Examples of other procedural programming languages are Fortran, ALGOL, COBOL, BASIC, Pascal, C, Ada and Go. The first of these are outdated and do not make very good choices. C is still very current and Go is actually quite new. With C one gets the advantage of having a faster application, at least assuming a good coding skill. In fact, since Python is interpreted instead of compiled, C can be much faster. However, it is harder to write C code and it is much less supported for our intended purpose. Of course speed can be an issue, as we will see in the discussion of the results, but in our case, the problem is in Aleph and not in Python, which can handle the necessary data in a timely fashion. As for Go, it is far to young to have any kind of libraries that would facilitate our work, we would have to build everything from scratch, so it made no sense to choose it.

Prolog

The choice of Prolog was mandatory once the decision to use Inductive Logic Programming as a learning tool was made. There are many other learning methods, but this one provides a theory that can be used as an explanation of the model it proposes. In this case, since we are trying to predict drug interactions, a theory could tell us the conditions that lead to drugs interacting with each other. This kind of explanation can be very useful in medical, financial and other settings, where even when the model guesses correctly, people still want to understand why.

There are several flavors of Prolog and we use two. The first one is YAP, which is used to run Aleph. This was also a very clear choice, since Aleph was made for YAP, where it has the most support and runs faster. However, to prepare the files for Aleph (in the modules word2vec.pl, drugbank.pl and drugbank_test.pl) we decided to use SWI-Prolog. There where two reasons for this. First, SWI-Prolog is very well documented and has several libraries that make certain tasks (such as importing XML files) much easier. Second, in their current versions SWI-Prolog is much more stable than YAP.

As far as we know, there are no alternatives to Prolog for Inductive Logic Programming, so the choice was just on which implementation of Prolog to use. The two used and considered seem the best for their respective jobs. Also, since they are both open source, they where preferred over others such as SICStus Prolog.

2.1.4 Other Applications

Python

As we discussed above there are many scientific applications of `Python`. The main libraries used by these applications are:

- `NumPy` - a low level library written in `C` and `Fortran` for high level mathematical functions (so it does not really count as a `Python` written application).
- `SciPy` - a library that uses `NumPy` for more mathematical functions and comes with modules for various commonly used tasks in scientific programming, including linear algebra, integration (calculus), ordinary differential equation solving and signal processing.
- `Matplotlib` - a flexible plotting library for creating interactive 2D and 3D plots that can also be saved as manuscript-quality figures.
- `Pandas` - a data manipulation library based on `NumPy` which provides many useful functions for accessing, indexing, merging and grouping data easily.
- `Rpy2` - a `Python` binding for the `R` statistical package allowing the execution of `R` functions from `Python`.
- `PsychoPy` - a library for cognitive scientists allowing the creation of cognitive psychology and neuroscience experiments.
- `TensorFlow` - an open source software library for numerical computation using data flow graphs.

The main thing relating these applications to ours (in addition to the fact that we make use of many of them) is the fact that each one of them provides a set of tools to deal with common scientific programming needs.

Prolog

There are several uses of `Prolog` and `Aleph` in the setting of data mining. For concrete examples, one can see the scientific publications of Rui Camacho, Inês Dutra and Vítor Santos Costa, among others. These where the works that inspired the use of `Aleph` in this setting, so the similarity is obvious, although the results are quite less attractive.

2.2 Implementation

2.2.1 Implementation Details

The `Python` modules are essentially a collection of tools to perform several tasks. The original input is an `XML` file with a fixed structure. As for the output, it can be a redacted version of the `XML` file, a `NumPy` array or a `Pickle` file. The `Pickle` files and `NumPy` arrays are used as input from one `Python` module to another. The redacted `XML` is used by the `Prolog` modules. In this sense, there is no specially structured communication method between the `Python` and the `Prolog` parts. Since they can both process `XML` and the original data is in that format, that is the format used to pass data from one to the other.

2.2.2 Development Environment

We used `PyCharm` as the main development environment for `Python`. `Jupyter Notebook` was also wildly used, not just for its final task of extracting the clusters and showing the hierarchical clustering diagram, but also as a way to quickly test some `Python` code.

As for `Prolog`, we decided to use the text editor `Sublime Text` with a specific `Prolog` plugin.

The terminal application chosen was `iTerm 2`.

A portable computer running `macOS Sierra` is where the code was written and some faster parts where tested and run. A virtual machine, provided by INESC TEC, with 240GB of memory, 16 cores and running `CentOS` was used to run `Aleph` and some `Python` modules that where slower, such as `word2vec.py`, `linkage.py` and `cophenet.py`b, when there was a lot of data to process.

3. Results

Our first test was done using the first 100 drugs from `DrugBank` and the interactions they have with each other. In this test, `Word2Vec` usage improved the recall in about 5%, so we where encouraged. It was also clear from this test that 100 was about the number of drugs we could use in each test in order for `Aleph` to finish its task in a timely manner. In fact, using `Word2Vec` data, `Aleph` can take up to a day to finish, depending on the amount of information of the drugs.

In order to test our learning method, we used a set containing the first 500 drugs on the database. For that set, we performed 10 tests. In each test, we randomly selected 100 drugs for training and used the rest for testing. The drug to drug interactions considered are only the ones among each group. And for the negative examples we took about 10 times as much as the positives.

The results are detailed in the file `accuracy_precision_recall.xlsx` provided with this report. In Figure 3.1 we can observe a the precision recall curves for our training and test sets. For each set, we used two methods, one with only `Aleph` and another with `Aleph` and `Word2Vec`.

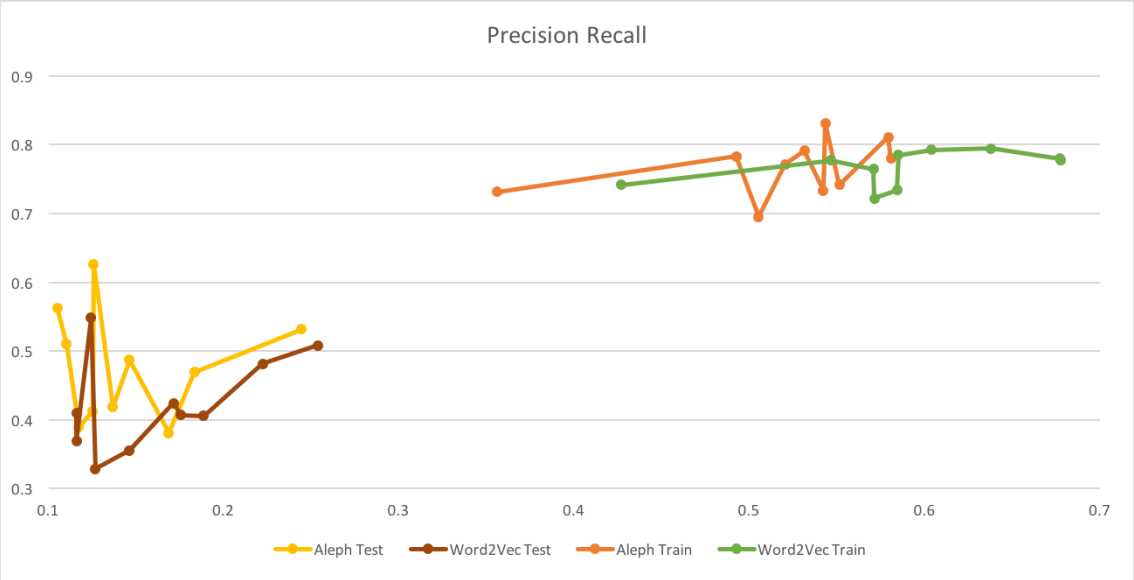

Figure 3.1: Precision recall curves

One can observe that although the training results are good, the models fail on all tests. Also, there is no significant difference in quality of the results between using `Word2Vec` data or not. However, when provided with `Word2Vec` data, `Aleph` does choose to use it in

its theories quite often. This can be checked in the files `drugbank_all_500_N.pl`, where `N` is the test number, between 1 and 10.

4. Conclusion

The results from our preliminary tests are very discouraging. In fact, our method fails to predict drug interactions in a satisfactory way. However, this was an important learning experience and it is our hope to be able to improve this method and its results.

It is quite clear that there is much to be gained from using different languages to do different parts of an application. Also, depending on the application different paradigms may be very useful and are one of the main reasons to use more than one language.

5. Improvements

This is very much a work in progress so there are plenty of improvements to be made. The code itself can be better organized, although it is quite better now than when the first experiments began. The main improvements though should be made in the method. For this we have several ideas, such as:

- Use a clustering method to obtain groups of drugs where `Aleph` can be used with more success, since `Aleph` is proposing a very large theory with each rule covering only some examples.
- Use other data such as molecular describers for the drugs or patient drug usage and effects reports.
- Change the way we are using `Word2Vec` in order to obtain better information from it, for example, use a different context size, a different clustering method or no clustering at all using only a maximum distance between word vectors.
- Change the settings in `Aleph` or the modes and determinations used.
- Try other parts of the `XML` in `Aleph`.

It is our hope that these and other ideas will lead to a good method for predicting drug interactions.

6. Resources

6.1 Bibliography

6.1.1 Publications

Distributed Representations of Words and Phrases and their Compositionality; Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado and Jeffrey Dean; http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.

Demand-Driven Clustering in Relational Domains for Predicting Adverse Drug Events; Jesse Davis, Vítor Santos Costa, Peggy Peissig, Michael Caldwell, Elizabeth Berg and David Page; http://icml.cc/2012/papers/644.pdf.

6.1.2 URLs

The Hitchhiker's Guide to Python - Scientific Applications, http://docs.python-guide.org/en/latest/scenarios/scientific/.

TensorFlow - Vector Representations of Words, https://www.tensorflow.org/tutorials/word2vec.

The Aleph Manual, http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph.html.

SciPy Hierarchical Clustering and Dendrogram Tutorial, https://joernhees.de/blog/2015/08/26/scipy-hierarchical-clustering-and-dendrogram-tutorial/.

Word2Vec Tutorial - The Skip-Gram Model, http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

Zichen Wang - TensorFlow Playground, https://github.com/wangz10/tensorflow-playground/blob/master/word2vec.py#L105.

6.2 Software

PyCharm, JetBrains, https://www.jetbrains.com/pycharm/specials/pycharm/pycharm.html.

iTerm 2, George Nachman, https://iterm2.com.

Sublime Text, Sublime, `http://www.sublimetext.com`.

TensorFlow, `https://www.tensorflow.org`.

Aleph, `http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph`.

Appendices

A. User Manual

1. Before using this software it is necessary to install some python packages. This can be done using pip. We recommend the usage of Python 3, since that was the tested version. Depending on your system, some packages may already be installed, so we will not list all of them. In any case, when calling any of the python modules, if a Python library is missing, it will say so and you can install it at that time.

2. Install TensorFlow, following the instructions in `https://www.tensorflow.org/install/`.

3. Execute `python3 drugbank.py word2vec N MODE [TYPES] &`, where `N` is the number of drugs (0 if all); `MODE` is either 'all' (for all types of drugs), 'only' for drugs with all types in the list TYPES or some for drugs with at least one type in the list `TYPES`; and `TYPES` is a list of drug types, a subset of 'approved', 'illicit', 'experimental', 'withdrawn', 'nutraceutical', 'investigational' and 'vet_approved'.

4. Execute `python3 drugbank.py names drugbank_MODE[_TYPES_N].xml &`, where `MODE`, `TYPES` and `N` are as above.

5. Execute `python3 word2vec.py NAME DIMENSION DIAMETER RAY METHOD M &`, where `NAME` is the word file name after `drugbank_`, that is, `MODE[_TYPES_N]`; `DIMENSION` is the dimension of the embedding vector space, `DIAMETER` is the full size of the context of each word (in both directions); `RAY` is usually half of the `DIAMETER`; `METHOD` is either `skip-gram` or `cbow` and `M` is the number of steps to take when training the neural network.

6. Execute `python3 filter.py MODE final_embeddings_NAME_METHOD_M_DIAMETER_RAY concatenated_drug_names_NAME true`, with the parameters as above.

7. Execute `python3 linkage.py final_embeddings_NAME_METHOD_M_DIAMETER_RAY_filter_concatenated_drug_names_NAME MODEL`, with parameters as above and `MODEL` the linkage model, for example 'ward'.

8. Execute `python3 cophenet.py final_embeddings_NAME_METHOD_M_DIAMETER_RAY_filter_concatenated_drug_names_NAME MODEL`, with parameters as above.

9. Observe the files `word2vec.log` and `cophenet.log` to determine which embedding size is the best, it should be the one that converges in `Word2Vec` and as the highest value of `cophenet`.

10. Adapt the `Jupyter Notebook` `word2vec-filter-500.ipynb` to your file names, in order to extract the actual clusters of words to be used in `Aleph`.

11. Copy the cluster list obtained in the previous step to the file `word2vec.pl` to generate the `Word2Vec Aleph` background; in `SWI-Prolog` import the module `word2vec.pl` and run `word2vec_2_aleph.`; a file named `word2vec_temp.pl` is generated; place it in the folder where you will run your experiments and rename it `word2vec.pl`.

12. Execute `python3 drugbank.py prolog N MODE TYPES`, with the parameters as above.

13. Execute `python3 drugbank.py hyphens drugbank_NAME.xml`.

14. In `SWI-Prolog` import the module `drugbank_test.pl` and run `xml_2_aleph('drugbank_NAME.xml', TESTS, SIZE, TESTNAME).`, where `NAME` is as above, `TESTS` is the number of tests to generate, `SIZE` is the number of drugs to use in the training of each test set and `TESTNAME` is the base name for the test `Aleph` files that will be generated.

15. Copy the test files generated in the last step into two folders; change the background files (the ones with extension `.b`) in one of those folders to include the `word2vec.pl` file previously generated, by adding the line '`:- [word2vec].`'.

16. Place the `aleph.pl` and `settings.yap` files supplied with this report in the test folders.

17. In `YAP` run
    ```
    [aleph].
    read_all(TESTNAME_I).
    induce.
    ```
    where `TESTNAME` is as above and `I` is the number of the text (between 1 and `TESTS`, inclusively).