Pathogenicity prediction with Neural Networks

Project documentation by Paul Eisenhuth

This documentation aims to provide help and an easy start to whoever continues this project. It is written with the greatest care but by no means can guarantee to be completely correct and covering each and every bit. Also, I am writing this being heads deep into this project, so there is high chance that I am not covering things, which are really important, because I forget, that everyone else has to start from 0 again.

But first things first:

Help

If you find yourself stuck don't hesitate to contact me and ask for help. All the scripts (or at least most of them) are written by me and made for me and I will happily explain what my thoughts and goals were. Also I decided on most of the steps during the project, so go ahead and don't fear to ask "stupid" questions. Only expect some delay before I answer because first, I live in Germany, so I am always 7 hours later than you and second, I might also need to have a look at the script or step again, but help will come! Just write an email to **eisenhuth451@gmail.com**

Further I highly recommend asking all the friendly people around you. Greg was always really great helping me with the *pdbmap* and general programming issues, whilst Jonathan from the Meiler Lab is awesome at explaining the biology behind everything, as well as an introduction to *ddG.* If you have trouble with *pathprox*, ask Chris from the Meiler Lab. He is developing this tool further and probably the only person that fully understands what is happening there. Also, he has great knowledge in *Python* and is super helpful at bug fixing.

As a last resort you can also read my lab notes, but these are mainly written for myself to remember stuff I did, but then again, there might be something very useful for you hidden in it.

Project description

The main goal is to develop a new score for pathogenicity prediction of variants of unknown significance. There is currently a lot work done just incorporating DNA and RNA sequence data, but we are expecting an increase in performance through additional incorporation of structure related information. In my twelve weeks working at the project I tried to build up a proof a concept, that using structural features alone already provides some predictiveness to pathogenicity, which was successful.

What I did was collecting experimentally determined structures of proteins of high resolution and low size from the *pdb database* to have data of high accuracy which don't need super much computational time to calculate scores. Then, I collected missense variants which are lying within these structures and which are known to benign or pathogenic, using *gnomad* and *clinvar*. After this step I filtered the structures further, so that only structures with a minimum amount of benign and pathogenic variants remain and further I made sure, that for each structure only one chain is used and that only one structure belongs to one *uniport id* to guarantee a wide variety of different data.

The next step was filling *ACCRE*, the computational cluster at Vanderbilt, with jobs to calculate *ddG* and *pathprox* scores. Afterwards I loaded *uniprot annotations* and grouped them together. This resulted in my final dataset to train a machine learning classifier.

As a preparation for training I had to scale all my features to [0,1], because of machine learning reasons. This is super easy for everything except the *ddG* score. With the help of a *roc curve* I decided to treat every absolute score below 1.306 as not destabilizing and therefore set it to 0 and every score above 7.312 as highly destabilizing and therefore set it to 1. Every score in between is scaled accordingly to the interval.

I then trained eight *multilayer perceptrons (MLP)*, one half with *identity* as *activation function* and the other half with the *rectified identity*. In each group I had a *MLP* with one layer, two layers, three layers and one with four layers of 100 nodes per layer.

Results

- Using *identity* as *activation function* does not work at all, but the other one does well.
- 100 nodes are way too much and cause overfitting
- With 4 layers around 10000 training iterations are absolutely enough
- Using structure resolution and size as features, as I did, is a stupid idea and should not be done
- All *uniprot annotations* did not perform well for me except "Helix" and maybe "disulfide bond"

Note: I produced these results within my last 7 days and had to deal with a lot of technical difficulties, therefore my analysis here has high chance of being wrong. Also, my evaluation methods were made up by me, so… These are more ideas on things to change.

Folder structure…

…or where to find what. I saved all my files at */dors/capra_lab/users/eisenhp/*. In there you will find the following things:

- **classifiers/** - a directory I created for training *MLP classifiers* with *ACCRE*, but never used it, because of software trouble
  - **annotatedScores**.csv – csv file containing all my data
  - **classificationTrainer**.py – python script for reading training data, training the classifier and saving and evaluating the classifier
  - **train_classifier**.slurm – slurm script for ACCRE, almost ready to run if *scikit-learn* is finally installed on ACCRE. You just need to load the right modules with *lmod* before running
  - **logfile_*.**out – logfiles of failed ACCRE runs. Not important.
- **classifiersLOCAL/** - locally run and trained *MLPs*
  - **annotatedScores.**csv – see above
  - **classificationTrainer.**py – see above
  - **logfile_*.**log – logfiles of successful training runs for each MLP
  - **clf_*/ -** the numbers describe the amount of nodes and the lines divide each layer. So 100_100_100 means three layers with 100 nodes each.
    - ***.**save – text representation and performance information of a MLP after a number of training iterations
    - ***.**ann – a *python2 joblib dump* of the *scikit-learn classifier*. This can be loaded an reused. Visit the scikit-learn page for more information.
  - **actfunc_ident_failed/ -** contains the same structured files and directories as classifierLOCAL but only for the failed MLPs with identity as activation function
- **uniprotXML/ -** simply holds .xml files for uniprot annotations collected by *unpantGrabber.py*

- **pathproxPrep/ -** contains the prepared files for pathprox slurm runs as well as all the results
    - **<pdbid>/ -** directories holding the information for each used structure need for the run as well as all the results. Chris is probably better in explaining all this.
    - ***.**out – logfiles from slurm runs
    - **pathproxPrepper.**log – logfile of the python script preparing all this data
    - **pathprox.**slurm – slurm file submitted to ACCRE to calculate all the results
- **newPathprox/ -** the same as above, but redone after Chris discovered some mistakes
- **newFastas/ -** directory holding only fasta files for all my structures, they are the solution to the mistake that Chris discovered
- **toChris/ -** simply a directory holding things that I wanted Chris to have to solve my mistake
- **pathprox/ -** some random files which I used for tests of my scripts
- **ddGprep_out0/ -** directory holding all the files and data for ddG slurm runs and the results
    - **<pdbid>/ -** containing flag files, pdb files and everything else needed to calculate the ddG scores for this structure. Results are saved in ddg_predictions.out, also all files created during calculation are in here. All these files are better explained in my lab notes, also you can ask Jonathan.
    - **ddGcollector.**log – logfile for collecting ddG scores out of this file system
    - **ddGprepper.**log – logfile of the slurm run preparation
    - **ddg_monomer.**slurm – slurm file submitted to ACCRE for the runs
    - **rerun_ddg_monomer.**slurm – slurm file to continue with calculations not done within the first run
    - ***.**out – logfiles of slurm runs
- **dCC_out2/ -** contains logfiles and csv files for data collected from pdbmap with the final criteria. Only important is resultUnique.csv, it holds the finally used data, unique by chain and uniprot id.
- **All other directories –** are for testpurpose only and don't hold any useful information

- **unpantGrabber –** python script and logfiles, this is used to grab uniprot annotations
- **annotatedScores.**csv – Finally used data, copied into classifier directories
- **pathproxFailureReportNEW.**txt – holds information which pathprox runs failed even so they finished and reported a success. Created by pathproxCollector.py
- **scoredVariantsNEW.**csv – result of pathproxCollector.py, holds all the variants which were scored by ddG and pathprox as well as their scores
- **pathproxCollector.**py – goes through the result directories and files of pathprox slurm runs and collects all the scores as well as provides information on failed runs
- **pathproxPrepper.**py – creates slurm file and directories for pathprox runs
- **annogroups –** needed for unpantGrabber.py, specifies how to group uniprot annotations
- **ddGscores.**csv – holds all variants which got a ddG score calculated as well as their score. Result of a ddGcollector.py run
- **ddGcollector.**py – goes through the directories of a ddG run, collects all scores and prepares a new slurm file as well as new flagfiles and mutation files to rerun ddG_monomer on missing variants
- **ddGprepper.**py – prepares directories and slurm file for a first ddG_monomer run
- **convert_to_cst_file.**sh – a script needed during ddG_monomer preperations which just lives here because I couldn't run it in its original place… See ddG_monomer documentation for more information
- **chainPicker.**py – ridiculously complicated script to pick the best chain for each structure to maximize the number of variants globally available.

- **dataCollecterComplete.**py – create the dCC directories and is used for initial data collection from pdbmap

How to use the scripts

You must use the scripts in a specific order because I tailored all of them to process input created by a script run before. If you wish to change parameters of the scripts it might be necessary to directly change code in the script, as some are unfinished and not using all the command line arguments, because I never needed them. Also if you change things in one script, it might happen, that you must change more in following scripts. Sorry for that!

Data collection – dataCollectorComplete.py

Creates your first list of variants. All the command line arguments are working. I used the following command:

```
python3 dataCollectorComplete.py --max_residues 450 --benign_maf 0.05
```

This leads to the creation of a directory named dCC_outX/ with X representing the number of times you've run this program before. Within this directory is the file result.csv, containg all collected variants.

Chain decision – chainPicker.py

Takes result.csv containing directory created before as positional input and creates resultUnique.csv. To be honest, I am not quite sure what the other commandline arguments do and if they are implemented, but you should be able to use this perfectly fine without worrying about them. Important is, that you specify the directory as input, not the csv file. So I probably used:

```
python3 chainPicker.py dCC_out2/
```

Since I had my final data collected into dCC_out2/, because it was my in total third run.

After chainPicker.py finishes you will find a logfile and the new resultUnique.csv in the dCC_out directory. This new csv file only contains one chain per structure and one structure per uniprot id.

Prepare ddG_monomer – ddGprepper.py

This script loads a lot of pdb files, preprocesses them and creates a directory structure ready to be used for slurm runs on ACCRE as well as the slurm file to submit. It only needs the csv file holding all the variants that should be used, so I ran it as:

```
python3 ddGprepper.py dCC_out2/resultUnique.csv
```

This resulted in the creation of the directory ddGprep_out0/ because I never ran it more than one time. Running it takes a long time for the first run, because it needs to download a lot, but reuses this data if you rerun it and set it to overwrite.

Important! Change the mail address in the slurm file. I don't want to get mailed.

After this you should submit the slurm file to ACCRE and wait, these calculations can take very long. The slurm file created will run for 8 days before aborting, but this wasn't even enough to calculate all variants for all structures, even so it is happening highly parallel.

If you use your own directories you should go through the code and make sure, that the slurm file is created with paths to you place. I advise you to even change the python script, so that it produces the correct paths for you.

Collect scores and prepare for rerun – ddGcollector.py

After you ACCRE jobs are done, you should run this script. It collect all the scores and searches for scores that should have been calculated but are not present. For these scores it also creates a new slurm file, rerun_ddg_monomer.slurm, that can directly be submitted to calculate missing values.

Important! Change the mail address in the slurm file. I don't want to get mailed.

ddGcollector.py can be rerun as often as you wish, it will always collect all scores present and only overwrite files it has created on its own. So after you started the rerun of ddg, you can use this script again to collect scores and create a new rerun slurm file to calculate any scores that are still missing. You only shouldn't do it while the slurm jobs are still in progress.

To run it I used:

```
python3 ddGcollector.py dCC_out2/resultUnique.csv ddGprep_out0/
```

So it needs the total list of variants used to search for missing ones and the directory structure holding all the run information and results. This script will produce the files ddGscores.csv containing all variant information and ddG scores and it will also produce the files needed for the reruns.

Prepare for pathprox – pathproxPrepper.py

This script does essentially the same as ddGprepper.py, but for pathprox. It reuses loaded pdb files from ddG runs, but there could still potentially be some problems within this fact. Pathprox is a WIP program and a little bit weird. I did my best here, but maybe it is wrong, or it is wrong when you are using it. Definitely you should use the –fast_dir flag, because it fixes a problem discovered by Chris. What I ran:

```
python3 pathproxPrepper.py ddGprep_out0/ dCC_out2/resultUnique.csv
newPathprox/ --fast_dir newFastas/
```

This, as I said, reuses pdb files from ddGprepper.py, takes again the complete variant list, creates a new directory structure and uses fasta files provided by Chris. In the specified new directory structure, newPathprox/ in my case, you will find again a slurm file which is ready to be submitted to ACCRE.

Important! Change the mail address in the slurm file. I don't want to get mailed.

Finally collect the last last scores – pathproxCollector.py

Not very surprising, but this script navigates through all the files and folders created during the pathprox runs to collect the scores and provide some information, which and how many runs failed. It does not create a new slurm file, since pathprox is quite fast, so I reran all of it after I discovered some errors. I used:

```
python3 pathproxCollector.py -o scoredVariantsNEW.csv -r
pathproxFailureReportNEW.txt scoredVariants.csv newPathprox/
```

After the run I had the failure report and all my scores with all scored variants in one csv file.

Collecting uniprot annotation – unpantGrabber.py

Browses the uniprot server for annotation data and further enhances the score file from the previous run. I used:

```
python3 unpantGrabber.py scoredVariants.csv annogroups
```

This created annotatedVariants.csv containing all the final information. Annogroups specifies how to group annotations together. This follows a syntax <groupname>|<annotation1>,<annotation2>,… The

groupnames specify how the flags are going to be called that will be set for the variants while the annotations specify which annotations set the flag. So if at least one of the annotations is present, the flag for the whole group will be set. You can call your groups however you want except “None”, this is a keyword to ignore certain annotations. During the run you will get warnings in the log for annotations which are not covered by you group file. These will be ignored, but you might want to adapt the grouping file to also include them. I made some hardcoded exceptions within the code to deal with annotations that are written in multiple different ways. This might lead to problems for you!

All the other options and flags here shouldn’t be used. This script was written very fast and with less care, so it might behave very odd, but I am happy to help with that.

Training a classifier – classificationTrainer.py

This program caused a lot of problems with python versions and modules, so it is not well tested, but what it does is basically training a MLP classifier. You should try to understand this program and maybe rewrite it. This is so much not done, that I will not tell you how to use it right of the shelf, since this is way to specific, however I am again happy to answer your questions whilst you are digging through it.

Conclusion

The most important thing, don’t get frustrated, but ask for help. As is said multiple times you can always mail me. This documentation is probably not enough, but it should help getting into everything. All python scripts also contain comments which might help further understanding them. The lab notes further elaborate my thoughts and scikit-learn provides good sources to get into classifiers.

To continue this project there a many many things you could do. Test different classifiers, change the settings of MLP classifiers, enlarge the dataset, use other annotations, incorporate more scores and features, use more elaborate analysis of my classifier, change pathprox scores, change interpretation of ddG scores and so on. There is a lot of fun and science ahead!

I wish you the very best for continuing this project.

Paul Eisenhuth, 12/14/2018