POSE 0.75 user manual

Please contact David Masica (david.masica@gmail.com) with questions, comments, or bugs. Please read the manual in its entirety and try one or two of the Examples before contacting the "help desk" ☺. If you find potential bugs or runtime errors, please cut and paste the python error into an email, and attach your Arguments and Paths files.

`Courier` font denotes a POSE command, and is defined in the Commands section.

What is the POSE algorithm?

The POSE algorithm is a suite of computational modules primarily developed for the purpose of predicting the impact of protein amino acid substitution on phenotype. Here, phenotype is used in a broad sense and describes diverse phenomena such protein function, organism-level traits, specific diseases, or even specific disease symptoms or causes (e.g., cholesterol level in the context of heart disease). The POSE algorithm can be trained to derive a classifier that is tailored to your specific phenotype, which requires that you provide mutations whose phenotypic impact is known. The POSE algorithm facilitates the prediction of dichotomous (binary) phenotypes, such as positive or negative for a particular disease, or continuous-valued traits (so-called endophenotypes) such as blood pressure or enzyme catalytic rates. In addition to training mutations, the POSE algorithm requires a multiple sequence alignment. And, you can optionally supply a 3D protein structure in PDB format. To clear up a potentially confusing naming convention at the start, the classifier produced by the POSE algorithm is call a phenotype-optimized sequence ensemble (POSE); therefore, the POSE algorithm is used to create a POSE. I encourage you to read the POSE methods papers referenced at the end of this tutorial.

Installation

The recommended installation follows. This is a simple installation with no automated installer; therefore, advanced users can tailor most steps to their own liking. This assumes a Linux environment, so if you're using a Mac or any flavor of Linux you are ready to proceed. If you are using windows, install a VirtualBox running Linux (I recommend Ubuntu). There are many great tutorials for accomplishing this, simply google "virtualbox ubuntu windows".

Download POSE (*https://github.com/KarchinLab/pose*) to your home directory and unzip it. To unzip do: unzip POSE-master.zip -d POSE.

Add the POSE directory to your python path. On Linux this will likely be adding *export PYTHONPATH=/home/user/POSE/:* to your *.bashrc* file. On a Mac you'll probably be

adding *export PYTHONPATH=/Users/user/POSE/:* to your *.profile* file. Of course, "user" would be your username on your machine. Then add the line alias *pose='python ~/POSE/pose.py'* to either your *.bashrc* or *.profile*.

Install pip (`sudo easy_install pip`), NumPy (`pip install numpy`), and RPy2 (`pip install rpy2`). RPy2 requires that you have R installed. Your R installation will need access to certain non-default R libraries; if you are missing any required libraries for your desired POSE implementation the runtime error will tell which library to install.

POSE is written for Python2.7. If your machine's default python is not Python2.7, you need to add the correct interpreter to the pose.py file in the POSE directory. As an example, the first line of your copy of pose.py could be *#!/home/user/Python-2.7/python*, if you wanted to point POSE to a local copy of Python2.7. Once you've done this, all POSE code in the *POSE/POSE/* directory will know where to find the correct python installation.

Input Data

The two types of data provided to POSE are lists of mutations (of known impact for training, or those for prediction) and the optional protein structure file in PDB format.

Mutations

Input mutation files for the purposes of deriving (e)POSEs (i.e., training) should consist of a column of amino acid substitutions and a column for the known phenotype if making POSEs or endophenotypes if making ePOSEs. Here is an example of a mutation file for deriving a POSE:

```
A243P 1
T315Y 1
I373P 0
G539H 0
```

In the above example, the first two mutations cause the phenotype under consideration, so they have a "1" (i.e., `True`). The last two mutations do not cause the phenotype and therefore have a "0" (i.e. `False`). Keep in mind this is just an example, in practice you'd want many more than four mutations. Next is an example input mutation file for deriving ePOSEs:

```
R133H   109
G134R   181
G199K   107
K209D   117
N222G   179
```

In that example you simply have mutations and continuous-valued numbers. Examples of endophenotypes include blood pressure, cholesterol level, enzyme catalytic rate, tumor growth rate, gene expression, etc. These can range from patient- or tissue-level phenomena down to cell- or gene-level phenomena; essentially any continuous-valued parameter.

The two above examples of mutation files are formatted for deriving POSEs or ePOSEs; that is to say, these are the mutations for which you know the associated (endo)phenotype and can therefore be used for training. When you make subsequent predictions on new mutations, the files will just be lists of amino acid substitutions (i.e., no second column).

Running POSE

Once your data is formatted correctly, you just need to tell POSE you want to use it, where it is, select your POSE mode (what type of POSE calculation you want to do) and hit "go"! Below I outline the recommended method for running a POSE calculation, but those familiar with Linux will quickly recognize that this customizable. If python knows where the POSE directory is, you have a copy of the POSE executable (pose.py), and POSE knows where the data is via the Paths file, then you are ready to run POSE.

Recommended method: All examples in the Examples section assume you are using this approach.

1. The POSE directory should be in your home directory, and python needs to know where it is (see Installation). This step is quite arbitrary and can be tailored to your liking with no impact on POSE's ease of use; however, the Examples will assume this is where POSE is.
2. Keep your POSE input data in the *POSE/Databases* folder. This way you only need one copy of any data file, and it can remain stationary in the Databases folder (the Paths file will tell POSE where it is).
3. Run your calculations from the *POSE/Projects* folder, making a new folder in *POSE/Projects* for each project.

Running on multiple processors (`MultiProcessMode`):

For large calculations, the power user can run POSE in parallel across multiple processors (cores, nodes, cpu's, etc.). This is achieved using the `MultiProcessMode` command and giving each processor its own Arguments file. The `MultiProcessMode` command takes two arguments: an integer telling POSE which processor it is being executed on and an integer telling POSE the total

number of processors used for the calculation. The default is `MultiProcessMode = 0 1`, meaning POSE is running on the zeroth processor and that is the only processor in use. Multi-process mode works by taking a list of items to compute and uses python's built-in extended-slicing functionality. Here is a toy example, from the python terminal, illustrating how POSE would farm out a computation based on ten mutations:

```
>>>                          Mutations                          =
["M0","M1","M2","M3","M4","M5","M6","M7","M8", "M9"]
>>> TotalNodes = 3
>>> for Node in range(TotalNodes):
...     print Node, Mutations[Node::TotalNodes]
...
0 ['M0', 'M3', 'M6', 'M9']
1 ['M1', 'M4', 'M7']
2 ['M2', 'M5', 'M8']
```

In this example, Node 0 is taking care of Mutations M0, M3, M6, and M9. Node 1 is taking care of Mutations M1, M4, and M7…etc. Ideally, the user will write a bash script (or something similar) to utilize a cluster resource manager. In the POSE folder, there are examples of such a script, written to utilize both the Portable Batch System (*MultiProcessMode.PBS*) and the Sun Grid Engine (*MultiProcessMode.SGE*). To use either of these submission scripts, do: ./*MultiProcessMode.XXX* `integer string`, where the integer is the number of processors you want to use and string is the name extension of the directories you want the results from each node to be saved.

Customizing POSE (`MyPOSE`):

Advanced users and python coders can easily create and extend POSE functionality. This will typically amount to writing one or more functions in the *MyPOSE.py* script, and these functions will call existing POSE functions that you will be able to tweak and combine in you own way. See the directions in *MyPOSE.py* to get started.

Examples

The following examples all assume you are running POSE using the recommended method, as described in Running POSE. Each example has its own directory, which can be found in the *POSE/Projects/Examples* directory. These are simple, trivial examples that are meant to familiarize you with POSE and only take a few minutes each. More info on the mutations included in this tutorial can be found in the two references at the end of the document.

In each example, open the provided Arguments and Paths file, and familiarize yourself with some commands. Probably the most important and common similarity among all Examples (and every calculation you can do with POSE) is `Mode = $mode` and `Mutations = $mutation_tag` in the Arguments file, and the corresponding tags in the Paths file (e.g., `$mutation_tag = PathToMutationFile`). As an example, your Arguments file could have the command:

```
Mutations = MutationTag
```

Then your Paths needs to have:

```
MutationTag = SomePath/SomeMutationFile
Fasta = SomePath/FastaFile
```

Where *SomeMutationFile* is your input list of mutations and *FastaFile* is the required multiple sequence alignment (MSA); the `Fasta` arguments is not referenced in the *Arguments* file, and is always instantiated in the *Paths* file.

Both of the following examples consider amino acid substitutions in the cystic fibrosis transmembrane conductance regulator (CFTR). CFTR mutation can cause the disease cystic fibrosis, while many CFTR mutations appear to have no impact on disease. The purpose of these examples will be to derive POSEs and ePOSEs using variants of known impact, followed by the assessment of some new mutations.

The first step in a new POSE calculation is to create the POSE-formatted *.fasta* file, which is a sequence alignment to the target gene that has been reduced in sequence to that of the target gene (i.e., no gaps in the target), and the pickled as a python dictionary of gene-name keys and sequence values. For the examples in this tutorial, I made the *.fasta* file using the following `Arguments` file (the resulting *.fasta* file is in *Database/Examples/Sequences/*):

```
Mode = POSE
MakePOSE = False
PreProcess = MakeAlignment
GeneIdentifier = CFTR
Domain = 376 628
Email = your_email@email.com
Filename = CFTR_NBD1
```

In the following examples, we will be working with variants in the CFTR gene, some of which are known to be associated with cystic-fibrosis disease. All variants are located in distinct evolutionarily conserved domains called nucleotide-binding domains, which is why I used the domain flag in the above arguments file, for creating the *.fasta* file, which I called *CFTR_NBD1.fasta*. It is never necessary to isolate specific domains of your target protein, but if you believe the protein is made of distinct evolutionary domains, this can

be helpful. If you want to try and create your own *.fasta* file, simply type "pose" on the command line in the Projects directory, and hit "enter"; the Arguments file in that directory has all the above arguments, necessary to make the *.fasta* file. Note: this involves doing blast searches and a uniprot alignment, which can sometimes take a while. Also, please change the dummy email address to your own.

Example 1: Derive a POSE using 11 CFTR mutations causally implicated in cystic fibrosis (phenotype-positive class, "1") and 9 CFTR mutations found in patients without cystic fibrosis (phenotype-negative class, "0").

In a command-line terminal, go into *POSE/Projects/Examples/POSE/*, type "pose" and hit "enter", with the *Arguments* and *Paths* files as is. See the index at the end of this document for a description of all parameters in the *Arguments* file. You'll notice `Mode = POSE` (rather than `ePOSE`). We are calling our mutation list *CysticFibrosis*, which in-turn tells the *Paths* file how to find the mutations. We are also going to use the structure of full-length CFTR in the inward-facing conformation; this structure was predicted in *Mornon et al.* Cellular and Molecular Life Sciences (**2009**) and is found in POSE/Database/Examples/Structure. To tell POSE to use the structure we set `Structure = True`. It's also good practice to tell POSE the chain ID; in this case, there is only one chain (*A*). I have also provided `Seed` with an integer, which will help control stochastic elements of computation and hopefully you'll get the same results that I present in this tutorial (Note: different operating systems might respond to the seed differently, but our results should be very similar, regardless). Finally, notice that the *Path* file points POSE to a *.fasta* file. About 4-5 minutes after execution you should have three POSEs: *CysticFibrosis.POSE.0-2*.

Now you have some POSEs trained to predict cystic fibrosis from CFTR mutation, so let's make some predictions on other mutations. First change `MakePOSE` to `False`, and set `PostProcess = Predict`. Next, change your *Paths* file so that `Mutations` points to the *Test* set rather than the *Train* set. Now run pose. The first data printed to the screen is a list of the new mutations, each with their mean and standard deviation POSE score, and the predicted phenotypic (CFTR positive or negative, in this case). Next is the cutoff score used to make the phenotypic predictions. Finally, you'll see some performance statistics about how well POSE did when it was training on the initial mutations. The only point in these performance metrics is to help you decide whether or not you trust the POSEs ability to make predictions for your phenotype given the mutations you used to train the POSE.

Example 2: Do leave-one-out cross validation and see how well POSE can predict the impact of CFTR mutation on *in vivo* chloride conductance. The relative chloride conductance of cells expressing mutant CFTR has been shown to be a good predictor of

how the mutants will impact cystic fibrosis disease, where decreasing chloride conductance is correlated with increased disease severity.

Go to POSE/Projects/Examples/ePOSE and look at the *Arguments* file. This time we have `Mode = ePOSE` because chloride conductance is a continuous-valued trait, or endophenotype (i.e., we will be making endophenotype-optimized sequence ensembles). We now have `Mutations` pointing to the *Conductance* file, and `Correlated = False` because chloride conductance is inversely proportional to disease severity (and POSE score). I have `LeaveSomeOut = 1`, which will result in a leave-one-out cross-validation calculation. If you wanted to do 10-fold cross validation, you would set `LeaveSomeOut = 2` because there are 20 mutations.

This calculation could take around an hour. You could speed it using multiprocessor mode (see above and index), or you could increase the number of mutations left out in each data split. Once the calculation is complete, set `LeaveSomeOut = False` (or delete it) and set `PostProcess = Validate`. Now run POSE and see how well the ePOSE performed during cross validation.

The output will be each of the mutations used for cross validation, the corresponding experimental measurement (chloride conductance), mean and standard deviation score, and predicted mean chloride conductance and standard deviation. Finally, you'll see the R-squared, Pearson correlation coefficient, and P-value. This type of cross validation is a more rigorous way of determining whether or not you find the ePOSE classifier suitable for your endophenotype, given the mutations you provided for training. If indeed the classifier performed well, you could make a POSE and score new mutations as in Example 1.

Commands

Use any of the following commands to override the MOCA defaults. In the Arguments file use the `Command = Option(s)`.

`Mutations`: Just provide a name, put that name in your paths file and point it to a file that has your training mutations or the mutations you want to score.

- Options: Any string
- Default: `None`

`Mode`: Phenotype- or endophenotype-optimized optimized ensembles.

- Options: `POSE/ePOSE`

- Default: `None`

`Correlated`: Describes relationship between POSE score and the (endo)phenotype of interest. POSE scores range from -3.0 to 3.0, where zero is equivalent to wild type and increasing values are increasingly disruptive/deleterious (negative scores indicate a mutation predicted to be advantageous to wild type). If the (endo)phenotype you're trying to predict is anti-correlated with disruption to the protein of interest, then you'd want to set this to True.

- Options: `True/False`
- Default: `True`

`Trials`: How many times to restart (e)POSE optimization over from the beginning. Having multiple starts from the beginning can be important, because the sequences selected in the first few random selections can considerably bias the final sequence composition of a POSE. The number of trials you do is the number of POSEs you will have when the calculation is finished.

- Options: Any integer
- Default: `1`

`Optimizations`: How many sequence selection to make for each `Trial`.

- Options: Any integer
- Default: `1000`

`OptimizationParameters`: How often, in cycles, to repopulate the sequence pool with top-performing sequences (Default = 100) and what percent of top-performing sequences get appended back to sequence pool (default = 1%; 0.01). The third parameter says how many sequence ensembles should be written to each POSE (i.e., for each Trial). The default is 1, meaning that each POSE will only contain one sequence ensemble.

- Options: `Integer, float, and integer.`
- Default: `100 0.01 1`

`LeaveSomeOut`: Performs leave-some-out cross-validation; you specify the integer number to leave out. For the common leave-one-out cross-validation, you'd just specify "1". If you want to do, say, 10-fold cross-validation, divide the total number of mutations by 10. So, if you have 116 mutations, you'd set `LeaveSomeOut = 11.6` to do 10-fold cross validation. Beholden to your class imbalance (not relevant for ePOSEs), POSE will try to keep the same number of phenotype-positive and phenotype-negative mutations in the testing (hold-out) group. And of course, each mutation appears in the testing group only a single time.

- Options: Any integer.
- Default: `False`

`PostProcess`: A few different methods for analyzing POSE output.

- Options: `Score, Validate`
    - `Score`: If you have already trained a POSE classifier using MakePOSE you can score some new mutations.
    - `Validate`: Validate the (e)POSE you derived from leave-some-out cross validation.
- Default: `False`

`Structure`: Supply PDB-formatted 3D structural coordinates if you want the POSE score function to consider protein structure. If True, you need the Structure pointer in the Paths file.

- Options: `True/False`.
- Default: `False`

`Chain`: The normalized residue burial calculation considers the entire input PDB by default. Use this argument if you want to restrict this calculation to a particular PDB chain.

- Options: Any character, same as in your PDB file.
- Default: `False`

`Filename`: Specify the prefix name (i.e., anything before the ".") for POSE input or output.

- Options: Any string indicating the name you want to use for input/output files.
- Default: Joins the arguments supplied to `Mutation` and `Mode`. For instance, the default filename for *Example* 2 in this tutorial would be *ChlorideCondutance.ePOSE*

`Seed`: For any calculation that is too large to be exhaustive, there will be some random sampling of features. Use this command to control these stochastic elements of the calculation and make your calculation reproducible. Note: this might only guarantee reproducibility on a single machine or operating system, but will probably be consistent across operating systems.

- Options: Any integer.
- Default: `False`

`MultiProcessMode`: Use this command to parallelize calculations across multiple processors. This is useful for large calculations.

- Options: Two integers. The first specifies the node that that particular job is running on and the second specifies the total number of nodes to be used. See Running MOCA for more details.
- Default: `False`

`MyPOSE`: For the experienced user and Python programmer that wishes to create and expand upon existing POSE functionality. See Running POSE for more detail.

- Options: Any user-defined function.
- Default: `False`

References

Masica, D. L., Sosnay, P. R., Cutting, G. R., & Karchin, R. (2012). Phenotype-optimized sequence ensembles substantially improve prediction of disease-causing mutation in cystic fibrosis. *Human mutation*, *33*(8), 1267-1274.

Masica, D. L., Sosnay, P. R., Raraigh, K. S., Cutting, G. R., & Karchin, R. (2014). Missense variants in CFTR nucleotide-binding domains predict quantitative phenotypes associated with cystic fibrosis disease severity. *Human molecular genetics*, ddu607.