Detecting Selective Sweeps Using `S/HIC`

This exercise is a somewhat stripped-down pipeline for using the `S/HIC` software to detect selective sweeps. We will both train and test the software on some simulated data before moving on to a reasonably small real data set.

Let's start by creating a working directory for this exercise.

```
mkdir sweepPipeline
cd sweepPipeline
```

Now let's copy over all the scripts we need to complete it.

```
cp /home/schrider/sweepPipeline/*.sh .
cp /home/schrider/sweepPipeline/*.py .
```

Now we're ready to get to work!

Step 0: Simulating training and test data

Unfortunately, machine learning requires training data, and if we don't have at our disposal a large dataset for which the ground truth is known (as is the case when it comes to detecting selective sweeps), then we have to rely on simulation. Luckily, there are a few coalescent simulators that can simulate sweeps relatively quickly. We are going to use one called `discoal`, which is already installed on the server.

Recall that S/HIC seeks to detect both hard and soft sweeps, and do narrow down the selective region by explicitly handling regions that are affected by nearby selective sweeps but not themselves under direct positive selection. To accomplish this, S/HIC tries to discriminate among five classes: 1) Neutrally evolving loci 2) Loci centered around a recent hard selective sweep 3) Loci located near a hard sweep 4) Loci centered around a soft sweep 5) Loci located near a soft sweep. So we have to simulate each of these. For class 1, we simply simulate large regions with no selection. For classes 2 and 4, we simulate large regions with a sweep occurring near the center (hard and soft, respectively). For classes 3 and 5, these sweeps are not in the center of the simulated window, but instead some distance away (which we will vary across training examples).

So that's great, but we have to know how to simulate data with `discoal`. Let's start off with a neutral simulation:

```
discoal 20 100 100 -t 110 -r 110
```

This simulates 100 replicates of a sample of 20 individuals. (A set of 100 replicates is not really enough but never mind that for now). The output will be in the same format as Hudon's `ms` simulator. Our population-scaled mutation and recombination rates 4*Nu* and 4*Nr* are both 110. (These values are probably too small but never mind that for now.) This command line is a bit messy. Let's set some `bash` variables first to make it more understandable.

```
# set some bash variables
sampleSize=20
numReps=100
recSites=100
theta=110
rho=110

# run our neutral command
# note: I know it is weird to put code into a pdf and this
# is one reason why: the command below is all one line and
# it can be hard to tell if it is one or two. I have tried
# to note these cases in the comments to avoid confusion.

discoal ${sampleSize} ${numReps} ${recSites} -t ${theta} -r
${rho} > neut.msOut
```

So now we can see where each of these variables goes on the command line for running discoal. (Note that this is similar to `ms` but not quite the same). You can ignore the `recSites` variable for this exercise (this is necessary for handling the locations of recombination events along the chromosome but the exact value is not that important—as long as it is "large enough" we are fine—ask me about this later if you are really interested in doing lots of coalescent simulations in the future and curious about how this parameter might affect things). Anyway, that is your introduction to `bash` and setting/using `bash` variables. You're welcome!

Now that we can simulate neutrally evolving regions, we need to know how to simulate selective sweeps. We will do this by adding this segment to the end of our neutral command listed above:

```
-ws 0 -a ${alpha} -Pu 0 ${maxSweepAge} -x ${sweepLoc}
```

Here the `-ws` flag tells the simulator that we will have a complete selective sweep (ignore the zero following it), `alpha` species the strength of selection in units of 2*Ns* where s is the selective advantage of the sweeping allele, the `-Pu` flag is used to specify the range of fixation times allowed (i.e. when did the sweep complete) which in this example we are allowing to range uniformly from zero (the sweep finished yesterday) to `maxSweepAge` (the sweep finished `maxSweepAge`*4*N* generations ago). Finally, `-x` specifies the location of our sweeping mutation along the chromosome. `-x` can range between 0 and 1, so a

setting of `-x 0.5` means that the sweep is right at the center of the chromosome.

So, let's simulate a sweep in the center of the chromosome that occurred at most 0.01*4*N* generations ago:

```
# set some variables
maxSweepAge=0.01
sweepLoc=0.5
alpha=500

# build our neutral command (this is all one line)
neutralCmd="discoal ${sampleSize} ${numReps} ${recSites} -t
${theta} -r ${rho}"

# run our command (this is all one line)
$neutralCmd -ws 0 -a ${alpha} -Pu 0 ${maxSweepAge} -x
${sweepLoc} > hard_5.msOut
```

We have named this thing `hard_5.msOut` because we are going to simulate 11 sweep locations (which we will label 0 – 10), so 5 is our central location. Finally, we need to add soft sweeps, which has just one additional parameter: the frequency of the sweeping mutation (which was previously evolving under drift alone) at the onset of selection:

```
# set our new variable
maxInitFreq=0.05

# run our command (this is all one line)
$neutralCmd -ws 0 -a ${alpha} -Pu 0 ${maxSweepAge} -x
${sweepLoc} -Pf 0 0.05 > soft_5.msOut
```

Here, the frequency of our adaptive allele at the onset of selection ranges uniformly from 0 (which `discoal` treats as 1/2*N*) to 0.05.

We now know how to simulate all of the training data that we need to train S/HIC. You can do this by simply running the following command:

```
./0_simulate_data.sh
```

If you get a permission error here you can instead run:

```
bash 0_simulate_data.sh
```

You will see a bunch of simulations showing up in `trainingSims` and `testSims`.

Note: The commands below require the version of `python` that has all of the necessary packages installed, which is the same one used for the jupyter notebooks, not the default pipeline on the cluster. So if you are running the commands below individually rather than using the bash scripts I have created then replace all calls to `python` with calls to `/opt/jupyterhub/bin/python`. Or, you can use a bash variable (e.g. `py=/opt/jupyterhub/bin/python` and then replace all python calls with `$py`).

Another note: in the next steps you will be using a piece of software called `diploSHIC`, which consists primarily of `python` scripts but also calls some `C` code for some of its slower operations. To get this to work you will have to move the pre-compiled library into your current working directory thusly (normally we wouldn't have to do this when installing `diploSHIC` to work with our own `python` install, but we are using the jupyter one so things are weird): `cp /home/schrider/diploSHIC/build/lib.linux-x86_64-3.6/shicstats.cpython-36m-x86_64-linux-gnu.so`.

Step 1: Calculating summary statistics and visualizing them

Now we need to calculate our feature vector, which contains bunch of statistics calculated within 11 sub-windows within each simulated region. This can be done using the `diploSHIC` software (present in my home directory) as follows:

```
mkdir trainingFvecLogs

# the stuff below is all one line
python /home/schrider/diploSHIC/diploSHIC.py fvecSim
--numSubWins=11 diploid trainingSims/neut.msOut.gz
trainingFvecs/neut.fvec &> trainingFvecLogs/neut.log
```

This command just calculates a bunch of statistics so it is not super interesting. For more information on how to use this command, you can type:

```
python /home/schrider/diploSHIC/diploSHIC.py fvecSim -h
```

To run it on all of our training and test data, you can simply run our second `bash` script:

```
./1_calculate_stats.sh
```

You should run this and verify that everything is working properly—it should print periodic messages showing its progress. But it will probably take a while to run

serially on every file (compute clusters come in handy for this step), so let's cheat! Go ahead and interrupt the process using CTR+C (you may have to hit these keys a bunch of times). Then, you can copy some pre-computed statistics that I have set aside in `/home/schrider/preCookedData/`:

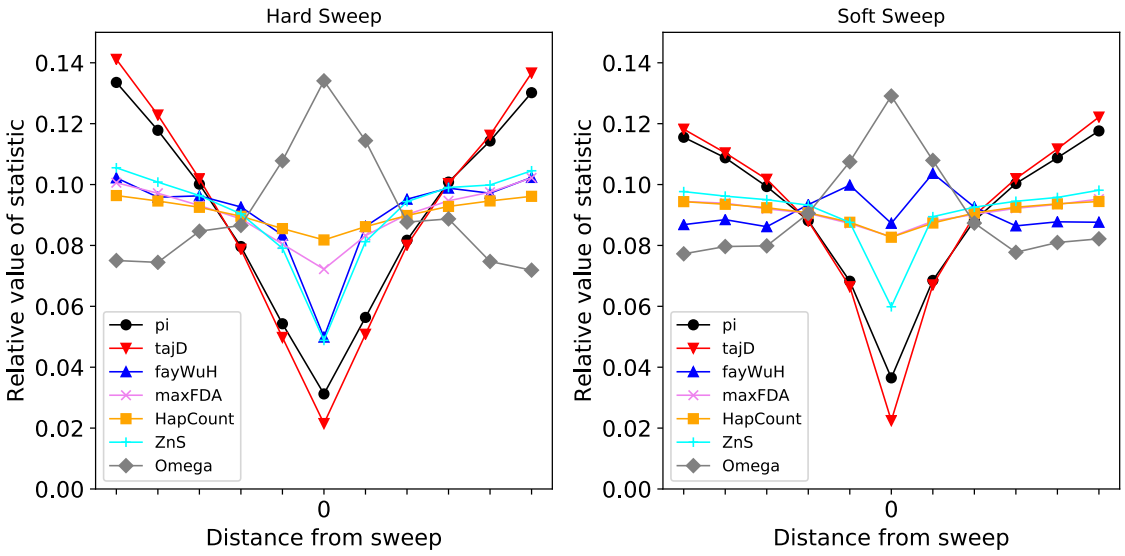
```
cp -r /home/schrider/preCookedData/trainingFvecs/ .
cp -r /home/schrider/preCookedData/testFvecs/ .
```

As an added bonus, these pre-computed statistics have more data (thousands instead of hundreds of reps).

Before moving on to the next step, we might want to visualize our feature vectors to see if they look at all like we should expect. I have written a handy script for doing this, which should also now be in your working directory. For now, let's just plot the cases where the sweep is in the center, and see if the spatial patterns of variation around these sweeps make sense. Generate these plots as follows:

```
python plotStatMeans.py trainingFvecs/hard_5.fvec
trainingFvecs/soft_5.fvec sweep_stats.pdf
```

When you open `sweep_stats.pdf` you should see something like the figure below:



Note how our statistics are recovering toward neutral expectations, but for soft sweeps the recovery seems to be happening much faster than for hard sweeps, which agrees with theory and intuition.

If what you see doesn't look anything like this, then we will have to troubleshoot before moving on—which in our context generally means experimenting with different simulation parameters until finding a parameter combination that is more

appropriate for your task. When using supervised machine learning methods, if there is something seriously wrong with your training data then there is no point in continuing.

Step 2: Training our classifier

Now we are ready to train our classifier. First, we have to compile our training data into sets of examples of each of our five classes. Recall that S/HIC's five classes are hard sweeps, regions linked to hard sweeps, soft sweeps, regions linked to soft sweeps, and neutrally evolving regions. The neutral evolution class corresponds to `neut.fvec`. Our hard and soft sweeps correspond to our `hard_5.fvec` and `soft_5.fvec` files, respectively. The hard-linked and soft-linked classes actually correspond to all of the `hard_$x.fvec` and `soft_$x.fvec` files where `x` is not equal to 5. We could just combine all of these together, but we generally (not always) want a balanced training set, so when combining these things we will have to downsample our "linked" examples. Only then can we run the script to train our classifier.

Both of these tasks can be completed by simply running `2_train_classifier.sh` but let's first take a look at what's in here by running

```
cat 2_train_classifier.sh
```

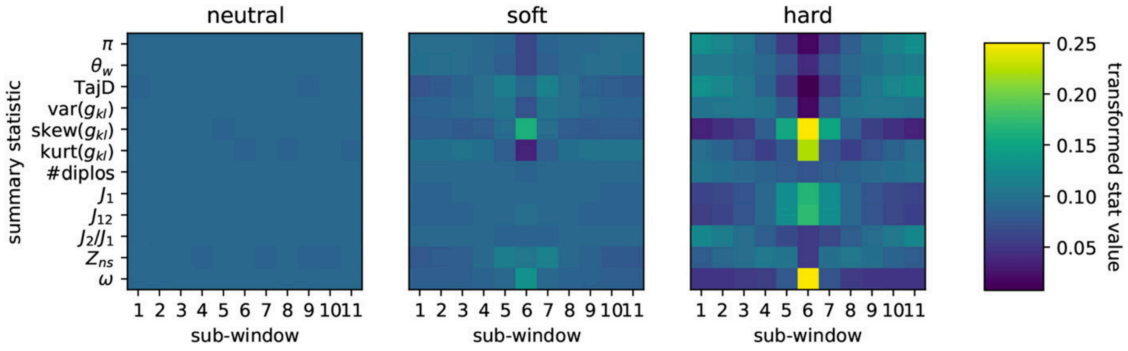
This prints the contents of the script to the terminal:

```
#!/bin/bash

# first create a directory to stash our training set
mkdir -p trainingSet

# then set a path to our diploSHIC script to shrink our
commands a bit
diploShicPath=/home/schrider/diploSHIC/diploSHIC.py

# step 1: build our training set (this is all one line)
python $diploShicPath makeTrainingSets
trainingFvecs/neut.fvec trainingFvecs/soft_
trainingFvecs/hard_ 5 0,1,2,3,4,6,7,8,9,10 trainingSet/

# step 2: train our classifier
python $diploShicPath train trainingSet/ trainingSet/ clf
```

The two commands at the bottom compile our training set and then train our classifier, respectively. The final command is the more interesting one. It runs `diploSHIC`'s "`train`" command which takes three arguments: the path to a training set, the path to a test set, and the name of the classifier to be created.

For now we are simply using our training data as the test set (which is not extremely helpful), and ignoring the accuracy on test data which `diploSHIC` outputs after completing its training. Don't fret about this for now, we will test our classifier soon enough!

The original `S/HIC` simply through our features into a type or random forest called an Extra-Trees Classifier. The `train` command of this newer version of `S/HIC` does something a bit fancier. First, it assembles our feature vector into a rectangular shape that looks something like this:



(Note: the above example was created using summary statistics for diploid data, whereas the examples we are calculating for this exercise are calculated from phased haploid data, so the set of statistics is not identical to what you would see in your `.fvec` files. You also may not know what some of these statistics are referring to. That's okay. Feel free to ask me later or see this paper: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5982824/.)

With our data in this arrangement, it is now possible to use a Convolutional Neural Network (a popular tool for image classification) to detect sweeps on the basis of this "image" of summary statistic values. It turns out this is slightly more accurate than the original random forest approach (any ideas as to why this might be?). Go ahead and train your network by running `./2_train_classifier.sh`

Once the training is completed, you will see two new files in your working directory: `clf.json` and `clf.weights.hdf5`. These files contain the neural network architecture and network weights for our classifier; we can load these when classifying additional data sets in the ensuing steps.

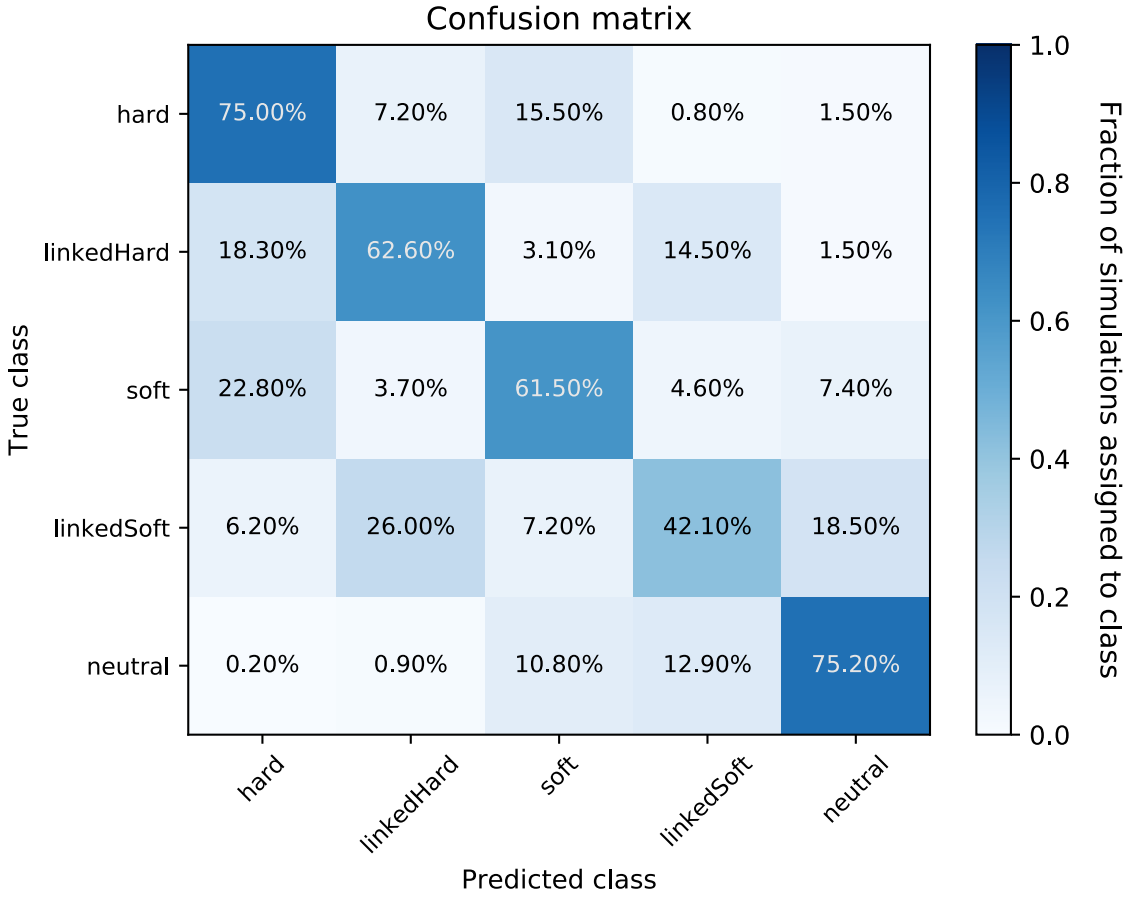
Step 3: Testing our classifier

Now for the most important step in the process: testing. This is especially important because machine learning methods are typically (but not always) less interpretable that model-based statistical approaches. So the only way to convince ourselves that our classifier is working is by extensive testing. It is not

surprising then that researchers in the field of machine learning are obsessive about testing and have annual competitions applying state-of-the art methods to a variety of standardized test sets to see where each of them excels or fails. We in pop gen could learn a lot from them!

For now we will just test on one simulated test set. Ideally we should also test on a few other simulated data sets, perhaps with parameter values depart from those used in generating our training data in various ways. In this way we can get a feel for our classifier's robustness to model misspecification.

Anyway, the commands for doing this are in `3_test_classifier.sh`. The key command is the script `testClassifierAndPlotConfusionMatrix.py` which we will use to visualize our accuracy (you don't have to look through this script as it is not the cleanest piece of code in the world—I blame `matplotlib`). The results will be written to `covfefe.pdf` (sorry, American joke). Go ahead and run the `bash` script, and let's take a look at the resulting plot, which should look something like the image below:



We call this a confusion matrix, not because it should be confusing for you, but because it shows the manner in which a classifier might tend to get confused. Spend some time with this figure to see how the classifier is behaving on the test

data. How do you think the classifier is doing? What are its strengths/weaknesses? Confusion matrices are a very useful tool so let me know if you are having trouble following it so I can help you out.

Step 4: Finally, run our classifier on real data

Now we are ready to apply this thing to some real data. `diploSHIC` takes input data in `VCF` format, and I have set aside a reasonably small dataset here: `/home/schrider/preCookedData/CEU50.chr2LCT.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.vcf.gz`

`4_apply_to_data.sh` has an example command for how to calculate summary statistics on these data, but this can take a while so I have gone ahead and calculated these for you. So you can go ahead and copy these over to your working directory as instructed in the `bash` script (again, you can take a look using `cat`) before running the classification step.

You should then have a file called `real_preds.txt` which contains the classifier's predictions. Let me know when you are here and let's all take a closer look at this together.

Supervised machine learning methods can be extremely powerful, flexible, and fun, but something that should be clear by this point is that it can take a bit of work to get them working on population genetic data. This is in large part because we have to simulate the training data—in this respect these methods are somewhat like ABC. We also have a training step that can sometimes be computationally arduous (though not for the `S/HIC` example above which was pretty quick). However, once we have a trained classifier, we can apply it to as many additional data sets as we want (provided our training data are not too misspecified), and the actual classification step is usually lightning-fast. So in practice the majority of the computational burden is imposed by simulating training/test data and calculating summary statistics (on both simulated and real data).