Detecting Selective Sweeps Using `S/HIC`

Preamble

In this lab exercise we will use a train a machine learning method capable of finding regions of the genome that have experienced recent positive selection. Although typical machine learning applications involve real-world training data for which the variable of interest is known—in our case, has there been a selective sweep or not—in evolutionary settings this information is typically hard to come by. So instead, we are going to simulate our training data. In this sense, our approach is fairly similar to that of approximate Bayesian computation: simulate a bunch of data and then make inferences by seeing which of our simulated examples "look like" our real data.

Although machine learning is still relatively new to population genetics, it is becoming increasingly more widely used and thus might be worth an exercise. But I think what really makes this exercise valuable are all of the pieces involved, including simulation (which is SUPER important in pop gen), plotting population genetic summary statistics (which can be very informative for understanding how different evolutionary scenarios can shape patterns of genetic variation), and getting our hands dirty with one of the most widely used population genomic data sets. So hopefully you find at least some of these parts useful. I have kind of erred on the side of dumping all of my relevant thoughts into this document along with the essentials/instructions, so if you don't understand every detail don't worry too much (although I am happy to discuss this stuff at length if you have any questions!).

This exercise is a somewhat stripped-down pipeline for using the `S/HIC` software to detect selective sweeps (specifically, the more recent `diploSHIC` version which is already installed for us). We will both train and test the software on some simulated data before moving on to a reasonably small real data set.

We will be doing our work here on the "Anthro" container, so log in using the credentials you received from the organizers and open a terminal window. Let's start by creating a working directory for this exercise. You could just call it your name (in case you are using a shared file system):

```
mkdir danSchrider
cd danSchrider
```

Now, there is some code we need to grab from GitHub, so let's clone the repository for this exercise.

```
git clone https://github.com/SchriderLab/selectionScanExercise.git
```

For simplicity, let's just use the GitHub repository directory as our working directory (probably not the best practice in general).

```
cd selectionScanExercise
```

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 in our container.

Brief conceptual description of S/HIC: this method seeks to detect two types of selective sweeps (hard and soft), and to 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). For more info see the S/HIC paper see https://doi.org/10.1371/journal.pgen.1005928.

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 $\theta=4Nu$ and $\rho=4Nr$ 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 #4*N*mutation rate per site*number of sites
rho=110 #4*N*recombination rate per site*number of sites

# 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, so copy and
# paste with care! 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, for the uninitiated, this was 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 (the leftmost end of our simulated chromosome/region) and 1 (the rightmost end of our simulated chromosome/region), 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`.

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 (already installed) as follows:

```
mkdir trainingFvecLogs trainingFvecs
```

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

This command just calculates a bunch of population genetic summary statistics in different windows surrounding the region we wish to classify. Some of these tell us about skews in the site frequency spectrum, others tell us about linkage disequilibrium (the correlation of alleles across individuals in the sample, which increases during and a sweep). So together the resulting "feature vector" should do a decent job of capturing the signatures of selection in regions experiencing sweeps. As an aside, the high-dimensional nature of this summary of diversity is the motivation for using machine learning here, as such methods are well-suited for high dimensional data.

Anyway, for more information on how to use this command, you can type:

```
python ~/libraries/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 the `preCookedData/` directory within the GitHub repository:

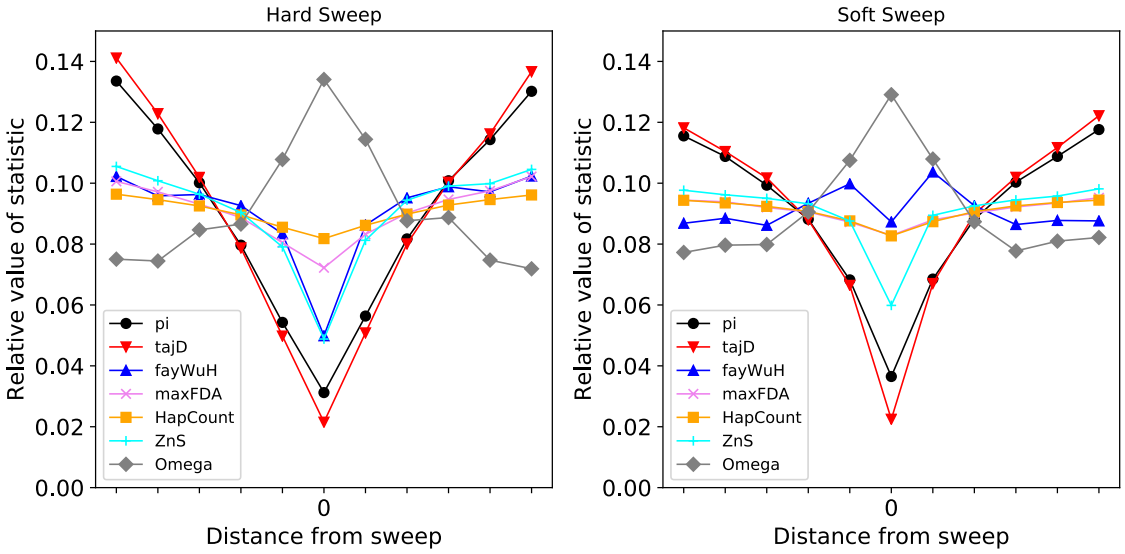
```
cp -r preCookedData/trainingFvecs/ .
cp -r preCookedData/testFvecs/ .
```

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:

```
# this is all one line
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 at the sweep site (center of the plot), we are seeing some extreme values of some of these statistics. For example, pi (average nucleotide diversity) is very low at the sweep site. As we move away from the sweep site (toward the left and right ends of the plots), 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—overall soft sweeps take much longer to complete because they have a neutral phase, and therefore more recombination events have taken place during the course of the sweep.

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 or approximate Bayesian computation, if there is something seriously wrong with your training data/reference table 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

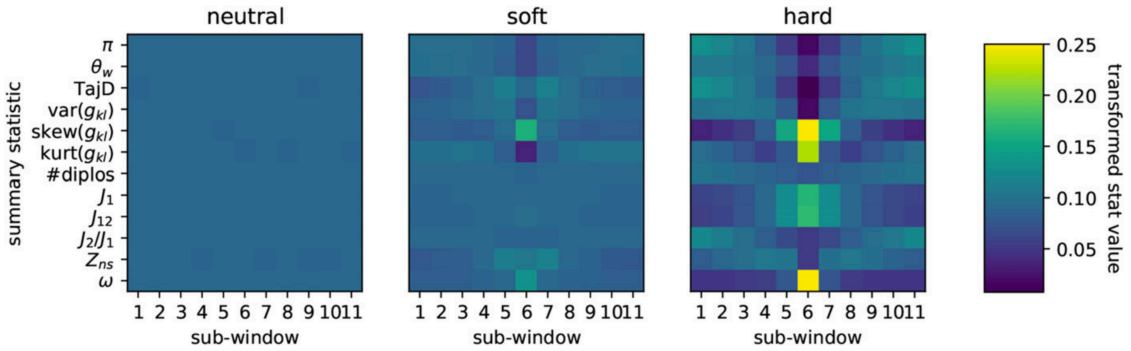
```
cat 2_train_classifier.sh
```

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/libraries/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.

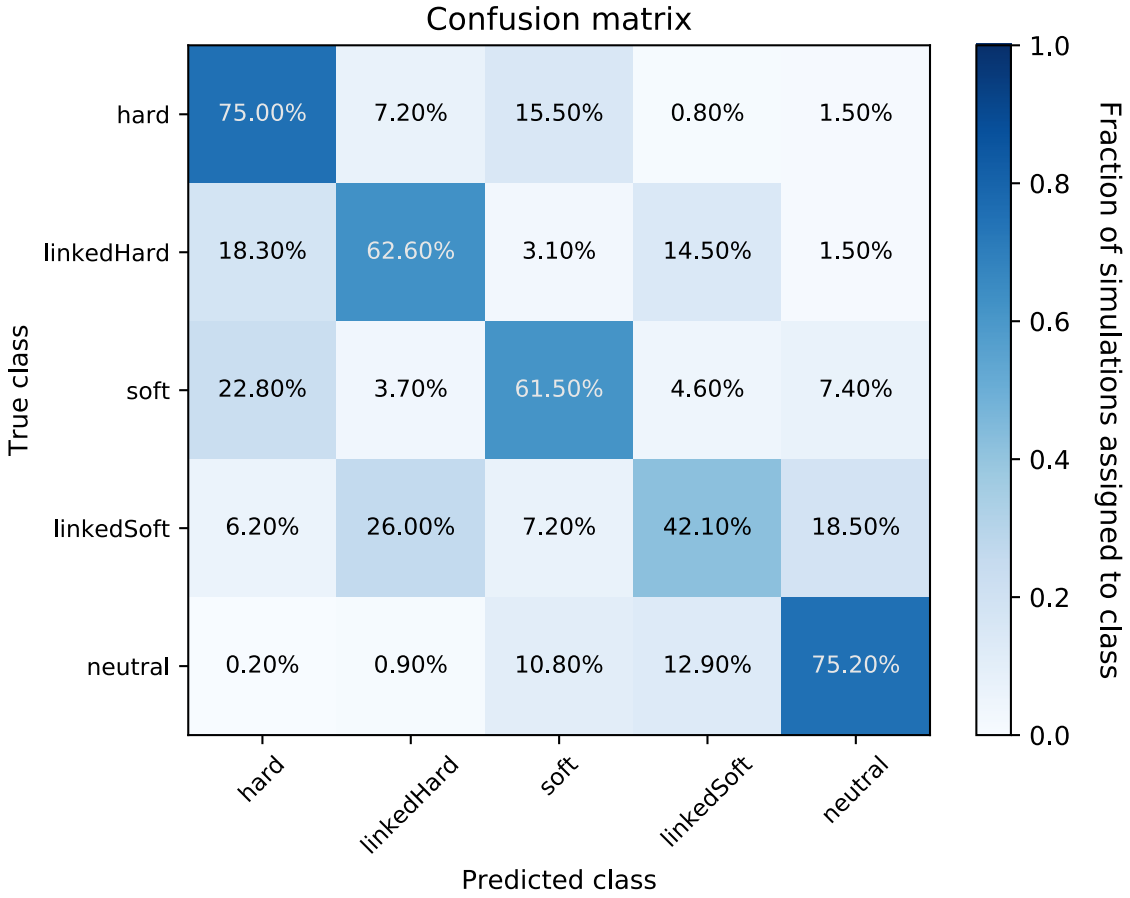
Step 3: Testing our classifier

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 (although we should probably do this more extensively with more conventional methods too). 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, old 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 and 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.

Note: if you dig in the bash scripts, you will note that the above step, and the one below, are actually using a different `diploSHIC.py` script. Don't be alarmed!

This is because of a compatibility issue with `diploSHIC` (since resolved, but not in our container), so we are pulling in a new one as a workaround.

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: `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. The key fields in this tab delimited file, for our purposes, are the first three (which show the coordinates in version hg19 of the human genome), and the fifth, which tells us which class was predicted. You will probably see a region in which several adjacent or nearly adjacent windows are classified as sweeps. If you are curious you can look these up in the UCSC Genome Browser (https://genome.ucsc.edu/cgi-bin/hgGateway; be sure to use version hg19) to try to figure out which genes are in this vicinity. Any idea what could be going on here?

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—again, 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 simulated under appropriate parameters), 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).