CSE280A
Assignment 2

Andrey Bzikadze

List 1 | ①

① On List 2 I attach code for this problem (Python 3)

Notes about the code:

1) I've erased columns with less than 15 ones or zeros. The reason is in applying χ^2 test. It requires $n_{ij} \geq 5$ where n_{ij} is the ij-th observed frequency. In our case n_{00} (n_{01}) - is the number of zeros in the first column. So, hopefully if $n_{00} + n_{01} \geq 15$ (and all others) the $n_{ij} \geq 5$. But that's not guaranteed, however, we don't need to be very strict.

2) I didn't implement Yates correction for pvalues in χ^2 test, but it could be beneficial. as w/o it χ^2 could be radical

3) I've vectorized code, so it works in less than 5 sec for the whole matrix (w/o erasing columns). Straigh forward implementation would require > 1 min.

4) The χ^2 has 1 degree of freedom // For counting table of size $K \times L$ χ^2 $(K-1)(L-1)$ //.

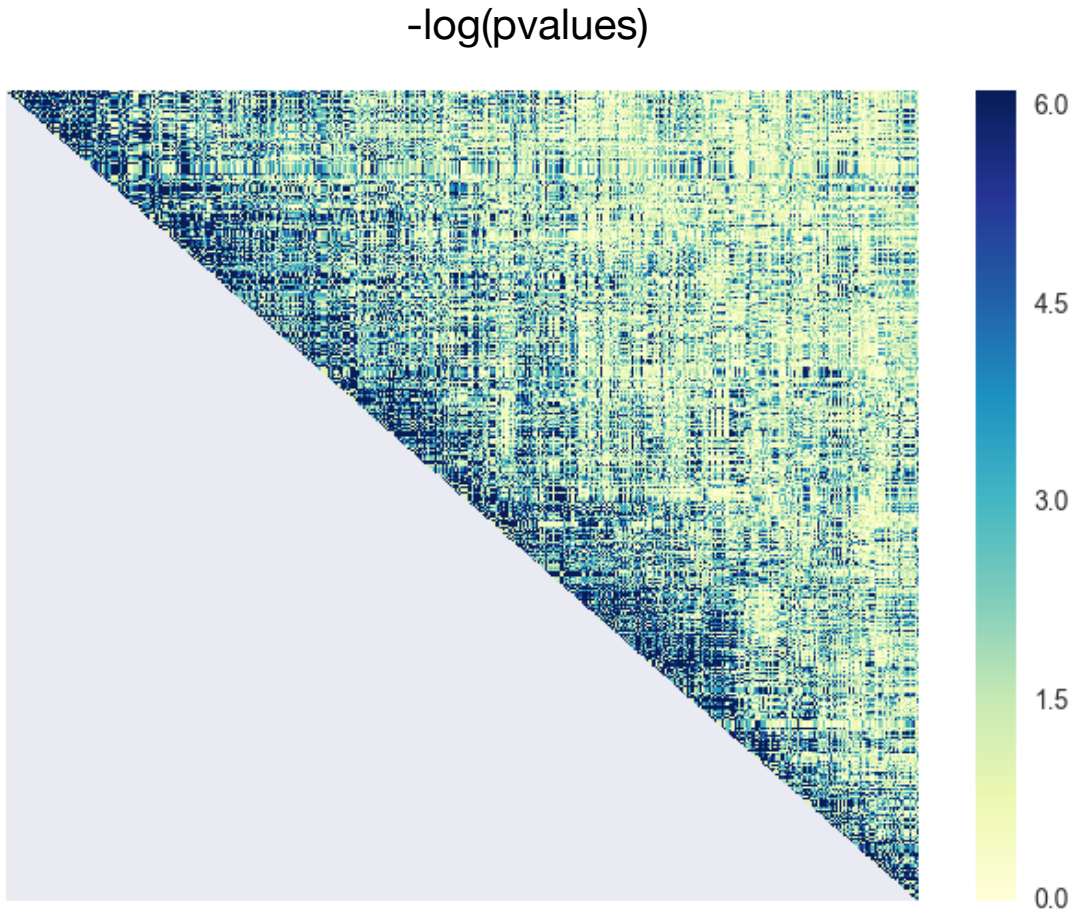
On page 34 we see $-\log(\text{pvalues})$ and D' plots. Generally, larger values of both metrics correspond to higher LP.

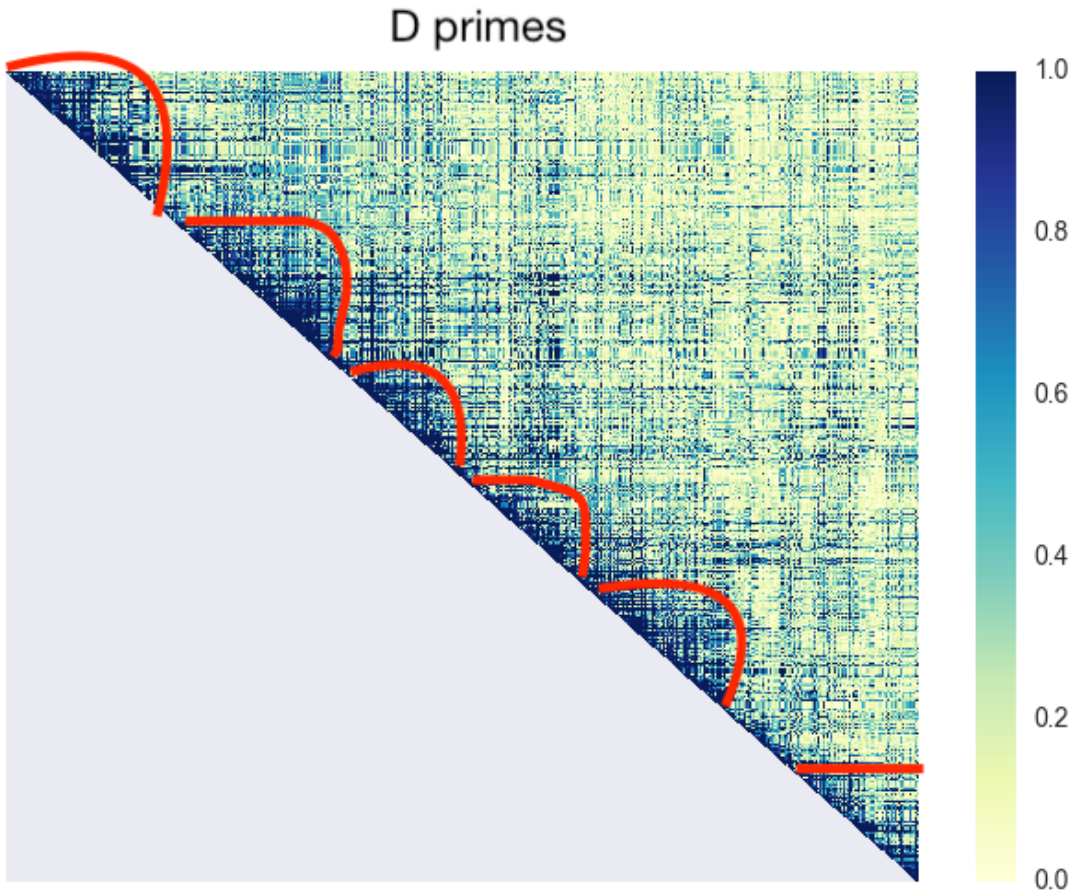
D primes are a little better representation of the fact that higher LP is seen closer to diagonal (=> closer mutations as they are ordered in the original file).

I tried to catch putative haplotypes on Dprime plot with red lines. However, it's still a pretty wild guess of the total # as 5-6.

```
import numpy as np
import scipy as sp
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm


def DprimeFast(df, Yates=False):
    p1 = np.mean(df, axis=0)
    p0 = 1-p1
    nrow, ncol = df.shape
    Dprimes = np.zeros((ncol, ncol), dtype=float)
    Chi2sStat = np.zeros((ncol, ncol), dtype=float)
    Chi2sPv = np.zeros((ncol, ncol), dtype=float)
    zeros = df == 0
    print("DprimeFast started")
    for i in range(ncol):
        if i % 500 == 0:
            print(i, ncol)
        pair_zeros = np.mean(zeros[:, i] * zeros.T, axis=1)
        D = pair_zeros - p0[i] * p0
        Dmax = np.zeros(ncol)
        Dmax[D > 0] = np.min([p0[i] * p1, p1[i] * p0], axis=0)[D >
0]
        Dmax[D <= 0] = -np.min([p0[i] * p0, p1[i] * p1], axis=0)[D
<= 0]
        Dprimes[i, :] = D / Dmax
        Chi2sStat[i, :] = nrow * D**2 / (p0[i] * p1[i] * p0 * p1)
        Chi2sPv[i, :] = \
            -np.log(1 - sp.stats.chi2.cdf(Chi2sStat[i, :], 1))
    Chi2sPv += 1e-10
    print("DprimeFast done")
    return Dprimes, Chi2sStat, Chi2sPv


def FilterInput(df):
    n1 = np.sum(df, axis=0)
    nrow = df.shape[0]
    return df[:, (15 <= n1) * (n1 <= nrow - 15)]


def main():
    inp = "pop2.txt"
    with open(inp, 'r') as f:
        df = np.array([list(x.strip()) for x in
f.readlines()]).astype(int)
    print(df.shape)
    df = FilterInput(df)
    print(df.shape)
    Dprimes, Chi2sStat, Chi2sPv = DprimeFast(df)
    Chi2sPv[Chi2sPv == np.inf] = 1

    mask = np.tril(np.ones_like(Dprimes))
    print(np.min(Chi2sPv))
```

```
    sns.heatmap(Chi2sPv, xticklabels=False, yticklabels=False,
mask=mask,
                cmap="YlGnBu",
                vmin=np.min(Chi2sPv), vmax=np.max(Chi2sPv) / 6)
    # plt.savefig('1.pdf', format='pdf')
    nrow, ncol = df.shape
    print("Dprime exporting")
    with open("1_dprime.tsv", "w") as f:
        for i in range(ncol - 1):
            for j in range(i + 1, ncol):
                print("%d\t%d\t%f\t%f\t%f" % (i, j, Dprimes[i, j],
Chi2sStat[i, j], Chi2sPv[i, j]),
                      file=f)


if __name__ == "__main__":
    main()
```


-log(pvalues)
6.0
4.5
3.0
1.5
0.0


D primes
1.0
0.8
0.6
0.4
0.2
0.0

CSE 280A
Assignment 2

Andrey Bzikadze

List 5 || (2)

(2) I've only done a) as b) is not required for Assignment 2.

The simulator code is on page 6.

The tree topology is generated in linear time (see function simulate_tree). The distribution of edges lens is different for exponent. increasing N. The simulation is done by inversing the cumulative distribution function.

Formally let $\ell_T = \ell_T^{(k)}$ be the length of the edge starting at some generation T ($\Rightarrow N_T = e^{-\alpha T} N_0$) // for k individuals.

$$P(\ell_T = 1) = \binom{k}{2} / N_T, \quad P(\ell_T = 2) = \frac{\binom{k}{2}}{N_T e^{-\alpha}} \left(1 - \frac{\binom{k}{2}}{N_T e^{-\alpha}}\right)$$

$$\Rightarrow P(\ell_T = \ell+1) = \frac{\binom{k}{2}}{N_T e^{-\alpha\ell}} \prod_{s=1}^{\ell} \left(1 - \frac{\binom{k}{2}}{N_T e^{-\alpha s}}\right).$$

The simulation idea comes from the fact that

$$\frac{P(\ell_T = \ell+1)}{P(\ell_T = \ell)} = e^{-\alpha}\left(1 - \frac{\binom{k}{2}}{e^{-\alpha k} N_T}\right)$$ — a very simple close form that allows one to go from $P(\ell_T = \ell)$ to $P(\ell_T = \ell+1)$ in short time.

The mutations are simulated by Poisson as was described on the lecture.

Simulator saves the tree in newick format and SNP matrix as np.savetxt result.

On page 7 I also saved SNP matrix and a tree for $n = 10$.

```
from random import randint, seed, random
from math import exp
import numpy as np
from numpy.random import poisson
from argparse import ArgumentParser


class Vertex:
    def __init__(self, l=None, r=None, ind=None, gen=None):
        self.l = l
        self.r = r
        self.ind = ind
        self.gen = gen
        self.mut = 0

    def __str__(self):
        if self.l is None:
            return str(self.ind)

        return "(%s:%d, %s:%d)" % (self.l.__str__(), self.gen -
self.l.gen,
                                   self.r.__str__(), self.gen -
self.r.gen)


class Tree:
    def __init__(self, root, total_mut, n):
        self.root = root
        self.total_mut = total_mut
        self.n = n

    @classmethod
    def simulate_tree(cls, n, N, theta, alpha):
        def pseudogeometric(N_start, ealpha, size):
            k = 0
            count = .5 * size * (size - 1)
            s = r = count / N_start
            beta = random()
            kealpha = 1
            while (beta > s):
                k += 1
                kealpha *= ealpha
                r *= (1 - count / (kealpha * N_start)) / ealpha
                s += r
            return k + 1  # as at least one generation

        N *= 2  # Diploidy
        mu = theta / (2 * N)
        ealpha = exp(-alpha)

        nodes = [Vertex(ind=i, gen=0) for i in range(n)]
        cind = n
        total_mut = 0
        while len(nodes) > 1:
```

```
            r = randint(0, len(nodes) - 1)
            l = randint(0, len(nodes) - 2)
            if l >= r:
                l, r = r, l + 1

            gen_start = max(nodes[l].gen, nodes[r].gen)
            N_start = N * ealpha**gen_start

            lp = pseudogeometric(N_start=N_start, ealpha=ealpha,
size=len(nodes))
            parent = Vertex(l=nodes[l], r=nodes[r], ind=cind,
                            gen=gen_start + lp)
            nodes[l].mut = poisson(lam=mu * (parent.gen -
nodes[l].gen), size=1)[0]
            nodes[r].mut = poisson(lam=mu * (parent.gen -
nodes[r].gen), size=1)[0]
            total_mut += nodes[l].mut + nodes[r].mut
            nodes[l] = parent
            nodes[r], nodes[-1] = nodes[-1], nodes[r]
            nodes.pop()
            cind += 1

        return cls(root=nodes[0], total_mut=total_mut, n=n)

    def __str__(self):
        return self.root.__str__()

    def tree2SNPmatrix(self):
        m = np.zeros((self.n, self.total_mut ))
        print("total_mut", self.total_mut)

        self.cur_pos = 0
        def set_mutation(node, indeces):
            new_mut = list(range(self.cur_pos, self.cur_pos +
node.mut))
            self.cur_pos += node.mut
            if node.l is None:
                m[node.ind, indeces + new_mut] = 1
            else:
                set_mutation(node.l, indeces + new_mut)
                set_mutation(node.r, indeces + new_mut)

        set_mutation(self.root, list())
        self.cur_pos = None
        return m

def main():
    parser = ArgumentParser()
    parser.add_argument("-n", dest="n", type=int, required=True)
    parser.add_argument("-N", dest="N", type=int, required=True)
    parser.add_argument("--theta", dest="theta", type=float,
required=True)
    parser.add_argument("--alpha", dest="alpha", type=float,
required=True)
```

```
    parser.add_argument("--output_tree", dest="output_tree",
required=True)
    parser.add_argument("--output_matrix", dest="output_matrix",
required=True)
    params = parser.parse_args()
    n, N, theta, alpha = params.n, params.N, params.theta,
params.alpha
    print("n = %d, N = %d, theta = %f, alpha = %f" % (n, N, theta,
alpha))
    tree = Tree.simulate_tree(n, N, theta, alpha)
    with open(params.output_tree, 'w') as f:
        print(tree, file=f)
    np.savetxt(params.output_matrix, tree.tree2SNPmatrix(),
fmt="%d")


if __name__ == "__main__":
    main()
```

1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0

((0:70244, (2:25791, 6:25791):44453):1614263, ((((1:81742, (4:23025, 9:23025):58717):64450, 8:146192):132404, 7:278596):397099, (3:14872, 5:14872):660823):1008812)

CSE 280A
Assignment 2
Artyom Barmazel
List 8. ‖ ③

③ I. The hint:

$$p_{ki} = \frac{\binom{n-i-1}{k-2}}{\binom{n-1}{k-1}}$$

← the same stars and bars. We put aside i elements to put to one specific tree. All others n−i are put in k−1 trees. But S&B theorem: $\binom{n-i-1}{k-2}$

← stars and bars. Place n (n indiv.) obj between k bars (k trees)

By stars & bars theorem: $\binom{n-1}{k-1}$

Note: we don't take into account the topology of trees. Only placing vertices to various trees

each bar (in each tree)

II. $$p_{ki} = \frac{\binom{n-i-1}{k-2}}{\binom{n-1}{k-1}} = \frac{(n-i-1)!}{(k-2)!\,(n-i-k+1)!} \cdot \frac{(k-1)!\,(n-k)!}{(n-1)!} =$$

$$= \frac{(n-k)!}{(n-k-i+1)!\,(i-1)!} \cdot \frac{(i-1)!}{(n-1)!\,(n-i-1)!} \cdot (k-1) = \boxed{\binom{n-k}{i-1} \Big/ \binom{n-1}{i} \cdot \frac{k-1}{i}}$$

$$\frac{1}{\binom{n-1}{i}} \cdot i$$

III. To get $\mathbb{E}\, n_i = \frac{\theta}{i}$ we need to write n_i in an indicator fashion. Let ζ_{kl} be the size of l-th lineage at epoch k. Also let n_{kl} be # of mutations occured in l-th lineage at epoch k.

$$\Rightarrow n_i = \sum_{k=2}^{n} \sum_{l=1}^{k} \mathbb{1}[\zeta_{kl} = i] \cdot n_{kl}$$

$k=2$: all epochs; $\mathbb{1}[\zeta_{kl}=i]$: we need the epoch to be the size i

$$\Rightarrow \mathbb{E}\, n_i = \sum_{k=2}^{n} \sum_{l=1}^{k} \underbrace{P(\zeta_{kl}=i)}_{p_{ki}} \underbrace{\mathbb{E}(n_{kl} \mid \zeta_{kl}=i)}_{\frac{k(k-1)}{\binom{n-1}{i}}} = \frac{1}{2} \sum_{k=2}^{n} \frac{k(k-1)}{\binom{n-1}{i}} \binom{n-k}{i-1} \theta =$$

$$= \frac{\theta}{i}$$

CSE 280 A
Assignment 2
Andrey Bokodev
List 9 || ④, ⑤

④ By valid I suppose we mean unbiased.

① $\theta_{FL} = m_1$

$E\theta_{FL} = Em_1 = \frac{\theta}{1} = \theta \Rightarrow$ unbiased

② $\theta_W = \frac{1}{\sum_{i=1}^{n-1} \frac{1}{i}} \sum_{i=1}^{n-1} m_i$

$E\theta_W = \frac{1}{\sum \frac{1}{i}} \cdot \sum \frac{\theta}{i} = \theta \Rightarrow$ unbiased

③ $\theta_L = \frac{1}{n-1} \sum_{i=1}^{n-1} i m_i$

$E\theta_L = \frac{1}{n-1} \sum i \frac{\theta}{i} = \theta \Rightarrow$ unbiased

④ $\theta_\pi = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} i(n-i) m_i$

$E\theta_\pi = \frac{2}{n(n-1)} \underbrace{\sum i(n-i)}_{\frac{n(n-1)}{2}} \frac{\theta}{i} = \theta \Rightarrow$ unbiased

⑤ $\theta_H = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} i^2 m_i$

$E\theta_H = \frac{2}{n(n-1)} \underbrace{\sum i \cdot i}_{\binom{n}{2}} \frac{\theta}{i} = \theta \Rightarrow$ unbiased

---

⑤ I suppose that we have set of reads with (nearly) uniform coverge across genome and the same sequencing depth for all individuals. + ~~reads are the~~

We can build the „SNP matrix" but for reads. (sort of)

The Tajima estimate could be calculated between reads in such a way: $\theta_T = \frac{1}{\binom{n}{2}} \sum_{i,j} T_{ij} \frac{l_g}{l_r}$

where l_g is the len of genome and l_r is the len of reads; #reads; Hamming distance between reads

$\theta_\pi = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} i(n-i) m_i$ where n - # of reads / coverage depth per individual and m_i - # mutations in ith indiv. / Coverage depth. per ind.