replication

February 16, 2017

1 Problem description and notation

We have N hard drives we want to use for backup. Each one has capacity C_i for $i = 1 \dots N$. For each file to be backed up we want to store R replicas and we want the hard drives to be more or less used according to their capacity in such a way that, after some time being used, all of them are, for example, at 60% of their capacity, instead of one being fully filled and others empty.

When a new file arrives we must make the decision of which set of hard drives are to be used. Let's represent the result of this decision by a vector s made of R components, where the first component s_1 represents the hard drive chosen for the first replica, s_2 the one chose for the second replica, etc…

$$\boldsymbol{s} = (s_1, s_2, \dots, s_R)$$

We are going to use a probabilistic algorithm to choose $\boldsymbol{s}$. If we call $\boldsymbol{S}$ the random variable over all possible selections $\boldsymbol{s}$ the probability of a particular selection will be written as:

$$P(\boldsymbol{S} = \boldsymbol{s})$$

Let's call also U_i the random variable that has value 1 when drive i is used for a particular file and 0 otherwise:

$$U_i = 1 \iff i = s_j \text{ for any } j$$

Abusing the notation a little we will reuse the same symbol but in bold for the set of selections that use hard drive i:

$$\boldsymbol{U}_i = \{\boldsymbol{s} | i = s_j \text{ for any } j\}$$

The probability distribution of U_i, for all i between 1 and N is determined by the probability distribution of $\boldsymbol{S}$, since:

$$P(U_i = 1) = P(S_1 = i \cup S_2 = i \cup \cdots \cup S_R = i)$$

These probabilities determine at which rate drives are filled, since after M files replicated the expected number of files on each hard drive is $P(U_i = 1)M$. Since the total number of files is RM and the expected number of total files is $\sum_{i=1}^{N} P(U_i = 1)M$ we have necessarily that:

$$\sum_{i=1}^{N} P(U_i = 1) = R$$

We want to **design** the probability distribution $P(\boldsymbol{S} = \boldsymbol{s})$ so that the rate at which hard drives are filled it's proportional to their capacity.

Let's call C the total capacity of the hard drives:

$$C = \sum_{i=1}^{N} C_i$$

Let's call also c_i the fraction of the total capacity that hard drive i provides:

$$c_i = \frac{C_i}{C}$$

Then we want for some constant k:

$$P(U_i = i) = kc_i$$

Since all the above probabilities must sum to R as we have seen before it must be therefore that $k = R$ and so:

$$P(U_i = i) = Rc_i$$

Let's suppose that we are going to choose it over a familiy of distributions π parameterized with some parameters θ (which could be a vector of them, but for simplicity, we will assume just one parameter):

$$P(\boldsymbol{S} = \boldsymbol{s}) = \pi(\boldsymbol{s}; \theta)$$

We want to find the optimal value θ^* for the parameters where optimality is achieved when the probabilities of $P(U_i = 1)$ are equal, or as a equal as possible, to Rc_i. Let's define:

$$q_i = \frac{1}{R} P(U_i = 1)$$

Then both q_i and c_i sum to one and can be interpreted as two probabilities distributions, where we want q_i to be close to c_i. We can use the Kullback-Leibler divergence as our loss function L, that we want to minimize:

$$L = \sum_{i=1}^{N} c_i \log \frac{c_i}{q_i}$$

And so:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, L$$

Let's suppose that there are constraints on which selections $\boldsymbol{s}$ are acceptable. We will condense all this constraints on a single function $g(\boldsymbol{s})$ that returns 1 when the selection is valid and 0 otherwise:

$$g(\boldsymbol{s}) = 1 \iff \boldsymbol{s} \text{ is a valid selection}$$

Let's call $\boldsymbol{G}$ the set of all valid selections:

$$\boldsymbol{G} = \{\boldsymbol{s} | g(\boldsymbol{s}) = 1\}$$

The probability distribution of valid assignments is:

$$P(\boldsymbol{S}=\boldsymbol{s}|g(\boldsymbol{s})=1)=\frac{P(\boldsymbol{S}=\boldsymbol{s})g(\boldsymbol{s})}{\sum_{\boldsymbol{s}\in\boldsymbol{G}}P(\boldsymbol{S}=\boldsymbol{s})}$$

And the probability of use of each drive is:

$$P(U_i=1)=\sum_{\boldsymbol{s}\in U_i}P(\boldsymbol{S}=\boldsymbol{s}|g(\boldsymbol{s})=1)=\frac{\sum_{\boldsymbol{s}\in\boldsymbol{G}\cap\boldsymbol{U}_i}P(\boldsymbol{S}=\boldsymbol{s})}{\sum_{\boldsymbol{s}\in\boldsymbol{G}}P(\boldsymbol{S}=\boldsymbol{s})}$$

Let's call to simplify further manipulations:

$$Z_{\boldsymbol{A}}(\theta)=\sum_{\boldsymbol{s}\in\boldsymbol{A}}P(\boldsymbol{S}=\boldsymbol{s})$$

And it's gradient, which we will use later is:

$$\frac{dZ_{\boldsymbol{A}}}{d\theta}=\sum_{\boldsymbol{s}\in\boldsymbol{A}}\frac{dP(\boldsymbol{S}=\boldsymbol{s})}{d\theta}$$

Then we have that:

$$P(U_i=1)=\frac{Z_{\boldsymbol{G}\cap\boldsymbol{U}_i}(\theta)}{Z_{\boldsymbol{G}}(\theta)}$$

The gradient of the loss function is:

$$\frac{dL}{d\theta}=-\sum_{i=1}^{N}c_i\frac{d}{d\theta}\log P(U_i=1)=-\sum_{i=1}^{N}c_i\left(\frac{1}{Z_{\boldsymbol{G}\cap\boldsymbol{U}_i}}\frac{dZ_{\boldsymbol{G}\cap\boldsymbol{U}_i}}{d\theta}-\frac{1}{Z_{\boldsymbol{G}}}\frac{dZ_{\boldsymbol{G}}}{d\theta}\right)$$

With the loss function and it's gradient computed we can use any optimization method to find the optimal policy.

```
In [1]: import itertools
        from abc import ABCMeta, abstractmethod

        import numpy as np


        def add_constraints(constraints):
            """Given a list of constraints combine them in a single one.

            A constraint is a function that accepts a selection and returns True
            if the selection is valid and False if not.
            """
            if constraints is None:
                return lambda s: True
            else:
                return lambda s: all(constraint(s) for constraint in constraints)


        class BasePolicyFamily(metaclass=ABCMeta):
            @abstractmethod
            def log_prob(self, s, params):
                """Compute the log probability of the selection 's' given the
                parameters 'params'"""
                pass

            @abstractmethod
            def jac_log_prob(self, s, params):
```

```
        """Compute the jacobian of the log probability relative to the
        parameters evaluated at the selection 's' and the parameters 'params'"""
        pass

    @abstractmethod
    def sample(self, params):
        """Extract just one random selection.

        The result should be a numpy array with as many elements as replicas.
        The i-th component represents the hard drive selected for the i-th replica.
        """
        pass

    @abstractmethod
    def params_point(self):
        """Give an example of valid parameters.

        This method is used as an starting point for optimization methods and testing.
        """
        pass

    def normalize_params(self, params):
        """Return new params making sure they fall betwenn valid bounds

        The new params should be as close as possible as the ones provided.
        For example it the parameters represent a probability distribution
        they should sum to 1.
        """
        return params

    def sample_constrained(self, params, size, constraints=None):
        samples = []
        g = add_constraints(constraints)
        while len(samples) < size:
            sample = self.sample(params)
            if g(sample):
                samples.append(sample)
        return np.stack(samples)

    def loss(self, params, constraints=None):
        g = add_constraints(constraints)
        z_g = 0
        z_u = np.zeros(len(self.c))
        jac_z_g = np.zeros_like(params)
        jac_z_u = np.zeros((len(self.c), len(params)))
        for s in itertools.permutations(np.arange(len(self.c)), self.R):
            if g(s):
                p = np.exp(self.log_prob(s, params))
                q = p*self.jac_log_prob(s, params)
                z_g += p
                jac_z_g += q
                for i in s:
                    z_u[i] += p
                    jac_z_u[i, :] += q
        L = (np.log(self.c) - np.log(z_u/z_g/self.R)) @ self.c
        J = -((jac_z_u.T/z_u).T - jac_z_g/z_g).T @ self.c
        return L, J

    def build_selector(self, params, constraints=None):
        """Builds a function accepting no arguments that returns a valid selection.

        A selection is represented by an array with as many components as hard drives.
        A zero entry means the hard drive is unused, otherwise it says what replica
        is stored there.
        """
        g = add_constraints(constraints)
        def selector():
            sample = None
            while sample is None:
                candidate = self.sample(params)
                if g(candidate):
                    sample = candidate
            return sample

        return selector
```

2 A simple policy family

The probability of a selection when no hard drive is repeated in the selection is:

$$P(\boldsymbol{S} = \boldsymbol{s}) = p_1^{s_1} \frac{p_2^{s_2}}{1 - p_2^{s_1}} \cdots \frac{p_R^{s_R}}{1 - p_R^{s_1} - \cdots - p_R^{s_{R-1}}}$$

If the hard drive appears more than once then:

$$P(\boldsymbol{S} = \boldsymbol{s}) = 0$$

In this way the probability distribution already satisfies the constraint that no hard drive stores more than one replica of the same file.

We will also impose that:

$$\boldsymbol{p}_2 = \boldsymbol{p}_3 = \cdots = \boldsymbol{p}_R$$

In this way we just have two vectors of unknowns, p_1 and p_2, of size N each one.

If we want that the first replica is equally distributed on all hard drives then it must be that:

$$p_1^i = \frac{1}{N}$$

However, since:

$$P(\boldsymbol{S} = \boldsymbol{s}) = P(S_1 = s_1) + P(S_2 = s_2, \ldots, S_R = s_R) = p_1^{s_1} + P(S_2 = s_2, \ldots, S_R = s_R)$$

Then computing the probability of using hard drive i:

$$P(U_i = 1) = p_1^i + \sum_{s \in \boldsymbol{U}_i} P(S_2 = s_2, \ldots, S_R = s_R) = p_1^{s_1} + P(S_2 = s_2, \ldots, S_R = s_R)$$

Since we want $P(U_i = 1) = Rc_i$ and since probabilities are positive don't have solution if for some i we have that:

$$c_i < \frac{1}{RN}$$

Or equivalently:

$$C_i < \frac{1}{R}\frac{C}{N}$$

In this case we can try to distribute it as uniformly as possible. Let's call L the set of hard drives that are too small:

$$L = \left\{ i \mid c_i < \frac{1}{RN} \right\}$$

For the hard drives in this set ($i \in L$) we define:

$$p_1^i = Rc_i$$

And for the hard drives not in this set ($i \notin L$):

$$p_1^i = \frac{1}{N - |L|}\left(1 - \sum_{j \in L} p_1^j\right)$$

And so since the $\boldsymbol{p}_1$ vector is fixed by the first replica distribution constraint the only remaining parameters are $\boldsymbol{p}_2$

```
In [2]: class SimplePolicyFamily(BasePolicyFamily):
            def __init__(self, capacities, n_replicas):
                self.c = np.array(capacities) / np.sum(capacities)
                self.R = n_replicas
                # Initialize the probability of choosing the first hard drive
                L = self.c < 1/(self.R*len(self.c))
                M = np.logical_not(L)
                self.p_1 = np.empty_like(self.c)
                self.p_1[L] = self.R*self.c[L]
                self.p_1[M] = 1/np.sum(M)*(1 - np.sum(self.p_1[L]))

            def params_point(self):
                return np.copy(self.p_1)

            def normalize_params(self, params):
                return params/np.sum(params)

            def log_prob(self, s, params):
                p_2 = params
                logP = np.log(self.p_1[s[0]])
                for r in range(1, self.R):
                    logP += np.log(p_2[s[r]])
                    d = 1
                    for i in range(r):
                        d -=  p_2[s[i]]
                    logP -= np.log(d)
                return logP

            def jac_log_prob(self, s, params):
                p_2 = params
                jac = np.zeros_like(self.c)
                for r in range(1, self.R):
                    jac[s[r]] += 1.0/p_2[s[r]]
                    d = 1
                    jac_d = np.zeros_like(self.c)
                    for i in range(r):
                        d -= p_2[s[i]]
                        jac_d[s[i]] -= 1
                    jac -= jac_d/d
                return jac

            def sample(self, params):
                p_2 = params
                i = np.random.choice(len(self.p_1), p=self.p_1)
                selection = [i]
                p = np.copy(p_2)
                for r in range(1, self.R):
                    p[i] = 0
                    p /= np.sum(p)
                    i = np.random.choice(len(p), p=p)
                    selection.append(i)
                return np.array(selection)
```

```
In [3]: def test_policy_jacobian(policy, params=None):
            if params is None:
                params = policy.params_point()
            sample = policy.sample(params)
            jac = np.zeros_like(params)
            dh = 1e-6
            for i in range(len(params)):
                params[i] += dh
                logP_1 = policy.log_prob(sample, params)
                params[i] -= 2*dh
                logP_2 = policy.log_prob(sample, params)
                params[i] += dh
                jac[i] = (logP_1 - logP_2)/(2*dh)
            assert(np.max(np.abs(jac - policy.jac_log_prob(sample, params))) < 1e-3)
```

```
In [4]: policy = SimplePolicyFamily([10, 10, 5, 5, 4, 3], 3)
        test_policy_jacobian(policy)
        policy = SimplePolicyFamily([10, 10, 5, 5, 4, 3], 3)
        test_policy_jacobian(policy, np.array([0.3, 0.3, 0.1, 0.1, 0.1, 0.1]))
        caps = 10*np.random.rand(6)
        params = np.random.rand(6)
        params /= np.sum(params)
        policy = SimplePolicyFamily(caps, 2)
        test_policy_jacobian(policy, params)
```

```
In [5]: print(policy.sample_constrained(policy.params_point(), 5))
```

```
[[2 5]
 [4 0]
 [1 2]
 [5 2]
 [5 1]]
```

```
In [6]: odd_constraint = lambda s: s[1] % 2 != 0
        print(policy.sample_constrained(policy.params_point(), 20, [odd_constraint]))
```

```
[[4 3]
 [2 3]
 [4 3]
 [2 3]
 [4 5]
 [5 3]
 [3 5]
 [5 3]
 [4 1]
 [4 5]
 [1 3]
 [4 5]
 [2 5]
 [2 5]
 [1 3]
 [4 5]
 [5 3]
 [3 1]
 [4 1]
 [0 3]]
```

```
In [7]: import warnings


        def optimal_params(policy_family, start=None,
                           constraints=None, step=1e-2, eps=1e-3, max_iter=10000, verbose=0):
            """Apply gradient descent to find the optimal policy"""
            if start is None:
                start = policy_family.params_point()
```

```
    def loss(params):
        return policy_family.loss(params, constraints)

    params_old = np.copy(start)
    loss_old, jac_old = loss(params_old)
    it = 0
    while True:
        params_new = policy_family.normalize_params(params_old - step*jac_old)
        loss_new, jac_new = loss(params_new)
        jac_norm = np.sqrt(np.sum(jac_old**2))
        if loss_new > loss_old or jac_norm < eps:
            # converged
            break
        else:
            loss_old, jac_old = loss_new, jac_new
            params_old = params_new
            if it > max_iter:
                warnings.warn('max iter')
                break
        it += 1
        if verbose:
            print('it={0:>5d} jac norm={1:.2e} loss={2:.2e}'.format(it, jac_norm, loss_old))
    if verbose:
        print('Converged to desired accuracy :)')
    return params_old
```

Now that we have a simple policy we can differentiate numerically the loss function to test the consistency between the loss function value and its jacobian

```
In [8]: def test_loss(policy, params):
            dh = 1e-6
            jac = np.empty_like(params)
            for i in range(len(params)):
                x = np.copy(params)
                x[i] += dh
                f1, j1 = policy.loss(x)
                x[i] -= 2*dh
                f2, j2 = policy.loss(x)
                jac[i] = (f1 - f2)/(2*dh)
            f0, j0 = policy.loss(params)
            assert(np.max(np.abs(jac - j0)) < 1e-3)
```

```
In [9]: policy = SimplePolicyFamily([10, 8, 6, 6], 2)
        test_loss(policy, policy.normalize_params(np.random.rand(4)))
```

And a small test for convergence to optimal parameters

```
In [10]: policy = SimplePolicyFamily([10, 10, 10, 8, 8, 6, 6], 3)
         optimal_params(policy, eps=1e-2, verbose=1)
```

```
it=    1 jac norm=3.03e-01 loss=1.99e-02
it=    2 jac norm=2.97e-01 loss=1.90e-02
it=    3 jac norm=2.91e-01 loss=1.82e-02
it=    4 jac norm=2.85e-01 loss=1.74e-02
it=    5 jac norm=2.79e-01 loss=1.66e-02
it=    6 jac norm=2.73e-01 loss=1.59e-02
it=    7 jac norm=2.68e-01 loss=1.52e-02
it=    8 jac norm=2.62e-01 loss=1.45e-02
it=    9 jac norm=2.57e-01 loss=1.38e-02
it=   10 jac norm=2.52e-01 loss=1.32e-02
it=   11 jac norm=2.47e-01 loss=1.26e-02
it=   12 jac norm=2.41e-01 loss=1.20e-02
it=   13 jac norm=2.36e-01 loss=1.15e-02
it=   14 jac norm=2.32e-01 loss=1.09e-02
it=   15 jac norm=2.27e-01 loss=1.04e-02
```

```
it=   16 jac norm=2.22e-01 loss=9.93e-03
it=   17 jac norm=2.17e-01 loss=9.46e-03
it=   18 jac norm=2.13e-01 loss=9.01e-03
it=   19 jac norm=2.08e-01 loss=8.58e-03
it=   20 jac norm=2.04e-01 loss=8.17e-03
it=   21 jac norm=1.99e-01 loss=7.78e-03
it=   22 jac norm=1.95e-01 loss=7.40e-03
it=   23 jac norm=1.91e-01 loss=7.04e-03
it=   24 jac norm=1.86e-01 loss=6.69e-03
it=   25 jac norm=1.82e-01 loss=6.36e-03
it=   26 jac norm=1.78e-01 loss=6.05e-03
it=   27 jac norm=1.74e-01 loss=5.75e-03
it=   28 jac norm=1.70e-01 loss=5.46e-03
it=   29 jac norm=1.66e-01 loss=5.19e-03
it=   30 jac norm=1.62e-01 loss=4.93e-03
it=   31 jac norm=1.58e-01 loss=4.68e-03
it=   32 jac norm=1.55e-01 loss=4.44e-03
it=   33 jac norm=1.51e-01 loss=4.21e-03
it=   34 jac norm=1.47e-01 loss=3.99e-03
it=   35 jac norm=1.44e-01 loss=3.79e-03
it=   36 jac norm=1.40e-01 loss=3.59e-03
it=   37 jac norm=1.37e-01 loss=3.40e-03
it=   38 jac norm=1.34e-01 loss=3.23e-03
it=   39 jac norm=1.30e-01 loss=3.06e-03
it=   40 jac norm=1.27e-01 loss=2.90e-03
it=   41 jac norm=1.24e-01 loss=2.74e-03
it=   42 jac norm=1.21e-01 loss=2.60e-03
it=   43 jac norm=1.18e-01 loss=2.46e-03
it=   44 jac norm=1.15e-01 loss=2.33e-03
it=   45 jac norm=1.12e-01 loss=2.21e-03
it=   46 jac norm=1.09e-01 loss=2.09e-03
it=   47 jac norm=1.06e-01 loss=1.98e-03
it=   48 jac norm=1.03e-01 loss=1.87e-03
it=   49 jac norm=1.00e-01 loss=1.77e-03
it=   50 jac norm=9.76e-02 loss=1.68e-03
it=   51 jac norm=9.50e-02 loss=1.59e-03
it=   52 jac norm=9.25e-02 loss=1.50e-03
it=   53 jac norm=9.00e-02 loss=1.42e-03
it=   54 jac norm=8.76e-02 loss=1.34e-03
it=   55 jac norm=8.52e-02 loss=1.27e-03
it=   56 jac norm=8.28e-02 loss=1.20e-03
it=   57 jac norm=8.06e-02 loss=1.14e-03
it=   58 jac norm=7.84e-02 loss=1.08e-03
it=   59 jac norm=7.62e-02 loss=1.02e-03
it=   60 jac norm=7.41e-02 loss=9.62e-04
it=   61 jac norm=7.20e-02 loss=9.10e-04
it=   62 jac norm=7.00e-02 loss=8.61e-04
it=   63 jac norm=6.81e-02 loss=8.15e-04
it=   64 jac norm=6.62e-02 loss=7.71e-04
it=   65 jac norm=6.44e-02 loss=7.29e-04
it=   66 jac norm=6.26e-02 loss=6.90e-04
it=   67 jac norm=6.08e-02 loss=6.53e-04
it=   68 jac norm=5.91e-02 loss=6.18e-04
it=   69 jac norm=5.74e-02 loss=5.85e-04
it=   70 jac norm=5.58e-02 loss=5.54e-04
it=   71 jac norm=5.43e-02 loss=5.24e-04
it=   72 jac norm=5.27e-02 loss=4.96e-04
it=   73 jac norm=5.13e-02 loss=4.70e-04
it=   74 jac norm=4.98e-02 loss=4.45e-04
it=   75 jac norm=4.84e-02 loss=4.22e-04
it=   76 jac norm=4.71e-02 loss=4.00e-04
it=   77 jac norm=4.57e-02 loss=3.79e-04
it=   78 jac norm=4.45e-02 loss=3.59e-04
it=   79 jac norm=4.32e-02 loss=3.40e-04
it=   80 jac norm=4.20e-02 loss=3.22e-04
it=   81 jac norm=4.08e-02 loss=3.06e-04
it=   82 jac norm=3.97e-02 loss=2.90e-04
it=   83 jac norm=3.86e-02 loss=2.75e-04
```

```
it=   84 jac norm=3.75e-02 loss=2.61e-04
it=   85 jac norm=3.65e-02 loss=2.47e-04
it=   86 jac norm=3.55e-02 loss=2.35e-04
it=   87 jac norm=3.45e-02 loss=2.23e-04
it=   88 jac norm=3.36e-02 loss=2.12e-04
it=   89 jac norm=3.26e-02 loss=2.01e-04
it=   90 jac norm=3.17e-02 loss=1.91e-04
it=   91 jac norm=3.09e-02 loss=1.81e-04
it=   92 jac norm=3.00e-02 loss=1.72e-04
it=   93 jac norm=2.92e-02 loss=1.64e-04
it=   94 jac norm=2.84e-02 loss=1.55e-04
it=   95 jac norm=2.77e-02 loss=1.48e-04
it=   96 jac norm=2.69e-02 loss=1.40e-04
it=   97 jac norm=2.62e-02 loss=1.34e-04
it=   98 jac norm=2.55e-02 loss=1.27e-04
it=   99 jac norm=2.48e-02 loss=1.21e-04
it=  100 jac norm=2.42e-02 loss=1.15e-04
it=  101 jac norm=2.36e-02 loss=1.09e-04
it=  102 jac norm=2.29e-02 loss=1.04e-04
it=  103 jac norm=2.23e-02 loss=9.91e-05
it=  104 jac norm=2.18e-02 loss=9.43e-05
it=  105 jac norm=2.12e-02 loss=8.98e-05
it=  106 jac norm=2.06e-02 loss=8.55e-05
it=  107 jac norm=2.01e-02 loss=8.14e-05
it=  108 jac norm=1.96e-02 loss=7.76e-05
it=  109 jac norm=1.91e-02 loss=7.39e-05
it=  110 jac norm=1.86e-02 loss=7.04e-05
it=  111 jac norm=1.81e-02 loss=6.71e-05
it=  112 jac norm=1.77e-02 loss=6.40e-05
it=  113 jac norm=1.72e-02 loss=6.10e-05
it=  114 jac norm=1.68e-02 loss=5.82e-05
it=  115 jac norm=1.64e-02 loss=5.55e-05
it=  116 jac norm=1.60e-02 loss=5.29e-05
it=  117 jac norm=1.56e-02 loss=5.05e-05
it=  118 jac norm=1.52e-02 loss=4.81e-05
it=  119 jac norm=1.48e-02 loss=4.59e-05
it=  120 jac norm=1.45e-02 loss=4.38e-05
it=  121 jac norm=1.41e-02 loss=4.18e-05
it=  122 jac norm=1.38e-02 loss=3.99e-05
it=  123 jac norm=1.34e-02 loss=3.81e-05
it=  124 jac norm=1.31e-02 loss=3.64e-05
it=  125 jac norm=1.28e-02 loss=3.47e-05
it=  126 jac norm=1.25e-02 loss=3.32e-05
it=  127 jac norm=1.22e-02 loss=3.17e-05
it=  128 jac norm=1.19e-02 loss=3.02e-05
it=  129 jac norm=1.16e-02 loss=2.89e-05
it=  130 jac norm=1.14e-02 loss=2.76e-05
it=  131 jac norm=1.11e-02 loss=2.64e-05
it=  132 jac norm=1.08e-02 loss=2.52e-05
it=  133 jac norm=1.06e-02 loss=2.41e-05
it=  134 jac norm=1.03e-02 loss=2.30e-05
it=  135 jac norm=1.01e-02 loss=2.20e-05
Converged to desired accuracy :)
```

```
Out[10]: array([ 0.19228084,  0.19228084,  0.19228084,  0.1334386 ,  0.1334386 ,
                 0.07814014,  0.07814014])
```

```
In [11]: def run_sim(N, selector, n=100000):
             results = np.zeros((n, N), dtype=int)
             for it in range(n):
                 selected = selector()
                 for r, i in enumerate(selected):
                     results[it, i] = r + 1
             return results


         def report(sim, expected, actual):
```

```
    print('Expected vs actual use ({0} samples)'.format(sim.shape[0]))
    for i in range(len(expected)):
        print(' disk {0}: {1:.2e} {2:.2e}'.format(i, expected[i], actual[i]))
```

The following simulation tests the cases for R=2 and R=3. Keep in mind that running time grows as N^R, that's the reason we don't test for higher R. A possible solution could be using Monte Carlo methods.

Also keep in mind that the simulation with R=3 and constraints cannot satisfy the expected probabilities for the first replica due to the constraint that the biggest hard drive cannot be used for the first replica.

```
In [12]: cap = np.array([10, 8, 6, 10, 8, 6, 10, 8, 6])

         def constraint_1(s):
             """Never store the first replica on the biggest ones"""
             return cap[s[0]] != 10


         def constraint_2(s):
             """Suppose that there are three groups:

                       Hard drives
             Group 1: 1, 2, 3
             Group 2: 4, 5, 6
             Group 3: 7, 8, 9

             Don't store two replicas in the same group.
             """
             group = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2])
             count = np.array([0, 0, 0])
             for i in s:
                 count[group[i]] += 1
             return np.all(count <= 1)


         for R in [2, 3]:
             for constraints in [None, (constraint_1, constraint_2)]:
                 print('Simulation: R={0}, constraints: {1}'.format(R, constraints is not None))
                 print(72*'-')
                 pol = SimplePolicyFamily(cap, R)
                 opt = optimal_params(pol, constraints=constraints)
                 sim = run_sim(len(cap), pol.build_selector(opt, constraints=constraints))
                 print('First replica on each hard drive')
                 report(sim, np.repeat(1/len(cap), len(cap)), np.mean(sim == 1, axis=0))
                 print('All replicas on each hard drive')
                 report(sim, cap/np.sum(cap), 1/R*np.mean(sim > 0, axis=0))
                 print('Sample:')
                 print(sim[:10, :])
                 print()
```

```
Simulation: R=2, constraints: False
------------------------------------------------------------------------
First replica on each hard drive
Expected vs actual use (100000 samples)
 disk 0: 1.11e-01 1.12e-01
 disk 1: 1.11e-01 1.09e-01
 disk 2: 1.11e-01 1.11e-01
 disk 3: 1.11e-01 1.10e-01
 disk 4: 1.11e-01 1.11e-01
 disk 5: 1.11e-01 1.10e-01
 disk 6: 1.11e-01 1.12e-01
 disk 7: 1.11e-01 1.13e-01
 disk 8: 1.11e-01 1.11e-01
All replicas on each hard drive
Expected vs actual use (100000 samples)
```

```
 disk 0: 1.39e-01 1.39e-01
 disk 1: 1.11e-01 1.10e-01
 disk 2: 8.33e-02 8.37e-02
 disk 3: 1.39e-01 1.38e-01
 disk 4: 1.11e-01 1.11e-01
 disk 5: 8.33e-02 8.33e-02
 disk 6: 1.39e-01 1.40e-01
 disk 7: 1.11e-01 1.12e-01
 disk 8: 8.33e-02 8.37e-02
Sample:
[[0 0 1 2 0 0 0 0 0]
 [0 0 0 1 2 0 0 0 0]
 [0 0 0 0 0 1 2 0 0]
 [2 0 0 0 0 0 0 1 0]
 [2 0 0 0 0 0 1 0 0]
 [0 0 0 2 0 0 0 1 0]
 [0 0 0 0 0 1 2 0 0]
 [2 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 2 1]
 [2 0 1 0 0 0 0 0 0]]

Simulation: R=2, constraints: True
------------------------------------------------------------------------
First replica on each hard drive
Expected vs actual use (100000 samples)
 disk 0: 1.11e-01 0.00e+00
 disk 1: 1.11e-01 1.72e-01
 disk 2: 1.11e-01 1.62e-01
 disk 3: 1.11e-01 0.00e+00
 disk 4: 1.11e-01 1.70e-01
 disk 5: 1.11e-01 1.62e-01
 disk 6: 1.11e-01 0.00e+00
 disk 7: 1.11e-01 1.73e-01
 disk 8: 1.11e-01 1.61e-01
All replicas on each hard drive
Expected vs actual use (100000 samples)
 disk 0: 1.39e-01 1.39e-01
 disk 1: 1.11e-01 1.12e-01
 disk 2: 8.33e-02 8.31e-02
 disk 3: 1.39e-01 1.38e-01
 disk 4: 1.11e-01 1.11e-01
 disk 5: 8.33e-02 8.34e-02
 disk 6: 1.39e-01 1.38e-01
 disk 7: 1.11e-01 1.13e-01
 disk 8: 8.33e-02 8.24e-02
Sample:
[[0 0 1 0 0 0 0 2 0]
 [0 0 0 0 1 0 2 0 0]
 [0 0 1 0 0 0 2 0 0]
 [0 1 0 2 0 0 0 0 0]
 [0 0 0 0 1 0 0 2 0]
 [0 0 1 0 2 0 0 0 0]
 [0 1 0 0 0 0 2 0 0]
 [2 0 0 0 1 0 0 0 0]
 [2 0 0 0 0 1 0 0 0]
 [0 2 0 0 0 1 0 0 0]]

Simulation: R=3, constraints: False
------------------------------------------------------------------------
First replica on each hard drive
Expected vs actual use (100000 samples)
 disk 0: 1.11e-01 1.11e-01
 disk 1: 1.11e-01 1.12e-01
 disk 2: 1.11e-01 1.13e-01
 disk 3: 1.11e-01 1.09e-01
 disk 4: 1.11e-01 1.09e-01
 disk 5: 1.11e-01 1.11e-01
 disk 6: 1.11e-01 1.12e-01
 disk 7: 1.11e-01 1.11e-01
 disk 8: 1.11e-01 1.11e-01
All replicas on each hard drive
Expected vs actual use (100000 samples)
 disk 0: 1.39e-01 1.39e-01
 disk 1: 1.11e-01 1.12e-01
 disk 2: 8.33e-02 8.33e-02
 disk 3: 1.39e-01 1.39e-01
 disk 4: 1.11e-01 1.10e-01
 disk 5: 8.33e-02 8.32e-02
 disk 6: 1.39e-01 1.39e-01
 disk 7: 1.11e-01 1.11e-01
 disk 8: 8.33e-02 8.36e-02
Sample:
[[0 2 0 0 0 1 3 0 0]
 [2 0 1 0 0 0 0 3 0]
 [2 3 0 1 0 0 0 0 0]
 [0 0 0 2 0 3 1 0 0]
 [0 2 0 1 0 0 3 0 0]
 [0 3 0 1 0 0 0 2 0]
 [0 0 1 3 2 0 0 0 0]
 [0 0 1 0 0 2 0 0 3]
 [0 0 0 1 3 0 0 2 0]
 [1 0 0 3 0 0 2 0 0]]

Simulation: R=3, constraints: True
-----------------------------------------------------------------------
First replica on each hard drive
Expected vs actual use (100000 samples)
 disk 0: 1.11e-01 0.00e+00
 disk 1: 1.11e-01 1.76e-01
 disk 2: 1.11e-01 1.60e-01
 disk 3: 1.11e-01 0.00e+00
 disk 4: 1.11e-01 1.73e-01
 disk 5: 1.11e-01 1.59e-01
 disk 6: 1.11e-01 0.00e+00
 disk 7: 1.11e-01 1.73e-01
 disk 8: 1.11e-01 1.59e-01
All replicas on each hard drive
Expected vs actual use (100000 samples)
 disk 0: 1.39e-01 1.38e-01
 disk 1: 1.11e-01 1.11e-01
 disk 2: 8.33e-02 8.38e-02
 disk 3: 1.39e-01 1.39e-01
 disk 4: 1.11e-01 1.11e-01
 disk 5: 8.33e-02 8.32e-02
 disk 6: 1.39e-01 1.39e-01
 disk 7: 1.11e-01 1.11e-01
 disk 8: 8.33e-02 8.32e-02
Sample:
[[0 0 1 3 0 0 2 0 0]
 [2 0 0 0 0 1 3 0 0]
 [2 0 0 0 1 0 3 0 0]
 [2 0 0 0 3 0 0 0 1]
 [0 1 0 0 0 3 0 0 2]
 [3 0 0 2 0 0 0 0 1]
 [0 0 1 3 0 0 0 0 2]
 [0 1 0 2 0 0 3 0 0]
 [0 2 0 0 0 1 3 0 0]
 [2 0 0 0 0 1 0 3 0]]
```

```
In [ ]:
```

```
In [ ]:
```