The following is an extract of my graduate thesis. It has been (hastily) translated from the original in spanish and it's provided here as motivation and context for the Pattern Search Optimization module.

Thesis: “Multi-trait Genomic Selection with Regularized Models”

Student: Matías Florián Schrauf-García

Director: Sebastián Munilla

Defended in March 2016, for the grade of *Ingeniero Agrónomo* at the University of Buenos Aires.

RESULTS

1. GwFlasso implementation

[...]

With regard to cross validation, hyper-parameters were selected and evaluated by pattern search optimization (figure 3 and Appendix A). This accelerated the implementation in a variable factor between 30x and 50x, relative to exploring the whole grid of possible values.

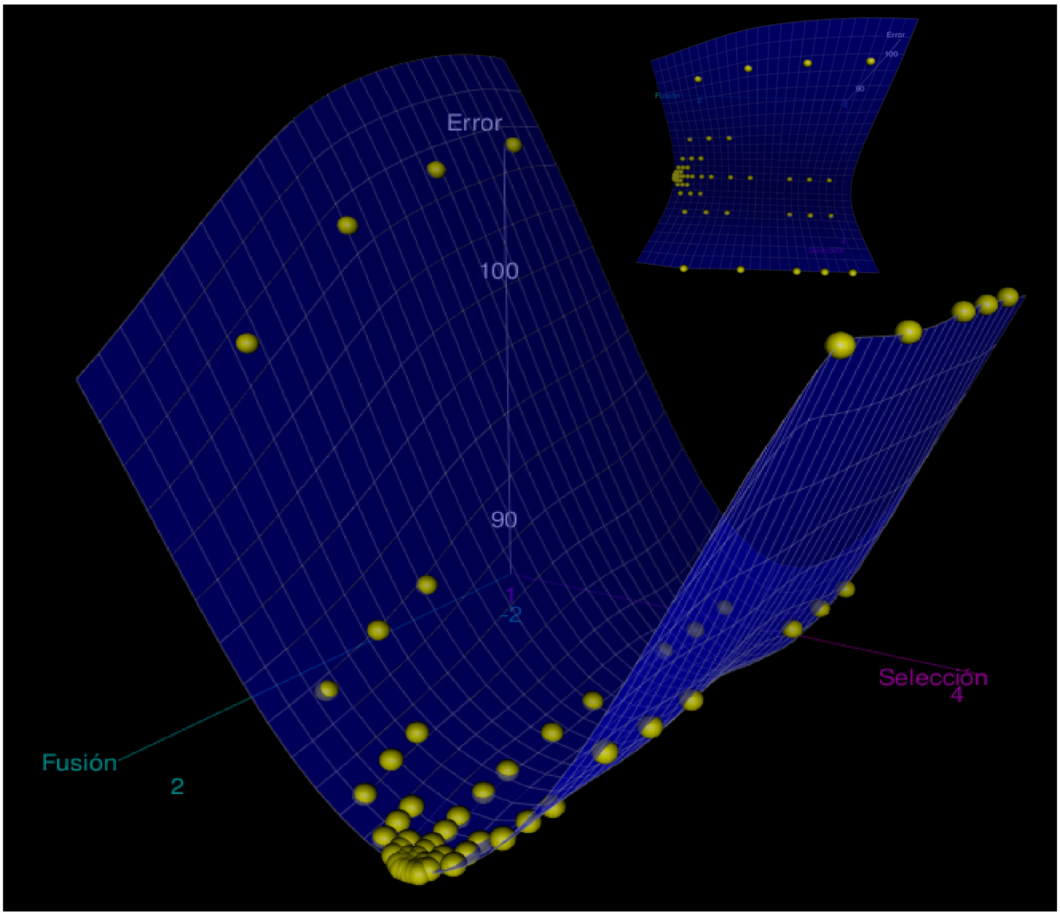


Figure 3. Generalization error surface estimated by cross-validation and minimized with pattern search optimization (inset: top view).

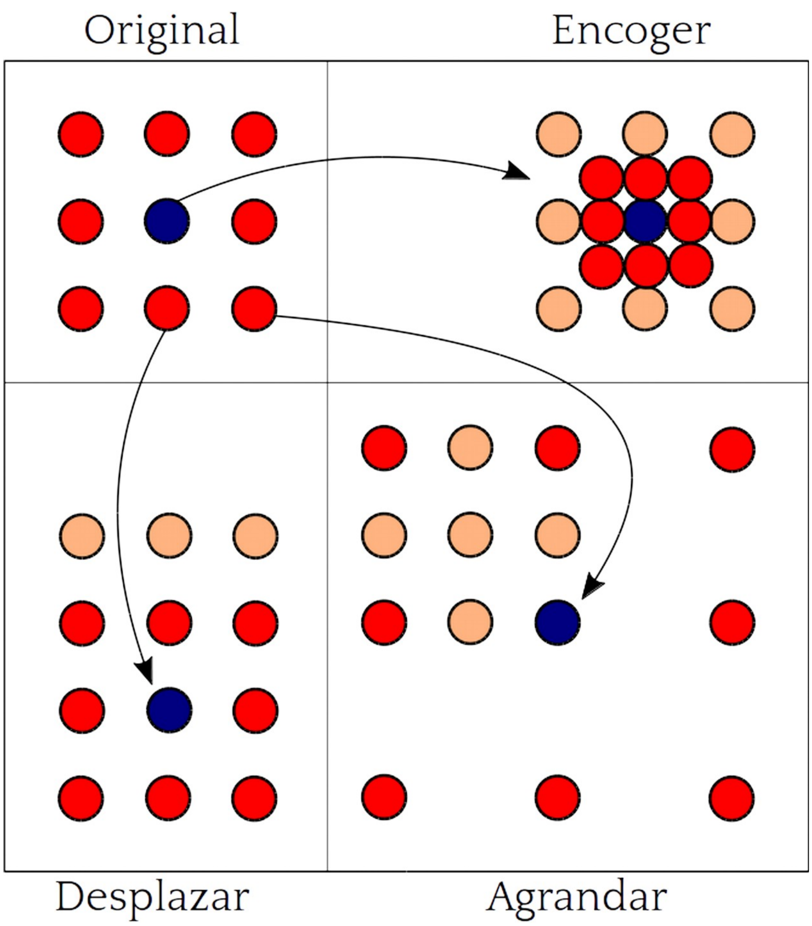
[...]

APPENDIX

A. Pattern search optimization of hyper-parameters

Choosing the hyper-parameters of a regularized model with cross-validation incurs in a high computational cost. Commonly, a grid is drawn over the space of candidate values for the hyper-parameters and cross-validation is conducted for each point of such grid. Nevertheless, for a gird of given density, the number of explored points grows exponentially with the number of hyper-parameters being considered. The pattern search optimization was chosen as an heuristic to efficiently explore a dense grid without evaluating a cross-validation on each of it's points[1] (Hooke and Jeeves, 1961).

In this implementation, the pattern search works by evaluating the cross-validation in a set of 9 points (values of the hyper-parameters), which form a square pattern of 3x3. Once the function to be minimized has been evaluated for each of the 9 points, the pattern is updated to a new configuration (see "*Pattern search (optimization)*", Wikipedia Collaborators, 2015). Depending on which of the 9 original points was evaluated at a smaller value, the update step is different (figure 11). Through the successive application of these updating rules, it is expected to find a value acceptably close to the minimum (Torczon, 1997).



1 Pattern search optimization was chosen because it does not require the derivative of the objective function (being what is sometimes called a 'derivative-free' or 'zeroth order' method).

Figure 11. Sketch of the pattern used and it's updating steps.

To implement this optimization method, a small module was written in the Python language (box 1). An interesting aspect of the implementation is the incorporation of a cache to memorize the previously evaluated values of the function to be minimized. This is called 'memoization' and is very important for the efficiency of the method, given the superposition between successive patterns.

Box 1. Code of the module *pattern_search.py* in the Python language.

```
"""Minimize a function over a 2D grid
with a square pattern search"""
import pandas as pd
import numpy as np
from collections import namedtuple

grid_pt = namedtuple('grid_pt','i j')

class PatternError(Exception):
    """Some problem with the pattern"""
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)


empty_cache = pd.DataFrame(columns = ['f_val'],
        index = pd.MultiIndex.from_arrays(
            [[],[]],
            names = ['i','j']
        )
    )


class Pattern:
    def __init__(self, center, step, cache=None):
        self.center = grid_pt(*center)
        self.step = grid_pt(*step)
        self.a = np.vstack([[-1,0,1]]*3)
        self.b = self.a.T
        self.i = self.center.i + self.a * self.step.i
        self.j = self.center.j + self.b * self.step.j
        self.df = pd.DataFrame({
                'i': self.i.ravel(),
                'j': self.j.ravel(),
                'f_val': np.nan
            }, index = pd.MultiIndex.from_arrays(
                [self.a.ravel(), self.b.ravel()],
                names = ['a','b']
            ))
        if cache is None:
            self.cache = empty_cache.copy()
        else:
            self.cache = cache

    def __repr__(self):
        return "Pattern("+str(self.center)+","+str(self.step)+")"
```

```python

    def fill(self,f):
        for row in self.df.itertuples():
            # retrieve from cache or evaluate
            try:
                newf = self.cache.loc[(row.i,row.j),'f_val']
            except KeyError:
                newf = f(row.i,row.j)
                self.cache.loc[(row.i,row.j),'f_val'] = newf
            # set
            self.df.loc[row.Index,'f_val'] = newf

    def update(self):
        first = self.df.f_val.argmin()
        if pd.isnull(first):
            raise PatternError(self)
        if first == (0,0):
            # shrink
            newcenter = self.center
            newstep = map(lambda x: x//2, self.step)
        elif first in [(-1,-1),(-1,1),(1,-1),(1,1)]:
            # grow
            newstep = map(lambda x: x*2, self.step)
            newcenter = self.df.loc[first,['i','j']].astype(int)
        else:
            # move
            newcenter = self.df.loc[first,['i','j']].astype(int)
            newstep = self.step
        return Pattern(newcenter,newstep,self.cache)


# Example: minimize Rosenbrock “banana” function
if __name__ == "main":
    path = []
    n = 2**8
    step = (2**5,2**5)
    center = (0,0)

    def rosenbrock_func_gen(n=10, a=1, b=100):
        xs = np.linspace(-0.5, 2, n)
        ys = np.linspace(-1.0, 3, n)

        def f(i,j):
            if not (0 <= i < n and 0 <= j < n):
                return np.nan
            x = xs[i]; y = ys[j]
            return (a-x)**2+b*(y-x**2)**2

        return f

    f = ps.rosenbrock_func_gen(n)
    p = ps.Pattern(center, step, ps.empty_cache.copy())

    maxiter = 50
    minstep = (0,0)
    maxcache = 200
    niter = 0
    while all([
        niter < maxiter,
        p.step[0] > minstep[0],
```

```
                p.step[1] > minstep[1],
                p.cache.dropna().size < maxcache
                ]):
                print(p)
                path.append(p.center)
                p.fill(f)
                # evolve! 81)
                p = p.update()
                niter += 1
```

BIBLIOGRAPHY

- Hooke, R., y Jeeves, T. A. 1961. *“Direct Search” Solution of Numerical and Statistical Problems*. Journal of the ACM (JACM), 8(2), 212-229.

- Colaboradores de Wikipedia. 2015. *Pattern search (optimization)*. Wikipedia, The Free Encyclopedia. [fecha de consulta: 18 de febrero del 2016].

- Torczon, V. 1997. *On the convergence of pattern search algorithms*. SIAM Journal on optimization, 7(1), 1-25.