Optimized_Query_Design

December 17, 2023

1 Optimized Query Design for Models with Normally Distributed Error

There's a pretty common problem that shows up in any kind of data science - getting data is usually pretty labor and cost intensive. What this means is that if you're going to go out and collect data to fit a model, you want to collect *the best* data you possibly can. Now naively one might think that just taking a random sample is always the best path forward. But that's not always the case.

To illustrate, let's start with the simplest possible example. Suppose we are trying to fit a model of the form:

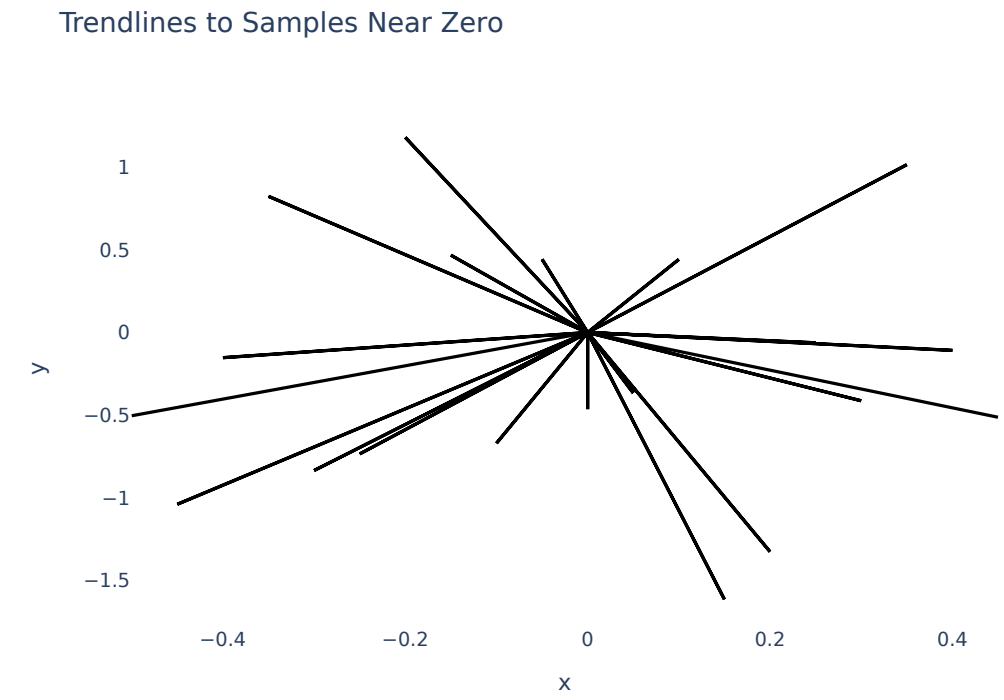
$$y_i = mx_i + e_i$$

where e_i is an error term that is normally distributed with variance σ^2. In other words - $e_i \sim \mathcal{N}(0, \sigma^2)$.

The question is now this - if we have a limited number of samples to grab, which x_i should we prioritize? Well let's start by looking at the trendlines of samples near the origin.

[ ]:
```
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.io as pio
pio.renderers.default = "notebook+pdf"

np.random.seed(42)

m = 2
sigma = 1
x = np.arange(-0.5, 0.5, 0.05)
y = m * x + np.random.normal(0, sigma, len(x))
df = pd.DataFrame({'x': x, 'y': y})
df['trace'] = df['x']
origins = df.copy()
origins['x'] = 0
origins['y'] = 0
df = pd.concat([df, origins])
df = df.sort_values(by=['trace', 'x'])
```

```
fig = px.line(df, x='x', y='y', title="Trendlines to Samples Near Zero")
fig.update_traces(line_color='black', marker=dict(color='black'))
fig.update_layout(plot_bgcolor="rgba(0,0,0,0)")
```

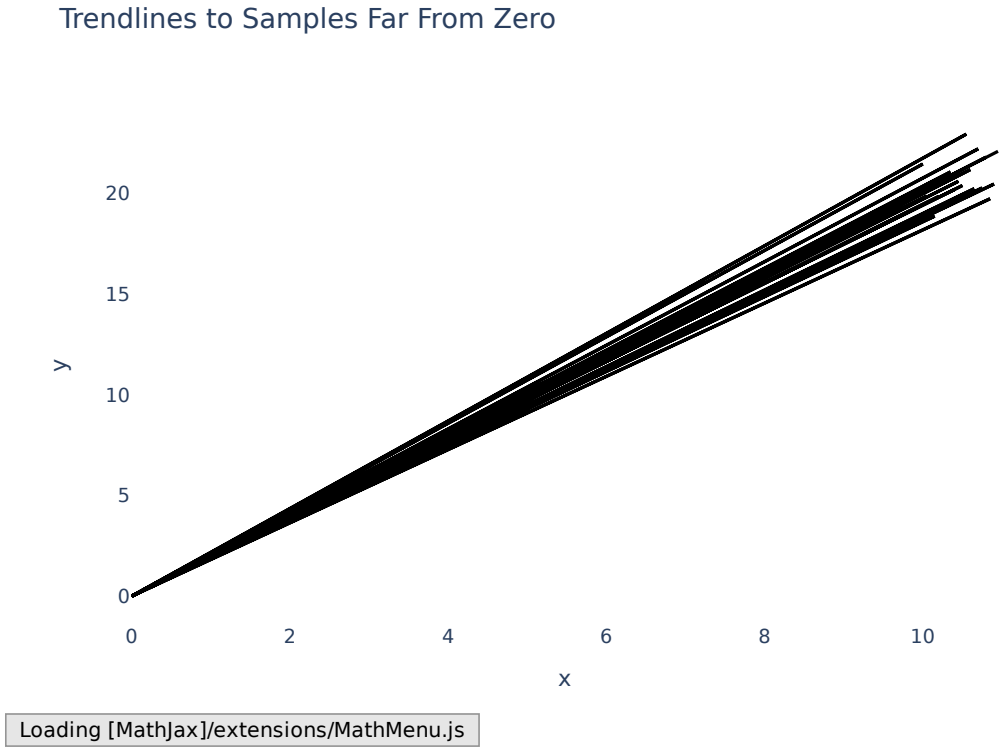


Loading [MathJax]/extensions/MathMenu.js

These go in all kinds of directions! In fact from any one of these it's pretty much impossible to see our specific $m = 2$.

So what happens if instead we sample far away from the origin?

[ ]:
```
x = np.arange(10, 11, 0.05)
y = m * x + np.random.normal(0, sigma, len(x))
df = pd.DataFrame({'x': x, 'y': y})
df['trace'] = df['x']
origins = df.copy()
origins['x'] = 0
origins['y'] = 0
df = pd.concat([df, origins])
df = df.sort_values(by=['trace', 'x'])
fig = px.line(df, x='x', y='y', title="Trendlines to Samples Far From Zero")
fig.update_traces(line_color='black', marker=dict(color='black'))
fig.update_layout(plot_bgcolor="rgba(0,0,0,0)")
```



Now our m is really clear!

So what's going on here? Well, when we sample near the origin our noise e_i overwhelms our signal, but when we sample far away from the origin the response of our model mx_i becomes much larger than our noise e_i and so it becomes far easier to pick out m. Because of a higher "signal to noise" ratio, the samples where $|x_i|$ is large are more informative.

Alright so this was a pretty simple case. How can we generalize this for more complex models? The process of doing so starts with the math behind Maximum Likelihood Estimation (MLE).

1.1 Maximum Likelihood Estimation

For the time being, let's continue working with our super simple model:

$$y_i = mx_i + \epsilon_i \sim \mathcal{N}(mx_i, \sigma^2)$$

Now taking advantage of the fact that the probability density function for a normal distribution is well known, we have, for our model, that the probability density function is:

$$f(y_i; m) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{1}{2}(\frac{y_i - mx_i}{\sigma})^2}$$

This function gives us a sense of how "likely" each potential y_i is given m, x_i, and σ. Now during

fitting we will have some dataset composed of $\mathbf{x} = (x_1, x_2, ..., x_n)$ and $\mathbf{y} = (y_1, y_2, ..., y_n)$. So, we could say that the likelihood of this data, given m is just:

$$L(\mathbf{y}; m) = \prod_i f(y_i; m)$$

However given typically $f(\mathbf{y}; m) << 1$ and the product of many small numbers is a far smaller number, it is more useful to deal with the logarithm of this quantity:

$$l(\mathbf{y}; m) = \ln L(\mathbf{y}; m) = \sum_i \ln f(y_i; m)$$

Put this way we can see that the "best" m is simply whichever m is most likely, i.e. the one that maximizes $l(\mathbf{y}; m)$. From calculus you may remember that wherever a function $h(u)$ is maximized we have $\partial_u h = 0$ (you can think of this as equivalent to saying the top of a mountain is flat). Therefore we are really looking for:

$$\partial_m l(\mathbf{y}; m) = 0$$

So let's work out what $\partial_m l(\mathbf{y}; m)$ is in our case:

$$\ln f(y_i; m) = -\ln \sigma\sqrt{2\pi} - \frac{1}{2}(\frac{y_i - mx_i}{\sigma})^2$$

$$\partial_m \ln f(y_i; m) = -\frac{y_i - mx_i}{\sigma}(\frac{-x_i}{\sigma}) = \frac{x_i}{\sigma^2}(y_i - mx_i)$$

$$\partial_m l(\mathbf{y}; m) = \sum_i \frac{x_i}{\sigma^2}(y_i - mx_i)$$

Therefore what we want is for the individual components of our summand to be as close to zero as possible. Let's give those components a name. We'll call:

$$\chi_m = \partial_m \ln f(y_i; m) = \frac{x_i}{\sigma^2}(y_i - mx_i)$$

our penalty (because we want it as close to zero as possible).

Now given we know our "signal to noise" ratio increases with increasing $|x_i|$ shouldn't our penalty follow the same trend? Well let's assume that the real m is 2 but we guess an m of 2.5. What does our penalty as a function of x_i look like?

[ ]:
```
real_m = 2.0
m_guess = 2.5
sigma = 1
x = np.arange(0, 10, 0.05)
y = real_m * x
penalty = x / (sigma ** 2) * (y - m_guess * x)
```

```
px.line(x=x, y=penalty, title="Penalty for Guessing the Wrong Slope",␣
 ↪color_discrete_sequence=['black']).
 ↪update_layout(plot_bgcolor="rgba(0,0,0,0)")
```

Loading [MathJax]/extensions/MathMenu.js

And there you have it - the magnitude of the penalty does in fact get far far higher as the magnitude of x_i increases. Furthermore if x_i is near the origin, guessing the wrong m incurs a penalty close to zero (i.e. bad guesses are as good as good guesses).

Another way of putting this is that our penalty's sensitivity to changes in m is a function of x_i, but the mathematical representation of sensitivity to change in a variable is just the derivative with respect to that variable. Therefore we can capture the pattern we just discovered in the derivative of χ_m with respect to m:

$$\partial_m \chi = \partial^2 \ln f(y_i; m)$$

In our case this is just:

$$\partial_m \chi = -\frac{x_i^2}{\sigma^2}$$

And now we can see for sure that as the magnitude of x_i increases so does the signal to noise ratio of our sample.

It turns out the negative expectation of $\partial_m \chi$

$$I = -E[\partial^2 \ln f(y; m)]$$

has a name - it's called the Fisher Information and it has exactly the interpretation you'd expect. The larger the magnitude of the Fisher Information the easier it's going to be fit your parameter - i.e. the better the signal to noise ratio you're going to get.

Why the expectation? Well because for more complicated models $\partial^2 \ln f(y_i; m)$ is a function of y_i itself and so, in order to get the information we have to get the expected value across all possible y_i given some x_i and m.

1.2 The Fisher Information in General

Alright so we've got a more mathematically sound measure of the "informativeness" of a sample. But we've still only done this in the case of a single parameter. What happens when we have multiple parameters? Well consider the following model:

$$y_i = mx_i + b + e_i \sim \mathcal{N}(mx_i + b, \sigma^2)$$

In this case we have:

$$f(y_i; m, b) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{1}{2}(\frac{y_i - mx_i - b}{\sigma})^2}$$

And we actually have two scores now:

$$\chi_m = \partial_m \ln f(y_i; m, b) = \frac{x_i}{\sigma^2}(y_i - mx_i - b)$$

$$\chi_b = \partial_b \ln f(y_i; m, b) = \frac{1}{\sigma^2}(y_i - mx_i - b)$$

Given we have two parameters to vary this means that we also have two sensitivities to look at for each of our scores. In other words we are interested in four quantities:

$$\partial_m \chi_m = -\frac{x_i^2}{\sigma^2}$$

$$\partial_b \chi_m = -\frac{x_i}{\sigma^2}$$

$$\partial_b \chi_b = -\frac{1}{\sigma^2}$$

$$\partial_m \chi_b = -\frac{x_i}{\sigma^2}$$

Now as before, the Fisher Information is just the negative expectation of each of these. So, taking that and throwing this into a matrix to make our lives easier we have:

$$I = \begin{pmatrix} -E[\partial_m \chi_m] & -E[\partial_m \chi_b] \\ -E[\partial_b \chi_m] & -E[\partial_b \chi_b] \end{pmatrix} = \begin{pmatrix} x_i^2/\sigma^2 & x_i/\sigma^2 \\ x_i/\sigma^2 & 1/\sigma^2 \end{pmatrix}$$

So what are we to make of this? Before we had a nice simple number - now we have four… how are we to tell where our overall "information" (or signal to noise) is maximized?

Well one way of thinking about this is to imagine that each of the rows of our matrix is like a vector that tells us how informative our sample is at estimating each of our respective parameters. The first row tells us about estimating m whereas the second tells us about b. Obviously as each row gets larger in magnitude the informativeness with respect to the corresponding parameter gets larger as well. Enter now the determinant! It turns out one way of interpreting the determinant is that if you take the vectors represented by each of the rows, the absolute value of the determinant gives you the volume of the parallelpiped described by those vectors as edges. Therefore, intuitively we can use the determinant to summarize the informativeness of our sample.

Let's give it a try:

$$\det \begin{pmatrix} x_i^2/\sigma^2 & x_i/\sigma^2 \\ x_i/\sigma^2 & 1/\sigma^2 \end{pmatrix} = x_i^2/\sigma^2(1/\sigma^2) - x_i/\sigma^2(x_i/\sigma^2) = 0$$

Hold on… our determinant is *always* zero no matter which x_i we choose? How can that be? Well remember that we have two parameters now. You cannot fit two separate parameters with one single sample - the problem is degenerate. So actually, rather than this being disturbing, the fact that our determinant is zero here is actually pretty cool! It's telling us that the two vectors defining our "information" volume are actually parallel and therefore the area of our volume is zero. The determinant is not only telling us how informative our individual vectors are but also how well separated they are too! Very cool.

Okay so in order to get any informativeness we're actually going to need to sample twice. What does our information look like for multiple samples? Well assuming our samples are independent, we just get to sum the individual components.

$$I = \begin{pmatrix} -\sum_{x_i} E[\partial_m \chi_m] & -\sum_{x_i} E[\partial_m \chi_b] \\ -\sum_{x_i} E[\partial_b \chi_m] & -\sum_{x_i} E[\partial_b \chi_b] \end{pmatrix} = \begin{pmatrix} \sum_{x_i} x_i^2/\sigma^2 & \sum_{x_i} x_i/\sigma^2 \\ \sum_{x_i} x_i/\sigma^2 & \sum_{x_i} 1/\sigma^2 \end{pmatrix}$$

So, for example, if we had $x_1 = 2$ and $x_2 = 0$ this gives us:

$$I = \begin{pmatrix} 4/\sigma^2 & 2/\sigma^2 \\ 2/\sigma^2 & 2/\sigma^2 \end{pmatrix}$$

from which we get:

$$\det \begin{pmatrix} 4/\sigma^2 & 2/\sigma^2 \\ 2/\sigma^2 & 2/\sigma^2 \end{pmatrix} = 4/\sigma^2$$

With two samples we indeed end up with non-zero information.

Alright, that was a lot. What did we get out of it? Well we've got something to maximize now. Specifically maximizing the determinant of the information matrix:

$$|\det I|$$

should allow us to maximize the informativeness of the sample we are taking. So let's now move beyond our linear model and look at the general case for normally distributed error.

1.3 The General Case for Normally Distributed Error

Now that we've got ourselves some intuition for why we're doing things this way, let's do the general case where:

$$y_i = g(x_i, \ ) + e_i \sim \mathcal{N}(g(x_i, \ ), \sigma^2)$$

and $\ = (\theta_1, \theta_2, ...\theta_n)$ is our list of parameters.

This form works any time we have constant variance and a normal error distribution. So if we determine our information matrix for this form, we'll get a whole bunch of cases sorted at once.

So we now have:

$$f(y; \ ) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{1}{2}(\frac{y-g(x; \ )}{\sigma})^2}$$

Let's start with our penalties:

$$\ln f(y; \ ) = -\ln \sigma\sqrt{2\pi} - \frac{1}{2}(\frac{y - g(x; \ )}{\sigma})^2$$

$$\chi_{\theta_j} = \frac{y - g(x; \ )}{\sigma^2} \partial_{\theta_j} g(x; \ )$$

Then we can get the sensitivities of our penalities:

$$\partial_{\theta_i} \chi_{\theta_j} = \frac{y - g(x; \ )}{\sigma^2} \partial^2_{\theta_j} g(x; \ ) - \frac{1}{\sigma^2} \partial_{\theta_i} g(x; \ ) \partial_{\theta_j} g(x; \ )$$

And finally to turn this into the form the Fisher Information expects:

$$I_{i,j} = -E[\partial_{\theta_i} \chi_{\theta_j}] = \int \left(\frac{y - g(x; \ )}{\sigma^2} \partial^2_{\theta_j} g(x; \ ) - \frac{1}{\sigma^2} \partial_{\theta_i} g(x; \ ) \partial_{\theta_j} g(x; \ ) \right) f(y; \ ) dy$$

$$E[\partial_{\theta_i} \chi_{\theta_j}] = \frac{1}{\sigma^2} \partial^2_{\theta_j} g(x; \ ) \int y f(y; \ ) dy - \frac{g(x; \ )}{\sigma^2} \partial^2_{\theta_j} g(x; \ ) - \frac{1}{\sigma^2} \partial_{\theta_i} g(x; \ ) \partial_{\theta_j} g(x; \ )$$

$$E[\partial_{\theta_i}\chi_{\theta_j}] = \frac{E[y] - g(x;\ )}{\sigma^2}\partial^2_{\theta_j}g(x;\ ) - \frac{1}{\sigma^2}\partial_{\theta_i}g(x;\ )\partial_{\theta_j}g(x;\ )$$

But $E[y] = g(x;\ )$ and so we have:

$$I_{i,j}(x_i) = -E[\partial_{\theta_i}\chi_{\theta_j}] = \frac{1}{\sigma^2}\partial_{\theta_i}g(x_i;\ )\partial_{\theta_j}g(x_i;\ )$$

Which means that all we need to compute our information is to get the first derivatives of our model with respect to each of our parameters. Beyond that we don't even really need to know what σ is because it does not affect the maximization of our information. Pretty cool!

1.4 Practically Solving the General Case

We've got the math in hand now, but remember that ultimately we're trying to find the set of samples $\mathbf{x} = (x_1, x_2, ..., x_n)$ that maximizes $|\det I|$. Given most of the time the x_i are continuous vectors of more than one dimension - that's a pretty darn big space to explore! So what we're going to need is an optimization that can explore the space for us. One obvious candidate is a genetic algorithm.

A genetic algorithm is nice here because our problem lends itself really well to how genetic algorithms work. First, our individual is really clear - it's just our sample $\mathbf{x}$. In turn the genes of our individual are also pretty clear - each of the x_i would be its own gene. Then mutation would simple be shifting the x_i around and crossover would be exchanging x_i between samples. Furthermore our objective just requires the first derivatives of our model, and we can get those through automatic differentiation. So given how simple the formulation is, let's go ahead and build one real quick.

We'll take advantage of the `mygrad` and `deap` libraries from Python.

[ ]:
```
import mygrad as mg
from deap import base, creator, tools
```

First we'll go ahead and define the things our GA (genetic algorithm) is going to need. In this case we'll use the following two dimensional model:

$$y_i = mx_i + pu_i + b$$

[ ]:
```
def model(vector):
    m, p, b = params
    return m * vector[0] + p * vector[1] + b

bounds = [(0, 100), (0, 100)]
sample_size = 10
params = [mg.tensor(1.0), mg.tensor(2.0), mg.tensor(10.0)]
```

Next let's define our individual and the population:

[ ]:
```
import random
random.seed(42)
```

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)


def init_from_bounds(bounds):
    return np.array([random.random() * (upper - lower) + lower for lower, upper␣
 ↪in bounds])

toolbox = base.Toolbox()
toolbox.register("vector", init_from_bounds, bounds)
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.
 ↪vector, n=sample_size)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Moving onto mutation:

[ ]:
```
def move_vector(vector, rel_distance, bounds):
    direction = np.random.random(len(vector)) * 2 - 1
    direction = direction / np.linalg.norm(direction)
    ranges = np.array([upper - lower for lower, upper in bounds])
    distances = rel_distance * ranges
    direction = direction * distances
    new_vector = vector + direction
    for i in range(len(new_vector)):
        if new_vector[i] < bounds[i][0]:
            new_vector[i] = bounds[i][0]
        elif new_vector[i] > bounds[i][1]:
            new_vector[i] = bounds[i][1]
    return new_vector

def mutate(individual, prob, rel_distance, bounds):
    for i, vector in enumerate(individual):
        if random.random() < prob:
            individual[i] = move_vector(vector, rel_distance, bounds)
    return individual,

toolbox.register("mutate", mutate, prob=0.5, rel_distance=0.1, bounds=bounds)
```

Crossover and selection:

[ ]:
```
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("select", tools.selTournament, tournsize=3)
```

And finally evaluation (this is where we use automatic differentiation):

[ ]:
```
def evaluate(individual):
    covs = []
```

```
    for vector in individual:
        y = model(vector)
        y.backward()
        covs.append(
            np.array([
                [params[row].grad * params[col].grad for row in␣
↪range(len(params))]
                for col in range(len(params))
            ])
        )
    score = np.abs(np.linalg.det(np.sum(covs, axis=(0))))
    return score,

toolbox.register("evaluate", evaluate)
```

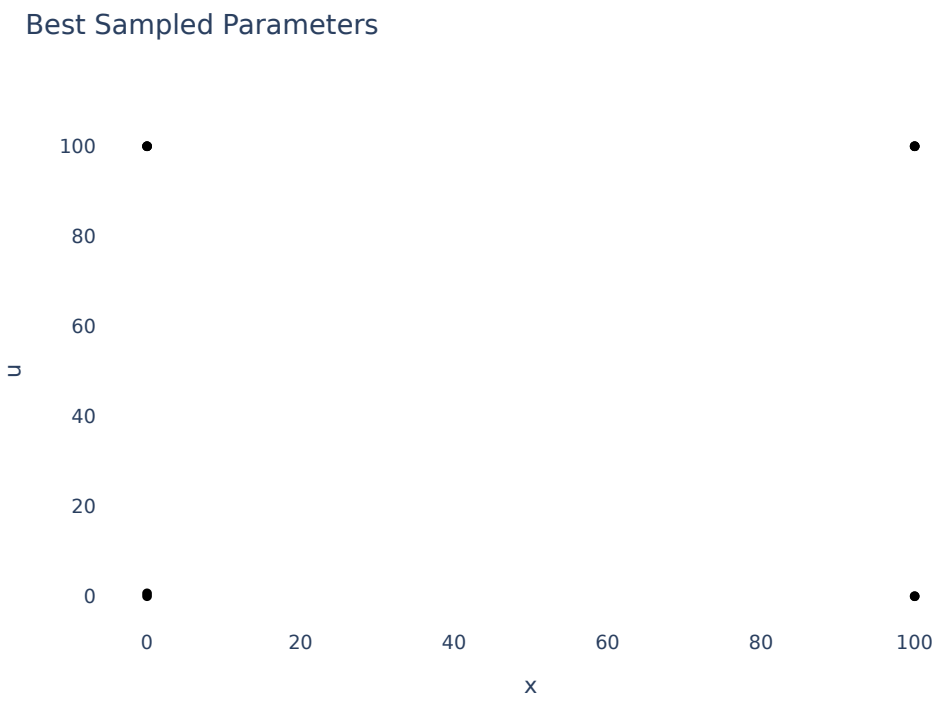
And with that all defined we can now shamelessly pull the most basic GA formulation straight from the `deap` documentation:

[ ]:
```
pop = toolbox.population(n=50)
CXPB, MUTPB, NGEN = 0.5, 0.2, 100

# Evaluate the entire population
fitnesses = map(toolbox.evaluate, pop)
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit

for g in range(NGEN):
    # Select the next generation individuals
    offspring = toolbox.select(pop, len(pop))
    # Clone the selected individuals
    offspring = list(map(toolbox.clone, offspring))

    # Apply crossover and mutation on the offspring
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < CXPB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = map(toolbox.evaluate, invalid_ind)
```

```
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # The population is entirely replaced by the offspring
    pop[:] = offspring
```

Let's see what we got:

[ ]:
```
df = pd.DataFrame(tools.selBest(pop, k=1)[0], columns=['x', 'u'])
px.scatter(df, x='x', y='u', title="Best Sampled Parameters",␣
 ↪color_discrete_sequence=['black']).
 ↪update_layout(plot_bgcolor="rgba(0,0,0,0)")
```



And there you have it! An orthogonal experiment design just as you would expect from more classical methods of solving this problem. However unlike more classical methods, if you wanted to solve a new problem all you'd need to do is update:

```
def model(vector):
    m, p, b = params
    return m * vector[0] + p * vector[1] + b

bounds = [(0, 100), (0, 100)]
sample_size = 10
```

```
params = [mg.tensor(1.0), mg.tensor(2.0), mg.tensor(10.0)]
```

to reflect your specific model. Then you'd just run the GA again and get a new result. Pretty nifty!

2 Some Useful Sources

MLE and Fisher Information

Determinant for Maximization

Determinant as Volume

Parametrized Density Functions

Automatic Differentiation