In silico Experiments

Understanding In Silico Experiments

*Notes by Frank D'Agostino (2021)

Preamble

> All models are wrong, but some are useful.

George E. P. Box

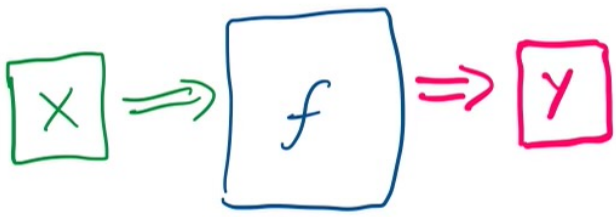
Objectives

1. Learn about positive controls for models
2. Actually simulate data
3. Install kallisto and run kallisto commands

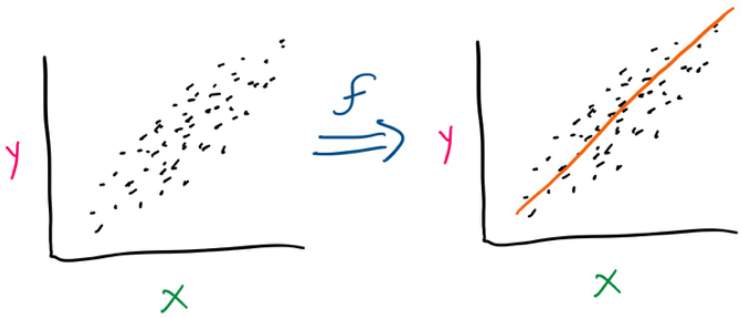
1. Positive Controls for Models

Often with computation work, there are many models that promise a wide array of functionalities and results. However, often times, it is dangerous to assume models work as intended or work for all types of inputs. It is a valuable skill to find the fault points of these models. We can do this by artificially generating data where we "know" the answer.

We begin with a toy example to illustrate this point. Let's consider a black box model that predicts money made from sales based on money spent on advertisting.



For example, we have the unknown model *f*, which takes in money spent on advertisting as the input to predict money made from sales. Since we may not be able to tell what *f* does in every scenario, we can artificially generate data. If we generate data based on a Gaussian (Normal) distribution:



It looks good! What about if we generated data some other way?

Wow! The model is working really well! If *f* is a model that should predict y (money made from sales) from x (money spent on advertisting), then if we generate data we know follows an exponential relationship, the model should also pick up on this, right?

Hmm... It looks like the function *f* performed poorly and did not fit the data well. We know the right answer that the data follows an exponential relationship, and so a good model would be a fitted exponential curve. Clearly, this is not what occurs.

Based on us generating the data, we can see that the model *f* has **limitations**. If this model *f* was in a published paper, for example, then an **assumption** of this model would have to be that x (money spent on advertisting) has a *linear* relationship with y (money made from sales). Anything other than linear a relationship the mode would not be able to accurately predict our target variable.

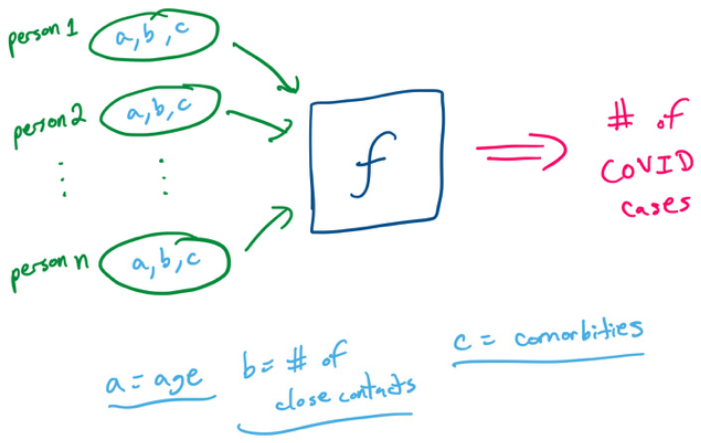
While this was a simple example, this idea of generating data artificially to experiment with model outputs can provide insight into the inner workings of the model or reveal further areas of work.

2. Simulating Data

Let's think of a more complex example. Say we find a paper where the authors claim they have a model that predicts the counts of COVID cases when given the following data points for each person in the population:

- age
- number of close contacts
- comorbidities

With the following structure:



Now, our goal is to make sure the model works for simple cases where we know how the data is generated. How do we go about doing this? Firstly, we want to generate data with the answer in mind. Let's say we want to artificially simulate 100 people, where each person has a 20 percent chance of being positive for COVID. We can begin with a simple for loop, using the np.random.rand() function, which generates a random number between 0 and 1:

```
# Generate 100 people and randomly assign them pos
for i in range(100):
    p = np.random.rand()      # Sample a single ran
    if p > 0.8:
        # Generate COVID negative person
    else:
        # Generate COVID positive person
```

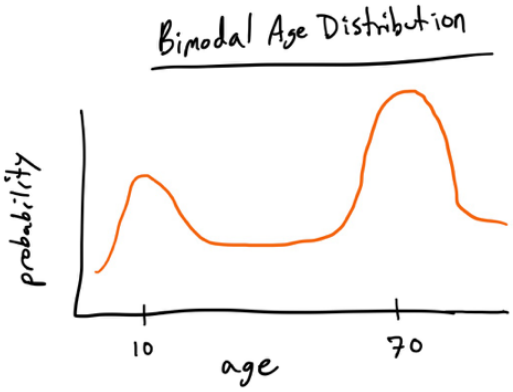
Now, what does it mean to "generate" a COVID positive or negative person? Well, we can begin by thinking how to do this in a smart way. Since we have to make 100 people, and possibly more if we want to do different experiments, we should make a **function**. We can generate a COVID negative person by thinking about what would a person less likely to get/test positive for COVID have? Research has shown that older people are more at risk of COVID than younger people, so maybe we can generate COVID negative people who are are younger. To simplify things, we can assume a normal distribution for this. Further, we know that COVID negative people likely have close contacts, due to less chances of coming into contact with other positive people. We can randomly generate this number, since human behavior is hard to predict and its not unreasonable to think there's a wide array of different living situations from people living alone to living with extended family. Finally, it seems that comorbidities increase the risk of contracting COVID, and so we can assign comorbidities to a positive or negative person with a higher or lower probability, respectively. Our function can look something like this:

```
# Generate COVID negative person
def gen_covid_neg_person():

    # Generate random age on a normal distribution,
    # The first parameter is the mean, and the secc
    age = np.random.normal(30, 20)

    # Uniformly randomly sample a number of close c
    # np.random.rand() generates a random number be
    # a number between 0 and 8
    close_contacts = int(np.random.rand()*8)

    # Assign a comorbidity to person, where COVID n
    # np.random.choice randomly selects things from
    # If p is left blank, it chooses the items unif
    comorb = np.random.choice(['heart disease', 'ob

    # Return these variables which describe our per
    return [age, close_contacts, comorb]
```

Seems fine! However, something is a bit restricting. We hard coded some parameters, such as the average age of a negative COVID person, and the probabilities of comorbidities. The power of functions is that we can abstract these hard coded values away so they can be easily changed. Rather, we can code the function as follows:

```
# Generate COVID negative person
def gen_covid_neg_person(age_mean, age_var, max_cc

    # Generate random age on a normal distribution,
    age = np.random.normal(age_mean, age_var)

    # Uniformly randomly sample a number of close c
    close_contacts = int(np.random.rand()*max_cc)

    # Assign a comorbidity to person, where COVID n
    comorb = np.random.choice(['heart disease', 'ob

    # Return these variables which describe our per
    return [age, close_contacts, comorb]
```

Now, we easily tweak the values to run our experiments. We can do something similar to generate a COVID positive person based on certain differences. Maybe we increase the probability of comorbidities, and increase the maximum number of close contacts. Additionally, for age, maybe we can generate age based on a biomodal distribution like this:

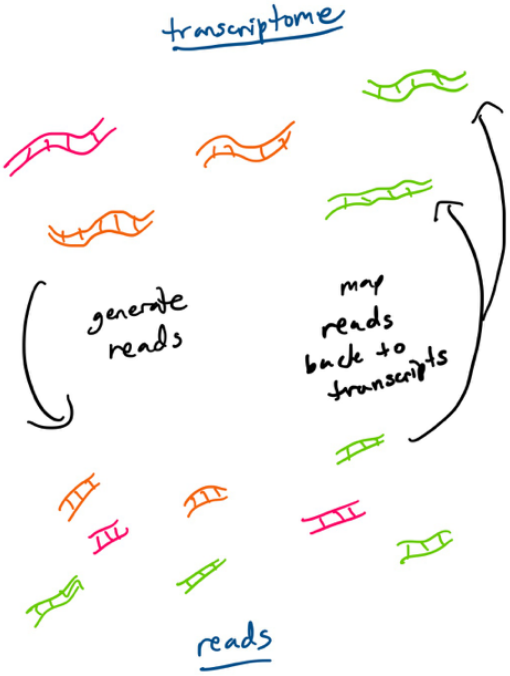


The key is that how we generate our population can give us insight into where the model fails. For example, if we generated the COVID positive people and the COVID negative people the exact same way, we'd expect the model to perform as if it was random. We might be able to tweak the parameters and make observations. Maybe this specific model uses age as its main predictor, and so changing the distribution of the ages effects the model drastically.

The main idea here is that you can perform computational **experiments**. You can generate hypotheses about why the model would or would not work and test these hypotheses by simulating data in a systematic way.

kallisto Commands

As learned in lecture, kallisto is one of these models. As you remember, we start out with the transcriptome, which becomes many fragments that we must map back to distinct transcripts.



We want to know which transcript, and subsequently which gene, it came from so we can get the counts, and thus the TPM for each gene. The model kallisto uses is structured roughly as follows:

So for the problem set, think about 1) how to

simulate the data and 2) what type of "experiments" to run on kallisto. Don't forget about the circular nature of the loci in the problem set! (Think about if this makes sense in terms of kallisto).

To do this process, you must be able to run kallisto. Firstly, you want to install it. You can run:

```
conda install kallisto
```

Once installed, you can run command line commands using the bang symbol:

```
!kallisto
```

If nothing happens, then it is not properly installed. You can also check:

```
!kallisto cite
```

Once installed, you must first generate the de Bruijn graph using:

```
!kallisto index -i transcripts.idx trascripts.fasta.g
```

Then, you actually want to map the reads using the graph:

```
!kallisto quant -i transcripts.idx -o reads_1.fastq.g
```

And finally, as a quick reminder:

FASTA files:

```
> transcript name
sequence
```

FASTQ files:

```
@ read name
sequence
+
quality
```

Hopefully this gave you everything you needed to tackle the problem set! Best of luck!