The Intuition Behind the Apriori Algorithm

Analyzing shopping trends is a pretty big deal in data science, and market-basket analysis is one way in which that's done. Techniques in this sub-field seek to understand how buying certain items influences the purchase of other items. This allows retailers to increase revenue by up-selling or cross-selling their existing customers.

Understanding the Apriori Algorithm is foundational to building your understanding of many techniques for market-basket analysis. It's used to find groups of items that occur together frequently in a shopping dataset. This is usually the first step to finding new ways in which to promote merchandise.

So... What does shopping data look like?

Imagine we have a file where each line represents a customer's shopping cart at checkout time. Let's call this a *basket*.

```
eggs flour butter milk sugar bananas apples tomatoes
potatoes spinach carrots cheese beer crackers wine
```

Each line above represents a basket

Each basket is made up of items. Our objective is to find sets of items that occur frequently together in the dataset. We'll call these *frequent itemsets*.

> Our objective is to find sets of items that occur frequently together in the dataset.

We'll set our own definition of what *frequent* means. This definition will likely change based on the number of baskets in the dataset. Typically, we'll be interested in frequent itemsets of a particular size. Today, let's assume that we're looking for frequent triples (i.e. itemsets of size 3).

As we continue exploring below, let's use this example dataset to enhance the discussion. It takes on the format described above.

Some quick stats about the example dataset:

- It contains 100,000 baskets
- There are 5,000 items that are sold at this particular supermarket
- Each item is a fake ISBN number (you know, like for books📚)

Let's also say that appearing more than 1,000 times makes an itemset *frequent*. (**Note:** The count of an itemset is often called its *support.*)

The Naïve Approach

The first thing that comes to mind is to scan through the dataset, count up the occurrences of all the triples, then filter out the ones that aren't frequent.

The naïve approach is appealing for its simplicity. However, we end up counting a lot of triples.

In the example dataset, there are over 7 million total triples, but only 168 of them are frequent.

> There are over 7 million total triples, but only 168 of them are frequent.

```
beep boop... calculating stuff...

There are 7104320 total triples, 168 of which are frequent
```

Lots of triples, but not a lot of frequent triples.

This is a problem for two reasons:

- Keeping track of the counts for those millions of triples takes up a lot of space.
- Building and counting all of those triples takes a lot of time.

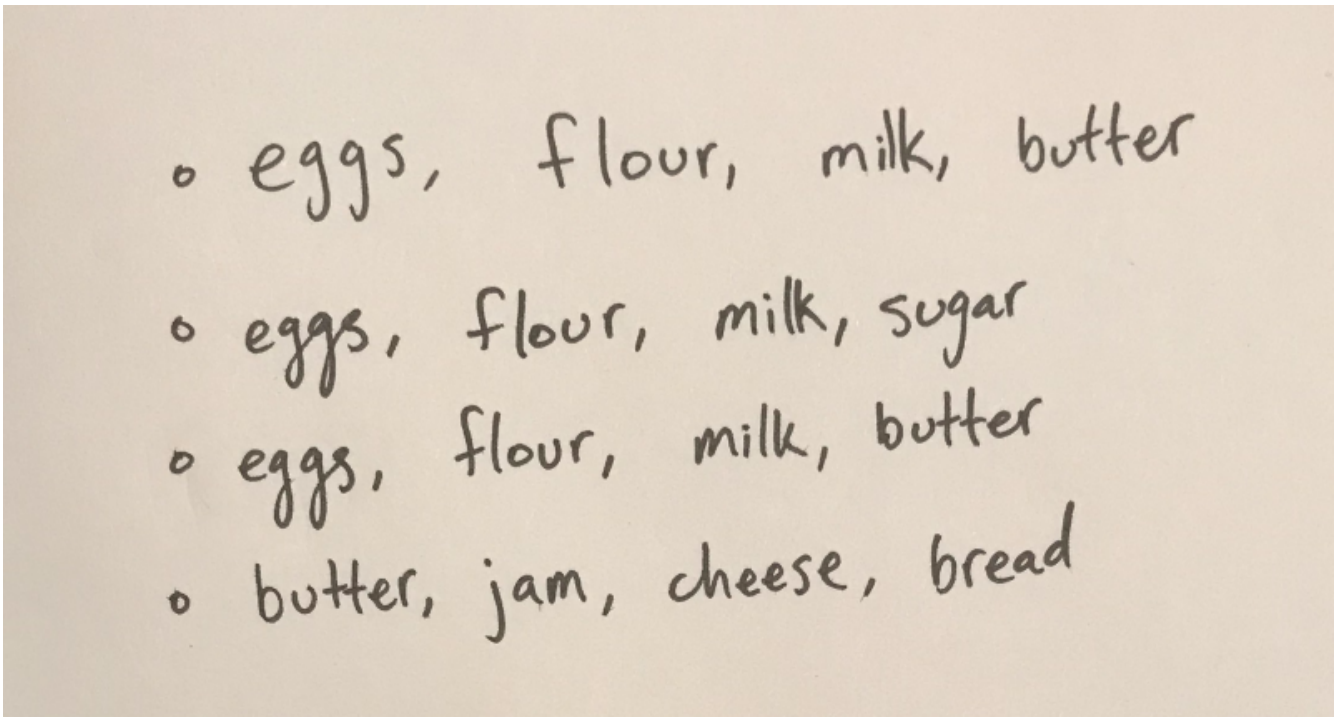
If only there were a way to avoid building and counting so many triples...

A Key Intuition

There *is* a way to avoid doing that! And it relies on a key piece of information. Don't worry, we'll break it down together, but here it is:

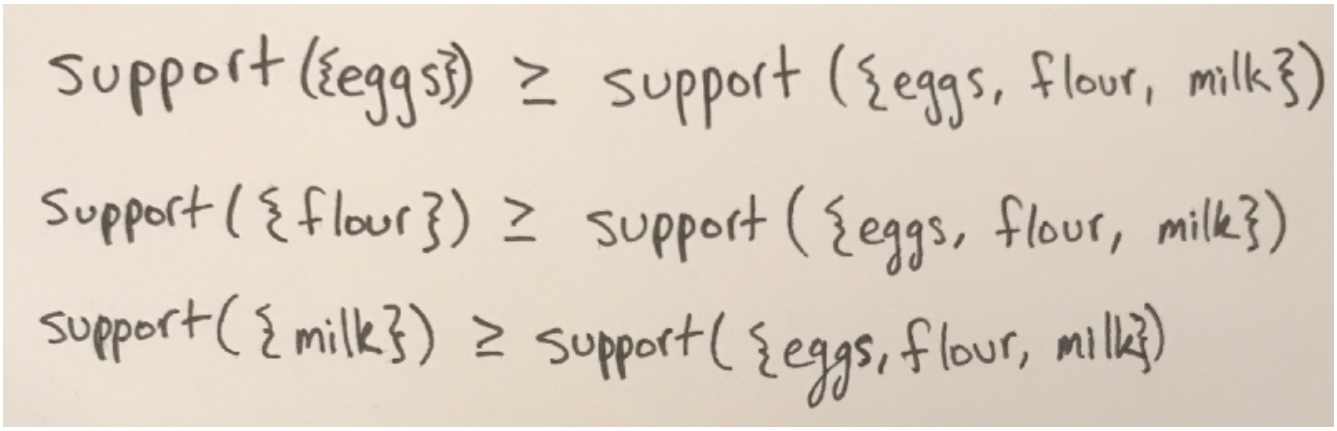
The key intuition is that a*ll subsets of frequent itemsets are also frequent itemsets*.

To add some clarity, consider these four baskets:



Eggs, flour, and milk seem to be a pretty popular combo

We can clearly see that the set {eggs, flour, milk} occurs 3 times. And at risk of sounding overly simplistic: each time we see the group {eggs, flour, milk}, we are seeing the individual items {eggs}, {flour}, and {milk}. As a result, we can safely say that the support (i.e. count) of {eggs} will be at least as large as the support of the set {eggs, flour, milk}. The same applies to flour and milk.



In fact, we can extend this fact to pairs of items inside of the set {eggs, flour, milk}. That means Support({eggs, flour}) ≥ Support({eggs, flour, milk}). Again, that's because the set {eggs, flour} necessarily occurs whenever the set {eggs, flour, milk} occurs, since it's a subset of it. Of course, this generalizes to sets and subsets of larger sizes.

Okay... So what?

The fact outlined above becomes useful when you think about it in the other direction. Since sugar, for example, only occurs once, we know that any set that contains sugar can only occur once.

Think about it: if a triple that contained sugar occurred more than once, that would mean that sugar occurs more than once. And since sugar only occurs once, we can guarantee that any triple that contains sugar will not appear more than once.

That means that we can simply ignore all of the sets that contain sugar, since we know that they can't be frequent itemsets.

We can apply this same logic to larger subsets as well. In the image above, {milk} and {butter} each appear three times. But as a pair, they only appear together twice (i.e. Support({milk, butter}) = 2). This means that any triple that contains the pair {milk, butter} can appear at most 2 times. When scanning for triples that appear 3 or more times, we could simply ignore {milk, butter, flour} since it contains {milk, butter}, so we know that it *must*appear no more than 2 times.

The Apriori Algorithm makes use of this fact to eliminate needlessly constructing and counting itemsets.

The Apriori Algorithm

Unlike the naïve approach, which makes a single pass over the dataset, the Apriori Algorithm makes several passes—increasing the size of itemsets that are being counted each time. It filters out irrelevant itemsets by using the knowledge gained in previous passes.

Here's an outline of the algorithm:

First Pass

- Get the counts of all the items in the dataset

- Filter out items that are not frequent

Second Pass

- Get the counts of pairs of items in the dataset. BUT: only consider pairs where both items in the pair are frequent items. (These are called *candidate pairs.*)
- Filter out pairs that are not frequent.

Third Pass

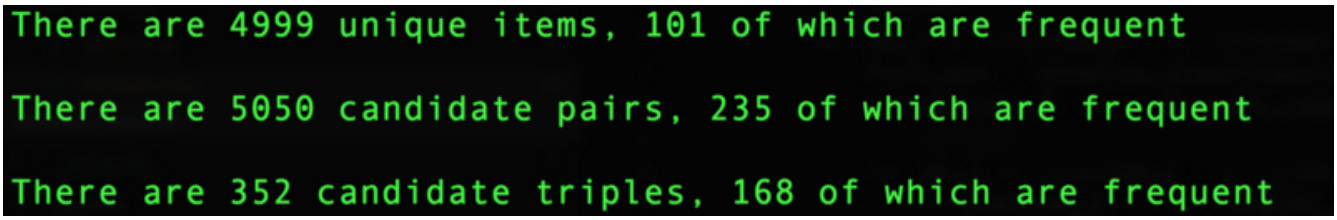
- Get the counts of triples in the dataset. BUT: only consider *candidate triples.* If any of the items in the triple are not frequent, the triple is not a candidate triple. If any of the *pairs of items* in the triple are not frequent, the triple is not a candidate triple.
- Filter out triples that are not frequent.

Nth Pass

- Get the counts of candidate itemsets of size N. (**Remember:** If any of the subsets of items in the itemset are not frequent, then the itemset cannot be frequent.)
- Filter out itemsets that are not frequent.

While this algorithm ends up taking more passes over the data, it saves a lot of time by cutting down on the number of itemsets that it builds and counts.

Take a look at these stats from our example dataset:



We only consider 352 of the 7m+ possible triples.

Ultimately, this reduction in the number of itemsets considered is what makes the Apriori Algorithm so much better than the naïve approach.

Want to truly understand it...?

I encourage you to implement the Apriori Algorithm yourself, as a way of cementing your understanding of it.

To get you started, I've set up a Github repo with some example datasets and other resources to get you started.

Here are two challenges—one much harder than the other.

The Challenge

Extract frequent triples from the example datasets by implementing the Apriori Algorithm and conducting three passes through the dataset.

Check you implementation by also implementing the naïve approach and comparing your results.

The Bonus Challenge

Implement the Apriori Algorithm such that it will extract frequent itemsets of any given size. For example, if I want to extract frequent itemsets of, say, size 13 it should be able to do that.

The Github repo includes a script that you can use to generate very large datasets. Use this to test your more general implementation.

(If you choose to take on the bonus challenge, I encourage you to do the 3 pass implementation first. I think it'll help build your understanding.)

Avoid this common mistake

One of the most common mistakes when implementing this algorithm happens when checking for candidate itemsets. Folks often remember to check if the individual items in an itemset are frequent, but they forget to check if *all subsets* of the current itemset are frequent.

> Folks often remember to check if the individual items in an itemset are frequent, but they forget to check if all subsets of the current itemset are frequent.

Remember: Triples contain pairs. Quads contain triples and pairs. And so on.

Let's chat!

If you take on either of the challenges above, please let me know. I'd love to swap implementations of the algorithm with you and hear about your experience.

I've listed my email on the Github repo, so please feel free to reach out.