How to build a Recommendation Engine quick and simple

Part 1: an introduction, how to get to production in a week and where to go after that

Jan Teichmann Apr 6, 2019 · 11 min read ★

Recreation of Rina Piccolo's cartoon (Cartoonist Group)

This article is meant to be a light introduction to the topic and provide the first steps to get you into production with a recommendation engine in a week. I will also tell you what the steps are to go from basics to fairly advanced.

We won't tab into the cutting edge yet but we won't be far off either.

Read part 2 for the more advanced use-cases when quick and simple is no longer good enough:

Advanced Use-Cases for Recommendation Engines

Part 2: How to build your next recommendation engine when quick and simple is no longer good...

towardsdatascience.com

what made Ama nending you which books to read. There are many other companies which are all build around recommendation systems: YouTube, Netflix, Spotify, Social Media platforms. In parallel, consumers came to expect a personalised experience and sophisticated recommendation systems to find relevant products and content, all to save consumers time and money.

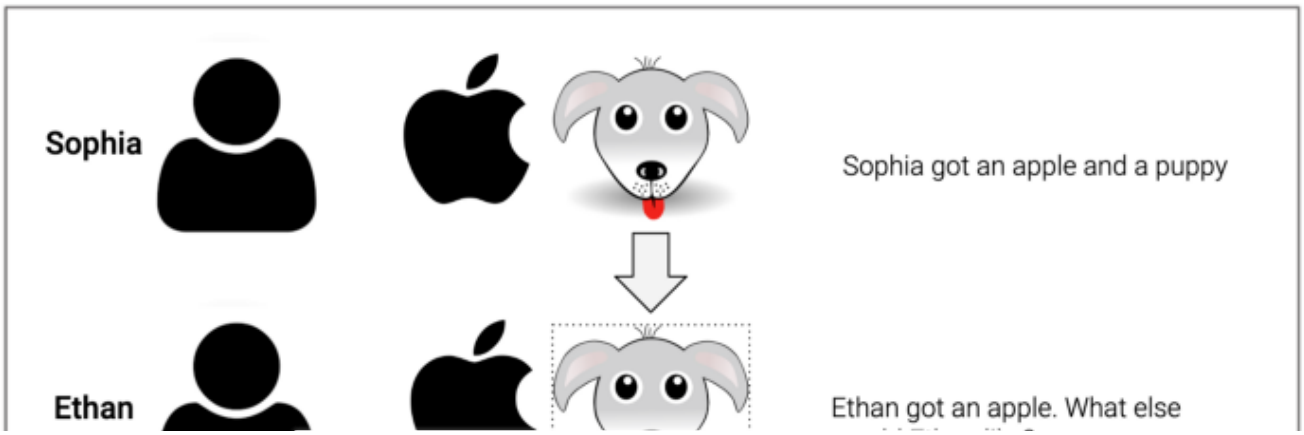
But how do you build a recommendation engine?

Background

Recommendation engines have been around for a while and there have been some key learnings to leverage:

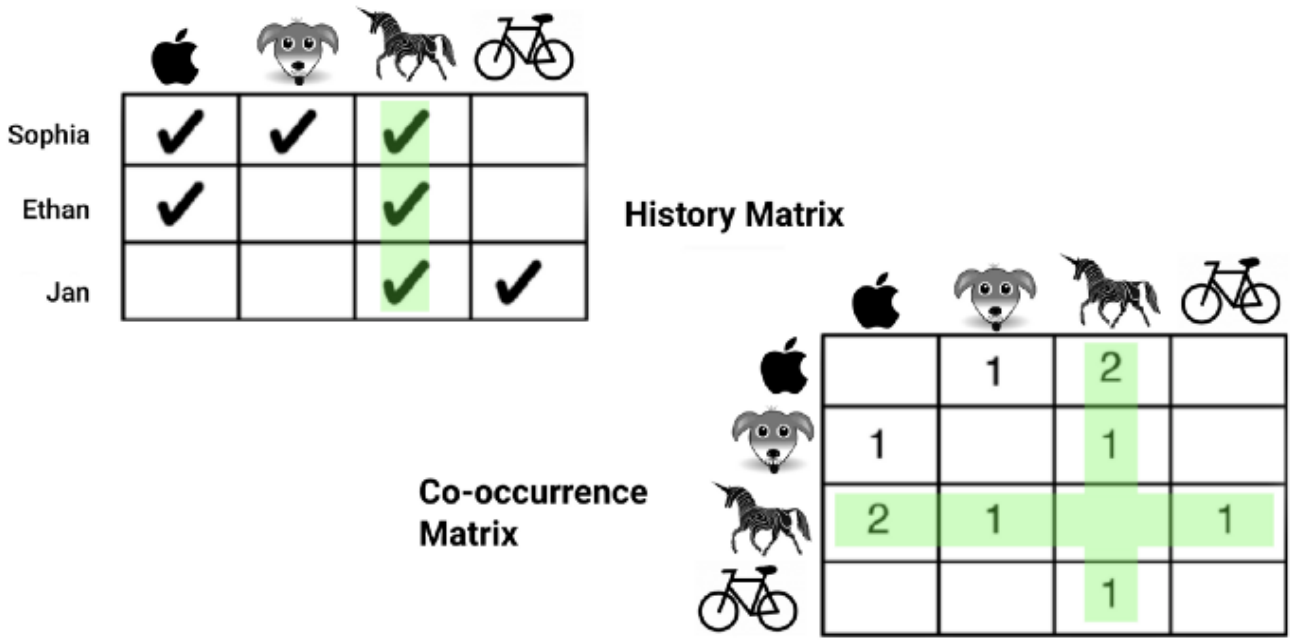
1. A user's actions are the best indicator of user intent. Ratings and feedback tends to be very biased and lower volumes.
2. Past actions and purchases drive new purchases and the overlap with other people's purchases and actions is a fantastic predictor.

Recommendation systems generally look for overlap or **co-occurrence** to make a recommendation. Like in the following example where we recommend Ethan a puppy based on a similarity of Ethan with Sophia:



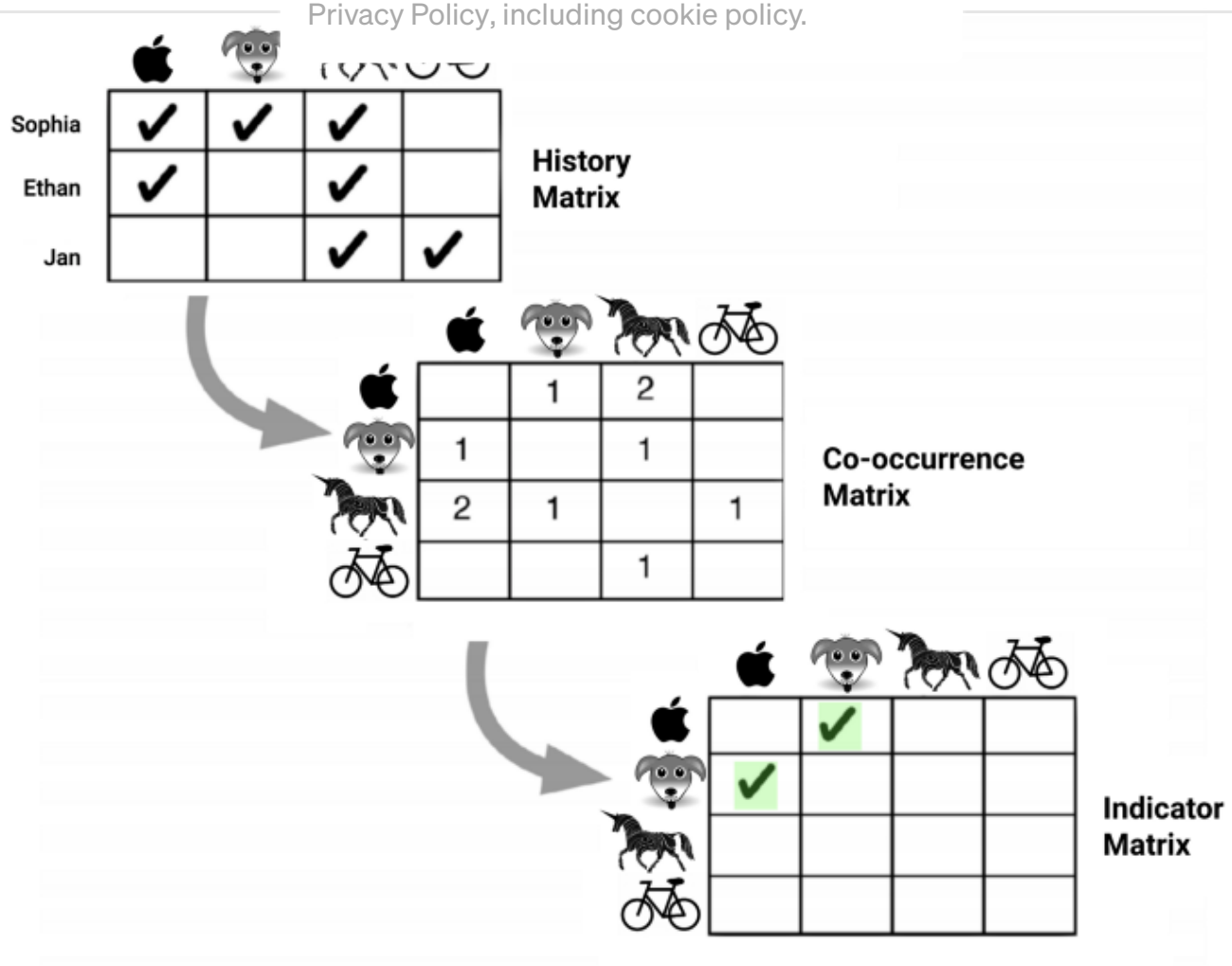
Recreation of illustration in “Practical Machine Learning, Ted Dunning & Ellen Friedman, O’Reilly 2014

In practise, a recommendation engine computes a **co-occurrence matrix** from a **history matrix** of events and actions. This is simple enough but there are challenges to overcome in real world scenarios. What if everyone wants a unicorn? Does the high co-occurrence of unicorns in the following example make a good recommendation?



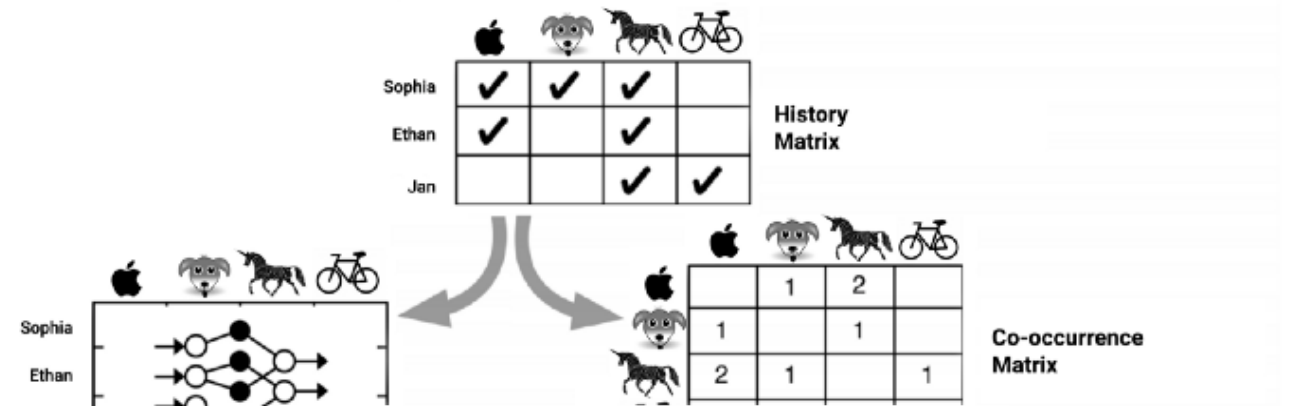
Recreation of illustration in “Practical Machine Learning, Ted Dunning & Ellen Friedman, O’Reilly 2014

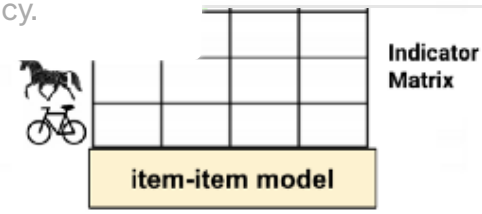
After the recommendation system has computed the co-occurrence matrix we have to apply statistics to filter out the sufficiently



Recreation of illustration in “Practical Machine Learning, Ted Dunning & Ellen Friedman, O’Reilly 2014

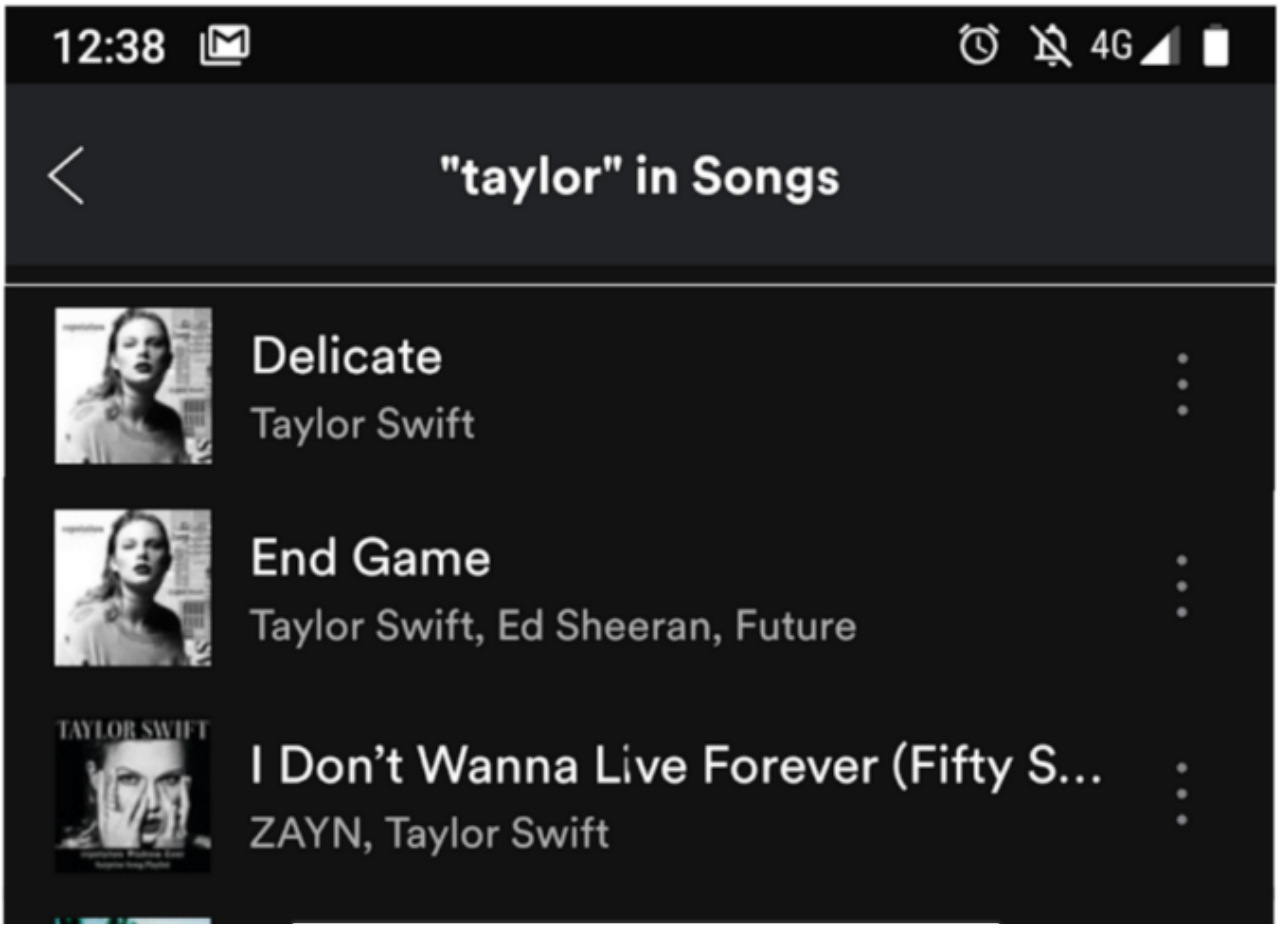
The algorithms and statistics which can extract **relevant indicators** from the co-occurrence matrix are what makes a good recommendation system. The path of creating an item-to-item indicator matrix is called an **item-item model**. There is obviously also an **user-item model:**

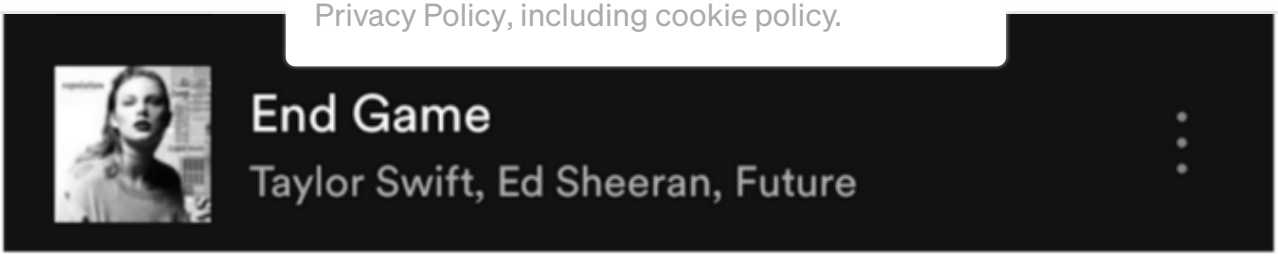




To create a user-item model we could apply a simple matrix factorisation or train a neural network to predict the scores of a user-item input. Usually, item-item models are more robust and produce better results when we do not invest more into feature engineering and model tuning etc.

However, there are a few more challenges a good recommendation system has to overcome. Recommending the same things over and over is **boring**. Even worse, recommending the same things produces **bad data** and causes **content fatigue**.





Spotify wouldn't produce such a result set! Their search and recommendation engines are top notch! This example needed editing. ❤Spotify

Two simple and intuitive strategies to improve the value of recommendations are

1. Anti-Flood: Penalise the second and third recommendations if they have the same similarity scores to the top recommendation.
2. Dithering: Add a wildcard recommendation to create interesting new data points for the recommendation system to keep learning about other content.

These steps ensure an interesting user experience and new data on alternative recommendations.

Data and Requirements

Recommendation systems work best with **unbiased explicit user feedback** like the purchase of products, watching a video or listening to

mainly implicit f h queries. Unfortunately, that data is also heavily biased e.g. the click through rate is heavily dependent on the position of content on a page. Implicit feedback also tends to perform less well, e.g. a click on a search result might just teaching you about how clickbaity a headline or CTA was rather than the relevance of the actual content. This results in high bounce rates after initially high click through rates (a very common case!)

A simple approach like collaborative filtering requires co-occurrence between items. This means collaborative filtering is suitable and will lead to great results if

- Your product catalogue is not too big, items are very long lived and can be interacted with easily by multiple users. Let's use Zoopla as an example where we get into trouble: Zoopla.co.uk is a property portal and has over 1.3+ million listings live at any point in time. A rental listing in London is very short lived and it can just take days before a property has been rented and is taken off the market. Obviously, you cannot rent out or sell a flat to multiple people at the same time! With the size of the Zoopla catalogue it is really difficult to generate a significant amount of co-occurrence even at Zoopla's traffic volumes.

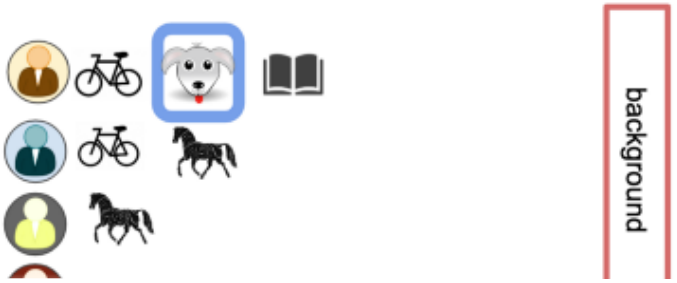
- You do not depend on recommendations for the discovery of new products. Because collaborative filtering requires co-occurrence to generate signals the algorithm has a big **cold start problem**. Any new item in the product catalogue has no co-occurrence and cannot be recommended without some initial engagement of users with

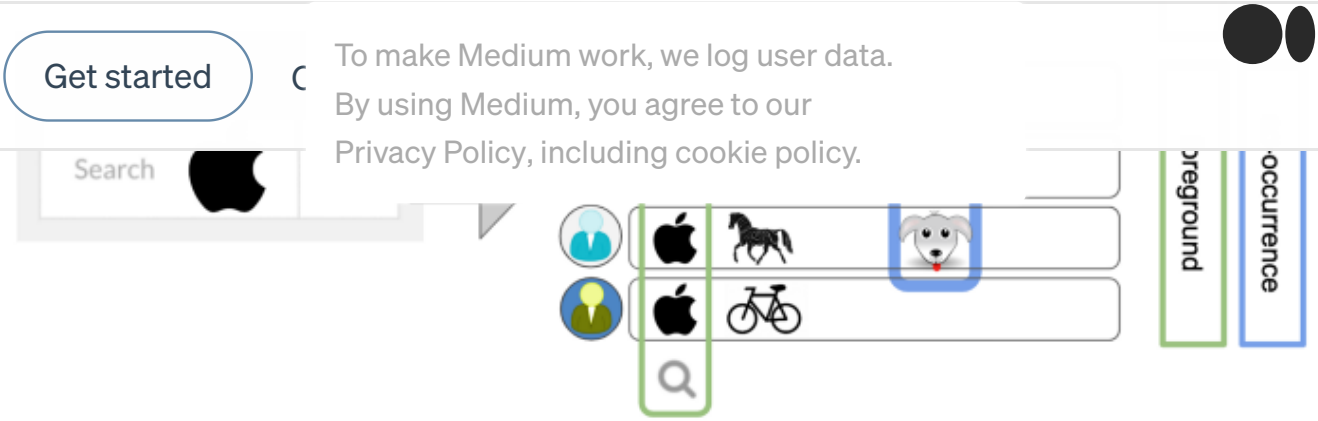
Collaborative Filtering Quick and Simple

One option would be to use Spark and the alternating least squares (ALS) algorithm (link) which is a simple solution for model training but does not provide an immediate solution for deployment and scoring. I recommend a different approach to get started:

As it turns out the maths of search and recommendation problems are strikingly similar. Most importantly, a good user experience in search and recommendations are almost indistinguishable. Basically, search results are recommendations if we can formulate recommendations as search queries. It's an ideal solution as many websites and businesses already operate search engines in their backends and we can leverage existing infrastructure to build our recommendation system. Elasticsearch scales well and exists as fully managed deployments e.g. on AWS. There is no safer bet if you want to deploy your recommendation engine into production fast!

How do you create a recommendation with a search engine?





1. We store all user-item interactions in a search index.
2. When a user is on a page for apples we search for all users who have apples in elasticsearch. This defines our foreground population.
3. We look for co-occurrence in our foreground which gives us puppies.
4. We search for puppies in the background population.
5. We calculate some kind of score for our puppy recommendation.

The good news: Elasticsearch implements all 5 steps for us in a single query!

If we store our user-item interactions in elastic search as follows

```
{
  "_id": "07700f84163df9ee23a4827fd847896c",
  "user": "user_1",
  "products": ["apple", "book", "lemon", "puppy"]
}
```

with a document mapping like this:

```
    products
}
```

then all what's needed to produce some recommendations is the following query e.g. using Python:

```
from elasticsearch import Elasticsearch,
RequestsHttpConnection
from aws_requests_auth.boto_utils import
BotoAWSRequestsAuth

es = Elasticsearch(
  host=host, port=port,
  connection_class=RequestsHttpConnection,
  http_auth=BotoAWSRequestsAuth(),
  scheme=scheme
)

es.search(
  index=index, doc_type=doc_type,
  body={
    "query": {
      "bool": {
        "must": {
          "term": {"products": "apple"}
        }
      }
    },
    "aggs": {
      "recommendations": {
        "significant_terms": {
          "field": "products",
          "exclude": "apple",
          "min_doc_count": 100
        }
      }
    }
  }
)
```

```
{
 ...
 "aggregations": {
    "recommendations": {
        "doc_count": 12200,
        "bg_count": 130000,
        "buckets": [
            {
                "key": "puppy",
                "doc_count": 250,
                "score": 0.15,
                "bg_count": 320,
            }
         ]
      }
   }
}
```

In our example, the search for apple returned a foreground population of 12,200 users with a background population of 130,000 users who did not have any apples in their products. Puppy co-occurred 250 times in the foreground and 320 times in the background. The JLH score is a simple magnitude of change between the background collection to the local search results given by **(fg_percentage - bg_percentage) * (fg_percentage / bg_percentage)** which gives a score of 0.15

As the JLH score is a magnitude change it's important to remember that fleas jump higher than elephants and the JLH score is very volatile for small data sets. You can adjust the `min_doc_count` parameter in the query to quality assure your recommendation results.

This is it, a simple but powerful first iteration of a recommendation engine which can be live within a week or less! Importantly, any version

recommendatic ptimise the UI and UX rather than the maths.

Next Steps

Elasticsearch is not just a very powerful backend for recommendations, it is also highly flexible! There're many options to improve our recommendation system while keeping the elasticsearch backend. Win!

Read part 2 for the more advanced use-cases when quick and simple is no longer good enough:

Advanced Use-Cases for Recommendation Engines

Part 2: How to build your next recommendation engine when quick and simple is no longer good...

towardsdatascience.com

Step 1:

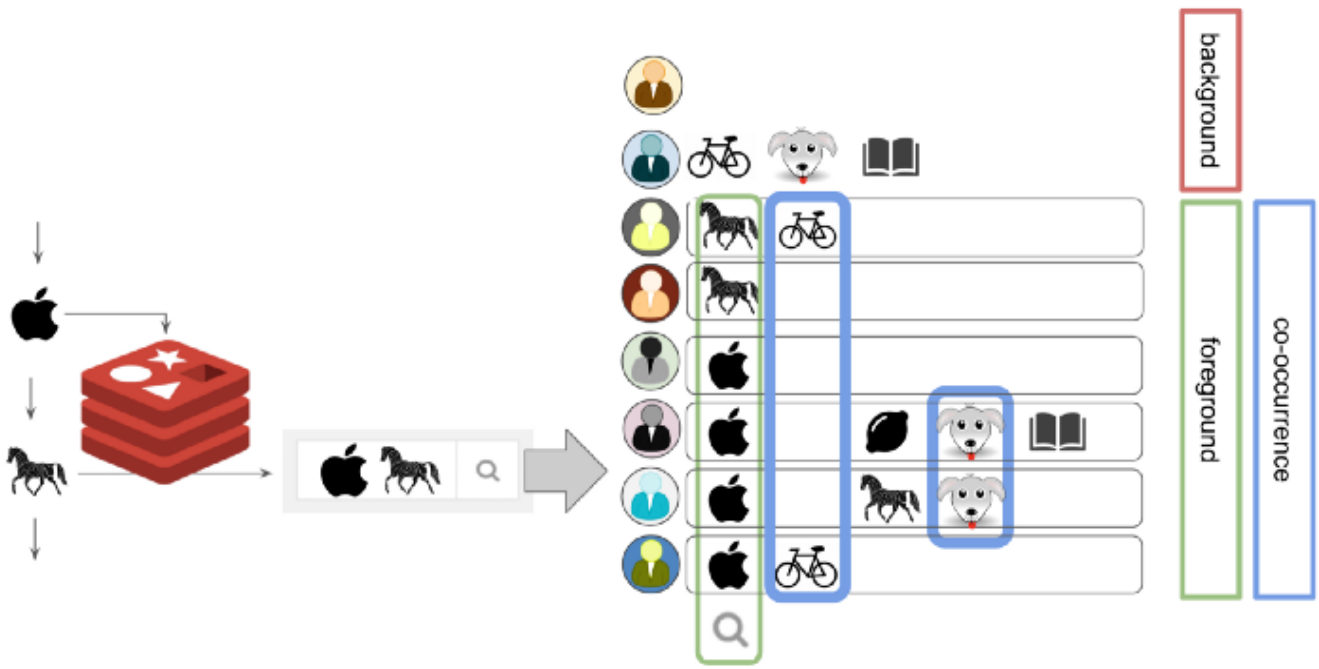
We can use more sophisticated algorithms such as ALS to create the indicators for our recommendations and we put these indicators into elasticsearch. This simplifies the recommendation scoring to a simple look-up as we do the heavy lifting in the training phase e.g. using Spark. This way elasticsearch is just a performant presentation layer of our ahead-of-time computed indicators for relevant recommendations. You can add this easily to an existing product catalogue as new metadata.

Step 2:

Without many cl re the product occurrence itself to capture a richer signal. We could use a click count. Or even better we could use a click score by normalising a click by the average expected click through rate of the page location generating the click. E.g. in a list of search results we can calculate an expected CTR for items in first position, second etc. We can then calculate the JLH magnitude change from the sum of item metric scores instead of their simple counts.

Step 3:

Users usually generate a series of events relevant for recommendations, e.g. clicking on multiple items or adding multiple products to a basket. It's worth adding a user interaction cache to your recommendation engine (1) to create more complex search queries using a series of events and (2) create a delta between the batch ETL process which updates your elasticsearch index and the user interactions which occurred since the last refresh of your recommendation engine.



population to ge ces in case of low traffic volumes or very big catalogues. It only needs a minor change to the elasticsearch query to switch to a `should` query:

```
es.search(
  index=index, doc_type=doc_type,
  body={
    "query": {
      "bool": {
        "should": [
          {"term": {"products": "apple"}},
          {"term": {"products": "pony"}},
        ],
        "minimum_should_match": 1,
      }
    },
    "aggs": {
      "recommendations": {
        "significant_terms": {
          "field": "products",
          "exclude": ["apple", "pony"],
          "min_doc_count": 10
        }
      }
    }
  }
)
```

The `minimum_should_match` parameter allows you to optimise between increasing the foreground population size or making the results more relevant by matching users with increasing similarity.

Step 4:

Currently, our search is a precise lookup of items. This has some consequences: everything we learn from user interactions in terms of

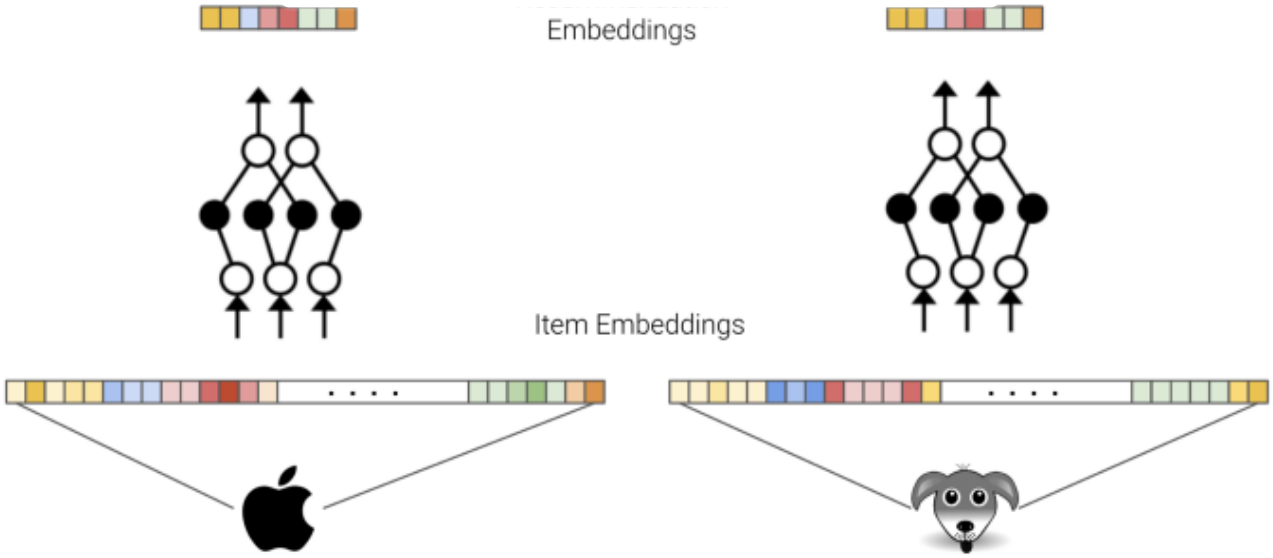
cannot generalis ples and green apples are distinctive items and co-occurrence is limited to precise matches of red apples or green apples. To overcome this we need to describe items mathematically to compute a similarity between items. This is called an embedding. Read my previous blog post were I create a geographic area embedding. Other options to create embeddings are auto-encoders or the matrix factorisation in the user-item model as described above. After we turned a simple product_id into an embedding we can use probabilistic or fuzzy search to find our foreground population and/or co-occurrences.

This should get you started with recommendations. It also gives you ample opportunity to build on your first iteration as you learn from production feedback.

> *The early steps into recommendations stand or fall much more often by UI and UX rather than the simplicity of the maths.*

Beyond Elasticsearch

Usually, products have a wealth of metadata we should use, e.g. price, descriptions, images, review ratings, seasonality, tags and categories. After we turn a rich metadata set into an embedding describing our products we can train a Neural Network to map the input embeddings into a recommendations embedding which has (1) lower dimensionality and (2) a desired behaviour of the cosine similarity of suitable recommendations being high. One great solution for this are Siamese Neural Networks.



The input is a high dimensional vector of concatenated embeddings of a product's metadata. The output of the Neural Network is a much more compact recommendation embedding vector. The error function is given by the cosine similarity of the output vectors. We can use the collaborative filtering data to create our supervised learning labels for combinations which should be similar or not. Importantly, in siamese neural networks the weights of both networks are always identical which gives them their name. Such a recommendation engine would have no more cold start issue! Finally, producing a recommendation can be done with a k-nearest-neighbour search of the output recommendation embeddings.

You can read more on next steps in the follow up:

Advanced Use-Cases for Recommendation Engines

Part 2: How to build your next recommendation engine when quick and simple is no longer good...

towardsdatascience.com

Jan is a successful thought leader and consultant in the data transformation of companies and has a track record of bringing data science into commercial production usage at scale. He has recently been recognised by dataIQ as one of the 100 most influential data and analytics practitioners in the UK.

Connect on LinkedIn:
https://www.linkedin.com/in/janteichmann/

Read other articles: https://medium.com/@jan.teichmann

Machine Learning Data Science Recommendations Elasticsearch