eCommerce Recommendation Systems: A Collaborative Filtering Approach

Oliver Benham

Abstract

To maintain a competitive advantage, eCommerce businesses need to offer an extensive and varied inventory of products, while also making it easy for their customers to choose what they want to buy. Recommendation systems enable businesses to personalise their product catalogue in order to offer the most relevant choices for each customer, leading to uplift in conversion rates, basket value and engagement metrics. This whitepaper, written with technical professionals in mind, provides details of Intechnica's algorithms and their implementation in the cloud. We explain how to go from raw unprocessed data to a production ready recommendation system, across a variety of different personalisation approaches.

Introduction

Recommendations are common place in many areas of the internet and there are a wide range of algorithms that can be used to produce them[8]. Classically, the main techniques used to produce recommendations are based on collaborative filtering and content based filtering, or some combination of the two[8].

Collaborative filtering algorithms rely on past user behaviour to find relationships between users and interdependencies between products that are used to identify new user-item or item-item associations[6]. Content based filtering algorithms try to match up user or item characteristics, such as preferences or features, with the attributes of an item in order to produce user-item or item-item recommendations[7].

Although Intechnica's static recommendation algorithms and dynamic recommendation algorithms are very different, they both fall into the bracket of collaborative filtering as they use a history of customer data to produce recommendations. The remainder of this white paper describes example applications and explains the mathematics that underpin our proprietary algorithms. In addition, we describe the technical implementation; how our system works in production and the AWS (Amazon Web Services) architecture that holds everything in place.

Static Recommendations

For the purpose of this article, *static recommendations* are recommendations that are produced using statistical and mathematical techniques commonly used in recommendation systems. This section demonstrates how our static recommendation algorithms are relatively straight forward to understand and implement. Static recommendations can be used in a variety of situations, two of which are described in subsections below: *Static Recommendations For Email Campaigns* and *Static Recommendations For Product Pages*.

There are a wide range of recommendation algorithms that are used across the internet, many of which make use of a variety of data, such as customer product ratings, historic transactions, product attributes etc[8]. However, there are

often situations in which only a limited amount of data is available. Intechnica's static recommendation algorithms, inspired by the item-to-item collaborative filtering algorithm described by G Linden et al in [4], attempt to address the issue of producing recommendations with limited data. The following section is dedicated to explaining how this is done.

Product Co-occurrence

The foundation of our static recommendations is product co-occurrence. Co-occurrence is a basic technique that extracts implicit feedback from historical customer data, such as transaction and browsing history, to lay the foundations for recommendations. At a basic level, the only information needed to run the co-occurrence algorithm is a product identifier, such as a unique product name or product ID, and a feature to aggregate on, such as a customer ID, session ID, order number etc. Aggregate features need careful thought because they determine the kind of product co-occurrence that is calculated. Choosing the correct aggregate feature for a task should be simple after reading the remainder of this section.

Instead of using general terms to describe the algorithm we will use a specific example. The example we will focus on uses a unique customer ID to distinguish between customers based on their account information and a product ID to uniquely identify products that can be purchased on a website. An example of the initial dataset is depicted in figure 1 below.

customer_id	product_id
17119	4897
10342	3100
11329	1039
15987	1142
15987	3198
etc...	etc...

Figure 1: This table contains the top five rows of a dataset consisting of customer ID's and product ID's. The customer ID's uniquely identify each customer and the product ID's uniquely identify the product that the customer has purchased.

The first step in the process is to *one-hot encode* the product ID column. This converts the product ID column into multiple columns, each one corresponding to a specific product, containing ones (indicating a purchase) and zeros (no purchase). We then aggregate the table using the customer ID and take the maximum of each product column. The resulting table contains one row per customer, with each product column containing either a 1 or a 0, depending on whether that customer has previously purchased that product (1) or not (0). Continuing with the example introduced above, the resulting table can be seen in figure 2 below.

customer_id	1009	1039	1102	1142	1220	etc..
17119	0	0	0	0	0	...
10342	0	0	0	0	0	...
11329	0	1	0	0	0	...
15987	0	0	0	1	0	...
etc...	...	...	...	...	...	...

Figure 2: The customer ID column in this table is a unique list of the customer ID's that appear in figure 1 above. Each of the remaining columns correspond to one product ID. A 1 in any product column indicates that the customer on that row has previously purchased the product. In this example the customer with ID 15987 *has previously purchased product* 1142 *but none of the other products that appear in the first five product columns.*

We are now able to calculate the product co-occurrence matrix. Firstly, let X represent the table in figure 2 above, excluding the customer ID column and the product ID column headers. We can view this table X as a $c \times m$ matrix, where c is the total number of customers in our dataset and m is the number of different products. The

product co-occurrence matrix N is then given by

$$N = X^T X, \tag{1}$$

where X^T is the transpose of X. The resulting N looks like the following.

$$N = \begin{bmatrix} n_{11} & n_{12} & n_{13} & \dots & n_{1m} \\ n_{21} & n_{22} & n_{23} & \dots & n_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n_{m1} & n_{m2} & n_{m3} & \dots & n_{mm} \end{bmatrix}$$

In this matrix the component n_{ii} corresponds to the total number of customers that have purchased the i^{th} product and n_{ij} is the total number of people that have purchased both the i^{th} product and the j^{th}. We can intuitively deduce from this that $n_{ij} = n_{ji}$, and therefore the resulting co-occurrence matrix must be symmetric. This can also be easily deduced from basic linear algebra and the fact that

$$N^T = (X^T X)^T = X^T (X^T)^T = X^T X = N \text{ [1]}, \iff N \text{ is symmetric.}$$

To help explain this in other words, we can view this matrix as a table and continue with the example introduced in figure 1. An example table can be seen in figure 3 below.

product_id	1009	1039	1102	1142	1220	etc..
1009	1800	100	70	125	40	...
1039	100	2500	190	60	30	...
1102	70	190	2150	310	75	...
1142	125	60	310	3250	110	...
1220	40	30	75	110	975	...
etc..	...	...	...	...	...	...

Figure 3: This shows an example table version of a co-occurrence matrix. Product ID's make up both the column and row indices. In this example, 1800 customers have purchased the product with ID 1009 and of these 1800 customers 100 have also purchased 1039 etc.

This co-occurrence matrix is the foundation of our static recommendation algorithms. The idea behind the co-occurrence matrix is that it enables us to capture the combinations of products that are purchased (or viewed if you're considering browsing data) together; with the most frequent combinations having a larger impact on the resulting recommendations.

An example of using the co-occurrence matrix to produce recommendations is given in the section *Static Recommendations For Product Pages*. The co-occurrence matrix can also be used as a foundation to make recommendations to each customer. This idea is expanded upon in the section *Static Recommendations For Email Campaigns*. Before covering some of the case studies where we've used product co-occurrence, we are going to describe some of the issues with using the co-occurrence matrix in its current format and the methods that can be used to help compensate for these issues. Going forward we will refer to the co-occurrence matrix described above as the *basic co-occurrence matrix*.

The main issue with the basic co-occurrence matrix is its susceptibility to producing recommendations that are biased toward popular items, meaning products that are less popular can be overlooked. A consequence of this popularity bias is that when using the basic co-occurrence matrix to produce recommendations, the *coverage* can be quite low. Coverage, specifically *item space coverage*, is the ratio of items that are able to be recommended to users[9]. In mathematical terms, coverage is defined as

$$cov = \frac{|I_r|}{|I|}$$

where $|I_r|$ is the number of distinct items that can be recommended to users and $|I|$ is the total number of items. Low coverage hinders users from finding items to consume, impacting users' satisfaction and the overall sales of the system[9]. The question is, how can we change the basic co-occurrence matrix described in this section to improve the coverage of our recommendations and reduce the popularity bias. In the proceeding two sections we will explore different methods that attempt to answer this question.

Similarity Matrix

The quality of recommendations depends heavily on what we mean by products being "related" [2]. This leads us to the question; how can we increase the coverage (the number of different products we are able to recommend) of our recommendations by finding relations between items and take into consideration the popularity of the item? This section follows an approach used by Amazon, as outlined by B Smith et al in [2], to provide an answer to this question.

This method accounts for popularity by finding the difference between the basic co-occurrence matrix and an expected co-occurrence matrix, to highlight values that are higher than expected. To explain how we calculate the expected co-occurrence matrix, consider two distinct products, X and Y, and the group of customers that have previously purchased X. For the group of customer who have purchased X, we now show how to calculate the expected number that have also purchased product Y.

The intuitive way of estimating $\mathbb{E}_{Y|X}$, the expected number of customers who have purchased Y, given that they've purchased X, is to multiply the number of people who have purchased X by the probability of purchasing Y, $\mathbb{P}(Y)$. $\mathbb{P}(Y) = |C_Y|/|C|$, where C_Y corresponds to the customers that have purchased Y and C represents the customers in the whole cohort. The problem with using $|C_X|\mathbb{P}(Y)$ as our estimate of $\mathbb{E}_{Y|X}$ is that we are assuming that the customers who have purchased X have the same probability of purchasing Y as the general population, which is generally not the case[2]. In more mathematical terms $\mathbb{P}(Y|X) \neq \mathbb{P}(Y)$. Surprisingly, for most pairs of items $\mathbb{P}(Y|X) > \mathbb{P}(Y)$.

To explain this further, imagine the following scenario; there are three customers, c_1, c_2 and c_3. c_1 has purchased every item available, c_2 has purchased 100 of the available items and c_3 has purchased 10. When we look at all the customers who have purchased X then c_1 is certain to be included, similarly c_2 is approximately 10 times more likely to be included then c_3. The consequence of these varying purchase quantities is that sampling the customers that have purchased a random product doesn't give a uniform probability of selecting customers. Any sample is generally biased toward customers who have purchased more products and therefore, for any product X, a random customer who has purchased X is more likely to have purchased an additional product Y than a customer from the general population.[2]

This means that we have to take into consideration the individual customers who purchased X when calculating $\mathbb{E}_{Y|X}$, as opposed to just the total number of customers who have purchased X. For any customer c_i to calculate the probability that they've purchased Y, given that they've purchased X, we need to account for the number of different products c_i has purchased. To help construct the equation for $\mathbb{P}(Y|X)$, let's examine a few simple examples. Initially, imagine c_i has purchased X and one other product. In this situation the probability of c_i having purchased Y, given that they've purchased X, is simply $\mathbb{P}(Y)$. Similarly, in the case where c_i has purchased three products then the probability that they've purchased Y, given that they've purchased X, is given by

$$\begin{aligned}
&\mathbb{P}(Y)^2 + \mathbb{P}(Y)(1-\mathbb{P}(Y)) + (1-\mathbb{P}(Y))\mathbb{P}(Y) \\
&= \mathbb{P}(Y)^2 + 2\mathbb{P}(Y)(1-\mathbb{P}(Y)) \\
&= 2\mathbb{P}(Y) - \mathbb{P}(Y)^2 \\
&= 1 - (1-\mathbb{P}(Y))^2.
\end{aligned}$$

The format of the final line above is important as this helps explain the general case. For a customer c_i, who has purchased X, let $|c_i|$ be the total number of items they've purchased, then the probability that they've also purchased Y is given by

$$\mathbb{P}_{c_i}(Y|X) = 1 - (1-\mathbb{P}(Y))^{|c_i|-1}. \tag{2}$$

Using (2) we sum over all customers who have purchased X, C_X, to get $\mathbb{E}_{Y|X}$. In mathematical terms,

$$\mathbb{E}_{Y|X} = \sum_{c \in C_X} \mathbb{P}_c(Y|X) = \sum_{c \in C_X} 1 - (1-\mathbb{P}(Y))^{|c|-1}. \tag{3}$$

After equation (3) has been calculated, we combine the resulting expected co-occurrence value with the corresponding basic co-occurrence value to help determine whether or not N_{YX}, the actual number of customers who purchased X and Y, is higher or lower than expected. One way to combine N_{YX} and $\mathbb{E}_{Y|X}$ is to take the difference $N_{YX} - \mathbb{E}_{Y|X}$. This gives an estimate of the number of non-random co-occurrences.

Another way to combine the two is to take the percentage difference, $[N_{YX} - \mathbb{E}_{Y|X}]/\mathbb{E}_{Y|X}$. Both of these approaches have their issues, with the first being biased toward more popular products and the percentage difference making it too easy for unpopular or niche items to get higher scores. There is no single metric that works best in all settings but the one we've opted for is the chi-squared score, given by

$$[N_{YX} - \mathbb{E}_{Y|X}]/\sqrt{\mathbb{E}_{Y|X}}. \quad [2] \tag{4}$$

After (4) has been calculated for all combinations of products, we construct the resulting *similarity matrix* S. The similarity matrix is an $m \times m$ matrix where m is the total number of products. Each entry of the similarity matrix, excluding the diagonals, is computed using expression (4) above. In mathematical terms,

$$S_{ij} = [N_{ij} - \mathbb{E}_{i|j}]/\sqrt{\mathbb{E}_{i|j}}, \quad i \neq j.$$

The diagonal elements of the similarity matrix are the same as the diagonal elements of the matrix N.

$$S_{ii} = N_{ii}. \tag{5}$$

The main reason for (5) is the computational ease when producing recommendations; the diagonals could be set to zero but it makes the algorithmic implementation harder. One important point to be aware of is that, although $N_{YX} = N_{XY}$ for all X, Y, there exists X, Y such that $\mathbb{E}_{Y|X} \neq \mathbb{E}_{X|Y}$ and therefore the similarity matrix is not symmetric. As a consequence, both $\mathbb{E}_{Y|X}$ and $\mathbb{E}_{X|Y}$ need to be calculated for all X and Y.

Below is a diagram that shows how using the similarity matrix to produce recommendations can increase coverage. For this experiment we use a dataset from [11] and [12], specifically the Amazon instant video dataset which can be downloaded from http://jmcauley.ucsd.edu/data/amazon/. The dataset contains a user ID, item ID, a rating and a timestamp. We ignore the timestamp and the user rating because we are only interested in non-temporal implicit feedback. The initial dataset contains $529,711$ rows, $426,922$ unique customers and $23,965$ unique products. To calculate the coverage, we generate recommendations as described in the section *Static recommendations for email campaigns.*

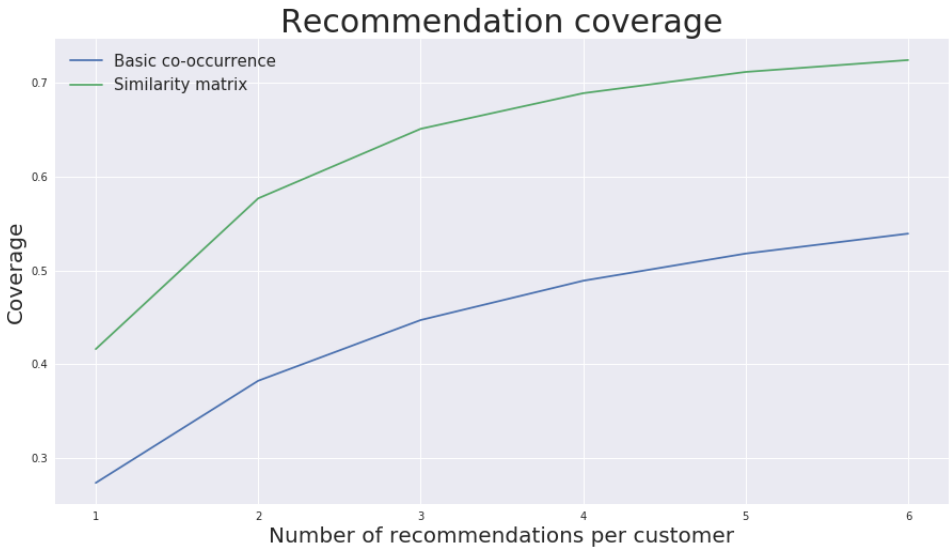


Figure 4: The coverage of recommendations changes when using both the basic co-occurrence matrix and the similarity matrix to produce recommendations, depending on the number of recommendations displayed. For the Amazon Instant Video dataset, using the similarity matrix improved the coverage of the recommendations dramatically. A recommendation system with a higher coverage exposes customers to a larger range of products and subsequently reduces the effects of popularity bias.

Details of how to use the similarity matrix to produce recommendations can be found in the sections *Static Recommendations For Email Campaigns* and *Static Recommendations For Product Pages.*

Segmented Co-occurrence

In this subsection we describe how to combine the basic co-occurrence matrix with customer segmentation and how this can improve the coverage of the resulting recommendations. This method, known as segmented co-occurrence, is arguably more simplistic than the similarity method described above but similarly lays the foundation for us to produce more meaningful recommendations.

Continuing from the basic $m \times m$ co-occurrence matrix N as described in equation (1). The dataset used to generate the basic co-occurrence matrix is now split into a variety of smaller datasets, using a segmentation feature such as a product attribute or customer segment. We then produce a co-occurrence matrix for each subset of the initial dataset. Another way of thinking about this is to extend the currently two-dimensional co-occurrence matrix by adding an additional dimension. Each element of this additional dimension will represent a different segment or product attribute. For example, if we bucket customers into age brackets, $18-29$, $30-49$ and $50+$, we end up with three separate co-occurrence matrices. Mathematically, this is like going from a matrix to a three-dimensional tensor. A visual depiction of this can be seen below.

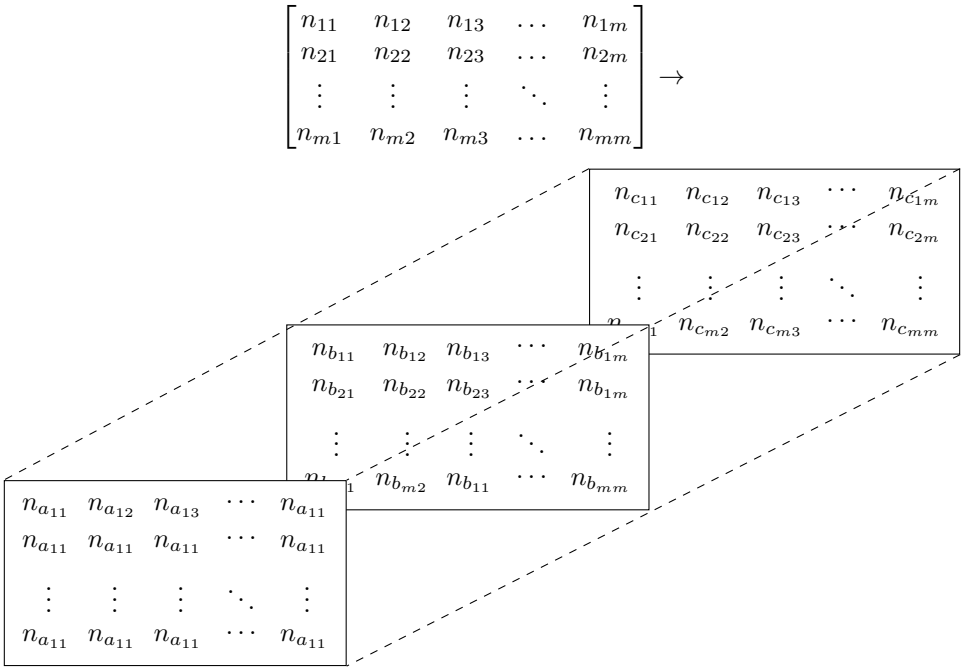


Figure 5: This figure depicts an example of incorporating a segmentation feature into a basic co-occurrence matrix. Here the subscripts a, b, and c represent the different segments used to split the basic co-occurrence matrix. In this case n_{11} is split into $n_{a_{11}}$, $n_{b_{11}}$ and $n_{c_{11}}$.

In our experience, when using the basic co-occurrence matrix to produce recommendations the most popular products tend to dominate. However, by using segmented co-occurrence it is possible to reduce popularity bias and increase the coverage of recommendations, provided the segmentation feature produces sufficiently diverse co-occurrence matrices.

Using segmented co-occurrence enables customers in specific segments to receive recommendations based on products other customers in the same segment have purchased. Similarly, for products with a specific attribute, it's possible to produce recommendations for other goods with the same attribute. To summarise, combining co-occurrence with segmentation can help shift from recommending the most popular products overall to popular products from a tailed subset, resulting in more personalised recommendations and increased coverage.

An example of the effect segmentation can have on coverage can be seen in figure 6 below. We have used a dataset from [10], specifically the RentTheRunway dataset, which can be downloaded from https://cseweb.ucsd.edu/ jmcauley/datasets.html. This dataset contains customer clothing review information. It contains $5,850$ products, $105,571$ individual users and $192,544$ total rows. To construct a segmentation feature we used the items' numerical sizes to construct three categories: small, medium and large. To calculate the coverage we generate recommendations as described in the section *Static Recommendations For Email Campaigns*.

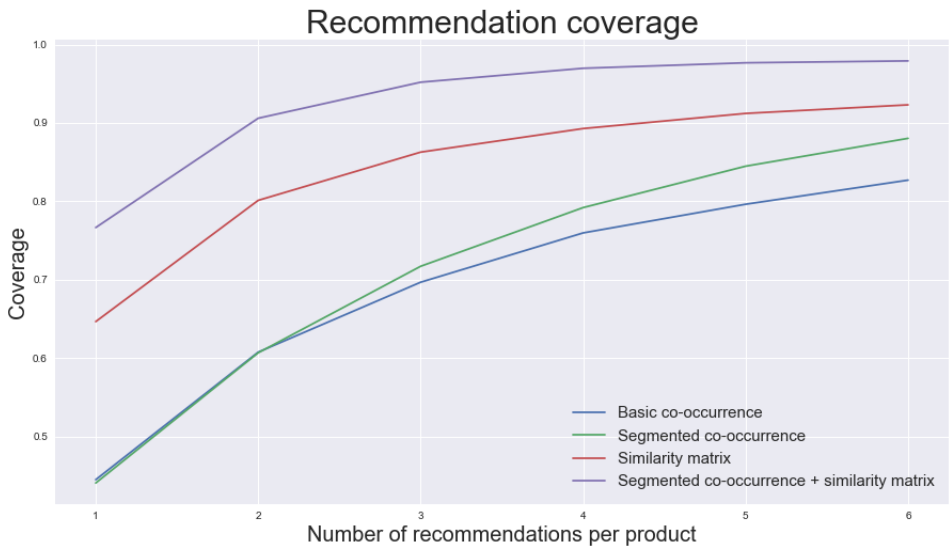


Figure 6: This figure shows how the coverage of the recommendations changes dependent on the chosen algorithm. When compared to basic co-occurrence coverage, using segmented co-occurrence increases the coverage slightly as we increase the number of recommendations. The best coverage is produced when segmentation is combined with the similarity matrix. A higher coverage means more products can be recommended to customers.

Choosing an appropriate segmentation feature is essential for this method to be successful. For example, imagine splitting the basic co-occurrence matrix into three identical matrices (values permitting); the co-occurrence values between products would all be the same and therefore the customers in each segment would receive the same recommendations as they would if the basic co-occurrence matrix was kept intact. We use an equation to calculate the *equivalence score* between matrices and this is used as a guide to help determine which segmentation features are useful. The equivalence score between $\mathtt{k}$ matrices can be calculated as follows. Let $d(A, B)$ be the $L1$ norm of the difference between two $m \times m$ matrices A and B, then

$$d(A, B) = \sum_{i=1}^{m} \sum_{j=1}^{m} |a_{ij} - b_{ij}|$$

Let N_k, a $m \times m$ matrix, represent one of the sub matrices, where $1 \leq k \leq \mathtt{k}$ denotes the specific sub matrix. The equivalence score eqv is given by

$$eqv = \frac{1}{\mathtt{k}} \sum_{k=1}^{\mathtt{k}} \sum_{l>k}^{\mathtt{k}} d(N_k, N_l)$$

In words, the equivalence score is the sum of the average pairwise difference between elements in different sub matrices.

Although the equivalence score is a useful metric that can be used to help differentiate between segmentation features, it can't be relied on entirely. Splitting the basic co-occurrence matrix by using a segmentation feature is a balancing act between producing more diverse recommendations (increasing coverage) and having enough data to actually find meaningful co-occurrences between products. If the segmentation feature produces *too many* sub matrices then there may not be enough data to populate the sub matrices with meaningful values and this is likely to lead to nonsensical recommendations. We have not established any specific rules around the amount of data required but try to ensure that the number of rows in each segmented dataset (see *figure 1* for an example) is at minimum a couple of orders of magnitude larger than the number of products in that segment.

Static Recommendations For Email Campaigns

Using email to communicate with customers is fundamental to most eCommerce businesses. In addition to receiving information via email about orders or specific offers, customers now expect personalised emails with recommenda-

tions that correspond to their buying patterns and browsing behaviour.

This section describes how you go from any co-occurrence matrix or similarity matrix to produce recommendations for individual customers that can be used to power personalised email campaigns. We use the term product matrix as an umbrella term to represent any co-occurrence or similarity matrix. Operationally all matrices are interchangeable, hence the generalization.

For a customer c_i, we define the customer product vector $\mathbf{c}_i$ to be an m length vector, where m is the number of products that can be recommended. Let $\mathbf{c}_{i_j}$ be the j^{th} element of vector $\mathbf{c}_i$ then,

$$\mathbf{c}_{i_j} = \begin{cases} 1 & \textit{if user } i \textit{ has purchased product } j, \\ 0 & \textit{otherwise} \end{cases}$$

We define the *possible recommendations* vector as $\tilde{\mathbf{c}}_i$, where, similarly to above, $\tilde{\mathbf{c}_{i_j}}$ is the j^{th} element of vector $\tilde{\mathbf{c}}_i$ and

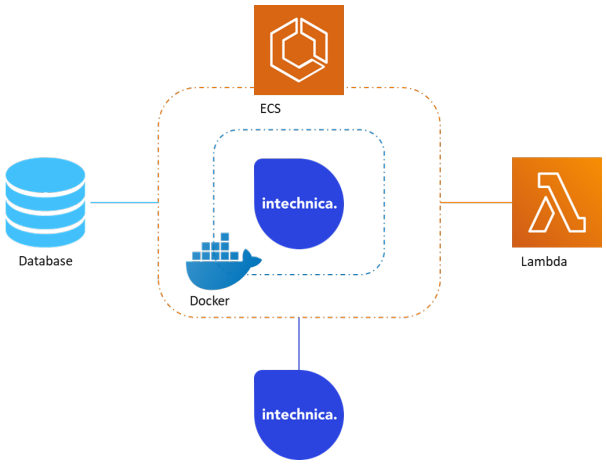
$$\tilde{\mathbf{c}_{i_j}} = \begin{cases} 0 & \textit{if user } i \textit{ has purchased product } j, \\ 1 & \textit{otherwise} \end{cases}$$

We now combine the product matrix with the customer product vector and the possible recommendations vector to produce a results vector which we use to directly make recommendations. Let N be the product matrix (either co-occurrence or similarity), then the recommendations vector $\mathbf{r}$ is given by

$$\mathbf{r} = N\mathbf{c}_i * \tilde{\mathbf{c}}_i \tag{6}$$

Here $*$ represents the element-wise multiplication of the two vectors, $N\mathbf{c}_i$ and $\tilde{\mathbf{c}}_i$. The resulting vector $\mathbf{r}$ is therefore an m length vector where the values indicate the customer affinity to each product. To make recommendations using $\mathbf{r}$ we simply choose the products with the highest corresponding values. This process can be used in email campaigns to recommend specific products to individual customers based on what they've previously purchased or viewed. The technical details of this application are described in the next section.

Implementation



Our application that creates the recommendations is containerised and stored in ECS (Elastic Container Service). The application is triggered by a Lambda function. Triggering can happen on a either a periodic basis or in response to a specific action e.g. uploading a file to Intechnica's bespoke client portal. The results are then available to download via our portal for use in a CRM system or we can integrate our application with an existing CRM system.

Static Recommendations For Product Pages

Another common feature of many modern eCommerce businesses is product page recommendations; recommendations that appear on a page that is dedicated to a specific product. More often that not, product page recommendations relate to the specific product on display. This section describes how you can use any co-occurrence matrix or similarity matrix to produce recommendations for product pages.

Static recommendations for product pages are arguably more simplistic and easier to implement then the individual recommendations for email campaigns as described in the section above. The recommendations we produce for product pages can be deduced directly from any co-occurrence or similarity matrix. We will again use the umbrella term product matrix to describe all these matrices as they are functionally the same in this setting.

For our product page recommendations, instead of using transactional data, we tend to use web browsing data. Gathering this data is described in more detail in the following section on implementation. We use browsing data because there is a larger quantity of data available for any given time period and it helps us capture the combinations of products that people are viewing as well as buying.

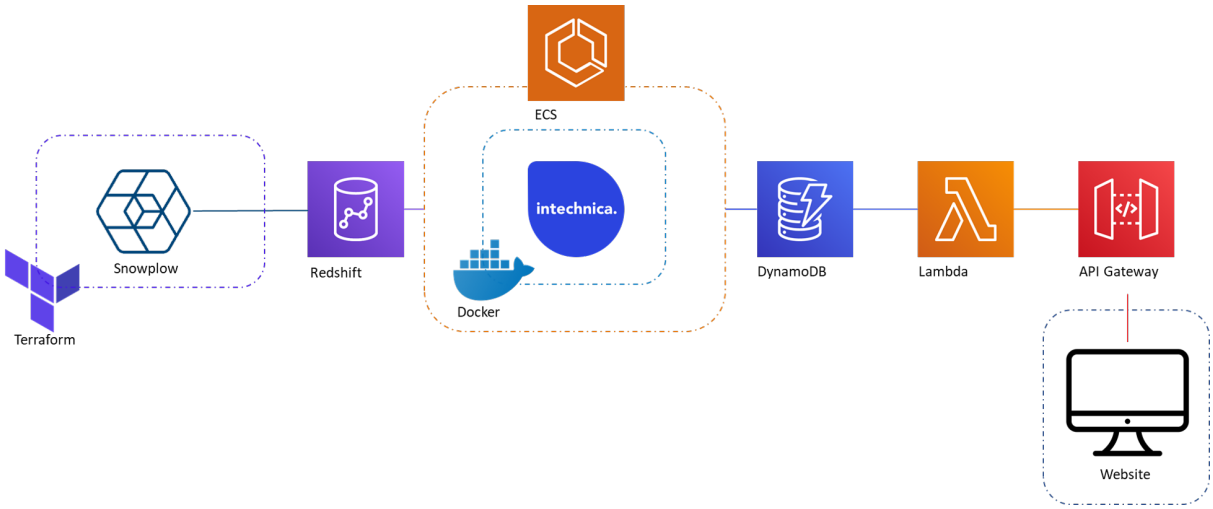
The main features needed to produce a product matrix from browsing data is a session ID (or an equivalent identifier that uniquely identifies each customer session on the website) and a product ID (or an equivalent identifier that uniquely identifies each product on the website). Once a product matrix has been produced from the browsing data, for each column in the product matrix we simply pick the products that correspond to the top n values, ignoring the diagonals. Here n is the number of products we want to recommend.

It is important to note that for a segmented or basic co-occurrence matrix it doesn't matter if the rows or columns are used to determine which product recommendations go with each product because these matrices are symmetric. However, for the similarity matrix, it must be the columns that are used because of the way the matrix is constructed and the fact that it is not symmetric. To help explain this, imagine the first column of the *similarity matrix*, constructed using equation (4) and (5).

$$\begin{bmatrix} S_{11} \\ S_{21} \\ S_{31} \\ \vdots \\ S_{m1} \end{bmatrix} = \begin{bmatrix} N_{11} \\ (N_{21} - \mathbb{E}_{2|1})/\sqrt{\mathbb{E}_{2|1}} \\ (N_{31} - \mathbb{E}_{3|1})/\sqrt{\mathbb{E}_{3|1}} \\ \vdots \\ (N_{m1} - \mathbb{E}_{m|1})/\sqrt{\mathbb{E}_{m|1}} \end{bmatrix}$$

In this case, excluding the first element, all other values are conditional on a customer having purchased (or in this case viewed) product one. It therefore only makes sense to use this column vector when producing recommendations associated with product one.

Implementation



The diagram above depicts the main elements used in our product page recommendations architecture. A Terraform script deploys Snowplow[13] onto the desired website. We use Snowplow to capture web browsing data when clients don't have an existing data source. The browsing data produced by Snowplow data is stored in a Redshift database. The data in this Redshift database is then used to calculate product recommendations, which are subsequently stored in DynamoDB. Our recommendations application is containerised and stored in ECS. The website that displays the recommendations can then make calls to API Gateway which triggers a Lambda function to retrieve the required recommendations from DynamoDB.

That concludes our section on static recommendations. We are now going to move onto another type of recommendation system that Intechnica specialise in, *dynamic recommendations*

Dynamic Recommendations

Our dynamic recommendations utilise modern deep learning techniques to produce recommendations. We use a neural network consisting of LSTM layers and fully connected layers to make predictions about the most likely products a customer will purchase based on their browsing behaviour. The inspiration for the proceeding section came from S. Wu el al as describe in [3]. The remainder of this section describes the steps to produce our dynamic recommendations.

The first step is to vectorise the URL paths of the website that will be hosting the recommendations. In other words, we give each URL path a vector representation. To help capture connections between products, such as products that appear on the same list page, where possible we try and capture the website structure in the vectorisation process. To demonstrate this, consider a simple fictional URL of a website with the domain name removed: /holidays/uk/manchester. Instead of one-hot-encoding the URL as it is, we first split it up into its constituent parts: holidays, uk and manchester. Once this is done for all URLs, we chose which elements we want to keep from each section and then one-hot-encode the three sections individually. The one-hot-encodings for each section are then combined to produce the *path vector*. For example, imagine there are three possible options for the first section that we are interested in, holidays, mini-breaks and other. Say there are 15 possible options for the second section and 30 possible options for the third. This gives a resulting path vector of length $48 = 3 + 15 + 30$.

We can now represent each customers journey through the website as a collection of path vectors. At a basic level it is these vectors that are fed into our neural network. The next processing step is to remove all sessions from customers who have never made a purchase. Depending on the amount of data available, it is also preferable to remove the sessions from customers who have made purchases across multiple sessions.

We then create *history vectors* from historical customer sessions that do not contain any purchases. A history vector is a single vector that summarises a collection of path vectors that are generated by a single customer in one session. The reason this is useful is because it enables us to capture some of the customer's behaviour from previous sessions, before a purchase was made. We found this was useful during the model training phase, especially for examples with only a handful of path vectors.

There are several possible options when it comes to computing history vectors. The one we chose to use is a finite harmonic series. An example of this is as follows; consider three vectors

$$\mathbf{t_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \end{bmatrix}, \mathbf{t_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \end{bmatrix}, \mathbf{t_3} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Here $\mathbf{t_1}$ represents the first path visited in time. The history vector $\mathbf{h}$ is given by the following formula

$$\mathbf{h} = \sum_{i=0}^{n-1} \frac{1}{n+1} \mathbf{t_{n-i}} \tag{7}$$

For the three vector example we get

$$\begin{aligned} \mathbf{h} &= \mathbf{t_3} + \frac{1}{2}\mathbf{t_2} + \frac{1}{3}\mathbf{t_1} \\ &= \begin{bmatrix} \frac{3}{2} & \frac{4}{3} & 0 & \frac{4}{3} & \frac{1}{2} \end{bmatrix} \end{aligned}$$

Once the history vectors have been computed for all relevant customer sessions, we combine the history vectors for customers who have more than one. These history vectors are combined in the same way as the individual path vectors are combined as described by equation (7) above.

We are now left with a dataset that, for each customer, contains a time series of path vectors from a session where a purchase has been made and, if the customer has historical sessions, a history vector to summarise their behaviour. The penultimate processing step is to remove any website requests (path vectors) from individual examples that occur after a purchase has been made and limit the number of rows (path vectors) before the purchase is made to a pre-set maximum.

Consider an example with 45 rows and no history vector. If the purchase occurs at row 40 and the maximum number of rows we are interested in is 30 then we remove the last five rows and the first 10. The reason we set this maximum is because our model input needs a maximum value. Although the maximum number of rows is a subjective choice, it can be aided by looking at the distribution of session lengths. We choose this maximum value such that approximately 75% of session lengths are less than this value.

The final step is to scale the summary history vector rows so that each element in the row is at most one. The reason for this being that neural networks are highly sensitive to the scaling and distribution of their input features[5] and scaling these vectors ensures that all input values lie in the range $[0, 1]$. This type of input scaling helps stabilise the network and therefore improve performance of the model. Figure 7 below depicts an example of the kind of input our model is expecting.

$$\begin{bmatrix} 0 & 1 & 0.2 & 0.1 & 0.6 & 0.2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 7: This matrix is an example of an input that our model could receive. The top row is the summary history vector and the remaining rows correspond to individual path vectors from a session that contains a purchase. The purchase event is represented by the row at the bottom. In this specific example, the customer clicked on five different paths when making a purchase and has at least one historical session summarised by the history vector at the top.

The model is trained in a conventional way that one could train a neural network used for classification, with cross entropy loss and Adam optimization. The main difference between this model and a conventional classification model is that conventional accuracy is not used as the evaluation metric; we use top k accuracy instead. This means we consider a single prediction to be correct if the correct class appears in the top k predictions, in terms of probability. For this recommendation model we are satisfied if the model can reliably predict the correct class in the top four or five predictions. The main reason being that recommendations are usually displayed in groups of four or five, depending on the website, so if the model can capture the general trend then it can be used to serve up recommendations.

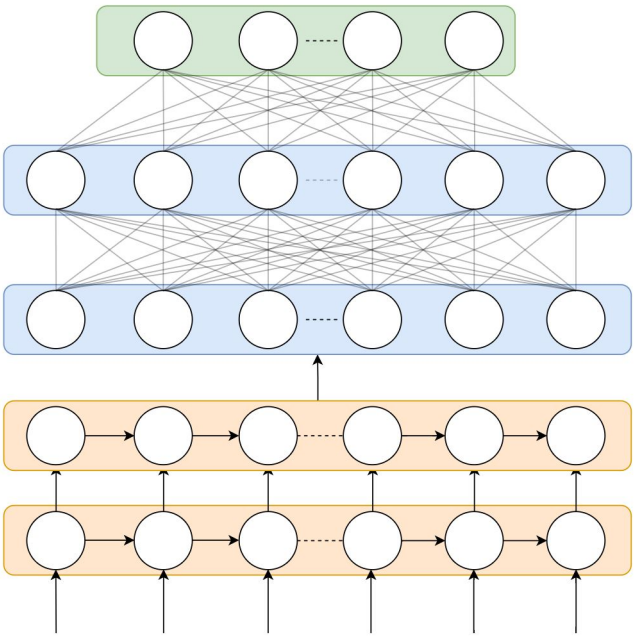


Figure 8: An example of the kind of architecture used in our model. Here the orange layers $(1, 2)$ are LSTM layers. The blue layers $(3, 4)$ are standard fully connected layers and the green layer (5) is a softmax layer. The single

arrow between $(2, 3)$ *indicates that only the last output from the second LSTM layer is passed to the first fully connected layer.*

There are some additional considerations that are worthwhile discussing. Firstly, as some readers will have questioned, what happens in the situation where a customer has purchased multiple products in one order? In this case we duplicate the training examples such that there is one for each product the customer purchased. This ensures the model learns some of the underlying structure of which pages contribute to the purchase of each product and simultaneously builds up an idea of which products are likely to be purchased together. Secondly, not all websites are well suited to training this kind of model. Websites with a hierarchical structure that is reflected in their URL paths are best placed for these recommendations. Additionally, a large number of purchases and a low ratio of products to purchases is necessary for good accuracy. In other words, it is desirable for the number of purchases to far outweigh the number of products and ideally each product will have thousands of purchases. There are some options available for websites with a vast amount of products such that it would be impractical to train our model with individual products as the target. In this situation, we adjust the problem so that, instead of using individual products, the model is trained using a higher-level product attributed e.g. category, brand, etc. There is then some additional business logic put in place to choose which products to recommend based on the attribute predictions.

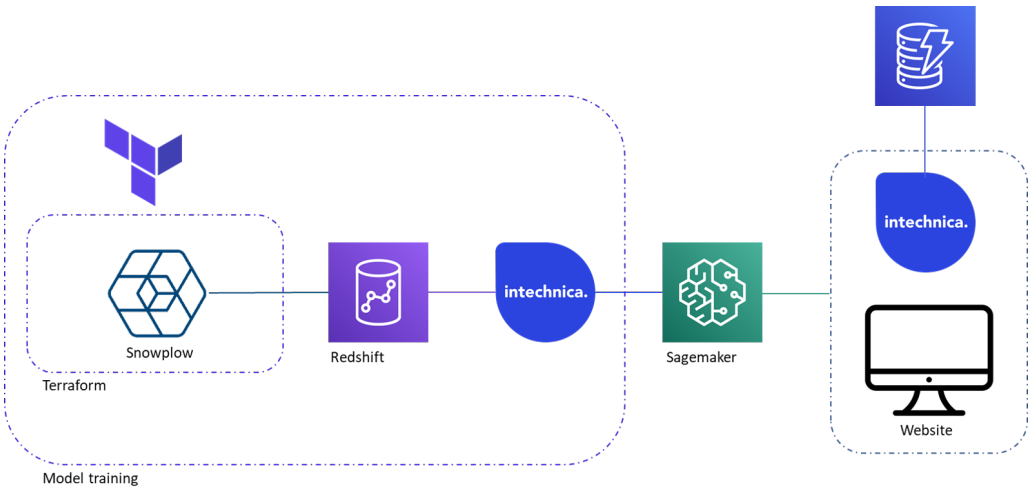
Applications of Dynamic Recommendations

The two applications of this model that we currently cater for are on site recommendations, in the form of a conventional recommendations banner, and email recommendations for personalised email campaigns. For both applications we required a method to store and manage the history vectors associated with each customer session and the summary history vectors that are used as the first row of each example. To manage these vectors we use two databases, one that stores the history vectors associated with each individual customer session, and another that stores the summary history vectors for customers with multiple historical sessions. These vectors could be associated with individual customer accounts or, if this is not possible, a user ID generated from the cookie. Every 24 hours we update these databases for all users with sessions on that day.

For personalised email campaigns, when the application is triggered, we process all sessions within the relevant period of time; these processed sessions, combined with any history vector, are then passed through the model. The resulting recommendations can then be included in an email to each required customer.

For on-site recommendations, each customer's activity is captured and temporarily stored during their session and then, when required, transformed into the vectorised format accepted by the model. We combine the vectorised session with a summary history vector, if it exists, then pass the results through the model to return recommendations. The next section on implementation gives an overview of how these applications work in practice.

Implementation



The diagram above gives a top-level overview of the main elements used in our dynamic recommendations architecture. The model training implementation on the left side of the diagram relies on the same infrastructure as described in the implementation subsection of *Static Recommendations For Product Pages*. A Terraform script

is used to deploy Snowplow[13] onto the desired website. The website browsing data is then stored in a Redshift database and the model preprocessing and training is done using this. The resulting model is hosted using Sagemaker. With the help of our collection and transformation process, the website that displays the recommendations can make calls to the hosted Sagemaker endpoint when recommendations need to be displayed. For email campaigns, we simply process the data in Redshift, pass the results through the model and make the recommendations available to download via our portal or integrate directly with an existing CRM system.

Conclusion and Discussion

We have presented a selection of collaborative filtering algorithms that can be used to produce recommendations for a variety of different applications. We demonstrated how variations of standard product co-occurrence can be used to produce recommendation systems with a higher coverage, increasing the numbers of recommendations visible to customers. The section *Dynamic Recommendations* described a deep learning approach to recommendations that can be used to produce both personalised real-time recommendations for websites and recommendations for email campaigns. In addition to the mathematical details, we described an overview of the AWS architecture that we use to host our applications.

Although the focus of the white paper has been collaborative filtering algorithms, Intechnica have also developed several content-based recommendations systems. One of the drawbacks of using collaborative filtering algorithms is their inability to deal with the cold start problem; they struggle to produce recommendations for new customers or new products. Our content-based recommendations, developed using natural language processing, aim to address these drawbacks.

References

1. G Strang. Linear Algebra and its Applications. Thomson Learning, 2006

2. B Smith, G Linden. Two Decades of Recommender Systems at Amazon.com, IEEE Internet Computing, 21(3), 2-18, 2017

3. S. Wu, W. Ren, C. Yu, G. Chen, D. Zhang, and J. Zhu. Personal recommendation using deep recurrent neural networks in NetEase. In Proc. IEEE 32nd Int. Conf. Data Eng. Helsinki, Finland, pp. 1218–1229, May 2016.

4. G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. Internet Computing, 7(1):76–80, 2003.

5. S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. CoRR, abs/1502.03167, 2015.

6. Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. Computer, 42(8): 30–37, 2009.

7. P. Lops, M. de Gemmis, G. Semeraro. Content-based Recommender Systems: State of the Art and Trends. In: Ricci F., Rokach L., Shapira B., Kantor P. (eds) Recommender Systems Handbook. Springer, Boston, MA, 2011

8. J.Bobadilla, F. Ortega, A.Hernando, A.Gutiérrez. Recommender systems survey, Knowledge-Based Systems, 46, 109-132, July 2013.

9. T. Silveira, M. Zhang, X. Lin, et al. International Journal of Machine Learning and Cybernetics. 10(5): 813-831, 2019

10. R. Misra, M. Wan, J. McAuley. Decomposing fit semantics for product size recommendation in metric spaces. RecSys, 2018

11. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering R. He, J. McAuley WWW, 2016

12. Image-based recommendations on styles and substitutes J. McAuley, C. Targett, J. Shi, A. van den Hengel SIGIR, 2015

13. Snowplow Analytics Limited. https://snowplowanalytics.com/, 2019