UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Bachelor Thesis

Entity2Vec

Embeddings in Retail

Finished on:

27 July 2020

Author:
Rico Meinl
3E Transept St
NW1 5EP London
Tel.: (049) 176 43 89 72 15
E-mail: rmeinl97@gmail.com

Advised by:
Prof. Dr. Dennis Säring
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Phone: (041 03) 80 48-43
E-mail: dsg@fh-wedel.de

Contents

List of Figures

Listings

1

Introduction to Recommender Systems

If we look back at our last week, we realize: a Machine Learning (ML) algorithm determined what songs we might like to listen to, what food to order online, what posts we see on our favorite social networks, as well as the next person we may want to connect with, what series or movies we would like to watch, and so on.
Machine Learning already guides many aspects of our life without us even realizing it. There is one type of algorithm that drives all the applications mentioned above: Recommender Systems.

1.1 Business Value

Harvard Business Review (HBR) made a strong statement by calling Recommenders the *"single most important algorithmic distinction between 'born digital' enterprises and legacy companies"* [1]. HBR also described the virtuous business cycle these could generate: the more people use a company's Recommender System, the more valuable they become. The more valuable they become, the more people use them.

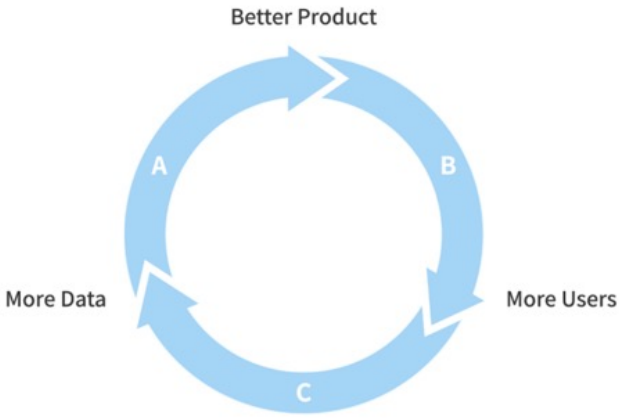

Figure 1.1: The Virtuous Cycle generated by a Recommender System.

We are encouraged to look at Recommender Systems, not as a way to sell more online, but rather see them as a renewable resource to relentlessly improve customer insights and our own insights as well. If we analyze Figure 1.1, we can reason that many legacy companies also have many users and, therefore, many data. Their virtuous cycle has not picked up as much as the ones like Amazon, Netflix or Spotify because of the lack of knowledge on how to convert their user data into actionable insights, which they can then use to improve their product or services.
For example, looking at Netflix shows how crucial this is, as 80% of what people watch comes from recommendations. In 2015, one of their papers [2] quoted: *"We think the combined effect of personalization and recommendations save us more than $1B per year."* Then, if we look at Amazon, 35% [3], of what customers purchase on Amazon.com comes from product recommendations, and at Airbnb, Search Ranking and Similar Listings drive 99% [4] of all booking conversions.

1.2 Problem Formulation

Now that we have seen the immense value, companies can gain from Recommender Systems; we will take a look at the type of challenges that they can solve. Generally speaking, technology companies are trying to recommend the most relevant content to their users. That could mean:

- Similar home listings (Airbnb [4], Zillow [5])
- Relevant media, e.g., photos, videos and stories (Instagram [6])
- Relevant series and movies (Netflix [2], Amazon Prime Video [7])
- Relevant songs and podcasts (Spotify, Pandora [8])
- Relevant videos (YouTube [9])
- Similar users, posts (LinkedIn [10], Twitter [11], Instagram [6])
- Relevant dishes and restaurants (Uber Eats [12])

The formulation of the problem is critical here. Most of the time, companies want to recommend content that users are most likely to enjoy in the future. The reformulation of this problem, as well as the algorithmic changes from recommending 'what users are most likely to watch' to 'what users are most likely to watch *in the future*' allowed Amazon Prime Video to gain a 2x improvement, a 'once-in-a-decade leap' for their movie Recommender System: *"Amazon researchers found that using neural networks to generate movie recommendations worked much better when they sorted the input data chronologically and used it to predict future movie preferences over a short (one- to two-week) period"* [7].

1.3 Data

Recommender Systems usually take two types of data as input:

- User Interaction Data (Implicit/Explicit)
- Item Data (Features)

The 'classic', and still widely used approach to Recommender Systems, based on collaborative filtering (used by Amazon [13], Netflix [2], LinkedIn [10], Spotify and YouTube [9]), uses either User-User or Item-Item relationships to find similar content.
The user interaction data is the data we gather from the weblogs. We can divide it into two groups:

- Explicit data: explicit input from users (e.g., movie ratings, search logs, liked, commented, watched, favorited)
- Implicit data: information that is not provided intentionally but gathered from available data streams (e.g., search history, order history, clicked on, accounts interacted with)

The item data consists mainly of an item's features. In YouTube's case, that would be a video's metadata, such as title and description. For Zillow, this could be a home's zip code, city region, price, or the number of bedrooms, for instance.
Other data sources could be external data (for example, Netflix might add external item data features [14] such as box office performance or critic reviews) or expert-generated data (Pandora's

Music Genome Project [8] uses human input to apply values for each song in each of approximately 400 musical attributes).
A critical insight here is that obviously, having more data about our users will inevitably lead to better model results (if applied correctly). However, as Airbnb shows in their 3-part journey to building a Ranking Model for Airbnb Experiences, one can already achieve a lot with lesser data: the team at Airbnb initially improved bookings by +13% with just 500 experiences and 50k training data size. The main takeaway is: *"Do not wait until you have big data, you can do quite a bit with small data to help grow and improve your business"* [15].

1.4 Algorithms

Often, we limit Recommender Systems to collaborative filtering. In the past, this has been the go-to method for many companies that have deployed successful systems in practice. Amazon was probably the first company to leverage item-to-item collaborative filtering [13]. When they first released their method's inner workings in a paper in 2003, the system had already been in use for six years.
In 2006, Netflix followed suit with its famous Netflix Price Challenge, which offered $1 million to whoever improved the accuracy of their existing system called Cinematch by 10%. Collaborative filtering was also a part of the early Recommender Systems at Spotify and YouTube [16]. LinkedIn even developed a horizontal collaborative filtering infrastructure, known as Browsemaps [10]. This platform enables rapid development, deployment, and computation of collaborative filtering recommendations for almost any use case on LinkedIn.
Let us take a step back and generalize the concept of a Recommender System. While many companies used to rely on collaborative filtering, today, there are many other different algorithms at play that complemented or even replaced the collaborative filtering approach. Netflix went through this change when they shifted from a DVD shipping to a streaming business. As described in one of their papers [2]: *"We indeed relied on such an algorithm heavily when our main business was shipping DVDs by mail, partly because in that context, a star rating was the main feedback that we received that a member had actually watched the video.* [...] *But the days when stars and DVDs were the focus of recommendations at Netflix have long passed.* [...] *Now, our recommender system consists of a variety of algorithms that collectively define the Netflix experience, most of which come together on the Netflix homepage."*

If we zoom out and look at Recommender Systems more broadly we find that they essentially consist of two parts:

1. Candidate Generation
2. Ranking

We will use YouTube's Recommender System [9] as an example in Figure 1.2. That very same concept is applied by Instagram for Recommendations in 'Instagram Explore' [6], by Uber Eats in their Dish and Restaurant Recommender System [17], by Netflix for their movie recommendations [14] and probably many other companies.

According to Netflix, the goal of a Recommender System is to present several attractive items for a person to choose from, which is usually accomplished by selecting some items (Candidate Generation) and sorting them (Ranking) in the order of expected enjoyment or utility.

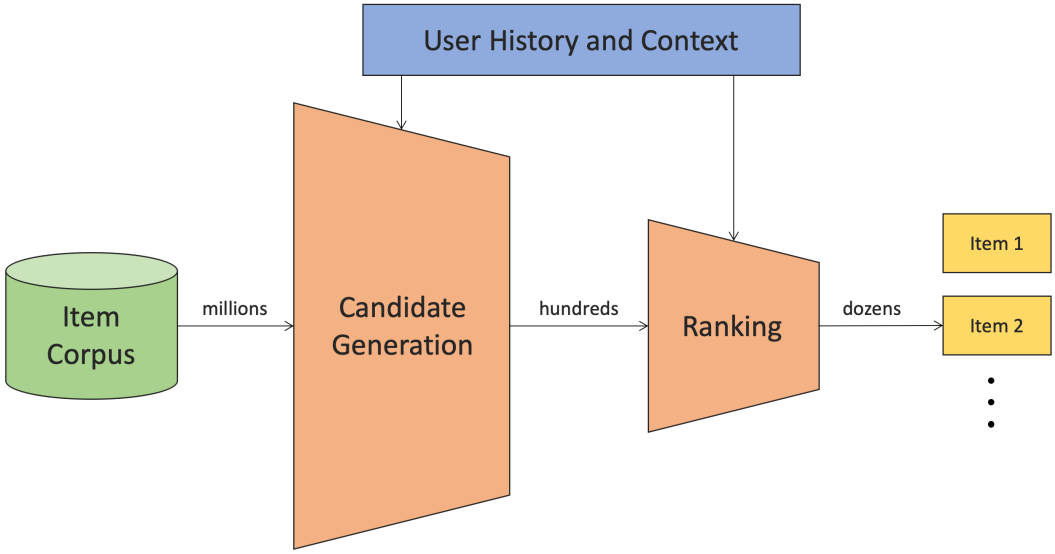

Figure 1.2: Two-Stage Recommender Architecture

1.4.1 Candidate Generation

At this stage, we want to source the relevant candidates that could be eligible to show to our users. Here, we are working with the whole catalog of items that can be large (YouTube and Instagram are excellent examples). The key to do this is entity embeddings, which is the main topic of this thesis. We are going a lot deeper into this in the next sections but to start broadly, one can define them like this: an entity embedding is a mathematical vector representation of an entity such that its dimensions might represent specific properties.
Twitter has a great example of an entity embedding in a blog post about Embeddings@Twitter [11]: say we have two NBA players (Stephen Curry and LeBron James) and two musicians (Kendrick Lamar and Bruno Mars). We expect the distance between the NBA players' embeddings to be smaller than the distance between the embeddings of a player and a musician. We can calculate the distance between two embeddings using the formula for Euclidean distance. How do we come up with these embeddings? One way to do this would be collaborative filtering. We have items and users. In Figure 1.3 we can see an example of the resulting user-item matrix (for the example of Spotify).

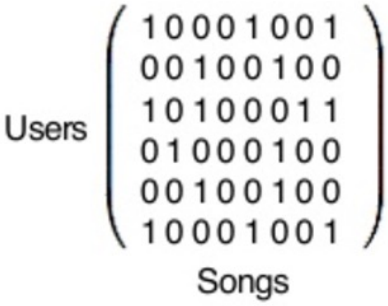

Figure 1.3: User-Item Matrix used in Collaborative Filtering

After applying a matrix factorization algorithm, we end up with user vectors and song vectors. Collaborative filtering, to determine which users' tastes are most similar to one another, compares one user's vector with all of the other users' vectors, ultimately spitting out which users are the closest matches. The same goes songs: one can compare a single song's vector with all the others, and find out which songs are most similar to the one in question.
Another way to do this takes inspiration from applications in the domain of Natural Language Processing (NLP). Researchers generalized the Word2Vec [18] algorithm, developed by Google in the early 2010s to all entities appearing in a similar context [19]. In Word2Vec, a shallow network is trained by directly considering the word order and their co-occurrence, based on the assumption that words frequently appearing together in the sentences also share more statistical dependence. As Airbnb describes, in their blog post about creating Listing Embeddings [4]: *"More recently, the concept of embeddings has been extended beyond word representations to other applications outside of the NLP domain. Researchers from the Web Search, E-commerce, and Marketplace domains have realized that just like one can train word embeddings by treating a sequence of words in a sentence as context, the same can be done for training embeddings of user actions by treating sequence of user actions as context. Examples include learning representations of items that were clicked or purchased [20] or queries and ads that were clicked [21]. These embeddings have subsequently been leveraged for a variety of recommendations on the Web."*
Apart from Airbnb, this concept is used by Instagram (IG2Vec) to learn account embeddings [6], by YouTube to learn video embeddings [9] and by Zillow [5] to learn categorical embeddings. Another, more novel approach to this is called Graph Learning, and Uber Eats uses it for their dish and restaurant embeddings [17]. They represent each of their dishes and restaurant in a separate graph and apply the GraphSAGE [22] algorithm to obtain the representations (embeddings) of the respective nodes.
Finally, we can learn an embedding as part of the neural network for our target task. This approach generates an embedding well customized for a particular system but may take longer than training the embedding separately. The Keras Embedding Layer would be one way to achieve this. Once we have this vectorial representation of our items, we can use Nearest Neighbor Search to find our potential candidates. Instagram [6], for example, defines a couple of seed accounts (accounts that people have interacted with in the past) and uses their IG2Vec account embeddings to find similar accounts that are like those. Based on these accounts, they can find the media that these accounts posted or engaged with. By doing so, they can filter billions of media items down to a couple thousand, sample 500 candidates from the pool, and send them downstream to the ranking stage. This phase can also be guided by business rules or just user input (the more information we have, the more specific we can be). As Uber Eats refers to in one of their blog posts [17], for instance, pre-filtering can be based on factors such as geographical location.

To summarize: in the candidate generation (or sourcing) phase, we filter our whole content catalog for a smaller subset of items that our users might be interested in. To do this, we need to map our items into a mathematical representation called embeddings to use a similarity function to find the most similar items in space. There are several ways to achieve this: Collaborative Filtering, Word2Vec for Entities, and Graph Learning.

1.4.2 Ranking

Let us loop back to the case of Instagram. After the Candidate Generation stage, we have about 500 media items potentially relevant to show to a user in their "Explore" feed.
But, which ones are going to be the most relevant? After all, there are only 25 spots on the first page of the "Explore" section, and if the first items are irrelevant, the user will not be impressed nor intrigued to keep browsing. Netflix's and Amazon PrimeVideo's web interfaces show only the top

six recommendations on the first page associated with each title in their catalog [7]. Spotify's Discover Weekly Playlist contains only 30 songs. All of this is subject to the users' device, with smartphones allowing for less space for relevant recommendations than a web browser.

"There are many ways to construct a ranking function ranging from simple scoring methods to pairwise preferences, to optimization over the entire ranking. If we were to formulate this as a Machine Learning problem, we could select positive and negative examples from our historical data and let a Machine Learning algorithm learn the weights that optimize our goal. This family of Machine Learning problems is known as 'Learning to rank' and is central to application scenarios such as search engines or ad targeting. In the ranking stage, we are not aiming for our items to have a global notion of relevance, but rather look for ways of optimizing a personalized model"
- Extract from Netflix Blog Post [14].

To accomplish this, Instagram uses a three-stage ranking infrastructure [6] to help balance the tradeoffs between ranking relevance and computation efficiency. In the case of Uber Eats, their personalized ranking system is *'a fully-fledged ML model that ranks the pre-filtered dish and restaurant candidates based on additional contextual information, such as the day, time, and current location of the user when they open the Uber Eats app'* [17].
In general, the complexity of a model depends on the size of the feature space. One can use many supervised classification methods for ranking. Typical choices include Logistic Regression, Support Vector Machines, Neural Networks, or Decision Tree-based methods such as Gradient Boosted Decision Trees (GBDT). On the other hand, many algorithms specifically designed for learning to rank have appeared in recent years, such as RankSVM or RankBoost.

To summarize: after selecting initial candidates for our recommendations, we need to design a ranking function that ranks items by their relevance in the ranking stage. One can formulate this as a Machine Learning problem, and the goal is to optimize a personalized model for each user. This step is essential because we have limited space to recommend items in most interfaces. Therefore, we need to make the best use of that space by putting the most relevant items at the top.

1.5 Baseline

As for every Machine Learning algorithm, we need a good baseline to measure the improvement of any changes made. A good baseline to start with is to use the most popular items in the catalog, as described by Amazon in [7]: *"In the recommendations world, there is a cardinal rule. If I know nothing about you, then the best things to recommend to you are the world's most popular things".*
However, if one does not even know what is most popular, because a set of products or new items was just launched — as was the case with Airbnb Experiences [15] — one can randomly re-rank the item collection daily until enough data for the first model has been gathered.

1.6 Evaluation Metrics

Once an algorithm for a Recommender System is selected, we need to find a way to evaluate its performance. As with every Machine Learning model, there are two types of evaluation:

1. Offline Evaluation
2. Online Evaluation

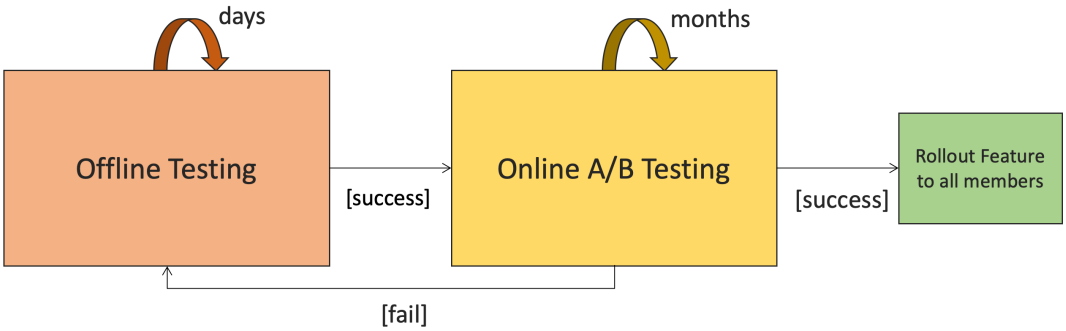

Figure 1.4: Offline/Online Testing Framework

Generally speaking, we can consider the Offline Evaluation metrics as low-level metrics, that are usually easily measurable. The most well-known example would be Netflix choosing to use root mean squared error (RMSE) as a proxy metric for their Netflix Prize Challenge. The Online Evaluation metrics are the high-level business metrics that are only measurable as soon as we ship our model into the real world and test it with real users. Some examples include customer retention, click-through rate, or user engagement.

1.6.1 Offline Evaluation

As most of the existing Recommender Systems consist of two stages (candidate generation and ranking), we need to pick each stage's right metrics. For the candidate generation stage, YouTube, for instance, focuses on high precision [9] so 'out of all the videos that were pre-selected how many are relevant'. This makes sense, given that in the first stage, we want to filter for a smaller set of videos while ensuring all of them are potentially relevant to the user. In the second stage, presenting a few best recommendations in a list, it requires a fine-level representation to distinguish relative importance among candidates with high recall ('how many of the relevant videos did we find'). Often, most cases use the standard evaluation metrics used in the Machine Learning community: from ranking measures, such as Normalized Discounted Cumulative Gain (NDCG), Mean Reciprocal Rank, or Fraction of Concordant pairs, to classification metrics including Accuracy, Precision, Recall, or F1-Score.
Instagram formulated the optimization function of their final pass model a little different [6]: they predict individual actions that people take on each piece of media, whether they are positive actions such as 'like' and 'save' or negative actions such as 'See Fewer Posts Like This' (SFPLT). They use a multi-task multi-label (MTML) neural network to predict these events. As appealing as offline experiments are, they have a significant drawback: they assume that members would have behaved in the same way, like playing the same videos if the new algorithm being evaluated had been used to generate the recommendations. We need an online evaluation that measures the actual impact our model has on the higher-level business metrics.

1.6.2 Online Evaluation

Most companies that launched successful Recommender Systems have also developed a rigorous A/B testing approach. One slight variation is Netflix's approach called 'Consumer Data Science' [23]. The most popular high-level metrics that companies are measuring here are click-through rate

and engagement. Uber Eats goes further here and designs a multi-objective tradeoff that captures multiple high-level metrics [12] to account for the overall health of their three-sided marketplace (among others: Marketplace Fairness, Gross Bookings, Reliability, Eater Happiness). In addition to medium-term engagement, Netflix focuses on member retention rates as their online tests can range from between 2–6 months [2]. YouTube famously prioritizes watch-time over click-through rate. They even wrote an article [24], explaining why: ranking by click-through rate often promotes deceptive videos that the user does not complete ('clickbait'), whereas watch time better captures engagement.

1.6.3 Evaluating Embeddings

As covered in the algorithms' Section 1.4, embeddings are a crucial part of the candidate generation stage. However, unlike with a classification or a regression model, it is not easy to measure the quality of an embedding [11] given that they are often used in different contexts. A sanity check we can perform is to map the high-dimensional embedding vector into a lower-dimensional representation (via PCA, t-SNE, or UMAP) or apply clustering techniques such as k-means and then visualize the results. Airbnb did this with their listing embeddings [4] to confirm that listings from similar locations are clustered together.

2 Foundation

This chapter gives a foundational overview of product embeddings and the theory behind the main methods that we will use for the rest of this thesis.

2.1 Problem Characteristics

As we have seen in Section 1.4.1, generating embeddings for our target items is a crucial step of the first stage (Candidate Generation) in most Recommender Systems. The rest of this thesis's focus is, therefore, on the evaluation of different methods to generate such embeddings of items and evaluate how well they can be used as a Recommender System for items in the Instacart transactions data set [25]. The following chapters are structured as follows: this chapter will lay the foundations for embeddings by introducing the theoretical foundation for the methods we will use. In chapter 3, we will give an overview of the methods, dataset, and the evaluations we will use. After that, in chapter 4, we will present and interpret the achieved results and, finally, in chapter 5, give an outlook on what can be improved in future work.

To reiterate, Recommender Systems use similarities between items and items or items and users in an embedding space to select the most relevant items for a given query. One can learn an n-dimensional embedding vector from data within this embedding space for each user and each item. The higher the quality of these embeddings, the better the recommendations.
As described in Section 1.4.1, there are multiple ways of generating these item embeddings, and we chose to focus specifically on the Word2Vec algorithm adapted to items, also called Item2Vec, with several of its iterations.
This algorithm makes recommendations mostly based on item-based similarities, which differentiates it from the traditional Collaborative Filtering, or more generally user-item recommendations, as it is often used when making recommendations based off a specific user interest in a specific item (e.g., by visiting its product page) or in the context of an explicit user intent to purchase (e.g., when entering the checkout process). Consequently, these types of algorithms can generate higher click-through rates and are responsible for higher revenue. Applying item-item recommendations makes sense when there are significantly more users than items, or the user data is not available, as is the case with anonymous checkouts.
In general, there are two types of recommendations:

- Personalized: 'Products you may like'
- Non-personalized: 'Similar Products', 'Complementary Products', 'Popular Products'

Similar Products are typically found on a product web page, and are usually substitutes, as they can be bought interchangeably. Complementary Products can be purchased in addition to each other when a user already has an intention to purchase. As we can see, the Item2Vec method's qualities make it a strong contender for the non-personalized recommendations category, which is what we will focus on over the next sections. As many papers already focus on the 'most similar'

product recommendations, we are interested in exploring the method in the context of finding the 'most complementary' products. This is an essential problem in e-commerce as it is:

- Reminding the customer about other relevant complementary items to purchase
- Enabling catalog product discovery
- Encouraging additional purchases and basket expansion
- Increasing the average order price and the number of products in baskets

The complementary relationship of a set of products has some interesting properties [26]:

- Asymmetric = HDMI cable –> TV, but TV -|-> HDMI cable
- Non-transitive = HDMI cable –> TV and Cable Adaptors –> HDMI cable, but Cable Adaptors -|-> TV
- Transductive = HDMI cables are also likely to complement other TVs with similar model and brand
- Higher-order = Complementary products for meaningful product combos, such as (TV, HDMI cable), are often different from their complements

It is also important to note here that our algorithm does not consider explicit user feedback (e.g., a star rating) for an item but works solely with implicit user feedback, making it more challenging. With explicit data, we truly know how a user feels towards a product. If that data is not available, we have to fall back to using implicit feedback data such as views, clicks, or purchases.

2.2 Method Overview

Now that we have introduced the importance of embeddings in recommender systems, we will describe different methods to generate these embeddings. Fundamentally they are all based on one algorithm from the NLP domain: Word2Vec [18][27]. Word2Vec is a proven algorithm to learn word vector representations that are good at predicting nearby words. It was published in 2010 as a continuation of methods based on the Distributional Hypothesis, which states that words that appear in a nearby context have similar if not the same semantic meaning. We will first give an introduction into Word2Vec, as well as one of its predecessors, Doc2Vec, in the context of words. Then we will switch to the product context and explain the foundations of the other methods, namely Item2Vec [19], Prod2Vec [20], and Meta-Prod2Vec [28], as well as how we are going to use them for our implementation.

2.2.1 Word2Vec

Before Word2Vec, the common way of coming up with a mathematical representation for words or sentences was through count vectors, also known as bag-of-words vectors. A count vector was essentially a vector with the length of all the existing words, which had 1's at certain indices to signalize a particular word's existence.
The problem with count vectors was that they were highly sparse and memory inefficient. They were also unable to encode any semantic meaning as they got rid of any word ordering within a sentence. The goal of Word2Vec, a method developed at Google, was to learn high-quality word vectors in an unsupervised fashion on a massive dataset with billions of words. It was based on the theory that distributed representations of words can help Learning algorithms achieve better

performance in NLP tasks. The original paper [18] proposes two training algorithms: Continuous Bag of Words and Continuous Skip Gram.

Continuous Bag of Words Model

As we can see in Figure 2.1 below, the projection layer is shared for all words, meaning all words get projected into the same position and then aggregated by either taking their average or sum. This means that the order of words does not influence the projection, which is why the algorithm is called Continuous Bag of Words. During training, we specify a window size w, and for each word in a sentence, we take w words from the left and w words from the right, aggregate their representation and try to correctly predict the missing (query) word based on the given context. The algorithm goes as following:
For each word in the sentence: we want to maximize
$P(avg(w_{t-window}, .., w_{t+window}), given\_word)$.

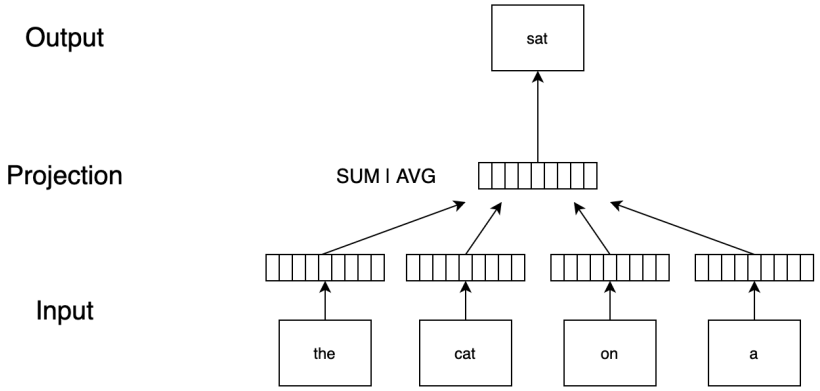

Figure 2.1: Word2Vec CBOW Algorithm

Continuous Skip Gram Model

In the skip gram model, the goal is to train the model to maximize the probability of a word appearing based on another word in the same sentence. We use the current word as input and predict the surrounding words within a given window size, also known as context. Using a bigger context increases the quality of the vectors but also the computational complexity. The authors found that more distant words are less relevant, so they assign less weight to those words by sampling them less frequently. The algorithm iterates through all the sentences in a given dataset and, for each word in the sentence, it tries to predict all its surrounding words:
for each word in the sentence, we want to maximize $P(given\_word, w_{t-window}, .., w_{t+window})$.

Given the sequence of training words $w1, w2, ...$ the objective is to maximize the average log probability (here, c is the window size of the training context).

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t) \tag{2.1}$$

It defines $P(w_{t+j}|w_t)$ using the function shown in Equation 2.2 (vw and $v'w$ are the "input" and "output" vector representations of w and W is the number of words in the vocabulary).

$$p(w_O, w_I) = \frac{\exp({v'_{wO}}^\top v_{wI})}{\sum_{w=1}^{W} \exp(v_w^\top v_{wI})} \tag{2.2}$$

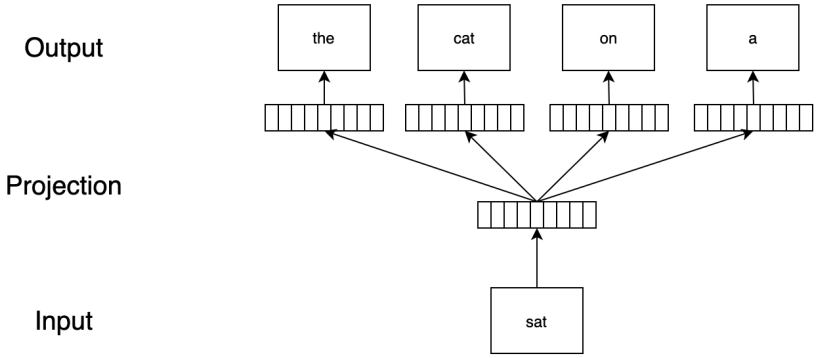

Figure 2.2: Word2Vec Skip-Gram Algorithm

Speeding up the Implementation

To speed up the implementation, the authors implemented a hierarchical softmax function, which is a lot faster as it only calculates the loss for about $log_2(W)$ nodes. It uses a binary Huffman tree that assigns short codes to the most frequent words, resulting in faster training. Another novel improvement, which made the implementation even faster, was achieved through the implementation of Negative Sampling. Here, the model's goal changes to distinguish the target word w_O from randomly drawn words of the noise distribution $P_n(w)$ using logistic regression, where there are k negative samples for each data sample. When using Negative Sampling, we get an additional parameter k, which determines the number of drawn negative samples from our distribution and a Negative Sampling Exponent, which determines the power to which we raise the Unigram Distribution used as Noise Distribution. The more negative samples we provide, the better, but on large datasets, the authors found a small amount (2-5) of negative samples to work well. Another parameter to be aware of is alpha which determines the subsampling of our most frequent words: each word is discarded with a probability according to the following formula ($f(w_i)$ is the word frequency, and t is a chosen threshold):

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{2.3}$$

Within the NLP domain, Word2Vec represented one of the significant breakthroughs in the quality of its results and its scalability. One of the exciting outcomes the teams showed was that they could ask the model questions about word relations in the form of 'big is to biggest like small is to what?' Then they did the vector calculation of $vector(biggest) - vector(big) + vector(small)$ and searched for the closest vector in the space using the cosine distance. They found that indeed the closest result was $vector(smallest)$. Surprisingly, the learned vectors encode many linguistic regularities and patterns. The authors showed many more examples, as seen in the Table 2.3.

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Figure 2.3: Word2Vec Relations

2.2.2 Doc2Vec

In this section, we will introduce the Doc2Vec or Paragraph2Vec algorithm [29], which iterates on Word2Vec by adding the concept of a label or ID for a paragraph, a document, or a sentence. The resulting paragraph vectors allow for easy comparison between two paragraphs or allow us to classify a piece of text to a preexisting label based on its content. The algorithm can learn fixed-length feature representations from variable-length pieces of text such as documents, paragraphs, or sentences. In the past, researchers used two ways to represent a sentence or paragraph, each of which came with their limitations:

- Bag-of-words
- Bag-of-n-grams

The Bag-of-words representation is essentially a count vector with the size of the number of words in a given vocabulary in which the number signifies the appearance of a word in a document at its index. The limitation of this method is that we lose the word ordering and ignore the sentences' semantics in the document. The vectors often suffer from high dimensionality and data sparsity. We face similar problems with the Bag-of-n-grams representation in that we ignore semantics of the broader sentence and suffer from data sparsity and high dimensionality, which requires increased memory and computing power. Researchers tried to go beyond word-level representations to sentence or paragraph level by taking a weighted average of all the word vectors in a given sentence, ignoring its word order and, therefore, any underlying semantics.

Distributed Memory Model

The Doc2Vec algorithm was born from the Word2Vec algorithm's success and built on top of it: as in Word2Vec, it maps every word to a vector in matrix W. Every paragraph gets a label assigned and is also mapped to a vector in matrix D.
Then the paragraph vector, which can be thought of as just another word and word vectors are concatenated or averaged to predict the next word in a context. The algorithm consists of two key stages:

1. Training Stage: generate the word vectors in W, the Softmax weights U, b and the paragraph vectors in D on already seen paragraphs

2. Inference Stage: calculate the paragraph vectors in D for new paragraphs by adding more columns in D and gradient descending on D while holding W, u and b fixed

One of the algorithms' main benefits is that it can learn from unlabeled data and therefore work well for tasks that do not have many labeled data to start with. The model we just described is also known as the Distributed Memory (DM) Model, and it is similar to the Distributed bag-of-words model in Word2Vec, which we introduced in Section 2.2.1.

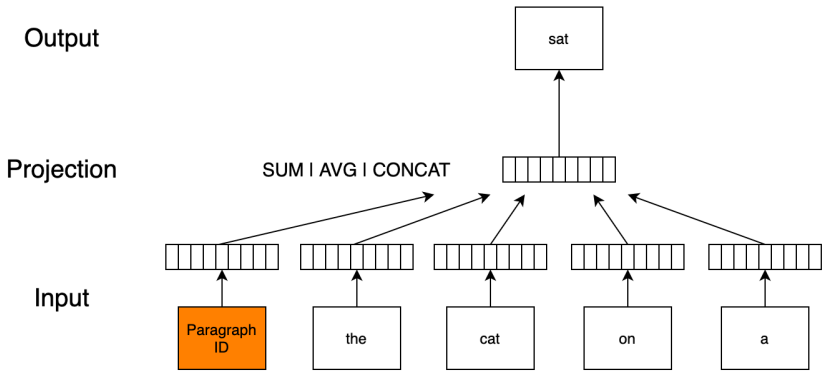

Figure 2.4: Doc2Vec DM Algorithm

Distributed Bag of Words Model

The authors [29] also present a second algorithm, which they call the Distributed Bag of Words (DBOW) Model. It works similarly to the Continuous Skip Gram Model we introduced in the Word2Vec section. It does not consider the word ordering, and mostly ignores the words in the input, but tries to predict words randomly sampled from the paragraph in the output using the paragraph vector. The authors found that using a combination of the PV-DM and PV-DBOW models is usually more consistent across many tasks.

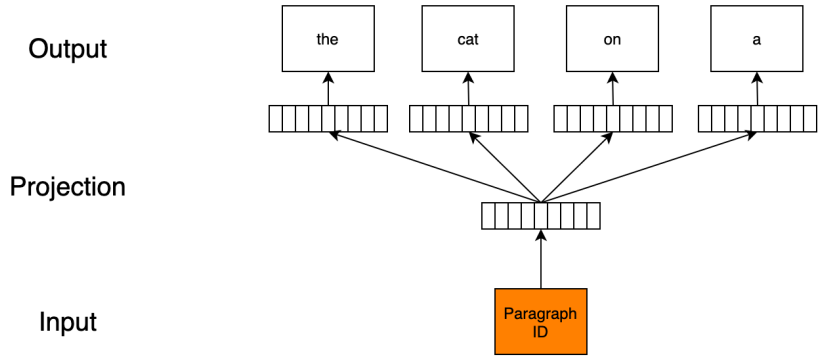

Figure 2.5: Doc2Vec DBOW Algorithm

2.2.3 Item2Vec

In the previous sections, we introduced two landmark papers in the NLP domain, Word2Vec, and Doc2Vec. These algorithms are used to generate high-quality word and paragraph embeddings. In the following sections, we will switch our focus from words to items, or, more generally, entities. After the successful applications to words, researchers were intrigued to apply the highly effective implementation provided by the team at Google to other domains.
One of the first papers exploring this space was Item2Vec [19], which is essentially applying the idea and algorithm behind Word2Vec to products (or items) instead of words. The only change they made is to let the window size be determined from the set's full size, which slightly modifies the objective for a given set of items.
In the introductory chapter 1, we have seen that a widespread use case in E-Commerce is to find the 'most similar' or 'most complementary' items for a given query item. Word2Vec is based on the Distributional Hypothesis, which states that words that appear in a similar context have similar if not the same meaning, making it an interesting method for such E-Commerce applications.

Motivation

In this paper, the authors work with the example of recommending music on the XBox360 store and apps in the Microsoft Play Store. One is interested in finding similar items based on a single query item (e.g., the last song played). Item-based similarities are one way to achieve that and are therefore often used by online retailers for recommendations based on a single item. They are different from traditional user-item recommendations because they are shown in the context of specific user interest in a specific item and in the context of an explicit user intent to purchase. Other potential use cases are:

- Candy-Rank recommendations: similar items (usually of lower price) shown on the checkout page right before the payment
- Bundle recommendations: similar items grouped and recommended together
- Enabling better exploration, discovery and to improve the overall user experience

Application

In traditional Collaborative Filtering, we analyze user-item or item-item relations to produce embeddings in the latent space that can be used to compare item similarities. With Word2Vec, one can learn these latent representations of words using highly scalable neural embedding algorithms. The authors described Item2Vec as an item-based Collaborative Filtering algorithm that produces embeddings for items in a latent space. They directly apply the Word2Vec Skip Gram framework with Negative Sampling on sets of items, to represent associations.
In contrast to traditional Collaborative Filtering, the authors discard the user information and solely focus on the item sequences. A sequence of words used to train embeddings in Word2Vec is equivalent to a set of a basket of items, and the authors are using a window size based on the set size: each item that shares the same set is treated as a positive item. By moving from a sequence to a set, the items' spatial and time information is lost.
In a sentence, we can usually reason that the words that appear closer together are most relevant while in a shoppers' basket, all product combinations can potentially be relevant. The rest of the process remains identical to the training methodology described in Word2Vec.

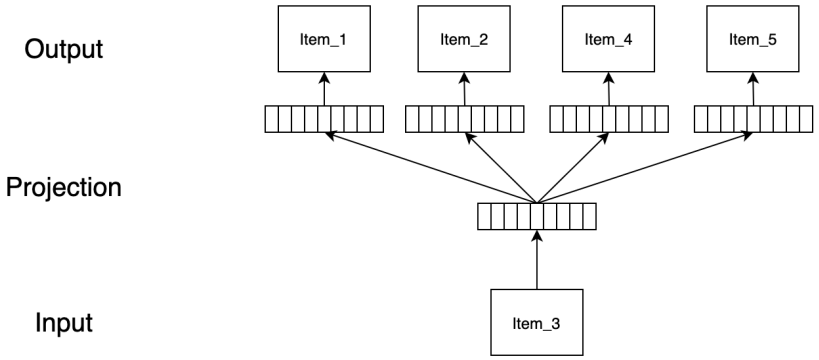

Figure 2.6: Item2Vec Skip-Gram Algorithm

2.2.4 Prod2Vec

Another landmark paper, which was released shortly after Item2Vec, is Prod2Vec [20], which also contains the idea of User2Vec. The authors apply the Skip Gram algorithm to recover product co-occurrence information across the same user's transactions. The application domain is product ad recommendations in Yahoo's e-mail interface. Like Item2Vec, the authors propose an approach of learning a representation of products in a low-dimensional space from historical logs using the neural language model described in Word2Vec. After learning the product embeddings, product recommendations can be made through Nearest Neighbor Search in the learned embedding space.

Application

The dataset consists of S e-mail receipt logs from N users. Each users log $s = (e1, e2, \ldots, e_m) \epsilon S$ is an uninterrupted sequence of M e-mail receipts. Each email receipt $e_m = (p_{m1}, p_{m2}, \ldots, p_{mt})$ consists of T_m purchased products. For generating the product embeddings, each purchase sequence is treated as a sentence and the products in that sequence as words. The authors introduce the following novel methods:

Prod2Vec: the standard Prod2Vec algorithm treats a user's full purchase history as a sequence and does not take into account that an e-mail receipt may contain multiple products purchased at the same time.
Bagged-Prod2Vec: the bagged version of Prod2Vec introduces the notion of a shopping bag, and it operates on the level of e-mail receipts rather than at the level of products. It also adds the twist that items of the same e-mail receipt do not predict each other during training, which is how it differentiates from Item2Vec.

Given these two base methods, the authors introduce several ways of generating recommendations:

Product to Product Predictive Models

Prod2Vec-TopK: given the purchased product, the model calculates the cosine similarity to all other products and recommends the top k most similar.

Prod2Vec-Cluster: this method groups similar products into a cluster and, for a given purchased product, first identifies its cluster, then determines the top related clusters and finally selects the top k products based on cosine similarity.

User to Product Predictive Models

User2Vec: in addition to product-to-product predictive models, this paper also introduces a user component. This method is essentially the Doc2Vec method we described in Section 2.2.2, and it simultaneously learns vector representations of products and users by considering the user as a global context. During training, the user vectors are updated to predict the products from their e-mail receipts, while product vectors are updated to predict the other products in their context. The main advantage of this method is that we can generate product recommendations specifically for a given user. The disadvantage of it is that it needs to be updated more frequently, especially in domains where users frequently purchase. The algorithm first calculates the cosine similarity between a given user vector and all product vectors and then retrieves the top k nearest products in space to generate recommendations. The authors also add a time decay function that gives higher weight to products that were recently purchased by multiplying cosine similarity by decay to the power of the current day of purchase.

2.2.5 Meta-Prod2Vec

The last algorithm we are going to introduce to lay the foundation for the application section, later on, is Meta-Prod2Vec [28], which iterates on Prod2Vec by adding categorical side-information during training. The Prod2Vec training algorithm remains unchanged, but the original pairs of items are supplemented with additional pairs that involve metadata.
In their paper, the authors apply the algorithm to the domain of music recommendations, and their main goal is to overcome some of the major constraints of the state-of-the-art methods that use matrix factorization of item-item and user-item matrices to generate embeddings. The major constraints are scaling to large amounts of user information, supporting real-time changes, and handling the cold-start problem.

Motivation

According to the authors, one way the Prod2Vec method could be improved is by leveraging existing metadata. In its basic form, Prod2Vec uses only the local product co-occurrence information established by the product sequences. The main idea behind Meta-Prod2Vec is to create a general approach for adding categorical side information to Prod2Vec to be able to inject item metadata into the model to regularize the item embeddings. This side information is only available during training time because of the memory constraints faced when performing real-time scoring in production.

Prod2Vec Limitations

One of the improvements Prod2Vec offers compared to a Collaborative Filtering approach is that it considers the local co-occurrence information established by product sequences that is much richer than the global co-occurrence information used in Collaborative Filtering. Though, it does not take into account metadata and therefore, does not model the following interactions:

- Given the current visit of the product p with category c, it is more likely that the next visited product p' will belong to the same category c
- Given the current category c, it is more likely that the next category is c or one of the related categories c' (e.g., after a swimwear sale, it is likely to observe a sunscreen sale, which belongs to an entirely different product category)
- Given the current product p, the next category is more likely to be c or a related category c'
- Given the current category c, the more likely current products visited are p or p'

Application

To overcome these limitations, the authors of Meta-Prod2Vec propose the following additions. As covered in the introductory section, it extends the Prod2Vec loss by taking into account the items' metadata:

$$L_{MP2V} = L_{J|I} + \lambda(L_{M|I} + L_{J|M} + L_{M|M} + L_{I|M}) \tag{2.4}$$

- $L_{I|M}$ is the weighted cross-entropy between the observed conditional probability of input product ids given their metadata and the predicted conditional probability
- $L_{J|M}$ is the weighted cross-entropy between the observed conditional probability of surrounding product ids given the input products' metadata and the predicted conditional probability
- $L_{M|I}$ is the weighted cross-entropy between the observed conditional probability of surrounding products' metadata values given input products and the predicted conditional probability
- $L_{M|M}$ is the weighted cross-entropy between the observed conditional probability of surrounding products' metadata values given input products metadata and the predicted conditional probability

The original Prod2Vec algorithm remains unchanged, with the addition that original pairs of items are supplemented with additional pairs that involve metadata.

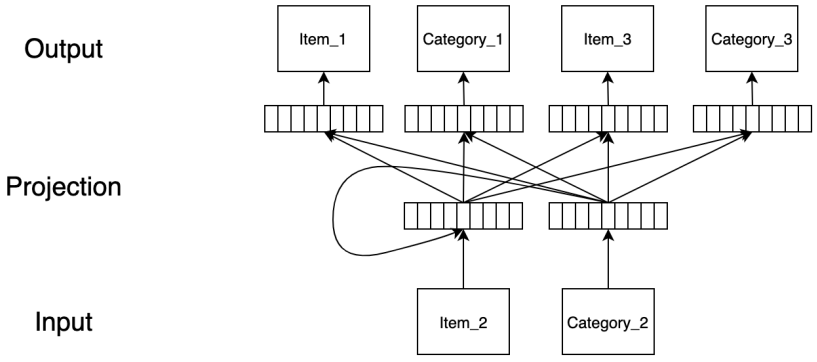

Figure 2.7: Meta-Prod2Vec Skip-Gram Algorithm

3 Methodology

In this chapter, we will cover the problem to solve as well as the methods, dataset, and evaluation techniques we will use.

3.1 Problem Formulation

This thesis aims to evaluate which of the techniques described in the previous section produce the best results for a given dataset. The dataset is from Instacart, an online grocery ordering, and delivery app, and we will apply our learned embeddings to two types of recommendations:

1. Within-Basket Recommendations
2. Next-Basket Recommendations

The Within-Basket Recommendations can be applied to the customer journey steps where the customer just added an item to her basket or during the checkout process. The goal here is to recommend complementary items rather than similar items. The same goes for the Next-Basket Recommendations, which could be applied in different contexts, such as displaying items on the landing page after a returning customer logged in, or for e-mail marketing campaigns.

3.2 Methods

The different algorithms we will evaluate are either directly inspired or taken from one of the algorithms described in Section 2.2. Essentially we will evaluate four variations of the original Item2Vec algorithm, with slight adjustments. We decided to go for Item2Vec rather than Prod2Vec because Item2Vec looks at each basket set individually, while Prod2Vec looks at all transactions a user ever made. Another option would have been the Bagged-Prod2Vec method, which tries to predict products from other baskets of the same user when given a query product. For the Instacart dataset, which consists of shopper IDs and their baskets, we deemed the Item2Vec to be more appropriate. Exploring the Bagged-Prod2Vec or Prod2Vec method could be the subject of future work.
These are the four methods we are going to evaluate:

1. **Item2Vec**: the very same algorithm that was introduced in Section 2.2.3, which uses the Word2Vec Skip Gram with Negative Sampling algorithm to generate item embeddings.
2. **Meta-Item2Vec**: inspired by the ideas from the Meta-Prod2Vec algorithm introduced in Section 2.2.5, we inject item metadata into the model during training.
3. **User-Item2Vec**: inspired by the User2Vec method that was introduced in Section 2.2.4, we add a user label to each basket of items to concurrently learn a vector for the user. During prediction, we add the user vector to get more tailored recommendations.

4. **User-Meta-Item2Vec**: we inject metadata into the User-Item2Vec method to see if we get an additional boost in performance.

To understand how the algorithm makes recommendations, we have to go back to the Word2Vec loss described in Section 2.2.1 and adjust it to our product context. As we are using the skip gram algorithm, the probability of a product appearing within the same basket as another product is defined by:

$$\sum_{w_t \epsilon B} \sum_{-c \leq j \leq c, j \neq 0} \log P(w_{t+j}|w_t) \tag{3.1}$$

In equation 3.2 below, $P(w_{t+j}|w_t)$ is defined by:

$$P(w_O, w_I) = \frac{\exp({v'_{wO}}^\top v_{wI})}{\sum_{w=1}^{W} \exp(v_w^\top v_{wI})} \tag{3.2}$$

In Equation 3.2, vw and $v'w$ are the 'input' and 'output' vector representations of words in the vocabulary. Now, we consider a product p. By definition, complementary products are frequently purchased together and therefore have high conditional probability $P(m|p)$. The simplest way to estimate $P(m|p)$ would be by co-counting, where we simply count the number of times p appeared in the same basket as m. In the given equation 3.2, n_{mp} is the number of times m and p appeared in the same basket and n_p is the number of times p appeared in any basket. For simplicity, we are assuming that each product can only appear once in a basket.

$$P(m|p) \approx \frac{n_{mp}}{n_p} \tag{3.3}$$

With Item2Vec, we can estimate the probability $P(m|p)$ by using the input and output vector representations of our products, vw and $v'w$, and infer complementary products for a given item p by selecting items m with a high score ${v'_m}^\top v_p$ because $P(m|p) \sim \exp({v'_m}^\top v_p)$.

Our recommendations then work like this: each of the four methods implements the same two stage prediction function we have seen in the introductory section:

```
def predict_items(self, user_id, given_items):
    candidate_scores = self.generate_candidates(given_items)
    ranked_candidates = self.rank_candidates(
        user_id, candidate_scores
    )

    return ranked_candidates
```

Listing 3.1: Predict Items Method

We then generate candidates simply by averaging the product vectors of our given items (the users' shopping basket), which consists of input vectors vw and taking its dot product with all items in our trained output vector-matrix which consists of output vectors $v'w$.

```
def generate_candidates(self, given_items):
    candidate_list = []

    # slice the word vectors array to only keep the relevant items
    item_embeddings = self.embedding_vectors[given_items]

    mean_basket_vector = np.mean(item_embeddings, 0)

    # complementary items need to be calculated via dot product not cosine similarity
    distances = np.dot(self.context_vectors, mean_basket_vector)

    return distances
```

Listing 3.2: Generate Candidates Method

For the two methods which also calculate the user vectors, we add a conditional statement into our ranking function which calculates, out of all existing products, the ones closest to a given user and then takes an α-weighted ensemble of these scores with the calculated output from the candidate generation stage. Here, we ran tests for all ten values in the range of 0.0 to 1.0, and we found 0.5 to work best.

```
def rank_candidates(self, user_id, candidate_scores):
    if self.user_vectors:
        user_distances = np.dot(self.embedding_vectors, self.user_vectors[user_id])

        candidate_scores = (candidate_scores * (1 - self.alpha)) + (
            user_distances * (self.alpha)
        )

    return candidate_scores
```

Listing 3.3: Rank Candidates Method

3.3 Dataset

After going over the methods that we will evaluate, we will give an overview of the data set we are planning to use, which will give more exposure to the problem we are trying to solve.

3.3.1 Overview

The dataset is from Instacart, an online grocery ordering and delivery app, aiming to make it easy to fill one's refrigerator and pantry with personal favorites and staples when one needs them. After selecting products through the Instacart app, personal shoppers review the order and do the in-store shopping and delivery. The dataset they open-sourced is anonymized and contains a sample of over 3 million grocery orders from more than 200,000 Instacart users. They provide between 4 and 100 of each user's orders, all containing a sequence of purchased products. The dataset does not contain any timestamps of the order data, but it does contain the week and hour of the day the order was placed and a relative measure of time between orders.
After some initial merging and filtering, we end up with two data frames. The first data frame, in Table 3.1, consists of 3,346,084 orders from 206,209 users, where each user has a unique user ID and a list of orders. Each order comes with a unique order ID and contains a list of products, each represented by a unique product ID, which makes up the users' shopping basket.

Eval Set	User ID	Order ID	Order Number	Product ID
prior	17	1737705	1	[product_47141]
prior	17	1681401	2	[product_42356, product_16797]
prior	17	3197376	4	[product_47141]
prior	17	2616505	6	[product_47141, product_38444]
prior	17	2430354	8	[product_47141, product_9387]
prior	17	2373492	9	[product_49131]
prior	17	805025	11	[product_47141, product_16797]
prior	17	912404	12	[product_47141]
prior	17	603534	14	[product_38444]
prior	17	1719551	16	[product_47141]

Table 3.1: Instacart Transactions Dataframe

The second data frame, in Table 3.2, contains all the products and their metadata. There are 49,686 products in total, and each product has a unique product ID and a product name. It also belongs to one of 21 departments, and one of 134 aisles. Each one has a unique department ID and aisle ID, respectively, and a name identifier.
The data set contains some of the following categories:
Departments: snacks, pantry, beverages, frozen, personal care, dairy eggs, household, babies, meat seafood, dry goods pasta, breakfast, canned goods, produce, missing, international, deli, alcohol, bakery, bulk
Aisles: cookies cakes, energy granola bars, chips pretzels, crackers, popcorn jerky, fruit vegetable snacks, candy chocolate, nuts seeds dried fruit, trail mix snack mix, spices seasonings, marinades meat preparation, doughs gelatins bake mixes, salad dressing toppings, spreads, oil vinegar, preserved dips spreads, honey syrup nectars, baking ingredients, condiments, pickled goods olives, tea, juice nectars.

Product ID	Product Name	Department	Department ID	Aisle	Aisle ID
22362	Original Rice Krispies Treats	snacks	19	cookies cakes	61
40063	Gluten Free Chocolate Chip Cookies	snacks	19	cookies cakes	61
40199	Chocolate Chip Cookies	snacks	19	cookies cakes	61
45374	Newman O's Creme Filled Chocolate Cookies	snacks	19	cookies cakes	61
45866	Fig Newmans Fruit Filled Cookies	snacks	19	cookies cakes	61
5322	Gluten Free Dark Chocolate Chunk Chewy	snacks	19	energy granola bars	3
10753	Peanut Butter Bar	snacks	19	energy granola bars	3
14778	Organic Chocolate Chip Chewy Granola Bars	snacks	19	energy granola bars	3
16254	ZBar Organic Chocolate Brownie Energy Snack	snacks	19	energy granola bars	3
17224	Oats & Honey Gluten Free Granola	snacks	19	energy granola bars	3

Table 3.2: Instacart Products Dataframe

3.3.2 Feature Exploration

In this section, we will explore some features of the data to provide the reader with a better understanding of its structure. First, in Figure 3.1, we can see how many times customers have placed an order.

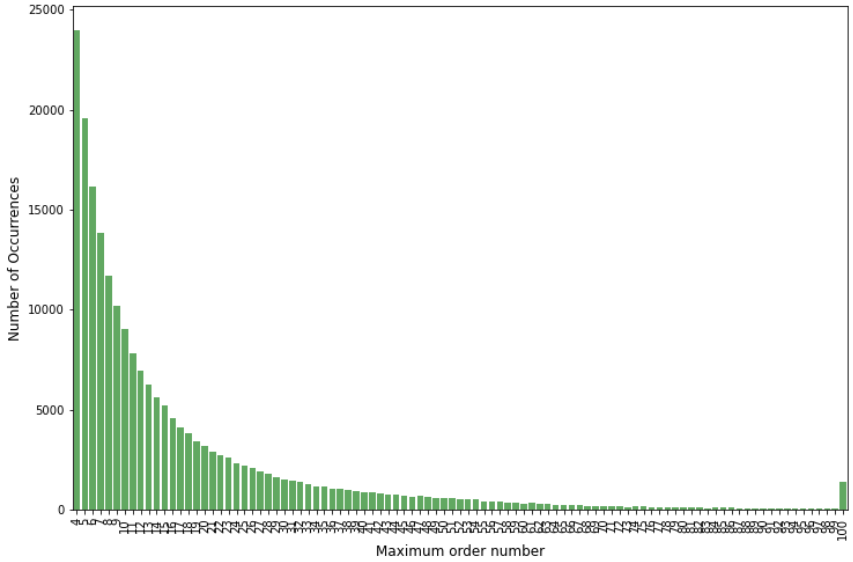

Figure 3.1: Number of Orders per Customer

Then, in Figures 3.2 and 3.3, we can visualize on which day of the week, and which hour of the day, respectively, these orders have occurred.

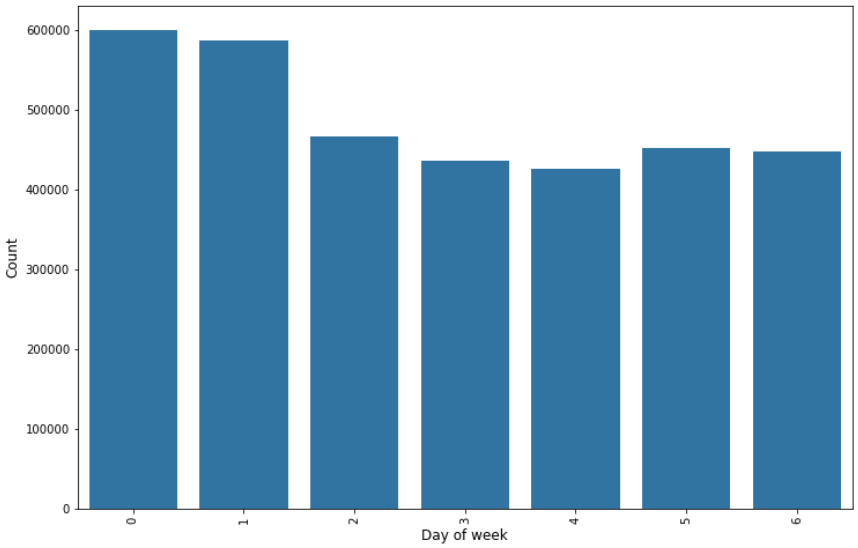

Figure 3.2: Number of Orders per Week Day

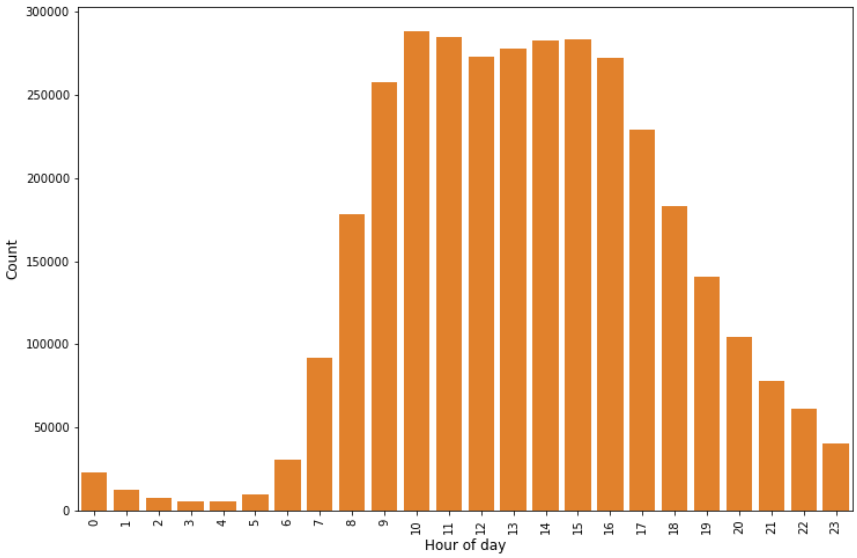

Figure 3.3: Number of Orders per Hour of Day

In our implementation of Item2Vec, we need to set the window size for a given sentence, determining how many context words, or in our case context products, we want to sample for each target word. Figure 3.4 shows how many products a user basket usually contains.

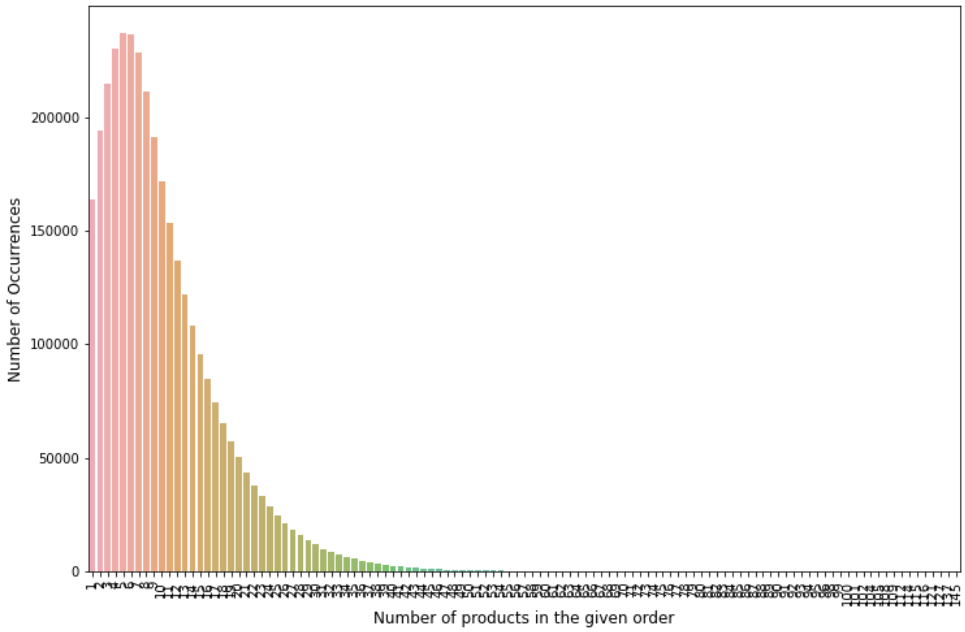

Figure 3.4: Number of Products per Order

In Recommender Systems, the most popular items recommendations are usually a tough baseline to beat. In Table 3.3, we can see which products are most popular in our dataset.

Product Name	Frequency Count
Banana	491291
Bag of Organic Bananas	394930
Organic Strawberries	275577
Organic Baby Spinach	251705
Organic Hass Avocado	220877
Organic Avocado	184224
Large Lemon	160792
Strawberries	149445
Limes	146660
Organic Whole Milk	142813
Organic Raspberries	142603
Organic Yellow Onion	117716
Organic Garlic	113936
Organic Zucchini	109412
Organic Blueberries	105026
Cucumber Kirby	99728
Organic Fuji Apple	92889
Organic Lemon	91251
Organic Grape Tomatoes	88078
Apple Honeycrisp Organic	87272

Table 3.3: Most Popular Products

Finally, we want to explore the distribution of aisles and departments, shown in Figure 3.5 and Figure 3.6, respectively, to understand the skewness of classes.

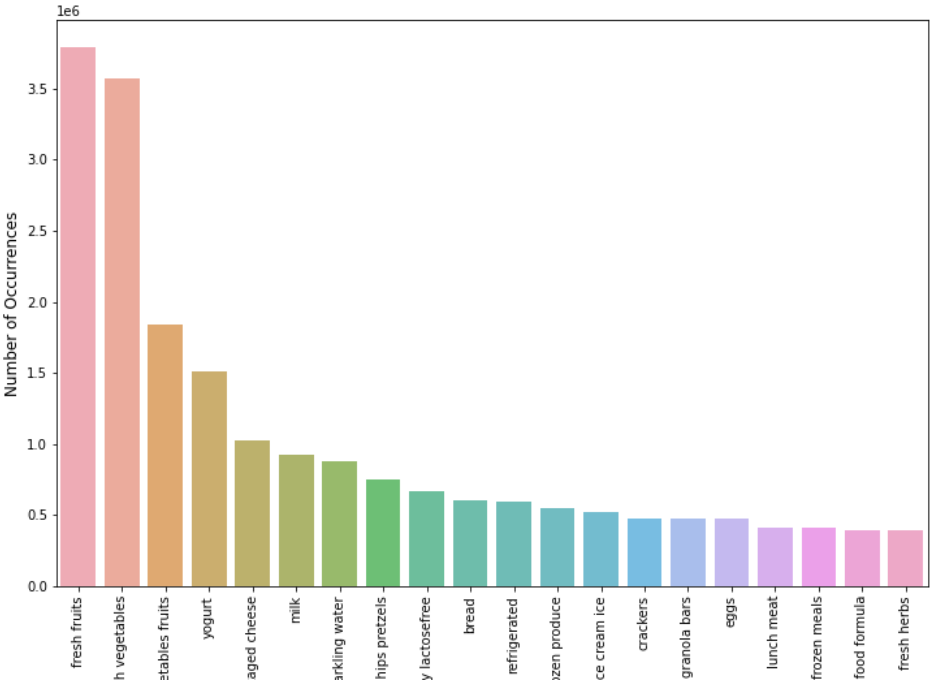

Figure 3.5: Most Popular Aisles

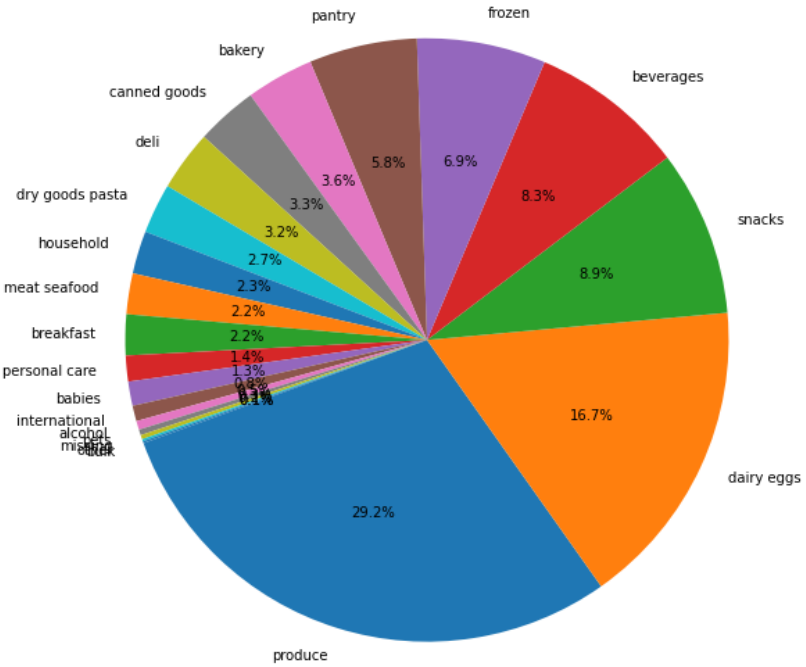

Figure 3.6: Departments Distribution

3.3.3 Data Preparation

We adopt the same data preparation steps as in [30]. First, we remove all items with less than ten purchases. If a user has more than one transaction, we use the last transaction in the testing set, and if a user has more than two transactions, we the second-to-last in the validation set. As we have seen in the previous section, every user has a minimum of four orders, so both our test and validation set contains 206,209 rows, one for each user. Our training set consists of 2,933,665 orders. The input to all our implementations is simply a dataset that contains a list of transactions. Each transaction has a user ID and a list of purchased products, each represented by their product ID. We are going to train embedding vectors for each of these products. We create these lists of product ids in the format of ['user ID', 'product 1', 'product 2', ..., 'product n'] and remove the user ID for the Item2Vec and Meta-Item2Vec model and add it as a label for the models that include the user.

3.4 Evaluation

Now that we understand what the problem is and what the dataset looks like, we will give an overview of how we are going to evaluate the four methods described in Section 3.2. There are two steps in our pipeline that we will perform and that we want to evaluate. Firstly, we want to create vector embeddings for all products (and users) using either one of the four embedding methods. Secondly, we want to use those embeddings to generate product recommendations for the Instacart online grocery dataset. The generated recommendations are the end goal of our offline testing, and the assumption is that better embeddings will lead to better recommendations.

3.4.1 Embedding Evaluation

The embedding evaluation serves us as a proxy that attempts to measure the quality of the embeddings we generate.

Quantitative Evaluation

Every product in the Instacart dataset belongs to a department (higher-level categorization) and an aisle (lower level categorization). Inspired by the Item2Vec paper, we use a KNeighbors classifier, which takes each items' embedding vector as an input and its category or aisle label respectively as a label. After generating our n-dimensional embeddings for every product, we split our dataset equally with a 0.5 split into a training and testing set. Then we fit one K-Neighbors classifier on each the training set for the department and aisle labels and try to predict the target label on the testing set. The K-Neighbors classifier makes predictions by doing a simple majority voting of the category on the k nearest neighbors in the vector space. The parameter k can be changed, but we choose to use 10 after some trials because we found it to be a good tradeoff.
Based on the skewness of the classes that we found in Section 3.3, we choose to report the Micro and Macro level F1 scores. The Macro F1 computes the metrics for each class, treating each one equally and then report the average over all classes, therefore not taking into account class imbalance.
The Micro F1 score aggregates the scores of all classes independently and then reports the result.
As we are using the F1 scores, we can consider the reported values from [30] as a baseline, as the respective authors applied a similar method on the same dataset.

Qualitative Evaluation

For the qualitative evaluation, we stick to a standard method to qualitatively evaluate word embeddings: visualization. We create a visualization via T-SNE, which is a subjective way of analyzing, but it can serve as a valuable indicator of how well the model can separate between the classes in space.

3.4.2 Recommender Evaluation

After applying the dataset split into training, validation, and test set proposed in Section 3.3.3, we use the validation set to tune the hyperparameters of our models and report all results on the test set. As described in Section 3.1, we will evaluate our four methods on the following two tasks:

- **Within-Basket Recommendations**: we take all the data in the training set to generate the embeddings, and we validate and test on the n-1th or n-th transaction respectively. We assume half the products in the shoppers' basket are given and try to predict the other half. To achieve this, we first take the first half of the items in the basket, calculate the mean of their item vectors and look for the most relevant items by taking the dot product of this average basket vector and the models' context vectors, which determines item-item complementarity.
- **Next-Basket Recommendations**: we apply the same procedure as described above, but instead of taking half the shoppers basket, we take all the products a user has ever interacted with as given products and try to predict the complete basket of the validation and test set.

Metrics

To evaluate our recommendations' quality, we use the ROC-AUC score as an overall ranking metric and the Normalized Discounted Cumulative Gain (NDCG), as a ranking-focused metric that scores the items based on how high they were ranked compared to their level of relevance. We also consider Precision and Recall@K scores where we are more focused on Recall@K, or the so-called Hitrate@K. The Hitrate measures, based on K recommended items, how many items appeared in the set of target items. This score is useful because, as we saw in the first section, we only have limited space to show recommended items to our users. Because of this limitation, we chose 10 as our value for K.

3.4.3 Baseline

To ensure that our reported results are relevant, we have selected three baseline algorithms that are often used in Recommender Systems. We have also used the reported results from [30] as our algorithmic baseline, as they evaluated similar methods on the same dataset.

- **Most Popular Items** - simply the items most often purchased (*"In the recommendations world, there is a cardinal rule.* [...] *If I know nothing about you, then the best things to recommend to you are the most popular things in the world."* [7])
- **Most Popular Items for User** - the items most often purchased by a specific user.
- **Co-Counting** - as described in Section 4.1 of [31], Co-Counting is the simplest way to estimate the conditional probability $P(m|k)$ which is used in the Word2Vec algorithm. Hence it serves well as a baseline method.

4 Results

In this chapter, we will show in detail how we applied each one of the methods described in Chapter 3 Section 3.2 and what results we generated.

4.1 Applying each Method

To implement the four embeddings methods, we chose to use the popular gensim framework [32], which is widely used across the industry and academia. We use its implementation of Word2Vec for our Item2Vec and Meta-Item2Vec experiments and its Doc2Vec implementations for the User- and User-Meta-Item2Vec experiments. Then, we implemented a class for the Recommender, which can be found in the Github repository accompanying this thesis [33]. For each model, we first applied the data preparation steps outlined in Section 3.3.3. Then we ran a hyperparameter search to find an optimal set of hyperparameters that we used for each model (Section 4.2).

Item2Vec

As previously stated, we used the standard gensim implementation of Word2Vec Skip Gram with Negative Sampling. The input to the model was the sequences from our training set which were in the form of: ['user ID', 'product 1', 'product 2', ..., 'product n']. We discarded the user ID for this model and only worked with the product sequence.

Meta-Item2Vec

The only change to the Item2Vec implementation was that we also substituted the training sequences with the department and aisle categories, respectively. The input sequences looked like this: ['user ID', 'product 1', 'category 1', 'product 2', 'category 2', ..., 'product n', 'category n']. Again, we discarded the user ID for this model and only worked with the product sequence.

User-Item2Vec

To add the user vector to our model, we switched to gensim's Doc2Vec Distributed Memory with Negative Sampling implementation. The input sequences were the same as in Item2Vec, but we also passed the user ID as an input label to train the user vector-matrix D.

User-Meta-Item2Vec

The implementation was the same as User-Item2Vec with the additional injection of our items' category metadata, as seen in the Meta-Item2Vec section.

4.2 Hyperparameter Optimization

To generate the best results possible, we ran a hyperparameter optimization over about 100 different parameter combinations. As the authors in [34] argue, optimizing the hyperparameters in Word2Vec is critical, in particular, because, in our case, we are not applying it to words, but to products. We evaluated several combinations with the following values for each parameter:

Parameter	**Values**
Epoch	**5**, 15
Window Size	**5**, 50, 100
Sample	0.001, **0.01**, 0.1
NS Exponent	-0.5, 0, 0.25, **0.5**, 0.75
Embedding Size	32, 64, **128**, 256
Number of Negative Samples	3, **7**, 21

Table 4.1: Parameter Search

We chose to use the following set of parameters as they represented the best tradeoff in terms of performance and quality of results:

Parameter	**Final Value**
Epoch	5
Window Size	5
Sample	0.01
NS Exponent	0.5
Embedding Size	128
Number of Negative Samples	7

Table 4.2: Final Parameters

As gensim's Word2Vec implementation uses Stochastic Gradient Descent and goes over the whole dataset for each epoch, running it for more epochs does not affect the model as much. Though the results get slightly better if we go up to 50 or 100 epochs, we decided to stick with five because of the time tradeoff.
In our experiments, a window size of 5 and a window size of 100 performed equally well, which seems to point out that perhaps products that are purchased closer in time are more significant. If we take a closer look at the dataset exploration in section 3.3.3, we can see that most users have basket sizes ranging from 1 - 10 products, which explains why our window size of 5 works as well.
The Sample parameter determines the threshold for configuring which higher-frequency words are randomly downsampled, and we achieved the best results with 0.01.
The NS Exponent parameter determines the Negative sampling noise distribution parameter used in the Word2Vec with Negative Sampling. Negative samples are sampled from the distribution using a frequency smoothing parameter where the frequency of items is raised to the power of our NS Exponent. That means, we can adjust the probability of selecting popular or rare items as negatives. An NS Exponent of 1 would be a uniform distribution, and the original item frequencies would be used. If its value is between 0 and 1, high-frequency items are smoothed down. At 0, it is a unigram distribution, which means that the item frequency in the dataset is equal to 1 and below 0, low-frequency items are weighted up. Therefore, with 0.5, we chose a value that smooths our

most popular items and should lead to more diverse recommendations in practice.
The embedding size is self-explanatory and refers to the dimension of our embedding vectors. We choose 128, due to time and memory tradeoffs.
Lastly, the number of negative samples refers to the number of samples used when performing the Word2Vec loss calculation using Negative Sampling. As the authors explain in their paper [27], using all the items in the dataset to calculate the loss would lead to better results and much higher training time. The more samples we use, the better our results should get. We found 7 to be a good tradeoff.

4.3 Embedding Evaluation

In this section, we will evaluate the results of the embeddings generated by our four methods. Firstly, we will focus on the quantitative evaluation, where we tried to predict department categories and aisle categories, using the embeddings as input. Then, we will look at the qualitative evaluation, where we used dimensionality reduction techniques to visualize the embedding vectors in space.

4.3.1 Quantitative Evaluation

Method	**Micro - Department**	**Macro - Department**	**Micro - Aisles**	**Macro - Aisles**
Item2Vec from [30]	0.377	0.283	0.187	0.075
Triplet2Vec from [30]	0.382	0.294	0.189	0.082
Item2Vec	0.648	0.545	0.4409	0.3505
User-Item2Vec	0.5821	0.4864	0.375	0.2919
Meta-Item2Vec	**0.9585**	**0.9529**	**0.5992**	**0.4988**
User-Meta-Item2Vec	0.9434	0.925	0.5579	0.4595

Table 4.3: Embedding Evaluation

As described in Section 3.4.1 we chose a train-test split of 0.5 and the results in Table 4.3 are reported on the test set. We are also using the Item2Vec and Triplet2Vec implementation from [30] as an algorithmic baseline because they were evaluated on the same data set. Firstly, we can see that our implementation performs better than those two baselines, which might be due to our extensive parameter search, which eventually led to a highly optimized version of Item2Vec. It also shows that for industry applications, where speed and cost are critical criteria, one might put more effort into optimizing a simple method rather than going for a more complex one. We can also see that the User-Item2Vec method performs slightly worse than the Item2Vec method which could be due to the influence of the added user vector and the therefore slightly different training method (Item2Vec uses the Skip Gram algorithm from Word2Vec, while our User-Item2Vec uses the Distributed Memory algorithm which is more like the Distributed Bag of Words algorithm in Word2Vec). We can see the same effect between Meta-item2Vec and User-Meta-Item2Vec. Meta-Item2Vec performs extremely well on the Micro and Macro F1 score for the departments, and it outperforms all other methods. In Table 4.3, we only injected the department categories into Meta-Item2Vec, which is most likely the reason why the department category prediction performed so well. We also tested injecting the aisle categories and both the aisle and department categories as metadata to validate this, and present the results in Table 4.4.

We can see that when we use the aisles as metadata, the score for departments slightly improves, so does the score for aisles. This might be explained by the fact that the aisles are the lower-level category and are therefore related to the departments, the higher-level categories, as they are a

Method	Micro - Department	Macro - Department	Micro - Aisles	Macro - Aisles
Department	0.9585	0.9529	0.5992	0.4988
Aisle	0.973	0.9611	0.9502	0.9357
Department & Aisle	**0.9956**	**0.9947**	**0.9586**	**0.9396**

Table 4.4: Testing out different combinations of Metadata

subset of each department. When we inject both department and aisle categories, we achieve the best results for all scores.

4.3.2 Qualitative Evaluation

Item2Vec

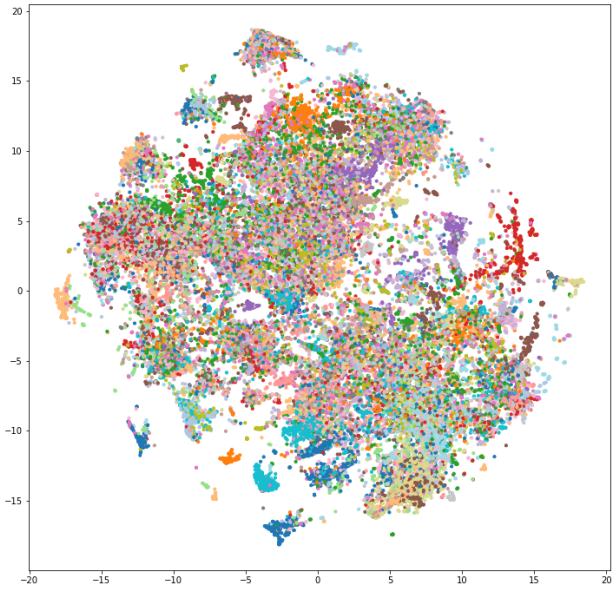

Figure 4.1: Item2Vec Embedding Visualization

While in Figure 4.1, we can see that the Item2Vec method clusters quite well with a distinction between the different categories; some clusters are hard to separate. One reason for that could be that the model struggled with the items that are less popular and were therefore exposed less during training, which resulted in an embedding of lower quality.

Meta-Item2Vec

For the Meta-Item2Vec method, we visualized the Item Embedding Vectors and also added the Category Embedding Vectors for Department and Aisles in Figure 4.2 and 4.3 respectively, and marked them in the graphic to see if the model learned something sensible for them as well.

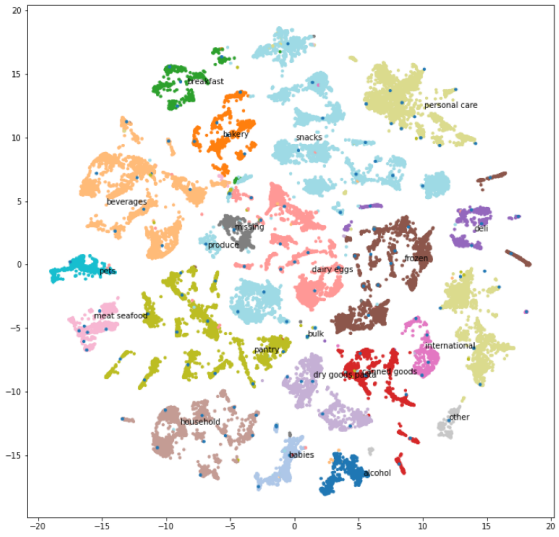

Figure 4.2: Meta-Item2Vec Embedding Visualization with Department Labels

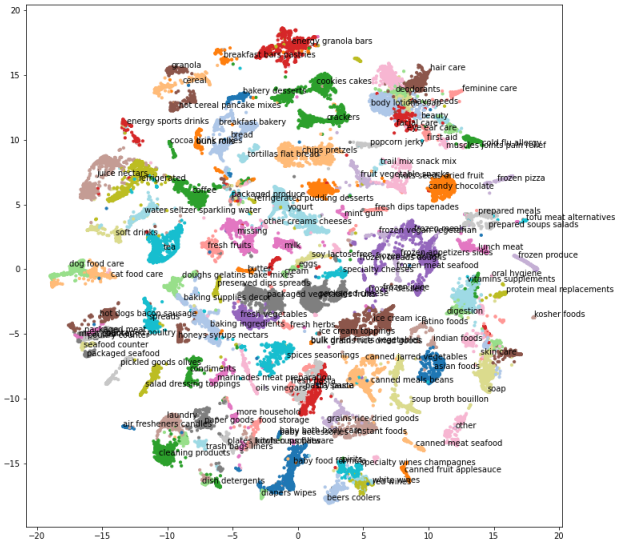

Figure 4.3: Meta-Item2Vec Embedding Visualization with Aisle Labels

First of all, we can see that the Meta-Item2Vec method with aisle and department categories injected clusters the embeddings extremely well. There are very visible separations, a small number of outliers and it seems like the category embedding vectors are building the centroids of their respective cluster. This boost in quality is likely due to the injection of the aisle and department categories as metadata into the model (as described in the [28]) to regularize the creation of the item embedding vectors.

User-Item2Vec

Finally, in Figure 4.4, we visualized the learned user vector embeddings from the User-Item2Vec method, where we can see some clusters which signify that the model differentiated between different groups of users. Unfortunately, the dataset did not include any further information about the users, which made the embeddings difficult to validate.

4.4 Recommender Evaluation

In this section, we will present each method's performance on the Within- and Next-Basket Recommendations task and show some novel item vector calculations inspired by those shown in the Word2Vec Section 2.2.1 in Figure 2.3.

4.4.1 Within- and Next-Basket Recommendations

Overall, in Table 4.6 and 4.7 we can see that the Most Popular for User method dominated all the other ones for NDCG, Recall and Precision. In [30], the authors suggest that around 70% of users in Instacart have products that are repeatedly purchased in more than half of their transactions. A possible explanation could be that the data is collected from regular shoppers on an online grocery shopping platform, where users may repeatedly seek the same products for efficiency rather than browsing and exploring as in physical stores. The AUC metric for the Most Popular for User method is weak in comparison; which may be explained as the takes into account the whole dataset

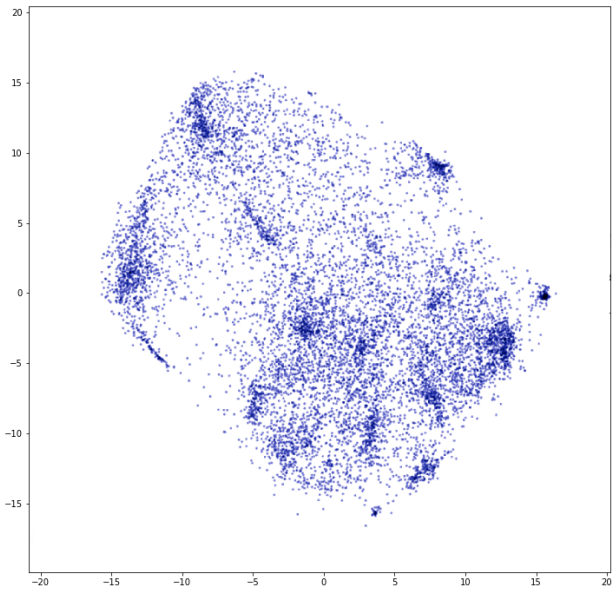

Figure 4.4: User-Item2Vec User Embedding Visualization

and - apart from the products that the user has already purchased - the rest is given a random value close to 0 which is negatively impacting the overall ranking metric.
To be able to pinpoint the weaknesses of this strong performing model, we split the users into 'warm' and 'cold' users, where the 'warm' group is the users with more than three transactions and the 'cold' group is users with less or equal than three transactions. The results in the table 4.5 below are pretty clear: as we are looking at the past transactions of a specific user, the model struggles for users who have not purchased a lot, which is known as the cold-start problem and its prevalent in modern recommender systems.

Method	**AUC**	**NDCG**	**Recall@10**	**Precision@10**
Within-Basket (Cold)	0.6610	0.1895	0.2023	0.0686
Within-Basket (Warm)	0.7933	0.2267	0.2286	0.0837
Next-Basket (Cold)	0.7086	0.2661	0.2818	0.2382
Next-Basket (Warm)	0.8334	0.3201	0.3229	0.2968

Table 4.5: Most Popular for User - Warm- and Cold-Start Users

Ensemble Method

To balance out the weakness of the cold-start problem while still leveraging the strength of our Most Popular for User-method, we calculated scores for an ensemble method between Item2Vec and Most Popular for User. Our motivation was that the Item2Vec method helps out with the cold-start problem and gives context-specific recommendations. To calculate the scores for the

ensemble method, we used the formula in Equation 4.1:

$$Ensemble(Item2Vec, MostPopularforUser) = \alpha \cdot Item2Vec + (1 - \alpha) \cdot MostPopularforUser \tag{4.1}$$

We tried ten values for the α-value within the range of 0.0 - 1.0 and found 0.8 to work best.

Method	**AUC**	**NDCG**	**Recall@10**	**Precision@10**
Most Popular	0.9216	0.133	0.0543	0.0258
Most Popular for User	0.7271	0.1987	0.1978	0.0731
Co-Count	0.8917	0.1527	0.0791	0.0335
Item2Vec	0.9546	0.1556	0.0789	0.0338
User-Item2Vec	0.96	0.153	0.0724	0.0317
Meta-Item2Vec (with category)	0.952	0.1544	0.0769	0.0327
User-Meta-Item2Vec (with category)	0.9592	0.1527	0.0723	0.0312
Item2Vec/Most Popular for User	**0.9701**	**0.2364**	**0.227**	**0.0802**

Table 4.6: Within-Basket Recommendations

Method	**AUC**	**NDCG**	**Recall@10**	**Precision@10**
Most Popular	0.9276	0.1449	0.0706	0.0757
Most Popular for User	0.769	0.2791	0.283	0.2585
Co-Count	0.9553	0.1671	0.0893	0.0904
Item2Vec	0.9642	0.1743	0.1037	0.1003
User-Item2Vec	0.9678	0.1736	0.0989	0.098
Meta-Item2Vec (with category)	0.9617	0.1728	0.102	0.099
User-Meta-Item2Vec (with category)	0.9673	0.1753	0.1023	0.1005
Item2Vec/Most Popular for User	**0.9768**	**0.3244**	**0.3173**	**0.2833**

Table 4.7: Next-Basket Recommendations

We observe that the results are almost equal for all four models and for both evaluation scenarios. This might be because we chose a parameter combination that optimizes the models to a high degree. The Item2Vec/Most Popular for User ensemble performs the best by a significant margin. The main takeaway is that the additional user vector does not add value in generating embeddings that make better recommendations. Another conclusion we can draw here is that, in our case, adding the metadata categories does not provide any additional value for the result. It does provide value for the item embedding evaluations, but these are only a proxy metric that aims to evaluate whether the embeddings have learned anything useful. When we evaluated the different types of metadata injections (Figure 4.4 on Page 32), we were able to observe that although the F1 scores were significantly improved, the created embeddings added no uplift at all to our Within-Basket and Next-Basket recommendations task. This could be because we are trying to recommend the most complementary rather than most similar items which mean we are using the out- (or context-) vectors rather than the in-vectors used for clustering and most similar item predictions (even though the basket vectors that we average are from the in-vectors). This suggests that our embeddings might perform well on a most-similar item prediction task that could be performed on a product page where we could prompt the user with something like: 'See more items like this'. Therefore one improvement for future work could be to evaluate the embeddings on the task of similar item recommendations (e.g., on the product page) and see if the metadata adds more value in this context.
Finally, as explained in Chapter 1 in Section 1.6, all these results would have to be tested in a production environment against real business metrics to evaluate them properly.

4.4.2 Product Vector Calculations

Similar to the experiments in the Word2Vec paper [18] with the commonly given example calculation of $Vector(King) - Vector(Man) + Vector(Woman)$, which is approximately equal to $Vector(Queen)$ we try to do our own vector calculations using the product embedding vectors. The goal here is to see whether the item embedding vectors can encode some underlying semantic characteristics.
We used the final Item2Vec model, did the calculations shown in Table 4.8 in the 'Query' column, and used the resulting vector to search the embedding space for the most similar vectors. If the target item was within the top five results, we deemed the search as successful. We can see that the model is somewhat able to encode semantics like 'Organic', 'Frozen', 'Unsweetened', 'Low Fat', 'Gluten-Free', 'Sun-Dried' and 'Spicy'.

Feature	**Query**	**Top Results**
Organic	Organic Banana - Banana + Fuji Apples	Apples, Clementines, Organic Simply Pita Chips, **Organic Fuji Apples**, ...
Frozen	Frozen Organic Blueberries - Organic Blueberries + Strawberries	Gluten Free Cheese Pizza, Classic Vermont Cheddar Cheese, Organic Strawberries, Strawberries Clamshell, **Frozen Strawberries, ...**
Unsweetened	Unsweetened Almondmilk - Original Almondmilk + Coconut Milk	**Unsweetened Organic Coconut Milk, ...**
Low Fat	Low Fat Plain Yogurt - Whole Milk Plain Yogurt + Whole Milk	Almond Orgeat Syrup, Mango Syrup, 2% Reduced Fat Milk, Creme Brulee Coffee Creamer, **100% Lactose Free Reduced Fat Milk, ...**
Gluten-Free Bread - 100% Whole Grain	Gluten Free Whole Grain Gluten Free Cinnamon Raisin Bread, Bread + Multi Grain Waffles	Gluten Free Omega Flax & Fiber Bread, Gluten Free Millet-Chia Bread, **Mighty Bagels, Wheat Gluten Free Waffles, Organic Gluten Free Wildberry Waffles, Gluten Free Blueberry Waffles, Gluten Free Apple Cinnamon Waffles, ...**
Sun-Dried	Sun Dried Tomatoes - Vine Ripe Tomatoes + Raw Goji Berries	**Sun-Dried Goji Berries, ...**
Spicy	Spicy Hummus - Original Hummus + Tomato Basil Pasta Sauce	Mushroom Marinara Pasta Sauce, Goat Cheese & Basil Ravioli, **Spicy Marinara Pasta Sauce, ...**

Table 4.8: Product Vector Calculations

5 Summary

This chapter summarizes the results of this thesis and provides an overview and outlook on what can be improved in future work.

5.1 Summary

In this thesis, we evaluated different approaches to generate vector embeddings for products within Recommender Systems. Starting with an overview of the current state-of-the-art industry Recommender Systems, we continued by laying the foundation for methods that have been successfully used to generate word embeddings in the NLP domain. We then switched the focus from word embeddings to product embeddings in an E-Commerce setting and introduced the landmark papers, which are the cornerstone of this thesis. The goal of Chapter 4 was to evaluate our four chosen techniques, namely Item2Vec, Meta-Item2Vec, User-Item2Vec, and User-Meta-Item2Vec on the Instacart transactions dataset and explore the actual impact of some of the proposed additions. We decided to focus on the two additions of metadata and user data, which extended the base implementation of Item2Vec.
As it turned out, the proposed extensions did not provide any additional value for our end result, which we defined as the quality of Within-Basket and Next-Basket Product Recommendations. Since around 70% of users in the Instacart dataset have repeatedly purchased products in more than half of their transactions, the proposed 2Vec methods were unable to beat the Most Popular for User baseline, which always recommends the most often bought items by each user.
This approach's main weakness is the cold-start problem, which applies when a new user signs up for the Instacart service and has not purchased enough to reliably use their purchase history as an indicator for her future preferences. In this scenario, though, the Item2Vec method should still perform well because it looks at local co-occurrences of products rather than user-specific global co-occurrences. Based on that assumption, we tried an ensemble method where we took a weighted combination of Item2Vec and the Most Popular for User recommendations and reported the test set results. The resulting method outperformed all other methods, supporting our hypothesis. Lastly, we experimented with the resulting embedding vectors to see, like in Word2Vec, if they can encode any underlying semantic characteristics. It turned out that they can encode semantics such as 'Organic', 'Frozen', 'Unsweetened', 'Low Fat', 'Gluten-Free', 'Sun-Dried' and 'Spicy'.

5.2 Outlook

Some of the areas that we could explore further to improve upon this thesis's work would include Hyperparameter Optimization, characteristics of the Dataset, User-Product Loyalty, Ranking, and other embedding methods.

Hyperparameter Optimization

Even though we evaluated around 100 parameter combinations, we could have further evaluated the bigger parameters and tried to max them out. We chose not to do this given that maxing out each of the parameter combinations would have dramatically increased the training time. The specific parameters we would have focused on are:

- Training epochs
- Number of Negative Samples
- Window Size
- Embedding dimension

Dataset

The Instacart dataset contains 3 million grocery orders from more than 200,000 Instacart users, which is the right size for the quick-iteration experiments we were trying out. In contrast, the dataset used in [20] contains 280.7 million purchases made by 29 million users, which means there is room for improvement especially considering that adding more data, if possible, is always an easy way to improve the results of a model.

User-Product Loyalty

As described in Section 5.1, the Most Popular for User baseline was valuable because, in our context, around 70% of users in the Instacart dataset have products that are repeatedly purchased in more than half of their transactions, meaning there is a high degree of product loyalty. In order to leverage this, we implemented the ensemble method, which takes a weighted average of Item2Vec and the Most Popular for User method to generate recommendations. It could have been beneficial to look at some more sophisticated methods for this weighting, such as a small logistic regression model or to explore something like the AdaLoyal algorithm described in [30].

Ranking

In Chapter 1 Section 1.4, we introduced a 2-stage architecture that is used for most Recommender Systems: Candidate Generation and Ranking. In this thesis, we mainly focused on the candidate generation aspect and a little on more sophisticated ranking methods. It could be interesting to explore one of the supervised classification methods for ranking. One one hand, as we have seen, typical choices include Logistic Regression, Support Vector Machines, Neural Networks, or Decision Tree-based methods such as Gradient Boosted Decision Trees (GBDT). On the other hand, there are a significant number of algorithms specifically designed for learning to rank that have appeared in recent years, such as RankSVM or RankBoost.
Another idea described in [20] was that for the methods involving a specific user, one could add a recency factor, which would weight more recently bought products higher, though we are unsure about the relevance of this in a grocery shopping context.

Other Embedding methods

Finally, there are other embedding methods that we did not explore either because of the lack of data features or because these methods would have been too far out of scope. One additional improvement would be to include users' browsing sessions as described in [31], but the Instacart dataset did not include them. Another method that we often referenced in this thesis is Triplet2Vec [30]. It would have been interesting to implement it and look closer at how it performs under different parameter settings, though, as we have seen in the results section, our optimized methods beat the Triplet2Vec significantly already.
Another approach to inject metadata into the model is described in [35], where the authors leverage textual title metadata and also introduce a novel quadruplet loss. It would be worthwhile to compare its results against our implementation of Meta-Item2Vec. In [26], the authors explored adding contextual information into the model, which seems like another interesting approach to explore. Lastly, there is a whole domain on Graph Learning, which we did not go into. One of the more successful graph embedding models, for example, include [22].

6
Bibliography

[1] Michael Schrage. Great digital companies build great recommendation engines.

[2] Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4), December 2016.

[3] Chris Meyer Ian MacKenzie and Steve Noble. How retailers can keep up with consumers.

[4] Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '18, page 311–320, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Sangdi Lin. Home embeddings for similar home recommendations.

[6] Taylor Gordon Ivan Medvedev, Haotian Wu. Powered by ai: Instagram's explore recommender system.

[7] Larry Hardesty. The history of amazon's recommendation algorithm.

[8] Michael Howe. Pandora's music recommender. 2007.

[9] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016.

[10] L. Wu, S. Shah, S. Choi, Mitul Tiwari, and Christian Posse. The browsemaps: Collaborative filtering at linkedin. *CEUR Workshop Proceedings*, 1271, 01 2014.

[11] Luca Belli Abhishek Tayal Dan Shiebler, Chris Green. Embeddings@twitter.

[12] Isaac Liu Yuyan Wang, Yuanchi Ning and Xian Xing Zhang. Food discovery with uber eats: Recommending for the marketplace.

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

[14] Xavier Amatriain and Justin Basilico. Netflix recommendations: Beyond the 5 stars (part 2).

[15] Pai Liu Chun How Tan Liang Wu Bo Yu Alex Tian Mihajlo Grbovic, Eric Wu. Machine learning-powered search ranking of airbnb experiences.

[16] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1235–1244, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Ankur Sarda Ankit Jain, Isaac Liu and Piero Molino. Food discovery with uber eats: Using graph learning to power recommendations.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[19] Oren Barkan and Noam Koenigstein. Item2vec: Neural item embedding for collaborative filtering, 2016.

[20] Mihajlo Grbovic, Vladan Radosavljevic, Nemanja Djuric, Narayan Bhamidipati, Jaikit Savla, Varun Bhagwan, and Doug Sharp. E-commerce in your inbox: Product recommendations at scale. *CoRR*, abs/1606.07154, 2016.

[21] Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic, Fabrizio Silvestri, Ricardo Baeza-Yates, Andrew Feng, Erik Ordentlich, Lee Yang, and Gavin Owens. Scalable semantic matching of queries to ads in sponsored search advertising. *CoRR*, abs/1607.01869, 2016.

[22] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[23] John Ciancutti. How we determine product success.

[24] YouTube. Youtube now: Why we focus on watch time.

[25] Instacart. 3 million instacart orders, open sourced.

[26] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. Modeling complementary products and customer preferences with context knowledge for online recommendation. *CoRR*, abs/1904.12574, 2019.

[27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. pages 3111–3119, 2013.

[28] Flavian Vasile, Elena Smirnova, and Alexis Conneau. Meta-prod2vec: Product embeddings using side-information for recommendation. *Proceedings of the 10th ACM Conference on Recommender Systems*, 2016.

[29] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents, 2014.

[30] Mengting Wan, Di Wang, Jie Liu, Paul Bennett, and Julian McAuley. Representing and recommending shopping baskets with complementarity, compatibility and loyalty. pages 1133–1142, 10 2018.

[31] Ilya Trofimov. Inferring complementary products from baskets and browsing sessions. *CoRR*, abs/1809.09621, 2018.

[32] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[33] Rico Meinl. Instacart2vec github repository.

[34] Hugo Caselles-Dupré, Florian Lesaint, and Jimena Royo-Letelier. Word2vec applied to recommendation: Hyperparameters matter. *CoRR*, abs/1804.04212, 2018.

[35] Mansi Ranjit Mane, Stephen Guo, and Kannan Achan. Complementary-similarity learning using quadruplet network. *ArXiv*, abs/1908.09928, 2019.

7

Assertion under Oath

I hereby declare on oath that I have done this work independently and without using any tools other than those specified; the thoughts taken directly or indirectly from external sources are identified as such. The work has so far not been presented in a similar form to any other examination board and has not been published.

London, 27.07.2020 Rico Meinl