Content Based Recommender Systems: Feature Embeddings From User-Item Interaction Sequences

Joseph Renner

Contents

Chapter 1

Introduction

I present a new approach of learning item features to be used for content-based recommendation. I developed this approach during my internship at Criteo in Paris, France.

1.1 Company Overview

Criteo is commerce marketing company, working with online retailers to develop ads to bring customers to their sites. It is a leading company in the ads ecosystem. It is headquartered in Paris, France.

1.1.1 Products and Customers

Criteo offers a few products to online marketers as well as a couple to online publishers. These products are centered around online advertising.

Criteo's main customers are online marketers; that is, providing ads for online marketplaces. These customers are known as *partners*. Partners can pay for specific types of ads they would like specific customers to see. For example, one of the main products Criteo offers is known as *Dynamic Retargeting*. Dynamic Retargeting involves showing personalized ads of a partner's products to users who have previously engaged with the partner's products on the partner's site.

When purchasing Criteo's services, partners embed a tracking cookie in their website code, so that Criteo can receive data about what products a user is interacting with and how. This data allows Criteo to (1) try to predict if this user is a good target for an ad, and (2) try to predict which products Criteo should show in the ad. Also, the partner provides catalog data for the goods they sell, allowing Criteo's recommendation algorithms to know which products to recommend in an ad.

Criteo also has services for bringing new customers to a partner's website. This is called *Customer Acquisition*. It involves providing personalized recommendations for customers that have not shopped on a partner's website.

Criteo partners are from all over the world and they are divided into platforms. There are three platforms: North America, Europe, and Asia.

1.1.2 Online Ad Environment

To understand Criteo's business and the context of my internship, I will give a brief introduction to the online advertising environment. *Publishers* are websites that publish content and want to offer advertising space on their websites or apps. These ad views are offered whenever a user lands on their page, and are called *impressions*. *Advertisers* are the ones who want these impressions to show their advertisements. Buyers of these impressions, such as Criteo, bid on the impressions on behalf of advertisers in real time. Higher bids are placed for users who have a higher predicted probability to buy products. Criteo bids on advertisements based on machine learning models and data from users and previous advertisements. If they are the highest bidder, the advertisements are created that will optimize traffic and sales to the advertiser's products.

Criteo's business model uses Cost Per Click (CPC) pricing for retargeting and customer acquisition. This means that partners only pay for ads if the ad is clicked and the user actually lands on the site.

1.1.3 Engine

The Criteo engine is what powers the products it offers. The engine consists of 3 main components: predictive bidding, product recommendation, and kinetic design. Machine learning is integrated into all three components of the system. Predictive bidding is responsible for bidding on the right ads at the right time that have the highest probability for converting. It achieves this by predicting the revenue of an ad given the user data. The recommendation team is responsible for designing and implementing the system for choosing which products to put in ads, utilizing machine learning and mass amounts of consumer data. More details and system architecture are presented in the following subsection. The kinetic design team is responsible for how the ads appear to shoppers. It uses machine learning and data from previous ad interactions to design the ad, including layout, colors, and branded elements. Like product recommendations, the ad designs are tailored to the individual shopper.

Efficiency is key for the engine as a whole. Criteo ads are seen by over 1.4 billion users per month, and it deals with over 30 billion bid requests per day. The Criteo engine typically has 200 milliseconds to respond to an ad request; meaning, 200 milliseconds to determine whether or not to display an ad for a specific user, determine which products to display among millions of possible products, and determine the optimal look and feel of the ad. Thus, these processes must be scalable and fast.

1.1.4 Recommendation System Overview

As my internship was with recommendation team, I will give an overview of the team's tasks, challenges, and architecture. As said above, the recommendation team is responsible for determining which products are shown to which users, efficiently. There are a couple of major challenges to this part of the Criteo engine. First of all, recommendations must be scalable, as there are billions of users and billions of products. Also, partner catalogs change daily, so recommendations must be updated regularly.

The recommendation task is as follows: given a user and list of historical products (products for which this user has interacted with in the past, whether it be viewing the product web-page, adding the product to their basket, or buying the product), compute the k products that will maximize their purchases on the partner website. As we will see in the recommendation algorithm overview chapter, factorization of a user-item interaction matrix is the textbook approach to generating recommendations, as each product is represented by a low-rank latent factor vector, with which the k-nearest neighbors algorithm can be applied to find similar product to historical products. But, given the two challenges at Criteo of scalability and reactivity, pure matrix factorization of user-item interactions is infeasible. First of all, explicit representation of a user-item matrix of all users and items is simply too large. Criteo ads see almost 1.4 billion users a month, and there are billions of products as well. Second, recommendations need to be updated every time new products or new interaction data for a user is available, and matrix factorization is not reactive enough to meet these needs.

Thus, a different system for computing recommendations is needed. First, offline jobs compute *similarity lists* for each product. Similarity lists amount to the k (usually k=15) most similar products to the key product. The similarity lists are precomputed so that when a request comes, scoring is only needed for a very small subset of the whole catalog for a partner. The scale for these jobs are huge, as there are over 12 billion products across all partners. These lists are product based, meaning they provide similar products to every product in a catalog, known as *key products*. The links between products will change only slightly between a couple days, but if a user views a product and receives an ad shortly after, the ad will contain products based on this new view, since it is added to his/her tracking cookie, which is taken into account by the recommendation component in the form of *historical products*. The similarity lists are then pushed to a key-value NoSQL database which can be accessed quickly when serving ads. At request time, the similarity lists are looked up for the historical products, and these lists, possibly concatenated with a non-product-specific list of best selling products for a partner, are the candidate products

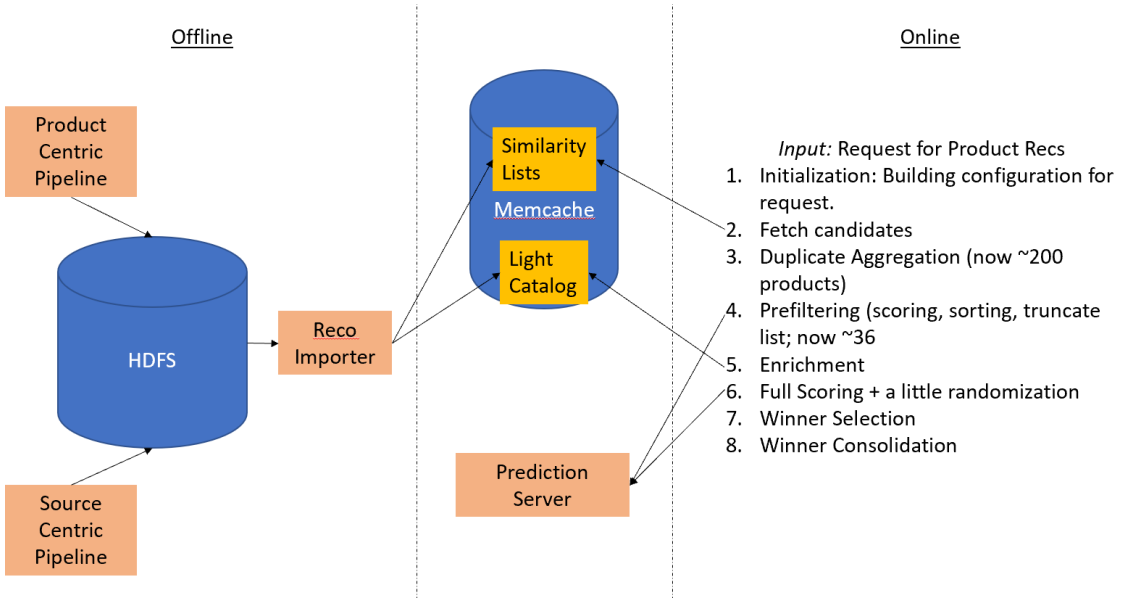

Figure 1.1: Architecture of the Recommendation Pipeline. On the left side are the offline pipelines, explained in the following pages. On the right side are the steps the systems takes when a request for product recommendations comes in.

sent to scoring. This pipeline can be divided into three processes: RecoComputer, RecoImporter, and RecoService. The full architecture can be found in figure 1.1.

RecoComputer

The RecoComputer component of the system pre-computes all the information needed for product recommendations, utilizing Hadoop clusters.

There are two main pipelines of the RecoComputer component. One is more product based: it extracts features such as cross-partner categories that will be used by scoring models later in the recommendation pipeline, unifies all catalogs needed for prediction later in the pipeline, and extracts the most popular products (by page-view or sales) for each partner. The architecture is shown in figure 1.2. Here we see product features such as product embeddings (more on these later), product metadata from the Catalogs, data from advertiser events such as page views or sales, and cross-partner categories from the Universal Catalog are merged in the RichCatalog job. From here these unified product features are outputted to multiple sources: BestOfs (most popular products by page view or sales that will be considered when recommending products in ads), EnrichedCatalog (partner catalogs enriched with other data such as embeddings of universal categories), and LightCatalog (a version of EnrichedCatalog with some information missing).

The other pipeline is more source-centric. It's architecture is shown in Figure 1.3. *Sources* are the product similarity list outputs of different recommendation algorithms. This pipeline computes similarity lists for each product using multiple algorithms, then merges these similarity lists using another model. I will now give a quick overview of the sources and merging model. The first source is based on co-event counts. Co-event counts are the number of times a product has a "co-event" with another product. A co-event is anytime a user triggers advertiser events (such as product page view or product sale) for two products within a set amount of time. This amount of time can be short (10 minutes) or long (7 days). Once co-events are counted for pairs of products, they can be used to build similarity lists by choosing the products that have the most co-events for each product.

The next source is based on collaborative filtering. As normal matrix factorization techniques for collaborative filtering are infeasible at this scale, a different approach must be used. The source is computed in MapReduce and uses statistics on weighted co-events grouped by users to produce similarity lists. Co-events are weighted according to the following equation:

$$w(c_{e_1,e_2}) = \exp(\frac{-d(c_{e_1,e_2})}{s}) \tag{1.1}$$

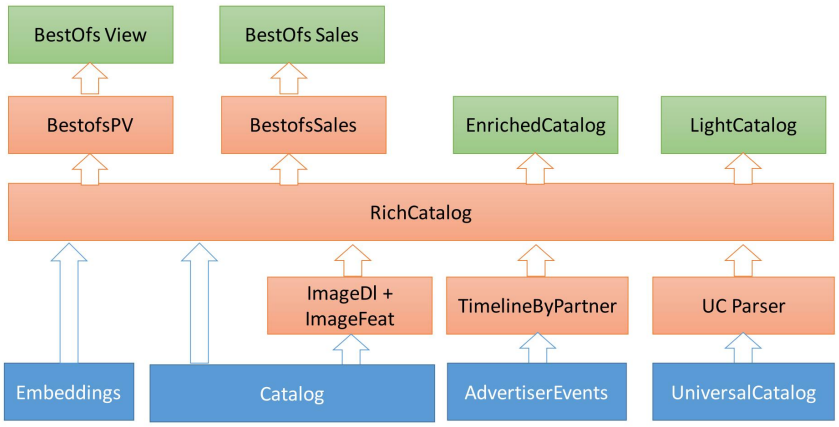

Figure 1.2: RecoComputer Product-Based Pipeline

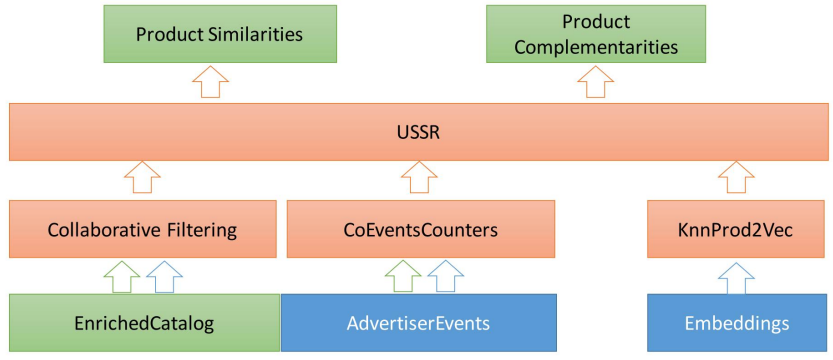

Figure 1.3: RecoComputer Source-Based Pipeline

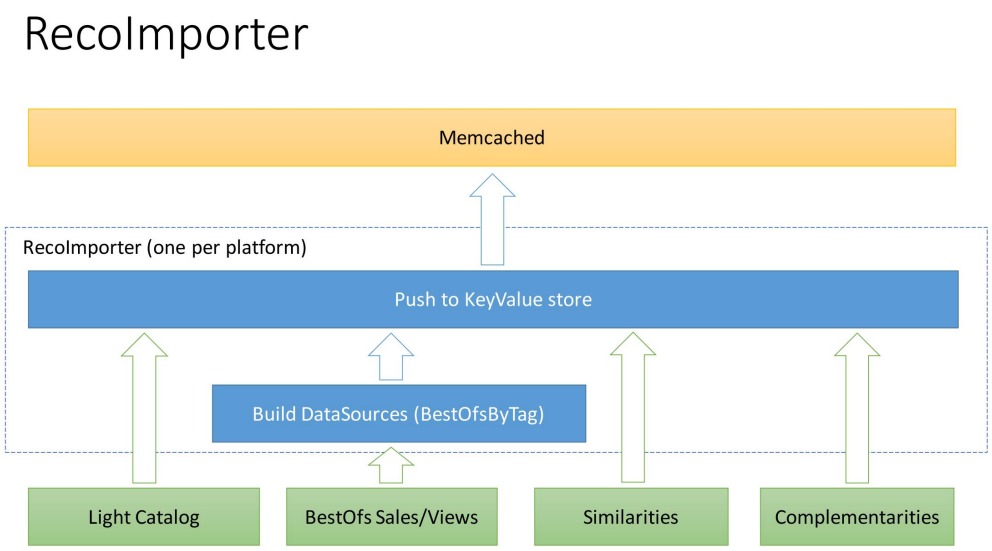

Figure 1.4: RecoImporter Pipeline

Where $w(c_{e_1,e_2})$ is the coevent weight between two events (e_1, e_2), $-d(c_{e_1,e_2})$ is the time between the two events, and s is a semi-life constant.

The final source is based on product embeddings. Embeddings are dense, low-rank vector representations of products. The same information used for collaborative filtering is used to compute product embeddings, using the algorithm proposed by [1]. More on this algorithm and embeddings in general in the next chapter. Once product embeddings have been computed, a k-nearest neighbor based on cosine distance is used to build the similarity lists. As the scale of products is quite large, a brute-force KNN approach is not feasible, thus the MapReduce implementation makes use of z-curve method [2], an algorithm that builds an approximate-KNN graph using space-filling curves. This source is not yet used in production, but at the time of writing this document it is being AB tested.

Once the three above sources are computed, their resulting similarity lists are merged using another model, called Universal Source Selection for Reco (USSR). The USSR model scores each of the products in the similarity list, choosing the best scored products as the final similarity list for a key product. It uses the similarity scores from the similarity lists as well as user-display information and product features from the EnrichedCatalog to score the products. This is contrast to the similarity lists themselves, which currently only use user-product interaction data to compute the lists. There are two USSR outputs for each key product, similarities and complementaries. The complementaries are products that are likely to be bought with a key product; for example, a cellphone case would be a complementary product to the cellphone. An example of similar product would be a different type of cell phone.

RecoImporter

The main purpose of the RecoImporter component is to import all the previously computed data from RecoComputer to CouchBase, which is a key-value storage similar to Memcached which provides fast lookups. Thus, it is the link between the online and offline components. The pipeline for RecoImporter is presented in figure 1.4.

RecoService

The RecoService component serves recommendations in real-time using the data stored in CouchBase. Once the service receive a request for recommendations (in the form of historical products and recommendation context) the following steps are taken to provide the recommendations: fetch from CouchBase the similarity lists of the historical products, remove any duplicates among the

lists, filter out products using the pre-filtering model (this includes products that the user has already bought), enrich the remaining products with LightCatalog (as these features will be used to score the products), score all the remaining products using the full scoring model (obtained from Prediction service, another component of the engine), and finally select and send the winners based on the score and some randomness.

1.2 Internship Tasks

1.2.1 Content-Based Recommendation

For my internship, I was tasked with creating a system to generate recommendations based on product content; namely, the text data associated with each product. As we have seen in the previous section, the current recommendation sources are either based on collaborative filtering based (CF or Prod2vec), co-counts, or best-ofs. A consequence of this absence of a content-based source is that the current system will be sub-optimal on new products. The cold-start problem will be apparent when user-item interaction data is sparse. While data sparsity maybe isn't as big as a problem in other recommendation systems (Criteo collects 2 petabytes of customer data per day and reaches 1.4 billion shoppers a month), it can still affect recommendations when a partner adds new products to a catalog. Namely, the new products, while possibly could be interesting to certain shoppers, won't be recommended until they have the appropriate interaction data to cause their selection. This could possibly be a larger problem for partners that have a high rate of product turnover, such as online marketplaces where customers can buy and sell products. Thus, my internship task is to leverage product text data to solve the cold-start problem.

Catalog Data

The partners upload their catalog to Criteo at fixed intervals. In this catalog, each product has meta data associated with it. This can be prices, categories, the name of the product, a textual description, and many other attributes. For my internship, I was just concerned with the text data: the name and possibly the description.

Challenges

There are a couple challenges when thinking of a text-based recommendation design. First, as there are about 20,000 partners from all over the world, there is great diversity in the text data. In specific, the language the product text data is in can vary from partner to partner. Also, in some bi- or tri-lingual countries, the product text language can vary inside a partner's catalog. This means that any text-based recommendation system must work for many languages. As many natural language processing techniques are specific for a language, such as stop word removal or pre-trained word embeddings, this adds a constraint to the design of the system. Furthermore, Asian languages create a larger challenge still, as their grammar makes it so even basic NLP tasks such as tokenizing non-trivial. In the end, I focused on western languages because (1) Asian partners make up only a minority of the partners and subsequent revenue and (2) I do not know enough about the linguistics of Asian languages to appropriately design a NLP system for recommendations.

Another challenge is the the scale of the data and system. Any text-based recommendation system I design must scale to around 15,000 partners (excluding partners with non-western languages), some with millions of products. This adds another constraint on the design of the system, which must be efficient enough to scale to volume of data and recommendations needed at Criteo.

1.2.2 Recommendation Metrics

When building any new component, it is important to be able to evaluate it, meaning you need (1) a naive baseline to compare the component to, (2) metrics to evaluate relative performance of the new component to the baseline, and (3) test data in order to use the metrics. To be able to build and appropriately evaluate the new text-based recommendation system, I needed to have these tools in place. Thus, a related task of my internship was to implement metrics for source similarity lists in order to compare the proposed system to baselines. As the similarity lists are computed offline and not directly shown to the users (as similarity are just the beginning of the

reco pipeline), metrics are not as trivial as a simple AB test. More on the implemented metrics in the section 4.2.4.

Chapter 2

Recommendation Systems Overview

Recommender systems aim to understand interactions between users and items in order to predict which items to suggest to each user. The interaction will help measure the interest of users for items, whether in an explicit way (e.g., the ratings of items given by users) or in an implicit way (e.g., the number of listens for music, the number of views for videos or the amount of time the user stays on a page for web pages).

Recommendation systems have become ubiquitous in recent years as enterprises seek new ways of recommending an increasingly large number products and services to customers. Recommendation algorithms have benefited from a surge in data volume as well, leading to more effective and complex models. Thus, the web experience is becoming more and more personalized for users. For example, users receive product recommendations from Amazon, movie recommendations from Netflix, music recommendations from Spotify, and friend recommendations from Facebook. These recommendation systems leverage the patterns of user-item (items being movies, products, songs, etc.) interactions and item features to predict what each user will like or find interesting in the future.

I will now give an introduction to the different kinds of recommendation systems.

2.1 Content-Based Recommender Systems

The system recommends items that are similar to the ones that the user liked (implicitly or explicitly) in the past. This similarity is calculated based on the features of an item, constructed from the content of the item. This content can be a number of things. Often, it is unstructured meta-data. Another example of the content is item text reviews. Some systems try to deduce ratings, moods, characteristics, opinions, or sentiments from these reviews in order to discover semantics used to recommend items. Data sources such as the semantic web can be used to collect external knowledge about items that will in turn help create more rich item features.

2.2 Collaborative Filtering

Collaborative filtering systems make recommendations based on finding items or users that have similar user-item interaction data to the user or item in question. This interaction data can be explicit (such as item ratings) or implicit (such as song listening). In the case of user-based collaborative filtering, the algorithm finds users that have interacted with some of the same products as the user in question, and then recommends products that these users have interacted with. In the case of item-based collaborative filtering, items that are interacted with by the same users that interact with the key item are recommended. This is the basic idea of collaborative filtering. More detail on the algorithms and techniques to implement these ideas is given in later sections. Collaborative filtering is known as the most popular and widely implemented recommender system technique [3]. However, when the number of users or items is large, sometimes this techniques can suffer from data sparsity, as many items are not rated or have very few ratings. This is known as the cold start problem, as new products with no are very little ratings are rarely recommended by collaborative filtering systems. The cold start problem is one of the main motivations for many

content based systems, since content-based systems do not rely on user-item ratings but instead rely on item content, which usually is available as soon as the item is.

Common implementations of collaborative filtering include matrix factorization of large, sparse item-item co-occurrence matrices to obtain a dense, low-rank vector that can effectively model item hidden factors. Recently, neural language models have been successfully applied to the recommendation task. More on this in the following section.

2.3 Hybrid

Hybrid recommender systems are a combination of collaborative filtering and content-based methods. As an example, collaborative filtering can be used for items that have been around for a while and thus have many user interactions, which in turn means they are more likely to be rated by similar users. However, new items that do not have many user interactions are unlikely to be recommended by collaborative filtering algorithms (which creates consequences discussed in the next section); content-based algorithms do not face this struggle with new items. Thus, using collaborative filtering to recommend established items and content-based methods to recommend lesser known items, then combining the results in some way would be an example of a hybrid system.

2.4 Neural Networks and the Recommendation Task

Neural Networks have seen an increase in popularity in the last 5 to 10 years due advances in fields such as computer vision and speech recognition, as well as more powerful computational resources. Given these advances, neural networks have increasingly been applied to the recommendation task. A thorough summary of recent neural network recommendation techniques has been presented in [4]. In the following subsections, I provide a short survey of relevant collaborative filtering, content-based, and hybrid neural-network recommendation techniques, as well as a section on Word2Vec, a neural language model which has been adapted to recommendation tasks and is the basis for my proposed method.

2.4.1 Neural Language Models and Word2Vec

Neural language models gained attention in the early 2000s after the work of [5]. The authors generalize statistical language model better than previous models by learning distributed representations of words and using these word embeddings to learn the parameters of the probability function of word sequences. Both the word embeddings and the probability function are learned simultaneously. Mathematically, a statistical model of a language can be represented by the conditional probability of the next word given the previous words:

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t | w_1^{t-1}) \tag{2.1}$$

where w_t is is the t-th word and w_1^{t-1} is the sub-sequence of words $(w_1, w_2, ..., w_{t-2}, w_{t-1})$.

The authors of [5] approximate this probability by a training a neural network with 3 layers: first, an embedding look-up that maps each input word index in a sequence to its corresponding word embedding and then concatenates the embeddings of a sequence; next, a hyperbolic tangent hidden layer; finally, a softmax output layer which will give the probability of a word given the input word sequence. This approach beat state of the art results at the time on predicting the next word in large corpuses. However, training took quite long (several weeks) as computers were less powerful and training was inefficient.

Word2vec

In 2013, [6] introduced word2vec, two novel neural network architectures for computing continuous vector representations of words from very large datasets. The models take advantage of the assumption that words closer together in a sentence are statistically more dependent. This assumption is known as the distributional hypothesis, first introduced in 1954 by [7]. What differentiates word2vec from previous neural language models such as [5] and the variants that followed is the

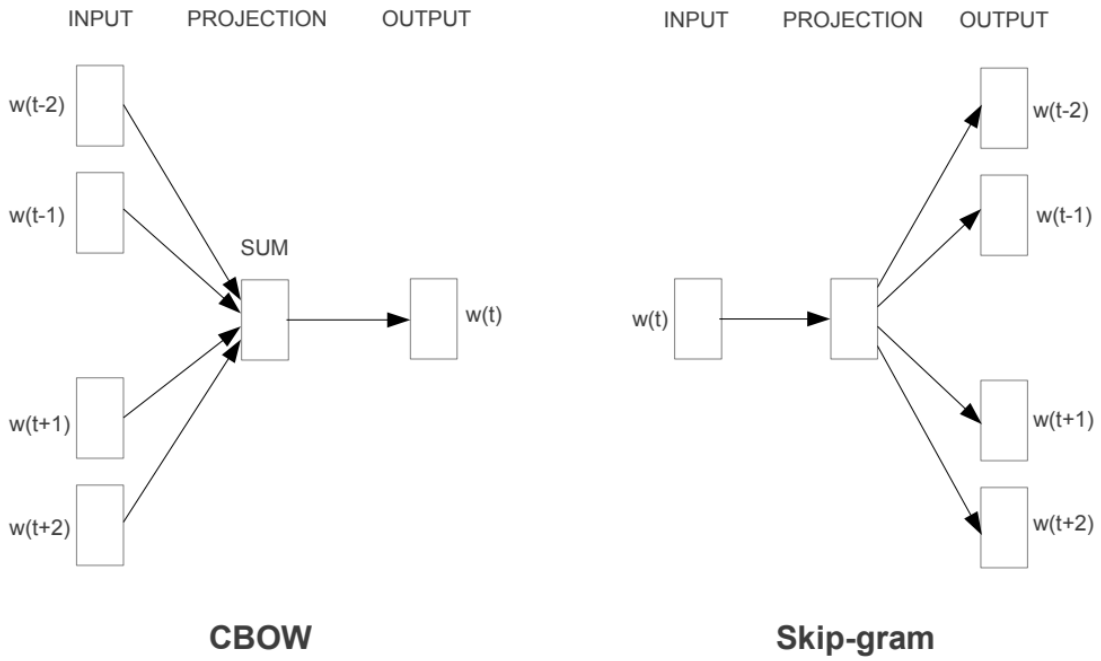

Figure 2.1: Word2vec architectures: Continuous bag of words and skip-gram.

authors do away with the nonlinear hidden layer used to learn the statistical language model, and just focus on learning the word embeddings using the single hidden layer. As most of the complexity of previous neural language models was in the non-linear hidden layer, the word2vec architectures are considerably more efficient. Furthermore, the performance of the resulting word vectors exceeded that of the state of the art on many NLP tasks.

Word2Vec Architectures

The authors of [6] introduced two neural network architectures for learning word embeddings. The first, known as continuous bag of words (CBOW) predicts words based on it's context (context of a word being the k words preceding the word and k words after the word in a sequence; the k value is a hyper-parameter known as *window size*). The second architecture, known as skip-gram (SG), does the opposite: predicts the context from a word.

Candidate Sampling

The authors of [6] published a paper [8] documenting extensions to the skip-gram architecture that significantly improve the efficiency of training and the quality of the resulting word embeddings. One extension is a replacement of the relatively expensive softmax final layer with a more simple training technique. To understand this replacement (and later recommendation system implications), we must understand the skip-gram objective function, which is maximized, over a training sequence $(w_1, w_2, ..., w_{T-1}, w_T)$, which is defined as:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_j) \tag{2.2}$$

where c is the window size. The probability $p(w_{t+j}|w_t)$ is then defined using the softmax function:

$$p(w_o|w_I) = \frac{exp(v'^T_{w_o} v_{w_I})}{\sum_{w=1}^{W} exp(v'^T_w v_{w_I})} \tag{2.3}$$

where $v_w^{'}$ is the output representation of the vector for word w, v_w is the input word embedding vector representation of the vector for word w, and W is the size of the vocabulary. As the vocabulary can be quite large when training over large datasets of sentences, this computation can be expensive. In [8], the authors mitigate this inefficiency by proposing a training procedure called Negative Sampling (NS), or skip-gram negative sampling (SGNS).

Negative sampling is a simplification of Noise Contrastive Estimation (NCE), proposed by [9]. NCE makes the assumption that a model should be able to differentiate real data from noise. Thus, the full softmax is replaced by a handful of contrastive classes (known as candidates) for each real training example, and the model learns to differentiate the two. It has been shown in [9] that NCE approximately maximizes the log probability of the softmax. However, the word2vec algorithm is only interested in learning good word embeddings, so the softmax log probability maximization property of NCE is not needed. NCE needs the noise samples as well as the numerical probabilities of the noise distribution. Negative sampling only uses the noise samples, and not the distribution probabilities, gaining efficiency while still maintaining word embedding quality. Mathematically, the negative sampling objective function is as follows for a training sequence of words:

$$\log \sigma(v_{w_o}^{'T} v_{w_I}) + \sum_{i=1}^{k} E_{w_i \sim P_n(w)}(\log \sigma(v_{w_i}^{'T} v_{w_I})) \tag{2.4}$$

where k is the number of negative samples and $P_n(w)$ is the noise distribution. This equation replaces every $\log p(w_o|w_I)$ in the skip-gram objective function (equation 2.2). The noise distribution $P_n(w)$ varies in the literature. In the original paper that introduced negative sampling applied to word2vec [8], the authors use a uniform distribution, the distribution of word frequencies aka unigram distribution (thus a more common word is more frequently sampled), and a unigram distribution raised where each frequency is raised to the 3/4th power, finding the latter approach to work best.

2.4.2 Collaborative Filtering Neural Models

Most collaborative filtering based approaches that follow in this subsection attempt to replace matrix factorization with neural network architectures.

Prod2Vec/Item2Vec

Prod2vec was a recommendation algorithm proposed by [1]. Intuitively, prod2vec is word2vec applied to product recommendations: products are the "words", user buying sequences (or more generally, user-item interaction sequences, ie the sequences of songs a user listens to) are "sentences". Thus, the algorithm learns product embeddings: dense low-rank representations of each product. Then, the k-nearest neighbors algorithm can be used with these embeddings to find similar products to any arbitrary product (similar meaning the products tend to appear in the same context in user-item interactions). The authors of [1] use a full softmax during training.

A similar approach was proposed by [10], called Item2Vec. In this work, the word2vec algorithm is again applied to products, but in a skip-gram negative sampling setting. The authors also directly compare their proposed method with a Singular Value Decomposition of an item-item matrix. They find that Item2Vec performs better on recommendation metrics and is more efficient to train as well.

Indeed, [11] proved that the Skip-Gram Negative Sampling formulation of Word2Vec is implicitly factorizing a word-context Pointwise Mutual Information (PMI) matrix; that is, a matrix where each row is a word, each column is a context word, and each element in the matrix is the PMI value of the word and associated context. Pointwise Mutual Information is defined as follows:

$$pmi(w, c) = \log \frac{p(w, c)}{p(w)p(c)}$$

Intuitively, PMI quantifies the association of a word and context word. In the case of Item2Vec, this is equivalent to a PMI matrix for products in sequences. This result is interesting for the recommendation task, as decomposing a PMI matrix of products will result in latent factors for each product that are based on item-item co-occurrences.

Criteo has implemented Prod2Vec at scale to produce recommendations for partners. Currently, the use of the Prod2vec is being evaluated in offline tests and an AB test.

Other Neural Matrix Factorization Techniques

[12] introduces another neural network technique for matrix factorization of user-item interaction data to compute latent factors. This model has parallel user embeddings and item embeddings, and the dot product of these embeddings are treated as inputs to a non-linear hidden layer whose weights can be interpreted as the importance of latent factor dimensions. The output of the network is the probability of user-item interaction.

[13] improves upon [12] by directly using the user-item interaction matrix as input to a multi-layer neural network whose layers' output dimensions decrease proportional to their proximity to the output layer. There are two separate multi-layer networks: one that takes the high-dimensional sparse user vectors, and the other that takes the same for item vectors. The resulting latent factor outputs from the networks are used to compute the cosine similarity for an arbitrary user-item combination, which can be compared to the actual value in the user-item interaction matrix to compute the loss and subsequent gradients.

[14] combines probabilistic matrix factorization with marginalized denoising stacked auto-encoders. Marginalized denoising stacked auto-encoders are one layer neural networks that are optimized to reconstruct input data from random corruption. The input data is sparse user-item interaction matrices, and thus the auto-encoding provides dense, low-rank item or user vectors.

2.4.3 Content Based Neural Models

Most content based neural models involve learning latent factors for item features.

[15] uses deep convolutional networks to produce latent factors on audio input for music recommendation, beating state of the art audio recommendation techniques.

[16] is similar to [13] in that it uses a item neural network and a user neural network to learn latent factors for each, and then their respective outputs are combined in the last layer to model the predicted interaction value. However, this approach differs in that instead of vectors of the user-item interaction matrix as input, textual data from the user's reviews and review of the item are used as input. The textual data is sent through a embedding look up layer, then fed to convolutional layers in an effort to preserve the order of the words, as opposed to the bag-of-words approach. After a max-pool layer and dense layer, the resulting latent factors are combined to model the user-item prediction.

[17] uses Gated Recurrent Unit based RNNs to produce text content vectors for documents. The RNN approach preserves the ordering of words in the item, unlike bag-of-words approaches. This leads to state of the art results on the task of scientific paper recommendation.

2.4.4 Hybrid Neural Models

[18] uses a deep neural network to map tag-based user and item profiles to the same abstract vector space, uses negative sampling as the training procedure to increase efficiency. The architecture maximizes the similarity between each user and their relevant products and the opposite for the negative samples.

[19] uses a deep neural net architecture to combine information from user-item interactions and natural language processing. Namely, the item text is used to learn a user-specific content-based classifier, while user ratings are used to produce collaborative filtering recommendations.

[20] uses a neural matrix factorization method and integrates document modeling of text item data through a convolutional neural network. The authors argue that the convolutional neural network applied to text data gives more of a semantic understanding than the bag-of-words model, given that it preserves the ordering of words.

[21] improves upon the Prod2Vec architecture proposed in [1] by using product meta-data to regularize the model. Specifically, the probability of observing the meta-data of context products and ids of the input products (and vice-versa), as well as the probabilities of observing the meta-data of the context products with that of the input products, are jointly learned by the architecture, and the loss values of these tasks are added to the loss of the original architecture, acting as a regularizer on the model. The authors show this approach outperforms Prod2Vec on recommendation metrics (see section 4.1.4 for an overview of recommendation metrics).

Chapter 3

Proposed Method

3.1 Task

The problem I am looking to solve is creating a content-based recommendation system based on the product name. More generally, given a collection of items and their associated text (or categorical metadata) and a key item (historical item in Criteo terms), find the k most similar items to the key item. From a content-based recommendation standpoint, this requires building vectors for items based solely on their associated metadata, then using KNN to find similar products. As my internship task is to build a system that will recommend items with similar text meta-data and that will alleviate the cold-start problem that collaborative-filtering based solutions suffer from, a couple of constraints must be put on any proposed solution:

1. If two items have the exact same textual meta-data, their similarity value should be maximized. In general, the closer the textual meta-data is in terms of similarity, the higher the similarity value. This characteristic is important because it limits the systems to purely content-based solutions, as collaborative filtering approaches are already implemented at Criteo.

2. Another constraint that would be nice to have followed (but not necessarily required) would be that any model should be able to handle new products without retraining. This characteristic would allow the retraining frequency to be relaxed, decreasing required resources when used at scale. Also, since the approach is supposed to alleviate the cold-start problem, this characteristic would ensure that products can be recommended as soon as their product vector is computed using the trained model, as opposed to as soon as a new model is trained. This can be useful because building product vectors from a trained model is potentially much less computationally expensive than retraining a model.

The proposed approach, named *feature2vec*, uses user-item interaction sequences to build "sentences" of meta-data (such as words) by randomly sampling words from the meta-data of items in a context window. These synthetic sentences can then be fed to a skip-gram word2vec architecture to learn distributed representations, or embeddings, for all meta-data values. The important result of this is that item sequence data will be encoded into the meta-data vectors (word vectors, in Criteo's use case, but as we will see, it can be generalized to any sparse, categorical feature with reasonably high cardinality); that is, item sequences will be taken into account when learning the meta-data vectors. Once meta-data vectors have been learned, a weighted bag-of-words approach is used to build item vectors, then a KNN is used to find similar products to recommend.

3.2 Related Work

3.2.1 Building Document Vectors from Text

The content-based recommendation method I will propose makes use of building item-vectors based on their sparse meta-data, and will build upon related methods. The most common application (or thus most active research area) of sparse meta-data to vector representation is computing dense, low-rank vectors for text documents.

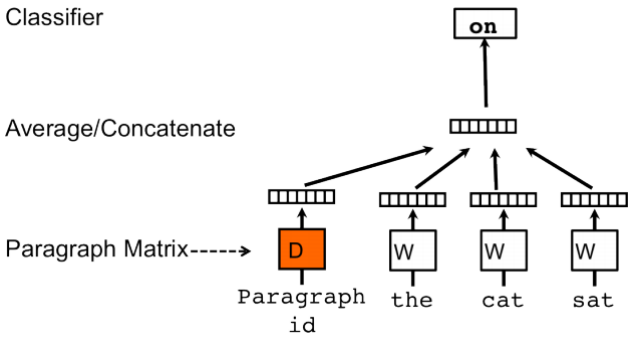

Figure 3.1: Paragraph2vec Architecture [24]

Latent Semantic Indexing

The state of the art for a long time was Latent Semantic Indexing (LSI), introduced by [22] in 1990. In the context of item text, this amounts to building a term-document matrix, where each row is a document/item, each column is a term/word, and each element in the matrix is the number of times the term occurs in the document. This matrix is often large and quite sparse. From here, a Singular Value Decomposition is performed on the term-document matrix, producing dense low-rank approximations of the term-document vectors for each document. Not only are these dense vectors easier to perform KNN on, as the dimensionality is greatly reduced, but also the vectors encode the the values of linearly independent components, or *factors*. These factors, in the application of text document representation, represent higher level concepts, an abstraction of simple term counts. The idea is that documents can talk about similar concepts without using all of the exact same words, and LSI will produce vectors for these documents such that their cosine similarity value will be relatively large.

Latent Dirichlet Allocation

A similar technique to produce low-rank representations of text documents is called Latent Dirichlet Allocation (LDA), introduced in 2003 by [23]. LDA is a generative probabilistic model of a corpus in which documents are represented as random mixtures over latent topics, which are distributions over words. These topic-document and word-topic distributions are assumed to have a sparse Dirichlet prior. Intuitively, a Dirichlet distribution is a probability distribution over a fixed number of categorical values, such as words or topics. The fact that it is sparse encodes the intuition that a document is only about a small set of topics, (similarly, a topic using only a small set of words frequently), as opposed to many topics. LDA is thus three-level hierarchical Bayesian model (words, topics, documents). In [23], the authors report improvements over the then state of the art in natural language processing tasks.

Neural Language Models

Just as word2vec [6] led to applications in the recommendation domain such as [1] and [10], it has been adapted to the application of computing vectors to collections of words such as sentences, paragraphs, or entire documents. The naive approach to building an item vector from meta-data vectors is the bag-of-words approach, which is to average the vectors of the meta-data to represent the item. An important draw-back of this approach that the following methods attempt to relieve is that the order of the meta-data (in this case the order of words in a sentence, paragraph, or document) is lost.

Paragraph2vec (also known as doc2vec) [24] extends word2vec to paragraphs, learning a dense feature vector for each variable-length collection of words in addition to the words themselves. The paragraph vector can be seen as additional context when predicting the next word in a paragraph, as the paragraph vectors are concatenated to the context words. This paragraph token can be seen as another word, it acts as memory of the topic of the paragraph and gives additional context when predicting the next word. The architecture is presented in figure 3.1. The authors show this approach achieves new state of the art results on the natural language processing tasks of

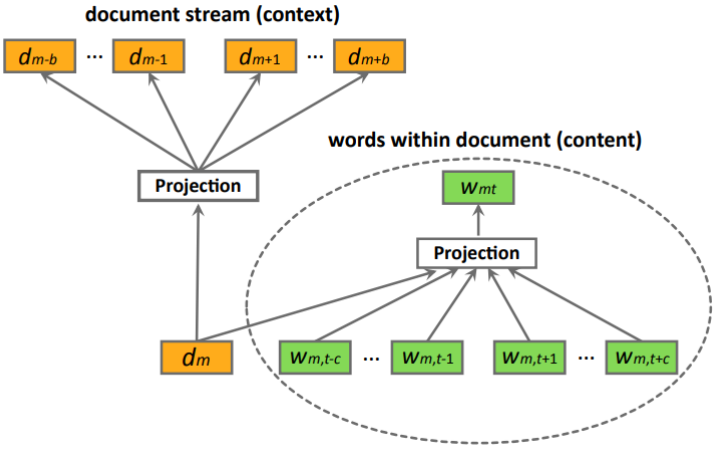

Figure 3.2: Hierarchical Neural Language Model Architecture [25]

sentiment analysis and text classification. To build a paragraph vector for a new paragraph, ie after the training of word/paragraph embeddings has finished, one needs to learn a paragraph vector using the words and corresponding word embeddings from the trained model. This is done through stochastic gradient descent, just as the embeddings for the original model are learned. In the case of text-based product recommendation, this amounts to learning either a new model or new paragraph vector every time a new item is added to a catalog. While not completely infeasible, it does require more computational resources and is not as flexible, as mentioned in constraint number 2 in the task section of this chapter. Furthermore, the paragraph embeddings learned in the paragraph2vec algorithm are in a different feature space than that of the word embeddings, unlike the bag-of-words approach. This makes it impossible to recommend documents to keywords or vice versa, which could be a nice feature for applications.

Hierarchical Neural Language Model [25] builds upon paragraph2vec [24], also learning paragraph or document vectors while learning word embeddings. However, [25] differs in that it also considers the context of a document in a document stream. It's architecture is presented in figure 3.2. In a recommendation setting, a document stream could be the sequence of items a user sees, sorted to be a chronological order. This is similar to prod2vec [1] and item2vec [10], but with adding learning of word embeddings. HNLM is quite similar to the proposed feature2vec approach in that it is merging prod2vec and word2vec to produce word embeddings. The difference is that in HNLM, the word sequences are from the documents themselves and document embeddings are learned from document sequences; where in feature2vec, the word sentences are formed from the document sequences, and document embeddings are formed using the bag-of-words approach. Indeed, HNLM document embeddings are more expressive than the bag-of-words approach in feature2vec. But as I will explain the following section, this result is not as important as it seems.

3.2.2 Content-Based Recommendation

An introduction to content-based recommendation systems as a whole as well as an overview deep learning content-based recommendation systems have been given in previous sections. Many state-of-the-art methods provide item recommendations on dense feature content, such as audio data ([15] [26] [27]) or image/video data ([28] [29] [30]). However, for the task of this internship, we are concerned with sparse, categorical features.

In the rest of this section, I will introduce methods that fulfill the task at hand; that is, building a recommendation system based on item text or other categorical features. The constraints on the system as presented in the beginning of this chapter must also be met. That is, items with the same meta-data must have a maximized similarity value (ie, the same item embedding), and no further model training must be done to accommodate new items. This means paragraph2vec [24] and the Hierarchical Neural Language Model (HNLM) [25] do not meet the constraints. While this may seem counter-intuitive, as these systems are learning higher order item representations using their sequences and meta-data, there is a simple explanation; as collaborative filtering algorithms

are already implemented at Criteo, using user-item interactions to differentiate the embeddings of two items with the same meta-data mitigates the potential gain of implementing a content-based recommendation algorithm in the first place, which is to provide recommendations that the collaborative filtering algorithms miss (such as new products, or other item similarities that have not been discovered by users and as such are not covered in previous user-item interactions). That is, the content-based algorithm must be fully content-based (constraint 1) and must not be retrained when new items appear (constraint 2). As HNLM violates constraint 1 (items with the same meta-data can have different embeddings) and paragraph2vec violates constraint 1 and 2 (item embeddings must be learned when a new item appears, as opposed to a bag-of-words approach, and are not in the same embedding space as the word embeddings), these approaches were not attempted or implemented for the task.

There are a few recommendation system approaches based on the categorical features of items that fit the task and constraints. One common method is to use LSI [22] or LDA [23] on an word-document matrix to produce embeddings for items before using a KNN to find similar items [31]. Instead of term counts in the matrix, term-frequency inverse-document-frequency (TF-IDF) values can be used. TF-IDF gives a value for the importance of a term (or in this case a meta-data value) for a specific document (or item). It is calculated by taking the term frequency of a document (how many times the term appears in the document) and multiplying it by the inverse document frequency (the total number of documents divided by the number of documents in which the term appears). Mathematically, this can be represented as follows:

$$tf(w, d) = f_{w,d} \tag{3.1}$$

$$idf(w, D) = \log \frac{N}{|\{d \in D : w \in d\}|} \tag{3.2}$$

$$tfidf(w, d, D) = tf(w, d) \cdot idf(w, D) \tag{3.3}$$

where w is the word or meta-data value, d is the document or item, D is the corpus or collection of all items/documents, N is the number of items/documents in the corpus. Indeed, building a large sparse vector for each document's tf-idf values over a feature's vocabulary can be another way to represent items. However, this approach has a couple drawbacks: first, as item vector dimensionality grows with the cardinality of the categorical feature, the resulting KNN will be inefficient and suffer from the curse of dimensionality; second, as similarity value between items will be based only on matching categorical values of the meta-data, this approach could possibly miss latent relationships between these values that are unveiled by LSI, LDA or word2vec approaches.

Another way to build content-based item embeddings is to use pre-trained word2vec (or another distributional word representation method, such as GloVe [32]) word vectors from a very large language corpus such as a Wikipedia dump and a bag-of-words approach. While computationally efficient, as only the task of bag-of-words building item vectors from their text is needed, this approach has some drawbacks that make it infeasible for the proposed task. First of all, using pre-trained word vectors predefines the vocabulary of the categorical feature, meaning the meta-data must be text, and must be in a language for which pre-trained embeddings are available (or must have large corpus, if you are training the embeddings yourself). This prevents the generalization to any categorical feature, such as item tags. Furthermore, at Criteo there are many different languages as partners are from around the world. Uploading and storing millions of vectors for each language is not memory efficient. Also, there may be millions of words in the pre-trained vector vocabulary that are simply never used in the product name context, leading to further inefficiency. This is why feature2vec learns the vocabulary from the items themselves.

3.3 Feature2Vec

Here, a new way to learn sparse categorical feature embeddings for recommendation, named feature2vec, is presented. Feature2vec uses item sequences to build "sentences" of meta-data (such as words) by randomly sampling words from the meta-data of items in a context window. Sentences are feed to a skip-gram formulation of word2vec to compute feature embeddings, which are used with a bag-of-words approach to build item embeddings. From item embeddings, KNN is used to find similar items for recommendations.

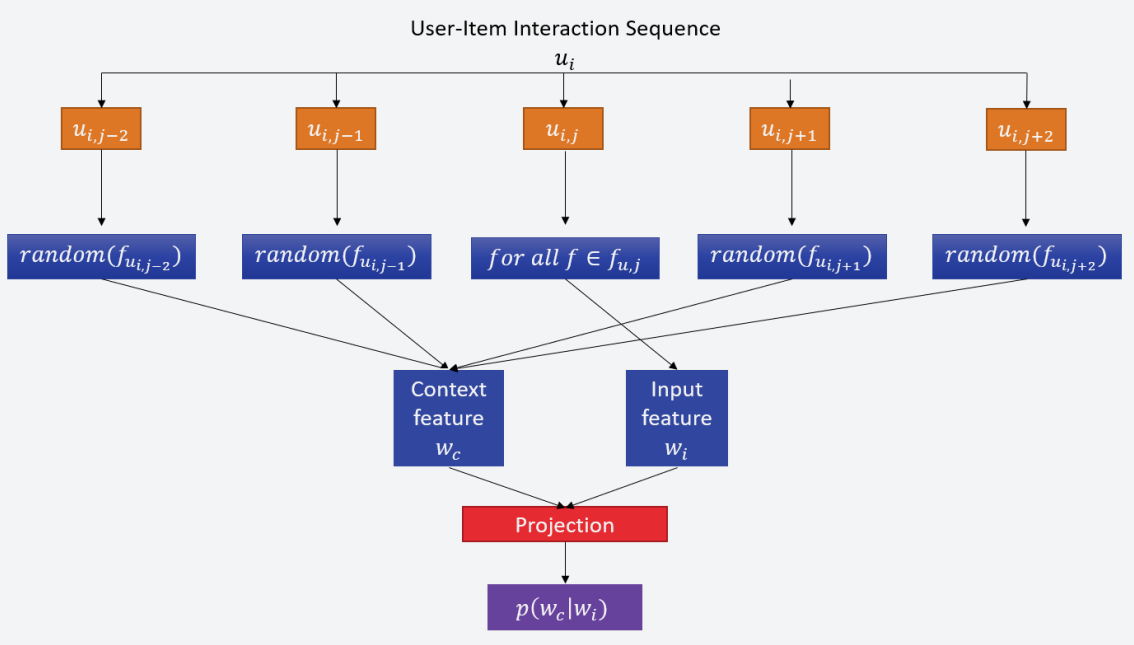

Figure 3.3: Feature2vec Architecture with $window\_size$=2

3.3.1 Notation

In order to understand and formalize the method, the following notation is introduced.

- u_i = variable-length sequence of item interactions for user i
- u_{ij} = jth item in ith user's sequence
- U = set of all user interaction sequences
- f_k = variable-length set of meta-data feature values that are associated with item k
- $random(f_k)$ = random selection of one element in the set f_k
- F = meta-data values for all items
- V = all possible meta-data values, also known as vocabulary
- $context(u_{ij}, window\_size) = (u_{ij-window\_size}, u_{ij-window\_size+1}, ..., u_{ij+window\_size1}, u_{ij+window\_size})$; sequence of context items for a specific item u_{ij} in sequence u_i; u_{ij} is not included in it's own context
- S = synthetic sequences of meta-data
- E_v = embedding vector for meta-data value v
- E_k = embedding vector for item k
- $window\_size$ = size of context window in sequences
- $embedding\_size$ = dimensionality of embedding vector

3.3.2 Feature2Vec General Case

The feature2vec architecture is shown in figure 3.3. From a user-item interaction sequence, the meta-data from the items are sampled to make meta-data sequences, which is fed into a word2vec skip-gram neural network.

The process for generating recommendations for users from item content and user-item subsequences is as follows:

1. Data input/preprocessing: F (meta-data associated with each item), and U (set of all user-item interaction sequences).
2. Generate synthetic meta-data sequences using Algorithm 1 (either before word2vec training or on the fly for each batch). An example of this sequence generation is given in figure 3.4.
3. Feed sequences into skip-gram word2vec architecture to compute meta-data embeddings (full softmax or negative sampling can be used, depending on application, implementation, and data).
4. Build item embeddings from meta-data embeddings using bag-of-words or weighted bag-of-words approach (Algorithm 2).
    - An optional argument is the *weights* associated to a meta-data value for each item. An bag-of-words approach without weights would assign each meta-data value a weight of 1.0, while a weight approach would be to weight each meta-data value v for each item k by it's $tfidf(v, k, F)$ value (defined in equation 3.3).
    - Notice that the item embeddings are normalized to have unit length. This is done because (1) items can have variable number of meta-data values (differing number of words) and (2) it allows for Euclidean distance use when computing KNN, as the ordering of nearest neighbors for normalized Euclidean distance is the same as cosine distance. The use of Euclidean distance allows for the use of data structures that speed up the KNN search, such as a Ball tree or k-d tree.
5. Use K-nearest-neighbors algorithm with cosine distance to find user recommendations, using user historical item as seed items. Implementation and historical item selection depend on application setting.

Algorithm 1: Generating meta-data sequences from user-item interaction sequences

Input: U, F, $window\_size$
Output: Sequences of Meta-Data Values S
S = [];
for u_i *in* U **do**
  for u_{ij} *in* u_i **do**
    for v *in* $f_{u_{ij}}$ **do**
      seq = [];
      for k *in* $(max(j - window\_size, 0), min(len(u_{ij}), j + window\_size))$ **do**
        if $k == j$ **then**
          seq.append(v);
        else
          seq.append($random(f_{u_{ik}})$);
        end
      end
      S.append(seq);
    end
  end
end
return S

Algorithm 2: Building item embeddings from meta-data embeddings using bag-of-words approach

Input: F, E_V, $embedding\_size$, $weights$ (optional: 1.0 if not defined)
Output: E_k for all items k
for f_k *in* F **do**
  $item\_vec$ = vector of zeros of dimensionality $embedding\_size$;
  for v *in* f_k **do**
    $item\_vec$ += E_v * $weights_{k,v}$
  end
end

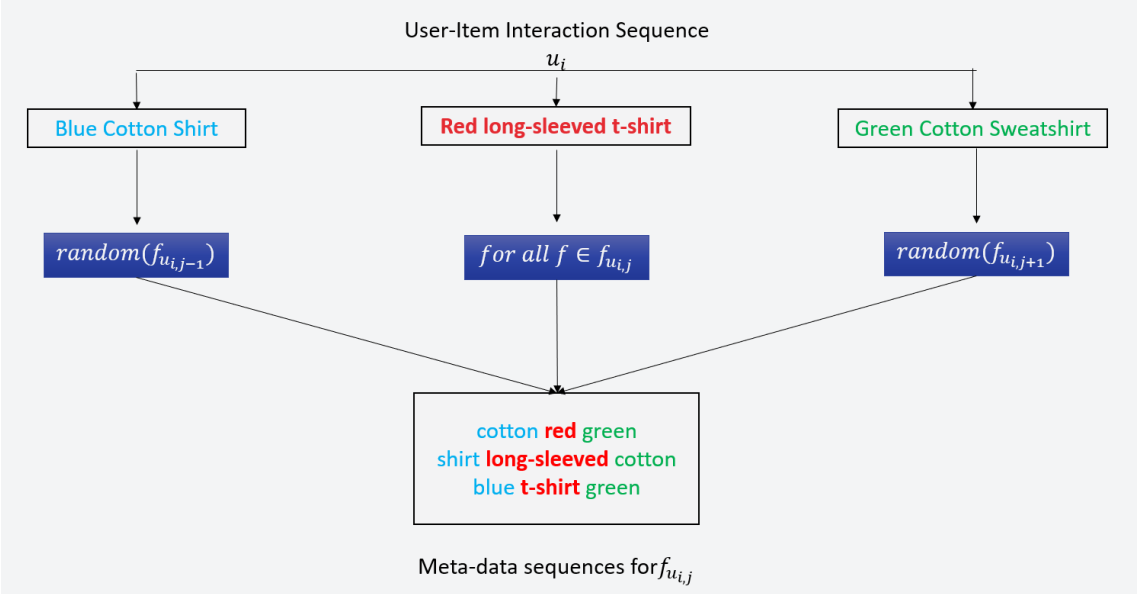

Figure 3.4: Example of feature2vec meta-data sequence generation

Objective Function

We can adapt the skip-gram word2vec objective function shown in equation 2.2 to the feature2vec setting. The feature2vec objective function for a user-item interaction sequence $u_i = (u_{i,1}, u_{i,2}, ..., u_{i,|u_i|-1}, u_{i,|u_i|})$ is as follows:

$$\frac{1}{|u_i|} \sum_{j=1}^{|u_i|} \sum_{v_I \in f_{u_{ij}}} \sum_{-c \leq k \leq c, k \neq 0} \log p(random(f_{u_{i,j+k}}) | v_I) \tag{3.4}$$

At the time of neural network training, the $random(f_{u_{i,j+k}})$ term is instantiated with a meta-data value.

We can also adapt the soft-max and negative sampling formulation of word2vec (equations 2.3 and 2.4, respectively) to the feature2vec setting. The soft-max version of $\log p(random(f_{u_{i,j+k}})|v_I)$ is defined as follows:

$$\frac{\exp x'^{T}_{random} x_{v_I}}{\sum_{v \in V} \exp x'^{T}_{v} x_{v_I}} \tag{3.5}$$

where $random = random(f_{u_{i,j+k}})$, x' is the output word embedding and x is the input word embedding representation. Similarly, we can adapt the negative sampling version of the probability of a input-context pair from equation 2.4 to the feature2vec setting. Thus, $\log p(random(f_{u_{i,j+k}})|v_I)$ from equation 3.4 is defined as:

$$\log \sigma(x'^{T}_{random} x_{v_I}) + \sum_{i=1}^{k} E_{v_i \sim P_{|V|}(v)} (\log \sigma(x'^{T}_{v_i} x_{v_I})) \tag{3.6}$$

3.3.3 Rationale

If we assume the the meta-data in question is words, the rationale behind this approach is that word embeddings are created that will take into account product sequences. This can be seen as an adaption of the original distributional hypothesis of linguistics introduced in [7] (which is, words that appear in similar contexts have similar meanings): words that are seen in the same context of user-item meta-data interactions have similar meanings. Basically, feature2vec is the distributional hypothesis applied to item meta-data for the task of item recommendation. Feature2vec is a system to learn embeddings in which the embeddings for two words (w_1, w_2 for example) will be close if the items that contain w_1, w_2 (we will call them I_{w_1} and I_{w_2}) are seen in the same user-item

context as items I_{w_x}, where $x \neq 1, 2$. Thus, if we assume the adapted distribution hypothesis to be true, words that have similar embedding values have similar meanings *in terms of user-item interactions*.

Another way to look at feature2vec is as follows: the knowledge a collaborative filtering algorithm learns from user-item interactions is abstracted a level up and encoded into meta-data value embeddings, which are then used to make item embeddings and to provide recommendations. In this sense, feature2vec is content-based in the sense that only item features are used when creating item embeddings and thus making recommendations (following constraint 1), but collaborative filtering information is "encoded" in the content-based recommendations, as user-item interactions are used to train the meta-data embeddings.

Feature2vec has nice advantages over similar methods. In contrast to using pre-trained word embeddings from a large language corpus, feature2vec computes embeddings for only the values that are in the vocabulary of the item meta-data. Also, it adapts to any language (or any sparse categorical feature; variable or fixed length). As mentioned in the previous paragraph, it learns feature embeddings that are adapted to the task of item recommendation by learning from sequences constructed by user-item interactions, unlike other document representation techniques such as LSI or LDA. Furthermore, feature2vec is completely content-based in the sense that once meta-data embeddings are learned, only the item's meta-data is used to build the item embeddings. Thus, two items with the same meta-data have the same embedding. This follows constraint 1 mentioned in the task section. Another result of the bag-of-words item embedding construction is that it follows constraint 2: once embeddings are trained, an item embedding can be efficiently built for any new item that appears in the catalog.

In the case where user-item interaction sequence data is relatively small, feature2vec still works because the sequences are "exploded" to create meta-data sequences. In the feature2vec formulation implemented in the experiments of the next chapter, the sequence generation algorithm (Algorithm 1) creates one meta-data sequence for each item meta-data feature, as shown in figure 3.4, in order to ensure proper representation of all user-item interactions and keep the data volume reasonable. However, if user-item interaction data is very sparse, the data can be exploded further so that all possible meta-data sequences from a user-item sequence are generated (reference to diagram). For example, if there exists 3 items in a user-item interaction sequence and each item has 3 tags associated with them, the algorithm will create 27 sequences per item, as shown in figure 3.5, instead of the usual 3 sequences that the normal sequence generation setting would create (figure 3.4). Granted, there would be much duplication in the sequences in this expanded setting, but if the user-item sequence was longer, the duplication would be less. The expanded sequence generation algorithm would be useful in the setting where there are very few user-item sequences to learn from.

3.3.4 Company Application

In the context of Criteo's product recommendation, feature2vec can be applied in the following way:

- Products in a partner's catalog have textual data such as the name of the product.
- Data is collected about sequences of user-item interactions, including product page views, adding items to basket, and purchase of item.
- At this point, we have U (user-item sequences) and F (meta-data for each item), which is all the data needed to implement the system.
- Word sequences are built from U and F and then fed into a word2vec skip-gram architecture to produce word embeddings.
- Word embeddings are used to build product embeddings, then similarity lists can be built using a KNN.

The implementation details and parameter choices are elaborated on in the next chapter.

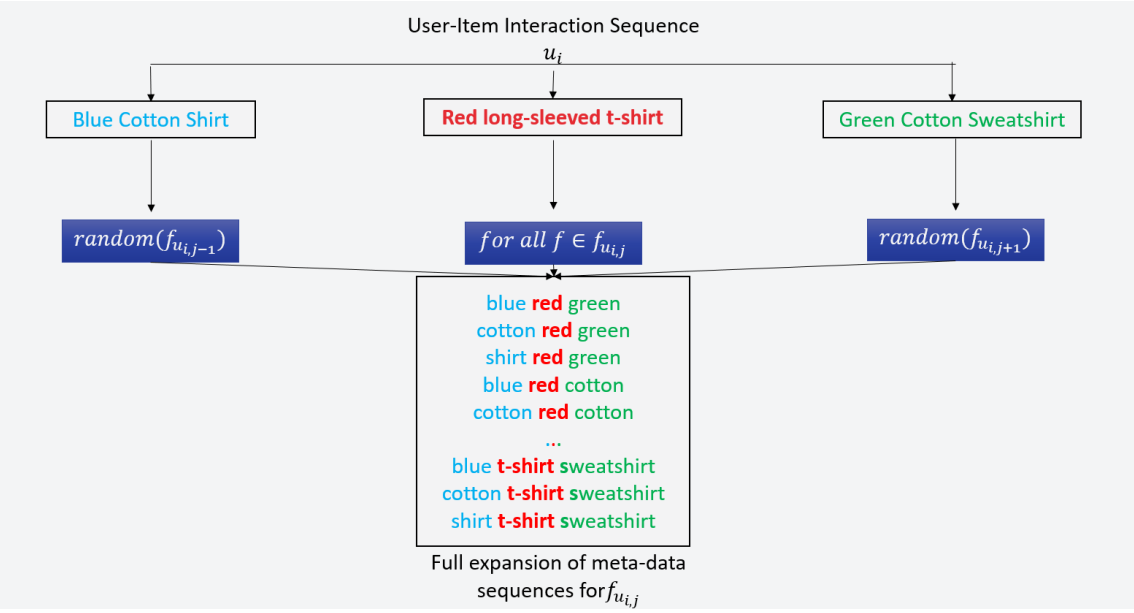

Figure 3.5: Example of feature2vec full explosion meta-data sequence generation. In this case, 27 meta-data sequences would be created for this input product instead of the usual 3.

Chapter 4

Experiments and Results

In this chapter, I present the experiments, implementation details, and results of feature2vec. I start with experiments on benchmark recommendation datasets, then move on to company experiments and implementation.

4.1 Tag Experiments

The first feature2vec experiments completed were on benchmark recommendation datasets. In these datasets, the meta-data used was not words but user-provided tags. Thus, each item has a variable length list of associated tags, and the feature2vec algorithm will learn embeddings for all possible tags, which will then be used to generate content-based recommendations.

4.1.1 Datasets

Two datasets were used these experiments, the MovieLens 20M dataset [33] and the KGRec music recommendation dataset [34].

MovieLens

The MovieLens 20M dataset includes data about movies and how users have rated them, along with user-generated tag associations and official genres for movies. The dataset includes 20,000,000 user-movie ratings from 138,000 users and 27,000 movies, with 465,000 tag associations from about 30,000 possible tags. As feature2vec learns from implicit user-item interactions (as opposed to explicit such as ratings), user-item sequences were built by throwing out the actual rating, grouping the interactions by user, and then sorting these interactions chronologically. In addition to user-generated tags, each movie has genre meta-data associated with it. These genres are treated as additional tags by the system.

KGRec

The KGRec dataset includes information about user-song interactions and song meta-data. Textual descriptions are given for each song, but upon inspection it is shown that they are too noisy to be of any use, as a lot of the time the text is about words in the title and do not actually give any information about the song itself. Thus, the tags are used as the feature in feature2vec. The dataset includes 750,000 implicit user-item interactions about 8,640 songs and 5,199 users. User-item interaction sequences are built by grouping by user, then chronologically sorting the interactions.

4.1.2 Baselines

The following methods are used as baselines to evaluate feature2vec's performance.

Latent Semantic Indexing

For these tag-based recommendation system experiments, the baseline will be latent semantic indexing, similar to [31]. A sparse item-tag tf-idf (equation 3.3) matrix is constructed in which the rows are the item, the columns are the tags, and the elements are the tf-idf values. From here, the matrix is decomposed using Singular Value Decomposition, producing dense low-rank vectors for each item. These vectors are used to find similar items and to recommend items to users.

Item2Vec

Another baseline implemented is Item2Vec [10], which is Prod2Vec [1] with a skip-gram negative-sampling architecture. This is a collaborative-filtering based method, not a content-based method. The rationale behind it's inclusion in these experiments is that it will demonstrate the cold-start problem and subsequent need for content-based recommendation methods. Indeed, the performance of Item2vec is expected to be much higher in general than content-based systems, as it is taking into account richer user-item interaction data, but is also expected to perform worse in a cold-start setting (more on this in the metrics section).

Item2Vec is implemented by treating user-item subsequences as "sentences", and items as "words", and feeding these subsequences into a word2vec skip-gram negative sampling architecture to produce item embeddings.

Tag-Matching

A naive baseline of tag-matching is implemented for the MovieLens dataset. A sparse item-tag matrix is constructed just as is done for the Latent Semantic Indexing method, except for tag-matching there is no matrix decomposition, and the sparse vectors are used as the item vectors to find similar items. It is called tag-matching because items will only have a non-minimal similarity value if they share at least one tag association.

4.1.3 Feature2Vec Application

In these experiments, feature2vec is applied to tags in user-item interaction sequences. For each tag in each item of a user-item sequence, a sentence of length $(window\_size * 2) + 1$ is created and fed in to a skip-gram negative sampling word2vec architecture. That is, tags from the sequences are seen as positive examples, and noise words are negative examples. The sequence exploding is done on the fly in batches and the feature2vec (and item2vec) embeddings are trained in Tensorflow.

4.1.4 Experimental Setup

Task

For these experiments, we evaluate the methods on the task of next event prediction, in a setup similar to [21]. The first $(n-2)$ elements time ordered user-item interaction sequences are used to train the models and the $(n-1)$-th element is used to tune hyper-parameters, then the models are trained on the first $(n-1)$ elements of the user-item sequences, and results are reported on how well they predict the nth element of each sequence. Since the tags are generated by the users themselves, if a user has tagged their test event, this tag is not considered part of the item's bag-of-tags when building the item embedding (only tags from the training set are used to build item embeddings).

Once item embeddings are built according to each method, the last item in the each user's training sequence is used as the query item to find similar items in order to predict the next item.

Hyper-Parameters

Hyper-parameters were tuned on the validation element of user-item sequences. I will now give an overview of the hyper-parameters tuned for each method. In the results section, I present the best-performing hyper-parameter settings on the validation items for each method, as these are the settings used on the test set.

For LSI, the number of singular values was tuned (equivalent to $embedding\_size$ for the 2vec methods). Also, two different sparse item-tag matrices were tested: term counts and tf-idf values.

Table 4.1: Results and optimal hyper-parameters on the KGRec dataset, 5,199 test recommendation lists.

Method	emb_size	window	neg_samp	matrix/weights	HR@10	nDCG@10	CSHR@10	CSnDCG@10
LSI	50			tfidf	0.001597	0.008803	0.001558	0.008679
F2V	100	2	20	tfidf	0.009581	0.05442	**0.003309**	**0.016322**
I2V	50	3	20		**0.014025**	**0.067112**	0.001115	0.005695

Table 4.2: Results and optimal hyper-parameters on the MovieLens dataset, with 138,000 test recommendation lists.

Method	emb_size	window	neg_samp	matrix/weights	HR@10	nDCG@10	CSHR@10	CSnDCG@10
LSI	100			tfidf	0.002158	0.014446	2.82E-05	0.000134
Tag-Matching					0.00211	0.015193	2.09E-05	0.000111
F2V	200	2	20	uniform	0.003762	0.024238	**6.50E-05**	**0.000309**
I2V	100	3	20		**0.007642**	**0.043283**	3.10E-05	0.000135

For item2vec and feature2vec: *embedding_size*, *window_size*, and the number of negative samples per true pair were tuned on the validation set.

Also, for feature2vec, when bag-of-words building the item embeddings from tag embeddings, two types of weighting are used when adding the tag embeddings for an item: uniform (all tags have the same weight) and tf-idf (tags are weighted by it's tf-idf value for that user). Once the tag embeddings are summed for an item, the item vector is normalized to have unit length.

Metrics

I use the following metrics, averaged over all users, to evaluate the methods:

- **HR@10**: Hit rate at 10, equal to 1/10 if the test item appears in the top-10 list of recommended items.
- **nDCG@10**: Normalized discounted cumulative gain at 10, favors higher ranks of the test item in the top-10 list of recommended items.

The preceding metrics are also computed for items in the *cold start setting*, similar to [21]. This is a subset of the test items in which the *(query item, test item)* pair has zero co-occurances in the training sequences. These metrics are denoted as **CSHR@10** and **CSnDCG@10**.

Other recommendation metrics used in recommendation system literature depending on the application include precision@k, recall@k and area under the curve. Precision@k is the proportion of the k recommendations that are "hits" (items in a user's test set, for example), recall@k is the proportion of hits that are in the k recommendation, and area under the ROC curve, which is a plot of the true-positive rate against the false-positive rate.

4.1.5 Results

KGRec

The results and optimal hyper-parameters are shown in table 4.1. As expected, item2vec has the best recommendation performance in the general (non-cold start) setting. However, feature2vec has significantly better results in the cold-start setting, with about a 3x larger HR and nDCG in this setting. LSI also outperforms item2vec in cold-start metrics, but feature2vec greatly outperforms LSI as well.

MovieLens

The results for the MovieLens experiments are shown in table 4.2. Similar to the KGRec experiments, item2vec performs best outside of the cold-start setting, while feature2vec performs best in the cold-start setting. Surprisingly, item2vec performs better than the LSI and tag-matching in the cold-start setting.

Table 4.3: Example of song tags and their nearest neighbors learned by feature2vec

key_tag	1980s	Swing-Jazz	Disney	Sentimental	folk-rock
nearest neighbors	80s-Pop	minnie-the-moocher	camryn	Ballad	folk
	80s	scat	pop-music-tag	sexy	Mellow
	classics	Drobna	Selena-Gomez	ballads	acoustic
	oldies	genrelessimo	Female-Artist	love-song	indie-folk
	lovely	1940s	i-love-being-a-girl	love-songs	beautiful

Analysis

The experiments on the KGRec and MovieLens datasets show that feature2vec is a valid approach to computing meta-data embeddings for content-based recommendations. Feature2vec gives a 3x improvement in the cold-start setting over the collaborative filtering-based item2vec approach in the KGRec dataset, and a 2x improvement in the cold-start setting over item2vec in the MovieLens dataset. Feature2vec also significantly outperforms LSI for building item embeddings from meta-data.

4.1.6 Feature2vec Embedding Analysis

In this section, I present some interesting tag embedding results by showing some key tags and their nearest neighbors in the tag embedding space in order to have a qualitative analysis of the tag embeddings learned by feature2vec.

KGRec Song Tags

Examples of song tags and their nearest neighbors in the embedding space learned by feature2vec are shown in table 4.3. As seen in the table, feature2vec was able to learn meaningful embeddings for the example tags, as almost every nearest neighbor directly relates to it's key song tag. For "1980s", the two nearest neighbors are also 80s related. For "Swing-jazz", the two nearest neighbors are very closely related as well, as "minnie-the-moocher" is the name of a swing-jazz song, and "scat" is a style of swing-jazz singing. The "Disney" tag's nearest neighbors are related to the fact that a few pop singers got their start as Disney show actors, such as Selena Gomez. "Sentimental"'s nearest neighbors are all closely related to it, as is the same with "folk-rock".

MovieLens Movie Tags

Examples of movie tags and their nearest neighbors in the embedding space learned by feature2vec are shown in table 4.4. As seen from the table, some of the nearest neighbors make intuitive sense, such as "western" => (American civil war, southern united states, time machine), "disney animated feature" => (animal movie, computer animation), "love triangle" => (death of a spouse), and "war crimes" => (child soldiers, native exploitation). Other tags don't provide a higher level meaning to the movie, or are arbitrary and/or subjective, so their associations are hard to evaluate, such as "seen 2010", "notable cast", "unlikely hero", "good music", and "better than everyone thinks". Other tag nearest neighbors are counter-intuitive, like "sad" => (slapstick humor, classic comedy). However, there can be an explanation to these seemingly poor quality embeddings. For example, "sad" and "slapstick humor" are close in the embedding space, meaning the movies that are seen in the same context as movies that have these two tags have similar tags. One could argue that users are likely to watch comedy movies in the same context (ie in the same sequence), and the same could be said for sad movies. However, because watching too many sad movies in a row can be depressing, thus it is also possible that users watching sad movies might also want to watch comedies in the same context as well. This can explain the fact that the "sad" and "slapstick humor" have similar tag embeddings. This shows that feature2vec can embed tags in the context of movie sequences.

4.1.7 Conclusion

The experiments presented in this section show the effectiveness of feature2vec, as meaningful embeddings are learned for a noisy categorical feature. The tag embeddings prove to be useful in the recommendation task when paired with a bag-of-words approach to building item embeddings.

Table 4.4: Example of movie tags and their nearest neighbors learned by feature2vec

key_tag	western	disney animated feature	love triangle	sad	war crimes
nearest neighbors	american civil war	seen 2010	death of a spouse	claire danes	child soldiers
	time machine	notable cast	baltimore	alter ego	native exploitation
	good music	videogame	predictable	slapstick humor	unlikely hero
	nihilism	animal movie	drug use	better than everyone thinks	hd
	southern united states	computer animation	oscar nominee: original screenplay	classic comedy	not as good as the original

This is shown in the significant improvements to recommendation performance in the cold-start setting over collaborative filtering algorithms as well as other content based approaches.

4.2 Feature2vec at Criteo

In this section, the implementation of feature2vec at Criteo is described, as well as the implementation of metrics and experimental results.

4.2.1 Dataset

While the last section described feature2vec as applied to tags in movie and music recommendation, it is applied to product text at Criteo. Meaning, for each partner, word embeddings are learned for all words that appear in a product's meta-data. At this point, I only consider the name of the product as it's textual meta-data. Also, a vast amount of user-item sequence data is collected every day at Criteo, which will be used to generate word sequences using the algorithm described in algorithm 1. As the sequences are "exploded", the user-item sequences must be sub-sampled or else the resulting word sequence data will be too large.

There is an difference between product name and item tags: item tags are an un-ordered set, while product names are ordered. However, as most product names are just a few words long, this loss of information when using a (possibly weighted) bag-of-words approach is not important.

Any implementation of feature2vec for product recommendations must scale, as there are over 15,000 partners in the Europe and North American platforms. The number of products in each partner's catalog ranges from a few hundred to millions.

4.2.2 Feature2Vec Implementation

As mentioned in the chapter about Criteo's engine, Criteo uses Hadoop clusters to compute the offline similarity list creation jobs. Thus, feature2vec is implemented in Scala using Spark. Spark's machine learning library, MLlib, contains a word2vec training algorithm out of the box, so all that is needed to do is create the synthetic word sequences, feed them to Spark's word2vec algorithm, use the word embeddings to build item embeddings, then perform a KNN to build the product similarity lists.

I briefly explain each component implementation below.

Creating Word Sequences

First, the catalog data must be loaded for the partner, to access the textual data for each product. The products are assigned an id to differentiate them. Then, the user-product interaction data loaded as well. There already existed functions to load both kinds of data into Spark Resilient Distributed Datasets (RDDs), which is the main data structure used in Spark. Spark is used on top of Hadoop Distributed File System (HDFS). As computation is distributed on a cluster, it can handle large amounts of data. The user-product sequence data is in the form of sequences of events. Events contain a partner id, user id, product id or ids, and event type, among other data. The main event types are product page view, adding a product to a basket, or buying a product/products. As we are only concerned with building similarity lists, the only user-product events I use when building synthetic word sequences are page views, as it is not common for users to buy many products that are very similar, but it is common for user's to browse similar products before selecting and buying the one they want. Once other event types have been filtered out, the events are stored in a key-value RDD with the key being the product id, and the value being the event data. The product text can then be joined with the events on the product id. From here, the events are grouped by user id and then sorted chronologically. Finally, the synthetic word sequences

are created using algorithm 1, the fed to Spark's word2vec algorithm. One important characteristic of the Spark word2vec implementation is that it uses a full softmax instead of negative sampling.

From Word Embeddings to Item Embeddings

Once word embeddings have been trained, product embeddings are built using a weighted bag-of-words approach. Word embeddings are weighted by their product tf-idf value (Tf-Idf also comes in Spark's MLlib for free), summed for each product, normalized to have unit length, then saved to HDFS.

K-Nearest Neighbors Implementation

Once product embeddings ave been computed, a KNN is used to find the similarity lists. Cosine distance is the metric used to find the closest neighbors for each product. However, since the process needs to scale, a brute force approach to nearest neighbors is infeasible. Thus, a spark fast approximate nearest neighbors library [1] is used to speed up the search. This library utilizes Locality Sensitive Hashing (LSH) to hash similar products to the same buckets with high probability, so that the search for nearest neighbors can be confined to the buckets as opposed to the entire dataset.

4.2.3 Baselines

I implemented LSI [22] and LDA [23] for building product embeddings based on their text data. Once product embeddings have been built using these methods, the same KNN job as mentioned above can be used to find similar products.

The LSI implementation builds a large sparse tf-idf matrix and uses Spark's MLlib implementation of Singular Value Decomposition to decompose the matrix and produce low-rank embeddings for each product based on their text. Similarly, the LDA implementation uses the Spark MLlib implementation to compute the topic vectors for each product, then the same scalable approximate nearest neighbors algorithm is used to find recommendations.

I implemented tf-idf product embeddings at Criteo as well. This means the large, sparse word vector (dimensionality = word vocabulary size) of each product is inputted directly into the KNN to find recommendations. However, for partners with large catalogs, the KNN job never finishes, as the vocabulary and resulting dimensionality of product vectors is too large. Thus, I did not include the results of tf-idf vector-based similarity list evaluations in this section.

4.2.4 Metrics

As stated in the introduction to this report, part of my internship task was to design and implement metrics for evaluating the sources, or off-line similarity lists, used in the recommendation pipeline. The rationale is that if I develop a text-based source, there should be ways to evaluate it (1) against different text-based sources, (2) against collaborative filtering sources already implemented. I designed a couple metrics to evaluate and understand product-similarity lists of the partner level.

Hit Rate and nDCG @ k

If we treat the source similarity lists as recommendations, with their key product taking the role of historical product in the recommendation task, we can evaluate how well similarity lists predict the next product page view.

The process is as follows:

- Load a user-product event sequence (only page views for the same reason as mentioned earlier).
- Break sequences into $(keyproduct, nextproduct)$ pairs.
- Join the similarity lists (or first k items in each similarity list) on the $(key, next)$ pairs.
- Check if/where the $next$ product is contained in the key product's similarity list.
- Compute HR and nDCG (defined in previous section) for each pair and average over all pairs.

[1] https://github.com/linkedin/scanns

Note that since we are using the similarity list to predict only the next event, the maximal hit rate is $1/k$.

I implemented this process in Scala/Spark. It can generalize to user-products sequences over any period of time for any source's similarity list. As of now, it only computes for one partner at a time, but in the future it can be extended to multiple partners to better understand the source behavior.

Note that HR and nDCG for next event prediction are not necessarily the metrics that the sources want to optimize on, as the similarity lists are not used in this way, but it does give a quantifiable value for how well the sources perform in this related task.

Universal Category Analysis

One project within Criteo is to provide/predict cross-partner categories for products. Thus, a phone being sold on one partner's website will have the same category as a phone being sold on another partner's website. These are called *universal categories*. I use these categories in the following way:

- Given a partner catalog (with universal categories for each product), and a similarity list, join each product with its universal category.
- Compute how many of a key product's similarity list are considered the same universal category as that of the key product.
- Average this ratio across all similarity lists in a partner source.

This metric gives an idea of how similar the products in a similarity list are to the key product by only looking at the universal category of the products. Similar to hit rate and nDCG, this metric was implemented in Scala/Spark. It can computed the metric for multiple partners in parallel, but the results are not merged in the job (as in, the ratios are reported per partner once they are computed).

While useful for quantifying how similar a similarity list is, it is quite noisy for the following reasons:

- As mentioned above, universal categories are predicted for each product, meaning they are sometimes wrong.
- There are different levels of universal category, ranging from coarse-grained (ie. electronics) to fine-grained (ie. iPhone charger). The categories can be seen as a tree: higher in the tree are more coarse grained categories, and their descendants get more and more fine-grained. Each product has a leaf id; that is, the most fine-grained universal category for that product. Thus, two items can be very similar, (eg. iPhone charger and Mac charger) but will be considered different categories by the metric. Meaning, according to this metric, an iPhone charger and Mac charger are considered the same level of similarity (None or 0) as an iPhone charger and a couch, for example.

Again, this metric is not something that should be optimized on for the reasons listed above, but it can still give an overall idea of how similar a source's similarity lists are to their key products.

Prod2Vec Embeddings Analysis

As mentioned in the overview of the recommendation pipeline at Criteo in an earlier chapter, embeddings are computed for products using the algorithm proposed by [1], giving similar embeddings for products that are interacted with by users in the same contexts. In the enriched catalog job described in the RecoComputer overview, product embeddings are joined with partner catalogs, meaning each product's embedding is contained along with its partner id and product id. These embeddings can be used to evaluate similarity lists in the following way:

- Join embeddings and similarity lists on product ids.
- For each similar product in a key product's similarity list, compute the cosine similarity between the key product's embedding and the similar product's embedding, and emit the value.

Table 4.5: Hit rate and nDCG for next event prediction evaluation of text-based similarity sources, for European (EU) and North American (NA) partners.

Method	Partner	Test Set Size	HR@10	nDCG@10	Partner	Test Set Size	HR@10	nDCG@10
LSI	EU	164489	0.0028073	0.01266	NA	15168	0.001011642	0.0050407
LDA	EU	164489	0.0014085	0.006337	NA	15168	0.00184254	0.0091615
F2V	EU	164489	**0.0054762**	**0.024779**	NA	15168	**0.0035655**	**0.016226**

Table 4.6: Key-similar product universal category agreement ratios of text-based similarity sources, for European (EU) and North American (NA) partners.

Method	Partner	Number of Pairs	UC Agree Rate	Partner	Number of Pairs	UC Agree Rate
LSI	EU	8457070	0.641477	NA	1383657	**0.727708**
LDA	EU	4026043	0.528322	NA	697364	0.645169
F2V	EU	8552411	**0.650261**	NA	1389172	0.679167

- Given the similarity values for the key-similar product pairs over all similarity lists, one can visualize the distribution using bucketizing and a histogram. Comparing the distributions between sources gives an idea of how much they agree with the Prod2vec embeddings.

As with the other metrics, these cosine similarity distributions can be computed for multiple partners in parallel, but their results are reported separately.

4.2.5 Results/Embedding Analysis

I experiment with the text-based similarity sources generated by feature2vec and the baselines on a partner from the European platform and a partner from the North American Platform. These mid-sized partners have around 560,000 and 690,000 products in their catalog, respectively. I denote these partners as EU for European and NA for North American. I use the metrics above to evaluate each method's similarity lists for both partners.

HR and nDCG @ 10

For feature2vec, I use user-item interactions from a specific time period. Then, I use user-item interactions from a time period in the week after the time period used in the feature2vec training. Thus, I don't use any test set data when building the feature2vec embeddings.

The results of these experiments are shown in table 4.5. Feature2vec has the highest HR and nDCG for both partners, significantly outperforming the other methods in next event prediction. Note that the best possible HR@10 is 0.1.

Universal Category Analysis

The similarity lists of the three methods are analyzed by the ratio of key-similar product pairs whose universal category is the same. The results of these experiments are shown in table 4.6. Feature2vec outperforms LSI and LDA for the European partner, while LSI performs the best for the north American partner.

Prod2Vec Embeddings Analysis

The distributions of cosine similarity values between the prod2vec embeddings of every (key, similar) product pair in a source's similarity lists for both partners in shown in figure 4.1 and figure 4.2.

The closer the distributions are to a 1.0 similarity value to the right of the x-axis, the more they align with the prod2vec sources, which are built from user-item interaction data. As seen from the figures, the feature2vec similarity lists provide (key product, similar product) pairs that overall have higher prod2vec similarity values, meaning their pairs are more aligned with the user-item interaction data. While this is not exactly what is needed from a content-based source (indeed, the content-based source is supposed to complement the collaborative filtering sources as opposed to recreate them), feature2vec also follows the constraints that were outlined when introducing the task. This means that in addition to providing recommendations that are more relevant in the context of user-item interactions than the baselines (as shown by the figures in this sub-section),

feature2vec also provides recommendations that are content-based in the sense that only product text is used when building the product embeddings from the trained word embeddings.

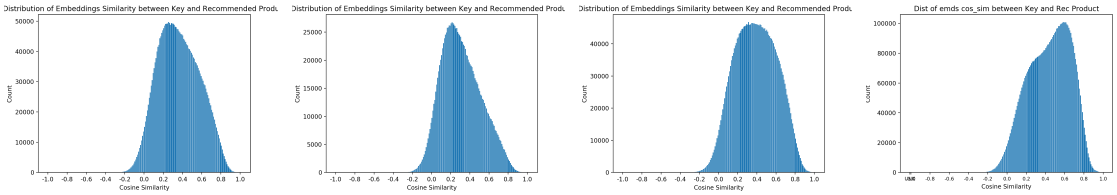

Figure 4.1: Cosine similarity distributions of Prod2Vec embeddings for (key product, similar product) pairs from LSI (left), LDA (middle-left), F2V (middle-right) and reference Prod2vec (right) sources on the European partner. Note that the distributions of F2V recommendations seems to be skewed closer to higher similarity values than the other text sources. The figure on the right shows the distribution if selecting recommendations only on the highest Prod2vec embedding similarity.

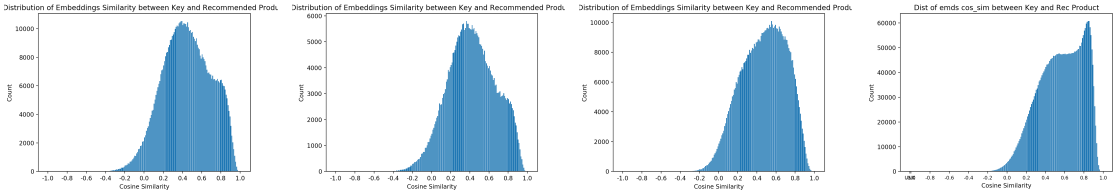

Figure 4.2: Cosine similarity distributions of Prod2Vec embeddings for (key product, similar product) pairs from LSI (left), LDA (middle-left), F2V (middle-right) and reference Prod2vec (right) sources on the North American partner. The same pattern of F2V being more skewed to higher prod2vec embedding similarity values than the other text source is present in this partner as well.

4.3 Scalability and Possible Extension

The feature2vec formulation used in the experiments of this section trains word embeddings for a partner at a time, meaning a new model must be trained for each partner. This feature2vec formulation is arguably still more scalable than LSI and LDA, because once word embeddings are trained, they can be used to build product embeddings for any new product that is added to the catalog of a partner, and thus models can be trained less often or simply fine-tuned to take into account recent user-item interactions. LSI and LDA, however, must train new models every time a new product is added to a catalog.

In terms of the model training itself (irrespective to training frequency), computation time LSI and LDA models scale linearly with the number of products in a catalog. In the case of LSI, a large sparse term-document matrix must be decomposed using Singular Value Decomposition, which makes the computation of product embeddings for partners with very large catalogs slow or even infeasible. Similarly, LDA uses Bayesian Inference to learn the word/topic/document distributions, whose computational complexity grows linearly with the number of documents (which are products, in this case). In the case of feature2vec however, the time complexity only grows with the catalog size when *building item embeddings from the trained word embeddings*, a much less computationally intensive, embarrassingly parallel process. That is, the word embedding training uses similar amounts of user-item interaction data for partners of all sizes (a very large amount of this data is collected at Criteo, so it's available; also, only a very small subset of this data (only a couple hours for partners with a lot a traffic) is needed as the sequences are exploded on their words). This, combined with the fact that feature2vec word embeddings can be trained much less frequently than LSI and LDA model training, as only the relatively cheap process of item embedding building from word embeddings needs to be run frequently, makes feature2vec more scalable than LSI and LDA.

However, we can do better for the scalability of feature2vec. Taking advantage of a Criteo process which groups user-interactions by user, across all partners of a platform, called *timelines*,

we can train feature2vec on platform-wide sequences, producing word embeddings that are cross-partner. In this sense, all products in a platform will have feature2vec product embeddings in the same embedding space. This drastically decreases the number of feature2vec word embedding training jobs needed, from around 15,000 (1 per partner) to just 2 (Europe and North American platforms). The fact that products from partners in the same platform are embedding in the same space also adds to the possible applications of the feature2vec embeddings. Remember that one of Criteo's products, called prospecting, shows ads to users for websites they have not visited yet. If products are embedding in the same embeddings space cross-partner, this enables product recommendations for a partner that a user has not seen yet. I have implemented cross-partner feature2vec embeddings at Criteo, but currently the embeddings need to be fully evaluated and possibly debugged.

4.3.1 Conclusion

The results in this section show feature2vec is an interesting alternative for text-based recommendation at Criteo. User-product page view sequences are used to produce word sequences, whose embeddings are learned and used to build item embeddings. Feature2vec word embeddings are learning collaborative filtering information, as shown in the prod2vec similarity value distribution figures. However, feature2vec is still content-based as it follows constraint 1 and recommends items in the same category at the same rate as pure content-based approaches LSI and LDA. The result of these characteristics is a system which builds product embeddings based on the product's words, just like LSI and LDA, but produces similarity lists that are better than LSI and LDA at predicting the next product a user will view, as seen in the HR and nDCG experiments. Furthermore, feature2vec is more scalable than other text-based similarity lists.

Chapter 5

Conclusion

In this report, I have introduced a novel categorical feature-based recommendation algorithm. First, embeddings for the values of the variable-length or fixed-length categorical feature are trained using a word2vec architecture on synthetic meta-data sequences built from user-item interaction sequences. Then, item embeddings are built using a bag-of-words approach. The bag-of-words approach works nicely for unordered categorical features (such as tags) or relatively short ordered categorical features (such as item name); also, it preserves the content-based characteristic of the system, as only the item meta-data is used to build item embeddings from the trained embeddings. I show that the performance of feature2vec on recommendation tasks is better than other content-based recommendation systems that utilize Latent Semantic Indexing or Topic Modeling to product item vectors. I also show that feature2vec is learning meaningful relationships between meta-data values.

At Criteo, feature2vec produces similarity sources that perform the same or better than baseline text-based similarity sources on the implemented metrics, while being more scalable. I also present a possible extension to make feature2vec more scalable still and to add a use case to the learned embeddings.

5.1 Acknowledgements

Many thanks to the entire Recommendation team at Criteo and especially Rafael Renaudin-Avino, without whom this project would not be possible.

Bibliography

[1] M. Grbovic, V. Radosavljevic, N. Djuric, N. Bhamidipati, J. Savla, V. Bhagwan, and D. Sharp, "E-commerce in your inbox: Product recommendations at scale," *CoRR*, vol. abs/1606.07154, 2016. [Online]. Available: http://arxiv.org/abs/1606.07154

[2] S. Sieranoja, "High dimensional knn-graph construction using space filling curves," Master's thesis, UNIVERSITY OF EASTERN FINLAND, 2015.

[3] F. Ricci, L. Rokach, and B. Shapira, *Recommender Systems Handbook*. Springer US, 2015.

[4] A. Singhal, P. Sinha, and R. Pant, "Use of deep learning in modern recommendation system: A summary of recent works," *CoRR*, vol. abs/1712.07525, 2017. [Online]. Available: http://arxiv.org/abs/1712.07525

[5] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944966

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[7] Z. Harris, "Distributional structure," *Word*, vol. 10, no. 23, pp. 146–162, 1954.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119. [Online]. Available: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[9] M. U. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," *J. Mach. Learn. Res.*, vol. 13, no. 1, pp. 307–361, Feb. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2503308.2188396

[10] O. Barkan and N. Koenigstein, "Item2vec: Neural item embedding for collaborative filtering," *CoRR*, vol. abs/1603.04259, 2016. [Online]. Available: http://arxiv.org/abs/1603.04259

[11] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2177–2185. [Online]. Available: http://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization.pdf

[12] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T. Chua, "Neural collaborative filtering," *CoRR*, vol. abs/1708.05031, 2017. [Online]. Available: http://arxiv.org/abs/1708.05031

[13] H.-J. Xue, X.-Y. Dai, J. Zhang, S. Huang, and J. Chen, "Deep matrix factorization models for recommender systems," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, pp. 3203–3209. [Online]. Available: http://dl.acm.org/citation.cfm?id=3172077.3172336

[14] S. Li, J. Kawale, and Y. Fu, "Deep collaborative filtering via marginalized denoising auto-encoder," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 811–820. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806527

[15] A. van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 2643–2651. [Online]. Available: http://papers.nips.cc/paper/5004-deep-content-based-music-recommendation.pdf

[16] L. Zheng, V. Noroozi, and P. S. Yu, "Joint deep modeling of users and items using reviews for recommendation," *CoRR*, vol. abs/1701.04783, 2017. [Online]. Available: http://arxiv.org/abs/1701.04783

[17] T. Bansal, D. Belanger, and A. McCallum, "Ask the gru: Multi-task learning for deep text recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys '16, 2016, pp. 107–114. [Online]. Available: http://doi.acm.org/10.1145/2959100.2959180

[18] Z. Xu, C. Chen, T. Lukasiewicz, Y. Miao, and X. Meng, "Tag-aware personalized recommendation using a deep-semantic similarity model with negative sampling," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '16. New York, NY, USA: ACM, 2016, pp. 1921–1924. [Online]. Available: http://doi.acm.org/10.1145/2983323.2983874

[19] G. Sottocornola, F. Stella, M. Zanker, and F. Canonaco, "Towards a deep learning model for hybrid recommendation," in *Proceedings of the International Conference on Web Intelligence*, ser. WI '17. New York, NY, USA: ACM, 2017, pp. 1260–1264. [Online]. Available: http://doi.acm.org/10.1145/3106426.3110321

[20] D. Kim, C. Park, J. Oh, S. Lee, and H. Yu, "Convolutional matrix factorization for document context-aware recommendation," in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys '16. New York, NY, USA: ACM, 2016, pp. 233–240. [Online]. Available: http://doi.acm.org/10.1145/2959100.2959165

[21] F. Vasile, E. Smirnova, and A. Conneau, "Meta-prod2vec - product embeddings using side-information for recommendation," *CoRR*, vol. abs/1607.07326, 2016.

[22] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science (JASIS)*, vol. 41, no. 6, pp. 391–407, 1990.

[23] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937

[24] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014. [Online]. Available: http://arxiv.org/abs/1405.4053

[25] N. Djuric, H. Wu, V. Radosavljevic, M. Grbovic, and N. Bhamidipati, "Hierarchical neural language models for joint representation of streaming documents and their content," *CoRR*, vol. abs/1606.08689, 2016. [Online]. Available: http://arxiv.org/abs/1606.08689

[26] P. Cano, M. Koppenberger, and N. Wack, "Content-based music audio recommendation," in *Proceedings of the 13th Annual ACM International Conference on Multimedia*, ser. MULTIMEDIA '05. New York, NY, USA: ACM, 2005, pp. 211–212. [Online]. Available: http://doi.acm.org/10.1145/1101149.1101181

[27] X. Wang and Y. Wang, "Improving content-based and hybrid music recommendation using deep learning," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 627–636. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654940

[28] Y. Li, H. Wang, H. Liu, and B. Chen, "A study on content-based video recommendation," in *2017 IEEE International Conference on Image Processing (ICIP)*, 2017, pp. 4581–4585.

[29] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel, "Image-based recommendations on styles and substitutes," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '15, 2015, pp. 43–52.

[30] D. Sejal, T. Ganeshsingh, K. R. Venugopal, S. S. Iyengar, and L. M. Patnaik, "ACSIR: ANOVA cosine similarity image recommendation in vertical search," *IJMIR*, vol. 6, no. 2, pp. 143–154, 2017. [Online]. Available: https://doi.org/10.1007/s13735-017-0124-0

[31] C. Musto, G. Semeraro, M. de Gemmis, and P. Lops, "Word embedding techniques for content-based recommender systems: An empirical evaluation," in *Poster Proceedings of the 9th ACM Conference on Recommender Systems, RecSys 2015, Vienna, Austria, September 16, 2015.*, 2015.

[32] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *In EMNLP*, 2014.

[33] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2827872

[34] S. Oramas, V. Ostuni, T. Di Noia, X. Serra, and E. Di Sciascio, "Sound and music recommendation with knowledge graphs," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, pp. 1–21, 10/2016 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3004291.2926718