Semantic Product Search

Abstract

We study the problem of semantic matching in product search, that is, given a customer query, retrieve all semantically related products from the catalog. Pure lexical matching via an inverted index falls short in this respect due to several factors: a) lack of understanding of hypernyms, synonyms, and antonyms, b) fragility to morphological variants (e.g."woman"vs."women"), and c) sensitivity to spelling errors. To address these issues, we train a deep learning model for semantic matching using customer behavior data. Much of the recent work on large-scale semantic search using deep learning focuses on ranking for web search. In contrast, semantic matching for product search presents several novel challenges, which we elucidate in this paper. We address these challenges by a) developing a new loss function that has an inbuilt threshold to differentiate between random negative examples, impressed but not purchased examples, and positive examples (purchased items), b) using average pooling in conjunction with n-grams to capture short-range linguistic patterns, c) using hashing to handle out of vocabulary tokens, and d) using a model parallel training architecture to scale across 8 GPUs. We present compelling offline results that demonstrate at least 4.7% improvement in Recall@100 and 14.5% improvement in mean average precision (MAP) over baseline state-of-the-art semantic search methods using the same tokenization method. Moreover, we present results and discuss learnings from online A/B tests which demonstrate the efficacy of our method.

Introduction

At a high level, as shown in Figure FIGREF4 , a product search engine works as follows: a customer issues a query, which is passed to a lexical matching engine (typically an inverted index BIBREF0 , BIBREF1) to retrieve all products that contain words in the query, producing a match set. The match set

passes through stages of ranking, wherein top results from the previous stage are re-ranked before the most relevant items are finally displayed. It is imperative that the match set contain a relevant and diverse set of products that match the customer intent in order for the subsequent rankers to succeed. However, inverted index-based lexical matching falls short in several key aspects:

In this paper, we address the question: Given rich customer behavior data, can we train a deep learning model to retrieve matching products in response to a query? Intuitively, there is reason to believe that customer behavior logs contain semantic information; customers who are intent on purchasing a product circumvent the limitations of lexical matching by query reformulation or by deeper exploration of the search results. The challenge is the sheer magnitude of the data as well as the presence of noise, a challenge that modern deep learning techniques address very effectively.

Product search is different from web search as the queries tend to be shorter and the positive signals (purchases) are sparser than clicks. Models based on conversion rates or click-through-rates may incorrectly favor accessories (like a phone cover) over the main product (like a cell phone). This is further complicated by shoppers maintaining multiple intents during a single search session: a customer may be looking for a specific television model while also looking for accessories for this item at the lowest price and browsing additional products to qualify for free shipping. A product search engine should reduce the effort needed from a customer with a specific mission (narrow queries) while allowing shoppers to explore when they are looking for inspiration (broad queries).

As mentioned, product search typically operates in two stages: matching and ranking. Products that contain words in the query (INLINEFORM0) are the primary candidates. Products that have prior behavioral associations (products bought or clicked after issuing a query INLINEFORM1) are also included in the candidate set. The ranking step takes these candidates and orders them using a machine-learned rank function to optimize for customer satisfaction and business metrics.

We present a neural network trained with large amounts of purchase and click signals to complement a lexical search engine in ad hoc product retrieval. Our first contribution is a loss function with a built-in threshold to differentiate between random negative, impressed but not purchased, and purchased items. Our second contribution is the empirical result that recommends average pooling in combination with INLINEFORM0 -grams that capture short-range linguistic patterns instead of more complex architectures. Third, we show the effectiveness of consistent token hashing in Siamese networks for zero-shot learning and handling out of vocabulary tokens.

In Section SECREF2 , we highlight related work. In Section SECREF3 , we describe our model architecture, loss functions, and tokenization techniques including our approach for unseen words. We then introduce the readers to the data and our input representations for queries and products in Section SECREF4 . Section SECREF5 presents the evaluation metrics and our results. We provide implementation details and optimizations to efficiently train the model with large amounts of data in Section SECREF6 . Finally, we conclude in Section SECREF7 with a discussion of future work.

Related Work

There is a rich literature in natural language processing (NLP) and information retrieval (IR) on capturing the semantics of queries and documents. Word2vec BIBREF4 garnered significant attention by demonstrating the use of word embeddings to capture semantic structure; synonyms cluster together in the embedding space. This technique was successfully applied to document ranking for web search with the DESM model BIBREF5 . Building from the ideas in word2vec, BIBREF6 trained neural word embeddings to find neighboring words to expand queries with synonyms. Ultimately, based on these recent advancements and other key insights, the state-of-the-art models for semantic search can generally be classified into three categories:

BIBREF7 introduced Latent Semantic Analysis (LSA), which computes a low-rank factorization of a term-document matrix to identify semantic concepts and was further refined by BIBREF8 , BIBREF9 and extended by ideas from Latent Dirichlet Allocation (LDA) BIBREF10 in BIBREF11 . In 2013, BIBREF12 published the seminal paper in the space of factorized models by introducing the Deep Semantic Similarity Model (DSSM). Inspired by LSA and Semantic Hashing BIBREF13 , DSSM involves training an end-to-end deep neural network with a discriminative loss to learn a fixed-width representation for queries and documents. Fully connected units in the DSSM architecture were subsequently replaced with Convolutional Neural Networks (CNNs) BIBREF14 , BIBREF15 and Recurrent Neural Networks (RNNs) BIBREF16 to respect word ordering. In an alternate approach, which articulated the idea of interaction models, BIBREF17 introduced the Deep Relevance Matching Model (DRMM) which leverages an interaction matrix to capture local term matching within neural approaches which has been successfully extended by MatchPyramid BIBREF18 and other techniques BIBREF19 , BIBREF20 , BIBREF21 , BIBREF22 , BIBREF23 . Nevertheless, these interaction methods require memory and computation proportional to the number of words in the document and hence are prohibitively expensive for online inference. In addition, Duet BIBREF24 combines the approaches of DSSM and DRMM to balance the importance of semantic and lexical matching. Despite obtaining state-of-the-art results for ranking, these methods report limited success on ad hoc retrieval tasks BIBREF24 and only achieve a sub-50% Recall@100 and MAP on our product matching dataset, as shown with the ARC-II and Match Pyramid baselines in Table TABREF30 .

While we frequently evaluate our hypotheses on interaction matrix-based methods, we find that a factorized model architecture achieves comparable performance while only requiring constant memory per product. Hence, we only present our experiments as it pertains to factorized models in this paper. Although latent factor models improve ranking metrics due to their ability to memorize associations between the query and the product, we exclude it from this paper as we focus on the matching task. Our choice of model architecture was informed by empirical experiments while constrained by the cost per

query and our ability to respond within 20 milliseconds for thousands of queries per second.

Neural Network Architecture

Our neural network architecture is shown in Figure FIGREF9 . As in the distributed arm of the Duet model, our first model component is an embedding layer that consists of INLINEFORM0 parameters where INLINEFORM1 is the vocabulary and INLINEFORM2 is the embedding dimension. Each row corresponds to the parameters for a word. Unlike Duet, we share our embeddings across the query and product. Intuitively, sharing the embedding layer in a Siamese network works well, capturing local word-level matching even before training these networks. Our experiments in Table UID37 confirm this intuition. We discuss the specifics of our query and product representation in Section SECREF4 .

To generate a fixed length embedding for the query (INLINEFORM0) and the product (INLINEFORM1) from individual word embeddings, we use average pooling after observing little difference (<0.5%) in both MAP and Recall@100 relative to recurrent approaches like LSTM and GRU (see Table TABREF27). Average pooling also requires far less computation, reducing training time and inference latency. We reconciled this departure from state-of-the-art solutions for Question Answering and other NLP tasks through an analysis that showed that, unlike web search, both query and product information tend to be shorter, without long-range dependencies. Additionally, product search queries do not contain stop words and typically require every query word (or its synonym) to be present in the product.

Queries typically have fewer words than the product content. Because of this, we observed a noticeable difference in the magnitude of query and product embeddings. This was expected as the query and the product models were shared with no additional parameters to account for this variance. Hence, we introduced Batch Normalization layers BIBREF25 after the pooling layers for the query and the product arms. Finally, we compute the cosine similarity between INLINEFORM0 and INLINEFORM1 . During

online A/B testing, we precompute INLINEFORM2 for all the products in the catalog and use a INLINEFORM3 -Nearest Neighbors algorithm to retrieve the most similar products to a given query INLINEFORM4 .

Loss Function

A critical decision when employing a vector space model is defining a match, especially in product search where there is an important tradeoff between precision and recall. For example, accessories like mounts may also be relevant for the query “led tv.”

Pruning results based on a threshold is a common practice to identify the match set. Pointwise loss functions, such as mean squared error (MSE) or mean absolute error (MAE), require an additional step post-training to identify the threshold. Pairwise loss functions do not provide guarantees on the magnitude of scores (only on relative ordering) and thus do not work well in practice with threshold-based pruning. Hence, we started with a pointwise 2-part hinge loss function as shown in Equation (EQREF11) that maximizes the similarity between the query and a purchased product while minimizing the similarity between a query and random products. Define INLINEFORM0 , and let INLINEFORM1 if product INLINEFORM2 is purchased in response to query INLINEFORM3 , and INLINEFORM4 otherwise. Furthermore let INLINEFORM5 , and INLINEFORM6 for some predefined thresholds INLINEFORM7 and INLINEFORM8 and INLINEFORM9 . The two part hinge loss can be defined as DISPLAYFORM0

Intuitively, the loss ensures that when INLINEFORM0 then INLINEFORM1 is less than INLINEFORM2 and when INLINEFORM3 then INLINEFORM4 is above INLINEFORM5 . After some empirical tuning on a validation set, we set INLINEFORM6 and INLINEFORM7 .

As shown in Table TABREF26 , the 2-part hinge loss improved offline matching performance by more

than 2X over the MSE baseline. However in Figure FIGREF12 , a large overlap in score distribution between positives and negatives can be seen. Furthermore, the score distribution for negatives appeared bimodal. After manually inspecting the negative training examples that fell in this region, we uncovered that these were products that were impressed but not purchased by the customer. From a matching standpoint, these products are usually valid results to show to customers. To improve the model's ability to distinguish positives and negatives considering these two classes of negatives, we introduced a 3-part hinge loss: DISPLAYFORM0

where INLINEFORM0 , INLINEFORM1 , and INLINEFORM2 denote indicators signifying if the product INLINEFORM3 was purchased, not impressed and not purchased, and impressed (but not purchased) in response to the query INLINEFORM4 , respectively, and INLINEFORM5 . Based on the 2-part hinge score distribution, INLINEFORM6 was set to INLINEFORM7 with INLINEFORM8 and INLINEFORM9 as before. The effectiveness of this strategy can be seen in Figure FIGREF14 , where one can observe a clear separation in scores between random and impressed negatives vs positives.

Tokenization Methods

In this section, we describe our tokenization methodology, or the procedure by which we break a string into a sequence of smaller components such as words, phrases, sub-words, or characters. We combine word unigram, word n-gram, and character trigram features into a bag of n-grams and use hashing to handle the large vocabulary size, similar to the fastText approach BIBREF26 .

This is the basic form of tokenization where the input query or product title is tokenized into a list of words. For example, the word unigrams of "artistic iphone 6s case" are ["artistic", "iphone", "6s", "case"].

In a bag of words model like ours, word unigrams lose word ordering. Instead of using LSTMs or CNNs to

address this issue, we opted for INLINEFORM0 -grams as in BIBREF27 . For example, the word bigrams of "artistic iphone 6s case" are ["artistic#iphone", "iphone#6s", "6s#case"] and the trigrams are ["artistic#iphone#6s", "iphone#6s#case"]. These INLINEFORM1 -grams capture phrase-level information; for example if “for iphone” exists in the query, the model can infer that the customer's intention is to search for iphone accessories rather than iphone — an intent not captured by a unigram model.

Character trigram embeddings were proposed by the DSSM paper BIBREF12 . The string is broken into a list of all three-character sequences. For the example "artistic iphone 6s case", the character trigrams are ["#ar", "art", "rti", "tis", "ist", "sti", "tic", "ic#", "c#i", "#ip", "iph", "pho", "hon", "one", "ne#", "e#6", "#6s", "6s#", "s#c", "#ca", "cas", "ase", "se#"]. Character trigrams are robust to typos (“iphione” and “iphonr”) and handle compound words (“amazontv” and “firetvstick”) naturally. Another advantage in our setting is the ability to capture similarity of model parts and sizes.

It is computationally infeasible to maintain a vocabulary that includes all the possible word INLINEFORM0 -grams as the dictionary size grows exponentially with INLINEFORM1 . Thus, we maintain a "short" list of several tens or hundreds of thousands of INLINEFORM2 -grams based on token frequency. A common practice for most NLP applications is to mask the input or use the embedding from the 0th location when an out-of-vocabulary word is encountered. Unfortunately, in Siamese networks, assigning all unknown words to the same shared embedding location results in incorrectly mapping two different out-of-vocabulary words to the same representation. Hence, we experimented with using the “hashing trick" BIBREF28 popularized by Vowpal Wabbit to represent higher order INLINEFORM3 -grams that are not present in the vocabulary. In particular, we hash out-of-vocabulary tokens to additional embedding bins. The combination of using a fixed hash function and shared embeddings ensures that unseen tokens that occur in both the query and document map to the same embedding vector. During our initial experiments with a bin size of 10,000, we noticed that hashing collisions incorrectly promoted irrelevant products for queries, led to overfitting, and did not improve offline metrics. However, setting a bin size

5-10 times larger than the vocabulary size improved the recall of the model.

There are several ways to combine the tokens from these tokenization methods. One could create separate embeddings for unigrams, bigrams, character trigrams, etc. and compute a weighted sum over the cosine similarity of these INLINEFORM0 -gram projections. But we found that the simple approach of combining all tokens in a single bag-of-tokens performs well. We conclude this section by referring the reader to Figure FIGREF21 , which walks through our tokenization methods for the example “artistic iphone 6s case”. In Table UID33 , we show example queries and products retrieved to highlight the efficacy of our best model to understand synonyms, intents, spelling errors and overall robustness.

Data

We use 11 months of search logs as training data and 1 month as evaluation. We sample 54 billion query-product training pairs. We preprocess these sampled pairs to 650 million rows by grouping the training data by query-product pairs over the entire time period and using the aggregated counts as weights for the pairs. We also decrease the training time by 3X by preprocessing the training data into tokens and using mmap to store the tokens. More details on our best practices for reducing training time can be found in Section SECREF6 .

For a given customer query, each product is in exactly one of three categories: purchased, impressed but not purchased, or random. For each query, we target a ratio of 6 impressed and 7 random products for every query-product purchase. We sample this way to train the model for both matching and ranking, although in this paper we focus on matching. Intuitively, matching should differentiate purchased and impressed products from random ones; ranking should differentiate purchased products from impressed ones.

We choose the most frequent words to build our vocabulary, referred to as INLINEFORM0 . Each token in the vocabulary is assigned a unique numeric token id, while remaining tokens are assigned 0 or a hashing based identifier. Queries are lowercased, split on whitespace, and converted into a sequence of token ids. We truncate the query tokens at the 99th percentile by length. Token vectors that are smaller than the predetermined length are padded to the right.

Products have multiple attributes, like title, brand, and color, that are material to the matching process. We evaluated architectures to embed every attribute independently and concatenate them to obtain the final product representation. However, large variability in the accuracy and availability of structured data across products led to 5% lower recall than simply concatenating the attributes. Hence, we decided to use an ordered bag of words of these attributes.

Experiments

In this section we describe our metrics, training procedure, and the results, including the impact of our method in production.

Metrics

We define two evaluation subtasks: matching and ranking.

Matching: The goal of the matching task is to retrieve all relevant documents from a large corpus for a given query. In order to measure the matching performance, we first sample a set of 20K queries. We then evaluate the model's ability to recall purchased products from a sub-corpus of 1 million products for those queries. Note that the 1 million product corpus contains purchased and impressed products for every query from the evaluation period as well as additional random negatives. We tune the model

hyperparameters to maximize Recall@100 and Mean Average Precision (MAP).

Ranking: The goal of this task is to order a set of documents by relevance, defined as purchase count conditioned on the query. The set of documents contains purchased and impressed products. We report standard information retrieval ranking metrics, such as Normalized Discounted Cumulative Gain (NDCG) and Mean Reciprocal Rank (MRR).

Results

In this section, we present the durable learnings from thousands of experiments. We fix the embedding dimension to 256, weight matrix initialization to Xavier initialization BIBREF29 , batch size to 8192, and the optimizer to ADAM with the configuration INLINEFORM0 for all the results presented. We refer to the hinge losses defined in Section SECREF10 with INLINEFORM1 and INLINEFORM2 as the L1 and L2 variants respectively. Unigram tokenization is used in Table TABREF26 and Table TABREF27 , as the relative ordering of results does not change with other more sophisticated tokenizations.

We present the results of different loss functions in Table TABREF26 . We see that the L2 variant of each loss consistently outperforms the L1. We hypothesize that L2 variants are robust to outliers in cosine similarity. The 3-part hinge loss outperforms the 2-part hinge loss in matching metrics in all experiments although the two loss functions have similar ranking performance. By considering impressed negatives, whose text is usually more similar to positives than negatives, separately from random negatives in the 3-part hinge loss, the scores for positives and random negatives become better separated, as shown in Section SECREF10 . The model can better differentiate between positives and random negatives, improving Recall and MAP. Because the ranking task is not distinguishing between relevant and random products but instead focuses on ordering purchased and impressed products, it is not surprising that the 2-part and 3-part loss functions have similar performance.

In Table TABREF27 , we present the results of using LSTM, GRU, and averaging to aggregate the token embeddings. Averaging performs similar to or slightly better than recurrent units with significantly less training time. As mentioned in Section SECREF8 , in the product search setting, queries and product titles tend to be relatively short, so averaging is sufficient to capture the short-range dependencies that exist in queries and product titles. Furthermore, recurrent methods are more expressive but introduce specialization between the query and title. Consequently, local word-level matching between the query and the product title may not be not captured as well.

In Table TABREF28 , we compare the performance of using different tokenization methods. We use average pooling and the 3-part L2 hinge loss. For each tokenization method, we select the top INLINEFORM0 terms by frequency in the training data. Unless otherwise noted, INLINEFORM1 was set to 125K, 25K, 64K, and 500K for unigrams, bigrams, character trigrams, and out-of-vocabulary (OOV) bins respectively. It is worth noting that using only character trigrams, which was an essential component of DSSM BIBREF12 , has competitive ranking but not matching performance compared to unigrams. Adding bigrams improves matching performance as bigrams capture short phrase-level information that is not captured by averaging unigrams. For example, the unigrams for “chocolate milk” and “milk chocolate” are the same although these are different products. Additionally including character trigrams improves the performance further as character trigrams provide generalization and robustness to spelling errors.

Adding OOV hashing improves the matching performance as it allows better generalization to infrequent or unseen terms, with the caveat that it introduces additional parameters. To differentiate between the impact of additional parameters and OOV hashing, the last two rows in Table TABREF28 compare 500K unigrams to 125K unigrams and 375K OOV bins. These models have the same number of parameters, but the model with OOV hashing performs better.

In Table TABREF29 , we present the results of using batch normalization, layer normalization, or neither

on the aggregated query and product embeddings. The “Query Sorted” column refers to whether all positive, impressed, and random negative examples for a single query appear together or are shuffled throughout the data. The best matching performance is achieved using batch normalization and shuffled data. Using sorted data has a significantly negative impact on performance when using batch normalization but not when using layer normalization. Possibly, the batch estimates of mean and variance are highly biased when using sorted data.

Finally, in Table TABREF30 , we compare the results of our model to four baselines: DSSM BIBREF12 , Match Pyramid BIBREF18 , ARC-II BIBREF15 , and our model with frozen, randomly initialized embeddings. We only use word unigrams or character trigrams in our model, as it is not immediately clear how to extend the bag-of-tokens approach to methods that incorporate ordering. We compare the performance of using the 3-part L2 hinge loss to the original loss presented for each model. Across all baselines, matching performance of the model improves using the 3-part L2 hinge loss. ARC-II and Match Pyramid ranking performance is similar or lower when using the 3-part loss. Ranking performance improves for DSSM, possibly because the original approach uses only random negatives to approximate the softmax normalization. More complex models, like Match Pyramid and ARC-II, had significantly lower matching and ranking performance while taking significantly longer to train and evaluate. These models are also much harder to tune and tend to overfit.

The embeddings in our model are trained end-to-end. Previous experiments using other methods, including Glove and word2vec, to initialize the embeddings yielded poorer results than end-to-end training. When comparing our model with randomly initialized to one with trained embeddings, we see that end-to-end training results in over a 3X improvement in Recall@100 and MAP.