Matching Restaurant Menus to Crowdsourced Food Data

A Scalable Machine Learning Approach

Hesam Salehian
Under Armour Connected Fitness
135 Townsend Street
San Francisco, California 94107
hsalehian@underarmour.com

Patrick Howell
Under Armour Connected Fitness
135 Townsend Street
San Francisco, California 94107
phowell@underarmour.com

Chul Lee
Under Armour Connected Fitness
135 Townsend Street
San Francisco, California 94107
clee3@underarmour.com

ABSTRACT

We study the problem of how to match a formally structured restaurant menu item to a large database of less structured food items that has been collected via crowd-sourcing. At first glance, this problem scenario looks like a typical text matching problem that might possibly be solved with existing text similarity learning approaches. However, due to the unique nature of our scenario and the need for scalability, our problem imposes certain restrictions on possible machine learning approaches that we can employ. We propose a novel, practical, and scalable machine learning solution architecture, consisting of two major steps. First we use a query generation approach, based on a Markov Decision Process algorithm, to reduce the time complexity of searching for matching candidates. That is then followed by a re-ranking step, using deep learning techniques, to meet our required matching quality goals. It is important to note that our proposed solution architecture has already been deployed in a real application system serving tens of millions of users, and shows great potential for practical cases of user-entered text to structured text matching, especially when scalability is crucial.

KEYWORDS

Short text matching, Nutrition estimation, Convolutional neural networks, Markov decision process

ACM Reference format:
Hesam Salehian, Patrick Howell, and Chul Lee. 2017. Matching Restaurant Menus to Crowdsourced Food Data. In *Proceedings of KDD'17, August 13–17, 2017, Halifax, NS, Canada..*, 9 pages.
DOI: http://dx.doi.org/10.1145/3097983.3098125

1 INTRODUCTION

Today's app-driven marketplace often leads to the need to integrate and connect sources of data that have been collected and structured in very different ways. The problem of connecting disparate data is not a new one, but the nature and characteristics of the data can cause great difficulties, even within such a well-studied topic. Specifically, our examination here is centered on a problem of matching short, structured text to items pulled from a much larger database of short, unstructured text data. Our proposed solution is a multistage architecture whereby the structured text is passed through a query generation process and then returned results are re-ranked through a deep learning model, using a convolutional neural network as the basis, to produce a probability vector of whether or not the candidate item is relevant to the query.

The source of the data that serves as the cornerstone of this analysis is collected through an app that allows users to track exercise, dietary habits, and weight loss. MyFitnessPal (MFP) is a free health and fitness app available in Android, iOS, and web formats that helps people set and achieve personalized health goals through tracking nutrition and physical activity. In fact, MFP is consistently ranked at the top of the health and fitness category in both the Apple App Store and in Google Play. Through the tracking of their health and nutrition, MFP enables users to gain insights that help them make smarter choices and build healthier habits. Upon setting a personal fitness goal, users can visually inspect their fitness and weight loss progress. MFPfis food data – namely, nutritional contents and the food descriptions – are constructed via users inputs. In Table 1, we illustrate a snapshot of MFP's food database, which contains various types of Hazelnut coffee. While MFPfis database carries no guarantees of nutritional accuracy due to its reliance on crowd-sourcing, the great popularity of the app partially speaks of the quality of its food DB that consists of over hundreds of millions of food items and tens of billions of individual food entries.

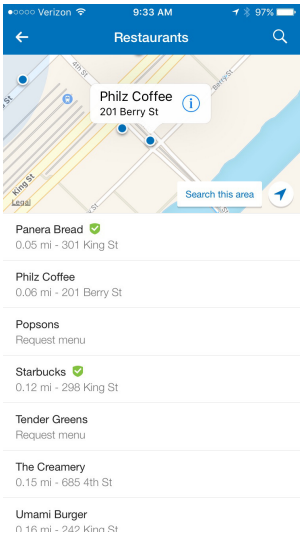
This database alone can provide numerous insights on user eating behavior, but one element that has been missing is an ability to show real-time food matches to users as they are eating, based on their location. In the real world, this translates to being able to match database foods to restaurant menus, which are tied to geolocation points of the venue. Related to this, one product feature that users of MFP consistently requested was "Restaurant Logging" (RL), or, that MFP should allow its users to log foods while they are eating out in restaurants based on menus at a location. This feature was first rolled out in 2015 and consists of 3 steps: (1) A venue navigation step in which users can choose the restaurant at which they had eaten. (2) A menu navigation step in which users have access to the menu data of the given restaurant that they have chosen. (3) A logging step in which users choose the food item from the given menu that they have eaten and log the selected food item. While several big food chain restaurants provide nutrition information in their menus, most food items in restaurant menus do not have directly associated nutrition information. Thus, it becomes critical to complement each food item in restaurant menus with accurate nutrition information

KDD'17, August 13–17, 2017, Halifax, NS, Canada..

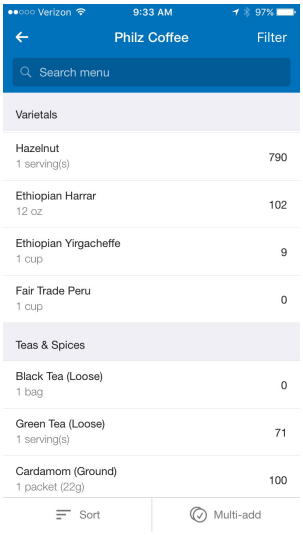
DOI: http://dx.doi.org/10.1145/3097983.3098125

Table 1: Snapshot of Food DB

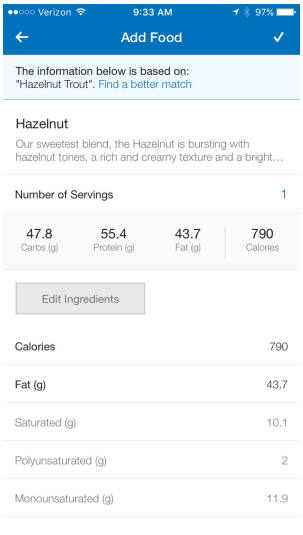
Food ID	Brand Name	Food Name	Calorie Information	...	...
3424	Coffee Mate	Hazelnut Cream	35	...	...
12358	Bailey's	Coffee Creamer - Hazelnut	35	...	...
98742	Bailey's	Coffee Creamer - Hazelnut	138	...	...
64628	Kroger	Hazelnut Coffee Creamer	35	...	...
09194	Hazelnut Coffee	Coffee 1 Creamer	35	...	...
63524	McDonald's	Iced Coffee (Hazelnut)	180	...	...
76654	Maxwell House	Coffee (Hazelnut)	70	...	...
22214	Starbucks	Hazelnut Coffee Cake	630	...	...
52343	Einsteins	Vanilla Hazelnut Coffee	22	...	...



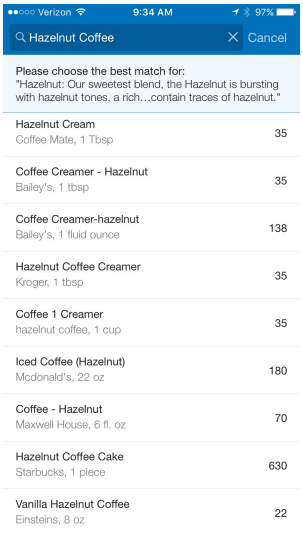
(a) Venues found from geolocation – Here selecting Philz Coffee as a motivating example



(b) Menu is loaded from third party – Preview of nutrition from "top" matches appear from a simple search call on the backend



(c) Item is opened for logging food – In this case showing "Hazelnut Trout" as original match to "Hazelnut" query, but likely irrelevant at a coffeeshop



(d) User can manually search for an alternative match – "Hazelnut Coffee" being a better search term to "Hazelnut" at Philz Coffee in this example

Figure 1: Restaurant Logging: Original Flow with Incorrect Matches

by retrieving the exact match from MFP's food DB if the given food item is available, or at least the approximate nutrition that exists in the DB. However, similar to there being no guarantee of nutritional accuracy, there is also no guarantee that a restaurant's menu item necessarily exists within the app food database, unless a user has specifically added it before. If the given restaurant food item is not found, then the best option is to match the given food item that is semantically closest to the given restaurant food item. We refer this problem as *restaurant food matching*. More precisely, given $\forall\ restaurant\_food_i \in restaurant\_menu_s$, *restaurant food matching* tries to learn a function M that

$$\underset{food_m \in foodDB}{\arg\max} \quad M(restaurant\_food_i, food_m)$$

At first glance, the restaurant food matching problem looks similar in nature to previously studied text matching problems, making the application of well known techniques like locality sensitivity hashing (LSH)[12] or improved neighbordhood-based algorithms[28] seem tempting. However, our *restaurant food matching problem* has its own peculiarities since both the given text to be matched (i.e. restaurant menu item) and the matching text in the app database (i.e. food item with its description) are short in length. In addition, both the lexical and grammatical structure of each word has a bigger impact on the overall accuracy than might be the case in standard settings: "spaghetti meat sauce" needs to be treated differently from "spaghetti with meat sauce." Word order also has a significant impact on the overall matching accuracy: "chocolate milk"' should be treated differently from "milk chocolate." All of these nuances combine to make the direct application of previous text matching problems a challenge. Finally, menu text data often comes in a form that is highly structured by possessing information such as the venue's name, menu section name, and the item itself;

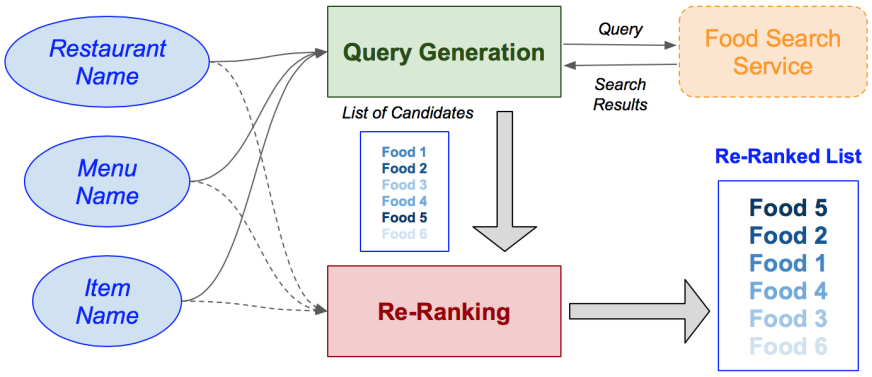


Figure 2: Overall architecture of the proposed system

meanwhile, foods from our crowd-sourced database have nowhere near the same consistency, owing to the random noise that accompanies user inputs – misspellings, information in the wrong fields, etc.

One additional and critical requirement for our restaurant food matching problem is that its matching speed has to be near-real time in practice and ideally with minimal additional system overhead. To take LSH as an example, there is significant overhead of applying LSH to a large collection of items, especially when our goal is fast performance on a scale of up to hundrends of concurrent queries. Applying LSH at this scale possesses several challenges: the maintenance of a large number of hash tables to achieve both high precision and recall; the fact that scaling LSH would require a distributed implementation due to it being a main-memory algorithm; and there is inherent difficulty in setting up key parameters for LSH to avoid excessive memory consumption or sub-optimal performance [27]. Thus, we propose a solution architecture that is highly optimized for scalability while enabling us to overcome the complexity of our matching problem, by breaking our problem into two distinct sub-problems, *query generation* and *re-ranking*. Figure 2 illustrates the high-level architecture of our proposed solution. The first step refers to the candidate set construction for foods to be matched, while the second step refers to the selection of the best food item in the given universe of match-possible food items. During the query generation step, the complexity to construct the initial candidate set of food items to be matched should be reasonable, even when the overall quality of matching candidate set could end up being sub-optimal. To handle this potential sub-optimality during the initial candidate set construction, it is important to employ the most sophisticated/advanced available machine learning technique to achieve our required quality goals and handle the peculiarity of our data. For both problems, we adapt prior state-of-the art machine learning techniques and run experiments to compare our proposed approaches against some traditional approaches to show that our proposed solution architecture can outperform the more basic models.

2 QUERY GENERATION ALGORITHM

As described earlier, we decompose the problem of restaurant food matching, into two sub-problems: (a) generation of matching candidates for the given food item, and (b) re-ranking of the set of candidates based on their similarity with the input restaurant food name. In this section, we present the algorithm proposed to tackle the first sub-problem. At a high-level, the algorithm takes a triplet of strings –the restaurant name, menu name and the item name – as input and generates the most relevant query (see figure 2). This algorithm is henceforth referred to as *Query Generation* (QG).

2.1 Related Work

The query refinement problem has been well studied in literature in recent years. This problem includes modifying the original query input by a user, based on the search results and users feedback [19, 23, 25]. The iterative nature of this problem and the partial existence of user feedback make it well-suited for a reinforcement learning framework. The use of Markov Decision Processes (MDP) [14] is one of the most well-known techniques to address such a problem [8]. However, the problem addressed here is substantially different from the classic query refinement, for a number of reasons.

The first challenge relates to the unique and complicated nature of restaurant food names. More specifically, the average length of food names tends to be very short (< 20), hence presence or absence of a particular word can be critical when generating the optimum query. This makes it impossible to directly apply any standard *tf-idf* based techniques [8], because the term frequency is in most cases 0/1. Furthermore, restaurants tend to organize their food items in their menus using a variety of different formats. For instance,"Caesar salad" may belong to "appetizers" or "salads" section. As a more complicated example, foods may interact differently with section headers, as "Caesar" found under a "Salads and Sandwiches" section may refer to either "Caesar Salad" (e.g., at Panera Bread) or "Caesar Sandwich" (e.g., at Subway).

Second, in our setting, we lack user interaction data, which is a fundamental component of any reinforcement learning technique. In our problem scenario, users only have access to the final re-ranked list of matched food candidates for each restaurant's food while the actual query that was used to retrieve the candidate set is hidden. Therefore, our method is significantly different in nature

compared to the standard algorithms for query refinement and session search problems [3, 8, 13]. *Pseudo-Relevance Feedback* (PRF) methods are one possibility to address the query expansion problem, when no user feedback is available [1, 20]. However, these methods are not directly applicable to our case, because typically under PRF a user inputs a query, then the algorithm aims to expand it in an *unsupervised* manner. In our case no user input is available even for query initialization, meaning that the initial query is created only based on the restaurant food's name.

In order to tackle the above challenges, we propose an iterative machine learning algorithm, which is mainly inspired by MDP techniques. Given the set of all words in the combined universe of restaurant names, menu names and food item names, the learning algorithm assigns an optimum weight to each term in a query, in an iterative manner. Decisions to keep or remove each term are then made based on the weights computed. Due to the lack of real user signals in this process, the top K search results in response to the query in each iteration serve as a feedback mechanism to modify the weight vector. In the next section, our proposed model is explained in more detail.

2.2 Markov Decision Process Algorithm

In the proposed MDP setting, the *state* of the system, $\mathbf{q}$, in iteration k, is the current *query string* which is passed to the search service. The query string is represented by a set of *terms*, t_i, i.e., $\mathbf{q}^{(k)} = \{t_1, t_2, ..., t_n\}$. The set of actions, $A = \{a_j\}$, contains *keeping, adding or removing* terms in, to or from the current query, in order to make a new (and hopefully more relevant) one. The dynamic of the model is controlled by the *transition function* $T(\mathbf{q}'|\mathbf{q}, a_j)$, which defines the probability of taking a certain action a_j, to transform current query $\mathbf{q}$ to $\mathbf{q}'$. The reward function, $R(\mathbf{q}, a_j)$ evaluates each (feasible) action a_j, taken on any given query $\mathbf{q}$, and is defined as the maximum *relevance score* of the search results retrieved in response to the new query, $\mathbf{q}'$. Finally, $\gamma \in [0, 1]$ is the discount factor which controls the importance of the previously visited states. Accordingly, an MDP model aims to find the optimum set of actions made sequentially given an initial state, which is denoted by $V^*(\mathbf{q})$. This optimum *strategy* is obtained using the well-known Bellman equation [14]:

$$V^*(\mathbf{q}) = \max_{a_j} R(\mathbf{q}, a_j) + \gamma \sum_{\mathbf{q}'} T(\mathbf{q}'|\mathbf{q}, a_j) V^*(\mathbf{q}') \tag{1}$$

A *Value Iteration* (VI) approach [24] is a standard solution to an MDP model. Given, the reward function ($R(.)$) defined for each state, and the transition function ($T(.)$), VI evaluates the *utility* ($V^*_{i+1}(.)$) at each state in iteration $i + 1$, based on the old utility values (V^*_i).

In the current problem, it is intuitively possible to estimate the transition function based on the popularity of each term of the query string in the search results, and specify the probability of moving from one query to another. However, the classic VI technique is not directly applicable here due to extensive time and space complexities involved. Given the initial query of size n, there are 2^n possible states in this standard model, which yields to a transition matrix of size $2^n \times 2^n$. Performing computation over such a transition matrix is not feasible in practice.

Alternatively, based on the idea of "term popularity" we present an iterative/greedy technique, to approximate the optimum strategy in Eq. 1. The proposed technique assigns a weight value to each individual query term. The weights are a measure of popularity of each term in the results set, hence they are updated based on the search results at each iteration. Accordingly, an appropriate action to keep/add/remove terms to the current query is taken. Although, term weighting is popular in session search [8], and information filtering [22], lack of any user feedback makes our approach considerably different. To the best of our knowledge, there is no similar term weighting technique in literature to generate the most relevant query from scratch, without any sort of user feedback loop, and purely based on the returned search results.

Let $\mathbf{q}^{(k-1)} = \{t_1, t_2, ..., t_n\}$ be the query word set passed to the search service, at iteration $k-1$, and let $D^{(k)} = \{d_1, ..., d_j\}$ be the document set resulted from the search service in response to $\mathbf{q}^{(k-1)}$. Then, the weight vector update equation is written as:

$$\hat{P}(t_i \in \mathbf{q}^{(k-1)}|D^{(k)}) = [\gamma + \sum_{d_j \in D^{(k)}} P(d_j|\mathbf{q}^{(k-1)}) P(t_i|d_j)] P(t_i \in \mathbf{q}^{(k-1)}|D^{(k-1)}) \tag{2}$$

where $P(t_i|D^{(k-1)}) \in [0, 1]$ denotes the weight assigned to the term t_i in the query, once the document set $D^{(k-1)}$ has been observed, and $\gamma \in (0, 1)$ is the discount factor.

Also, $P(d_j|\mathbf{q}^{(k-1)}) \in [0, 1]$ represents the relevance score of document d_j to the given query, which can be either computed using any string similarity measure (e.g., Jaccard or Edit distances [5]), or can be assigned the relevance score values given by the search service.

Lastly, the term $P(t_i|d_j)$ shows the contribution of the term t_i in document d_j, which is usually computed by $P(t_i|d_j) = \frac{\#(t_i, d_j)}{|d_j|}$, where numerator and denominator denote the frequency of term t_i in document d_j, and the length of d_j, respectively. According to 2, each search result changes the term weights, depending on its total relevance to the entire query. Note that the original ranking of the documents in the search list is not necessarily ideal, at each iteration. Therefore, we used the string similarity measures, instead of the rank-based scores (e.g., $\frac{1}{r}$, $\frac{1}{log(r)+1}$, ...) in the update process. Next, the new weight vector obtained from Eq. 2 needs to be normalized for $[0, 1]$ values.

$$P(t_i \in \mathbf{q}^{(k-1)}|D^{(k)}) = \frac{\hat{P}(t_i \in \mathbf{q}^{(k-1)}|D^{(k)})}{\sum_j \hat{P}(t_j \in \mathbf{q}^{(k-1)}|D^{(k)})} \tag{3}$$

Given a triplet of (restaurant name, menu name, food item name), the algorithm initializes $\mathbf{q}^{(0)}$ to be the union of all terms in the input triplet, with equal weights. At each iteration, $k-1$, the query string $\mathbf{q}^{(k-1)}$ is sent to the search service, and the list of top K results (D^k) is retrieved. We employed a customized Elastic Search Service[1] which returns a list of relevant documents for each query string, sorted based on the combination of TF-IDF scores and users click history. Next, the weights corresponding to the terms in $\mathbf{q}^{(k-1)}$ are updated using Eq. 2. Although, all terms in this equation are positive, weights of some terms can become smaller over time, due

[1] https://www.elastic.co/

to the normalization step in Eq. 3. Once the term weights are updated after each iteration, one of the feasible actions (a_j) is taken for each term, based on the new weights. In our experiments, terms whose weight is < 0.1 or > 0.2 are removed from/added to the query string, respectively. Finally, our algorithm stops when the (Euclidean) distance between the new and old weight vectors is less than a threshold (e.g., 0.001). Although our proposed technique is an approximate technique to the MDP model in Eq. 1, its effectiveness and efficiency are shown through experiments with real data.

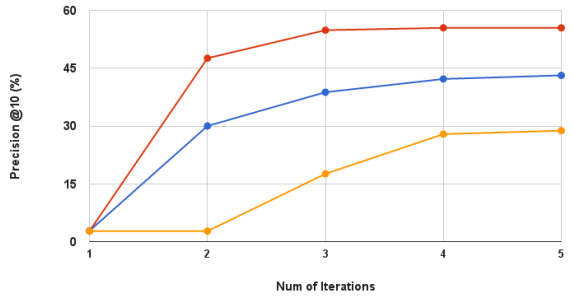


Figure 3: Average precision @10 for three MDP models: unweighted (yellow), weighted based on rank (blue) and weighted based on relevance score (red)

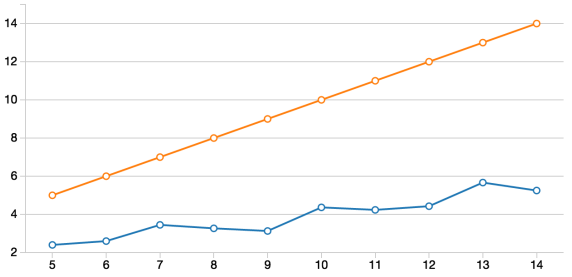


Figure 4: Average length of the optimum query (blue), with respect to the length of the input string (yellow)

2.3 Query Generation Results

In this experiment, 500 items from random restaurants are selected and inputted for QG. We use three different values for the *document weights*, in Eq. 2: (1) the relevance score between d_j and $\mathbf{q}^{(k-1)}$ given by the search service, (2) the rank of the retrieved document, i.e., $P(d_j|\mathbf{q}^{(k-1)}) = \frac{1}{j}, \forall j$, and (3) unweighted, i.e., $P(d_j|\mathbf{q}^{(k-1)}) = 1, \forall j$. Fig. 3 demonstrates this comparison. The average precision for all three models converged after 5 iterations, so the plots are limited only to the first 5 iterations. It can be seen that the relevance scores returned by the search service provide the best precision. This is because the relevance scores not only consider the common string similarity measures (i.e., TF-IDF), but also takes the users click/log history into account, which yields to a smarter weighting model.

Moreover, Fig. 4 shows the average length of the optimum query, using the first weighting scheme, with respect to the number of words in the input triplet. It can be seen that the length of the optimum query does not grow as rapidly as the input size. This is important, because in most cases, long input names are due to the presence of extra/irrelevant terms, which can be problematic when included in the query string.

Tabel 2 provides the results of the proposed query generation algorithm, for three sample inputs. In the first example, the words "restaurant" and "brewhouse" from the restaurant name, as well as "appetizers" and "shareable" from the menu section name are removed, mostly because they are not frequently used in conjunction with the rest of the words in the MFP food database. Similarly, the term "specialty" is removed from the menu name in the second example. In the third example, the restaurant name is not popular and the menu name means "appetizers" in Italian, hence both are not included in the final query, for similar reasons.

3 RE-RANKING

In this section we present the proposed algorithm to re-rank the match candidates, after retrieving them from the query generation algorithm. The need for quality results from the re-ranking algorithm is high. This is because the re-ranking process needs to also compensate for any quality deficiency that might have occurred during the query generation. Although there are several similarities between this task and the classic re-ranking algorithms, the unique nuances posed by food-related data requires that we move beyond existing techniques to achieve a satisfactory result quality threshold.

3.1 Related Work

Learning to rank is a well studied problem in the last decade, which involves the ranking of a set of documents with respect to a given query [9, 18]. Existing methods generally fall into three main categories, based on the way each learning instance is generated. In the *pointwise* ranking methods, each pair of (*query*, *doc*) is labeled as relevant or irrelevant, and the documents are then sorted with respect to the predicted label and the prediction confidence [6, 26]. The *pairwise* approaches take triplets of the form (*query*, doc_1, doc_2), and estimate the probability of doc_1 being more relevant to *query*, compared to doc_2. These relative probabilities are then exploited to re-rank the candidates [10]. In the *listwise* approaches, the *query* and the entire list of documents are considered as a single learning sample, and the ranking takes place over an entire list of candidates [2].

The choice of a suitable feature extraction technique is a fundamental step in any of these re-ranking algorithms. A widely used approach is to encode input text pairs using complex lexical, syntactic and semantic features and then compute various similarity measures between these representations [18, 21, 26]. In many cases, the final learning quality of a re-ranking algorithm is largely dependent on finding the right representation of input text pairs. In our particular scenario, directly applying previous techniques in our feature extraction task is not as feasible due to the small length of the food names and lack of external knowledge sources.

Table 2: Sample query generation results

Restaurant Name	Menu Section Name	Item Name	Final Query
BJ's Restaurant and Brewhouse	Appetizers and Shareable	Chicken Lettuce Wrap	bj's chicken lettuce wrap
Panera Bread	Specialty Breads	Sourdough	panera bread sourdough breads
Zero Zero	Antipasti	Marinated Italian Olives	marinated italian olives

The advent of Convolutional Neural Networks (CNNs) [16, 17] has opened new alternatives for complex, and mostly heuristic, feature engineering tasks, typically applied to image data. In recent years, more CNN-based methods have been proposed for text-based analysis. Taking inspiration from image studies, the concept in general is to transform the input text into an embedding matrix, and then feed this matrix through several convolution layers [7, 15, 29]. Many algorithms in this category are based on a *word-level embedding*, where a feature vector is assigned to each individual word or a *character-level embedding*, where an unknown feature vector is assigned to each character [30].

We present this variety of CNN architecture, following the structure for the short text matching approaches found in [26] and [11]. The proposed architecture is flexible enough to inject both word and character-level embedding. Unlike the character-level embedding presented in [30], we do not encode the characters by pre-defined sparse vectors, instead allowing the embedding to be learned during the training process. The character-level embedding proved particularly interesting since our food database is constructed via crowd-sourcing and therefore highly susceptible to misspellings. We present experimental results using both approaches for the sake of comparison.

3.2 Main Algorithm

An initial effort to address the food re-ranking problem involved a point-wise SVM algorithm, trained on $\mathbf{v} = \phi(q, f)$ instances with relevant or irrelevant labels, where q and f are the query restaurant food and the database food name, respectively, and $\mathbf{v}$ is the feature vector extracted via the function $\phi(.)$. The choice of an appropriate feature extraction function, i.e., $\phi(.)$, is a challenge. We opted to use a set of features based on well-known string similarity measures, combined to form a single vector. These low level features include Jaccard and Edit distances between different combinations of the restaurant food and the current database candidate. For a query of the form {restaurant name, menu name, item name}, and a database food candidate of type {brand name, description}, the string similarity scores are computed on (restaurant name, brand name), (item name, description), (item name + menu name, description), etc., and are appended together form a vector of size 28. Although the above SVM technique using an RBF Kernel provided reasonable accuracy on a small size of training data, it is unable to generalize over larger/more complicated datasets. This is expected, due to the sizable gap between the *low-level* heuristic feature extraction and *high-level* semantic complexities involved in this problem.

We found a CNN architecture well suited for this application, as it allows the model to learn an optimal representation of input food names together with a similarity function to relate them in a supervised way from a massive database of food names. The large complexity of this particular matching problem is far beyond the capabilities of standard feature extraction tools, as in some cases even a non-expert human might fail to make correct predictions. Moreover, equipped with millions of inputted text food items in our database, along with $500K$ restaurant menu-structured items, we are able to provide a CNN with plenty of training data, which is important for any deep learning technique.

The proposed CNN model is based on a pointwise learning approach to assign relevant/irrelevant labels to each pair of restaurant item (query) and food from the database (candidate). The architecture contains 2 convolution networks that are similar in components, but are trained separately. These networks create two dense feature vectors, corresponding to the query and the candidate. A fully connected layer is then used along with a final softmax layer to combine the dense feature vectors and transform them into into a two-length probability vector of relevant/irrelevant.

Two different CNN architectures are proposed here for testing, the difference between the two depending on the type of the embedding, whether word-level or character-level. A word-level approach carries a benefit of using a pretrained embedding model based on a much larger corpus than the CNN models themselves might use in this specialized application. As a counter-point, the character-level architecture is more flexible through its ability to include the embedding process itself directly into the training of the CNN model. Given these differences, we have constructed a flexible architecture that is able to test both, holding other aspects of the CNN constant.

For the word-level embedding CNN (wCNN), we applied Word2Vec [21], to represent each word by a numerical feature vector of size $e = 200$. To this end, a Word2Vec model was trained on a larger collected food names corpus with more than $5M$ unique food items. An input matrix is created by zero padding the number of word columns to a maximum of $L = 20$, and truncating a food name if it stretched beyond 20 words. The filter width for the wCNN model was set at $w = 3$, or to look at each item in trigram-length windows. Different parameters were necessary for the character-level embedding CNN (cCNN). The main advantage to the cCNN is the ability to learn embedding weights through backpropagation in training. The corpus of characters was limited to the Roman alphabet and a space character, keeping the number of learned embeddings at 27. We assigned to each character, an unknown vector of size $e = 10$ that is learned at the time of cCNN training. The maximum number of characters per food name was set to $L = 100$, again with zero padding if a word has less than 100 characters, and truncating if a word has more. The filter width for the cCNN is then set to only consider characters in a moving window of 5 neighbors.

The input to each query/match convolution layer is an embedding matrix of size $e \times L$, where e and L are the embedding vector size and the maximum length of the text (see Fig. 5). In each convolution network, there is 1 convolution layer with $n = 50$ filters of size $w \times e$, followed by a ReLU layer. Each convolution filter in the

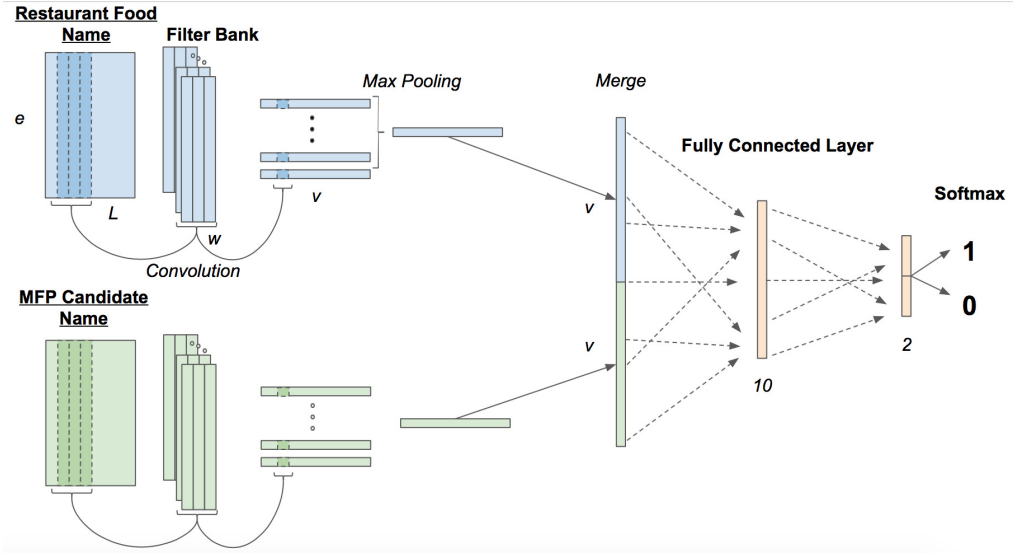


Figure 5: CNN Architecture

bank outputs a vector of size $v = L - w + 1$. Next, the 50 $1 \times v$ vectors generated from the filter bank are max pooled into a single dense vector. Therefore, in both the wCNN and cCNN, the output of the two parallel convolution networks for query and candidate string is a pair of dense feature vectors, each of size $1 \times v$. In order to combine the two dense vectors, the convolution layers are followed by 2 fully connected (FC) layers. The first layer transforms an input (combined) vector of size $1 \times 2v$ into a smaller vector of size 1×10. This vector is further shrunk via the second layer, into a 1×2 vector. There is one ReLU layer after each FC layer. Finally, a softmax layer is applied on top of this vector to make final probability vector of size 2, where each component correspond to the probability of relevant/irrelevant class membership. The implementation of CNN is carried out using Keras [4] with a backend in Theano. Also, a dropout value of 0.5 was used in both wCNN and cCNN on top of each convolution layer to prevent over-fitting.

4 EXPERIMENTS

4.1 Results

In this section, we compare the performance of the proposed CNN methods and the base SVM algorithm described earlier. A set of $1M$ labeled samples was collected to train each method. To evaluate the performances, a set of $4K$ instances were labeled by experts, to form our first test set. A second test set containing $100K$ instances was created from the implicit feedback of a set of users selecting foods. Items that were frequently logged by users were considered as "relevant" matches for the restaurant food item, while the "irrelevant" matches are the foods that were frequently skipped by users to pick better alternatives. These two test sets are called *hand-labeled* and *user-labeled*, respectively. A set of 5-fold validation was carried out for each method over each dataset, and the average accuracy values are reported in Table 3. The leftmost segment of Table 3 shows the accuracy of each technique, when tested over a random partition of the training set. It can be seen that SVM was able to provide a competitive accuracy over $10K$ samples, but for larger sets the word-level CNN approach outperformed the SVM classifier by wider margins, as expected.

It is evident that the word-level embedding had the best fit to the training data of size $1M$, compared to the competing methods. This is not surprising, as CNN-based approaches are able to learn many more feature complexities via larger training sets, while SVM is limited to the heuristically chosen features, which are independent of the training size. Meanwhile, the cCNN achieves similar accuracy to the wCNN on hand-labeled data, but lags behind SVM on the training and user-labeled experiments. Since cCNN is learning its embeddings at the time of the model training, its weaknesses might be explained from losing the advantage of preprocessing as in SVM and/or the advantage of a larger corpus for embeddings as in wCNN. We should also emphasize that the food candidates suggested to the users in the baseline flow were originally estimated using the same SVM approach that is used as the baseline comparison. Therefore, the user-labeled data is potentially biased towards the SVM model, which explains the larger accuracy of SVM in this case versus the hand-labeled data experiment (Table 3).

Table 4 contains a few examples of the input restaurant food information, along with the best matches retrieved by two different algorithms: SVM and word-level CNN (wCNN). Considering the food's ingredients and nutritional contents, in the first example, "Philadelphia sushi roll" is the best match, as successfully retrieved by wCNN. Meanwhile SVM failed to achieve the same result, apparently because of overemphasizing on the "hand roll" phrase. The "vegetarian roll" retrieved by SVM contains a very different list of ingredients from the input "Philadelphia roll", making its nutritional information far from the ground truth the user was interested in retrieving.

Table 3: Accuracy comparison

	Validation Data			Hand-labeled Data			User-labeled Data		
Training Size	wCNN	cCNN	SVM	wCNN	cCNN	SVM	wCNN	cCNN	SVM
$10K$	87.10	79.84	88.30	64.09	61.08	51.15	71.51	70.43	74.03
$100K$	91.77	84.91	90.56	64.61	63.92	51.28	81.95	71.80	74.14
$1M$	**95.23**	87.54	90.52	**66.16**	64.19	51.57	**87.41**	72.41	74.17

Table 4: Sample matching results using 2 different models: SVM and wCNN

Restaurant Name	Menu Section Name	Item Name	Ranking Algorithm	Best Match
Benihana	Sushi Bar	Philadelphia Hand Roll	SVM	Sushi - Vegetarian Hand Roll
			wCNN	**Wasabi Sushi Bar - Philadelphia Roll**
Buca di Beppo Italian Restaurant	Traditional Pasta	Spaghetti with Meat Sauce	SVM	Spaghetti sauce with Meat
			wCNN	**Homemade - Spaghetti Pasta With Meat Sauce**

4.2 Data Collection

In order to optimize any deep learning network parameters, a large amount of training data is required. Collecting necessary data is an important obstacle, because of the uniqueness and complexity of this specific problem. To the best of our knowledge, there is no publicly available training data dedicated to the task of food names matching, and existing text matching datasets are not applicable to this problem.

The main challenge in our matching problem relates to the fact that string/text similarity is not sufficient to assign the correct labels. Food items can look very similar in name, but refer to completely different entities with different nutritional contents, e.g., "spaghetti meat sauce" and "spaghetti *with* meat sauce". Also, in some cases the food descriptions are not so similar, but the items can be still considered as a true match, e.g., "grilled / marinated lamb or chicken or pork small sandwich" and "grilled lamb sandwich recipe".

To address this problem, a set of $4K$ pairs of food names, i.e., (restaurant food name, database food name), were generated and hand-labeled by our in-house experts. The labels assigned to each pair were either *relevant (2), somehow relevant (1)* or *irrelevant (0).* This training set is then expanded to a much larger scale, in order to train the CNN parameters. To this end, we applied a pairwise RankSVM model [9], because (a) a SVM-based model can achieve a reasonable accuracy on smaller training set, and (b) the pair-wise nature of this model allows to make comparisons between relevant and non-relevant labeled instances, and adds more flexibility to the classification task.

Our SVM-based data collection involved multiple steps. First, the labeled data, originally pointwise in nature, are transformed to a pairwise set. Let f be the restaurant food name and c_1, c_2 be two food candidates, with the labels being y_1, y_2, respectively. Then, a pairwise training instance is formed by $((c_1, c_2)|f)$ and is assigned the label $y_1 - y_2$, which is positive if c_1 is a more relevant match for f, compared to c_2, and is negative otherwise. Next, a SVM model is trained on these pairwise instances and employed to make predictions for new instances, along with prediction confidence. The features used in this SVM model are the same as in the pointwise SVM described in the previous section.

Equipped with the trained pairwise SVM model, a set of $200K$ restaurant menus were processed. Every item, f, was input to the query generation model, and a list of candidates $(c_1, c_2, ..., c_n)$ was retrieved from database. Then, the model was used to label $((c_i, c_j)|f)$ with a certain confidence level. This resulted in more than $80M$ labeled pairs, but evidently not all of which were correctly predicted. Therefore, we only kept the instances which were labeled with 99%+ confidence. Each survived pairwise instance, e.g., $((c_i, c_j)|f)$ was then decomposed into two pointwise labeled instances: $(c_i, f) \in relevant$ and $(c_j, f) \in irrelevant$, if $((c_i, c_j)|f)$ was labeled "positive". Consequently, were able to collect more than $1M$ labeled instances, to train our CNN model.

In the second example, SVM's suggested match looks very similar to the input restaurant food, with respect to the string similarity (without the words ordering they are identical!). However, after looking closely, it is evident that the input query is a "spaghetti" containing "meat sauce" as an additional ingredient, while the SVM's output is a type of "sauce" made with "meat". Therefore, they refer to completely distinct entities, hence their nutritional contents are drastically different. On the other hand, wCNN was able to find the correct match, which is close enough to the input food name.

5 CONCLUSION

Across the three main variations of test sets – random partition withheld from the training set, hand-labeled data by humans, and observed responses from users – the wCNN model outperforms the basic SVM technique once the scale of training data is on the scale of $100K$ or more examples, and becomes really evident once the data set size reaches $1M$. Since the limitations of this type of food data can make straightforward techniques difficult or impossible,

it is impressive that the fusion of multiple machine learning techniques working together can reach accuracy levels higher than any alone would be able to achieve. The combination of reinforcement via MDP for initial query generation, SVM for building synthetic training data, and CNN architectures for learning relevance, all come together to create a powerful tool for short text matching in the absence of context and/or user feedback.

REFERENCES

[1] Delphine Bernhard. 2010. Query expansion based on pseudo relevance feedback from definition clusters. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*. Association for Computational Linguistics, 54–62.

[2] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*. ACM, 129–136.

[3] Ben Carterette, Evangelos Kanoulas, and Emine Yilmaz. 2011. Simulating simple user behavior for system effectiveness evaluation. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 611–620.

[4] François Chollet. 2015. Keras: Theano-based deep learning library. *Code: https://github. com/fchollet. Documentation: http://keras. io* (2015).

[5] William Cohen, Pradeep Ravikumar, and Stephen Fienberg. 2003. A comparison of string metrics for matching names and records. In *Kdd workshop on data cleaning and object consolidation*, Vol. 3. 73–78.

[6] Koby Crammer, Yoram Singer, and others. 2001. Pranking with Ranking.. In *Nips*, Vol. 14. 641–647.

[7] Minwei Feng, Bing Xiang, Michael R Glass, Lidan Wang, and Bowen Zhou. 2015. Applying deep learning to answer selection: A study and an open task. *arXiv preprint arXiv:1508.01585* (2015).

[8] Dongyi Guan, Sicong Zhang, and Hui Yang. 2013. Utilizing query change for session search. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 453–462.

[9] LI Hang. 2011. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems* 94, 10 (2011), 1854–1862.

[10] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. 1999. Large margin rank boundaries for ordinal regression. *Advances in neural information processing systems* (1999), 115–132.

[11] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. 2014. Convolutional neural network architectures for matching natural language sentences. In *Advances in Neural Information Processing Systems*. 2042–2050.

[12] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*. 604–613.

[13] Jiepu Jiang and Daqing He. 2013. Pitt at TREC 2013: Different Effects of Clickthrough and Past Queries on Whole-session Search Performance. In *The Twenty-Second Text REtrieval Conference (TREC 2013) Proceedings*.

[14] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* (1996), 237–285.

[15] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[18] Hang Li. 2014. Learning to rank for information retrieval and natural language processing. *Synthesis Lectures on Human Language Technologies* 7, 3 (2014), 1–121.

[19] Yasunari Maeda, Fumitaro Goto, Hiroshi Masui, Fumito Masui, and Masakiyo Suzuki. 2011. The Bayesian Optimal Algorithm for Query Refinement in Information Retrieval. *IJCSNS* 11, 10 (2011), 91.

[20] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, and others. 2008. *Introduction to information retrieval*. Vol. 1. Cambridge university press Cambridge.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of Workshop at ICLR*.

[22] Nikolaos Nanas, Victoria Uren, Anne De Roeck, and J Domingue. 2003. A comparative study of term weighting methods for information filtering. *KMi-TR-128. Knowledge Media Institue, The Open University* (2003).

[23] Kriengkrai Porkaew and Kaushik Chakrabarti. 1999. Query refinement for multimedia similarity retrieval in MARS. In *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*. ACM, 235–238.

[24] Stuart Russell and Peter Norvig. 1995. Artificial intelligence: a modern approach. (1995).

[25] Eldar Sadikov, Jayant Madhavan, Lu Wang, and Alon Halevy. 2010. Clustering query refinements by user intent. In *Proceedings of the 19th international conference on World wide web*. ACM, 841–850.

[26] Aliaksei Severyn and Alessandro Moschitti. 2015. Learning to rank short text pairs with convolutional deep neural networks. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 373–382.

[27] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1930–1941.

[28] Wei Wang, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. 2009. Efficient approximate entity extraction with edit distance constraints. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 759–770.

[29] Wenpeng Yin and Hinrich Schütze. 2015. Convolutional neural network for paraphrase identification. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 901–911.

[30] Xiang Zhang and Yann LeCun. 2015. Text Understanding from Scratch. *arXiv preprint arXiv:1502.01710* (2015).