Short Paper

Recurrent Neural Networks for Dynamic User Intent Prediction in Human-Database Interaction

Vamsi Meduri
Arizona State University
vmeduri@asu.edu

Kanchan Chowdhury
Arizona State University
kchowdh1@asu.edu

Mohamed Sarwat
Arizona State University
msarwat@asu.edu

ABSTRACT

Prediction of human intent during a user interaction session with the database received a significant amount of attention in the recent past [1, 8]. State-of-the-art intent detection approaches such as [5] insist that the human intent is dynamic and is constantly changing throughout the user session. While the usage of classifiers like SVMs and decision trees have been proposed to capture static user intent [2], such models become ineffective in predicting dynamic or ever-changing human intent. Recurrent Neural Networks (RNNs) are powerful temporal predictors and have recently been prominent in the database research community for tasks such as entity matching [3, 6]. In this work, we discuss the application of RNNs to the problem of dynamic user intent prediction during a human-database interaction. We propose two variants of SQL-specific embedding vectors for RNNs. We also propose *active learning* strategies for RNNs which consume a fraction of the held-out training data to produce competitive prediction quality as full training or *supervised learning*. Our experiments on real user sessions upon the NYCTaxiTrip dataset [9] evaluate the effectiveness of vanilla, LSTM and GRU based RNNs.

1 INTRODUCTION

Prediction of user intent during a Human-Database Interaction (HDI) session helps prefetching the results of the anticipated queries [4] thus making the interaction seamless. Existing works such as [1, 2] define the user intent as the last (target) query asked by a user in an interaction session and the prediction of user intent as a binary classification problem. The test data at hand is classified as interesting or not by using a decision tree or a Support Vector Machine (SVM) based on the training the classifier has undergone. The positive class predictions are evaluated against the results of the target SQL query held as ground truth. This line of work assumes that the user intent is static, contrary to which [5] emphasizes that the user can refine and adapt her needs constantly until the termination of an HDI session i.e., the target query changes continuously. Capturing dynamic multi-user intent using [1, 2] requires a binary classifier for each user session. Instead, we can model dynamic intent discovery as a temporal prediction task using a single Recurrent Neural Network (RNN) for all the user sessions because of its ability to train on and predict several sequences of data.

For a query qu_i issued by the user at timestep i, we not only retrieve the results of qu_i but also predict the intent vector of the subsequent query at timestep $i + 1$. We use RNN as the intent prediction middleware between the user and the database as illustrated in Figure 1. A prior embedding layer can convert raw text either into real-valued embeddings or *one-hot* vectors or words in a *vocabulary* which are fed to the RNNs as intent vectors.

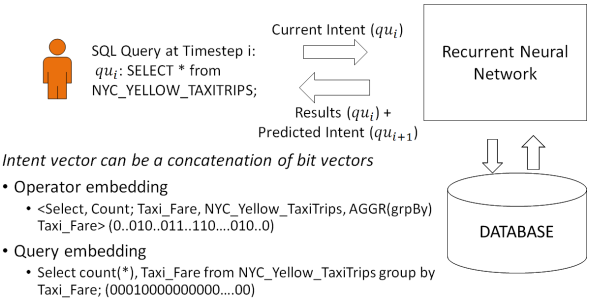

Figure 1: Intent Prediction Pipeline using RNNs

The performance of an RNN is dependent on the granularity at which these embeddings and the corresponding vocabulary are created. For instance, either an entire SQL query or each distinct substring or character in the query can be considered as a word in the vocabulary. In the latter case, external libraries such as word2vec (https://code.google.com/archive/p/word2vec/) or GloVe (https://nlp.stanford.edu/projects/glove/) can be used to borrow pre-trained real-valued embeddings [6].

We create two variants of SQL-specific binary embeddings for dynamic intent vector representation customized to relational databases rather than relying on external libraries which are SQL-agnostic. In the first variant of query embedding, we use a one-hot vector which is a bitmap of 1s and 0s to represent each query intent. It only has a single dimension set to 1 that corresponds to a specific query among the collective set of queries issued by the users until a given point of time. The second variant of operator-based embedding breaks down a SQL query into several smaller bitmaps each corresponding to a distinct family of SQL operators. A concatenation of all the operator-specific bitmaps produces an operator embedding as the user intent for that query.

RNNs are known to consume a lot of training data and time. To reduce the amount of training data, we use active learning on RNN to empirically test if it can make effective predictions with limited user sessions. Active learning selectively includes *ambiguous* (hard-to-classify) examples into the training data and incrementally refines the classifier based on the additional training data. Our approach differs from active learning for static intent in [2] as we select query sequences that is more scalable than labeling tuples. To minimize the training time and enhance interactivity, we propose using an incrementally trained RNN. We present our results on 139 sessions we collected from real user interactions on the NYCTaxiTrip [9] dataset. The remaining paper is organized as follows: we first present the details of the experimental dataset and the construction of the embedding vectors followed by the application of supervised learning and active learning on RNN for dynamic intent prediction. We conclude the paper with a discussion on the experimental results.

2 DATASET AND SYSTEM OVERVIEW

We collected 139 user sessions from 30 real world users with 4 to 5 average sessions per user. Each of them interacted with

a sample of 100,000 trips loaded from a total of 10M taxi trips that span over 5 weeks in June and July 2016. We used the visual interface of Tableau [7] connected to the database engine of PostgreSQL 9.5.10. The reason for sampling is to increase the interactivity between the user and the database without overloading the Tableau interface. The visual interactions are translated into SQL queries and stored in logs by Tableau. We obtained a total of 1958 SQL queries with 1190 distinct queries from the user interactions. Our offline vocabulary creation step ensures that the embedding vectors of all the queries have the same pre-fixed dimensionality. We do not have to handle "UNK"(unknown) tokens as RNNs conventionally do for out-of-vocabulary strings, because of the offline step.

Query Embedding: Query-based intent uses the actual query for intent representation. Each query is thus assigned a specific dimension within the entire vocabulary space of 1190 distinct SQL queries in the dataset. A bitmap of 1190 bits is created for each query of which a single bit is set to 1 specific to the query.

Operator Embedding: The NYCTaxiTrips dataset is stored as a single table and hence each SQL query intent vector is a composite bit vector of all possible operators allowed over a single relation and can be written as $vec(qu_{Aggr}) = \pi_{vec}\ Aggr_{vec}\ \sigma_{vec}\ GROUP\ BY_{vec}\ ORDER\ BY_{vec}\ HAVING_{vec}\ LIMIT_{vec}$. Each query vector $vec(qu_{Aggr})$ is a concatenation of bit vectors for operators such as PROJECT, AGGREGATE, GROUP BY, ORDER BY, HAVING and LIMIT. Every single operator out of these six has a dimensionality equal to the number of attributes $|Attr|$ in the database schema (indicating the columns that the operator can be associated with) except AGGREGATE and LIMIT. $Aggr_{vec}$ is a concatenation of five most common aggregate operators - AVG, MIN, MAX, SUM and COUNT and thus has a dimensionality of $|Attr| * 5$ bits. LIMIT operator contributes only to a single bit in the operator vector as it is not associated with any attribute and is more like a Boolean recording its presence in the query. The total dimensionality of an operator embedding bitmap is thus $|Attr|$x10+1 which is 18x10+1=181 for the NYC taxi trip dataset.

Operator	Operator sub-vector	#Dimensions
π_{vec}	000000000000100000	#Columns = 18
$Aggr_{vec}$	0...011111111111111111	#Columns x 5 = 90
σ_{vec}	0000..0	#Columns = 18
$GROUP\ BY_{vec}$	000000000000100000	#Columns = 18
$ORDER\ BY_{vec}$	0000..0	#Columns = 18
$HAVING_{vec}$	0000..0	#Columns = 18
$LIMIT_{vec}$	0	1

The above table lists the operator (sub-)vectors for each expected category of SQL operators for the example query qu_{i+1} from Figure 1, qu_{Aggr}: *SELECT COUNT(*), taxi_fare FROM nyc_yellow_tripdata GROUP BY taxi_fare*. Let taxi_fare be the 13^{th} attribute among the list of 18 possible attributes. We can see the corresponding bit position set to 1 in the 18-bit operator sub-vector for the projection and group by operator while setting all the attributes for the count operator to 1.

2.1 Supervised Learning

In this section, we describe how RNNs are trained and tested for dynamic intent prediction. We use Keras API (https://keras.io/) with TensorFlow library (https://www.tensorflow.org/) for RNNs. Figure 2 illustrates RNN for two consecutive timesteps T_i and T_{i+1} in intent prediction. We use a "many-to-one" sequential RNN with an input layer, a hidden layer and an output layer. The input layer is fed with the sequence of query/operator embedding

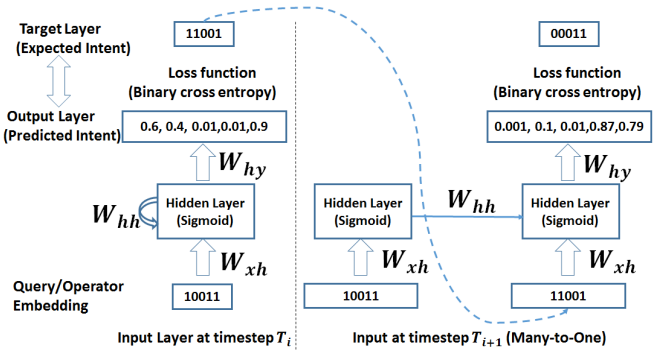

Figure 2: Recurrent Neural Network for Intent Prediction

vectors until the current timestep in the session to get a temporal prediction of the intent for the next timestep. The output layer emits a probability vector with same dimensionality as the intent vector. The value of each dimension indicates the probability of that dimension being a 1. For example, in Figure 2, the output vector $< 0.6, 0.4, 0.01, 0.01, 0.9 >$ at T_i denotes the probabilities of each of its five dimensions being 1.

During the training phase, the predicted probabilistic output vector is compared to the target query which is the successor query from the next timestep or the expected intent vector output (of 11001 at T_i in the figure) using a loss function. The hidden layer weights (w_{xh}, w_{hh}, w_{hy}) are updated using backpropagation either by fitting the model simultaneously on the sequences across all the timesteps so far (*fully trained*) or updating the model obtained thus far by fitting it only on the latest sequence (*incrementally trained*). We limit the number of learning epochs to 10 during training for interactivity. We append the target vector of timestep T_i to the session sequence as input to the RNN at the subsequent timestep T_{i+1}. We empirically choose sigmoid activation function for hidden layer and binary cross entropy as the loss function at the output layer. During the test phase, instead of predicting the most likely intent vector, we predict the top-K candidate output intent vectors having the highest cosine similarity to the probabilistic output vector. These top-K vectors are picked from the historical sessions that the RNN has been trained on, and this ensures that arbitrary bitmaps which may not correspond to a meaningful SQL query are not predicted.

2.2 Active Learning

Active learning can be applied to RNN to test if it can perform well with limited training data. Instead of learning the RNN upon all the training session queries, the training set can be divided into two parts - session query sequences which are made available to the RNN, and query sequences held out from the RNN. By training the RNN upon the available query sequences and selecting informative sequences from the held-out data for inclusion into the training set, the RNN is incrementally refined to verify if it achieves high F1-scores on the test set without exhausting the entire training set. It should be noted that by query qu_i, we refer to its intent vector throughout this section.

For a given user session with queries qu_1, qu_2 and qu_3, the temporal sequences constructed are $qu_1 \rightarrow qu_2$ and $qu_1, qu_2 \rightarrow qu_3$ as shown in Figure 3. A held-out session means that the antecedent (e.g., qu_1, qu_2 in $qu_1, qu_2 \rightarrow qu_3$) of a temporal sequence dependency is available while keeping the consequent (e.g, qu_3 in $qu_1, qu_2 \rightarrow qu_3$) invisible. Thus, the trained RNN is supposed to predict the consequent of each temporal dependency given its

Algorithm 1: Active Learning for intent prediction

input : Tr_A: Available training set of query sequences
Tr_H : Held-out training set of query sequences
$Test$: Test set of query sequences
L: learning algorithm for a classifier
output: C^*: An optimal temporal predictor of query intents
$qu_{predicted}$: Test query successors predicted by C^*

1 $i \leftarrow 0$
2 **while** $\{Tr_H\} \neq \phi$ **do** // stopping criterion
3 $C_i \leftarrow$learnClassifier(**L**, Tr_A)
4 $seq_{ambig} \leftarrow$ pickAmbiguousQuerySequence(**L**,Tr_H)
5 $seq_{labeled} \leftarrow$ obtainSuccessors(seq_{ambig})
6 $Tr_H \leftarrow Tr_H - seq_{labeled}$
7 $Tr_A \leftarrow seq_{labeled} \cup Tr_A$
8 $qu_{predicted} \leftarrow$ predictSuccessors($Test$, C_i)
9 $Test_{F1} \leftarrow$ computeF1($Test$, $qu_{predicted}$)
10 $i \leftarrow i + 1$
11 $C^* \leftarrow C_{i-1}$
12 return C^*, $qu_{predicted}$

antecedent. Among all such held-out sequences, if the RNN finds a particular sequence to be the hardest to predict the successor for, it is deemed to be an ambiguous sequence for RNN. The consequent or the successor query of that particular temporal sequence is made available to the RNN and is included into the available training set. In Figure 3, $qu_{11}, qu_{12}, ..., qu_{20} \rightarrow qu_{21}$ is the hardest held-out temporal sequence and hence, it is included into the training set.

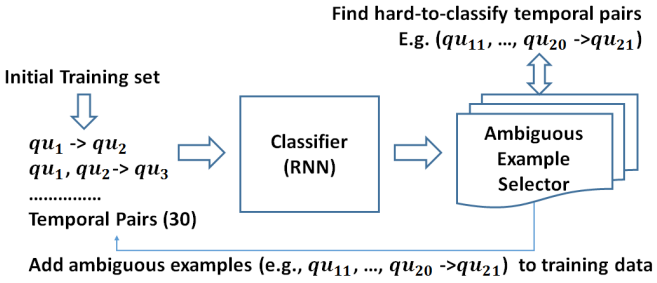

Figure 3: Active learning for RNNs

Algorithm 1 describes the active learning based intent prediction algorithm using RNN. Tr_A and Tr_H are the available and held-out training datasets. In each active learning iteration, the most ambiguous query sequence is picked from Tr_H for which the successor query is the hardest to predict and the corresponding successor query is made available to RNN (lines 4 and 5). The query sequence is removed out of Tr_H and is added to Tr_A (lines 6 and 7). At the end of each iteration, the RNN learned thus far is evaluated on the fold test data until the held-out training set is exhausted. The purpose of using active learning algorithm is to verify if all the training set needs to be completely exhausted during learning to achieve a significantly high enough test F-measure. While ambiguous sequence selection is time consuming, if it helps active learning reach an early convergence to high F-measures, it is a trade-off worth considering.

2.2.1 Ambiguous Example Selection Strategy (Minimax). We have described in section 2.1 that the output layer of the RNN emits a probability vector. We compute the cosine similarity between the probability vector and the intent vectors of all the historical session queries that the RNN has learned from until then, and select the top-K intent vectors with the highest cosine similarity as the predicted intents. Hence, we can use the value of the maximum cosine similarity as the confidence measure and the inverse of the confidence value denotes the ambiguity. Among several temporal held-out sequences $seq_1, seq_2, ..., seq_n$, the one for which predicting the successor query is the hardest is inferred based on the value of the highest cosine similarity between the weight vector and the historical intent vectors. If the corresponding values of the highest cosine similarities are $maxSim_1, maxSim_2, ..., maxSim_n$, the most ambiguous query sequence is seq_i if i=$arg\ min_{i=1}^{n}$ $\{maxSim_i\}$, i.e., it has the least maximum cosine similarity (minimax) among all the sequences.

3 EXPERIMENTS

All our experiments were conducted on a Mac machine with a 4-core 2.8 GHz Intel Core i5 processor, 16GB RAM and 1.02 TB hard disk. Our experiments for supervised learning were conducted on queries arriving in an interleaved order from concurrent user sessions. Active learning experiments were conducted on 10-fold splits of the data into 10 different training and test set pairs with 90% and 10% of the user session queries respectively. We compute the precision and recall as follows:

$$Precision = \frac{\#\{PredDim_i=1 \cap ActualDim_i=1\}}{\#\{PredDim_i=1\}}$$

$$Recall = \frac{\#\{PredDim_i=1 \cap ActualDim_i=1\}}{\#\{ActualDim_i=1\}}$$

Each dimension $PredDim_i$ in the predicted intent vector is compared to the corresponding dimension $ActualDim_i$ in the actual vector, and the hits and misses are measured based on the fraction of 1s predicted accurately. We report F1-scores in our results.

Concurrent User Sessions: The queries arrive in batches of 10 and hence the 1958 queries from the NYCTaxiTrip dataset are divided across 196 learning episodes. While the RNN gets updated at the end of each learning episode with additional training data obtained from the queries during the episode, it is tested throughout the episode with the arrival of new query. For each new query, its embedding (query or operator) vector is fed to the RNN, which predicts the top-K intent vectors for the next query. K is set to 3 and we report the maximum F-measure among the top-3 predictions by averaging it across all the queries in each episode. Response time is the sum of the intent vector creation and execution times for the current query added to the intent vector prediction time for the next query. It is heavily dominated by the RNN prediction time which also includes its cumulative training time based on all the queries seen thus far. For incrementally trained vanilla (simple backpropagation) RNN, the training time depends only on the batch of queries from the latest episode.

LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit) perform same as vanilla RNN on the operator embedding intent vector (Figure 5) but GRUs perform slightly better than LSTM and vanilla on query embedding (Figure 4) vectors. The reason for this is that the average user session length is ≈18 queries per session which is not lengthy enough for advanced RNN variants to exploit the long short term dependencies. On an average, vanilla RNN seems to give competitive F-measures while also consuming lesser response time than LSTM and GRU (Figure 6). Incrementally trained RNN sacrifices test F-measures on query embedding but produces comparable quality as fully trained RNNs on operator embedding while incurring very low response times. Time charts on query embedding demonstrate similar patterns as operator embedding for both supervised and active learning which is why we omitted them for brevity.

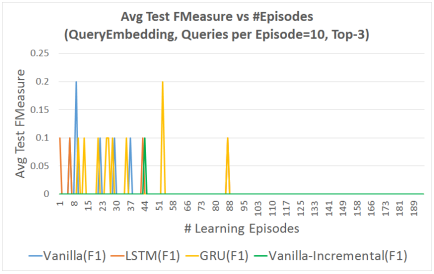

Figure 4: Concurrent Sessions (Query)

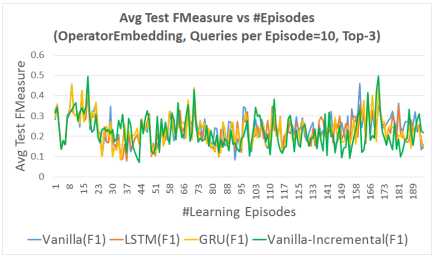

Figure 5: Concurrent Sessions (Operator)

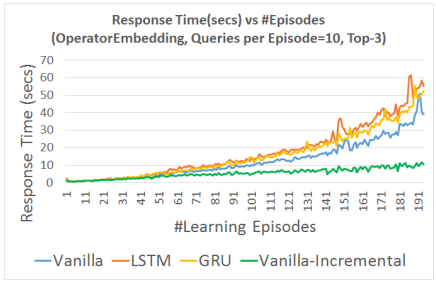

Figure 6: Response Times (Operator)

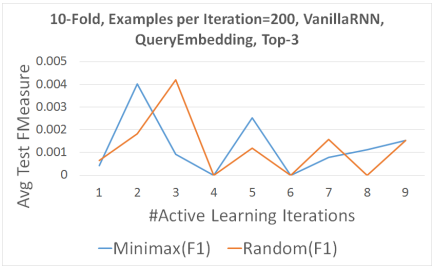

Figure 7: Active Learning (Query)

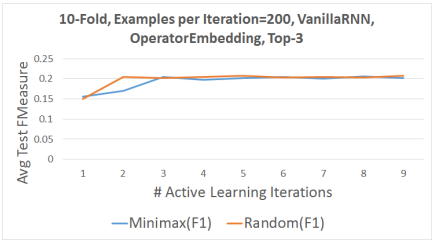

Figure 8: Active Learning (Operator)

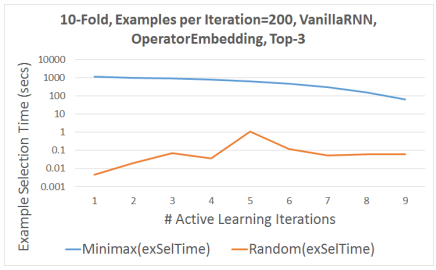

Figure 9: Selection Times (Operator)

A general observation is that operator embedding is more expressive and produces better F-measures than query embedding because predicting SQL operators in a query is easier than predicting the query in its entirety. We can also notice that the initial queries in concurrent sessions are easier to predict than the later ones. In the real world sessions we collected, each user arrives at a distinct goal by the end of the exploratory session. The first few queries may be similar across users, but each session becomes specialized with unique queries towards the end, thus inducing little overlap across sessions. This in turn makes workload prediction harder as the sessions progress concurrently.

Active Learning: Our 10-fold active learning experiments were upon the fully trained vanilla version of RNN using standard backpropagation algorithm. There were ≈1648 examples (temporal query sequence dependencies) in the training set and ≈190 test examples. We started with a seed available training set of 30 randomly chosen training examples and 1618 held-out training examples. We selected 200 examples in each iteration from the hold-out set and added them to the available set. We plot the average test F-measures over all the test sessions across 10 folds at each active learning iteration in Figures 7 and 8.

We compared the minimax example selection strategy described in section 2.2.1 with random selection. Our observations show that random strategy performs competitively to minimax on operator embedding as shown in Figure 8 and achieves earlier convergence to the eventual F-measure in the 2^{nd} iteration with 230 examples (14% training data). Minimax achieves slower convergence by iteration 3 with 430 examples and 26% training on operator embedding. Examples chosen in an iteration are used for training in the next iteration i.e., #training examples in iteration i = 30 + 200 x (i-1). Figure 7 shows that both strategies perform similarly on query embedding. A major drawback of using query embedding for active learning is the non-monotonic behavior with increasing iterations. This is due to the heavy sparsity in query embedding vectors (1 bit set out of 1190 dimensions) and fitting RNNs to more training points does not necessarily yield higher test F-measures. Minimax strategy incurs more example selection time than random selection (Figure 9) because of the expensive prediction done over the hold-out set which shrinks with increasing iterations thus also decreasing the selection times.

4 CONCLUSION

In this paper, we proposed the application of RNNs for dynamic intent prediction and two SQL specific embedding techniques for intent vector creation. Our experiments show that operator embedding is more effective than query embedding and that vanilla RNNs perform better than LSTMs or GRUs for user sessions of moderate length (#queries). We also show that an incrementally updated vanilla RNN model achieves substantially lesser response times than fully trained RNN models while making little to no sacrifice in prediction quality. Our active learning experiments show that random selection strategy achieves earlier convergence to competitive test F-measures as full training with just 14% of the training examples on operator embedding and lesser example selection time than minimax strategy.

REFERENCES

[1] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An Automatic Query Steering Framework for Interactive Data Exploration. In *SIGMOD*. 517–528.

[2] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2016. AIDE: An Active Learning-Based Approach for Interactive Data Exploration. *IEEE TKDE* 28, 11 (2016), 2842–2856.

[3] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *Proc. VLDB Endow.* 11, 11 (July 2018), 1454–1467.

[4] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and interactive cube exploration. In *ICDE*. 472–483.

[5] Ben McCamish, Vahid Ghadakchi, Arash Termehchy, Behrouz Touri, and Liang Huang. 2018. The Data Interaction Game. In *SIGMOD*. 83–98.

[6] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*. 19–34.

[7] Dan Murray. 2013. *Tableau Your Data!: Fast and Easy Visual Analysis with Tableau Software* (1st ed.). Wiley Publishing.

[8] Mohamed Sarwat, Raha Moraffah, Mohamed F. Mokbel, and Jamed L. Avery. 2017. Database System Support for Personalized Recommendation Applications. In *ICDE*. 1320–1331.

[9] Taxi and Limousine Commission. 2016. NYC Taxi Trip Dataset. {http://www.nyc.gov/html/tlc/html/technology/raw_data.shtml}. (2016).