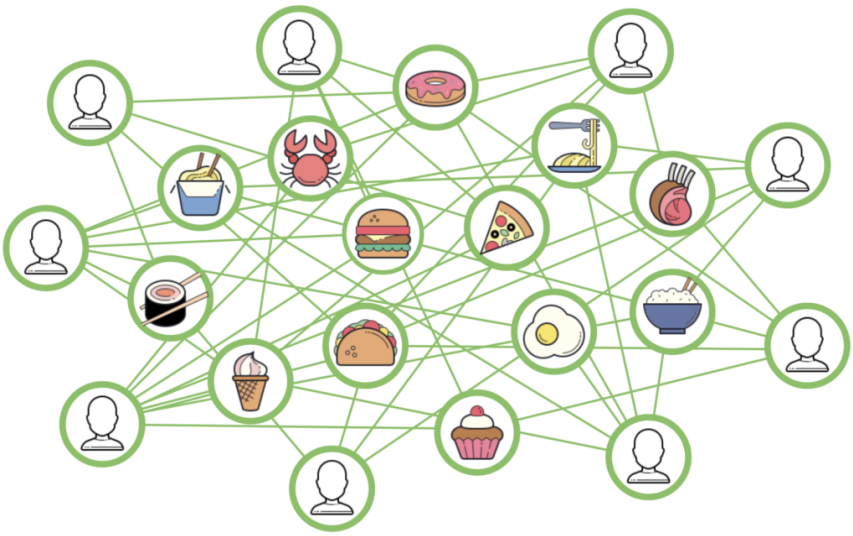
Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations

Ankit Jain, Isaac Liu, Ankur Sarda, and Piero Molino

December 4, 2019



The Uber Eats app serves as a portal to more than 320,000 restaurant-partners in over 500 cities globally across 36 countries. In order to make the user experience more seamless and easy-to-navigate, we show users the dishes, restaurants, and cuisines they might like up front. To this end, we previously developed ML models to better understand queries and for multi-objective optimization in Uber Eats search and recommender system in Uber Eats searches and surfaced food options.

Existing research [1] has shown the efficacy of graph learning methods for recommendation tasks. Applying this idea to Uber Eats, we developed graph learning techniques to surface the foods that are most likely to appeal to an individual user. Productionizing this method improves the quality and relevance of our food and restaurant recommendations on the platform.

Graph learning in a nutshell

To best understand how we made our Uber Eats recommendations more accurate, it helps to know the basics of how graph learning works. Many machine learning tasks can be performed on data structured as graphs by learning representations of the nodes. The representations that we learn from graphs can encode properties of the structure of the graph and be easily used for the above-mentioned machine learning tasks. For example, to represent an eater in our Uber Eats model we don't only use order history to inform order suggestions, but also information about what food items are connected to past Uber Eats orders and insights about similar users. Specifically, in order to obtain representations with such properties, we calculate a vector for each node in the graph (users, restaurants and food items in this case) such that node vector similarity approximates the strength of the connection between two nodes in the graph. Our objective is to find a that maps from a node to its vector representation (an encoding function) such that nodes that are structurally similar in the graph have similar representations.

For our Uber Eats use case, we opted for a graph neural network (GNN)-based approach to obtain an encoding function. This type of approach, although being initially proposed in the late 1990s and early 2000s [2, 3] have recently been adopted extensively by the research community for a variety of tasks [6,7,8] and has been shown particularly effective for recommendation problems [1].

The basic idea behind GNNs consists of using a neural network to obtain a representation for a node by aggregating the representations of neighboring nodes in a recursive fashion limited to a certain depth, as shown in Figure 1, below:

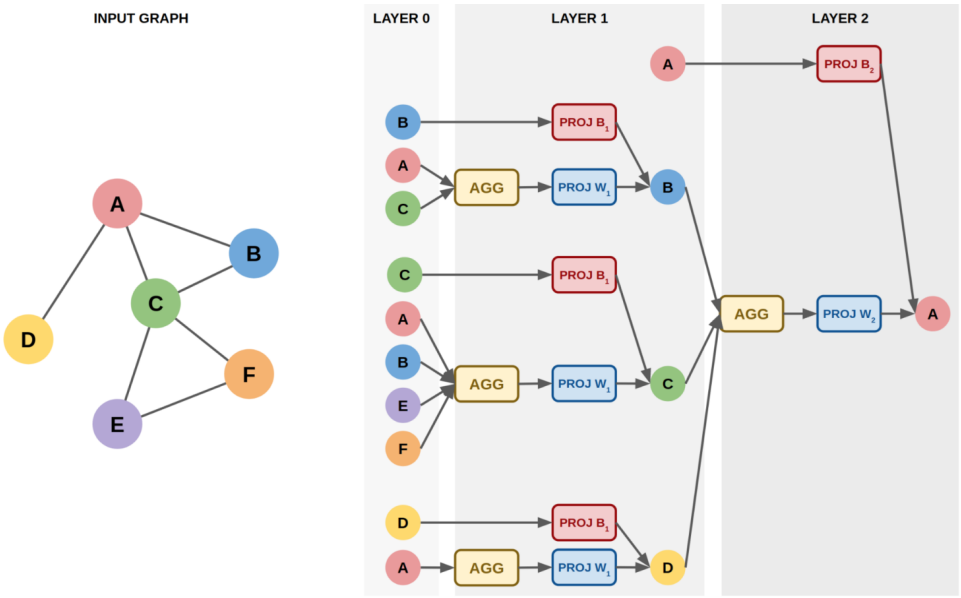


Figure 1: A graph neural network (on right) obtains the representation of node A from an input graph (on left).

Assuming we limit the depth of the recursion to two to obtain the representation of the node A in Figure 1, we first perform a breadth-first search starting from *A*. Next, we obtain the features *x* of the nodes at two steps removed from *A*. Features are aggregated by an aggregation/pooling function, for instance, by taking the average and projected by a matrix multiplication with a learned weight matrix *W* (PROJ W in the figure) to obtain a representation of the neighborhood of the nodes at one hop distance from *A*.
This neighborhood representation is combined with information about the node itself projected by a matrix multiplication with a learned weight matrix *B* (PROJ B in Figure 1), and this combination forms the representation *h* of nodes at a distance one from the node *A*. This representation is recursively aggregated and projected to obtain the representation of node *A*, and at each step of the recursion, new matrices *W* and *B* are used (*W1 and B1* for Layer 1 and *W2 and B2* for Layer 2 in Figure 1). The main advantage of obtaining a representation in this manner is that it captures both the properties of node A and the structural information about its neighborhood, aggregating information about the nodes that node A connects to, as depicted in Figure 2, below:

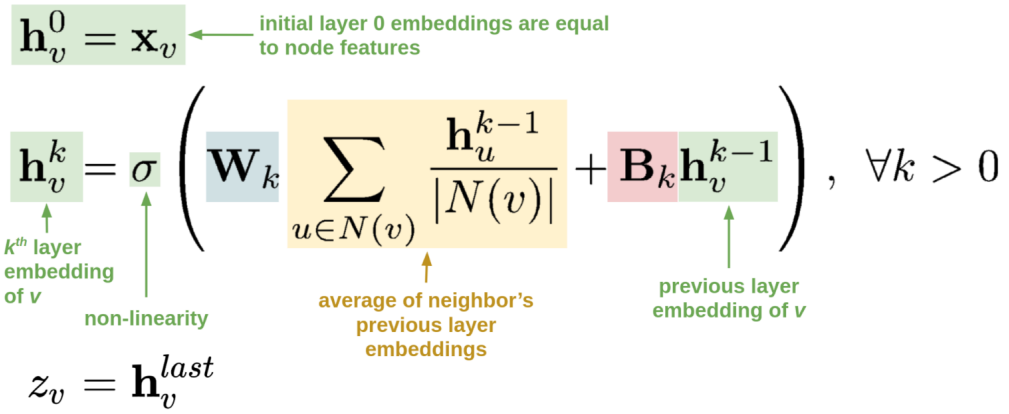


Figure 2: These equations represent the computation graph displayed in Figure 1.

We then use the node representations to predict the probability that a connection between two nodes exists and optimize a loss that maximizes the probability of two nodes that are actually connected in the graphs and minimizes the probability of disconnected nodes.
GNNs require only a fixed amount of parameters that do not depend on the size of the graph, making learning scalable to large graphs, especially if the neighboring nodes are sampled to be a certain fixed amount when obtaining the representation of a specific node. Moreover, a representation can be induced for a newly added node by virtue of its basic features and connections. These features of GNNs support recommendations at scale on Uber Eats, which adds new users, restaurants, and dishes daily.

Graph learning for dish and restaurant recommendation at Uber

There are several recommendation surfaces within the Uber Eats app, depicted in Figure 3, below:

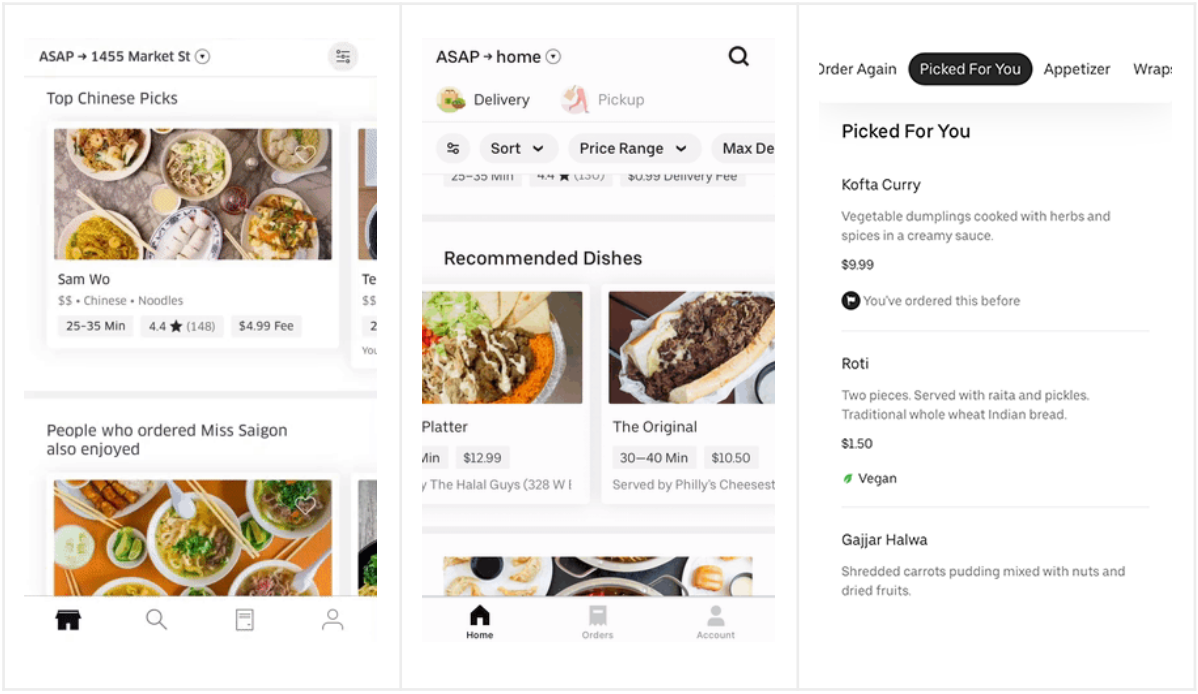


Figure 3: The Uber Eats UI surfaces a wealth of options for hungry users informed by past orders and previously specified user preferences.

On the main feed, we generate recommendation carousels for both restaurants and menu items based on user preferences. When browsing the menu of a restaurant, we also generate personalized recommendations of items within that restaurant to suit a user's tastes. These suggestions are made by recommender systems trained on past orders and user preferences.

The Uber Eats recommendation system can be broken down into two phases: candidate generation and personalized ranking.

The **candidate generation** component generates relevant candidates, in other words, dishes and restaurants, in a scalable fashion. We needed to make this phase highly scalable to enable pre-filtering of the huge and ever-growing number of dish and restaurant options on the platform. Pre-filtering can be based on factors such as geographical location, so we do not recommend a restaurant to a user that is out of its delivery range. Dish and restaurant candidates also need to be relevant to the given user to ensure we are not filtering out items they would like.

The second component of this system, the **personalized ranker**, is a fully-fledged ML model that ranks the pre-filtered dish and restaurant candidates based on additional contextual information, such as the day, time, and current location of the user when they open the Uber Eats app. An example of a recurring order pattern the model can learn to capture includes ordering certain types of food on specific days of the week or different types of dishes for lunch and dinner.

In order to use GNNs to improve Uber Eats recommendations, we create two bipartite graphs: one that represents users and dishes as nodes with edges representing the number of times a user ordered a specific dish, and a second graph which represents users and restaurants as nodes, and edges represent how many times a user ordered from a specific restaurant.

We chose GraphSAGE [4], a specific flavor of GNN in which the aggregation function is a max or mean pooling after a projection, for our modeling starting point because of its strong scalability. In this GNN, the combination of node information and neighbor information is obtained through concatenation. Additionally, GraphSAGE adopts a sampling strategy to constrain the number of nodes sampled at one and two-hop distance from the node of which we want to obtain the representation, making it possible to scale learning to graphs with billions of nodes and providing even better suggestions.

In order to apply GraphSAGE to our bipartite graphs, we had to modify it in a few ways. First, since each node type may have different features, we needed to add an additional projection layer to the GNN. This layer projects the input features into a vector of the same size depending on the type of input node (user, restaurant, or dish). For instance, since dishes can be represented by the word embeddings from their descriptions or features of their associated images, and restaurants can have basic features related to their menu and cuisine offerings, their feature size is different, but the projection layer needs to project them in a space of the same size.

Moreover, GraphSAGE only considers graphs with binary edges, but in our case the edges need to be weighted to include information about the number of times a user orders from a restaurant or a specific dish and the rating given by a user to a dish, as these are very important signals. For this issue, we introduced a few new concepts to add weights on the edges. The most impactful change was adopting a hinge loss, a type of loss that fits the ranking of items with respect to the user better than using binary edges.

$$L = \sum_{(u,v) \in E} \max(0, -z_u z_v + z_u z_n + \Delta)$$

Given a user *u* ordering a dish *v* at least one time, a weighted edge between them exists in the graph. If we want to predict a score for this pair of nodes that is higher than the score that we predict for the same node *u* and a randomly selected node *n* that is not connected to it (a dish the user never ordered), the difference between the scores should be greater than a margin .
The problem with this loss is that edges with a high weight and edges with a low weight are treated interchangeably, which doesn't work well given the difference between a dish a user ordered once and a dish a user ordered ten times. For this reason, we introduced the concept of low-rank positives in the loss.

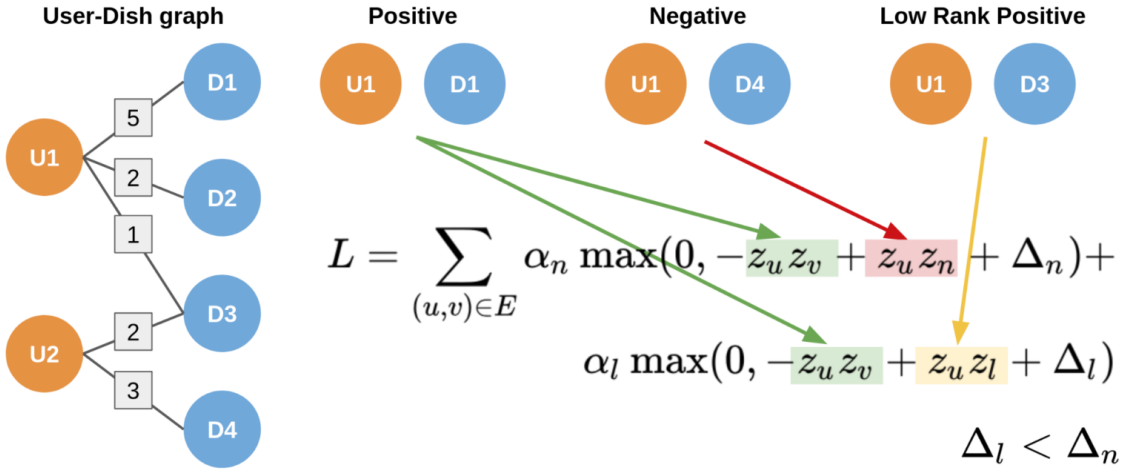


Figure 4: Our Uber Eats recommendation system leverages max-margin loss augmented with low rank positives.

Figure 4, above, shows an example of how our system leverages low-rank positives to revise our loss. Given a positive edge <*u, v*>, a low rank positive is an edge <*u, l*> where the node *u* is the same, but the node *l* is different from *v* and the weight on the edge of <*u, l*> is lower than the weight on <*u, v*>. We added a second piece to the loss to ensure that edges with higher weight are ranked higher than the edges with lower weight with a margin Δ_l, which we set to a value lower than Δ_n, the margin for the negative samples. Both pieces of the loss have a multiplier, α, a hyper-parameter controlling the relative importance of both the negative sample part of the loss and the low rank positive part of the loss.

Finally, we also used the weights in the aggregation and sampling functions. Once we obtain the representations of the nodes using the trained GNN, we can use the distance between the node representations to approximate the similarity between them. Specifically, we added the dot product and cosine similarity of user and items to both our dish and restaurant recommender

systems as features, and tested them both offline and online to determine their accuracy.

To evaluate how useful the embeddings are for our recommending task, we trained the model on four months of historical data up to a specific split date. We then tested the model performance on recommending dishes and restaurants using order data from the ten days following the split date. Specifically, we computed the cosine similarity between a user and all the dish and restaurant embeddings in the city and computed the rank of the dish and restaurant that the user ordered. During the experiment we observed a performance boost of over ~20 percent compared to the existing production model on metrics like Mean Reciprocal Rank, Precision@K, and NDCG.

The improved performance obtained from the embeddings trained with graph learning convinced us to add them as features in our Uber Eats recommendation system's personalized ranking model. When we trained the personalized ranking model with the graph learned embeddings similarity feature, we saw a 12 percent boost in AUC compared to the existing productionized baseline model, leading to improved recommendations for users.

Moreover, analyzing the impact of the feature on our predictions, we saw that the graph learning similarity feature was by far the most influential feature in the recommendation model. This gave us confidence that the graph learned embeddings captured more information than any existing feature in our system, as depicted in Figure 5, below:

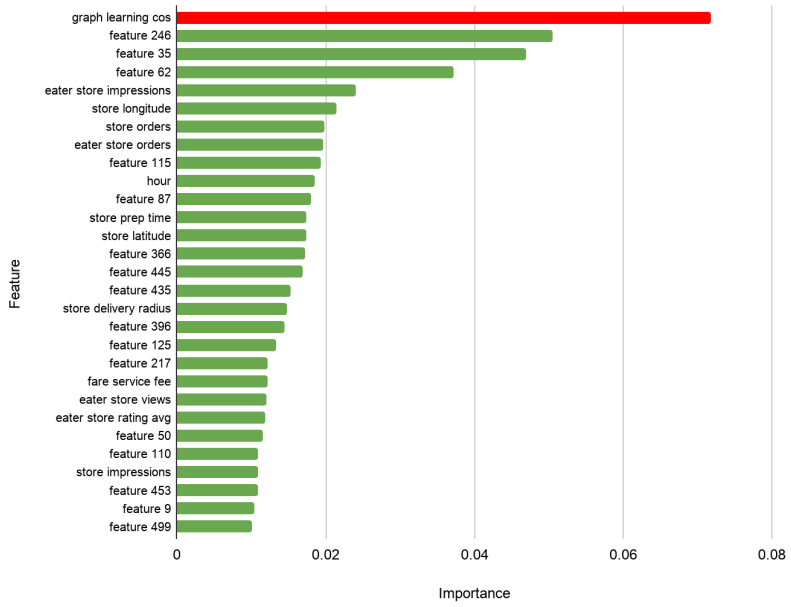


Figure 5: Our new graph learning feature proved the most valuable of all other implemented features when determining the quality and relevancy of our Uber Eats dish and restaurant recommendation systems.

Given the offline results, we felt comfortable rolling out the new model in an online experiment. We conducted an A/B test in San Francisco and observed a substantial improvement in engagement and click-through rate when leveraging the graph learning feature compared to the previous production model, demonstrating that the surfaced dishes predicted by our model appealed more to Uber Eats users.

Data and training pipeline

Once we determined the positive impact of graph learning on our recommendation system, we built a scalable data pipeline to both train models and obtain predictions in a real-time production environment.
We train separate models for each city, as their graphs are only loosely connected.
In order to do this, we used anonymized, aggregated order data from the past several months available and designed a four-step data pipeline to transform the data into the networkx graph format that is required to train our models. The pipeline also extracts aggregated features not directly available in the raw order data, like the total number of times users ordered dishes, which determines the weight of the graph's edges.
Additionally, the pipeline is also capable of creating graphs for older time frames, which can be used for offline analysis. The overall pipeline is depicted in Figure 6, below:

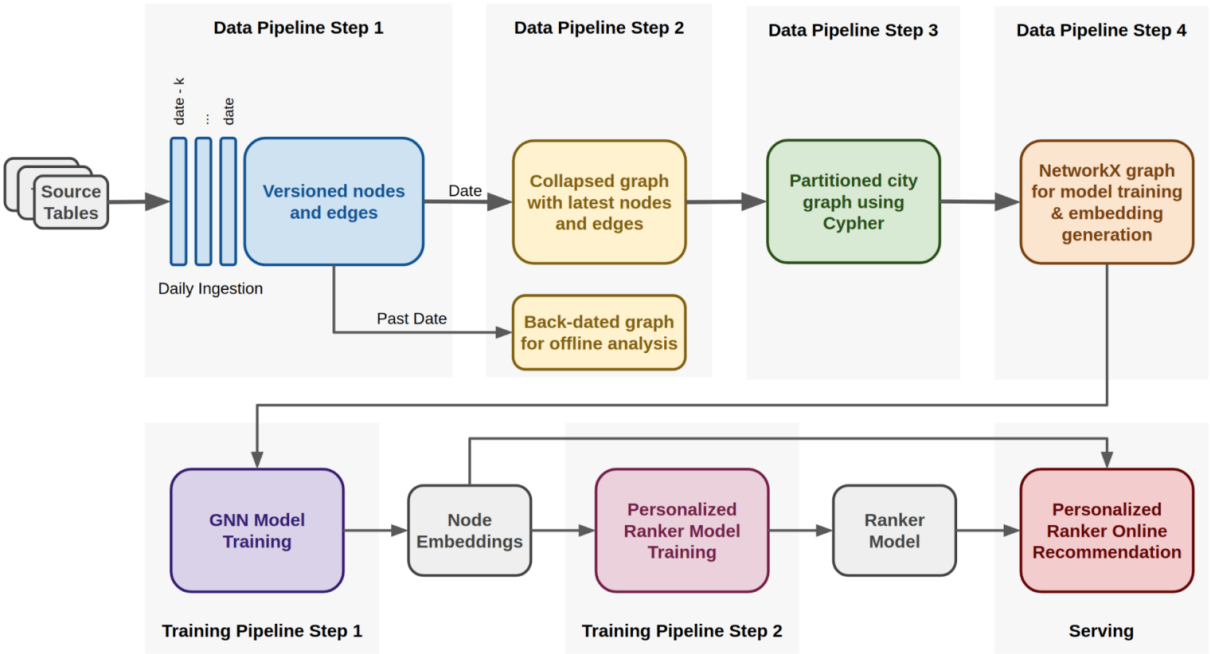


Figure 6: We built a data pipeline (top row) and training pipeline (bottom row) that helps us train our Uber Eats recommendation system using GNN embeddings for improved in-app dish and restaurant suggestions.

In the first step of the pipeline, multiple jobs pull data from Apache Hive tables, ingesting it into HDFS as Parquet files containing nodes and edges information respectively. Each node and edge has properties that are versioned by timestamp, which is needed for constructing back-dated graphs.

In the second step, we retain the most recent properties of each node and edge given a specific date and store them in HDFS using Cypher format. When training production models, the specified date is the current one, but the process is the same also if past dates are specified for obtaining back-dated graphs.

The third step involves using the Cypher query language in an Apache Spark execution engine to produce multiple graphs partitioned by city.
Finally, in the fourth step we convert the city graphs into the networkx graph format, which is consumed during the model training and embedding generation process, which are implemented as TensorFlow processes and executed on GPUs.

The generated embeddings are stored in a lookup table from which they can be retrieved by the ranking model when the app is opened and a request for suggestions is issued.

Visualizing learned embeddings

In order to provide an example capable of characterizing what is learned by our graph representation learning algorithm, we show how the representation of a hypothetical user changes over time.
Assuming we have a new user on Uber Eats who ordered a Chicken Tandoori and a Vegetable Biryani (both Indian dishes), we obtain a representation for such user at this moment in time.
The same user later orders a few other dishes, including: Half Pizza, Cobb Salad, Half Dozen Donuts, Ma Po Tofu (a Chinese dish), Chicken Tikka Masala and Garlic Naan (three Indian dishes). We obtain a representation of the user after these additional orders and we compute the distance of those two representations with respect to the most popular dishes from different cuisine types and display it in Figure 7 below using the explicit axes technique introduced in Parallax: Visualizing and Understanding the Semantics of Embedding Spaces via Algebraic Formulae.

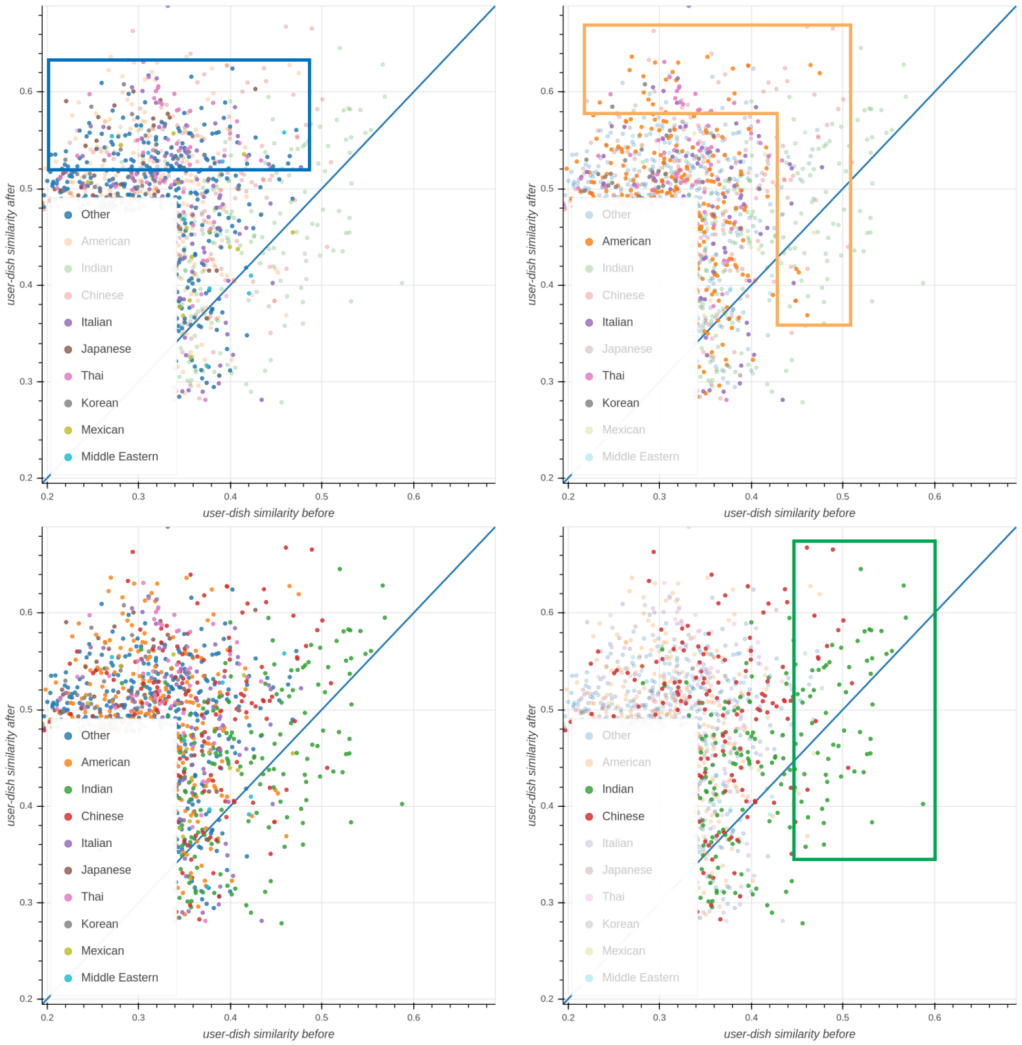


Figure 7: We compared the representation of a hypothetical user before and after ordering dishes and compared them to popular dishes from different cuisines. The four plots highlight dishes belonging to four different subsets of cuisines. The x axis measures how much a dish is similar to the user representation before ordering additional dishes, while the y-axis measures how a dish is similar to the the user representation after ordering additional dishes.

In the bottom left section of Figure 7, clear patterns emerge. The first pattern is highlighted in the green box in the bottom right: the dishes closest to the user representation before the additional orders are almost all Indian dishes (green dots) as expected given the fact that the initial orders were both of Indian food, but also some Chinese dishes end up ranked high on the x-axis, suggesting a second order correlation between these cuisine types (i.e., users who ordered many Indian dishes also ordered Chinese ones). Chinese dishes also rank pretty high on the y-axis, suggesting that ordering Ma Po Tofu influenced the model to suggest more Chinese dishes.

In the top right section of Figure 7, a second pattern is highlighted in the orange box: American, Italian, Thai, and Korean dishes are selected, showing how they are much closer to the user representation after the user ordered additional dishes. This is due to both ordering Pizza, Doughnuts, and the

Cobb Salad, but also due to second order effects from the increase of Chinese suggestions, as users ordering Chinese dishes are also more likely to order Thai and Korean.

Finally, in the top left section of the image, a third pattern is highlighted in the blue box: all the cuisines that are not among the top three closest to both user representations ended up increasing their similarity substantially after their subsequent orders, which suggests that the model learned that this specific user might like for new cuisine suggestions to be surfaced.

Future directions

As discussed, graph learning is not just a compelling research direction, but is already a compelling option for recommendation systems deployed at scale.
While graph learning has led to significant improvements in recommendation quality and relevancy, we still have more work to do to enhance our deployed system. In particular, we are exploring ways to merge our dish and restaurant recommendation tasks, which are currently separate, because we believe they could reinforce each other. Over time, we plan to move from two bipartite graphs to one single graph that contains nodes of all the entities. This will require additional work on the loss and aggregation function to work properly, but we believe it will provide additional information to both tasks leveraging common information.
Another limitation we want to tackle is the problem of recommending reasonable items to users even in situations with data scarcity, such as in cities that are new to the Uber Eats platform. We are conducting research in this direction through the use of meta graph learning [5] with encouraging results.
Interested in how Uber leverages AI to personalize order suggestions on Uber Eats? Learn more about our Uber Eats recommendation system through our Food Discovery with Uber Eats series:

- Food Discovery with Uber Eats: Recommending for the Marketplace
- Food Discovery with Uber Eats: Building a Query Understanding Language

Acknowledgments

We are grateful for the contributions of Jimin Jia, Alex Danilychev, Long Tao, Santosh Golecha, Nathan Barrebbi, Xiaoting Yin, Jan Pedersen, and Ramit Hora to this research.
The icons in the header are obtained from icons8.com.