Learning and Applying Airbnb Listing Embeddings in Two-Sided Marketplace

Siarhei Bykau
siarhei.bykau@airbnb.com
Airbnb, Inc.
San Francisco, CA, USA

Dekun Zou
dekun.zou@airbnb.com
Airbnb, Inc.
San Francisco, CA, USA

ABSTRACT

This paper presents a study on the application and learning of listing embeddings in Airbnb's two-sided marketplace. Specifically, we discuss the architecture and training of a neural network embedding model using guest side engagement data, which is then applied to host-side product surfaces. We address the key technical challenges we encountered, including the formulation of negative training examples, correction of training data sampling bias, and the scaling and speeding up training with the help of in-model caching. Additionally, we discuss our comprehensive approach to evaluation, which ranges from in-batch metrics and vocabulary-based evaluation to the properties of similar listings. Finally, we share our insights from utilizing listing embeddings in Airbnb products, such as host calendar similar listings.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Applied computing → Electronic commerce**; • **Information systems** → *Data analytics.*

KEYWORDS

Embeddings, two-tower neural network model, negative sampling, distributed training, two-sided marketplace

ACM Reference Format:
Siarhei Bykau and Dekun Zou. 2024. Learning and Applying Airbnb Listing Embeddings in Two-Sided Marketplace. In *Proceedings of Workshop on Two-sided Marketplace Optimization: Search, Pricing, Matching & Growth in conjunction with KDD Conference (TSMO '24).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Airbnb is an online marketplace for sharing homes and experiences. Hosts list their properties, which guests book for their stays. In order to facilitate the matching of listings and guests, Airbnb provides numerous products and services to both hosts and guests. Many of these tools are based on the ability to compare listings, i.e. finding similar listings or listings that may be viewed as equivalent substitutes. A naive approach of comparing attributes of listings enables interpretable and efficient comparisons, however, it cannot capture the nuances of listing properties, their complex features such as text and images; and it is not easily tunable for product needs. In this paper, we present an embedding based solution to quantitatively measure similarities of Airbnb listings.

Originally developed for text [13, 14], embedding models have been generalized to other domains of recommender systems [16] and have been used for text, image, product, document, audio items. The key idea is to find an encoding function that translates item's features (in our domain, it is a listing) into a vector (embedding) which represents the item in a high dimensional space. This approach has also been adopted within other two-sided marketplaces [2, 3, 5] to improve its efficiency.

In particular, we use guest-side engagement data to learn an encoder (a neural network) that translates listing features into embedding space where similar listings are close to each other. As a model, we leveraged the state-of-the-art two-tower neural network architecture. The first tower, denoted as the `Signal Tower`, represents the listings with which a guest interacted before a booking occurs. The second tower, denoted as the `Label Tower`, represents the booking. When training such a model for the domain of listings and guests, we faced the following challenges:

- **Negative examples overfitting:** Our guest engagement data naturally has only positive examples (i.e., booked listings), and we need to find ways to generate negative examples. The typical approach of using random negatives often leads to overfitting on the location features of listings, as the model can easily guess the correctly booked listing by using location proximity. We investigated various strategies for selecting hard negatives, which force the model to learn from features other than location.
- **Bias towards popular listings:** Typically, guests engage most with popular listings, thus skewing the training data towards the features of these popular listings. This leads to reduced performance when applied uniformly to all listings in production. To account for this bias in the training data, we explored adjusting the weight of each training data example based on listing popularity, thus giving more importance to long-tail cases.
- **Scaling up the training data size:** The main bottleneck in training our embedding model is the large I/O cost associated with reading the training data. In our case, each training example consists of more than 10 listings (viewed/booked listings), with each listing having over 100 features. This results in a significant increase in the size of our training data. Importantly, most of the features, such as location and number of bedrooms, are fairly static. Thus, we explored the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
TSMO '24, August 26, 2024, Barcelona, Spain
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

opportunity to cache them in order to reduce training time and cost.

In our domain, most applications don't directly use embeddings but are based on similar listing sets. Thus, there is a need for an additional step that involves finding the most similar listings (embeddings) for a given target listing embedding. The brute force search is computationally prohibitive, so we explored approximate k-nearest neighbor search solutions (also denoted as ANN) for embedding retrieval, such as Meta's FAISS [11] and Tensorflow's ScaNN [12].

Due to the multistep process of producing similar listings, we need to holistically evaluate the performance at every step and end-to-end. This paper discusses our approach to evaluation, where we measure the effectiveness of the embedding model at the level of embeddings, i.e., its ability to correctly predict guest preferences for similar listings. Additionally, we present our evaluation of the qualities of similar listings, which are important for potential applications (such as sensitivity of similar listings to feature changes and key feature similarity like distance).

Finally, we discuss the key applications of listing embeddings in the context of Airbnb's two-sided marketplace. First, we describe the applications of listing embedding and similar listing at the host level of the marketplace, such as Host Calendar and List-Your-Space products. Second, we discuss potential applications on the guest side of the marketplace, such as recommending similar listings to guests or offering them substitute listings in the event of canceled reservations.

The paper is structured as follows. Section 2 presents the overall architecture of the listing embeddings and the generation of similar listings. Section 3 provides the details of the training data, architecture, and scaling of the embedding model. Section 4 discusses the generation of similar listings. Section 5 presents our holistic evaluation of the listing embedding model, the properties of similar listings, and the sensitivity and importance of features. Section 6 discusses the product applications, and finally, Section 7 concludes the paper.

2 SYSTEM OVERVIEW

This section describes the overall architecture of the embedding model and the generation of similar sets. In order to support products with similar listing sets, we need to have components for training data generation, model training, and online serving, which are depicted in Figure 1.

During the training data generation, we collect historical guest engagement data, which consists of guest view/booking sessions. We augment this data with listing features, such as the number of bedrooms, capacity, rating, and produce training example files. The current implementation is based on Apache Spark [20]. The training data job supports three modes of data generation: training, validation, and inference data. It lands the output data in both the Apache Hive format (for analytics, debugging) and the TensorFlow record format (for training).

In order to run training, we implemented the model code in Python and TensorFlow [8] (see Section 3 for more details). We ran distributed training on Airbnb's internal ML training infra that integrated with Horovod [15]. The output of a job is a serialized model that is deployed for serving (to TensorFlow serving or other serving platforms)

To support a broad range of applications, we facilitate both online and offline serving of listing embeddings and similar listing sets. For offline similar listing generation, we run daily Airflow jobs that generate Airbnb listing embeddings and retrieve the similar listings using ANN. These embedding/similar sets are available for analytics and serving when freshness is not critical. We also support online serving (not shown in Figure 1) is based on fetching listing features online and invoking the embedding model. We also facilitate ANN search and any post-processing logic (i.e., enforcing geo-distance boundaries, availability checking, and so on.).

3 MODEL

This section presents the architecture of the embedding model as well as the details of the training data. We also discuss the specific changes we made to tune the model characteristics for the domain of Airbnb listings and their application on marketplace product surfaces.

3.1 Training Data and Features

The embedding model is learned from our guest session data, where each session consists of a sequence of viewed listings that result in one booked listing. Each training example can be viewed as an instance of a guest demand signal expressed through a view/booking session. Specifically, each training example is obtained in the following way:

- Fetch all bookings of the listings that have had at least `N` bookings in the past.
- For each booking, collect all prior long views (greater than `T` seconds) within a window of `M` days prior to the booking.
- Remove duplicates for each booking's views.

Note: `M`, `N`, `T` are parameters that may be different for different geo regions.

Although it is possible to learn listing embeddings just using the listing IDs of training examples (similarly to [10]), we also augmented each listing with its features. The primary reason for this is to handle the cold start problem of newly created listings or listings without a significant amount of engagements, where there are not enough guest view sessions in the training data, thus necessitating reliance on the listing features.

Each signal and label listing is augmented with the following features:

- **Listing properties:** number of bathrooms, bedrooms, beds, its capacity, listing tier and type, and amenities.
- **Listing location:** country, city, state, latitude/longitude, and zip code.
- **Listing ratings and reviews:** accuracy, cleanliness, check-in process, location, value, number of five-star reviews, and total number of reviews.
- **Listing images derived features:** image quality scores, cover image amenities, and cover image room type.
- **Other:** whether the listing offers instant booking, the cancellation policy, and the number of days the listing has been posted.

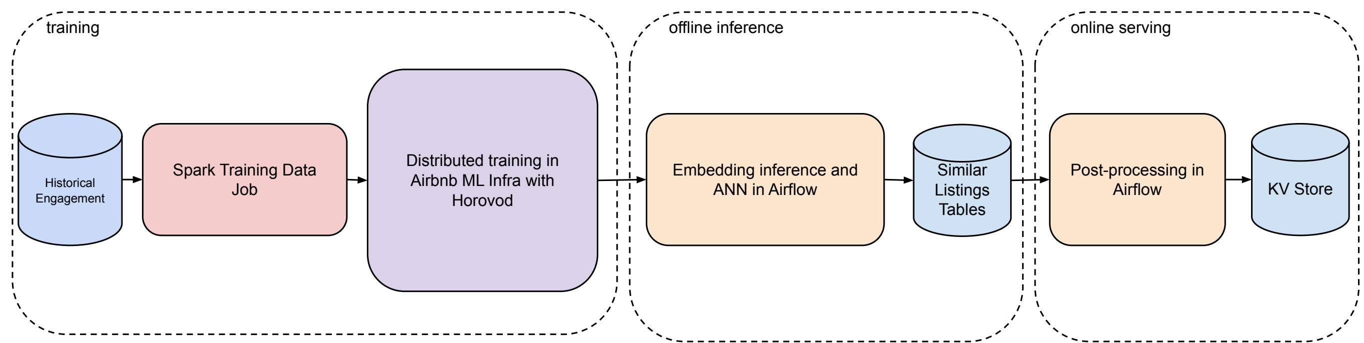

Figure 1: System Overview

Each input feature is processed either as dense or categorical. Each categorical feature is encoded as an embedding. Typically, the size of the embedding is determined as follows [1]:

$$embedding_dimension = number_of_categories^{0.25} \quad (1)$$

In most cases, the number of categorical values is large, and we only use those categories that have sufficient coverage (more than 1K training examples with a given value). To find the optimal cut-off threshold, we conducted several experiments to evaluate the trade-offs between model performance and the sizes of embedding vocabularies. Reducing the size of categorical feature embeddings using the techniques like Binary Code based Hash Embeddings [17] is considered in the future work. For each dense feature, we apply a normalization process where we shift and scale inputs into a distribution centered around 0 with a standard deviation of 1.

3.2 Model Architecture

Based on the training data and listing features (refer to Section 3.1 for more details), we designed and implemented a neural network model for embedding learning. For each session, the model encodes a sequence of viewed listings into a signal embedding and its corresponding booked listing into a label embedding (of the same dimension as the input). It then forces these two embeddings to be similar for positive examples and dissimilar for negative ones.

We adopt a two-tower neural network model (Figure 2), which is widely used by researchers and machine learning practitioners alike [10, 16]. There are different variations of this architecture, where the underlying data can be modeled as user/item or item/item interactions. In our case, we model the problem as listing/listing (each tower encodes a listing or its sequence). The main reason for this is that, for most targeted applications, we focus on the properties of listings as a determining factor of listing similarity rather than the properties of users (guests) who view or book those listings.

Overall, we consider the following advantages provided by a neural network model:

- **Rich features:** A neural network can ingest diverse signals such as dense and categorical features, image or text embeddings in a unified manner.
- **Scale:** network training can be scaled to billions of training examples with the assistance of distributed learning and training optimization techniques, such as in-model caching.
- **Architecture tuning:** The design and training of a neural network can be easily adjusted to meet varying application requirements. This can be achieved through layer structure modifications, loss function customization, training data structure alterations, and so forth.
- **Performance:** Typically, neural networks are capable of achieving state-of-the-art effectiveness.

In the following sections, we will provide a more detailed overview of how we utilize the aforementioned advantages in the realm of Airbnb's two-sided marketplace.

In our implementation, the `Signal Tower` and `Label Tower` have identical layer structures, as depicted in Figure 2. The only difference is that the signal tower averages signal listing embeddings before a sequence of fully connected layers is applied. During the prototyping, we didn't see a noticeable gain when using sequence-based layers like multi-headed self-attention instead of averaging, and thus we left further experimentation in that direction for future work. Each fully connected layer uses a `Tanh` activation function, and the dot product of the final layers of the two towers is taken as the distance between signal and label embeddings. During experimentation, we found that the best performance is achieved when the towers do not share parameters. During the training, we use the in-batch negative sampling [19] with the categorical cross-entropy loss where negatives are sampled from within the same batch examples. This loss function choice allows us to scale up training to billions of training examples since the negative example embeddings are already computed for other positive examples in the batch.

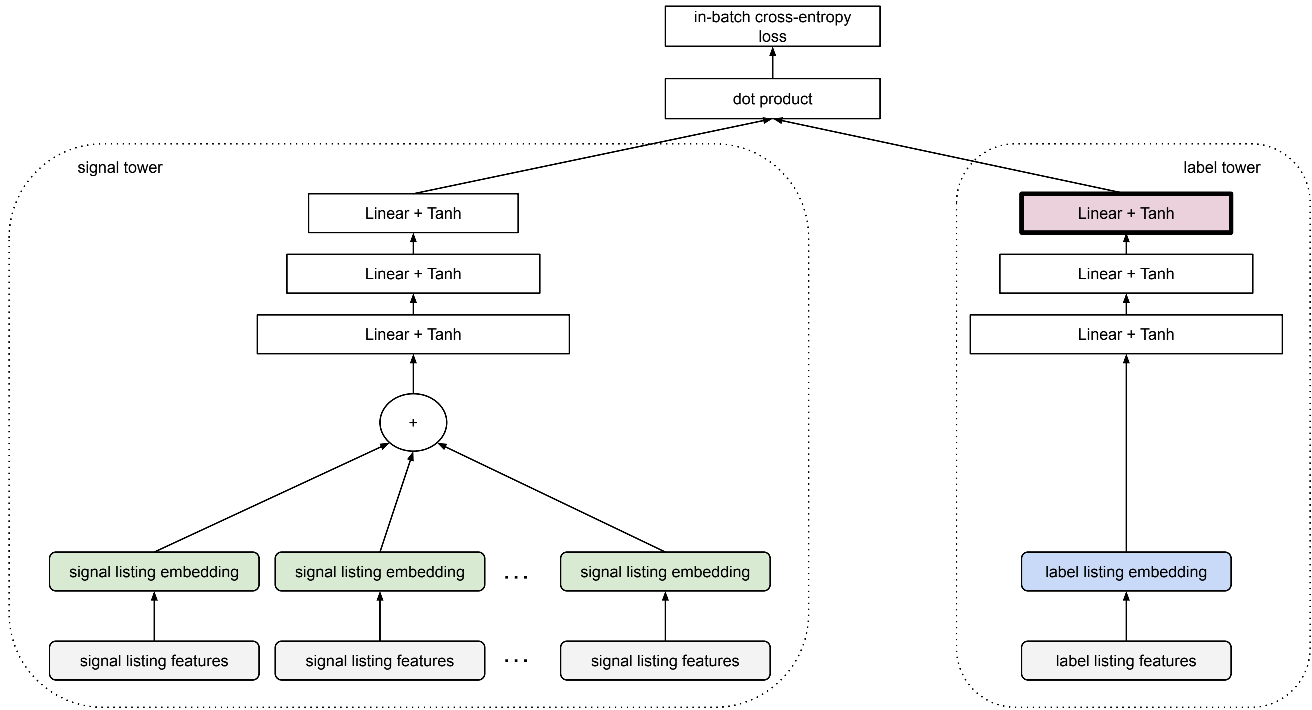

Figure 2: Listing Embedding Model Architecture

3.3 Negative Example Sampling

Negative example sampling plays a crucial role in model performance and scaling, particularly in the domain of Airbnb listings. However, we encountered several challenges that needed to be properly addressed. In the following sections, we discuss the issue of model overfitting on location features and the sampling bias problem arising from the domination of popular listings in the training data.

Typically, training data is randomly sampled, which means that in-batch negative listings are uniformly sampled from the entire population of listings. In the domain of Airbnb listings, we have listings that cover the entire globe. As a result, a random negative listing is very likely to be from another part of the world. The model can easily overfit and rely solely on the location features to correctly identify positive examples from all negative listings. To overcome this, we considered the following training data sorting orders:

- **Random Order (`random`):** In this approach, training examples are randomized, meaning each in-batch negative can come from any other possible guest session. Importantly, it is highly probable that negatives will originate from vastly different locations, which impacts the model's ability to learn from non-location features.
- **Location-Aware Sorting Order (`geo`):** In this method, training examples are randomized within their respective locations (e.g., within a country or province). In this case, the negatives are random; however, they are guaranteed to be from the same location as a positive example. This approach encourages the model to focus more on non-location features and less on location-based ones.
- **Mixed (`mixed`):** This approach involves training data consisting of two copies: one is randomly sorted, and the other is geographically sorted. This method encourages the model to learn both geographical and non-geographical features

In our experimental evaluation, we found that the best performance is achieved with mixed training data sorting. Please refer to Section 5 for more details.

3.4 Sampling Bias Correction

The generation of training data, as well as negative example sampling, is based on guest engagement data, which is heavily skewed towards popular listings. Consequently, the most popular listings would dominate both positive and negative examples, and the model's performance would be biased towards the features of these listings. This is a well-known problem of sampling bias [10]. We address this problem by adjusting the importance of items according to their probability in the population. Specifically, to correct this bias, we employed the logQ sampling distribution correction [9] which is also adjusted to the batch size using multinomial distribution. The formula for the correction is as follows:

$$Q = 1 - (1 - w)^{2048} \tag{2}$$

where w is the weight of the listing in the training data, and 2048 is the batch size. The above correction reduces the bias to popular listings and as a result improves the embeddings metrics (see Section 5 for more details).

3.5 Distributed Learning and Caching

In this section, we discuss the challenges and solutions associated with embedding model training on large amounts of user engagement data examples.

The in-batch negative sampling unlocks the potential for scaling up training, as the encoding of negative examples is highly efficient. In our training, we utilized a batch size of 2048 with 512 hard negatives (hard negatives are negatives with the largest logits, which are retained when computing cross-entropy loss). To manage this vast amount of training data, we employed several scaling techniques, such as distributed training and embedding caching, which are discussed below.

The model source code is implemented in the TensorFlow [8] framework and the TensorFlow Recommender [7] library. A single machine training of a production model takes more than a week for one epoch using a 24 cpu, 40gb ram with A10G Nvidia GPU machine. In order to accelerate training, we utilized the distributed training framework Horovod [15] which allowed us to efficiently scale the training to multiple machines. We were able to parallelize across 16 machines (each with 24 cpu, 40gb ram and A10G GPU) and achieve a one epoch training time under 24 hours. The total number of epochs during model training is typically in the range 13-20.

Although the distributed training allowed us to significantly reduce the training time, we also explored optimization techniques that were based on the model architecture and the properties of our data. Upon profiling the model's performance, we identified that the most expensive operation was the IO cost, given that a typical training example consists of more than 10 listings (viewed/booked listings of a guest session), each with their own features. In our domain, most of the listing features remain largely unchanged throughout the training timespan of one year (e.g. number of bedrooms, address, and so on), and the total number of training listings is in the order of millions. Based on these observations, we implemented in-model caching of static listing features, as depicted in Figure 3.

The feature cache is pre-generated before training by analyzing the frequencies of feature changes and saving non-changed features for each listing. Then, the cache is loaded into GPU memory in the form of an embedding table (`Variable` in Tensorflow) where each listing has their static float and int features stored. During training, the training example reader enriches the data with static features from the cache whereas non-static features (such as days listed on Airbnb) are still stored as part of the training example. Then, both static and non-static features are plugged at the appropriate layers on the tower architecture: for most int features we apply embedding lookup layers whereas float features are directly concatenated to the listing embedding layer. From the implementation standpoint, we completely decoupled the cache lookup logic from the tower's structure thus easily supporting cache/non cache modes (for the inference we don't need to use caching).

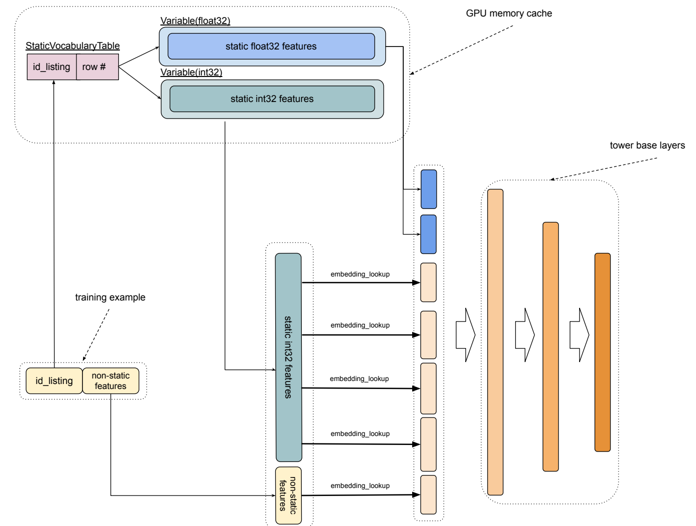

Figure 3: In-model feature caching

In order to have the caching in GPU we need to make sure that cache can fit in GPU memory. In our domain, we utilized NVIDIA A10G GPUs with 24GB memory, which was sufficient for caching all listing static features. In the future, we plan to rely on caching in order to ingest embedding based features such as text, image or location embeddings. As a result of the caching optimization, we observed a reduction of more than 50% in both training time and computation cost without any change in the efficiency.

4 SIMILAR LISTINGS

Most applications of the listing embedding model are based on generating most similar listing sets (also known as compsets in the industry). This section presents our approach to generating similar listings and the challenges we solved along the way.

After training the embedding model, we use the label tower (see Figure 2) as a listing encoder and its last layer as a representation of the listing embedding. Based on the training data formulation and model architecture, we expect similar listings to be close in the embedding space (we found that the optimal size of this vector is 32). As a result of the inference stage, we obtain an embedding vector for each active Airbnb listing. We use the dot distance to measure the magnitude of two listing similarity based on their embeddings.

In most downstream applications, the goal is to find a set (typically, this is 50 or 100) of the most similar listings. Given the embedding of all listings, a naive approach is to do the brute force search of the most similar embedding vectors. That process is an expensive process in our domain, as we need to inspect millions of candidates. To accelerate this, we employed two ideas: first, we leverage the approximate k nearest neighbors search (ANN) libraries such FAISS [11] or ScaNN [12] and second, we reduce the pool of candidates to some predefined location (e.g. we don't need to look for most similar listings in other countries or even cities). As a result of the above optimization, we were able to significantly optimize the generation of the most similar listing sets.

We also support additional processing of those similar listings sets (reranking, filtering) which are application specific. For example, we may need more strict requirements on similar listings distance to the target listing or the difference in the number of rooms/capacity. All of that additional logic is implemented by downstream applications and could be easily tuned.

5 EVALUATION

This section presents our approach to the evaluation of embedding models and similar listing sets.

In order to successfully use the listing embedding model in applications we need to evaluate model performance not only from its ability to minimize training data objective function (i.e. accurately predict label listings) but also from the quality of produced similar listings. We define the following evaluation datasets (Table 1) which are used across different steps of our pipeline (see Section 2 for the overall pipeline overview):

Eval dataset	Size	Sorting order	Filter
Training	80%	geo, random, mixed	listings with N bookings
Validation	20%	random	listings with N bookings
Vocabulary	100%	random	active
Similar Listings	100%	random	active

Table 1: Eval datasets

We split one year of listing bookings between the training dataset (80%) and the validation dataset (20%) where we ensure that validation bookings are chronologically after training ones in order to avoid information leakage. The training dataset has 3 versions: randomly ordered, geographically ordered, and mixed which is a combination of the previous two. Note, to address the cold-start problem of missing features (like no ratings/reviews) we apply a filter which uses only listings with at least `N` bookings. The vocabulary dataset consists of a snapshot of all listings present in both the training and validation datasets. We use the vocabulary dataset to generate (inference) the embeddings of all listings and use those embeddings for the similar listing generation. Lastly, we refer to the Similar Listings dataset as a vocabulary set of listings, which have the top K similar listings retrieved from the entire vocabulary (using ANN search).

5.1 Recall

Recall-based metrics focus on measuring the effectiveness of the embedding model in predicting label listings, and they assist in answering the question of how well the model can learn guest preferences from their sessions. The key metric is `Recall@K`, which computes the proportion of training examples where the model was able to correctly identify the label listing within the top `K` results.

During training, we measure in-batch `Recall@K`, which is solely based on listings within the current batch. Given that our batch size is 2048, this metric can be viewed as a sampled `Recall@K`, where the entire population consists only of the same batch listings. After each epoch, we also measure the in-batch vocabulary `Recall@K`. The in-batch metrics are easy to compute; however, they are merely approximations of the entire population. To compute non approximated recall on all listings is computationally prohibitive and is not very aligned to potential product uses (in most cases, we expect similar listings to be within certain pre-defined geographical boundaries). To address those issues, we selected the top 5 locations and computed vocabulary recalls using only respective location vocabularies, i.e. for Los Angeles we just use Los Angeles listings and filter out all others.

We evaluated the performance of `geo`, `random`, and `mixed` sorted training data and found that the best performance is achieved with the `mixed` ordering with the respective `Val Recall@50` gains of +3.77% and +14.81% over only `random` or `geo` sorted ones, respectively (see Table 2). Additionally, the `mixed` model demonstrated high neighborhood and capacity similarity metrics (Section 5.3). Our main takeaway is that we need to strike a balance between the geo and non geo features by carefully choosing the negative examples and our `mixed` sorting order was able to outperform the only `random`/`geo` based negatives.

model	`Val Recall@50`	neighborhood sim	capacity sim
`geo`	81.80%	11.22%	43.07%
`random`	92.84%	61.38%	39.17%
`mixed`	**96.61**%	**87.16**%	**45.22**%

Table 2: Negative sampling experiment results

The bias correction technique (Section 3.4) was applied to the `mixed` model and showed a similar `Val Recall@50` of 96.68%, however, it significantly improved the top5 location `Recall@50` from 8.13% to 11.56%. As a result of correcting the sampling bias, the model was able to better generalize to long-tail listings.

5.2 Feature sensitivity and importance

In order to successfully utilize listing embeddings in applications, it is useful to understand the feature importance as well as feature sensitivity.

When a host updates a listing, their similar listings also need to be updated. However, this process can be costly in both computational (i.e., making online inferences to regenerate similar listings) and engineering (implementing the online inference infrastructure) aspects. Therefore, it is useful to understand which features have the most impact on the embedding and should trigger similar listings regeneration, and which features have a minor impact and could lead to a similar listings set update at a later stage. To this end, we performed a sensitivity analysis where we randomly perturbed features and measured the impact on their embeddings. As a result, we were able to estimate which features are most likely to alter listing embeddings and their respective similar listings. Although the feature sensitivity is conditioned by listings itself, we found that those amenities are typically the most impactful ones (Table 3):

Rank	Amenity
1	Pool
2	Free parking
3	Jacuzzi
4	Wireless Internet
5	Allows Pets

Table 3: Most impactful amenities

These sets of high sensitivity amenities (and other features) are then used by the product to decide on similar listings refresh schedules.

In the current model, we use more than 100 listing features, and we aim to understand the importance of these features. Finding feature importance in complex neural network architectures is not straightforward, and we explored different approaches for this. Specifically, we used normalized mutual information between signal and label feature value distributions to measure feature importance ([4]). This approach allows us to compare different types of features (e.g., dense vs. categorical) uniformly. However, it does not take feature interactions into account.

Another approach is based on the idea of eliminating a feature signal by permuting its values during the inference process (referred to as permutation feature importance [4]). This approach leverages the labeled data and provides a clear, well-defined method for measuring feature importance. However, it does not consider feature dependencies. The above two approaches identified most location features as the most important, as well as certain amenities like AC, Heating, and Smoke Detector.

5.3 Similar listings metrics

Finally, most applications are based on a set of similar listings that are shown for a given target listing. Therefore, it is important to evaluate the properties of similar listings and ensure they meet product expectations. For example, hosts would expect their listings to be in the proximity of their own listing (e.g., within a certain radius) and their capacities (number of bedrooms) to be relatively close (e.g., +-1). To address this, we also measure the properties of similar listing sets with respect to their target listings:

- **Geographical similarity (`neighborhood sim`):** Average distance between the target listing and its similar ones or the proportion of listings that are from the same zipcode, city, and so on.
- **Listing similarity (`capacity sim`):** How close the number of bedrooms, beds, capacity of target listings and its similar ones. How far is the median price of similar listings to the price of the target one?

In our evaluation, we found that improvements in `Recall@K` generally translate to improvements in the similarity metrics (Table 2). However, we discovered that there is an inherent limit when further reduction of recall leads to an increase in some similarity metrics.

6 APPLICATIONS

This section presents the existing and future product applications of the similar listings model.

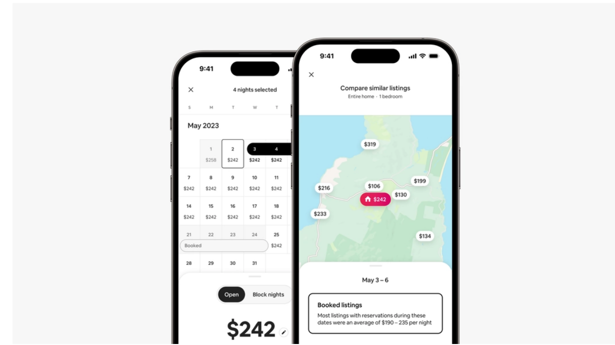

Figure 4: Similar listings on Host Calendar Product Surface

6.1 Host Calendar

Airbnb hosts manage their listings by posting their properties on Airbnb, adjusting their calendar for availability, and fine-tuning their prices and settings. The Host Calendar is a tool that enables them to perform most of these tasks on their availability calendar (see Figure 4).

Setting an appropriate price is a challenging task [18] for hosts. Assisting hosts in comparing their price with the average prices of similar listings has proven to be beneficial and has become a part of the Host Calendar product [6]. This product is based on the model discussed in this paper, and it utilizes the properties of similar listings discussed in Section 5. On one hand, we aim to display the most similar listings according to guest preferences. On the other hand, we also strive to ensure that these listings meet the host's key expectations such as geographical proximity and price categories. By tuning our model and post-processing logic, we aim to find the optimal balance between these two aspects. As a result of the rollout of this model to production, we observed positive feedback from hosts who found the similar listings feature helpful in setting their prices.

Another related application of similar listings used by Airbnb hosts is the List-Your-Space screen, where new hosts set their price for the first time. In this scenario, we continue to use our production model. However, we impute any missing features (a new listing may not have all information filled out yet) based on our understanding of the location and nearest listings.

6.2 Similar Listings for Guests

The previous section discussed existing product interfaces where hosts utilize similar listings. In this section, we will explore potential applications of similar listings for guests.

When guests are in search of a place to stay, a collection of similar listings can be shown along with the listing they are currently viewing. The existing model is based on the listing-to-listing two-tower architecture, which doesn't take into account guest-specific features such as their origin or past bookings. Therefore, the presented similar listings will not be personalized and will solely focus on the listing similarity perceived by all Airbnb guests. A possible extension could involve switching to a guest-listing two-tower architecture. In this case, the similar listing recommendation could be personalized for guests.

There are situations when hosts need to change their plans and cancel their existing reservations. In these instances, guests need to find a substitute listing that is comparable to the canceled one. The existing embedding model (or its extension to a guest/listing architecture) might be applicable as well.

7 CONCLUSION

This paper presents a study on the application and learning of listing embeddings in Airbnb's two-sided marketplace. We present our approach to building a system that learns listing embeddings based on guest engagement data and supports the generation of similar listing sets. We delve deeper into the technical challenges we faced during training and discuss our solutions and lessons learned from evaluating the model at all stages. Finally, we discuss the existing and future products in which listing embedding models and similar listing sets are used at Airbnb.

8 ACKNOWLEDGMENTS

Special thanks go to Win Aung, Shane Jarvie, Monu Kala, Lifan Yang, Lu Zhang who worked on the implementation of the data pipeline, serving infrastructure and application related systems.

REFERENCES

[1] 2017. Introducing TensorFlow Feature Columns. (2017). https://developers.googleblog.com/en/introducing-tensorflow-feature-columns/
[2] 2021. Using Triplet Loss and Siamese Neural Networks to Train Catalog Item Embeddings. (2021). https://doordash.engineering/2021/09/08/using-twin-neural-networks-to-train-catalog-item-embeddings/
[3] 2023. Innovative Recommendation Applications Using Two Tower Embeddings at Uber | Uber Blog. (2023). https://www.uber.com/blog/innovative-recommendation-applications-using-two-tower-embeddings/
[4] 2023. Interpretable Machine Learning. (2023). https://christophm.github.io/interpretable-ml-book/index.html
[5] 2023. lyft2vec - Embeddings at Lyft. (2023). https://eng.lyft.com/lyft2vec-embeddings-at-lyft-d4231a76d219
[6] 2023. New pricing tool compares similar listings nearby - Resource Center. (2023). https://www.airbnb.com/resources/hosting-homes/a/new-pricing-tool-compares-similar-listings-nearby-583
[7] 2024. tensorflow/recommenders: TensorFlow Recommenders is a library for building recommender system models using TensorFlow. (2024). https://github.com/tensorflow/recommenders
[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). arXiv:1603.04467 http://arxiv.org/abs/1603.04467
[9] Yoshua Bengio and Jean-Sébastien Senecal. 2008. Adaptive Importance Sampling to Accelerate Training of a Neural Probabilistic Language Model. *IEEE Trans. Neural Networks* 19, 4 (2008), 713–722. https://doi.org/10.1109/TNN.2007.912312
[10] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, Shilad Sen, Werner Geyer, Jill Freyne, and Pablo Castells (Eds.). ACM, 191–198. https://doi.org/10.1145/2959100.2959190
[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. *CoRR* abs/2401.08281 (2024). https://doi.org/10.48550/ARXIV.2401.08281 arXiv:2401.08281
[12] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3887–3896. http://proceedings.mlr.press/v119/guo20h.html
[13] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781
[14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1532–1543. https://doi.org/10.3115/V1/D14-1162
[15] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799
[16] Shoujin Wang, Longbing Cao, Yan Wang, Quan Z. Sheng, Mehmet A. Orgun, and Defu Lian. 2022. A Survey on Session-based Recommender Systems. *ACM Comput. Surv.* 54, 7 (2022), 154:1–154:38. https://doi.org/10.1145/3465401
[17] Bencheng Yan, Pengjie Wang, Jinquan Liu, Wei Lin, Kuang-Chih Lee, Jian Xu, and Bo Zheng. 2021. Binary Code based Hash Embedding for Web-scale Applications. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (Eds.). ACM, 3563–3567. https://doi.org/10.1145/3459637.3482065
[18] Peng Ye, Julian Qian, Jieying Chen, Chen-Hung Wu, Yitong Zhou, Spencer De Mars, Frank Yang, and Li Zhang. 2018. Customized Regression Model for Airbnb Dynamic Pricing. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 932–940. https://doi.org/10.1145/3219819.3219830
[19] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed H. Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019*, Toine Bogers, Alan Said, Peter Brusilovsky, and Domonkos Tikk (Eds.). ACM, 269–277. https://doi.org/10.1145/3298689.3346996
[20] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664