# Graph Neural Network approaches for Airbnb listings price prediction

Pietro Lodi Rizzini

## 1 Introduction

The goal of this project is to study new approaches for Airbnb prices prediction. Traditional approaches focus solely on the listings features without much stress on the geo-spatial context of the listings' environment. In this project, we want to explore how adding neighbourhood information influences the quality of the prediction. Intuitively, the price of a listing can highly depend, other than the features of the property itself, on the location and the amenities in the surroundings.

The proposed approach wants to leverage the recent techniques from the deep learning and graph theory, Graph Neural Networks [1] to accurately predict the price of the listing. We want to represent the data by means of a graph structure, where nodes represent listings and other points of interest, while edges represent the relationships between them.

We are going to explore two methods to solve the problem. In the first one, we are going to directly perform Node regression on a graph, while in the second we are going to produce node embeddings for the listings, on top of which traditional regression algorithms are trained.

## 2 Methods/Case Study

### 2.1 Method 1

The first approach is divided in: data collection and cleaning, graph dataset construction, GNN model construction and comparison with baselines.

#### 2.1.1 Data collection

We build our case study on the Chicago area. Listings data was collected from http://insideairbnb.com, a service that provides scraped data from the Airbnb website. Due to the lack of many values, some features were dropped. The cleaned dataset has the following features:

- **property_type:** The type of property, such as house, apartment, or condominium.
- **description:** A brief textual overview describing the key features or characteristics of the property.
- **room_type:** Specifies the type of room available, like entire home/apartment, private room, or shared room.

- **accommodates:** The maximum number of guests the property can comfortably accommodate.
- **bedrooms:** The number of bedrooms available in the property.
- **number_of_reviews:** The total count of reviews received by the property.
- **review_scores_rating:** The overall rating of the property based on guest reviews.
- **calculated_host_listings_count:** The count of listings managed by the host.
- **availability_30:** The number of days the property is available for booking within the next 30 days.
- **minimum_nights:** The minimum number of nights required for booking.
- **latitude:** The geographical latitude of the property.
- **longitude:** The geographical longitude of the property.
- **price:** The cost per night for renting the property, the variable to be predicted.

Given the distribution of the price skewed towards low values with some outliers, we apply the log function to make it more compact. We then collect neighbourhood data leveraging Yelp APIs and building a custom dataset. We perform queries in the Yelp APIs using the following keywords: "restaurant", "shop", "bar", "museum", and setting "Chicago" as location. We obtain a dataset with the following features:

- **type:** The general type or category of the POI, based on the query (restaurant, shop, bar, museum)
- **rating:** The overall rating or score assigned to the POI by users.
- **review_count:** The total count of reviews received by the POI.
- **latitude:** The geographical latitude of the POI.
- **longitude:** The geographical longitude of the POI.
- **category:** The specific category of the entry as provided by Yelp.
- **price range:** The price category.

### 2.1.2 Graph Construction

For the construction of the graph, we consider an Heterogeneous Graph structure with the node types **Listing** and **POI**. The node attributed of listings and POIs are the previously defined ones. We use Pytorch Geometric's HeteroGraph type to define this structure. For each Listing, we connect it to all the POIs whose distance is lower than a threshold *distance_thresh*, which is an hyperparameter. If a listing doesn't have any POI within the threshold, we still connect it to the nearest one. To compute the distances, we use the haversine formula, given that we have latitude and longitude coordinates for each pair listing-POI. As edge weights between the nodes, we use the inverse of their distance.

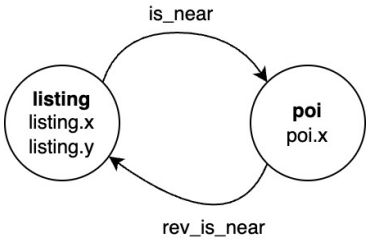


Figure 1: Heterogeneous graph structure

#### 2.1.3 Node regression

In this step, we train a Graph Convolutional Network (GCN) model to perfrom node level predictions on the listing node.

The model is able to capture, other than the listings characteristics, the features of the neighbourhood, given the edges that were defined in the previous step.

Figure 2 shows the architecture of the model, which consists of two GraphConv [2] layers with ReLU activation functions. Since we are working on an heterogeneous graph, we need to duplicate the message functions to work on each edge type individually. GraphConv are aggregated by the "sum" operator through Pytorch geometric's HeteroConv convolution, a wrapper for computing graph convolution on heterogeneous graphs. To compute the output variable, we apply a Linear layer over the listing.

### 2.2 Method 2

The main idea behind the second method we explored, consists in computing the node embeddings in a graph where nodes represent listings, and edges represent some sort of relationships between the listings. The node embeddings are then fed into a Linear Regression model to perform the prediction. Here we work on a Homogeneous graph structure, in which we have just one type of node: listing. We build the edges in the following way: we connect only listings:

1. whose relative distance is lower than a threshold *distance_ thresh*
2. whose descriptions are similar. To identify pairs of listings whose descriptions are similar, we compute the cosine similarity of their text embedding, computed by means of the Doc2Vec model. If, for every pair of listings, the cosine similarity exceeds a threshold *cos_ sim_ thresh*, we consider them similar.

As before, we use the inverse of the distances as edge weights.

To compute the node embeddings, we use the GraphSAGE [3] algorithm in an unsupervised fashion. After every epoch of training, we train a Linear Regression model and analyze its perfromance over unseen data. We continue the GraphSAGE training until the test perfromances of the Linear Regression are becoming better.

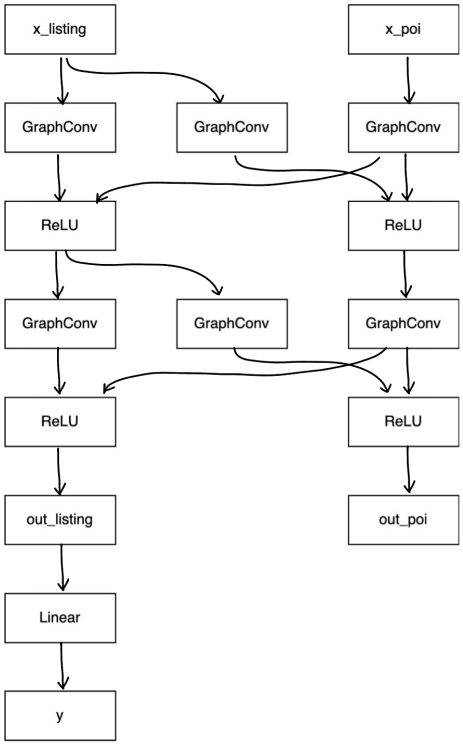


Figure 2: GNN model architecture

## 3 Results and Discussion

For the implementation and evaluation of the methods, Pytorch together with Pytorch Geometric [4] libraries were used. The data was represented with HeteroData and Data objects and lists for node edges and weights were created iterating over pairs of entities and filtering as mentioned before. Table 1 summarizes the performance of Method 1 for different values of distance_thresh. As we can see, lower values bring a better result, but the overall performance is not the best. This may be due to the relatively small amount of data available (about 8k listings).

Table 1: Test performance of Method 1 over different distance_thresh values (MSE)

| distance_thresh | MSE |
|---|---|
| 400 meters | 1.122 |
| 200 meters | 0.973 |
| 50 meters | 0.812 |

For method 2, we performed multiple trainings with different combinations of the thresholds, in order to study the contribution of each parameter to the final result. The results are summarized in Table 2. As we can see, the best performance is obtained with higher cosine

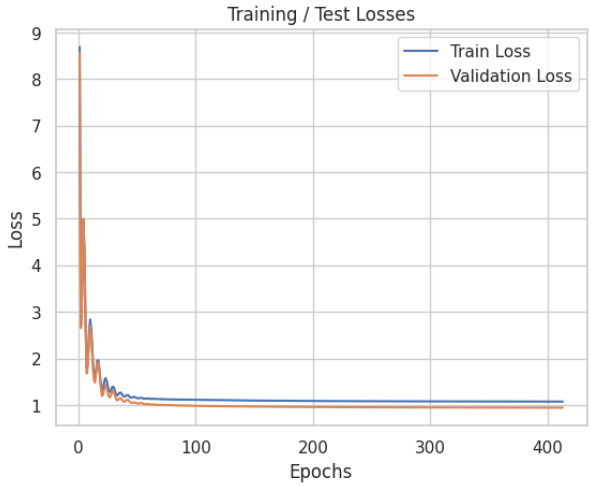


Figure 3: Training/Validation Loss for Method 1

similarity thresholds, if we keep the distance threshold fixed. The same happens for the distance threshold when the cosine similarity is constant. This is reasonable, since higher cosine similarity thresholds allow to build a more informative structure, while lower values would just end up in connecting listings that are near to each other, even if they are not that similar.

Table 2: Test performance of Method 2 over different configurations (MSE)

| distance_thresh - cos_sim_thresh | 0.95 | 0.6 | 0.3 |
|---|---|---|---|
| 700 meters | 0.136 | 0.254 | 0.264 |
| 400 meters | 0.143 | 0.193 | 0.237 |
| 100 meters | 0.167 | 0.173 | 0.188 |

# 4 Conclusion

In conclusion, this project has been an insightful exploration into the realm of graph-based data representation and learning. The application of GNNs has demonstrated promising results in various domains, ranging from social network analysis to molecular chemistry. The ability of GNNs to capture intricate relationships within graph-structured data has opened new avenues for solving complex problems.

However, this journey was not without its challenges. One significant hurdle encountered during the project was the difficulty in finding comprehensive and user-friendly documentation for the GNN libraries used. While GNNs are a powerful tool, the scarcity of well-documented resources made it challenging to grasp the intricacies of implementation.

Despite these challenges, the project succeeded in leveraging GNNs to achieve its objectives,

showcasing the potential of this technology in real-world applications.

## Reference

[1] M. Gori, G. Monfardini and F. Scarselli, "A new model for learning in graph domains," Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., Montreal, QC, Canada, 2005, pp. 729-734 vol. 2, doi: 10.1109/IJCNN.2005.1555942.

[2] Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., Grohe, M. (2018). Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks (Version 5). arXiv. http://doi.org/10.48550/ARXIV.1810.02244

[3] Hamilton, W. L., Ying, R., Leskovec, J. (2017). Inductive Representation Learning on Large Graphs (Version 4). arXiv. http://doi.org/10.48550/ARXIV.1706.02216

[4] https://pytorch-geometric.readthedocs.io/en/latest/