Travel Genius
- An all-in-one accomodation portal

Uditya Uday Laad
ulaad@uwaterloo.ca (20986041)
University of Waterloo, Waterloo, ON, CA

Niravkumar Vinubhai Talaviya
nvtalavi@uwaterloo.ca (20896128)
University of Waterloo, Waterloo, ON, CA

***Abstract*—Current decade has taken the concept of globalization to a larger scale, which has also lead to a boom of tourism industry in multiple countries around the world. And with technology at the face of enabling smooth travel planning and arrangement, the demand for centralized systems to facilitate reservation and management of temporary stay accommodations has grown exponentially in the last 2 decades. Customers (specially travellers) prefer an all-in-one interface where they can find ideal accommodations, book and manage reservations, make payments and so on. As for hosts, they prefer an interface to list and manage their properties. Airbnb, of late, has been a major player in the hospitality industry and has made available a wealth of data from their system that spans many regions. This project makes use of these large Airbnb datasets, translates them to a non-trivial relational form and provides a centralized client application to facilitate stay reservations and listings management for customers and hosts respectively.**

***Index Terms*—User, Host, Customer, Listing, Listing Calendar, Reserved Calendar, Reviews, Payment**

I. Introduction and related work

Recent advancements in technology and the internet have impacted virtually every industry, and the hospitality sector has been no exception to it. The instant growth of hospitality applications such as Airbnb, Trivago, MakeMyTrip, etc serves as good examples of how technology has made travel planning far more convenient than it used to be. It's not just about convenience; such systems also have processes in place, that allow customers to decide on the legibility of hosts, properties, or vendors. They also provide transparency with respect to price, location, host information, etc. All of this enables a customer to make reservations with greater confidence and security.

As for hosts, they can add and manage their listings, payments, etc far more conveniently with a trusted mediator in place. Again, this is not just about convenience; it also helps save money, which would otherwise be spent on logistics associated with the same. One of the major problems that are addressed by such applications, and is not immediately obvious, is the elimination of middle man (which otherwise causes a lot of inconsistency and non-transparency in the process).

Issues on either side can be quickly escalated through the systems and required action be taken, with all the necessary information readily available via the interface. This serves as an important (but often underestimated) aspect of these applications. It also helps the concerned authorities (if applicable), in case of any non-trivial issues.

In this project, we create a centralized rental system for temporary accommodations (similar to Airbnb) to help facilitate different transactions between customers and hosts. On the host's part, these include:

1) ***Creating/Update/Removing Listings:***
   a) *Create Listing:* allows to create a (non-modifiable) listing + add (conditionally modifiable) availabilities
   b) *Remove Listing:* To remove a listing availability (& not the actual listing) - [Why? Refer **Section-3**]
   c) *Manage Listing:* For listing - update license, Listing Calendar (Unreserved availability) - update price(s)
2) ***Reservations:*** Check reservations on listings for associated information & facilitate them accordingly.
3) ***Access Anonymous Reviews:*** Check for reviews made by customers on past/current listings.
4) ***Manage Account:*** To update (specific) account details

whereas on the customer's side, it includes:

1) ***Search Available Listings:*** with respect to a set of filters
2) ***Make Payments & Reservations:*** to reserve a listing (post payment) for any of the available days
3) ***Perform Reviews:*** for past or current reservations
4) ***Manage Account:*** to deactivate account / update (specific) account details

The data sets made available by Airbnb [1] cover aspects such as accommodation listings, host information, year-round availability, neighbourhood information, customer review, host verification and so on. All of this and more is available in the form of 3 accumulated files (for each region). Neither the first-hand structure of the files nor the data format, were good enough, to be directly utilized in the form of a relational model.

A lot of data cleaning and structuring activities had to be put in place to make sure that the final design covers our use case-specific requirements. (Note: the dataset also contained some aggregate data and also some data analysis results. As such, we do not include this in our relational schema; but have our own data mining process(es) in place to address certain questions). Although it was not feasible to go through all regions, the process has been made generic enough, so as to be able to parse majority of the regions (if not all). At the least, it is guaranteed to parse all the Canada-specific regions.

For all data cleaning, structuring and migration purposes, the following process has been put in place:

1) Load dataset in (temporary) constraint-free tables in DB
2) Design the first-hand structure with an Entity-Relationship model and translate it to a relational schema
3) Put in place - Views, Stored Procedures, Index(es) etc
4) Migrate data from temporary tables to ones in new relational schema

For detailed steps (to be followed in sequence), please refer **Section-IX**.

II. Architecture Design

As shown in **Fig-1**, our project follows the Model-View-Controller (MVC) architecture. It is put in place with agile development in mind, which is essentially how the entire project has been developed so far. The actors - customers and hosts can access their respective features by logging into React (Client) application, which in turn consumes the REST APIs provided by Node.js micro-services to fulfil all the database-specific requests.

The micro-services act as a middle-ware between the client application and the database server, which is in keeping with the standard practice followed in industry. It also provides the much-needed security aspect to our project - so as to shield the database server from direct exposure to user end-points.

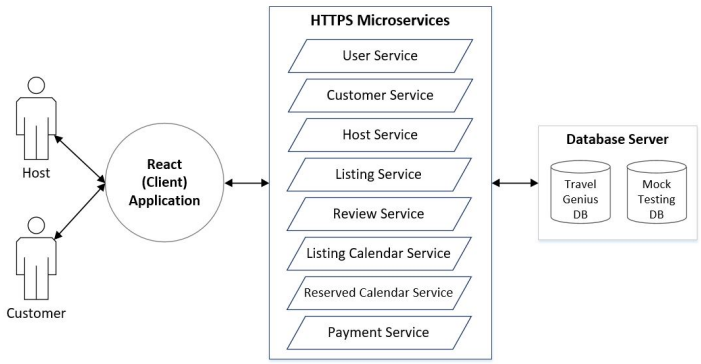


Fig. 1. MVC Architecture for Travel Genius Project

This is not structuring just for the sake of adhering to good practices. It has a crucial future application aspect to our project. For example, consider:

- *User Authentication:* This is usually performed via different services, and can take multiple forms such as OTP, fingerprint, authApp , etc.
- *Payment Service:* It is ideal to be facilitated via an isolate Payment Gateway.

Such updates, when required, can be made easily with minimal changes in the client application. Also, the micro-services mostly work in isolation, thus adhering to low coupling and high cohesion. The responsibilities, with respect to this architecture, have been divided as follows:

- **DB Design, Data Mining:** Nirav, Uditya
- **Microservices (REST APIs):** Uditya
- **Front End (React Application):** Nirav

III. Entity-Relationship (ER) Model

A. Dataset Breakdown

Fig-2 shows the Entity-Relationship model that was created on the basis of multiple observations made in the dataset. The original dataset [1] has 3 files in any region - namely 'reviews', 'listings' and 'listingCalendar'.

- 'listings.csv' [1] alone contains over 50 attributes on its own. This is where we get information related to **Listings** and **Hosts**.
- None of the datasets cover customer entity in full. However, the reviews dataset does contain the name and id of the customers (who had written the corresponding review). Hence, even though this need-not cover all the customers, still it can be utilized to a large extent. Hence, 'Review' and 'Customer' entities are created on the basis of data in 'reviews.csv' [1].
- Data in 'ListingCalendar.csv' helps create *'ListingCalendar'* and *'ReservedCalendar'* entities.

B. Some interesting design choices

In this section, we cover some of the interesting design choices we had to make while creating the relational model and the explanation behind those choices.

1) *User, Host, Customer*

- Host (implied by 'Listings.csv' [1]) and Customer (implied by 'Reviews.csv' [1]) contain 2 attributes in common - 'id' & 'name'. Hence, we create a Total Generalization on them.
- *Why an Overlapping Generalization?* The use-case for our application is such that only Customer-Accounts can make reservations and perform reviews. However, careful observation of the data shows that some host id(s) also exist as reviewerId(s) in 'Reviews.csv' [1] - which means that Hosts can also be Customers (and vice versa).
- *userEmail:* This is not readily available in the data set. However, it forms a crucial part of our application, as majority of the communication with users is expected to happen via email. Hence, we make this field nullable as of now, and allow for existing users to update their email-ids. Any new account going forward must have an email-id. Another aspect of this is with respect to login, which we will discuss in section **Section-V**.
- *Why no specialization for VerifiedHost? Aren't 'verificationSources' only available for VerifiedHosts?*
  No! - 'verificationSources' are even available for records where 'isIdentityVerified = false'. Hence, to a first guess, it is likely the case that 'verificationSources' also contain the sources from failed attempts.
- *Host has a field 'isSuperHost'? Wouldn't this imply a recurring relationship ?*
  Yes! - It must. However, to have such a relationship, we need the superHostId for all Hosts who are not superHosts. But this data is not made available in the dataset. Hence we are not considering this aspect in our project at this point.

2) *Listing, ListingCalendar, ReservedCalendar*

- *Why is ListingCalendar a weak entity set on Listing ?*
  'ListingCalendar.csv' [1] contains the multiple availabilities that were created for an existing **'Listing'**, where 'date' (in combination with 'listingId') forms the unique identifier.
- *In the given dataset, how can we identify records for the specialization 'ReservedCalendar'?*
  'isAvailable = false' could mean 3 possibilities:
  - **CASE-I:** All such listingCalendar(s) have been explicitly made unavailable by the 'host'.
  - **CASE-II:** All such listingCalendar(s) have been reserved by different customer(s).

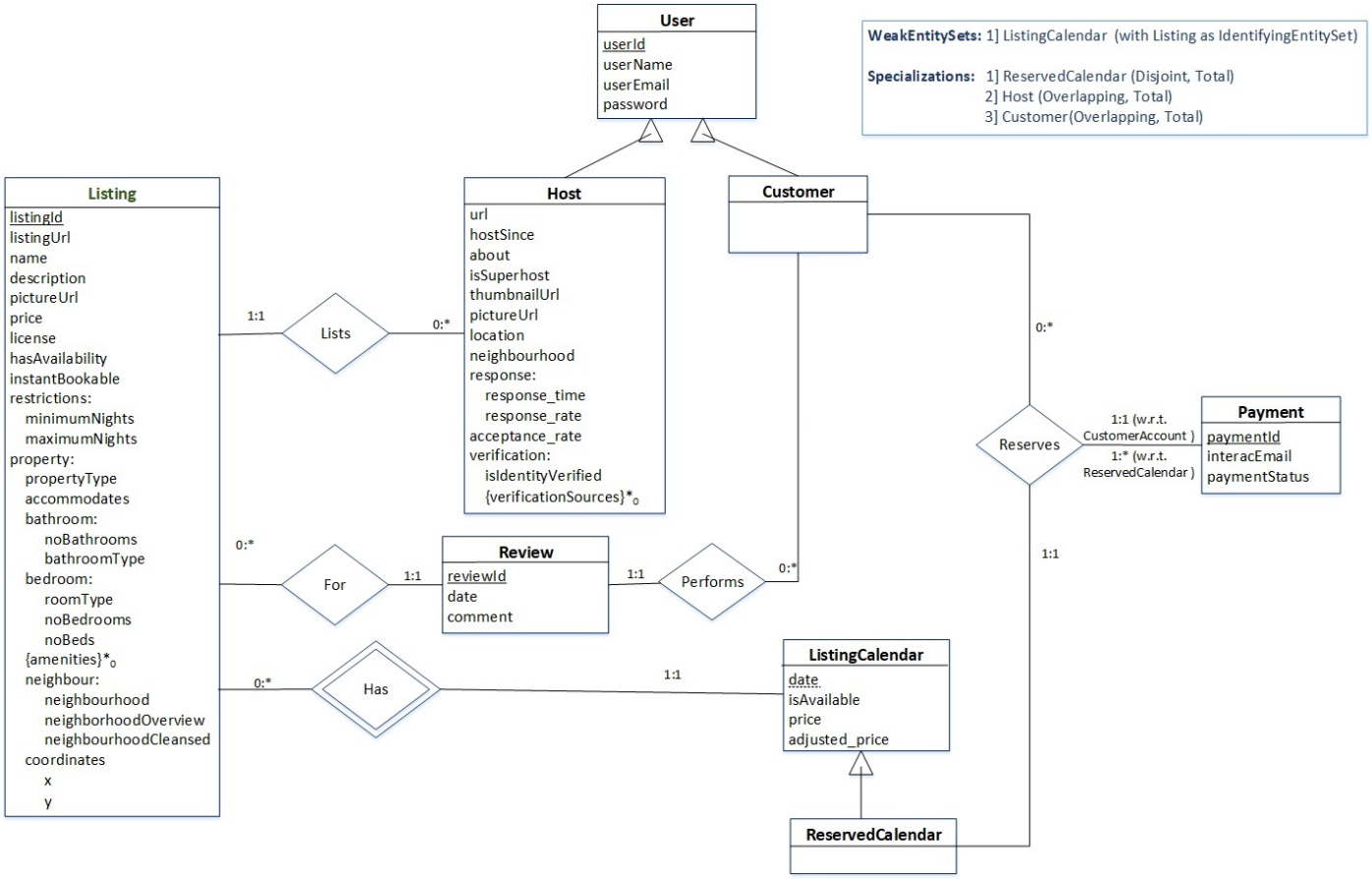


Fig. 2. Entity-Relationship (ER) Model

– **CASE-III:** Some of such listingCalendar(s) have been explicitly made unavailable by the 'host', and some have been reserved by different customer(s).

In the current dataset, there appears to be no way to know which of the 3 cases stand true. Hence, we make the following reasonable assumption and inference:

a) *(isAvailable = 'false')* $\Leftrightarrow$ *(listingCalendar isA reservedCalendar)*
b) If host explicitly wants to make a listingCalendar unavailable, then that listingCalendar is removed

3) Payment
- Just like Customer and ReservedCalendar, Payment is also not readily available in the datasets.
- However, for our use-case, we want to facilitate prepaid payments before confirming a reservation. Hence, we introduce a new table to allow this feature.
- But we keep the option open, to change the implementation to a dedicated Gateway-based payment (for likely future changes), in which case the paymentId in 'Reserves' relationship-set will then also come from a remote external entity.

C. *Summary*

- We have 2 types of *Users* (*Customers* & *Hosts*).
- A *Host* can have multiple *Listings*. Each listing can be made available for different dates (stored as *ListingCalendar*). A *ListingCalendar* (with current & future dates) can be reserved by a *Customer*, by making the required prepaid *Payment*. All the reservations are marked with 'isAvailable = false' in *ListingCalendar* and also have a corresponding entry in *ReservedCalendar*.
- A *Customer* can perform *Reviews* on past/current *Reservations*.

IV. Relational Schema

Now we translate the above ER model to a relational schema, that can be utilized by client application(s) via the REST APIs. A reverse-engineered depiction of the relational schema (in MySql) has been shown in **Fig-3**.

A. *Some important observations*

1) *Property as a Separate Relation*
Although, this was not required (as it had a 1-to-1 relationship with 'Listing' and was not resulting in any uncontrolled duplication), still we reallized that having such a large composite attribute as a separate entity would potentially make for more a adaptable design for (potential) future use - [E.g. if a future use case requires us to allow multiple listings for a particular property]

2) *Multi-valued Attributes*
- *verificationSources(Host) & amenities(Property)* are the only 2 optional multi-valued attributes in the schema.
- Hence we maintain them in separate tables with their individual value(s) and the corresponding (original) re-

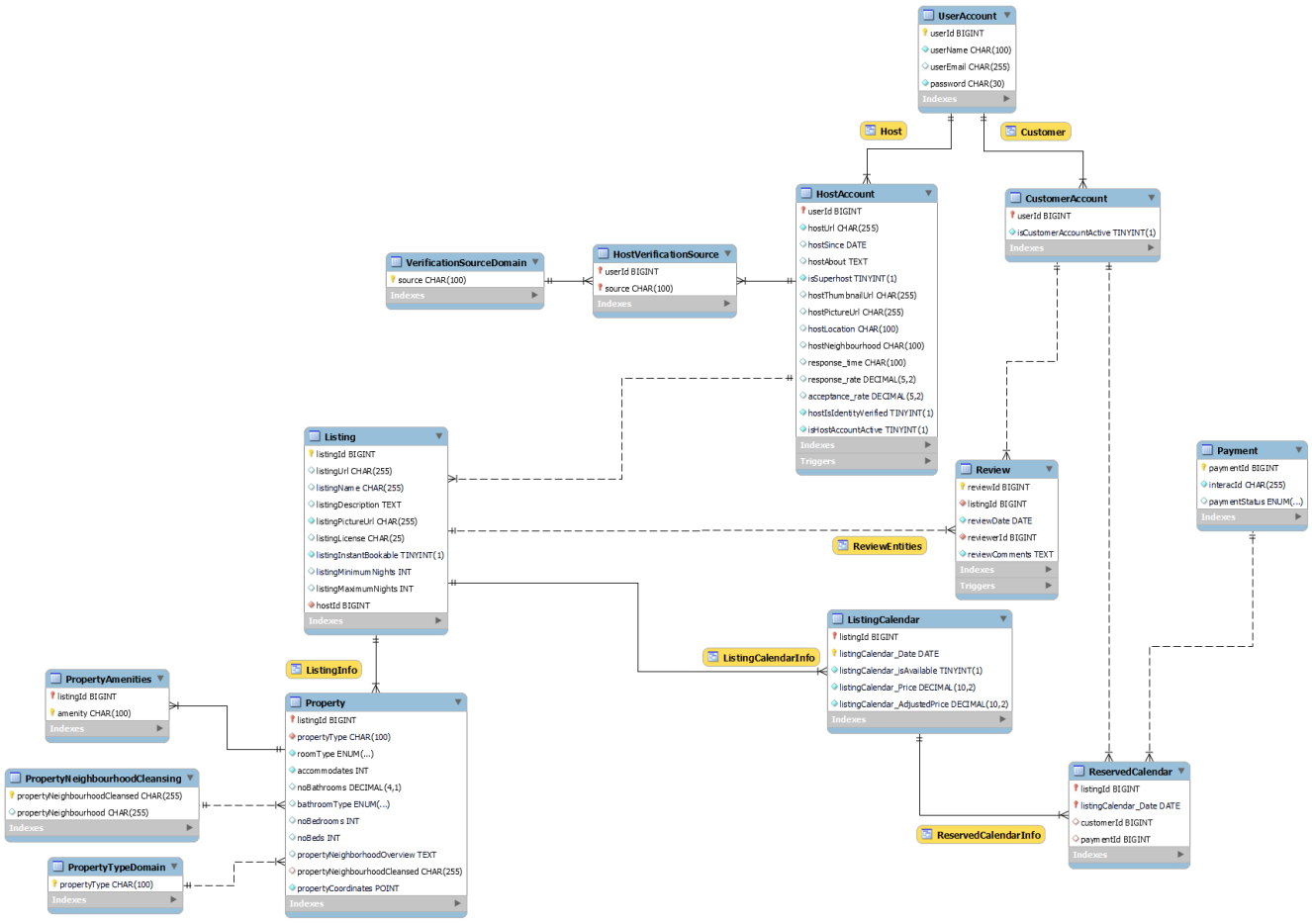


Fig. 3. Reverse Engineered depiction of Relational Model for Travel-Genius in MySql

lation's Primary Key. The combination of the 2 acts as the Primary Key for the new relation.

- We parse and process the array using Stored Procedures, which have been discussed in detail in next subsection.

3) *Domain Correctness*

- *'verificationSources(Host)'* currently holds values from among ['phone', 'email', 'photographer', 'work_email']. Hence we maintain a foreign key relation to a separate domain table ***'VerificationSourceDomain'*** for this attribute.
- *'paymentStatus(Payment)'* holds values from among ['Pending', 'Completed', 'Cancelled', 'Refunded']. An enum has been used here to enforce domain correctness.

- Why is the domain correctness performed differently for the two ?

- *paymentStatus* is something, for which we expect the domain to rarely change. Hence, ENUM() which requires higher privileges for any change to be made, can be facilitated by the respective authority.
- *verificationSources* on the other hand are different. These can take many forms, and have high potential for frequent changes. Hence, we want to allow other stakeholders also (with required DML privileges or maybe even via an interface) to be able to make those changes.

- Categorical data in Property:

- *propertyType:* contains over 90 categories and new types show up with each new added region. Hence we maintain a separate relation for domain correctness.
- For *roomType* & *bathroomType*, though, we make use of enum().

B. *Stored Procedures & associated feature abstractions*

Fig-4 shows some of the important stored procedures that we've defined in our relational database. (Refer: *'backend_travelgenius__data__\2_ViewsAndStoredProcedures' [4])* for detailed Stored Procedure implementations)

1) *Why have these procedures, when a lot of these functionalities could have also been facilitated via the REST-APIs?*

- *Need for database specific native Operations:* E.g. Consider saving array type data in separate relations. For new records, added by client, this can surely be facilitated by the REST middleware. *But what about records added in bulk from large datasets ?* →. This is where database native procedures like *'procedure_insertMultivaluedAttributes'* are really useful.
- *Rollbacks:* Transaction Management can be handled much more easily with stored procedures. Consider the previous example again. Constraint violation for one value, means that the system should undo inserts for all

S.N.	Name & I/O	Purpose
		Generic (multi-purpose) procedure
1	procedure_insertMultivaluedAttributes **Input:** tableName, identifierColumnName, valueColumnName, parentId, arrayString	Have an entry for each element from **arrayString** in the target **'tableName' -** against the corresponding **'parentIdentifier'** This generic procedure helps fill array (/vector) type data in separate relations.
		Helper procedures for multi-valued attributes
2	procedure_insertAmenities	These 2 procedures are used to properly format and insert array/vector data (available as strings) from the 3 temporary constraint-free tables (which contain all migrated data). They are only used when migrating data from the 3 temporary tables to the correct relational schema.
3	procedure_insertHostVerifications	They in-turn make use of **procedure_insertMultivaluedAttributes** , by calling it multiple times - once for each record to be migrated.
		Main Insert Procedures (for Inserts happening from Client-side only)
4	procedure_insertHost **Input:** all required attributes	1. Inserts record in UserAccount (Generalization) 2. Inserts record in HostAccount / CustomerAccount (Specialization) - with **userId** (auto-generated in Step-1) 3. Returns **userId** Rolls back entire process, in case of issues in between any operation.
5	procedure_insertCustomer **Input:** all required attributes	
6	procedure_insertListing **Input:** all required attributes	1. Inserts record in Listing (Parent Entity) 2. Sets the listingUrl using Simple Function, as defined in **'Section IV-D'** 3. Inserts record in Property (Composite as an Entity) - with **listingId** (auto-generated in Step-1) 4. Returns **listingId** Rolls back entire process, in case of issues in between any operation.
7	procedure_insertListingCalendar **Input:** all required attributes	1. Inserts record in ListingCalendar, with **isAvailable** set to '1'. **Note:** This is for insertion of a new record via the client-interface only. As such, when ListingCalendar is only added by a host (for a Listing) only if he/she wants to make it available for that day.
8	procedure_insertReservedCalendar **Input:** all required attributes	1. Sets **'isAvailable'** (in ListingCalendar) to **false** 2. Inserts record in ReservedCalendar Rolls back entire process, in case of issues in between any operation.
9	procedure_insertPayment **Input:** all required attributes	1. Inserts record in Payment 2. Returns **paymentId**
10	procedure_insertReview **Input:** all required attributes	1. Inserts record in Review, with **reviewDate** set to CURDATE(). 2. Returns **reviewId**
		DELETE Procedures
11	procedure_deleteListing **Input:** all required attributes	1. Deletes record from Property (Composite as an Entity). 2. Deletes record from Listing (parent Entity) Rolls back entire process, in case of issues in between any operation.
		Update Procedures
12	procedure_updateHost **Input:** all required attributes	1. Updates attributes in UserAccount (Generalization). 2. Updates attributes in HostAccount (Specialization) Rolls back entire process, in case of issues in between any operation.

Fig. 4. Important Stored Procedures

other values in that array (for the record in question) →. these can be handled much more efficiently with stored procedures (with less potential for making mistakes).

- *Multiple sequential operations:* Can be handled much more efficiently - natively via stored procedures, also with rollback benefits as discussed earlier.

2) *Some important observations*

- We've used stored procedures, only where we saw fit - on the bases of above potential benefits. All other operations, have been directly performed via the REST API middleware.
- *Why have procedure_insertPayment ?*
  It's true that this is a simple record insertion. But we had a procedure for every other potential insert that could happen in our application. Hence, we decided to have one for Payment as well, to maintain consistency.

3) *Some use-case specific omissions*

a) *Why no update/delete for listing ?*

- Listings have available and reserved days (Listing-Calendar (with isAvailable = true) and Reserved-

TABLE I
VIEWS & ASSOCIATED ABSTRACTIONS

S.N.	Name	Purpose
1	Host	1. Inner Joins UserAccount and HostAccount / CustomerAccount 2. User-password is omitted
2	Customer	
3	ListingInfo	1. Inner Joins Listing and Property (this is a one-to-one join) 2. Left outer joins the result with PropertyNeighbourhoodCleansed
4	ListingCalendarInfo	1. Joins ListingCalendar and ListingInfo
5	ReservedCalendarInfo	1. Inner Joins ReservedCalendar & ListingCalendarInfo and Left-outer-joins with Customer & Payment 2. Only provides paymentStatus and id from payment (Omits confidential details) 3. Note: The joins with Customer and Payment are LEFT OUTER Joins, since they might have missing data.
6	ReveiwEntities	1. Reviews is extended to get associated hostId (for the corresponding Listing) 2. The Goal is to have the identifiers for all entities involved

Calendar respectively) associated with them. Any changes made to a listing have implications on the reservations for different days. Hence, we have decided to not allow updates to a listing, once it is created.

- *What if the host wants to have changes in a Listing for any future reservations ?*
  This will have to be facilitated by creating a new listing for those availabilities →by the Host.
- *Deleting an Availability:* is allowed (directly implemented in the middleware)
  Deleting a Reservation: not allowed (unless cancelled by the customer)
- **Note:** 'Cancellation by Customer' is not yet implemented in the application (due to time limitations), nor was it included in the initial proposal.

b) *Why no delete for User (Host/Customer) ?*

- For this application, we are not allowing deletion of a user account. However, we do allow deactivation of individual Customer and/or Host account (not user-Account→since the specialization is overlapping)
- *Customer Deactivation:* Allowed only when customer has no current/future reservations.
  Host Deactivation: Allowed only when host has no listings with current/future reservations.
- Both deactivations are handled by the REST APIs.

c) *Why no delete for Payment ?*

- A payment record must never be deleted. However, the status can be updated to Cancelled/Refunded (E.g. when a reservation is cancelled)
- Reservation cancellation & related Payment status update feature is not yet implemented though, as discussed earlier.
- However, the database design is very well structured to incorporate these features in the future (as and when required).

d) *Why no delete/update procedure for Review?*

- Was simple enough to be implemented directly via the REST middleware.

e) *Why no update procedure for Customer?*

- A customer does not have any updatable attributes in CustomerAccount (accept for isCustomerAccountActive - which is deactivation specific). Hence, updating only UserAccount specific attributes was simple enough to be implemented directly in the REST middleware.

C. *Views & Associated Abstractions*

Table-I describes the views that we've created in the database. All these Views serve use-case specific purpose, in addition to protecting secure/confidential data. (For detailed implementation of these views, please refer: *'back-end_travelgenius__data__\2_ViewsAndStoredProcedures.sql' [4])*

- *What about userAuthentication?*
  - We have a dedicated REST service for this.
- *What about host verification?*
  - This feature is not yet implemented (as is consistent with initial proposal). But for any future implementation, we believe that it should have a dedicated REST service and will likely require remote calls to an external service for document verification.

D. *Some notable constraints & triggers (Other than Primary and Foreign Key Constraints)*

Fig-5 lists some of these additional constraints. Note that we use a trigger for dates, since CURDATE() cannot be used in a CHECK constraint in MySQL.

1) *Simple Functions*
   - hostUrl = concat('https:/www.airbnb.com/users/show/', userId)
   - listingUrl = concat('https://www.airbnb.com/rooms/', listingId)
2) *Why not have a check-constraint/trigger to enforce this?*
   - Since the identifiers have *AUTO_INCREMENT* property.
3) *How else are we enforcing this ?*
   - The inference on these Simple Functions was made after observing that the records in the dataset imply this.
   - Now, we need to ensure that any new records being inserted in the future do not violate this constraint.
   - We do this by having 'Stored Procedures' (Refer **Section-IV-B)** for inserting new Hosts and Listings, wherein the url is created and populated by the respective procedure after the Id is made available to it (via previous insertion).
4) *Why do we have PropertyNeighbourhoodCleansing relation ?*

S.N.	Constraint	Attributes Applied On
	CHECKs	
1	Email CHECK(attribute REGEXP "^[a-zA-Z0-9][a-zA-Z0-9.!#$%&'*+-/=?^_`{\|}~]*?[a-zA-Z0-9._-]?@[a-zA-Z0-9][a-zA-Z0-9._-]*?[a-zA-Z0-9]?\\.[a-zA-Z]{2,63}$")	• UserAccount(userEmail) • Payment(interacId)
2	URL CHECK (attribute REGEXP '^(http|https|ftp):\\/\\/[\\w\\-]+(\\.[\\w\\-]+)+([\\w\\-.,@?^=%&:/~+#]*[\\w\\-@?^=%&/~+#])?$')	• HostAccount (hostThumbnailUrl) • HostAccount (hostPictureUrl) • Listing (listingPictureUrl)
3	Password (Length - 8 to 20, with a special character) CHECK(password REGEXP "^(?=.*[!@#$%^&*(),.?:{}\|<>])[A-Za-z0-9!@#$%^&*(),.?:{}|<>]{8,20}$")	• UserAccount (password)
4	Positive Integer/Double Check	Applied to multiple attributes as applicable - including price, counters, etc ..... as applicable
5	Range Check CHECK(attribute BETWEEN 0 AND 50000)	• ListingCalendar (listingCalendar_Price) • ListingCalendar (listingCalendar_AdjustedPrice)
6	Multi-attribute Check CHECK (listingCalendar_AdjustedPrice <= listingCalendar_Price)	• ListingCalendar (listingCalendar_Price) • ListingCalendar (listingCalendar_AdjustedPrice)
7	Not Null	Applied as and where applicable (**Refer:** backend_travelgenius__data__\1_RelationalModel.sql) **[4]**
8	Unique	Applied as and where applicable (**Refer:** backend_travelgenius__data__\1_RelationalModel.sql) **[4]**
	ENUMs	
9	paymentStatus ENUM('Pending', 'Completed', 'Cancelled', 'Refunded')	• Payment(paymentStatus)
10	roomType ENUM('Shared room', 'Private room', 'Hotel room', 'Entire home/apt')	• Properties (roomType)
11	bathroomType ENUM('bath', 'shared half bath', 'shared bath', 'half bath', 'private half bath', 'private bath')	• Properties(bathroomType)
	TRIGGERs	

10	CREATE TRIGGER constraintReviewDate
	BEFORE INSERT ON Review
	FOR EACH ROW
	BEGIN
	IF NEW.reviewDate > CURDATE() THEN
	SIGNAL SQLSTATE '45000'
	SET MESSAGE_TEXT = 'ReviewDate must be a current or prior date';
	END IF;
	END;
		• Review (reviewDate)

11	CREATE TRIGGER constraintHostSince
	BEFORE INSERT ON HostAccount
	FOR EACH ROW
	BEGIN
	IF NEW.hostSince > CURDATE() THEN
	SIGNAL SQLSTATE '45000'
	SET MESSAGE_TEXT = 'hostSince must be a current or prior date';
	END IF;
	END;
		• HostAccount (hostSince)

Fig. 5. Some notable constraints & triggers (Other than Primary and Foreign Key Constraints) [2] [3]

We need this relation to make sure that the following functional dependency (from Property) is not violated:

propertyNeighbourhoodCleansed $\rightarrow$. ***propertyNeighbourhood***

E. Indices

The use-cases for our application were such, that most of the operations already had the required indices in place due to the presence of relevant primary keys and foreign keys as required. However, there is one complex query (for Searching Listings) that required more indexes. Every filter in that query is range-based; hence a B+ tree a good option for it. The dynamic implementation of the query in node.js has been shown in **Fig-11**. Further, user authentication queries (point queries) also require 2 additional indexes.

After doing analysis with 'explain' and 'explain analyze', the following indexes were found to be the most efficient and feasible:

1) *Search Query*
   CREATE INDEX
   index_listing_isAvailable_date_price_ListingCalendar ON
   ListingCalendar (listingId,
   listingCalendar_isAvailable,
   listingCalendar_Date,
   listingCalendar_Price,
   listingCalendar_AdjustedPrice);
2) *UserAuthentication Queries*
   - CREATE INDEX index_listing_userId_password ON UserAccount(userId, password);
   - CREATE INDEX index_listing_userEmail_password ON UserAccount(userEmail, password);

V. USER APPLICATION

We've already briefed the features that our application supports on Page 1, followed by the high level system architecture.

```
`SELECT * FROM ListingCalendarInfo
WHERE listingCalendar_isAvailable = 1
    AND (listingCalendar_Date BETWEEN '${input.dateRange.lowerBound}' AND '${input.dateRange.upperBound}')
    AND ( (listingCalendar_AdjustedPrice BETWEEN ${input.priceRange.lowerBound}  AND  ${input.priceRange.upperBound})
            OR
            (listingCalendar_Price BETWEEN ${input.priceRange.lowerBound}  AND  ${input.priceRange.upperBound})
        )
    AND (noBeds BETWEEN ${input.noBeds.lowerBound} AND ${input.noBeds.upperBound})
    AND ST_Distance(propertyCoordinates, ST_GeomFromText('POINT(${input.location.position.x} ${input.location.position.y})'))
                    BETWEEN ${-input.location.maxRange / 100} AND ${input.location.maxRange / 100}`;
```

Fig. 6. Search Query

All those features have been implemented via a React based User Interface. Fig 7, 8 and 9 show screenshots of log in page, search output, and Listings Management. Fig 10 presents date selection page to book any property. These are only for a small subset of the features that have been implemented. The entire suite of implemented features has been well demonstrated in the video. We've used a template from the internet to create this react interface.

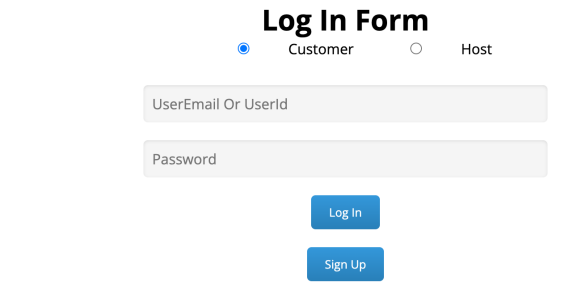


Fig. 7. Login form

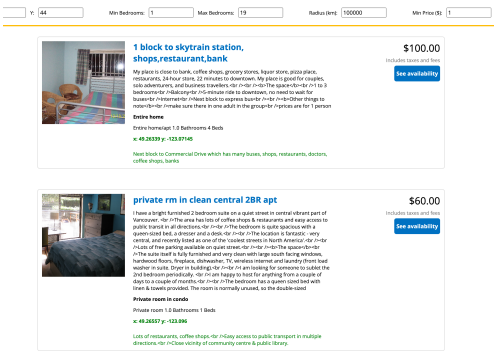


Fig. 8. Search Output

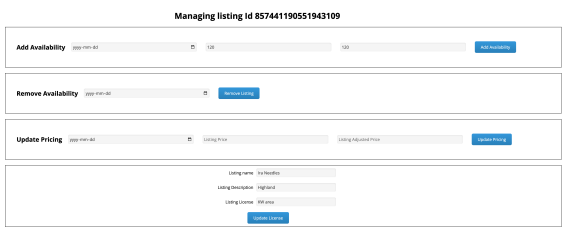


Fig. 9. Manage Listing

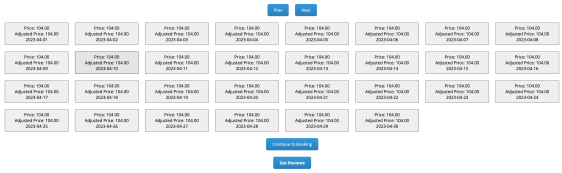


Fig. 10. Date Selection during booking step

VI. Testing

A. *REST APIs Testing*

The test suites for REST APIs are fully automated via the **'Mock Testing DB'**, which is a short lived, temporary database with dummy data and essentially the same schema as the original one. Hence, in addition to the REST APIs, the unit tests also end up testing several aspects of the database design.

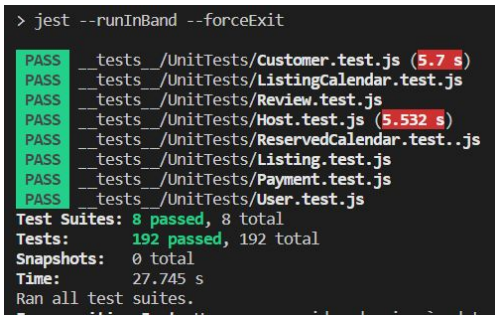


Fig. 11. REST APIs - Unit Tests (Sample Run)

We use JEST Framework for the test setup. To view the complete test-setup, one can refer the following set of files in REST APIs (Backend_TravelGenuis):

1) ***./test_setup.js:*** which is the configuration that is run at the start of each test-file implementation
2) ***./Config/__mocks__/dbProperties.js:*** Directs test framework to use (mock & temporary) testTravelGenius database (created in test_setup.js) instead of the real database.
3) ***./__tests__/MockData/3_LoadData.sql:*** (run in test_setup.js) Adds dummy (but relevant & feasible) data to mock database, to perform testing on.
4) ***./__tests__/UnitTests/<filename>.test.js:*** Each file in this directory is a dedicated test implementation for the corresponding service & contains multiple test-suites for the same.

The entire test-setup can be verified by running ***'npm test'*** command from the parent directory of ***Backend_TravelGenuis***

B. *Front-end Testing*

The front-end application has been manually tested and the detailed test suite implementation has been provided in ***'ece656_frontend / FrontendTests.xlsx'***. Considering the constraint in terms of time and resource, we have restricted ourselves to manual testing only.

Fig. 12 a screenshot for one of the test-suites. Remaining test-suites can be found in the corresponding tabs as marked below.

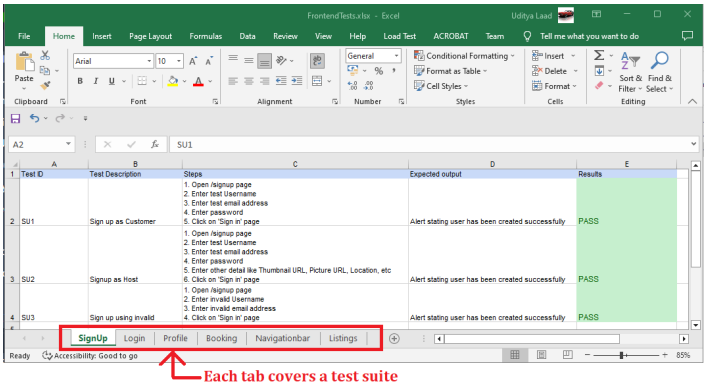


Fig. 12. Front-end Manual Testing (Each Tab covers a test-suite)

VII. Data Mining Exercise

A. *Domain Appropriate Question: What is the likelihood of success for a listing with respect to its features ?*

- *Dependent Attribute: How do we measure success ?*
  We have 2 Relations 'ListingCalendar' (Availabilities for a listing on multiple unique days, which may also be reserved) & 'ReservedCalendar' (ListingCalendar that are reserved by customers). Hence we define:

$$SuccessRate(listingId) = \frac{|ReservedCalendar(listingId)|}{|ListingCalendar(listingId)|}$$

  Success Rate is not directly available as a field in the database. Hence, we trigger an SQL query to compute this field and associate it with the corresponding Listing features vis Jupyter Notebook. (The query and resulting CSV can be found in *'backend_travelgenius__datamining_\DataMining.ipynb' [4]*)
- *Independent Attibutes*
  Subset of features associated with listings (to be selected)

B. *Technique(s) appropriate to the question*

The features associated with Listing are mostly dominated by categorical and numeric values. Note that fields like listingUrl, listingId, license are of no relavance here - since these essentially identifiers. Although, pictureUrl does indeed point to a hosted picture. However, due to time and resource limitation, we have decided not to pursue image processing (& will hence treat it under future scope for analysis).

There are a lot of numerical attributes where the distribution is over a small range, hence potentially convertable to categorical data (ordinal or otherwise). However there are some others which cover a relatively larger range (e.g. price).

Now, given the nature of the question and the features to choose from, there are 2 potential model building techniques that we see fit:

1) *Linear Regression:*
   There is a possibility (although very low) of a linear relationship between the dependent and independent attributes. We can treat this as our NULL hypothesis, the results of which ill be clear at the feature selection stage, when performing correlation analysis.
2) *Multi-Layer perceptron (MLP):*
   There is a high chance that the relationship we are seeking is not linear. In such a case, linear regression is infeasible. MLP, on the other hand, provides the flexibility of building a non-linear model using multiple hidden layers with (specific) activation functions, as applicable to the situation.

Note that the target value is probalistic (in range [0, 1]). This is the major reason behind potential selection of above 2 techniques, among other reasons - as will be discussed in the subsequent sections.

More techniques to be considered for feature selection are:

1) Correlation Analysis
2) Checking Frequency Distribution
3) Looking at potential use of Principal Component Analysis

C. *Pre-processing of Data*

Both the model-building techniques under consideration require features to be in numerical form. Having reasonably ommitted some features in step-1, we now look at some of the other features that require a look-in. For the first two attributes, we are ***defining fuzzy boundaries*** to derive meaningful categories for them.

1) *roomType*
   This is categorical data. Close observation shows that we can inbibe meaning in the labels that we intend to assign for these categories. E.g. An entire home/apt is typically considered to be better than a Hotel room. As such these categories are comparable, giving us an opportunity to (somewhat) make them ordinal. Hence, we define and utlize the following mapping: { *'Shared room': 1, 'Private room': 2, 'Hotel room': 3, 'Entire home/apt': 4* }
2) *bathroomType*
   This is also categorical data. Consider the following mapping: {*'private bath': 1, 'shared bath': 2, 'half bath': 3, 'bath': 4, 'shared half bath': 5, 'private half bath': 6*}
   Ideally, most people would consider 'Private-bath' to be the best of the lot. However, consider the 'bath' cateogory. It does not tell much about anything - as such, considering it to be a neutral type looks like a reasonable assumption to make. Why ? Different individuals are likely to treat this differently. Some customer may find it skeptical, while others may ignore it.
3) *propertyType*
   This is categorical data that needs to be translated to meaningful numerical labels. For this purpose, we use Vectorization technique available under scikit learn.
4) *propertyNeighbourhood & propertyNeighbourhoodCleansed*
   These fields can be made categorical and will be meaningful with respect to proximity of location. However, we

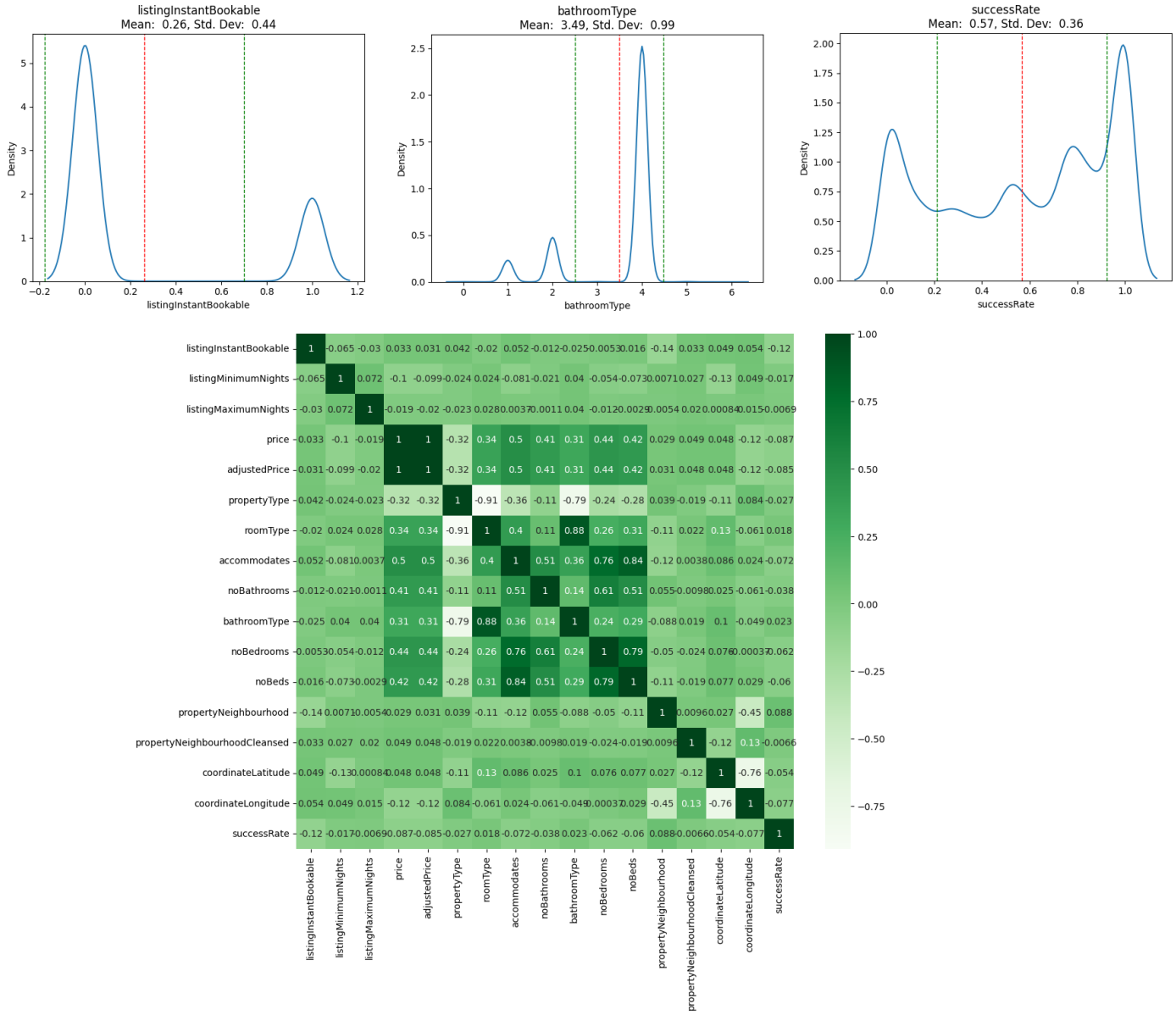


Fig. 13. Frequency Distribution for 2 interesting attributes and Results of Correlation Analysis [7]

see that they are functionally dependent on propertyCoordinates. propertyCoordinates is likely to cover for any affect they might have on the successRate.

D. *Correlation Analysis & Frequency Distribution*

Note: our feature selection process had already begun from step-1, when we gradually started eliminating certain features on the basis of reasonable arguments and/or techniques. We now look at the frequency distribution and correlation analysis chart (**Fig-13**) for better understanding of the relationships. For frequency distribution of all attibutes, please refer *'back-end_travelgenius__datamining_\DataMining.ipynb' [4].*

As is evident from **Fig-13**, none of the independent attributes have an individual correlation coefficient greater than 0.5 with respect to the successRate. Such low collinearity values falsify our initial null hypothesis, which suggested that there exists a Linear Relationship between success rate and listing feaures (of importance).

How can we still make use of correlation analysis? We also have frequency distribution and some more graphs to make certain inferences on feature selection. The middle portion of the correlation analysis signifies something intersting. There is high collinearity among some independent attributes (multi-colinearity). This can potentially mean that they act together in their effect on successRate. This however, is not an inference, but rather a logical observation by looking at the figures and types of attributes at play. The major point, being that, we cannot simply eleminate them due to low colinearity.

- *What about price & adjustedPrice ?*
  The frequency distrbution seems to suggest no major differences between the two - *Should we get rid of one of them ?* - No! Discounts often tend to occur in relatively smaller chunks; a small change in price may still have a big impact on the successRate, as is quite frequent with many discounted products.

- *Frequencies for 2 interesting attributes:* isInstantBookable is a binary attribute and bathroomType is a categorical attribute. listingInstantBookable for one of the regions has a true:false frequency ratio of 1:6 with almost negligible correlation, which could lead one to believe that the value has no impact on the target and will be simply an overhead on the model builder. However, when multiple regions are taken together, the ratio raises to nearly 1:3 with a comparably higher (even though still low) correlation. A similar pattern can be seen with bathroomType here which has most values concentrated at 4. It also has a very low correlation with the target. There is no clarity on such 2 attributes at the moment, but considering their domain size and nature, we decide to keep them in our feature set.

After considering all the eliminations so far, we are now left with the following attributes in our feature set: { price, adjustedPrice, propertyType, roomType, accommodates, noBathrooms, bathroomType, noBedrooms, noBeds, listingMinimumNights, listingMaximumNights, propertyCoordinates(lat, long), listingInstantBookable }

Do we need Principal Component Analysis (PCA)?
PCA would have been useful if we had a comparatively large set of attributes, and also in case there was a particular need to pick relavant characteristics from the feature set. However, for a total of 11 independent attributes, with meaningful data distribution in most, dimensionality reduction does not make sense here. One attribute that we had to be carefull about was 'propertyType' as we had applied scikit based vectorization for it's label encoding. But this is something, that MLP should be more than capable of handling with right configuration in place.

E. *Model Building using Multi-layer perceptron (MLP)*

- We use Tensorflow [6] to configure & build our model.
- An MLP based model makes use of numerous hidden-layer nodes, each with their dedicated activation functions to allow non-learity in the model.
- *Choice of parameters:*
  Activation Function(s): We tried 3 main activation functions in different configurations:
  1) **relu** - due to its non-linearity, sparsity, ability to avoid vanishing gradient problem
  2) **sigmoid** - since the target is probabilitic & since it also helps with input normalization
  3) **tanh** - since it is less prone to saturation than sigmoid

  Number of hidden layers: due to the number of features involved, we initially though that 3 layers with gradually increasing neurons might serve well. However, 3 layers proved to be insufficient and we ended up using: 4 layers with (256, 512, 1028, 1028) neurons respectively. The initial parameters were decided on the basis of our experience with other neural networks in the past (for other courses). We were expecting relu to be of particlular use here. However, the feature selection, and mapping turned out to be quite significant in being able to achieve such high accuracy with only sigmoid in place at each neuron. This was not straightfarward though. Initially, we made the mistake of assigning simple dataframe based labels for categorical data, which resulted in very poor accuracy. It was only after we devised the fuzzy boundaries and assigned meaningful labels for the attributes, that we were able to achieve the end results.
  Measure of Accuracy: We use Mean-square-error as the measure, due to use of Sigmoid as activation function & since it makes it easy to optimize gradient descent.
  Learning Optimization: We use Adam - since it combines momentum and adaptive learning rates, to achieve faster convergence and better performance.

F. *How model was validated ?*

The model was validated in 2 phases:

- *During model building process - with Test Set Validation:* This was utilized during initial model creation process - to check the effectiveness of the built model, look for overfitting issues, etc
- *K-Fold Cross Validation - Post model creation:*
  we validated our model thouroughly by performing K-fold validation with k = 5. The process was as follows:
  1) Divide the input-set into 5 equally distributed test sets
  2) For each test-set 't' in input-set 'T': Run the model builder with same properties; (T - t) as training set & then validate the model against 't'
  3) Compute the mean and standard deviation (of scores - in terms of Mean Square Error) for the 5 tests
  4) Compare results to assess the model

G. *Results from Model Builder*

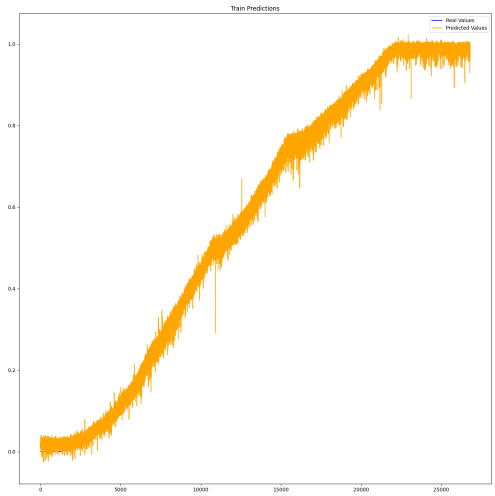


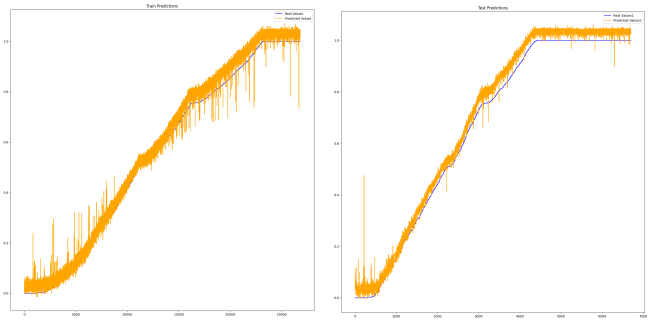
Fig. 14. MLP - Training Predictions

Fig-14 shows the training set predictions on the original model. We were able to achive these results in under 25 epochs.

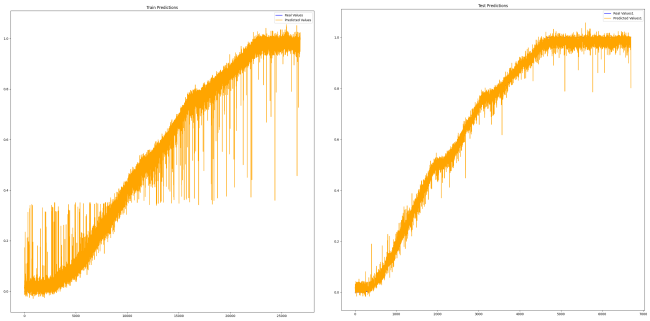
Fig-16 shows the testing set predictions for the same model. As is evident, the model is able to prdict the target with high accuracy. The MSE (Mean-Square-Error) for training was 0̃.001739, whereas for testing, it was 0̃.001731.

H. *Results from K-fold cross-validation*

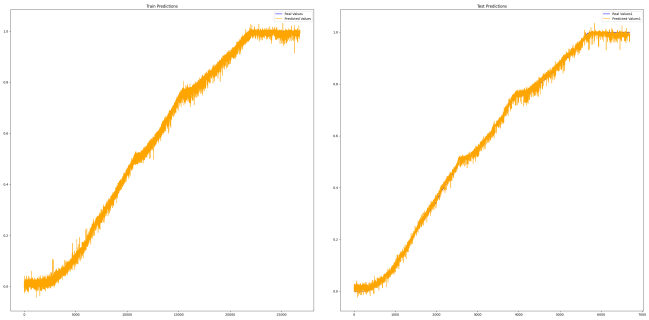
Fig-15 shows result of K-fold validation with k = 5. The scores(MSEs) for all individual folds as well as the mean and standard deviations are consistent with what was expected. Hence, we have succesfully created an MLP based model to answer our domain-specific question and then validated the same using K-fold cross validation.



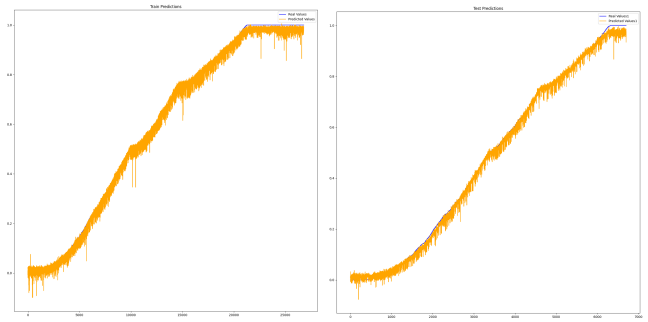
```
Fold 1:
Train MSE 0.001135197700932622:
Test MSE 0.001412962912581861:
```



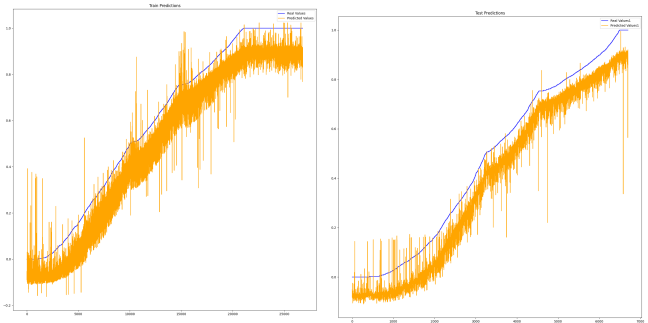
```
Fold 2:
Train MSE 0.0009392917854711413:
Test MSE 0.0005924122524447739:
```



```
Fold 3:
Train MSE 0.00012075934500899166:
Test MSE 0.00015159734175540507:
```



```
Fold 4:
Train MSE 0.0002336821635253727 4:
Test MSE 0.0002763844095170498:
```



```
Fold 5:
Train MSE 0.011858870275318623:
Test MSE 0.008773842826485634:
```

Overall:

```
Train MSE:
Mean:  0.00285756025405135
Standard Deviation:  0.004517625133420848

Test MSE:
Mean:  0.002241439948556945
Standard Deviation:  0.0032956589415159003
```

Fig. 15. K-Fold Validation Results

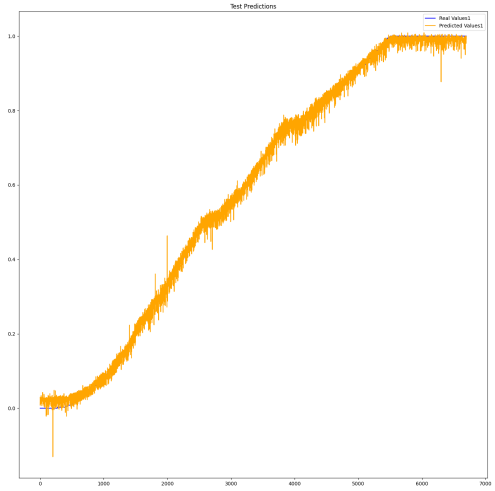


Fig. 16. MLP - Testing Predictions

VIII. Final words

1) The features that had been initially proposed have all been implemented using Views, Stored Procedures, REST APIs and the React application.
2) The reasons for each of the design choice made for the relational schema have been clearly mentioned in the report in detail.
3) The data mining exercise has also been thoroughly discussed in this report. The results achieved after performing K-fold validation were also quite significant.
4) For most parts, the report should be the first point of reference, with the video providing a brief of all aspects involved.

IX. Submission

In addition to the submissions made on LEARN, the project submissions can also be found in the following directories:

1) *DB Design & Setup:* https://git.uwaterloo.ca/ulaad/backend_travelgenius/-/tree/master/__data__
2) *Microservices:* https://git.uwaterloo.ca/ulaad/backend_travelgenius
3) *Frontend:* https://git.uwaterloo.ca/nvtalavi/ece656_frontend
4) *Data Mining:* https://git.uwaterloo.ca/ulaad/backend_travelgenius/-/tree/master/__datamining_

Access to both repos has been granted on GitLab. Follow the instructions provided in **Readme.md** in the given sequence, to get the app running on local. There are 2 options for database setup:

1) Marmoset
2) Local Setup

Readme.md provides instructions for both. Choose any one as per preference.

References

[1] Airbnb, Listings and Review dataset, http://insideairbnb.com/get-the-data/
[2] Email-Regular Expression, Stack Overflow, https://stackoverflow.com/questions/15560004/mysql-check-constraint-for-email-addresses
[3] Regular Expression, Regex 101, https://regex101.com/
[4] Database & Backend Directory, https://git.uwaterloo.ca/ulaad/backend_travelgenius
[5] Frontend Directory, https://git.uwaterloo.ca/nvtalavi/ece656_frontend
[6] Tensorflow Keras - MLP https://www.tensorflow.org/guide/core/mlp_core
[7] Feature Selection with sklearn and Pandas https://towardsdatascience.com/feature-selection-with-pandas-e3690ad8504b