Prepared by Group 4's – Meg Alapati, Connor Barker and Shwapneel Ishraq under the mentorship of Ricardo Rolan

# 356 PROJECT REPORT

USED CARS DATASET

# PROJECT DESCRIPTION

Group 4

## Contents

Project scope and functionality ........ 1
Project Design Assumptions ........ 1
Communication Plan and Group Organization ........ 2

## Project scope and functionality

The general purpose of this application is to allow users to buy and sell cars. Our application is going to be used by Dealers and Customers and maintained by System Administrators. Our database is going to be used by Data Science Analysts to gain insights to improve customer experience and inform the expansion of our product.

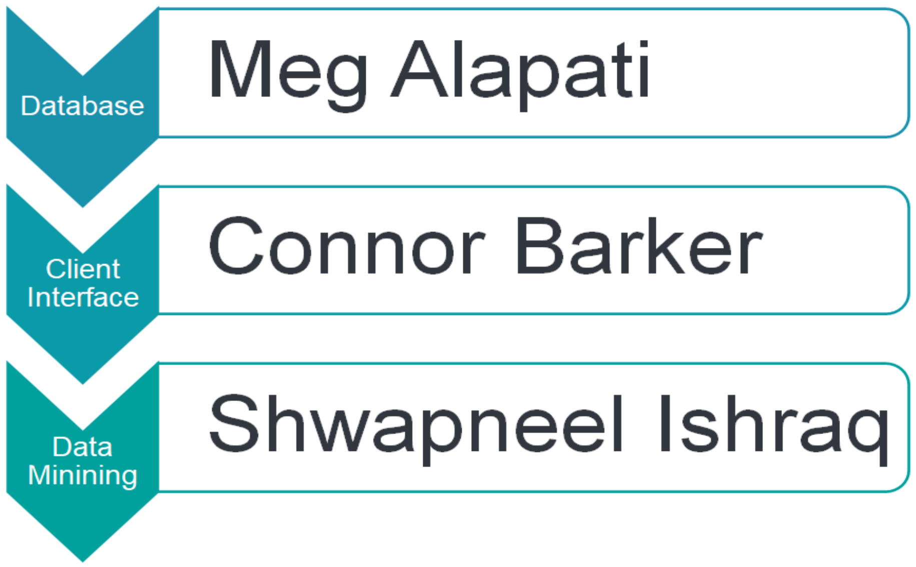
We set out to implement the following functionalities taking into consideration time constraints to complete the project and coordinating the release of lecture slides with our milestones:

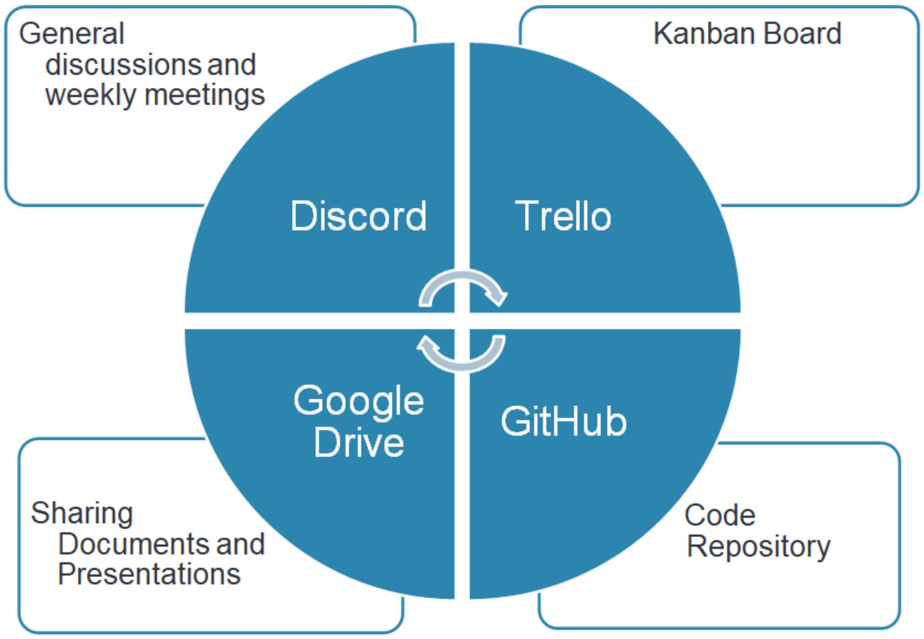
1. Sign up/ Sign In – To log onto the application
2. Implement views for Dealer, Customer, Admin
3. Make a listing with that new account
4. Edit the listing with the same account
5. Remove a listing with the same account
6. make appointment for listing
7. edit appointment that the user created
8. cancel appointment that the user created
9. show filtering/sorting while searching for listings and cars
    a. Year
    b. Make
    c. Model
    d. Price
    e. New/Old
    f. Zip Code
    g. Days on the market
    h. Seller rating, etc.

## Project Design Assumptions

1. A user can at any time list any car for sale that is not currently listed.
2. A user can at any time purchase any car they do not already own, i.e., they can’t buy their own car.
3. Any information on a listing, except VIN, can change at any time.
4. A user can only edit or remove listings which they own.
5. No upper limit on the number of appointments for a car sale
6. No upper limit on the number of listings that can be created by an account

## Communication Plan and Group Organization




General discussions and weekly meetings
Discord
Kanban Board
Trello
Google Drive
Sharing Documents and Presentations
GitHub
Code Repository

As far as a system for distributing tasks, we have a weekly meeting set in which we go over the work done in the last week & collectively agree on which tasks need to be done and adding them to the TODO lane of the Kanban board. So far, this has been implemented using Trello to create, size, assign, and track weekly tickets.

# DATABASE DESIGN

Meg Alapati

## Table of Contents

Entity Relationship Design ........ 1
Relational Model Design ........ 5
Physical Schema Indexes ........ 8
Changes to the database to support the next phase of the application ........ 9
Bibliography ........ 10

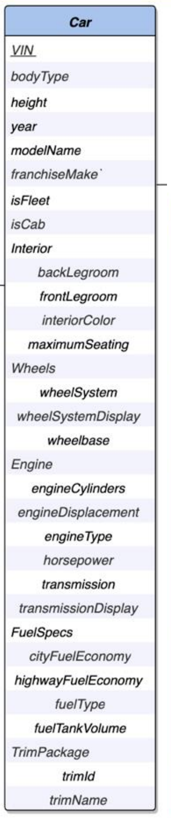
## Entity Relationship Design

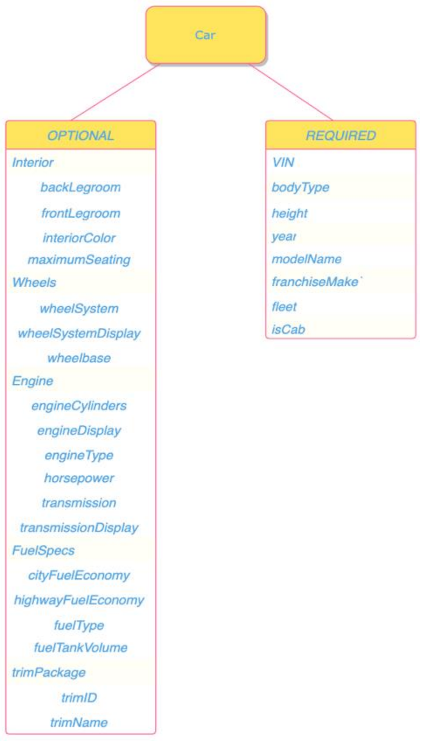
- We will refer to the entity relationship design/diagram/model in this part of the report as ERD for simplicity

ERD-Draft.pdf is an amalgamation of the alternative design choices we considered before creating our final entity relationship model, ERD-Final.

This section will aim to explain our choice of entity sets, multivalued, compound, optional attributes, aggregations, specializations while understanding why certain alternatives were finalized over others. Snippets of the draft and the final design will be compared to provide a holistic view of the design decisions made to build a strong ER-Model that can be converted to a Relational Schema. We will also discuss the cardinality constraints among different entities which will prove to be fundamental in this process.

1. Car
    - Car is an **entity set** that contains a set of specific cars that exist and are distinguishable from other such **entities** but share the same properties.



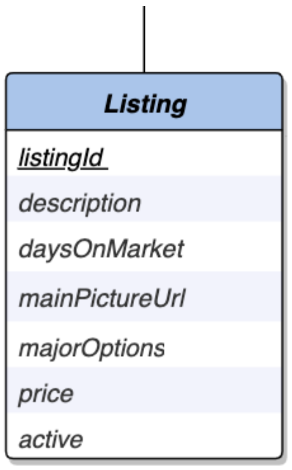


- The attributes that represent this entity can be divided into two categories: **required** attributes possessed by every single entity in our entity set and "**optional**" attributes that are possessed by some entities in our entity set. When we say Car **exists**, we mean it exists in Plato's world. Yes, cars in real world certainly need an engine to run but that doesn't mean every dealer using our database has the specs for every engine specification hence we leave it as an "optional" attribute that when added describes the entity.

- Here we can notice that all the required attributes are **single-valued simple** whereas all the optional attributes appear to be **single-valued composite**.

- **Entity Sets vs. Attributes**: We opted for bodyType, Interior etc. to be attributes rather than entities because the items under consideration are best described as an adjective rather than a noun

    - **Keys**: Cars has only one candidate key, (VIN) since none of the other attributes collectively or otherwise uniquely determine a specific car. Hence, we select our only candidate key to be our primary key.
2. Listing

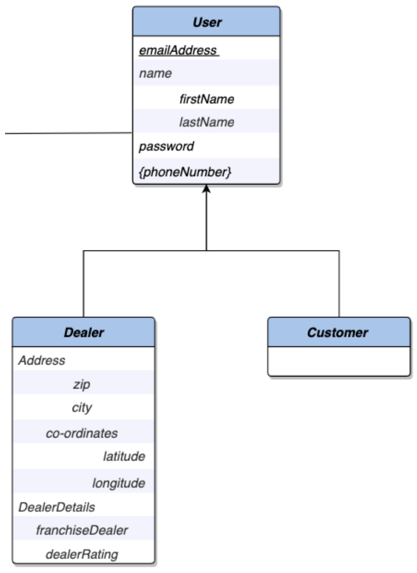


- Listing is an **entity set** that contains a set of specific listings in our system that are distinguishable from each other but share common descriptive properties listed as **attributes**.

- All the attributes are **required**, and we can see that they are **single-valued simple** attributes. We have opted to make them attributes instead of entity sets since they don't require more expressiveness allowing the representation of additional information about the say, price. Moreover, we only need one price listed under any listing at any time and don't see it changing in the future.

- **Keys**: As described in our project scope, we have not placed restrictions on dealers to under details into the description section that are unique in the whole database. If it were, multiple dealers could not have the freedom to describe their car in the same way although it is highly unlikely that they would. Since we haven't placed such constraints on any other attributes the only logical choice for a candidate key is (listingId) that becomes our primary key.

3. User (higher-level entity) and Dealer, Customer (lower-level entity)



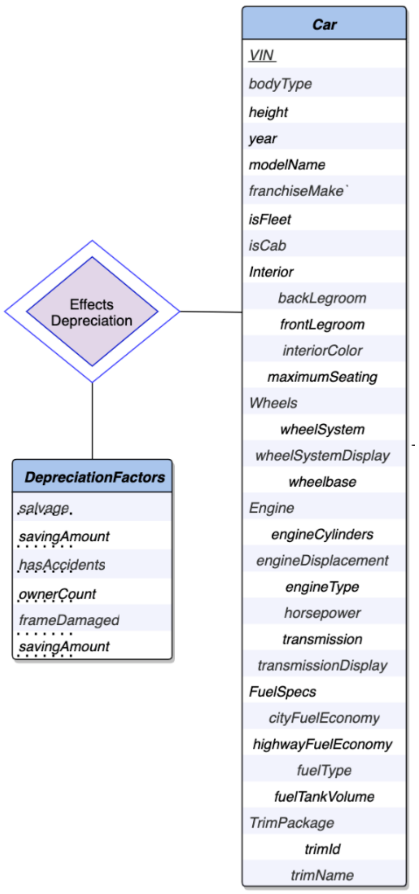
- **Partial Generalization**: We have decided that in our system, a User can be both a Dealer and a Customer. Our database will likely be used by more applications in the future and would require more roles like Data Science Analysts, Advertising Analysts etc., that have their own descriptive attributes not shared by others. Therefore, we felt like the specialization was a good strategy compared to just roles and attributes shared by every single user and with NULLs in case they don't exist. To add, although we say that a User can be both a Customer and Dealer, it is very likely that most Users who join as Customers might never sell their Car.

- **Disjoint Generalization**: For a specific instance, a User cannot be both a Dealer and a Customer. For example, for a specific listing relating to a specific car, a dealer cannot both create the listing and buy the car themselves. This is represented in the ER Model through combined arrows from lower-level entities to higher-lever entity

- **User Attributes**: password and email are simple single-valued attributes, name is a composite single valued attribute and phoneNumber is a simple multivalued attribute.

- **Keys**: Our system allows multiple Users to have the same name but have a single emailAddress. Since we are not using the whole email address along with the domain it is reasonable to expect that they would be **unique** by design. For example, if we send an email to only your uwaterloo email address then we can safely assume that you are the only intended recipient of that email. This means that in our system, a User can store multiple phoneNumbers with a single emailAddress. That logically makes emailAddress our **primary key** due to lack of other more favorable choices.

- **Dealer Attributes**: Address is a composite attribute with a mixture of single-valued attributes like zip, city, and composite attributes within that like co-ordinates. We have decided to make Address optional since all the dealers might not be comfortable with sharing such details due to primary concerns. DealerDealers is a required composite attribute that has a franchiseDealer flag style attribute that tells us whether are using our platform on behalf of a franchise and dealerRating is assigned to each use based on customer feedback among other factors to be describes in the Client Interface section.



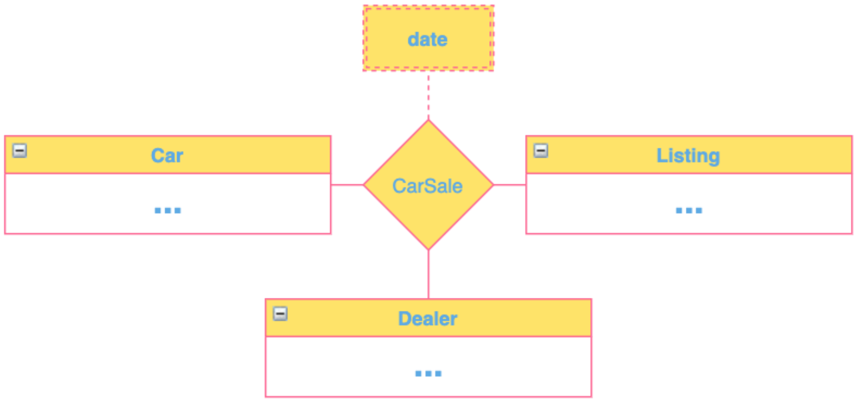
4. DepreciationFactors Entity and EffectsDepreciation relationship set

- **Weak Entity and Identifying Entity**: The existence of DepreciationFactors that EffectsDepreciation of the Car entity has no meaning. The former is predicated on the existence of the latter.

- **Primary Key**: We know that the primary key of a weak entity set is the combination of the primary key of the entity upon which it is dependent. Attributes underlined by the dotted line are the discriminators of our weak entity set and combine with VIN.

- **Cardinality**: Due to compelling arguments towards using better cardinality over traditional cardinality, we'll be using x:y notation – **Car** has a 0:1 cardinality, it can be in the relation one time and **DepreciationFactors** has 0:1 cardinality, you can have one set of factors effecting a car's valuation or none, can't have multiple values for the same factors. The primary key of DepreciationFactors is not just the set of discriminators, it includes the VIN, which means it can be in the relationship only once.

5. CarSale Relationship Set
    - Car, Listing, User entities are in a **ternary relation** with CarSale as the relationship set. 'date' is a **descriptive attribute** linked to this relation; it refers to the date a Listing was made by User for the sale of a Car. As desired 'date' is a simple single-valued attribute



Cardinality: Using the better cardinality-

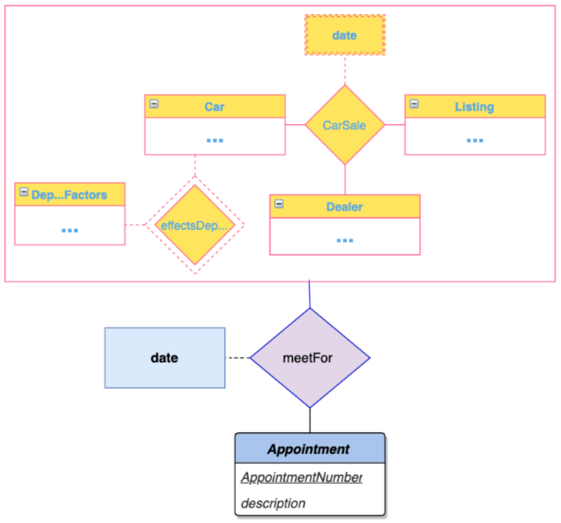
- Listing:Dealer is (1:1); Dealer:Listing is (1:*) because for a dealer to be in the system they must create a Listing but may have more than one and a Listing can have exactly one Dealer based on our application scope

- Car:Listing is (1:1); Listing:Car is (1:1) i.e., compulsory participation is enforced on both entities and they are exclusively and uniquely mapped to each other

- Dealer:Car is (1:*); Car:Dealer(1:1) because a Dealer must attempt a car sale to be in the relation but can sell as many Cars as they want, while a Car can only be sold by a single Dealer

- Instead of creating multiple binary relations, we have opted here to use the more effective and information preserving ternary relation because it reflects the actual relationship that these entities share and enforces that they are all connected.

6. Aggregation : meetFor relationship set, Appointment entity set and aggregate entity



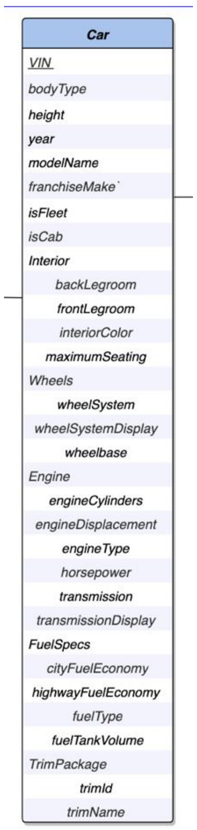
- **Motivation**: Relationship sets meetFor and CarSale represent overlapping information. Every meetFor relationship ties together a Car, Listing, Dealer just like CarSale but we cannot remove CarSale since it can exist without any meetings. In addition, we have some extra attributes like description related to the 'meet' called Appointment here

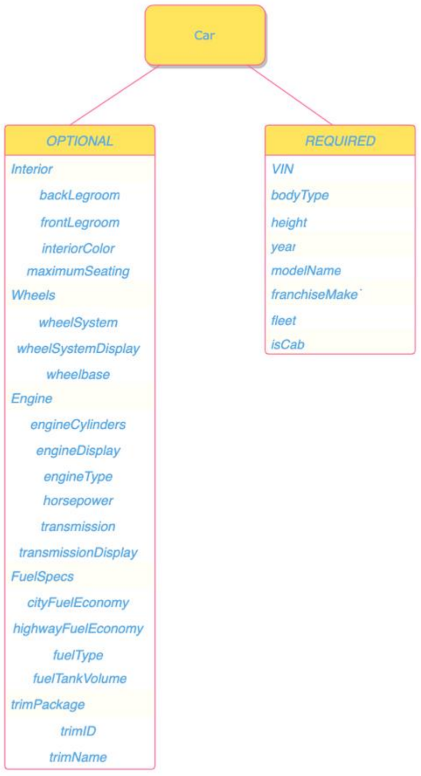
- Our aggregate entity is within the pink rectangle with our CarSale relation and the whole thing is connected to our meetFor relation connecting Appointment entity (with 'date' as descriptive attribute) to the aggregate entity

## Relational Model Design

- Refer to the createDb.sql file in the Database folder of our git repository for the SQL code that creates tables and views based on the final relational schema; primary, foreign, and unique key, checks and indexes to maintain the integrity and improve performance and load data from the csv files
- Some relational schema conversions are obvious, so we are going to provide our reasoning behind the motivations for only those that require explanations
- Integrity constraints like unique(), check(), set default, not null are used for several attributes that can be evident by the domain of each attribute and can be seen in our createDb.sql file

1. Car





1) Let's start with the easier step, we have the required attributes that the Car entity must have, so the relational schema with have all those attributes as part of the table, Cars.

2) Next, we have to deal with the optional attributes. Now our primary concern with the datasets on Kaggle was all the large number of NULLs and empty strings. By understanding the database and our applications usage we decided to create our entity and relational sets in such a way that the nulls are minimized, and they only have one connotation, missing information and not N/A. Every attribute to the left pertains to a Car. A huge factor in deciding which attributes would be optional was this as well.

3) We wanted to **reduce NULLs** as much as possible because:

i) Normalization concerns: These optional attributes cannot be part of the primary key and are not functionally dependent on it

ii) Simplify the client interface – We did not have to implement three-valued logic

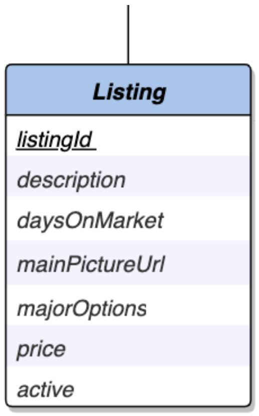
iii) RDBMS Treatment of NULL: From the lectures and personal experience of MySQL, having mitigatable NULLs in the system does not make for a well-maintained error-free application. We perform aggregations, arithmetic operations, and use

i) subqueries and having nulls just make things unnecessarily complicated considering we have other effective alternate solutions

4) Let's proceed creating **separate relations for our optional attributes** – we'll have the composite attribute name as our table name for clarity and since we don't have any naming conflicts. Our primary key for these tables will be (VIN) which also acts as a foreign key referencing Cars. This gives us Cars(<u>VIN,</u> bodyType, height, year, modelName, franchiseMake, isFleet, isCab), Interior(<u>VIN,</u> backLegroom, frontLegroom, interiorColor, maximumSeating), Wheels(<u>VIN,</u> wheelSystem, wheelSystemDisplay, wheelbase), Engine(<u>VIN</u>, engineCylinders, engineDisplay, engineType, horsepower, transmission, transmissionDisplay), FuelSpecs(<u>VIN</u>, cityFuelEconomy, highwayFuelEconomy, fuelType, fuelTankVolume), TrimPackage(VIN, trimID, trimName).

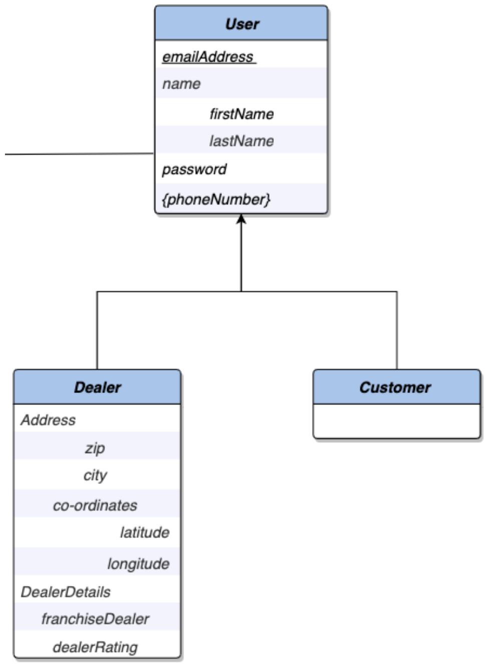
5) **Note** that we changed 'fleet' to 'isFleet' for clarity and all the attributes that start with is<attr name> must be either 'True' or 'False'. A check is added to ensure that.

## 2) Listing



- Listing (<u>listingId</u>, listingDate, dayOnMarket, mainPictureUrl, majorOptions, price, description). The primary key of the relation is same as the entity set.

- **Active** is a flag that indicates when the last time was there was any activity on the Listing. If it exceeds 180 days, then we delete the listing. This feature needs to be finessed to include intermediary email notifications informing the user that the listing has been inactive for 80 days and if they don't respond to the email (either by going to their listing or taking some action on the communication email, the listing would be deleted). When we say delete, we are speaking from the perspective of the dealer. It will be hidden from them but the system administrators, data science analysts will be able to see all the listings that were ever created. The cleanup of these listings is left to the discretion of those users.



3) User – Partial Disjoint Generalization

- User(<u>emailAddress,</u> firstName, lastName, password). Attributes within the composite attribute, name are now part of the User relation.

- '**phoneNumber**' is a multi-valued attribute and is optional so we create another relation called PhoneNumber(<u>emailAddress, phoneNumber</u>) since each user can have none or many numbers of phone numbers stored in our database. emailAddress is a foreign key referencing emailAddress in User.

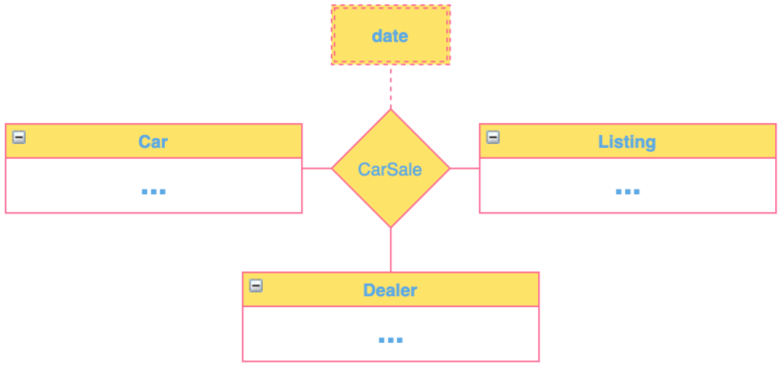
- Using a **Top-Down approach** and considering that Address is optional as well, our Dealer entity will be called DealerDetails for clarity and DealerDetails(emailAddress, franchiseDealer, sellerRating) with emailAddress as primary key and foreign key referencing emailAddress in User.

- Our **Address** will have Address(emailAddress, zip, city, latitude, longitude) with emailAddress as primary key and foreign key referencing emailAddress in User. But this is not a normalized relation. Now we need to normalize this:
- R = {emailAddress, zip, city, latitude, longitude}
- F = {emailAddress -> zip, city, latitude, longitude; latitude, longitude -> zip (because of accuracy of our co-ordinates it does not give city) }

- Keeping in mind that we are not restricting multiple users from using the same address and most users cannot be reasonably expected to provide us their co-ordinates, here is the relations we finalized on.
- **Address**(emailAddress, city, zip), **Coordinates**(emailAddress, latitude, longitude) with emailAddress as primary key and foreign key referencing emailAddress in User

4) DepreciationFactors – Weak Entity
   - DepreciationFactors(VIN, frameDamaged, hasAccidents, salvage, savingsAmount) is the final relation. As expected, VIN is the primary key and foreign key referencing VIN in Car because of the nature of weak entities and the cardinality constraints enabling us to identify each depreciation factor with just the associated VIN of the corresponding Car.
   - The weak entity previously had isNew attribute, but the relational schema does not due to a dependency. Whenever isNew is false, ownercount is not application. Adhering to our previous commitment of good database design, we have a separate relation called CarOwners(VIN, ownercount) with VIN as the primary key and foreign key referencing VIN in Car.

5) CarSale Relationship Set –
   - We have a couple of alternatives to establish this relationship:



a. CarSale(emailAddress, VIN, listingId, date)
b. CarSale(VIN, listingId, emailAddress, date)
c. Cars(VIN, listingId, …), Listing(listingId, dealerEmail, listingDate, …)
d. Cars(VIN, listingId), CarSale(listingId, dealerEmail, date)

Keeping in mind the cardinality constraints discussed earlier, we felt that option (c) is the most efficient alternative because both the Listing and Dealer have a 1:1 relationship, CarSale serves no special purpose if it's descriptive attribute gets absorbed and into the Listing entity.

The dealerEmail in Listing is a foreign key referencing emailAddress in User and listingId in foreign key referencing listingId in Listing.

6) Aggregation –

- Similar to CarSale relation, we don't necessarily need a separate meetFor relation if get it's absorbed into Appointment with the right primary keys to satisfy the cardinality constraints. Therefore, our Appointment(<u>appointmentNumber, dealerEmail, customerEmail, listingId</u>, appointmentDateTime, information). listingId is foreign key referencing listingId in Listing table, dealerEmail and customerEmail are foreign keys referencing emailAddress in User table. listingId is also added to primary key since a dealer and customer pair and meet with regards to multiple car deals and the appointment number is not sufficient enough to make that unique since the appointment number is a suite of numbers that are incremented in the subset of a particular set of dealer, customer, listing

- Similar to Listing entity set, there is an active attribute in appointments. The inactive Appointments and inactive Listings have an expiry date of 60 days and 180 days respectively after which they'll be hidden from the Dealers and Customers. The clock for a listing or appointment's expiry begins on the last day there was any activity on the listing.

## Physical Schema Indexes

We have already discussed the primary and foreign key attributes in the relational schema that act as indexes. In this section, we will look at some other indexes added to improve the execution time of our queries to improve the applications performance as the number of records increase.

Our application is defined to use several filters allowing the users to find cars that suite their needs instead of browsing through the 3 million listings in our database. As we have seen in the lectures, one way to increase the efficiency of the record lookup is by using indexes. These filters by definition don't have to scan the whole tables since they need to return only a subset of the records that match the key. You can see these indexes on our Cars table over maximumSeating, color; DepreciationFactors table over savingsAmount. Some strategically added indexes were able to decrease the execution time on queries that need to perform a lot of joins.

In some many tables, we were able to implement covering indexes, for example, counting the number of franchise dealers with sellerRating as 5. To decrease the size of joint attributes in the dataset and increase the efficiency of lookups, we also split the fuel economy into cityFuelEconomy and highwayFuelEconomy. We saw an improvement in the range of 40-65% after adding these indexes to our database with just primary and foreign keys.

## Changes to the database to support the next phase of the application

- Adding functionality for supporting multiple dealers to a single Listing and a single Appointment

- Right now, we are storing the password for the application in the same table as other User specific attributes without adopting any encryption techniques. If a hacker tries to extract from our database, they will own all the passwords in plain text despite the user having a strong password. This is not a secure methodology for storing passwords since the database administrators and website managers having access to them is a violation of privacy.

    - We will adopt the main principles of secure storage like hash, salt, pepper and iteration and use a reliable hash function for storage.

    - We will use a non-reversible cryptographic function so that the attacker cannot just guess the logic of the encryption and reverse it. While designing this function we will try to protect our passwords against brute force, dictionary, rainbow table attacks.

- Checking the correctness of VIN : Prior to 1981, the length of the VIN ranged from 11 to 17 characters but after that they are strictly 17 characters. As depicted in the picture [1], each segment of the VIN can be checked for correctness with the right database support. We can also split this number into different segments to reduce the length of the attribute which would make the lookup more effective and allow us to use char and decimal types.

    - We also plan to implement LUHN Algorithm to check correctness

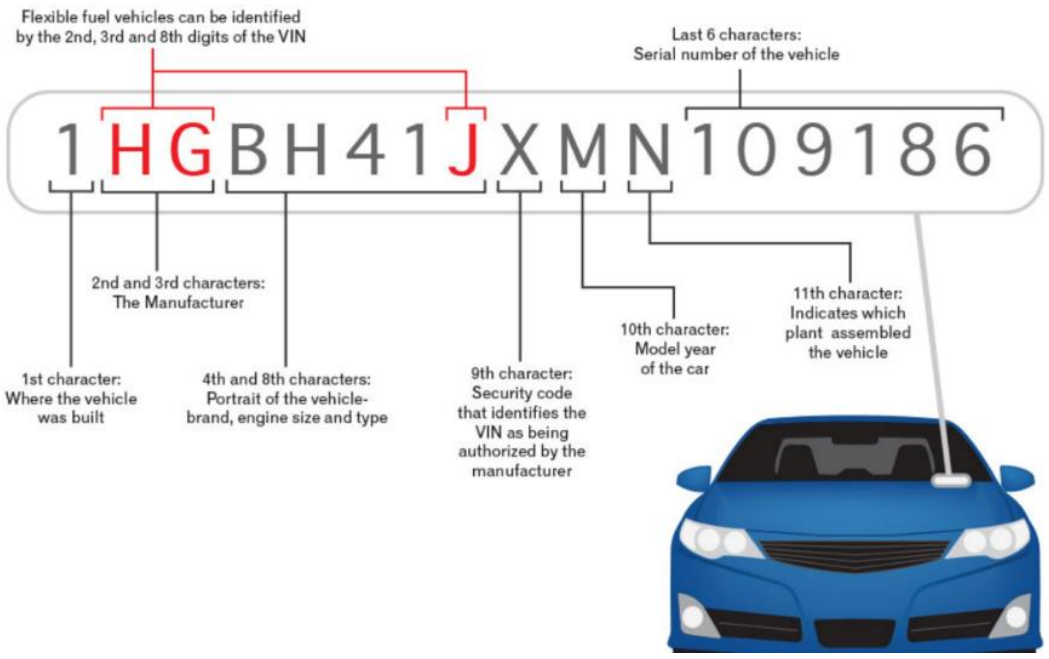

Flexible fuel vehicles can be identified by the 2nd, 3rd and 8th digits of the VIN
Last 6 characters: Serial number of the vehicle
1HGBH41JXMN109186
2nd and 3rd characters: The Manufacturer
1st character: Where the vehicle was built
4th and 8th characters: Portrait of the vehicle-brand, engine size and type
9th character: Security code that identifies the VIN as being authorized by the manufacturer
10th character: Model year of the car
11th character: Indicates which plant assembled the vehicle

*Figure 1: VIN decoding*

- Try to get the user coordinates by using a Geolocation API [2]. It takes care of user consent and allows us to adopt geotagging, tailoring advertisements on the application based on user location etc.

---

---

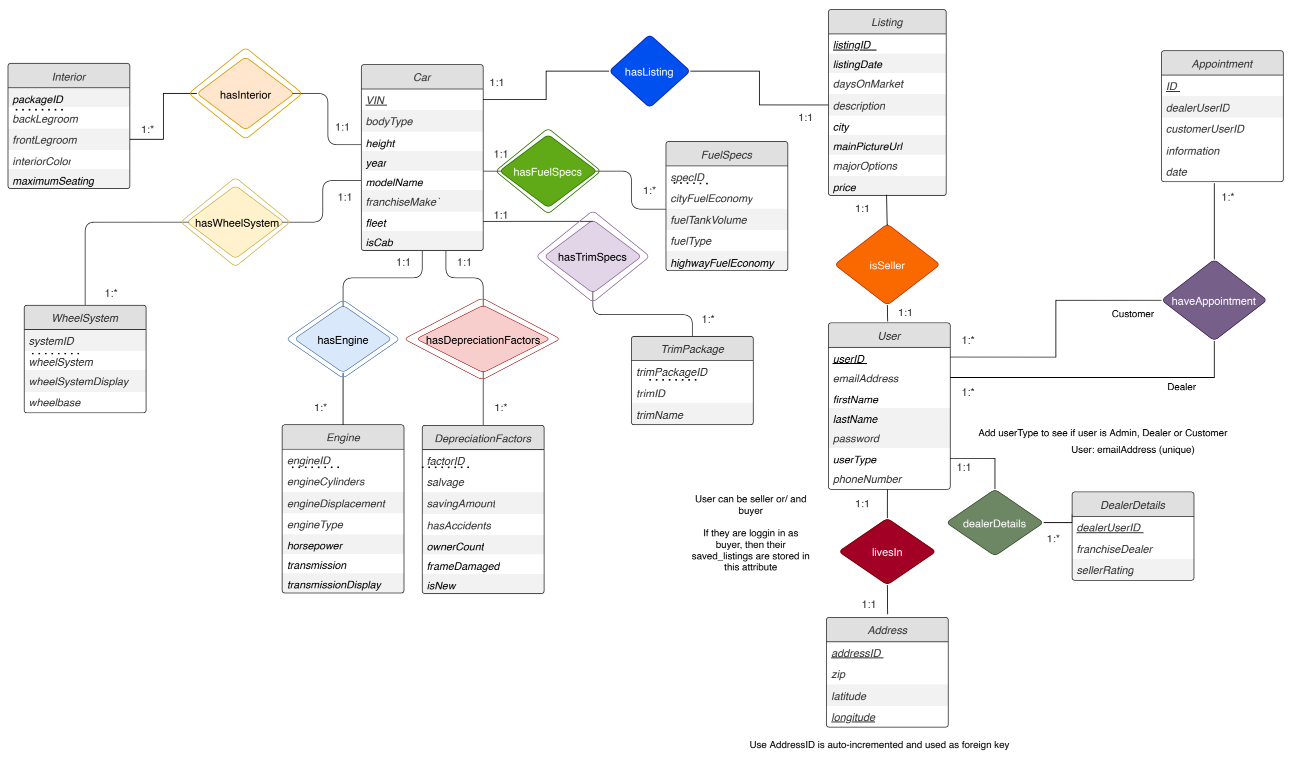
## Bibliography

[1] AutoCheck Experian, "AutoCheck Experian," 2019. [Online]. Available: https://www.autocheck.com/vehiclehistory/vin-basics.

[2] P. Kinlan, "Web Fundamentals," Google Developers, 12 February 2019. [Online]. Available: https://developers.google.com/web/fundamentals/native-hardware/user-location. [Accessed 15 April 2021].

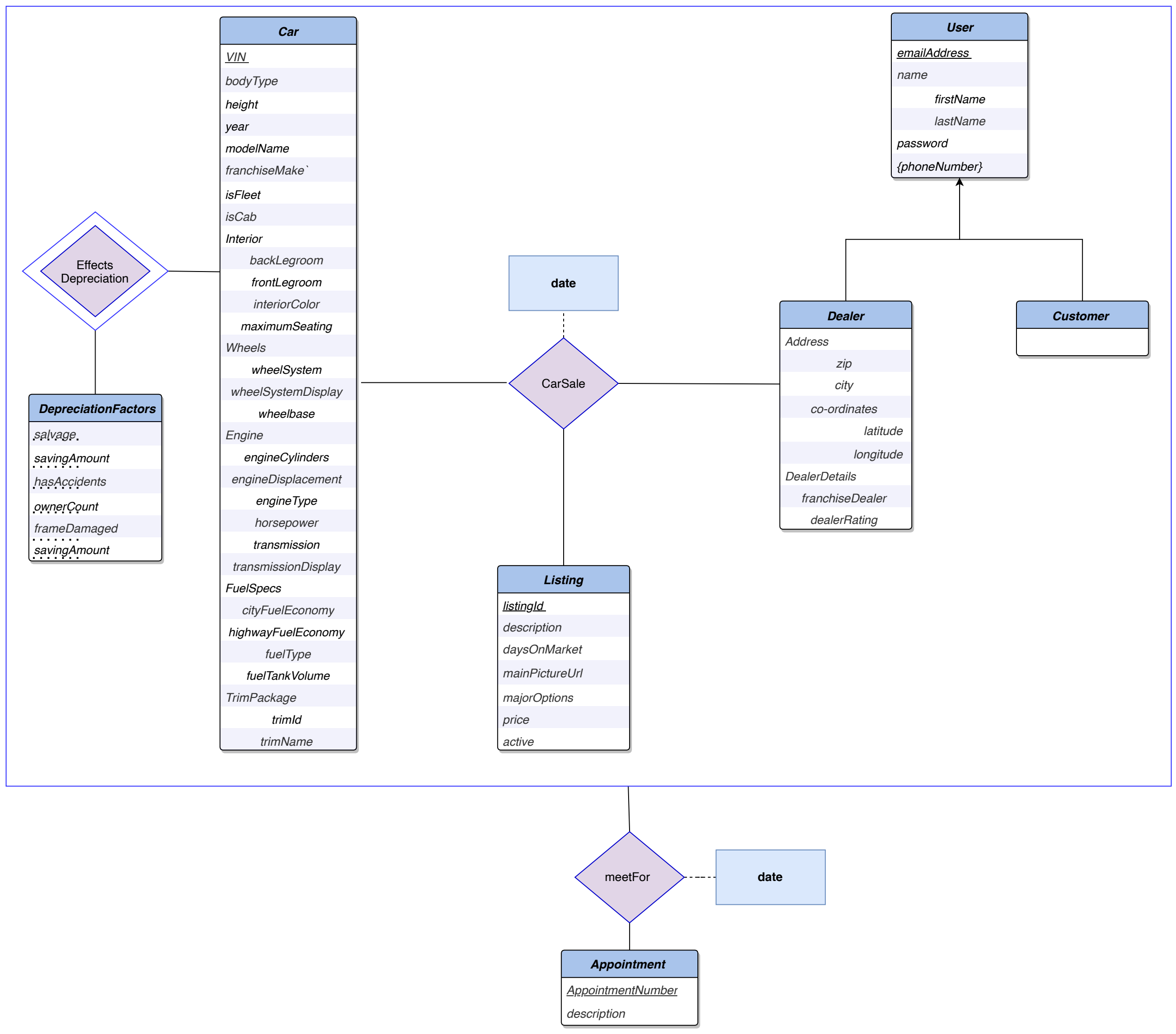
# Used Cars ERD

This is an amalgamation of the alternate design choices we made before finalizing our ER Model



## FINAL ENTITY RELATIONSHIP MODEL

cardinality constraints and motivations for choosing this model are explained in the report



# RELATIONAL TABLES

## Interior

- VIN
- backLegroom
- frontLegroom
- interiorColor
- maximumSeating

## WheelSystem

- VIN
- wheelSystem
- wheelSystemDisplay
- wheelbase

## Car

- VIN
- bodyType
- height
- year
- modelName
- franchiseMake
- isFleet
- isCab
- listingId

## Engine

- VIN
- engineCylinders
- engineDisplacement
- engineType
- horsepower
- transmission
- transmissionDisplay

## FuelSpecs

- VIN
- cityFuelEconomy
- fuelTankVolume
- fuelType
- highwayFuelEconomy

## DepreciationFactors

- VIN
- salvage
- savingAmount
- hasAccidents
- frameDamaged

## TrimPackage

- VIN
- trimId
- trimName

## Listing

- listingId
- listingDate
- daysOnMarket
- description
- mainPictureUrl
- majorOptions
- price
- dealerEmail
- activeListing

## User

- email
- firstName
- lastName
- pass
- userType

## Address

- listingId
- zip
- city

## PhoneNumber

- userEmail
- phoneNumber

## DealerDetails

- listingId
- franchiseDealer
- sellerRating

## Appointment

- appointmentNumber
- appointmentDateTime
- dealerEmail
- customerEmail
- information
- listingId
- active

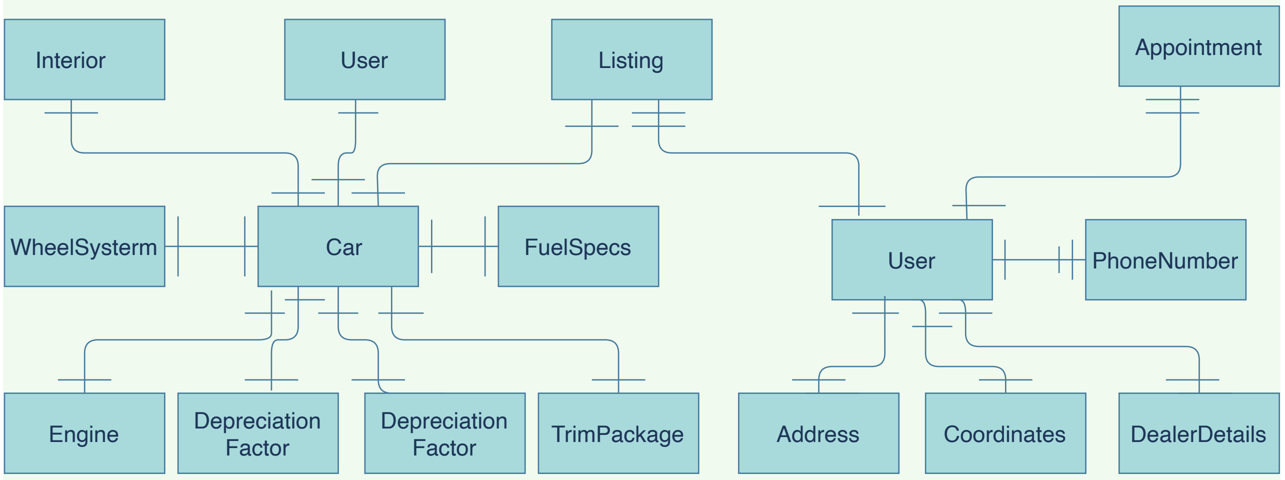
## CarOwners

- VIN
- ownercount

## Coordinates

- listingId
- latitude
- longitude

# Relational Schema



# CLIENT INTERFACE

Connor Barker

## Contents

Client Experience ........ 1

Technical Implementation ........ 4

Testing ........ 5

Login ........ 6

Search ........ 6

Appointments/Listings ........ 6

Future Implementation ........ 6

## Client Experience

The primary purpose of the application is to allow dealers to sell cars to customers, which collectively form the two categories of client user. Each is only able to perform actions related to their specific role; for example, a dealer cannot buy cars, and a user cannot sell them. There are also two types of administrator-level user, being admin and data scientist.

To distinguish the types of user, a login process determines who is using the application so that the views available to them can be limited accordingly. Users have unique identifying information in the form of emails and passwords to ensure that another user cannot fraudulently log in as someone they are not. If a user does not have an account, they can follow a signup process. In this process, they register as either a customer or a dealer, provide their personal information, and select a password. All information they input (email address etc.) is passed through a regex to ensure formatting is correct and invalid data isn't passed into the database.

The two administrator user types cannot be created through the CLI, but are instead manually inserted into the database via the MySQL CLI. An admin-type user is capable of viewing all components of the application visible to customers and dealers, and is equally capable of performing their respective tasks (listing a car, making appointments, modifying data). However, where users can only modify their own data, admins are capable of modifying all data in the database, as well as viewing listings and appointments that have been deleted and cancelled. Data scientists, on the other hand, only have access to the "Data Mining" section of the CLI. This view is only used by data scientist administrators, and allows them to view trends in the database. These trends can later be used to provide recommendations to customers to help them sell their cars faster, or to profit more from the sales.

After logging in, users land on the main view of the application, which will contain different features depending on the type of user. Both types of user are capable of viewing appointments and searching existing listings, and dealers are also able to list cars for sale and view their previous listings. While customers should obviously be able to search for vehicles, we have allowed dealers to do so as well on the grounds that they can price their cars more competitively by seeing what else is available on the market.

Users can access the search via the main view. Search results are able to be filtered, including searching for a car by make, model, year, whether it is new or not, and price. Additionally, a limited sorting functionality exists in the search view, as sorting by descending price borders on necessity for a used car sales application. Sorting is limited to price and year of manufacture to keep response time to a minimum.

Searches are performed from the search view, after setting desired filters and sorting criteria. Search results are truncated to the make, model, and price of a car, and display 10 at a time. The results are paginated, so the user can still view all listings in a database while maintaining readability. Customers can select a returned listing to view long-form details about it, as well as make an appointment with a dealer to purchase a listed vehicle. Multiple appointments can be made with the same dealer, by the same customer, regarding the same listing. This is intended to reflect the real-world process of purchasing a vehicle, which includes negotiations, test drives, and more than likely multiple appointments.

As mentioned previously, both types of client user can view appointments they are associated with. Their upcoming appointments are accessible from the main view. After displaying a paginated list of their appointments (10 per page), users can select them to view them in long form, edit the appointment time and notes, or cancel the appointment.

In addition to viewing appointments and searching active listings, a dealer can list a car for sale and view the listings they have previously posted in the application. When listing a car for sale, the dealer is walked through a series of prompts about the vehicle and its associated systems. Like the login process, regexes and other forms of type checking are used to ensure the format of the VIN and other information is correct. After filling out the relevant details, the listing is posted and is able to be found in a search by all other users.

A dealer can access their listings from the main view of the application. As before, the listings are paginated and displayed 10 at a time. By selecting a listing, a dealer can edit specific information about it (for example, adding engine information if they previously did not, changing details they had entered incorrectly, or altering the listing price) or remove the listing from the application. A hierarchical view of the CLI is shown below:

```
- Login/Signup
    - Main
        - Search
            - Add Filters
            - View & Remove Filters
            - Sort By
            - Display Results
                - Make Appointment [Customer Only]
        - View Appointments
            - Edit Appointment
            - Cancel Appointment
        - Create Listing [Dealer Only]
        - My Listings [Dealer Only]
            - Edit Listing
            - Remove Listing
```

When entering a new view, the user's current location is printed in breadcrumb format (e.g., entering the 'edit appointment' screen will print

```
main/view_appointments/edit_appointment
```

Navigation is typically accomplished by selecting one of a series of numbered options, typing 'b' to go back to the previous view, and typing 'q' to quit the application.

## Technical Implementation

The CLI is built in Python 3 and makes use of an Object-Relational Mapping library (referred to from now on as an ORM) for integration with the MySQL database where the application data is stored. An ORM is a tool which converts SQL relations into objects, as implied by the name. This is extremely helpful in object-oriented languages; SQL operates by storing scalar data in tables, while Python is an OOP language which uses objects like classes and structs to represent, store, and manipulate data. By bridging the gap between these two ordinarily incompatible type systems, ORMs allow programmers to more easily interface with databases and make use of the information they contain.

The ORM used by this application is called `sqlalchemy`. It works by having the programmer create a class for each table in the database using constructs native to the library. For example, a table is represented using a collection of `sqlalchemy.Column` objects. Each `Column` contains information like data type, whether the field is nullable, whether it is a primary key, and whether there are any foreign key constraints, mirroring the MySQL database.

The decision to use an ORM, rather than manually construct MySQL statements, was made for three main reasons:

1. Efficiency
2. Simplicity
3. Security

Firstly, the ORM is far more efficient than hard coding SQL statements, or even dynamically building them. It allows for easy and quick construction of MySQL statements by simply adding a `.where()` or `.join()` where needed, and makes updates and inserts similarly convenient. The quantity of code written as a result goes down immensely.

Secondly, while the ORM is extremely useful, we also acknowledge that this is a class meant to teach us SQL and the use of an ORM could be considered contrary to that goal. However, the queries performed in the CLI are all extremely simple. The most complex queries are still just a series of `WHERE` conditions, and considering the scope of the CLI, we felt our time was better spent on more difficult tasks. Additionally, while the syntax used by the ORM is different from pure SQL, the format is very analogous. Below are a series of SQL queries followed by their equivalent using the ORM:

| MySQL | Python ORM |
|---|---|
| `SELECT * FROM Car WHERE Car.year > 1990 LIMIT 20;` | `db.select(Car).where(Car.year > 1990).limit(20)` |
| `SELECT Car.VIN, Listing.price FROM Car JOIN Listing using (listingId) WHERE Car.franchiseMake = ‘Ford’;` | `db.select(Car.vin, Listing.price).join(Listing) .where(Car.franchiseMake == ‘Ford’)` |
| `UPDATE Appointment SET Appointment.dealerEmail = ‘psward@uwaterloo.ca’ WHERE Appointment.customerEmail = ‘psullivan@uwaterloo.ca’;` | `session.query(Appointment).filter(Appointment.customerEmail == ‘psullivan@uwaterloo.ca’).values({‘dealerEmail’:’psward@uwaterloo.ca’})` |
| `INSERT INTO Engine (VIN, engineDisplacement, engineCylinders) VALUES (90827452345, 3000.00, 6);` | `dbCar = Car(vin=90827452345, engineDisplacement=3000.00, engineCylinders=6) session.add(dbCar)` |

As you can see, the format of the queries is quite similar to the point that a good familiarity with SQL syntax is required to use the ORM. As a result, we made the decision that the use of an ORM was the best choice for the CLI, as the only work it makes easier is the repetitive task of building simple SQL queries. Finally, the ORM actually improves the security of the platform. By abstracting away, the raw SQL, we eliminate the risk of an injection attack, a common concern with SQL databases.

## Testing

To ensure that the CLI works as intended, it was subjected to a variety of tests. Initially, unit testing was performed on each discrete function to verify that the basic Python code was error-free, with dummy data being inserted when needed. For example, when testing the function which displays the long form data related to a listing, a predetermined listing ID was passed in rather than selecting one from the search page. After unit testing was accomplished, manual end-to-end tests were performed with a focus on the sections of code where the database was being modified. These are listed below.

### Login

The login/signup processes were repeatedly tested by creating multiple different dealer and customer dummy users. Specifically, a user cannot create an account using an email that is already associated with an account of any type, and a user must enter their password correctly.

### Search

The search, arguably the most important and database-focused element of the CLI, required a large amount of testing to ensure the filters and sorting functionality worked well. Each filter was tested on its own with a number of values, as well as in combination with others, to verify that multiple filters could be applied simultaneously while returning results correctly and quickly.

### Appointments/Listings

To test these functionalities, appointments and listings were repeatedly created, modified in different ways, and deleted. After each modification with the CLI, the database state was checked using the MySQL CLI, running in another Bash instance.

## Future Implementation

The biggest change that we would make to the application interface, given more time, is implementing it as a GUI rather than a CLI. While perfectly functional, CLIs are more difficult to navigate and display information with. Proper UX and UI would provide a more streamlined experience, and allow us to display more information in a more comprehensible way; for example, scrolling instead of pagination, or at the very least better-looking pagination.

# DATAMINING INVESTIGATION

SHWAPNEEL ISHRAQ

## Contents

The Question ........ 1
The Available Data ........ 1
Techniques of Data Mining ........ 1
Exploration of Data Mining Methods for Classification ........ 1
Decision Trees ........ 2
Code Flow ........ 2
Testing Validity ........ 2
Results ........ 2
Decision Tree Breakdown: Answering the Question ........ 3
Next Steps for Data Mining ........ 4

## The Question

Our application is one that allows for car dealerships to make a posting describing the price, condition, and physical characteristics of a vehicle. A potential customer can browse through these postings, find a car they like, book a meeting and buy that car they found through our application. Finally, our application logs how long each car takes from being listed to being sold. All these pieces allow for us to ask the question: what characteristics of cars lead to them being sold the fastest?

## The Available Data

When looking at the data being tracked we found no less than 66 different categories of data and about 3 million unique entries that we can use in our analysis. This is a substantial amount of data, but not all is needed. When we look at our question above and think from the customer's perspective about what is the most important information to consider we come to the conclusion that it is the most critical to include the price and the car's physical characteristics. In short: what am I buying and how much I am paying for it.

## Techniques of Data Mining

There are 3 types of data mining techniques: classification, association discovery, and clustering. When we look at the data we find many fields are not numerical in nature such as color or the model of the engine. Most of these fields can fall under the term categorical information. Categorical information is when the data falls into categories rather than be a specific number. For example, car color can fall into so many categories. Is the car: black, blue, red, white? This categorical data means we should use the classification technique where the algorithm looks though the categories an input belongs to and using it's previous experience, places the output in the appropriate category. For example, the algorithm has seen that blue cars sell well, so it puts the specific blue car we are inputting into the tree in the "sells well" category.

## Exploration of Data Mining Methods for Classification

We want to keep the conclusions to be to the point. We want to know what physical characteristics make a car sell fast or slow and then be able to target buying cars of those types to sell. So we will use a decision tree. We will take the daysOnMarket field in our database and classify it into 2 classes: sells fast, and sells slow. The algorithm will build a tree from an initial set of data. From there the tree will take physical characteristics of the car and predict if it belongs in the "sells fast" class or the "sells slow" class. We will use the median value for daysOnMarket as this boundary between selling fast or slow, which is 77 days.

## Decision Trees

For the project we will use sklearn’s decision tree library to construct our decision tree. We feed in 24 columns into the algorithm that constructs the decision tree, 23 of which are the price/physical characteristics of the car and the 24th is the daysOnMarket field that we put into the “sells fast” or “sells slow” classes. There are 2 important terms we need to know for decision trees: Gini index, and Cart algorithm. Gini is the measure on how well we have put the data into 1 specific category. For example, if we have 10 data values at a node and each of the two children receive 5 each then the Gini value is 0.5 for that node. If the node assigns all 10 data values to child 1 and none to child 2 then the Gini value is 0 meaning that the node properly classified the data and so that branch has done its job successfully. The lower the Gini index, the better for the project. The Cart algorithm is a greedy algorithm that seeks to split the tree in the best way in each step to best lower the Gini index. Note, the final classification always happens at a leaf node. This means that the best category to be able to see if something sells fast or slow are nodes that directly precede a leaf node.

## Code Flow

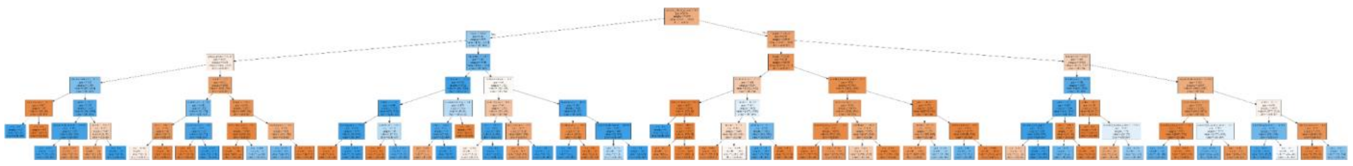
We have 6 steps to complete to get the tree for our data mining Investigation. First of all, we need to get the data from the database. Next is sklearn’s decision trees need every value to be a number, so we need to process the data. We use a class called label encoder that assigns each unique label a number so that all categorical data is numeric now and thus has the prefix encoded_ preceding its column name. We finish the processing step by removing any rows with nulls in it as the tree algorithm does not process nulls well. Finally we split our data in a training set and testing set, which we will cover in more detail in the ‘Testing Validity’ section. From there we use the train set to build the decision tree, and test its accuracy with the test set. The final step is that we print the decision tree as seen in the ‘Results’ section and do some analysis.

## Testing Validity

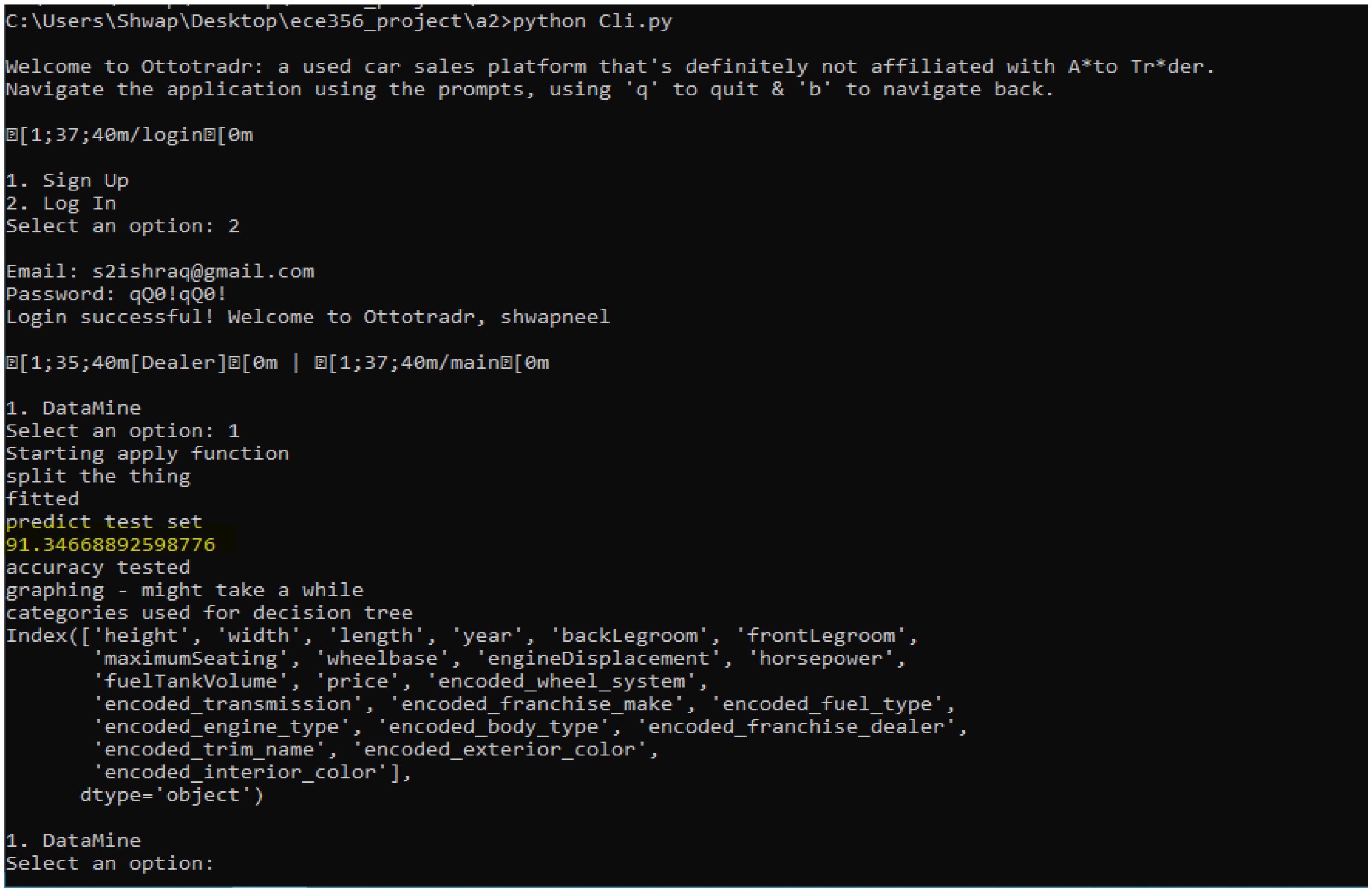
In the previous section we talked about testing. There are 2 parts to the decision tree, making it and testing it. We take all 3 million rows of car data and split 80% of it into the train set, the remaining 20% of data goes into the test set. We feed the train set into sklearn’s tree algorithm to make the tree. Then we feed only the 23 columns of price/car characteristics and generate an output set containing for each row of input data if the associated car will sell slow or sell fast. We then compare that generated set and compare it to the 24th column daysOnMarket classified set of the test set. The percentage in which they match is the accuracy score.

## Results

As seen below, here is the generated tree structure.
Full Table:



Note: In the github, in the ‘Decision Tree’ folder there is a ‘DaysOnMarketGraph.pdf’ file that has this tree diagram that you can access.It is highly recommended to view that file while reading this report as there is no good way to view a depth of 6 binary tree inside a document like this.



```
C:\Users\Shwap\Desktop\ece356_project\a2>python Cli.py

Welcome to Ottotradr: a used car sales platform that's definitely not affiliated with A*to Tr*der.
Navigate the application using the prompts, using 'q' to quit & 'b' to navigate back.

[1;37;40m/login[0m

1. Sign Up
2. Log In
Select an option: 2

Email: s2ishraq@gmail.com
Password: qQ0!qQ0!
Login successful! Welcome to Ottotradr, shwapneel

[1;35;40m[Dealer][0m | [1;37;40m/main[0m

1. DataMine
Select an option: 1
Starting apply function
split the thing
fitted
predict test set
91.34668892598776
accuracy tested
graphing - might take a while
categories used for decision tree
Index(['height', 'width', 'length', 'year', 'backLegroom', 'frontLegroom',
       'maximumSeating', 'wheelbase', 'engineDisplacement', 'horsepower',
       'fuelTankVolume', 'price', 'encoded_wheel_system',
       'encoded_transmission', 'encoded_franchise_make', 'encoded_fuel_type',
       'encoded_engine_type', 'encoded_body_type', 'encoded_franchise_dealer',
       'encoded_trim_name', 'encoded_exterior_color',
       'encoded_interior_color'],
      dtype='object')

1. DataMine
Select an option:
```

The generated graph above produced a 91% accuracy with the predicted set and test set. Meaning that out of test set, it was able to correctly guess 91% of the set to either sell fast or to sell slow.

## Decision Tree Breakdown: Answering the Question

Now we take a look at our tree, to keep the results readable and gain meaningful insight into the data. I have limited the tree to a depth of 6 as any higher makes a tree too large to meaningfully interpret for this report. Most of the Gini indices are below 0.25 and many of them are close to 0.0 which means many of the branches were able to properly classify the data into either the “sells fast” or “sells slow” class. In addition, our testing of the validity found the algorithm was 91% accurate in it’s guesses for the test set. This means that this tree is very effective in learning the patterns in our data.

Having established the tree is valid above we go to answer our question: what car characteristics will allow for our car to sell the fastest? We take a look at any nodes that precede a leaf node that has both a small Gini index and has the “sells fast” class as a child leaf node. For example, in the parent of one of the leaf nodes in the diagram (parent of the 7th leaf from the left) car year being greater than 2004 lands a “sells fast” child leaf node with Gini index of 0.0, which means that if we have a car that is newer than 2004 then it will sell fast. The following characteristics fit that criteria: Front Legroom, Year, Fuel Tank Volume, Wheel base, Maximum Seating, Trim Name, with Front Legroom,  Back Legroom, Year, and Fuel Tank Volume appearing the most before a leaf node that fits the Gini and sells fast criteria.

One last note: If we turn our attention from the bottom of the tree to the top 4 levels of the tree we see height, width, and length appear often, this suggests that while not having the most influential effect on whether a car sells fast or not, the size of the car can provide context on what Front Legroom, Back Legroom, Year, Fuel Tank Volume makes a car sell fastest.

## Next Steps for Data Mining

The next step for data mining (for this question) would be to implement a feature where if a user creates a listing, then input that listing into the decision tree to inform them if that car will sell fast on the market or not and provide them insights on how to make their listing attractive to increase the probability of a successful car sale.