**IE-406**

**MACHINE LEARNING**



**Project Report**

**TrainTripper: ML-Enhanced Predictive Ticket Cost Estimation and Smart Train Selection**

**Instructor: Dr. M.V.Joshi**

**Date: 07/12/2023**

[Abstract](#_kf57waqxi4yd)

[1. Introduction](#_3jglzxg7inuq)

[1.1 Need for better train trip planner for longer journeys](#_67oont25vq8)

[1.2 Proposed Approach](#_iqd5oir38ke0)

[2. Literature Review](#_8xigfhueyyph)

[3. Data Collection](#_k43p9e77mtcx)

[3.1 Scrapping train schedules](#_n380x88zuzu6)

[3.2 Scrapping prices and availability data](#_etwv8l1ubnj0)

[3.2.1 Scrap using stations](#_i0b5n3ixdf6v)

[3.2.2 Scrap using trains](#_w30ksn9735uq)

[4. Database Creation](#_7o99a3fu2q4j)

[4.1 Schema Design](#_vuvn61oi2yv1)

[4.2 Data Population](#_ltpzdtfumfsu)

[5. Search Algorithm](#_6cnp0ryzk372)

[5.1 Algorithm Overview](#_dv989nvshxhx)

[5.2 Performance Optimization](#_m4gr6e3h3ejw)

[6. Data Processing](#_2avjgfyqq7s8)

[6.1 Fares Analysis](#_j668jtxu699d)

[6.2 Data Preparation for Modeling](#_hiqlyma4kh8o)

[7. Modeling](#_xx32azqkjio9)

[8. Combined System](#_o2c5lb2755tq)

[8.1 Data Pipeline](#_3z3s47otl7bc)

[8.2 Trip Planner with Prices](#_5il928tqld53)

[9. Conclusion and Future Work](#_dzarzw33orxp)

[10. References](#_ua3ri9ual5p8)

# 

# 

# 

# 

# Abstract

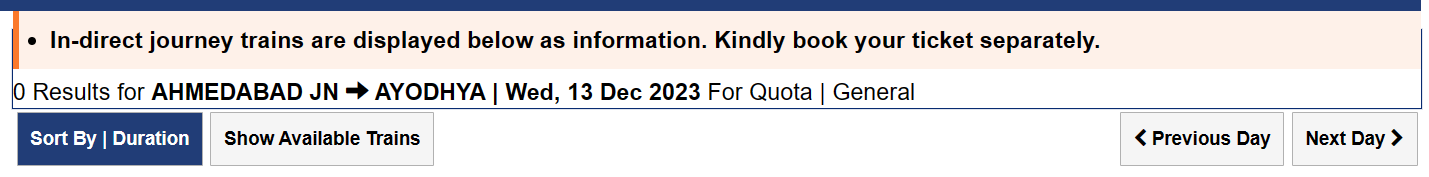
Finding and booking indirect train journeys in India can be challenging, especially for long trips or remote destinations. Existing platforms only offer direct trains, which may not be available or convenient for some travelers. This project aims to address this gap by developing a software application that can generate and compare hundreds of options for indirect trains, as well as direct ones. The application uses web scraping, search algorithms, and machine learning to collect, process, and estimate the data from the IRCTC website. The users can sort and filter the results by various criteria, such as estimated price and duration.

# 1. Introduction

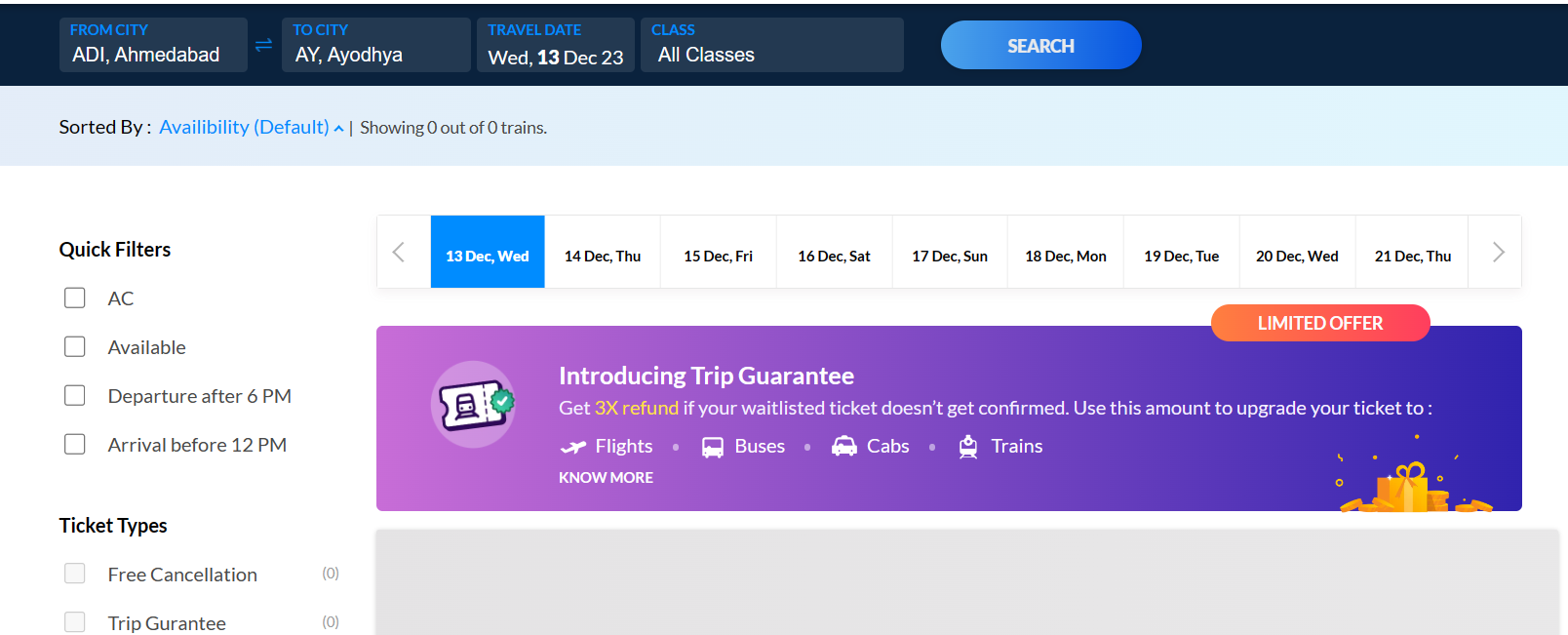
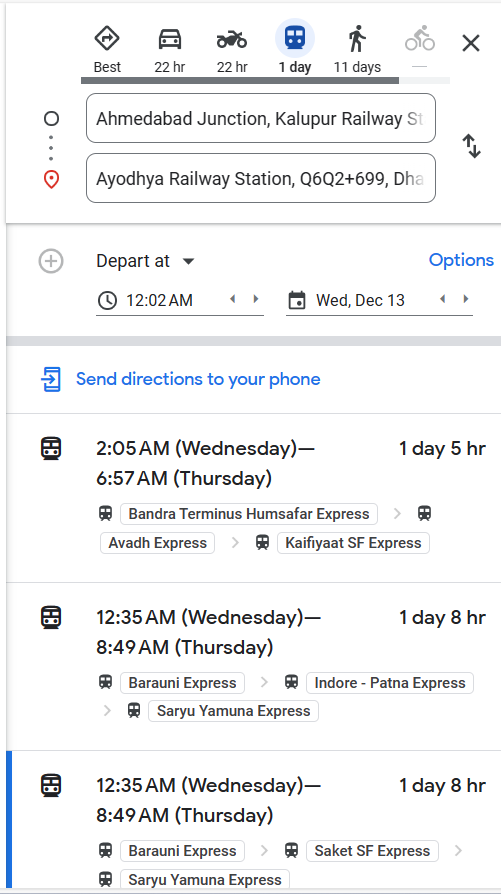
## 

## 1.1 Need for better train trip planner for longer journeys

The following is a screenshot taken from IRCTC after searching for trains from Ahmedabad to Ayodhya, departing three days after the search date. They gave a pop-up that no direct trains were available and, after asking to show indirect trains, this is what I got. Although we know that both stations are far apart, showing no trains and leaving it up to the user to search themselves is not a solution.

****

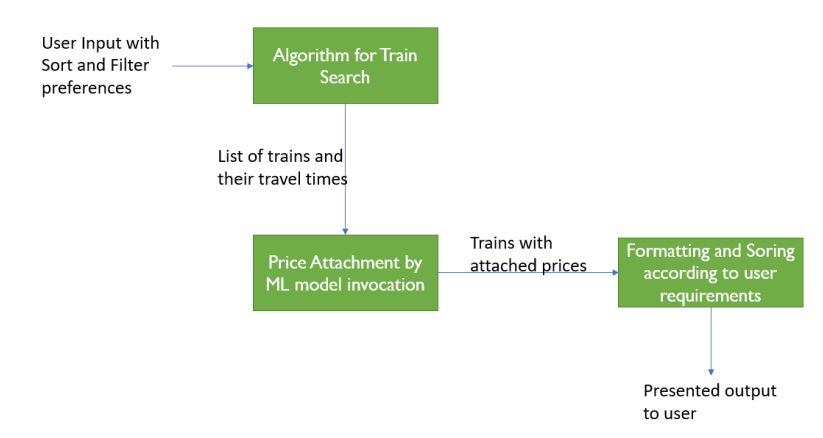
The following screenshots are taken for the same search on Google Maps and MakeMyTrip. Google Maps shows a few options, but they are not even close to optimal, as they include traveling in three trains. However, there are many good double train travel options available in the market. MakeMyTrip, the same as IRCTC, shows no options.



Thus, we see that there are no good agent websites to plan the best trip for longer journeys. Many people may need to book indirect train journeys between two less-known stations or stations that are very far apart, where direct train options are not appealing. Moreover, not everyone can always consider flight options because of high prices. Here we see the potential for a software that displays all the available booking options to the user, be it direct or indirect. Then, according to the user's needs, they can sort and filter by journey time, price and availability and pick up the best journey.

## 1.2 Proposed Approach

The approach we used to solve this problem is already present in the proposal document. We used the existing data collection platforms like Kaggle, Open Government Data (OGD) and other APIs that provide the government data like RapidAPI to collect the required data for our case. We used the data to separately create the search algorithm to find available train options and machine learning algorithm to predict the train prices. Following is the diagram that show the flow when the user inputs the query.

****

# 

# 2. Literature Review

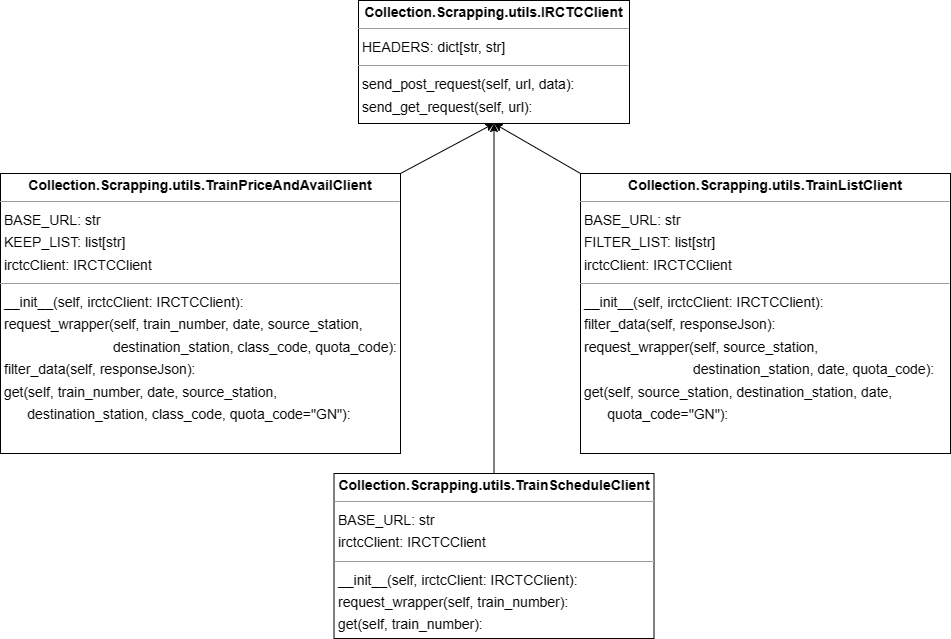
The authors of [1] studied the Spanish train market using a dataset of prices and demand. However, their method is not suitable for our problem, because we lack the data to apply it in our context. Another study [2] tries to estimate the likelihood of getting confirmed seats for waitlisted tickets, an important factor for train travel. However, the data they use is outdated, covering seven years, and we could not reach the authors on the topic of how they collected the data for their project. While waitlist prediction is a possible direction for our future work, we still need a suitable and recent dataset.

OGD (Open Government Data) [3] is a great resource for finding datasets that might be useful for different purposes. I was looking for some data on trains in India, and I found one dataset called train\_details that had 186000 records of train schedules, with about 8000 stations and 10000 trains. However, this dataset was outdated and had a lot of irrelevant data on goods trains which I was unable to filter. Rapid-api [4] is another option for getting data through public APIs, but it's very expensive and has a limit of 20 free requests. The API offers various queries, such as the price and availability of a train between two stations, the details of a train or a station, and so on. To get the most out of the free requests, I generated a list of the 20 most important stations in India. Then, I used the API to get the list of all the trains that start from, pass through, or end at each station. This way, I got 2548 unique train numbers. I could have gotten more by using another account, but I decided to stop there for now.

A github repository railway-master [5] which has schedules as the most important data json file. This gave 5248 unique trains but this is also 7 years old data and thus less believable that all the train codes are valid which is what I verified on IRCTC web page as well that some trains were not valid. Also the addition of many new Vande Bharat trains and other trains are obviously not reflected here. Though the idea of saving the data this way is taken from this repository, the exact data is not useful for our purpose.

# 3. Data Collection

We had a hard time finding any price data that already existed. We searched through various github repositories and public APIs, but most of them were useless. The only source for the current day data was rapid-api, but it was too expensive. So I decided to scrape the IRCTC website. I wrote some code that would mimic the HTTP requests that the IRCTC frontend sends to the backend. I could see these requests on my network tab of the web console whenever I clicked something on the website. It was a tedious and tricky process, but I managed to write several utility classes that I could use in my program and collect data. I wrote classes for getting trains between stations, getting train schedules, prices and availability. Now I can get real-time data for free with my program, instead of paying for rapid-api. I needed to write these classes because we wanted to collect different kinds of data from different URLs, and these three classes provide three endpoints for that.



## 3.1 Scrapping train schedules

To obtain the current train schedules, I could use the OGD data but I had to filter out the obsolete ones and find the new ones for trains such as Vande Bharat and Tejas Express which was not possible. I used the TrainScheduleClient class to query IRCTC for all the 5-digit train numbers, which are unique identifiers for each train. The task was tedious but I managed to speed it up by increasing the number of threads to 10. In about 2 hours, I collected around 3300 public train schedules, including the ones for the new trains. This data containing all the train schedules is enough to create an algorithm to determine the direct and indirect trains between a source and destination station for a given date.

## 3.2 Scrapping prices and availability data

To help users compare different train options, we need to show them the prices of each journey. We already know how to create an algorithm for finding trains, but we also need some data to link our trains output with their prices and if possible show the train availability as well. Why can't we just ask IRCTC to give us the prices every time the user searches for the indirect trains? There are two main problems with this approach:

1> For a given station pair, there could be thousands of possible indirect trains, and we would have to request IRCTC thousands of times to get the price data, which is not feasible.

2> If we rely on IRCTC for every user search, then our software is dependent on IRCTC and will not work if IRCTC is down. If we can pre-save or predict the prices accurately, then we don't need to depend on IRCTC.

These reasons show that we need to either precompute the prices data and store it for each station and train combination, or predict the prices in real-time instead of asking IRCTC for each query.

Now suppose we want to precompute the prices data and store it. There are 3766 stations where at least one train stops. There are 3292 trains in India. After some analysis, I found that there are more than 2 million unique combinations of train and station pairs. To collect this much data from IRCTC is time-consuming. Moreover, some trains have dynamic charges that depend on the difference between our query date and the train departure date. Therefore, the best solution is to collect enough data to build a model that can accurately predict the prices, which can be dynamic and change with dates.

### 3.2.1 Scrap using stations

The TrainListClient class allows us to query the trains between two stations, followed by use of TrainPriceAndAvailClient class to retrieve prices for all those trains between two stations for a specific date. We can use this method to collect data on how the prices vary for different train classes, such as 1A, 2A, 3A, SL, etc. However, this method is slow and inefficient for collecting data on many station pairs and distances. Therefore, we need a better method to capture the price variability based on distance.

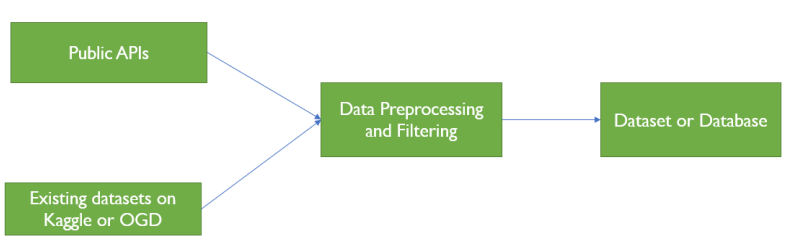
### 3.2.2 Scrap using trains

To collect the prices data, we use the schedules data that we have already obtained. We randomly select some station pairs where the train stops for each train. We also randomly choose some dates within four months when the train is available. Then, we collect the prices and availability for all the sitting classes that the train offers. This way, we can capture how the prices vary depending on the distance and the date.

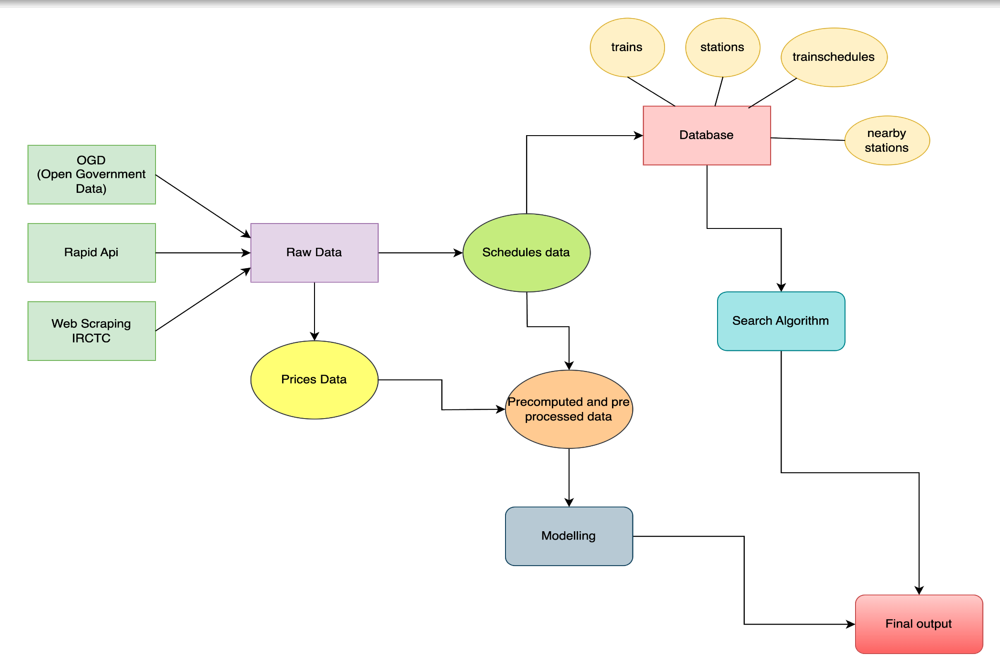
Using both these methods, we collected around 3 lakhs rows for prices and availability. This must be enough data for modeling the price and availability using the dependent variables like distance, duration, class code, etc.

# 4. Database Creation

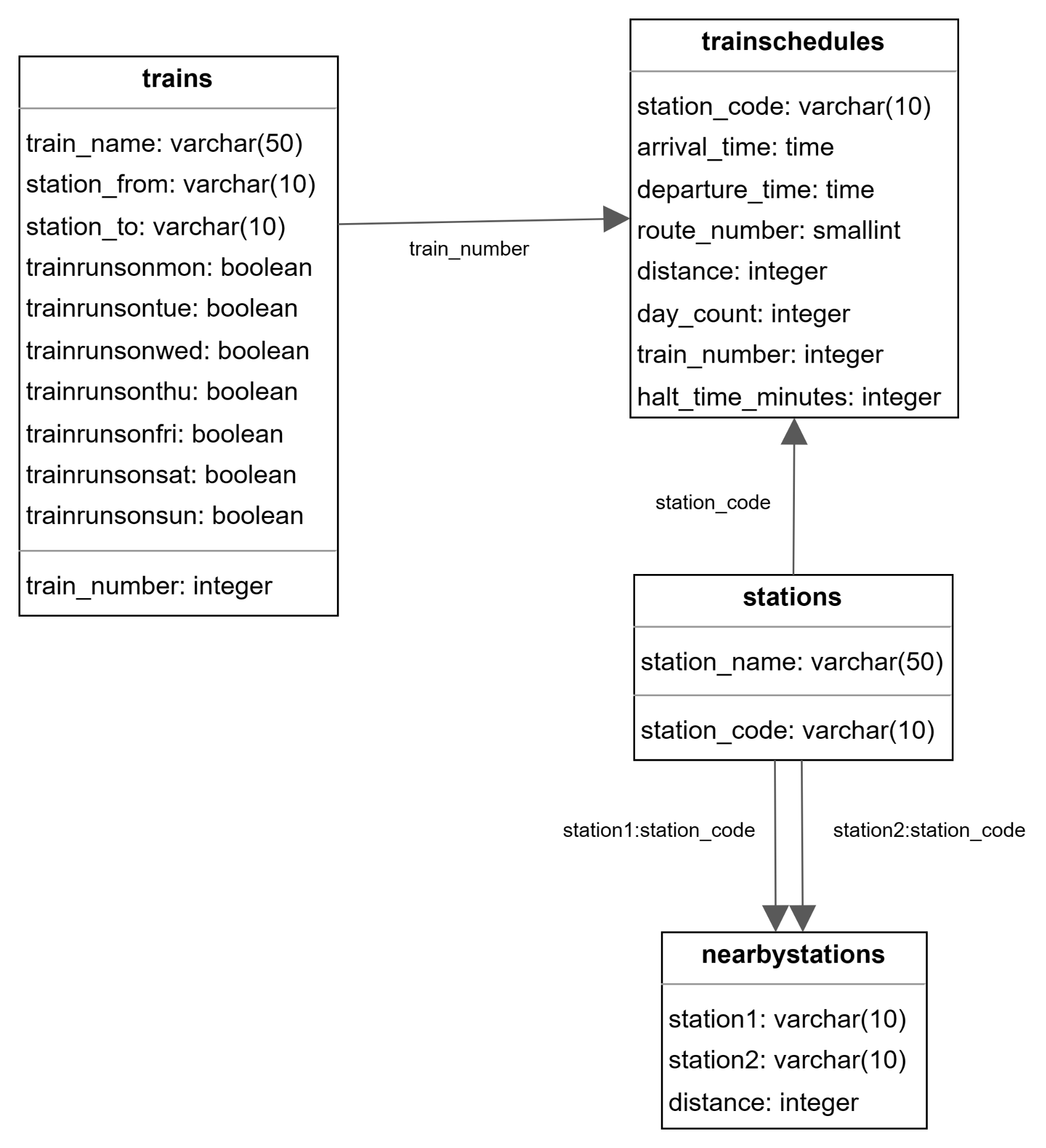
The schedule json file contains a list of trains with their details, such as number, name, departure days, and station list with halt times. This data is sufficient for writing a search algorithm to find the trains, but it is not easy to use in code. A relational database would be more convenient and flexible for this purpose and for future applications. Therefore, we need to convert the non-relational data to a relational format.

****

In the following diagram we can see that the database is only created using one schedules json file and this database for our case, is only used for creating the search algorithm.



## 4.1 Schema Design



The database contains a table called trains, which has 3292 rows with information about different trains. Each row has a station\_from and a station\_to column, which indicate the origin and destination stations of the train. However, this does not mean that the train returns to the same station\_from after reaching the station\_to. For instance, the Karnavati Express train operates between Ahmedabad and Mumbai stations, but it has a different train number for each direction, even though the train\_name is the same.

Another table in the database is trainschedules, which has multiple rows for each train number. Each row corresponds to a stop or halt of the train along its route. The station\_code column identifies the halting station, the day\_count column shows how many days have passed since the train departed from the station\_from, and the distance column shows how far the train has traveled from the station\_from. Note that the distance column is not always increasing, as it can decrease when the route\_number column changes. This happens for only 6 trains in the database.

The stations table is not used by the search algorithm at the moment, but it could be useful for other applications that need the coordinates of the stations. The coordinates are available in another data source.

The nearby\_stations table lists all the stations that are close to a given station1 as station2. For example, some of the nearby\_stations for station1=ADI (Ahmedabad junction) are Sabarmati junction and Maninagar railway station. The search algorithm will include trains that depart from these nearby\_stations as well, even if the user specifies ADI as the source station. This is similar to how IRCTC search works.

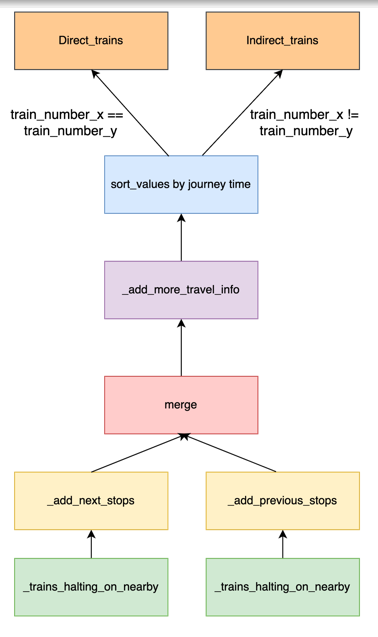
## 4.2 Data Population

The code for populating the database uses the schedules.json file as the source of data for all the tables. It also calculates the halt time in minutes and fills the nearby stations table with stations within 30kms. It converts the train numbers from string to integer for better indexing. The code does not have much logic beyond these tasks.

# 5. Search Algorithm

The Search Algorithm is the heart of our train recommendation system. It finds the best multi-train trips for the user. The TrainsFinder class runs the algorithm inside the multi\_train\_itineraries function. This function is the main way for the user to access the best journeys.

## 5.1 Algorithm Overview



1> \_trains\_halting\_nearby: Identifies nearby stations for a given station, selecting the one with the least distance if same train departs from multiple nearby stations. Filters trains halting at these nearby stations, considering the departure date if specified.

2> \_add\_next\_stops: Seeks potential halting stations for a two-train journey, generating a dataframe of departure-halting pairs by examining next stops.

3> \_add\_prev\_stops: Analogous to \_add\_next\_stops, computes potential halting-destination pairs.

4> merge: Merges departure-halting and halting-destination dataframes, yielding a comprehensive set of multi-train journey combinations.

5> \_add\_halt\_details: Computes arrival at halting stations, ensuring the next train departs within the next 48 hours. Calculates halt time in minutes using halt departure and arrival times.

6> \_filter\_best\_itinerary\_for\_train\_pair: Manages multiple potential halting stations for a train combination, retaining the one with the maximum halt time for added convenience.

7> \_filter\_by\_current\_date: Filters trains departing before the current date and time.

8> \_add\_more\_travel\_info: Computes reaching date and time, along with journey time in minutes.

9> sort: Orders combinations by journey time, prioritizing the least duration for user convenience.

10> separate: Distinguishes direct trains (where train-1 equals train-2) from multi-train journeys.

## 5.2 Performance Optimization

The algorithm's performance was enhanced from 4 seconds to 575ms, as shown by previous commits and the performance folder of the search algorithm. I replaced the pandas apply functions, which were iterative in nature and simpler to write, with vectorized operations for data manipulation. A function call graph is provided for better comprehension. The algorithm avoids data iteration and uses only pandas vectorization operations to handle complex data interactions. Insead of using the database connection, using preloaded tables as DataFrames also led to the performance improvement.

The algorithm matches the direct train results from IRCTC and adds a new feature of indirect trains that can offer high-quality journey times. For instance, there are 13-14 direct trains from Mumbai to Delhi on IRCTC, and my algorithm shows the same number plus more than 3000 indirect trains with one stop. These are the best 13 journey times for direct and indirect trains between Mumbai and Delhi: [932, 993, 1015, 1075, 1105, 1105, 1145, 1338, 1395, 1495, 1550, 1585, 1690] and [1003, 1007, 1052, 1065, 1072, 1078, 1080, 1082, 1082, 1095, 1107, 1108, 1120]. This feature is not available on IRCTC and other providers have poor search results. One can directly use this algorithm without any need for

# 6. Data Processing

Aim of this folder is to make the data ready for modeling so that we can also concatenate the estimated prices along with our train search. Assume the model as black box which has inputs as train number, from station, to station and class code and the output should be a good estimate of the price to book the ticket.

## 6.1 Fares Analysis

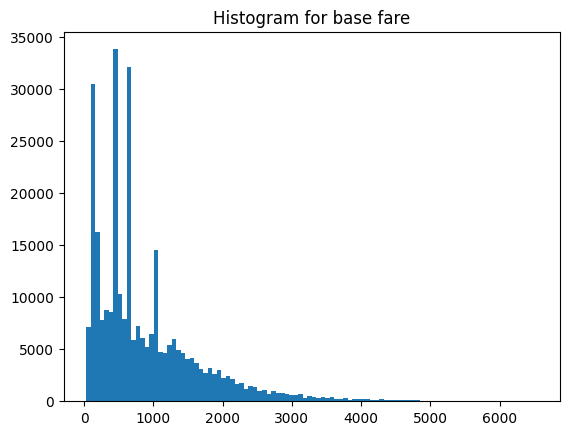
The JSON file with prices and availability data revealed various fare types and taxes, depending on factors such as stations, train speed, catering, quality, distance, duration, class, and quota.

Following are the key observations from my analysis.

Total Fare: Nearly the sum of all the other fares with a minor difference of one-two rupees in a few rows.

Irrelevant Fares: Fuel charge, concession, tatkal, and others.

Reservation Charge: Only class-dependent. For example 1A has the highest fare, followed by 2A and others. It is precomputed for model simplification and subtracted from the amount to be predicted.

Base Fare: Independent of common features like class code, distance and duration and thus retained in the fare to be predicted by model.

Superfast Charge: Fixed for a train-class combination. For a fixed train it could be different for varying class codes but it is fixed irrelevant of source and destination. Same as reservation charge, it is precomputed and subtracted from total fare for model simplification.

Catering Charge: Highly varying depending on the source and destination even for a fixed train and class code and the reason for it is that IRCTC offers catering on fixed timings like breakfast time, lunch time and dinner time. This charge is simplified to a binary column indicating if the train offers catering or not. Though we are losing some information by not considering the arrival and departure times, it makes the model simple.

Dynamic Fare: Present in premium trains like Vande Bharat or Rajdhani. This fare is again flagged as a binary column indicating if the train has a dynamic fare component or not.

## 6.2 Data Preparation for Modeling

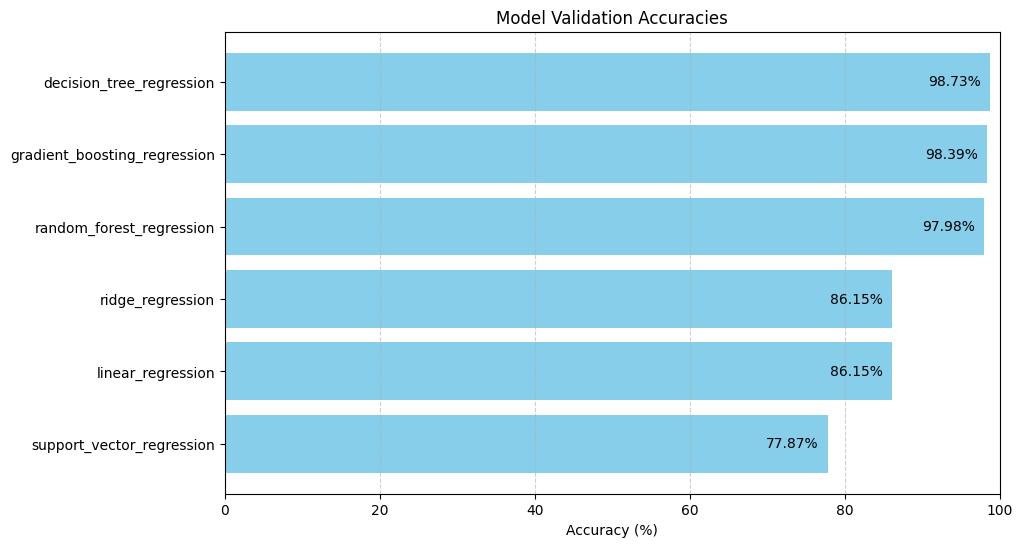
To calculate the distance and duration of a train journey, we need to use the from and to stations along with the train number. The distance\_map variable stores the cumulative distance and duration for each train and station pair as a dictionary. We use the schedules.json file to update the distance\_map variable as the train passes through different stations. We save this variable for later use in the model prediction.

The model also needs to know the class of the ticket, which is a categorical variable. We use one hot encoding to convert the class variable into numerical values that the model can understand. This way, we can use the class variable along with the distance and duration to predict the total fare.

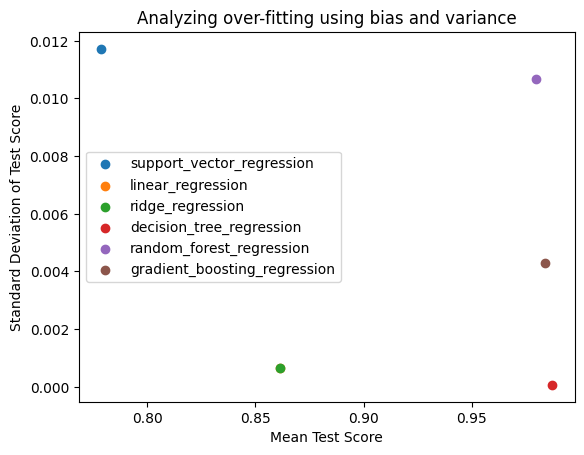
We also use the train number to get some additional information about the fare, such as the dynamic\_fare, superfast charge and catering information. These are fixed values that depend on the train type and service.

# 7. Modeling

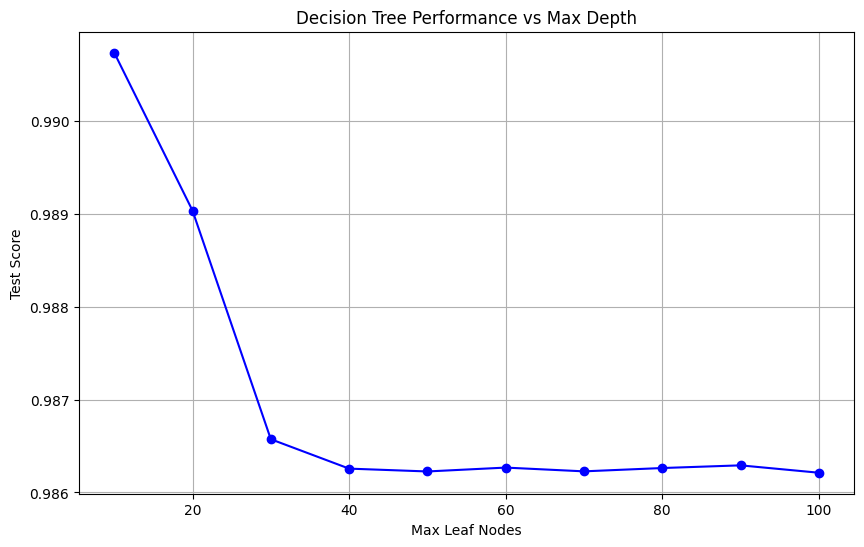
We experimented with various machine learning models for price prediction using a dataset of inputs and outputs. We applied LR, DTR, RFR, GBR, and SVR models. We tuned the hyperparameters of these models using the RandomizedSearchCV class of Sklearn. We set n\_iter=5 to randomly select 5 hyperparameter combinations and find the best one. We set cv=3 to perform 3-fold cross validation for all the models and iterations. We observed that DTR, GBR, and RF had better performance while the others had poor performance.

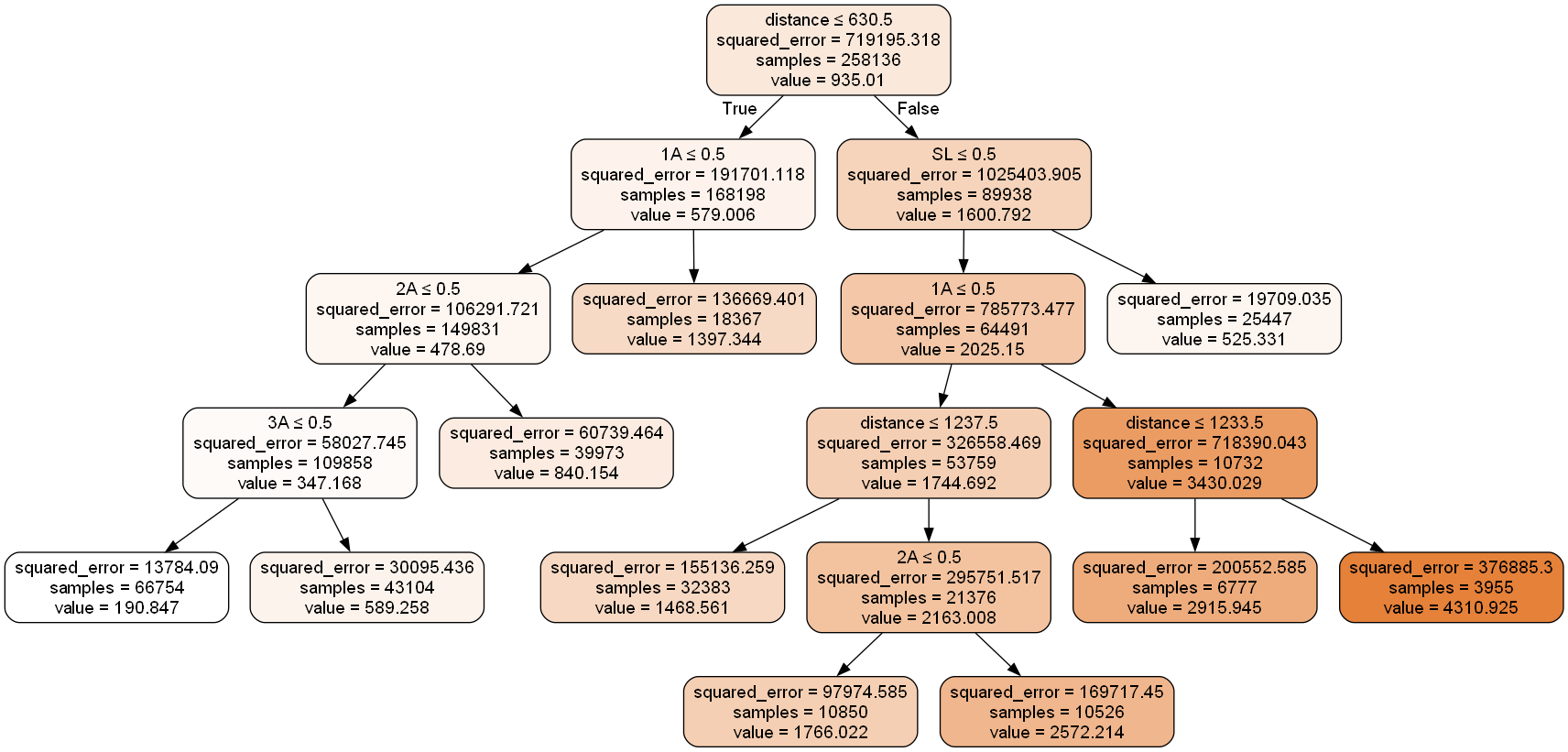


We performed a bias-variance trade-off analysis for the different models with their optimal hyperparameters. We obtained three accuracy values for each model, corresponding to the three folds of cross-validation. We calculated the mean and standard deviation of these values and plotted them as a scatter plot. The plot shows that Decision Tree Regression has the highest mean accuracy (lowest bias) and the lowest standard deviation (lowest variance) among all the models, indicating that it is the most suitable for generalizing unseen data. **The reason why DTR outperforms more complex models like GBR is that its hierarchical structure of decision rules matches the human decision-making process and our task is to predict the train prices which are determined by humans.**

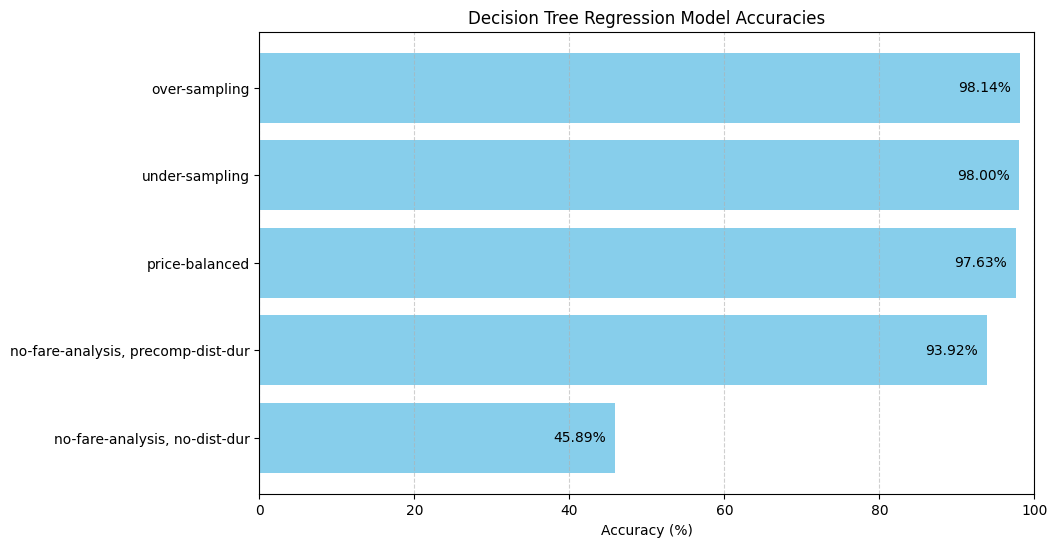


The hyperparameters max\_depth and max\_leaf\_nodes have the most influence on the decision tree regression model. The plots below show how the accuracy changes when one of these parameters is varied and the other is fixed. The accuracy goes down when the tree depth increases and goes up when the leaf nodes increase. Therefore, we choose max\_depth = 10 and max\_leaf\_nodes = 50 for further analysis.

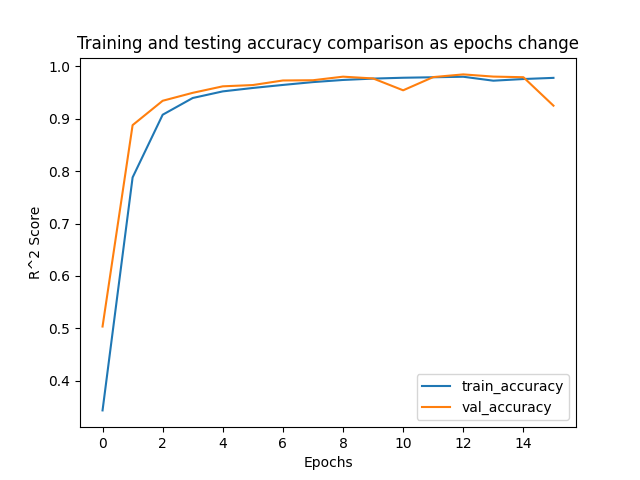


The tree diagram below shows how we can predict the price based on the max\_leaf\_nodes parameter set to 10. The most influential factor for the price is the distance, followed by the class code. The duration factor does not appear in the tree because it is highly correlated with the distance. Another observation is that the right side of the tree has a high mean square error, where the prices are very high. This is because we have limited data for the high prices, which are affected by dynamic and catering factors, as well as the availability of 1A, 2A classes in some trains.

To build the model, we experimented with different conditions on the data, such as balancing the class column by undersampling or oversampling with minimum and maximum number of data points per class, balancing the price data (importance of which is shown in the decision tree diagram above), and masking different columns to evaluate the importance of precomputed prices. We found that oversampling the class data gave the best performance on our data and saved the model.



We experimented with various artificial neural network models with different configurations of hidden layers, number of neurons, optimizers, activation functions and on the whole dataset, tuned the hyperparameters with 10 epochs. For dense neural networks, using cross validation, we discovered that the model with (hidden\_layers=2, neurons=16, optimizer=adam, activation=relu) performed better than other models. After selecting the best hyperparameters for our case, we ran the model with high epochs and plotted the changes of training and validation loss and accuracies. We did not choose this model, as it did not offer good accuracy even after multiple epochs. Other simpler models performed better and provided more insights.



The code is designed to allow easy switching of the models for price prediction. All the models are stored in a separate directory and the next step only requires one line of code to select the desired model.

# 8. Combined System

To produce a good output of train options and prices, we combine the coded search algorithm class and the prebuilt model. We also use the fixed prices and train features from the precomputed data when we call the model.

## 8.1 Data Pipeline

The DataPipeline class has a function called send\_input (to pipeline) that takes a dataframe with trainNumber, fromStnCode, and toStnCode as input. Based on the user class preference, it adds new columns and modifies existing columns to make the input suitable for the model. Since all the models have the same input format, it is easy to switch between them.

1> add\_class: uses the available classes in a train and selects the best class for the user according to their preference. It also adds dummy variables for each class.

2> add\_dynamic\_fare: adds two binary columns: one for whether the train has dynamic fares or not, and another for whether the train offers catering or not. These trains are usually more expensive.

3> add\_distance\_duration: uses the distance map (mentioned earlier) to calculate the distance in kilometers and the time in minutes for the train to travel between fromStnCode and toStnCode.

The send\_output function is responsible for the output pipeline. It adds the fixed charges that were subtracted from the model training to the model prediction to get the final fare. These fixed charges are reservation charges and superfast charges.

## 8.2 Trip Planner with Prices

The TripPlannerWithPrices class is the final component of our project that interacts with the user. It takes the user input of the origin station, the destination station and the date of travel. It then uses the TrainsFinder class to find direct and indirect trains for the user, the DataPipeline class to process the output of TrainsFinder and feed it to the model, and a pre built model to predict the prices. It also uses the send\_output function of DataPipeline to calculate the final fares and display them to the user.

1> get\_basic\_df\_direct and get\_basic\_df\_indirect: are used to extract only the train numbers, the origin and destination station codes from the output of TrainsFinder's multi\_train\_itineraries function. These are the only features that the model needs for precomputation.

2> format\_predictions: is used to remove unnecessary columns from the predictions and format them for display.

3> set\_prices\_direct and set\_prices\_indirect: use the model predictions to assign prices for direct trains and compute the sum of prices for indirect trains. They also select the best class available according to the user preference and show the price for that class.

We have made some optimizations in our code, which can be seen in our previous github commits. For example, we added the class preference of the user before making predictions and used vectorized operations instead of loops. These changes have reduced the final query time from 1200 milliseconds to around 600-700ms.

# 9. Conclusion and Future Work

The system has successfully achieved its objectives by providing users with reliable and efficient information on train schedules and fares. The integration of a machine learning model has notably enhanced the system's user-friendliness. Generating the final output is now as simple as running a single Python file, where users input source and destination stations along with the departure date. The output is presented in CSV format, comprising two files:

1> Direct Trains:

fromStnCode: Departure station (possibly a nearby station from the query source station).

fromArrival: Arrival date and time at the departure station (could be empty for the first station of the train).

fromDeparture: Departure date and time from the departure station (always present).

trainNumber: Identifies the chosen train.

toStnCode: Destination station (possibly a nearby station from the query destination station).

toArrival: Arrival date and time at the destination station.

toDeparture: Departure date and time from the destination station (could be empty for the last station of the train).

Journey\_time: Duration between toArrival and fromDeparture.

classCode: Optimal class based on user preference.

Price: Estimated fare for the selected class code.

2> Indirect Trains:

fromStnCode (as above).

fromArrival (as above).

fromDeparture (as above).

trainNumber1: First train to catch.

haltStation: Station where the passenger needs to switch trains.

haltArrival: Arrival date and time at the halting station for the first train.

trainNumber2: Second train to catch.

haltDeparture: Departure date and time for the second train at the halting station.

Halt\_time: Time difference between haltDeparture and haltArrival.

toArrival (as above).

toDeparture (as above).

Journey\_time (as above).

toStnCode (as above).

classCode1: Optimal class for the first train based on user preference.

price1: Estimated fare for classCode1.

classCode2: Optimal class for the second train based on user preference.

price2: Estimated fare for classCode2.

Price: Total fare (price1 + price2).

Some potential areas for further improvement are periodic data refreshes that can be done by using a python script to automatically reflect the data changes on the IRCTC website. Additionally, user feedback and system usage data can help in enhancing and expanding the system's functionalities. Another possibility for future work is a web application that allows the user to easily enter the query and interact with the output data.

# 

# 10. References

[1] <https://www.sciencedirect.com/science/article/pii/S1877050921020755>

[2] <https://github.com/ShubhangG/Train-Ticket-prediction>

[3] <https://data.gov.in/resource/indian-railways-time-table-trains-available-reservation-01112017>

[4] <https://rapidapi.com/IRCTCAPI/api/irctc1/>

[5] <https://github.com/datameet/railways/>