Machine Learning

Introduction

The entire process of training and testing the shortlisted models on the available datasets along with the deployment of the model on the cloud has been documented in detail.

Data Collection

Covered in Requirement Analysis Document.

- Software Requirement Specification
    - Machine Learning Requirements
        - Data Collection

Feature Engineering

Covered in Requirement Analysis Document (Theoretical) and Design Document (Practical).

- Requirement Analysis Document
    - Software Requirement Specification
        - Machine Learning Requirements
            - Feature Engineering

- Design Document
    - Machine Learning Design
        - Feature Engineering

Model Shortlisting

Covered in Requirement Analysis Document (Initial Research) and Design Document (Final Selection).

- Requirement Analysis Document
    - Software Requirement Specification
        - Machine Learning Requirements
            - Model Shortlisting

- Design Document
    - Machine Learning Design
        - Model Shortlisting

Training and Testing

In FYP I, we have mainly worked on 2 datasets. The local pharmacy dataset and the Corporación Favorita dataset. For each of the dataset, we have implemented different variations of 4 machine learning algorithms: Random Forest, XGBoost, Prophet, and Recurrent Neural Networks (LSTM and GRU).

Local Pharmacy Dataset

Initially, we did not have grocery store datasets, so we started to explore our machine learning options on a local (we collected it ourselves) pharmacy's dataset. The dataset contains sales data for 1 year (300,000 rows). The dataset was cleaned and compiled to as mentioned in Figure 1.0.1.

Pharmacy Dataset Update Glossary

File	Description
D1	Combines monthly pharmacy data from CSV files into a single DataFrame and saves it as `D1.csv`
D2	Reads `D1.csv`, performs analysis (correlation), drops irrelevant columns, and saves the refined data as `D2.csv`
D3	Streamlines date-related columns by splitting them into distinct attributes (date and time), eliminating redundant data columns. Optimizes column order for enhanced dataset clarity, placing the label (`looseqty`) at the end, yielding `D3.csv`
D4	Reads `D3.csv`, converts the 'date' column to datetime format, aggregates sales data based on 'date' and 'itemname,' and saves the combined data as `D4.csv`
D5	Reads `D4.csv`, filters the data for 'itemname' equal to 'PANADOL TAB,' and saves the filtered data as `D5.csv`

Figure 1.0.1 Pharmacy Dataset Update Glossary

Random Forest

We start our exploration with Random Forest (RF). We import the necessary libraries Figure 1.1.1), read a CSV file (D4.csv) into a pandas dataframe, and inspect the first 5 rows of the dataframe (Figure 1.1.2).

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_squared_log_error
```

Figure 1.1.1 Importing necessary libraries

```
df = pd.read_csv('../../Data/Pharmacy/D4.csv')
```

```
df.head()
```

	date	itemname	packunits	expiry	price	looseqty
0	01/07/2022	10CC SHIFA D/SYRINGE(UNJT)(BM)	100	12/12/24	30.00	6
1	01/07/2022	1CC BD SYRINGE	100	12/12/24	30.00	1
2	01/07/2022	3CC SYRINGE INJEKT	100	3/1/24	15.00	3
3	01/07/2022	ACCU CHECK LANCET (CHINA)	200	12/12/24	3.00	50
4	01/07/2022	ACDERMIN GEL	1	5/1/23	278.44	1

Figure 1.1.2 Loading the dataset and inspecting it

The dataframe contains aggregated sales (based on dates) of the products. It has 213,056 rows and 6 columns (Figure 1.1.3). 'looseqtyl' (sales) is the label and the other 5 columns are the features (Figure 1.1.4).

```
df.shape
```

```
(213056, 6)
```

Figure 1.1.3 Shape of the dataframe

```
X = df[['date', 'itemname', 'packunits', 'expiry', 'price']]
y = df['looseqty']
```

Figure 1.1.4 Separating features and label

Pandas' get_dummies function is used to convert categorical attributes to numerical values. This function converts each variable in as many 0/1 variables as there are different values. Boolean columns are generated for date, itemname, and expiry. As a new column is created for each date (both date and expiry), it increases the dimensionality of the dataset tremendously (Figure 1.1.5).

```
X = pd.get_dummies(X)
```

```
X.shape
```

```
(213056, 7477)
```

Figure 1.1.5 Converting categorical attributes using get_dummies and inspecting the shape of the dataframe after that

The dataset is split into training and testing sets using an 80-20 split ratio (Figure 1.1.6). As this is a time series forecasting problem, the first 80% is used for training and the last 20% is used for testing, instead of using a shuffled dataset.

```
split_index = int(0.8 * len(df))
```

```
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]
```

Figure 1.1.6 Splitting data into training set and testing set

Finally, a RF Regressor model is created and trained using the training set (Figure 1.1.7). However, the model never completed training (despite being given a sufficient amount of time to train).

```
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```
rf_model.fit(X_train, y_train)
```

Figure 1.1.7 Creating a RF model and training it on the training set

Google Colab was used then, however, it also crashed as soon as the notebook reached training (Figure 1.1.8).

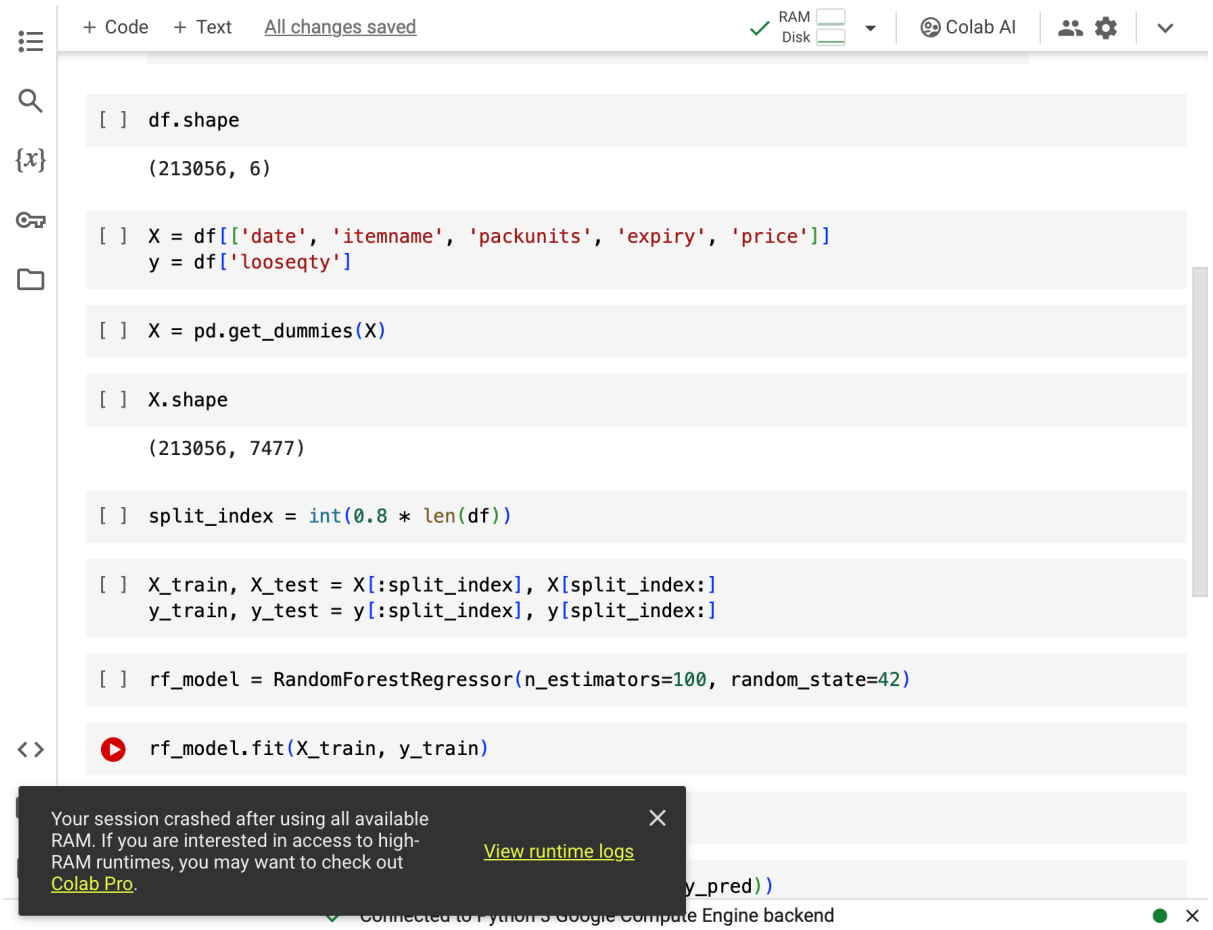

Figure 1.1.8 Google Colab crashing

We believe this is due to the dimensionality of the data. So, in order to overcome it, we reduced the scope of our dataset to only one product Panadol Tab (Figure 1.2.1). This new dataset has 337 rows and the same 6 columns (Figure 1.2.2).

```
df = pd.read_csv('../../Data/Pharmacy/D5.csv')
```

```
df.head()
```

	date	itemname	packunits	expiry	price	looseqty
0	01/07/2022	PANADOL TAB	200	4/25/24	1.70	60
1	02/07/2022	PANADOL TAB	200	4/25/24	1.70	70
2	03/07/2022	PANADOL TAB	200	4/25/24	1.70	55
3	05/07/2022	PANADOL TAB	200	4/25/24	1.45	20
4	08/07/2022	PANADOL TAB	200	4/25/24	1.70	70

Figure 1.2.1 D5.csv used instead of D4.csv

```
df.shape
```

```
(337, 6)
```

Figure 1.2.2 Shape of the dataframe

The same steps are performed with the new dataset to split into training set and testing set, create a RF model, and train it on the training set. Once the training is complete, the model is used to make predictions on the testing set (Figure 1.2.3). The predictions are evaluated using Root Mean Squared Error (RMSE) and Root Mean Squared Logarithmic Error (RMSLE). The model produced a RMSE of 137 and RMSLE of 0.338.

```
y_pred = rf_model.predict(X_test)
```

```
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
rmsle = np.sqrt(mean_squared_log_error(y_test, y_pred))
```

Figure 1.2.3 Making predictions on the test set and calculating RMSE and RMSLE

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 137.2606889869655
Root Mean Squared Logarithmic Error (RMSLE): 0.3384119387902401
```

Figure 1.2.4 Printing the RMSE and RMSLE values

Both RMSE and RMSLE showed reasonable performance by the model. We wanted to confirm that by plotting a graph of the actual values and the predicted values. While doing so, it generated an error about y_test and y_pred not being of the same dimensionality. This was resolved by converting y_test into a normal np array (Figure 1.2.5).

```
y_test.shape
```

```
(68,)
```

```
y_test = np.array(y_test)
```

Figure 1.2.5 Converting y_test to a normal np array

Upon the graph being plotted, we realised how severely under fitted the model was (Figure 1.2.6). This is due to the model not being able to learn the pattern in the data. Which is due to the fact that there are only 337 rows in the dataset. Out of which, only 269 rows are used to train the model.

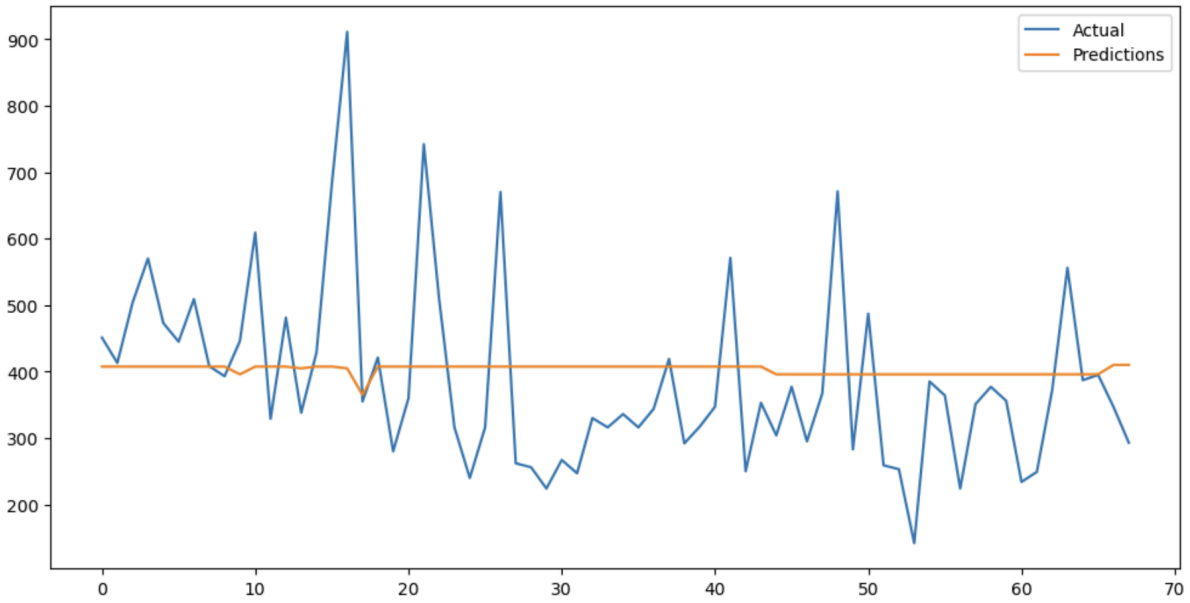

Figure 1.2.6 Actual values vs predicted values

XGBoost

Next, we experimented with a XGBoost model (Figure 1.3.1). The same steps were followed to train the XGBoost model that were performed to train the RF model.

```
xgboost_model = XGBRegressor(objective='reg:squarederror', random_state=42)
```

```
xgboost_model.fit(X_train, y_train)
```

Figure 1.3.1 Training a XGBoost model

The model produced a RMSE of 145 and RMSLE of 0.371 which are also reasonable on paper, however, we wanted to make sure that the model was not underfitting like the RF.

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 145.19829304207062
Root Mean Squared Logarithmic Error (RMSLE): 0.37083087932694353
```

Figure 1.3.2 Printing the RMSE and RMSLE values

To check whether the model is underfitting or not, we plotted the actual values and the predicted values again. The XGBoost model, as seen in Figure 1.3.3, is also underfitting.

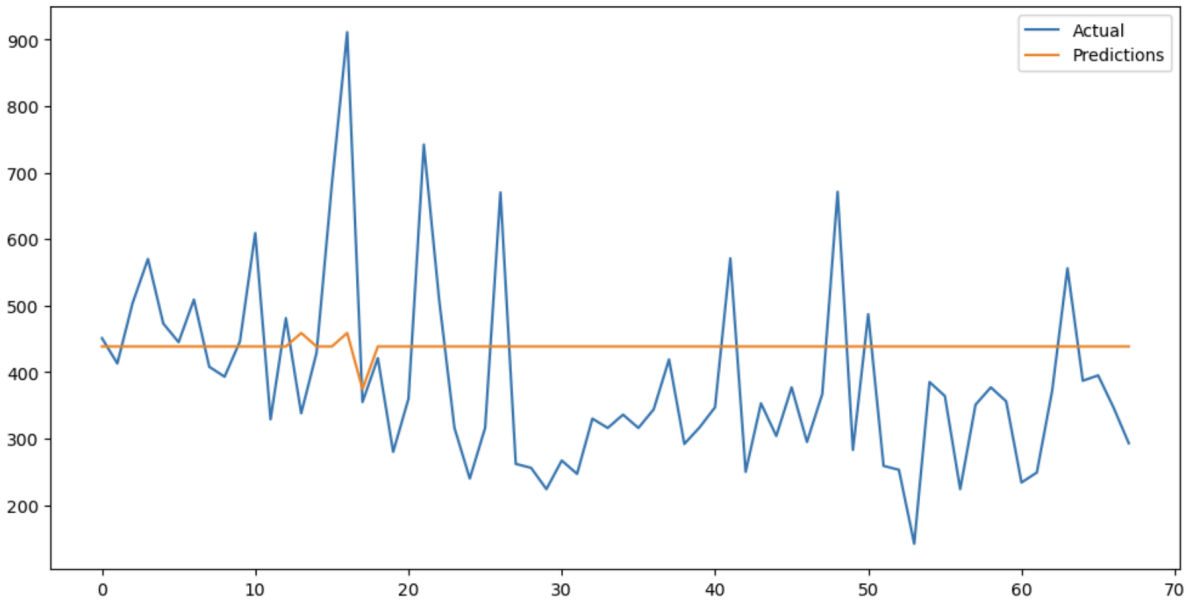

Figure 1.3.3 Actual values vs predicted values

Prophet

After the ensemble learning models failed to perform well on the dataset, we wanted to try a different kind of model. So, we used Prophet by Facebook. We started off by importing the necessary libraries (Figure 1.4.1). Installing and running Prophet was an extremely difficult task but we were finally able to make it work.

```
import pandas as pd
import numpy as np
from prophet import Prophet
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from math import sqrt
```

Figure 1.4.1 Importing necessary libraries

To match the column names expected by Prophet, the 'date' column is renamed to 'ds' and 'looseqty' column is renamed to 'y' (Figure 1.4.2).

```
df.rename(columns={'date': 'ds', 'looseqty': 'y'}, inplace=True)
```

Figure 1.4.2 Renaming date and looseqty columns for Prophet

Then, we create and train the Prophet model on only date (ds) and looseqty (y) as Prophet is a univariate model (Figure 1.4.3).

```
model = Prophet()
model.fit(train)
```

Figure 1.4.3 Creating and training the Prophet model

Then, we create a dataframe with future dates for which predictions will be made. In Prophet, you define the next number of days you want predictions for. Here we define the length of the test array as the number of days we want predictions for (Figure 1.4.4).

```
future = model.make_future_dataframe(periods=len(test))
```

```
forecast = model.predict(future)
```

Figure 1.4.4 Defining time period for predictions

The model produces an RMSE of 160 and RMSLE of 0.401 (Figure 1.4.5). This is worse than the scores from RF and XGBoost models, however, plotting the model shows it is decently fitted, unlike the RF and XGBoost models which were very underfitted (Figure 1.4.6).

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 160.64514907950073
Root Mean Squared Logarithmic Error (RMSLE): 0.4014388777975418
```

Figure 1.4.5 Printing the RMSE and RMSLE values

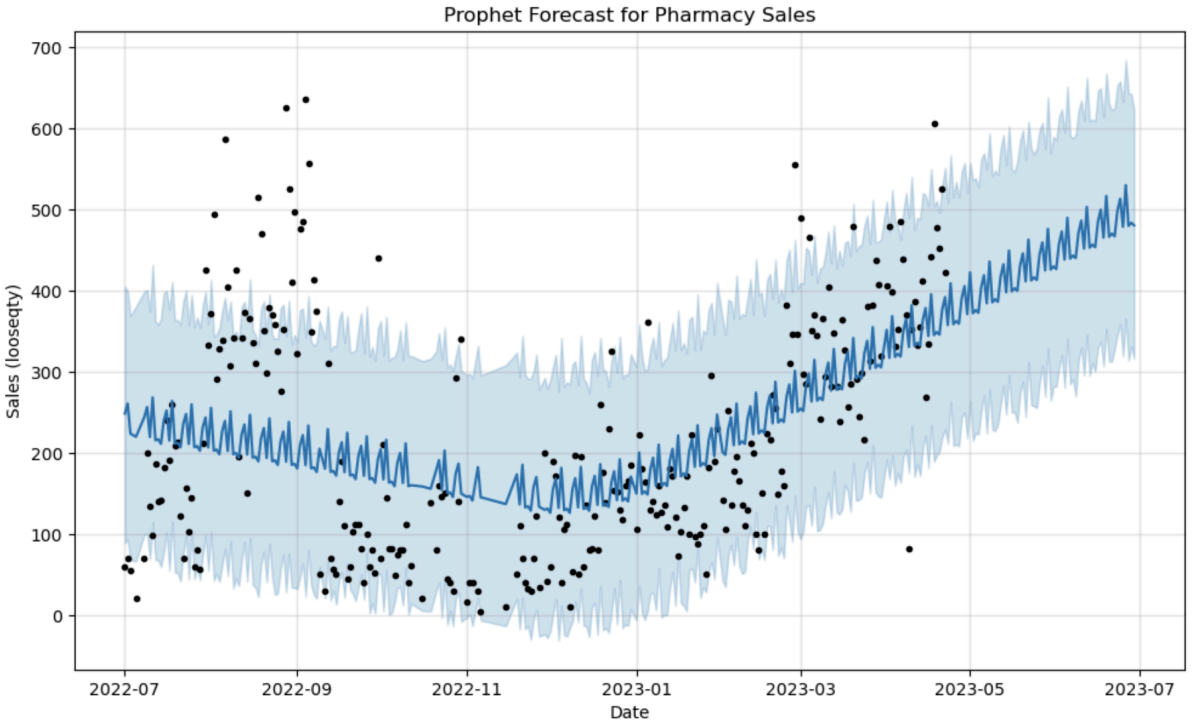

Figure 1.4.6 Prophet model's plot

After the success with Prophet on the Panadol dataset (D5.csv), we wanted to try it for other products as well. However, Prophet creates a different model for each time series (product) and this dataset contains 6053 different products (Figure 1.5.1).

```
unique_products = df['itemname'].unique()
unique_products.size
```

```
6053
```

Figure 1.5.1 Unique products in the Pharmacy dataset

It was not computationally feasible to train 6053 different models. So, we decided to train models for the top 10 products, based on the number of total sales (Figure 1.5.2).

```
total_sales_by_product = df.groupby('itemname')['y'].sum().reset_index()
```

```
top_10_products = total_sales_by_product.nlargest(10, 'y')['itemname'].tolist()
```

```
top_10_products
```

```
['PANADOL TAB',
 'LOPRIN 75MG TAB',
 "SURBEX Z TAB(30'S)",
 'GLUCOPHAGE 500MG TAB',
 'FACE MASK  3 PLY  GREEN  RS(5)',
 "DISPRIN 300MG TAB (600'S)",
 "CALPOL TAB (200'S)",
 'PANADOL EXTRA TAB',
 'METHYCOBAL TAB',
 'NUBEROL FORTE TAB']
```

Figure 1.5.2 Finding the top 10 products

Initially, there were errors while training the model for some of the products. To continue training the rest, try and except were implemented (Figure 1.5.3). Upon further inspection, it was found that the error stemmed from the model predicting slightly negative values and rmsle cannot be calculated for negative values. All the negative predicted values were converted to zero (Figure 1.5.3). We were able to train models for all of the top 10 products.

```
for product in top_10_products:
    product_data = df[df['itemname'] == product]

    try:
        model = Prophet()
        model.fit(product_data)

        future = model.make_future_dataframe(periods=len(product_data))

        forecast = model.predict(future)

        models[product] = model
        predictions[product] = forecast

        actual_values = product_data['y'].values
        predicted_values = forecast.tail(len(product_data))['yhat'].values

        predicted_values = np.maximum(predicted_values, 0)

        rmse = np.sqrt(mean_squared_error(actual_values, predicted_values))
        rmsle = np.sqrt(mean_squared_log_error(actual_values, predicted_values))

        rmse_scores[product] = rmse
        rmsle_scores[product] = rmsle

        print(f"\nRMSE for {product}: {rmse}")
        print(f"RMSLE for {product}: {rmsle}")

    except Exception as e:
        print(f"\nError processing {product}: {e}\n")
```

Figure 1.5.3 Training models for the top 10 products

The average RMSE for the top 10 products was 96 and the average RMSLE for the top 10 products was 1.385.

```
print(f"Average RMSE for the top 10 products: {average_rmse}")
print(f"Average RMSLE for the top 10 products: {average_rmsle}")
```

```
Average RMSE for the top 10 products: 96.95625655059175
Average RMSLE for the top 10 products: 1.3850059626898534
```

Figure 1.5.4

Recurrent Neural Networks

Lastly, we used a univariate LSTM model. We started off by setting the date as the index and keeping on looseqty (which is our label) in our dataset (Figure 1.6.1).

```
df['date'] = pd.to_datetime(df['date'], format='%d/%m/%Y')
df.set_index('date', inplace=True)
```

```
target_variable = 'looseqty'
df = df[[target_variable]]
```

Figure 1.6.1 Creating a univariate dataframe

Firstly, we create a function (Figure 1.6.2) to convert our dataframe into sequences. The sequence length (which is set to 10) is the number of previous time steps to consider as input features. The function iterates through the data and creates a list of lists containing the values of the previous 10 rows as input features. It then gets the value of the time step immediately following the window of previous time steps. We created a new dataframe which only has the sales (looseqty) column (as this is a univariate model) and used this dataframe to create training sequences for our LSTM model.

```
def create_sequences(data, seq_length):
    sequences = []
    for i in range(len(data) - seq_length):
        seq = data[i:i + seq_length]
        sequences.append(seq)
    return np.array(sequences)
```

```
sequence_length = 10
```

```
sequences = create_sequences(df_scaled, sequence_length)
```

Figure 1.6.2 Generating sequences for training the model

We create a sequential model (Figure 1.6.3), a model with a linear stack of layers. The first layer of the model is an LSTM layer with 50 neurons, where we pass our training sequences of length 10 and only 1 feature (looseqty) per time step. The second layer is the output layer with a single neuron as it is a regression task.

```
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(X_train.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

```
model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test), verbose=2)
```

Figure 1.6.3 Creating and training the LSTM model

The model produced a RMSE of 139 and RMSLE of 0.330 (Figure 1.6.4) which is similar to the evaluations on RF and XGBoost models, however, upon the creation of a plot of the actual values and the predicted values (Figure 1.6.5), it can be clearly seen that the model is not under-fitted.

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 139.43244021165586
Root Mean Squared Logarithmic Error (RMSLE): 0.33028218057622366
```

Figure 1.6.4 Printing the RMSE and RMSLE values

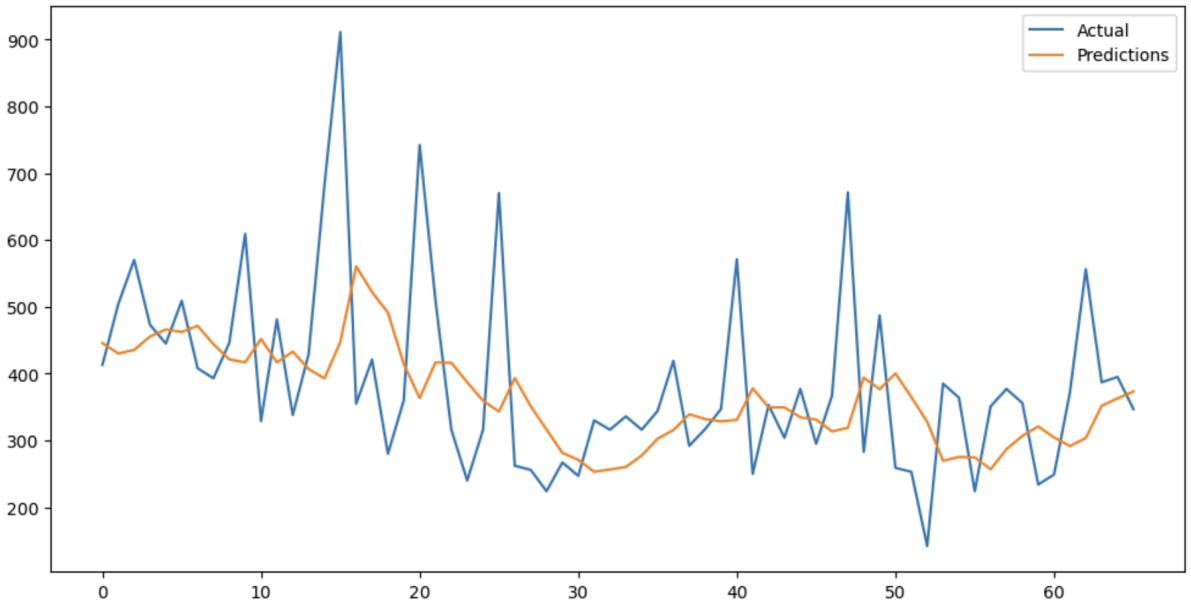

Figure 1.6.5 Actual values vs predicted values

Considering the success with LSTM, we used its variation GRU that achieves similar results while requiring lesser computation. All the same steps were followed to train the GRU model that were performed to train the LSTM model (Figure 1.7.1).

```
model = Sequential()
model.add(GRU(50, activation='relu', input_shape=(X_train.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Figure 1.7.1 Creating and training the GRU model

The model produced a RMSE of 137 and RMSLE of 0.325 (Figure 1.7.2) which is actually slightly better than the LSTM model (we expected the performance to fall) and the plot (Figure 1.7.3) also shows that the model is not under-fitted.

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 137.2803728457543
Root Mean Squared Logarithmic Error (RMSLE): 0.32572063927800055
```

Figure 1.7.2 Printing the RMSE and RMSLE values

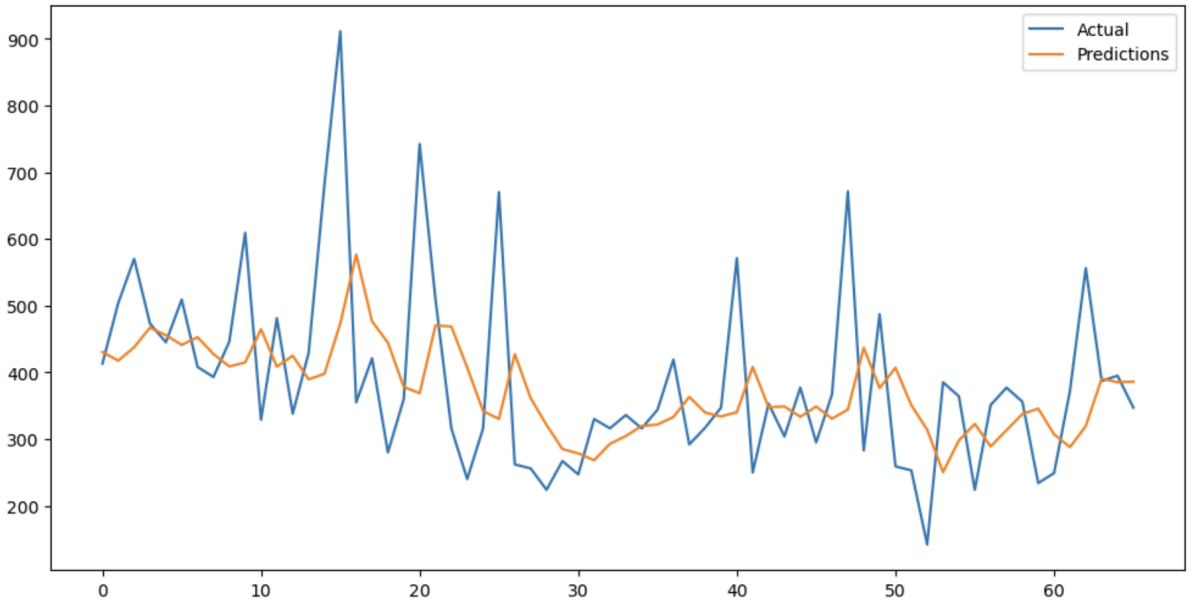

Figure 1.7.3 Actual values vs predicted values

Summary

Here is a side by side comparison of all the models trained and tested on the Pharmacy dataset (D5.csv).

Model	RMSE	RMSLE
Random Forest	137	0.338
XGBoost	145	0.371
Prophet	160	0.401
LSTM	139	0.330
GRU	137	0.325

Figure 1.8.1 Model Summary

Corporación Favorita Grocery Sales Forecasting

Initially, we were unable to train models on the Corporación Favorita dataset due to its sheer size (3+ million rows), despite utilising services like Kaggle (Figure 2.1.1) and Google Colab (Figure 2.1.2). As suggested by Ma'am Huda, we used batch training to overcome this issue.

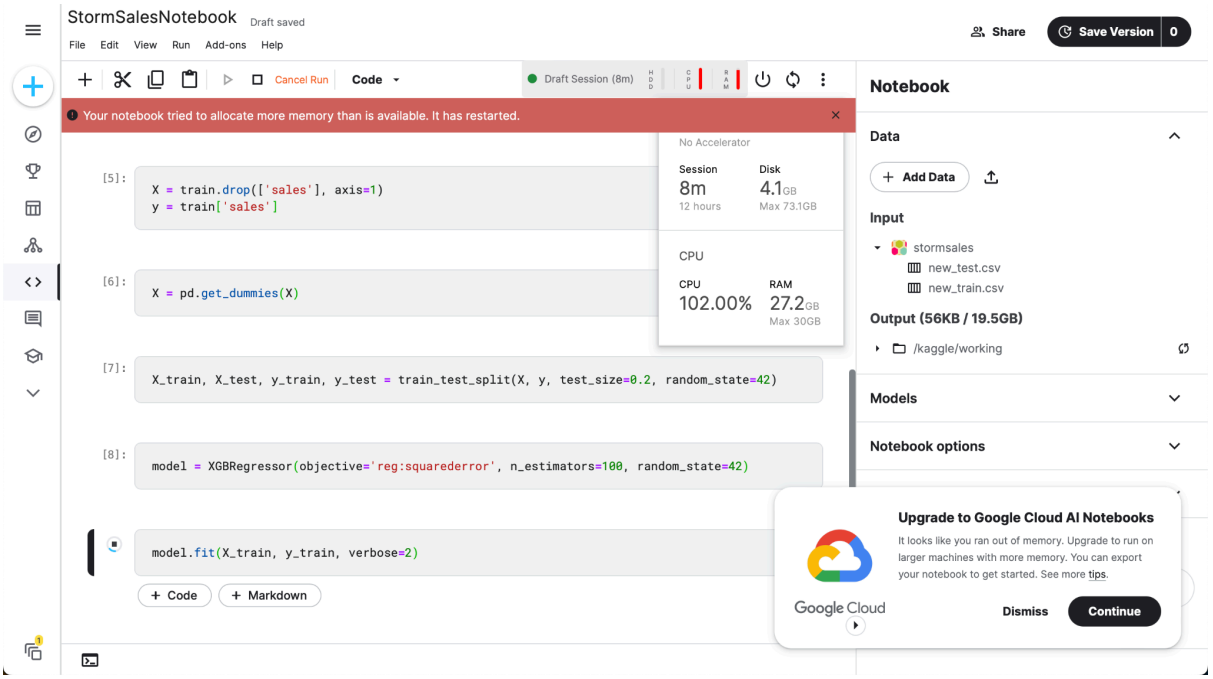

Figure 2.1.1 Kaggle crashing

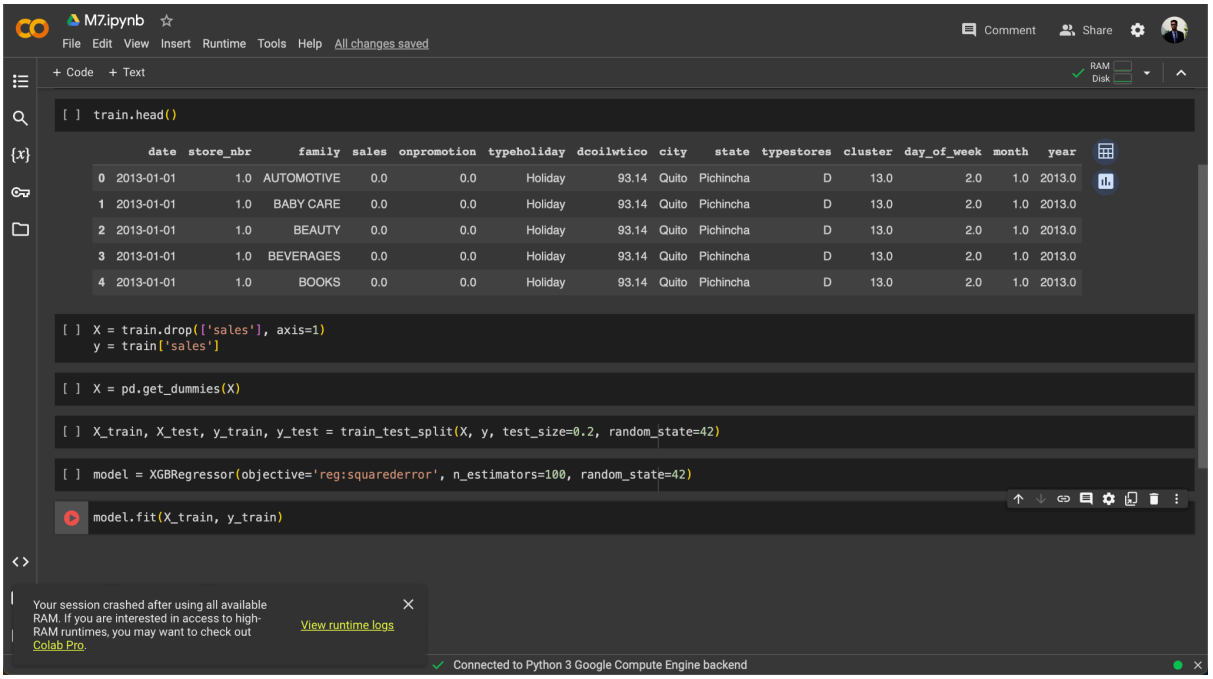

Figure 2.1.2 Google Colab crashing

During the initial exploration of the Corporación Favorita dataset, we kept the date feature as seen in the train.head() in Figure 2.1.2. The preprocessing was done on the available datasets as mentioned in the Design Document. While making predictions on the unseen

dataset we found an issue. We were using the Pandas' get_dummies function to convert our categorical attributes to numerical values. This function converts each variable in as many 0/1 variables as there are different values. The boolean columns generated for family, typeholiday, city, state, and typestore worked as their values stay consistent across the train and test data, however, the date column generated error as the model was not trained on the dates (columns) from the test data. To overcome this, we combined both the datasets (Figure 2.1.3) and then applied the get_dummies function (a temporary fix as it would not work on any dates other than the ones in the train and test data). The data was split back into test and train after being transformed.

```
In [3]: combined = pd.concat([train, test], ignore_index=True)

In [4]: combined = pd.get_dummies(combined)

In [5]: first_range = (0, 3000887)
        second_range = (3000888, combined['id'].max())

In [6]: first_range = (0, 3000887)
        second_range = (3000888, combined['id'].max())

In [7]: train = combined[(combined['id'] >= first_range[0]) & (combined['id'] <= first_range[1])]
        test = combined[(combined['id'] >= second_range[0]) & (combined['id'] <= second_range[1])]
```

Figure 2.1.3 Combining train and test dataset to apply get_dummies together

The function increased the dimensionality of the data, making the training even slower. The train data, containing 1782 boolean columns, occupied 5.2 GB in the memory (Figure 2.1.4).

```
In [8]: train.head()
```

Out[8]:

	id	store_nbr	sales	onpromotion	dcoilwtico	cluster	day_of_week	month	year	date_2013-01-01	...	state_Pastaza	state_Pich
0	0	1	0.0	0	93.14	13	2	1	2013	True	...	False	
1	1	1	0.0	0	93.14	13	2	1	2013	True	...	False	
2	2	1	0.0	0	93.14	13	2	1	2013	True	...	False	
3	3	1	0.0	0	93.14	13	2	1	2013	True	...	False	
4	4	1	0.0	0	93.14	13	2	1	2013	True	...	False	

5 rows × 1791 columns

```
In [9]: train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3000888 entries, 0 to 3000887
Columns: 1791 entries, id to typestores_E
dtypes: bool(1782), float64(2), int64(7)
memory usage: 5.2 GB
```

Figure 2.1.4 Information about the updated train dataset

Considering the size of the dataset, batch training was used (Figure 2.1.4). The dataset was divided in batches of 10,000 samples. The total number of batches were calculated to cover the entire training dataset (which in this case were 301). Lastly, the model is trained one batch at a time.

In [15]:
```
batch_size = 10000
```

In [16]:
```
num_batches = len(X) // batch_size + 1
```

In [17]:
```
model = XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
```

In [18]:
```
for i in range(num_batches):
    start_idx = i * batch_size
    end_idx = (i + 1) * batch_size

    X_batch = X.iloc[start_idx:end_idx]
    y_batch = y.iloc[start_idx:end_idx]

    model.fit(X_batch, y_batch)

    print(f"Batch {i + 1}/{num_batches} completed")
```

```
Batch 1/301 completed
Batch 2/301 completed
Batch 3/301 completed
Batch 4/301 completed
Batch 5/301 completed
```

Figure 2.1.4 Batch Training

The predictions were generated on the XGBRegressor model. All the slightly negative predictions were converted to zero to ensure calculating rmsle is possible (Figure 2.1.5). A submission csv file was created for the ongoing Store Sales competition on Kaggle.

In [19]:
```
predictions = model.predict(test_data)
```

In [20]:
```
submission = pd.DataFrame({'id': test['id'], 'sales': predictions})
```

In [21]:
```
submission['sales'] = submission['sales'].apply(lambda x: max(0, x))
```

In [22]:
```
submission.to_csv('../../Data/Kaggle/StoreSales/submission.csv', index=False)
```

Figure 2.1.5 Generating predictions, fixing them, and creating a submission file

We submitted the predictions generated using the XGBRegressor model in the Store Sales competition on Kaggle (Figure 2.1.6), producing an rmsle of 1.08120 on the unseen dataset and ranking 575th on the leaderboard.

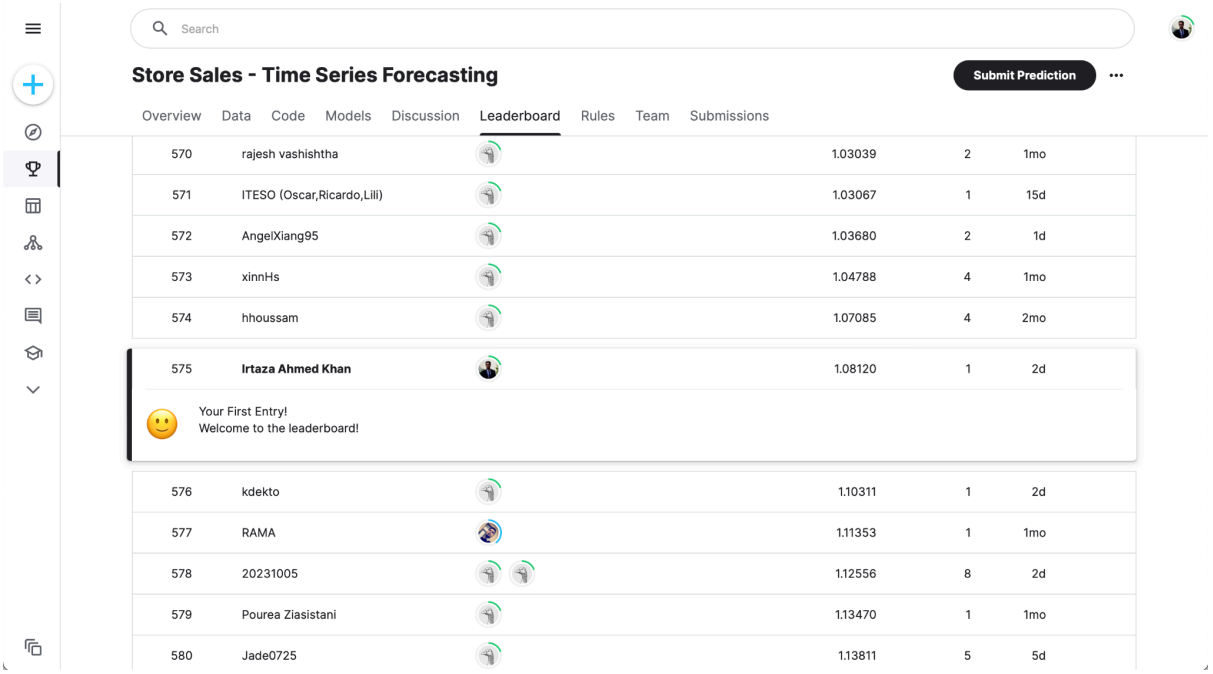

Figure 2.1.6 Store Sales - Time Series Forecasting Leaderboard

To overcome the issue of increased dimensionality, initially the date column was dropped and 4 separate features were generated from it (day_of_week, day, month, and year) and eventually get_dummies (one hot encoding) was replaced with Label Encoder to further reduce the dimensionality. However, Label Encoder provides an arbitrary order to categorical values which negatively affects the performance of the model.

Random Forest

Just like with the Pharmacy dataset, we started our implementation with Random Forest (RF). We read the processed CSV file (new_train.csv) into a pandas dataframe, and inspected the first 5 rows of the dataframe (Figure 2.2.1).

```
df = pd.read_csv("../../Data/Kaggle/StoreSales/new_train.csv")
```

```
df.head()
```

	family	sales	onpromotion	typeholiday	dcoilwtico	city	state	typestores	cluster	day_of_week	day	month	year
0	AUTOMOTIVE	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
1	BABY CARE	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
2	BEAUTY	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
3	BEVERAGES	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
4	BOOKS	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013

Figure 2.2.1 Loading the dataset and inspecting it

Pandas' get_dummies function is used to convert categorical attributes to numerical values (Figure 2.2.2). Boolean columns are generated for family, typeholiday, city, state, and typestores. As we did not use this function on date (we split it into day of week, day, month,

and year) this time, it did not increase the dimensionality to the same extent it did in the pharmacy dataset.

```
X = pd.get_dummies(X)
```

```
X.shape
```

```
(3000888, 90)
```

Figure 2.2.2 Converting categorical attributes using get_dummies

Batch of size 32 is a rule of thumb and considered a good initial choice so the batch size was set to 32 and the number of batches were calculated (Figure 2.2.3).

```
batch_size = 32
```

```
num_batches = len(X_train) // batch_size + 1
```

Figure 2.2.3 Setting batch size and calculating number of batches

We created and trained a RF model. Considering the size of the dataset, batch training was used (Figure 2.2.4). The dataset was divided in batches of 32 samples. The total number of batches were calculated to cover the entire training dataset (which in this case were 75023). Lastly, the model is trained one batch at a time.

```
model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```
for i in range(num_batches):
    start_idx = i * batch_size
    end_idx = (i + 1) * batch_size

    X_batch = X_train.iloc[start_idx:end_idx]
    y_batch = y_train.iloc[start_idx:end_idx]

    model.fit(X_batch, y_batch)

    print(f"Batch {i + 1}/{num_batches} completed")
```

```
Batch 1/75023 completed
```

Figure 2.2.4 Using batch training to train the RF model

It took an unreasonable amount of time to complete the training. The model produced a significantly high RMSE of 1418 and RMSLE of 2.283 (Figure 2.2.5)

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 1418.1705197469257
Root Mean Squared Logarithmic Error (RMSLE): 2.2832200118427144
```

Figure 2.2.5 Printing the RMSE and RMSLE values

The graph was plotted for 100 (for better visibility) actual and predicted values from the middle of the test set (Figure 2.2.6) to find out the cause for such high RMSE and RMSLE. As seen in Figure 2.2.7, the model is severely under-fitted.

```
plt.figure(figsize=(12, 6))
plt.plot(y_test[300100:300200], label='Actual')
plt.plot(y_pred[300100:300200], label='Predictions')
plt.legend()
plt.show()
```

Figure 2.2.6 Plotting 100 values from actual and predicted values

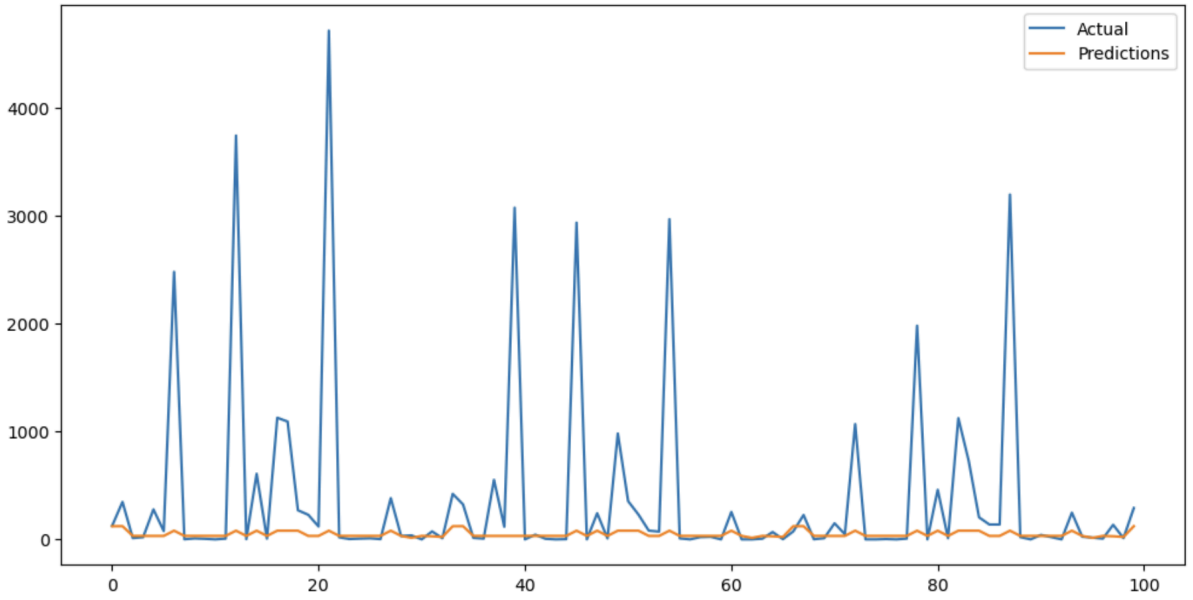

Figure 2.2.7 Actual values vs predicted values

While training the XGBoost, the kernel kept crashing with the batch size of 32. Upon increasing the batch size, XGBoost showed much better performance than RF so we trained another RF model with increased batch size (Figure 2.3.1). With batch size set to 512, the model was trained on 4689 batches (Figure 2.3.2).

```
batch_size = 512
```

Figure 2.3.1 Batch size

```
Batch 1/4689 completed
Batch 2/4689 completed
Batch 3/4689 completed
Batch 4/4689 completed
Batch 5/4689 completed
```

Figure 2.3.2 Total number of batches

The model did not only train faster but it also performed significantly better with a RMSE of 975 and RMSLE of 1.618 (Figure 2.3.3). To ensure the under-fitting problem was resolved, we plotted a graph of the 100 actual and predicted values and as it can be seen in Figure 2.3.4, the model is very well fitted (Figure 2.3.4).

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 975.9726172221381
Root Mean Squared Logarithmic Error (RMSLE): 1.6175720470454031
```

Figure 2.3.3 Printing the RMSE and RMSLE values

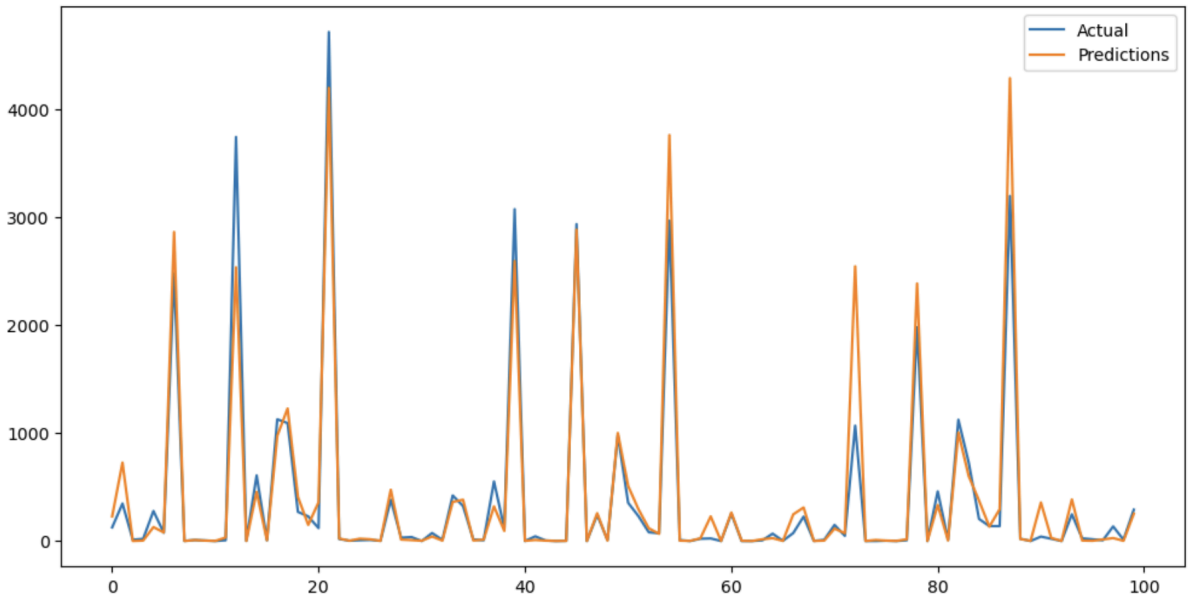

Figure 2.3.4 Actual values vs predicted values

Considering the improvement in results by increasing the batch size, we increased the batch size further (Figure 2.4.1). With a batch size of 1024, the model was trained on 2345 batches (Figure 2.4.2). However, there was no difference in the RMSE or RMSLE, as they stayed exactly the same at 975 and 1.618 respectively (Figure 2.4.3). The fit of the model as seen in Figure 2.4.4 is also identical.

```
batch_size = 1024
```

Figure 2.4.1 Batch size

```
Batch 1/2345 completed
Batch 2/2345 completed
Batch 3/2345 completed
Batch 4/2345 completed
Batch 5/2345 completed
```

Figure 2.4.2 Total number of the batches

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 975.9726172221381
Root Mean Squared Logarithmic Error (RMSLE): 1.6175720470454031
```

Figure 2.4.3 Printing the RMSE and RMSLE values

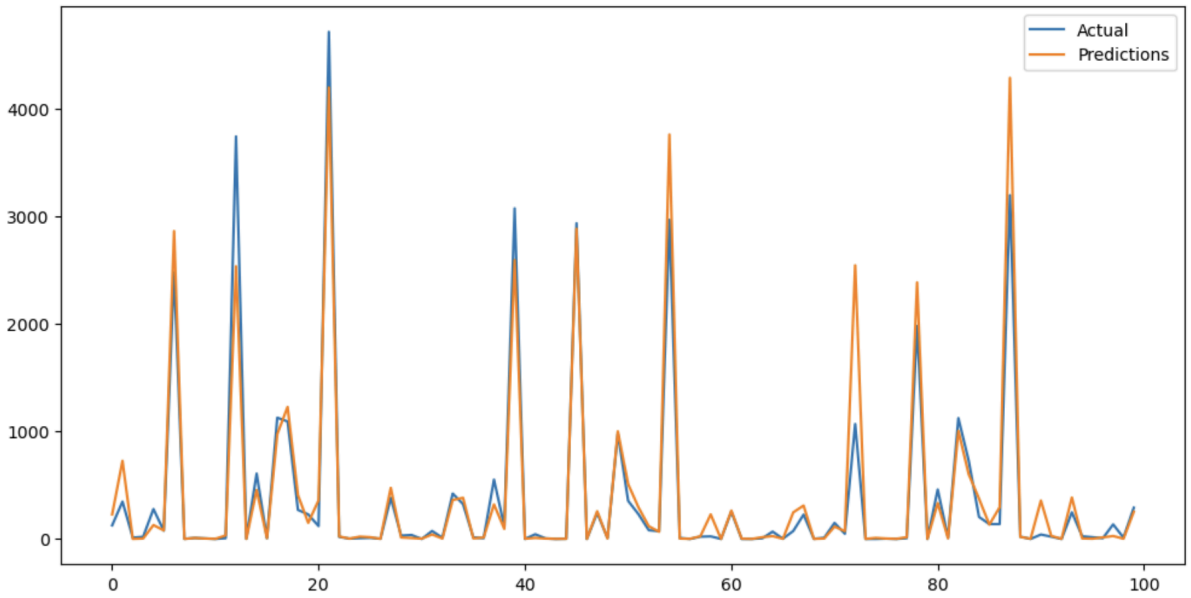

Figure 2.4.4 Actual values vs predicted values

Random Forest took a lot of time to train, even with a batch size of 1024, as it did not utilise all the available resources (Figure 2.4.5).

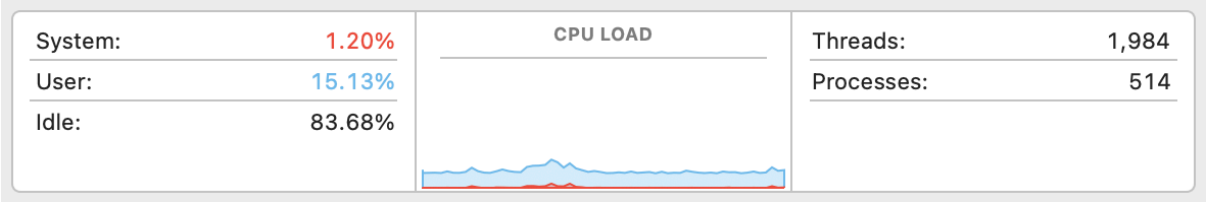

Figure 2.4.5 CPU usage

XGBoost

Next, we trained a XGBoost model (Figure 2.5.1) with the same configurations as the RF model, however, the kernel kept crashing (Figure 2.5.2).

```
model = XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
```

Figure 2.5.1 Creating a XGBoost model

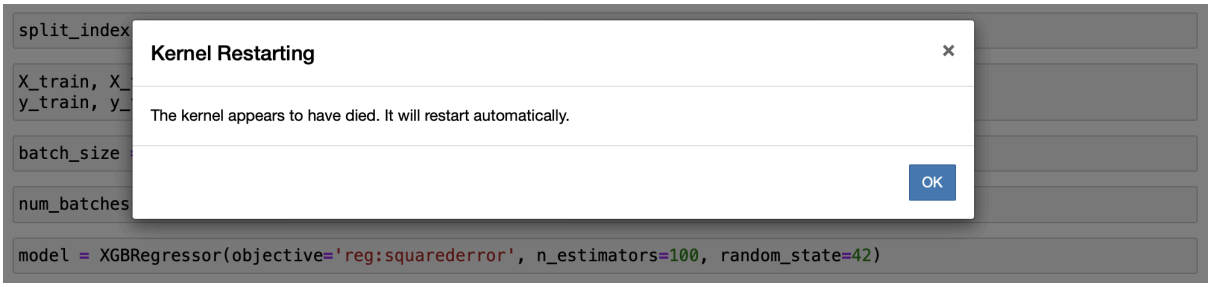

Figure 2.5.2 Kernel died error

Increasing the batch size to 512 solved the issue (Figure 2.5.3). The model performed significantly better (with an RMSE of 1119 AND RMSLE 1.462) than the RF model (with 32 batch size) but when the RF model was also trained with a batch size of 512, it outperformed XGBoost (Figure 2.5.4). The model is not under-fitted and is able to generalise well on unseen as seen in Figure 2.5.5.

```
batch_size = 512
```

Figure 2.5.3 Batch size

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 1119.766051910334
Root Mean Squared Logarithmic Error (RMSLE): 1.4624035138159117
```

Figure 2.5.4 Printing the RMSE and RMSLE values

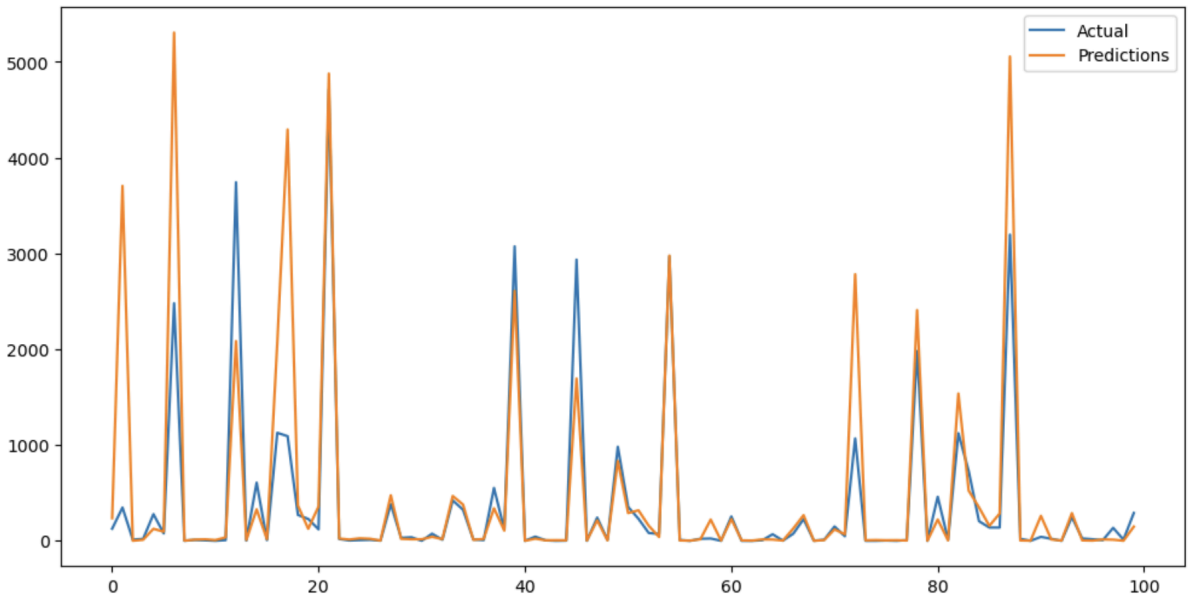

Figure 2.5.5 Actual values vs predicted values

The XGBoost trained significantly faster than the RF model. One of the reasons being that XGBoost utilised all the available resources while training (Figure 2.5.6). This was a key reason for XGBoost being used more than RF for experimentation during the project despite producing slightly worse results than RF.

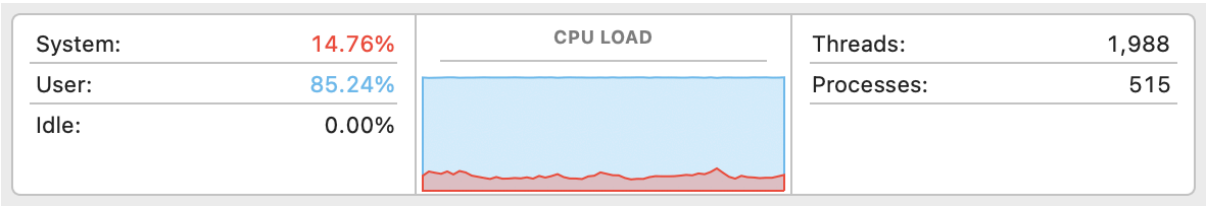

Figure 2.5.6 CPU usage

Prophet

Prophet works best with time series that have several seasons of historical data which makes Prophet a very good fit for this dataset as the dataset contains historical data for several years. As Prophet creates a different model for each time series, in this case store and product, it would create (number of stores * number of unique products) models which was computationally not infeasible. So, we crested one model for store number = 1 and family = 3 (Beverages) (Figure 2.6.1). The model was trained only on date and sales as it is a univariate model.

```
df = df[(df['store_nbr'] == 1) & (df['family'] == 3)]
```

```
df.head()
```

id	date	store_nbr	family	sales	onpromotion	typeholiday	dcoilwtico	city	state	typestores	cluster	day_of_week	day
3	2013-01-01	1	3	0.0	0	3	93.14000	18	12	3	13	2	1
1785	2013-01-02	1	3	1091.0	0	4	93.14000	18	12	3	13	3	2
3567	2013-01-03	1	3	919.0	0	4	92.97000	18	12	3	13	4	3
5349	2013-01-04	1	3	953.0	0	4	93.12000	18	12	3	13	5	4
7131	2013-01-05	1	3	1160.0	0	4	93.12009	18	12	3	13	6	5

Figure 2.6.1 Only sales for beverages (family = 3) at store number 1

The model produced a very low RMSE (653) and RMSLE (0.554) score (Figure 2.6.2), however, it cannot be compared with the evaluations earlier as we don't know if the Prophet will perform just as well on other families and stores. The plot of the model does show the model's ability to fit well on such time series (Figure 2.6.3).

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 653.0936302929073
Root Mean Squared Logarithmic Error (RMSLE): 0.5537333343983055
```

Figure 2.6.2 Printing the RMSE and RMSLE values

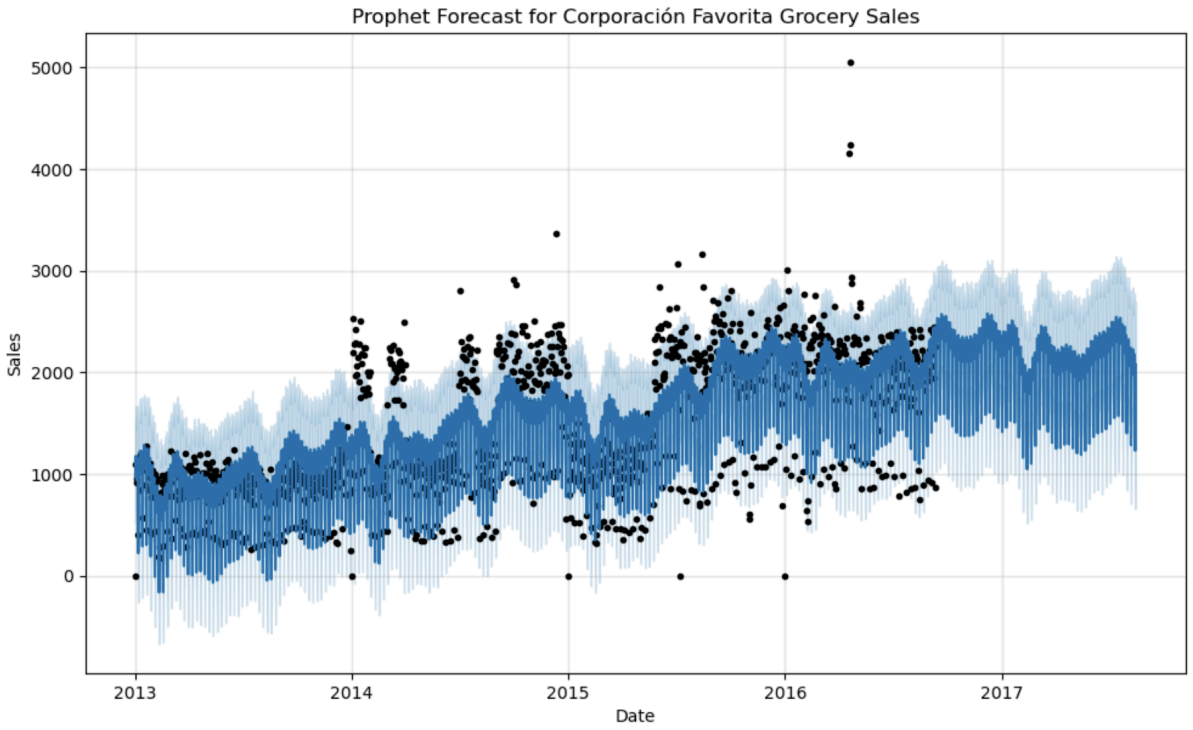

Figure 2.6.3 Prophet model's plot

Recurrent Neural Networks

We implemented 3 recurrent neural network (RNN) models for the Corporación Favorita dataset: LSTM (Univariate), GRU (Univariate), and LSTM (Multivariate). We started with the univariate LSTM model. In Figure 2.7.1, we import the necessary libraries and then read a CSV file (processed training dataset) into a pandas dataframe.

```
import tensorflow as tf
import pandas as pd
import numpy as np
```

```
df = pd.read_csv("../../Data/Kaggle/StoreSales/processed_train_v2.csv")
```

Figure 2.7.1 Importing libraries and loading the dataset

Firstly, we create a function (Figure 2.7.2) to convert our dataframe into X (training sequences) and y (labels) numpy arrays. Numpy arrays are used to make it easier to manipulate the data. The window_size (which is set to 7) is the number of previous time steps to consider as input features. The function iterates through the data, excluding the last 7 rows, and creates a list of lists containing the values of the previous 7 rows as input features. It then gets the value of the time step immediately following the window of previous time steps. We created a new dataframe which only has the sales column (as this is a univariate model) and used this dataframe to create training sequences for our LSTM (Univariate) model.

```
def df_to_X_y(df, window_size=7):
  df_as_np = df.to_numpy()
  X = []
  y = []
  for i in range(len(df_as_np)-window_size):
    row = [[a] for a in df_as_np[i:i+window_size]]
    X.append(row)
    label = df_as_np[i+window_size]
    y.append(label)
  return np.array(X), np.array(y)
```

```
WINDOW_SIZE = 7
```

```
X1, y1 = df_to_X_y(sales, WINDOW_SIZE)
```

Figure 2.7.2 Converting the data frame into training sequences

Then, we split the data into training (80%), validation (10%), and test sets (10%). Array slicing operations are used, as shown in Figure 2.7.3, to ensure the model is trained on the first 80%, validated on the next 10%, and finally tested on the last 10%.

```
X1.shape, y1.shape
```

```
((3000881, 7, 1), (3000881,))
```

```
X_train1, y_train1 = X1[:2400710], y1[:2400710]
X_val1, y_val1 = X1[2400710:2700799], y1[2400710:2700799]
X_test1, y_test1 = X1[2700799:], y1[2700799:]
```

```
X_train1.shape, y_train1.shape, X_val1.shape, y_val1.shape, X_test1.shape, y_test1.shape
```

```
((2400710, 7, 1),
 (2400710,),
 (300089, 7, 1),
 (300089,),
 (300082, 7, 1),
 (300082,))
```

Figure 2.7.3 Splitting the data in training, validation, and test sets

Next, we import various components from TensorFlow Keras specific to the training of the model. We create a sequential model (Figure 2.7.4), a model with a linear stack of layers. The first layer of the model is an InputLayer, where we pass our training sequences of length 7 and only 1 feature (sales) per time step as it is a univariate model. The second layer is an LSTM layer with 64 neurons. The third layer is a Dense layer with 8 ReLUs. Lastly, the output layer is a dense layer with a single ReLU neuron as it's a regression task.

```
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import *
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.metrics import RootMeanSquaredError
from tensorflow.keras.optimizers import Adam
```

```
model1 = Sequential()
model1.add(InputLayer((7, 1)))
model1.add(LSTM(64))
model1.add(Dense(8, 'relu'))
model1.add(Dense(1, 'relu'))
```

Figure 2.7.4 Importing libraries and initialising our model

Then, a ModelCheckpoint callback is created and is configured to save the best model during training based on the validation loss (Figure 2.7.5). The chosen loss function is Mean Squared Error, the optimizer is Adam (with a learning rate of 0.0001), and the metric for evaluation is Root Mean Squared Error. The training is performed with a batch size of 1000, over 10 epochs, and the ModelCheckpoint callback (cp1) is specified to save the best model. The lowest RMSE on validation set (566) is achieved in the 10th epoch (Figure 2.7.6).

```
cp1 = ModelCheckpoint('model1/', save_best_only=True)
```

```
model1.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=0.0001), metrics=[RootMeanSquaredError()])
```

```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use th
e legacy Keras optimizer instead, located at `tf.keras.optimizers.legacy.Adam`.
```

```
model1.fit(X_train1, y_train1, validation_data=(X_val1, y_val1), batch_size=1000, epochs=10, callbacks=[cp1])
```

```
Epoch 1/10
2397/2401 [============================>.] - ETA: 0s - loss: 316139.6875 - root_mean_squared_error: 562.2630INFO:te
nsorflow:Assets written to: model1/assets
INFO:tensorflow:Assets written to: model1/assets
2401/2401 [==============================] - 28s 12ms/step - loss: 316046.9375 - root_mean_squared_error: 562.1805
- val_loss: 659186.1875 - val_root_mean_squared_error: 811.9028
```

Figure 2.7.5 Compiling and fitting the model

```
Epoch 10/10
2398/2401 [============================>.] - ETA: 0s - loss: 146996.0312 - root_mean_squared_error: 383.4006INFO:te
nsorflow:Assets written to: model1/assets
INFO:tensorflow:Assets written to: model1/assets
2401/2401 [==============================] - 27s 11ms/step - loss: 146947.9375 - root_mean_squared_error: 383.3379
- val_loss: 321167.2188 - val_root_mean_squared_error: 566.7162
```

Figure 2.7.6 Lowest RMSE

Finally, we use the trained model to make predictions on the test set (data it has never seen before). The predictions are obtained by calling the predict method on the model, and flatten() is used to convert the predictions into a 1D array. Then, Matplotlib is used to plot the test predictions and actual values. For better clarity, only hundred data points (900 to 1000) are visualised. As it can be seen in Figure 2.7.7, the model does a very good job at making predictions as the predicted values are very close to the actual values.

```
test_predictions = model1.predict(X_test1).flatten()
test_results = pd.DataFrame(data={'Test Predictions':test_predictions, 'Actuals':y_test1})
```

```
9378/9378 [==============================] - 5s 485us/step
```

```
plt.plot(test_results['Test Predictions'][900:1000])
plt.plot(test_results['Actuals'][900:1000])
```

```
[<matplotlib.lines.Line2D at 0x324147250>]
```

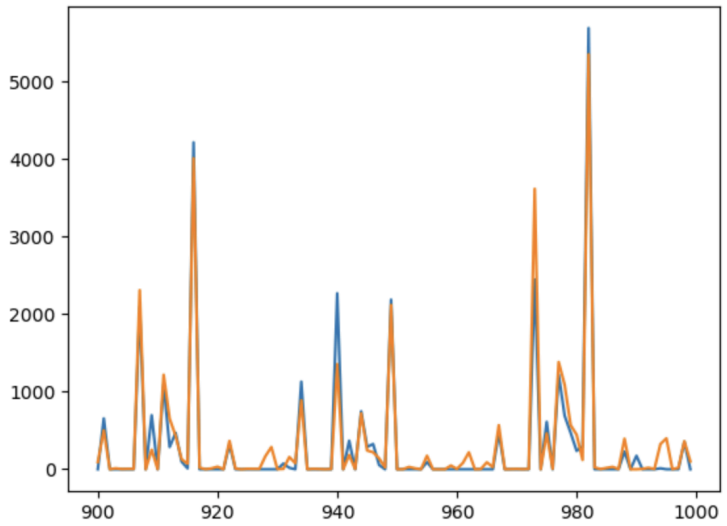

Figure 2.7.7 Test predictions vs actual values plotted

Shown in Figure 2.7.8, the second model follows a similar architecture as the first model with the main difference being the use of a GRU layer instead of an LSTM layer. The model showed similar performance to the LSTM model while taking less time to train.

```
model2 = Sequential()
model2.add(InputLayer((7, 1)))
model2.add(GRU(64))
model2.add(Dense(8, 'relu'))
model2.add(Dense(1, 'relu'))
```

Figure 2.7.8 GRU (Univariate)

Finally, the third model (Figure 2.7.9) follows a similar architecture as the previous models, but it has a different input shape, where each time step has 14 features.

```
model3 = Sequential()
model3.add(InputLayer((7, 14)))
model3.add(LSTM(64))
model3.add(Dense(8, 'relu'))
model3.add(Dense(1, 'relu'))
```

Figure 2.7.9 LSTM (Multivariate)

```
Epoch 10/10
74999/75023 [============================>.] - ETA: 0s - loss: 2.4119 - root_mean_squared_error: 1.5530INFO:tensorf
low:Assets written to: model3/assets
INFO:tensorflow:Assets written to: model3/assets
75023/75023 [==============================] - 109s 1ms/step - loss: 2.4115 - root_mean_squared_error: 1.5529 - val
_loss: 1.8977 - val_root_mean_squared_error: 1.3776
```

Figure 2.7.10 Lowest RMSE on LSTM (Multivariate)

Summary

Here is a side by side comparison of all the models trained and tested on the Corporación Favorita dataset.

Model	Batch Size	RMSE	RMSLE
Random Forest	32	1418	2.283
Random Forest	512	975	1.618
Random Forest	1024	975	1.618
XGBoost	512	1119	1.462
Prophet*	-	653	0.554
LSTM	1000	566	-
GRU	1000	1071	-

Figure 2.8.1 Model Summary

Prophet was only trained on a subset of the dataset (Figure 2.6.1).

Deployment

For the purpose of giving a demo of the ML model, we created a web application using Flask, a Python web framework (recommended by our external advisor). The development initially started on GitHub Codespaces, but was shifted to VS Code due to server issues. A Conda environment was set up and required packages were installed (Figure 3.1).

requirements.txt U

ModelDeployment > requirements.txt

```
blinker==1.7.0
click==8.1.7
Flask==3.0.0
gunicorn==21.2.0
itsdangerous==2.1.2
Jinja2==3.1.2
joblib==1.3.2
MarkupSafe==2.1.3
numpy==1.26.2
packaging==23.2
pandas==2.1.4
pickle-mixin==1.0.2
python-dateutil==2.8.2
pytz==2023.3.post1
scikit-learn==1.3.2
scipy==1.11.4
six==1.16.0
threadpoolctl==3.2.0
tzdata==2023.3
Werkzeug==3.0.1
xgboost==1.7.3
```

Figure 3.1 pip freeze > requirements.txt

To reduce the computational requirements, Label Encoder was used for each categorical attribute (Figure 3.2).

In [10]:
```
family_encoder = LabelEncoder()
typeholiday_encoder = LabelEncoder()
city_encoder = LabelEncoder()
state_encoder = LabelEncoder()
typestores_encoder = LabelEncoder()
```

In [11]:
```
X['family_encoded'] = family_encoder.fit_transform(X['family'])
X['typeholiday_encoded'] = typeholiday_encoder.fit_transform(X['typeholiday'])
X['city_encoded'] = city_encoder.fit_transform(X['city'])
X['state_encoded'] = state_encoder.fit_transform(X['state'])
X['typestores_encoded'] = typestores_encoder.fit_transform(X['typestores'])
```

Figure 3.2 Label Encoder

Then, the encoder for each attribute was exported using Pickle (Figure 3.3) and the model was exported using Joblib (Figure 3.4). Initially, Pickle was used for both however the model did not work and upon research we found out that Joblib is more suitable for exporting XGBoost models.

In [12]:
```
with open('pickle/family_encoder.pkl', 'wb') as file:
    pickle.dump(family_encoder, file)

with open('pickle/typeholiday_encoder.pkl', 'wb') as file:
    pickle.dump(typeholiday_encoder, file)

with open('pickle/city_encoder.pkl', 'wb') as file:
    pickle.dump(city_encoder, file)

with open('pickle/state_encoder.pkl', 'wb') as file:
    pickle.dump(state_encoder, file)

with open('pickle/typestores_encoder.pkl', 'wb') as file:
    pickle.dump(typestores_encoder, file)
```

Figure 3.3 Exporting all the encoders through Pickle

```
In [28]: dump(model, 'joblib/M10.joblib')

Out[28]: ['joblib/M10.joblib']
```

Figure 3.4 Exporting the model using Joblib

To check whether the model was working after being deployed locally, a set of hard coded values (Figure 3.5) were given to the model. The values were in the form as they would be input in the actual deployment. The only attribute that was not hard coded was the family (category of the product). This deployment made predictions for all the listed products, but only for one specific (hard coded) date. Functionality to dynamically manipulate variables was added later.

```
data = {
    'store_nbr': 1,
    'family_encoded': "",
    'onpromotion': 0,
    'typeholiday_encoded': "Holiday",
    'dcoilwtico': 46.8,
    'city_encoded': "Quito",
    'state_encoded': "Pichincha",
    'typestores_encoded': "D",
    'cluster': 13,
    'day_of_week': 3,
    'day': 16,
    'month': 8,
    'year': 2017
}
```

Figure 3.5 Hard coded values for the model

The encoders that were exported earlier were imported on Flask using Pickle. Each encoder was used to transform the value of their respective attribute (Figure 3.6).

```
if request.method == 'POST':
    df['family_encoded'] = request.form.get('family_encoded')

with open('models/family_encoder.pkl', 'rb') as file:
    family_encoder = pickle.load(file)

with open('models/typeholiday_encoder.pkl', 'rb') as file:
    typeholiday_encoder = pickle.load(file)

with open('models/city_encoder.pkl', 'rb') as file:
    city_encoder = pickle.load(file)

with open('models/state_encoder.pkl', 'rb') as file:
    state_encoder = pickle.load(file)

with open('models/typestores_encoder.pkl', 'rb') as file:
    typestores_encoder = pickle.load(file)

df['family_encoded'] = family_encoder.transform([df['family_encoded'].iloc[0]])[0]
df['typeholiday_encoded'] = typeholiday_encoder.transform([df['typeholiday_encoded'].iloc[0]])[0]
df['city_encoded'] = city_encoder.transform([df['city_encoded'].iloc[0]])[0]
df['state_encoded'] = state_encoder.transform([df['state_encoded'].iloc[0]])[0]
df['typestores_encoded'] = typestores_encoder.transform([df['typestores_encoded'].iloc[0]])[0]
```

Figure 3.6 Importing encoders and encoding categorical attributes

Once all the attributes were in a form that can be fed to the model, the model was loaded using Joblib and given the feature vector X (Figure 3.7). The model produced a prediction which was then sent to the homepage where it was displayed (Figure 3.8).

```
features = [
    'store_nbr',
    'onpromotion',
    'dcoilwtico',
    'cluster',
    'day_of_week',
    'day',
    'month',
    'year',
    'family_encoded',
    'typeholiday_encoded',
    'city_encoded',
    'state_encoded',
    'typestores_encoded',
]

X = df[features]

model = load('models/M10.joblib')
```

Figure 3.7 Creating feature vector X and importing the model using Joblib

```
prediction = model.predict(X)

return render_template(
    'index.html',
    prediction = prediction
)
```

Figure 3.8 Generating predictions using the XGBoost model

Figure 3.9 and 3.10 show a simple interface, which was improved later. The interface shows a dropdown list of all the listed products. The user can select the family (product category) and the model generates predicted sales for that family for the specified (hardcoded) store and the specified (hard coded) date.

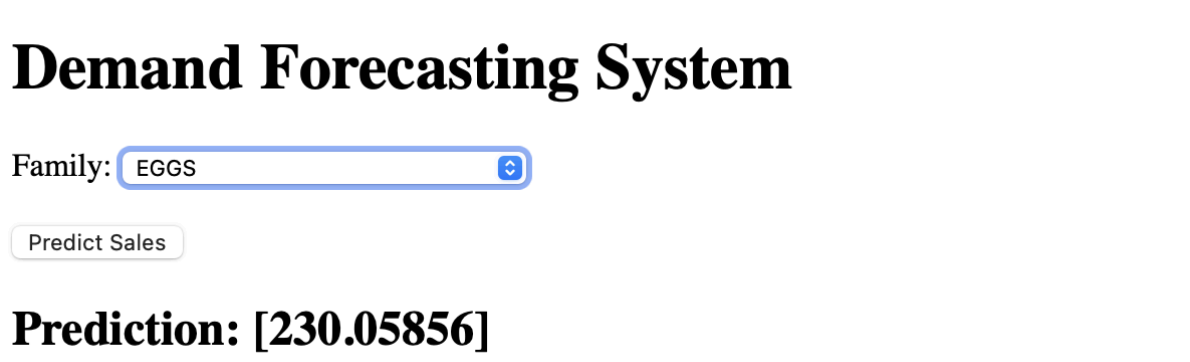

Figure 3.9 Prediction generated by the model for ‘eggs’ on the hardcoded store and date

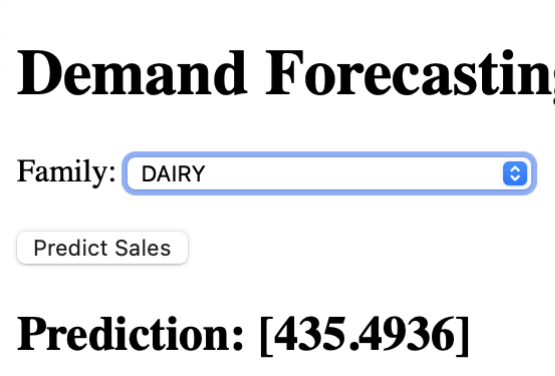

Figure 3.10 Prediction generated by the model for 'dairy' on the hardcoded store and date

For the demo, the users were given control over the selections dynamically, the data for holiday, oil pricing, and promotion was fetched in accordance to the date selected, and a side-by-side comparison was presented of the predicted sales and actual sales.

Bootstrap was used to improve the user experience visually and add functionality (Date Picker) as shown in Figure 3.11. The dates allowed to be selected are the dates from the test set. This was done so that we can show predicted values and actual values.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.9.0/js/bootstrap-datepicker.min.js"></script>
<script>
    $(document).ready(function(){
        $('.datepicker').datepicker({
            format: 'yyyy-mm-dd',
            startDate: '2016-09-14',
            endDate: '2017-08-14',
            autoclose: true
        });
    });
</script>
```

Figure 3.11 Date Picker

As discussed with Ma'am Huda, only 5 families (Beverages, Dairy, Frozen Foods, Meats, and Seafood) were selected for the demo (Figure 3.12).

```
<select class="form-control" id="family" name="family" required>
    <option value="BEVERAGES">BEVERAGES</option>
    <option value="DAIRY">DAIRY</option>
    <option value="FROZEN FOODS">FROZEN FOODS</option>
    <option value="MEATS">MEATS</option>
    <option value="SEAFOOD">SEAFOOD</option>
</select>
```

Figure 3.12 Shortlisted categories in the drop down list

To allow the user to select the date, the conversion from date to the extracted features we trained the model on had to be performed live (Figure 3.13).

```
if request.method == 'POST':
    df['date'] = request.form.get('datepicker')
    df['date'] = pd.to_datetime(df['date'])
    df['day_of_week'] = df['date'].dt.day_of_week
    df['day_of_week'] = df['day_of_week']+1
    df['day'] = df['date'].dt.day
    df['month'] = df['date'].dt.month
    df['year'] = df['date'].dt.year
```

Figure 3.13 Converting date into day_of_week, day, month, and year

In order to make it possible to fetch the holiday and oil price data for the selected date, a filtered dataset was created. The filter dataset only included dates from the test set, last 20% of the training data (Figure 3.14)

```
split_index = int(0.8 * train.shape[0])
```

```
filtered_train = train.iloc[split_index:]
```

```
filtered_train.shape
```

```
(600178, 10)
```

Figure 3.14 Filtering test dates

As discussed, the user was not allowed to select the store. So, for the sake of the demo, we selected store number 1 (Figure 3.15).

```
filtered_train = filtered_train[filtered_train['store_nbr'] == 1]
```

```
filtered_train.shape
```

```
(11088, 10)
```

```
filtered_train = filtered_train.drop(['store_nbr'], axis=1)
```

Figure 3.15 Filtering store number 1

The data was filtered for the shortlisted families (Figure 3.16) and inspected to ensure there were no issues.

```
filtered_train = filtered_train[(filtered_train['family'] == 'BEVERAGES') | (filtered_train['family'] == 'DAIRY')
```

Figure 3.16 Filtering shortlisted families

```
filtered_train.head()
```

	date	family	sales	onpromotion	typeholiday	dcoilwtico	day	month	year
2402139	2016-09-13	BEVERAGES	1942.000	0	NDay	44.91	13	9	2016
2402144	2016-09-13	DAIRY	703.000	0	NDay	44.91	13	9	2016
2402147	2016-09-13	FROZEN FOODS	95.000	0	NDay	44.91	13	9	2016
2402160	2016-09-13	MEATS	288.823	0	NDay	44.91	13	9	2016
2402168	2016-09-13	SEAFOOD	34.034	0	NDay	44.91	13	9	2016

```
filtered_train.tail()
```

	date	family	sales	onpromotion	typeholiday	dcoilwtico	day	month	year
2999109	2017-08-15	BEVERAGES	1942.000	11	Holiday	47.57	15	8	2017
2999114	2017-08-15	DAIRY	602.000	19	Holiday	47.57	15	8	2017
2999117	2017-08-15	FROZEN FOODS	89.000	1	Holiday	47.57	15	8	2017
2999130	2017-08-15	MEATS	274.176	0	Holiday	47.57	15	8	2017
2999138	2017-08-15	SEAFOOD	22.487	0	Holiday	47.57	15	8	2017

Figure 3.17 Inspecting the final dataset

In order to access the holiday, oil pricing, promotion, and actual sales data for the selected date and family, the filtered dataset is searched for the selected day, month, year, and family (Figure 3.18). As it would always return only one row, the data for holiday, oil pricing, and promotion is filled in the feature vector that is going to be used for the prediction. Actual sales are stored in a variable that is passed on to the html template (Figure 3.19).

```
filtered_df = pd.read_csv("data/filtered_train.csv")
filtered_df = filtered_df[(filtered_df['day'] == int(df['day'])) & (filtered_df['month'] == int(df['month'])) & (filtered_df['year'] == int(df['year']))]
filtered_df = filtered_df[(filtered_df['family'] == str(df['family'].iloc[0]))]
```

Figure 3.18 Filtering data based on day, month, year, and family

```
df['typeholiday'] = filtered_df['typeholiday'].iloc[0]
df['dcoilwtico'] = filtered_df['dcoilwtico'].iloc[0]
df['onpromotion'] = filtered_df['onpromotion'].iloc[0]
actual = round(filtered_df['sales'].iloc[0])
```

Figure 3.19 Collecting holiday, oil pricing, promotion, and actual sales data

The feature vector is passed on to the model as explained earlier and the prediction generated by the model along with the actual sales are sent to the home page where they are displayed (Figure 3.20).

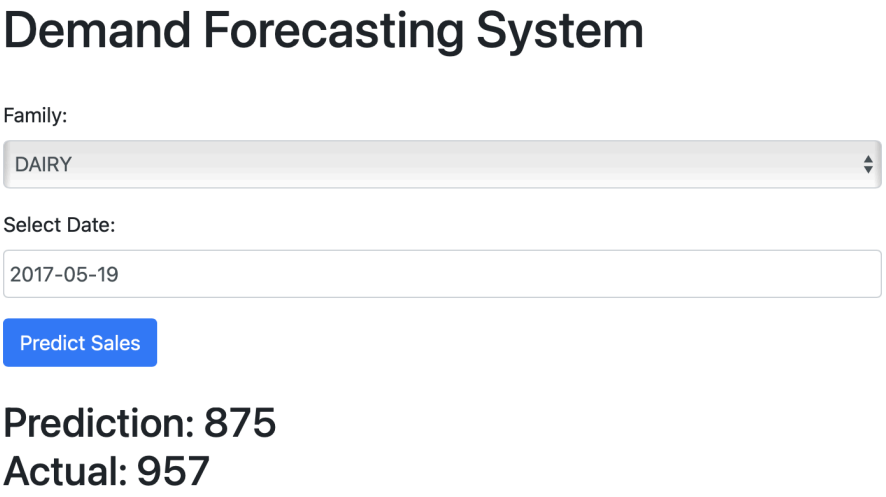

Figure 3.20 Flask Application

This concluded the development of the Flask application. The Flask application was then deployed on Digital Ocean (Figure 3.21). The demo of the ML model was given live on the cloud during the final presentation.

Deployment: https://hammerhead-app-6m6td.ondigitalocean.app/

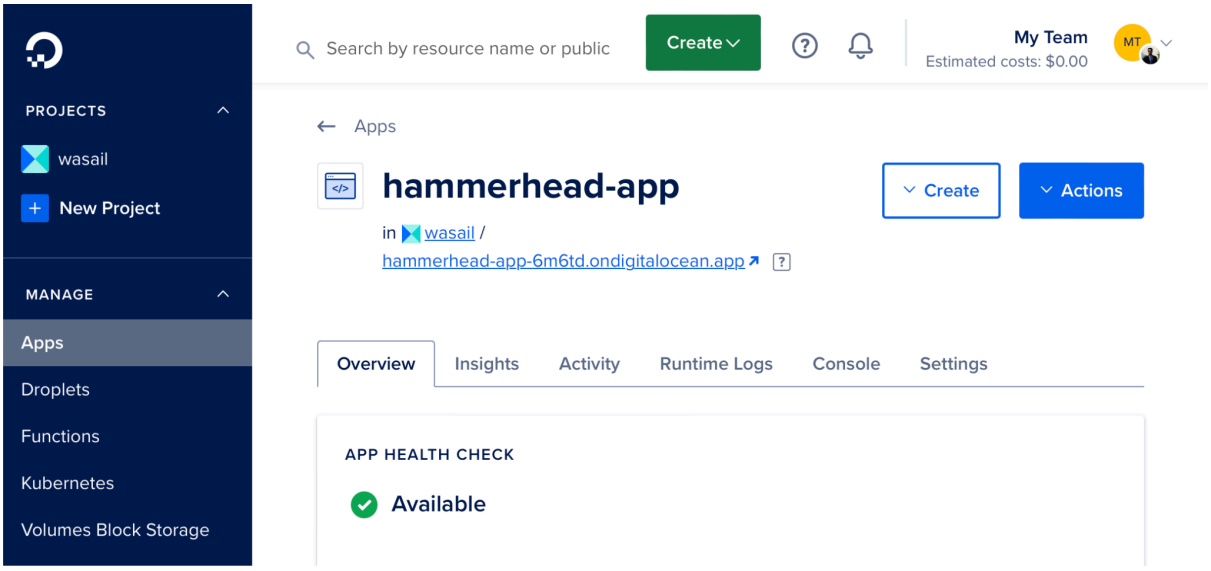

Figure 3.21 Deployment on Digital Ocean