FORECASTING RIDE REQUESTS BASED ON "OLA DATASET"

DOCUMENTATION

Mukul Malik
21ucs133@lnmiit.ac.in

1. *Problem Statement*

Given data of ride bookings for the year of 2020 (Ola Ride Dataset 2020), forecast the number of rides at a particular time in a particular region to optimize ride allotment.

The data is in the form: Customer ID, Timestamp, Latitude of Pickup, Longitude of Pickup, Latitude of Drop, Longitude of Drop

2. *Dataset Cleaning*

2.1 Basic Cleaning

Firstly, we remove duplicate entries. No customer can have two entries at the same timestamp. 1,13,540 entries will be removed post this.

```
df.drop_duplicates(subset=['ts','number'], inplace = True, keep = 'last')
df.reset_index(inplace = True, drop = True)
```

Next, we remove NULL records where customer ID doesn't exist. These are erroneous.

```
df['number'] = pd.to_numeric(df['number'], errors = 'coerce')
np.count_nonzero(df.isnull().values)
np.dropna(inplace = True)
```

Next, we coerce timestamps to Python DateTime!

```
df['number'] = pd.to_numeric(df['number'], errors = 'coerce', downcast='integer')
df['ts'] = pd.to_datetime(df['ts']
```

Next we can then break down time into its components for close analysis of time features, on an hourly basis for instance.

```
df['hour']          =   df['ts'].dt.hour
df['mins']          =   df['ts'].dt.minute
df['day']           =   df['ts'].dt.day
df['month']         =   df['ts'].dt.month
df['year']          =   df['ts'].dt.year
df['dayofweek']     =   df['ts'].dt.dayofweek
```

First four rows of our data will now look like this:

	ts	number	pick_lat	pick_lng	drop_lat	drop_lng	hour	mins	day	month	year	dayofweek
0	2020-03-26 07:07:17	14626	12.313621	76.658195	12.287301	76.602280	7	7	26	3	2020	3
1	2020-03-26 07:32:27	85490	12.943947	77.560745	12.954014	77.543770	7	32	26	3	2020	3
2	2020-03-26 07:36:44	5408	12.899603	77.587300	12.934780	77.569950	7	36	26	3	2020	3
3	2020-03-26 07:38:00	58940	12.918229	77.607544	12.968971	77.636375	7	38	26	3	2020	3

2.2 Business Logic Cleaning

There can be cases when a user requests a ride, and their booking request is logged in the database but this user re-books his/her ride due to longer wait hours or driver refused booking or user by mistake added wrong pickup or drop locations.

To make more logical sense out of the data, we will impose these rules:

1. Keep only one request of same user to same pickup latitude longitude in 1 hour time frame of first ride request.
2. Don't consider Pick Up and Drop Lat-Long Distance less than 50 meters = 0.05 kms; No user would like to ride for just 50meters trip.
3. Keep only last request of user within 8 mintues of first booking request.
4. Remove locations outside of Karnataka (we're not modelling interstate rides).

Let's first sort the values by customer ID and timestamp. So we get all orders of a customer together and sorted by timestamp.

```
df.sort_values(by = ['number','ts'], inplace = True)
df.reset_index(inplace = True)
```

Next, we keep the timestamp in a more usable fashion. Let's convert them into seconds since epoch.

```
df['booking_timestamp'] = df.ts.values.astype(np.int64) // 10**9
```

Note: `df.ts.values` *returns a numpy array of the ts values.*

Now, in order to process our cleaning rules, for each entry (row) that is now sorted by timestamp and user, we require previous booking's timestamp within it. For example, suppose we have 5 records for customer ID = 1 sorted by timestamp. With each row, we want to include the previous row's timestamp. For the first entry there is no previous row, so we can set its previous timestamp to 0 (NaN). Thus, we perform the shift operation after grouping the data on customer IDs to individually do this for all customer IDs.

```
df['shift_booking_ts'] = df.groupby('number')['booking_timestamp'].shift(1)
df['shift_booking_ts'].fillna(0, inplace = True)
```

Now, for each request, we can calculate the time before a previous request by the same user in hours and minutes! This will help us in processing our business rules.

```
df['booking_time_diff_hr'] = round((df['booking_timestamp'] - df['shift_booking_ts'])//3600)
df['booking_time_diff_min'] = round((df['booking_timestamp'] - df['shift_booking_ts'])//60)
```

2.2.1 Handling Case 1: Re-booking Again to Same Location within 1 hour by Same User

```
df = df[~((df.duplicated(subset=['number','pick_lat','pick_lng'],keep=False)) &
(df.booking_time_diff_hr<=1))]
```

2.2.2 Handling Case 2: One user Books rides are different lat-long within 8 mins Time (Ride Time + Driver Arrival Time) [Fraud/Human Error]

```
df = df[(df.booking_time_diff_min >= 8)]
```

2.2.3 Removing Ride Request Less Than 0.05 miles = 50 meters

To do this, we will need to calculate the distance between the source and destination using their latitudes and longitudes. The Earth is a spherical surface. The shortest distance between two points is called the geodesic. Using latitudes and longitudes, the distance can be calculated using tensor calculus which is summarized by study of geodesics. The python library **geopy** implements this.

```
def geodestic_distance(pick_lat, pick_lng, drop_lat, drop_lng):
    return round(geodesic((pick_lat, pick_lng), (drop_lat, drop_lng)).miles*1.60934,2)

df['geodesic_distance'] =
np.vectorize(geodestic_distance)(df['pick_lat'],df['pick_lng'],df['drop_lat'],df['drop_lng'
])

df = df[df.geodesic_distance > 0.05]
```

2.2.4 Removing Ride Request with Destination or Source Outside of Karnataka

We note the boundaries of location boundaries of the region.

```
outside_KA = df[(df.pick_lat <= 6.2325274) | (df.pick_lat >= 35.6745457) | (df.pick_lng <=
68.1113787) | (df.pick_lng >= 97.395561) | (df.drop_lat <= 6.2325274) | (df.drop_lat >=
35.6745457) | (df.drop_lng <= 68.1113787) | (df.drop_lng >= 97.395561)]

df = df[~df.index.isin(outside_KA.index)].reset_index(drop = True)
```

Now, our data is clean. Number of Good Ride Requests: **3,708,329**

3. *Feature Engineering*

In this section, we now have clean data from which we need to extract useful information. Currently, we have time stamp of booking, pickup and drop locations with customer ID. Can we organize this information in a better way? Can we engineer better features?

3.1 Clustering with Mini Batch K-Means

There's so many latitudes and longitudes here. What if we classify them into regions. We could model our forecasting around the regions instead of exact latitudes and longitudes. This compacts the information into more useful features. Since our dataset is huge, we'll choose Mini Batch K-Means.

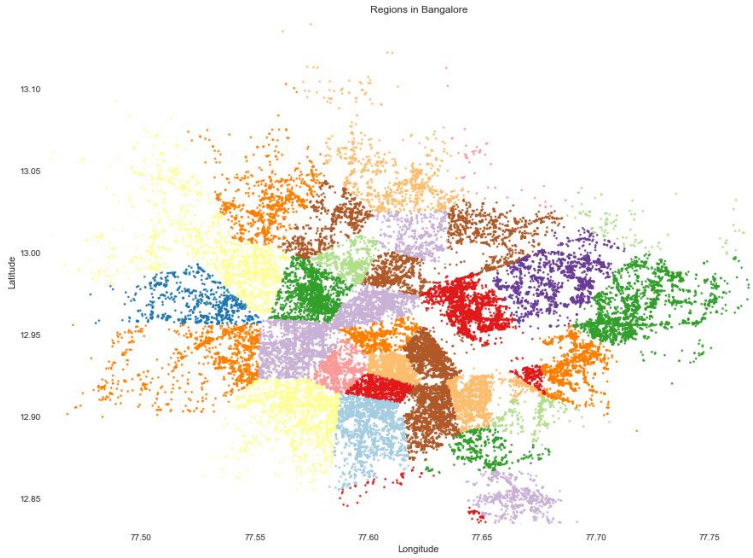
To choose the number of regions/clusters, we'll define a rule. The regions/clusters should be at max 0.5 miles apart for it to be an effective clustering. To find this number, we will try making 10 regions, then 20 ... up to 100 to see which one is forming the best clusters.

```
def makingRegions(noOfRegions):
     regions = MiniBatchKMeans(n_clusters = noOfRegions, batch_size = 10000,
random_state = 5).fit(coord)
     regionCenters = regions.cluster_centers_ totalClusters = len(regionCenters)
     return regionCenters, totalClusters

for i in range(10, 100, 10):
     regionCenters, totalClusters = makingRegions(i)
     min_distance(regionCenters, totalClusters)
```

This will show that when number of clusters is 50 then the maximum distance between the regions is 0.5 miles.

```
coord = df[["pick_lat", "pick_lng"]].values
regions = MiniBatchKMeans(n_clusters = 50, batch_size = 10000, random_state =
0).fit(coord)
df["pickup_cluster"] = regions.predict(df[["pick_lat", "pick_lng"]])
```



3.2 Temporal Feature Engineering

Well, our timestamps give information about the number of rides at a particular region. To predict number of ride request in a particular area, we need to present our data in a format that shows number of rides at a particular time in a particular region.

A good place to begin would be rounding off timestamps near to 30 minute intervals (7AM, 7.30AM, 8AM …). And then we can aggregate over these timestamps to find the number of rides per region per timestamp.

```
def round_timestamp_30interval(x):
    if type(x)==str:
        x = datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
    return x- timedelta(minutes = x.minute%30, seconds = x.second, microseconds =
x.microsecond)
```

```
df['ts'] = np.vectorize(round_timestamp_30interval)(df['ts'])
dataset=dataset.groupby(by = ['ts','pickup_cluster']).count().reset_index()
dataset.columns = ['ts','pickup_cluster','request_count']
```

A sample of our data after doing this:

	ts	pickup_cluster	request_count
0	2020-03-26 01:00:00	16	1
1	2020-03-26 01:00:00	28	3
2	2020-03-26 01:30:00	28	1
3	2020-03-26 01:30:00	41	2
4	2020-03-26 02:00:00	28	1

There are 366 days in 2020 so with 50 regions and 48 timestamps, there would be 366*50*48 = **8,78,400 data rows**. But some timestamps won't exist (no rides), so we'll add value of 0 as ride requests on those time stamps.

We'll need a bit of a complex transformation to achieve this. Here's how we can do this:

1. Create each possible timestamp and allot it region number -1 (dummy value) and ride count of 0. Append this to existing data.
2. Set index to timestamp and unstack the data on pickup clusters. Replace NaN values with 0 (no rides).
3. Now stack the data back and reset the index.
4. Sort data by cluster number.
5. Remove pickup cluster -1 rows to remove the dummy data we created.

Creating Dummy Data to Ensure We Have All Timestamps:

```
l = [datetime(2020,3,26,00,00,00) + timedelta(minutes = 30 * i) for i in range(0,
48 * 365)]
lt = []
for x in l:
    lt.append([x, -1, 0])
temp = pd.DataFrame(lt, columns = ['ts','pickup_cluster','request_count'])
dataset = dataset.append(temp,ignore_index=True)
```

Transforming Data:

```
data = dataset.set_index(['ts', 'pickup_cluster']).unstack().fillna(value=0)
.asfreq(freq='30Min').stack().sort_index(level=1).reset_index()
data = data[data.pickup_cluster >= 0]
```

At this point, we'll have 8,74,400 rows.

We can now add more temporal features to increase information.

```
data['mins']       = data.ts.dt.minute
data['hour']       = data.ts.dt.hour
data['day']        = data.ts.dt.day
data['month']      = data.ts.dt.month
data['dayofweek']  = data.ts.dt.dayofweek
data['quarter']    = data.ts.dt.quarter
```

Features like hour of the day, day of week, month and quarter may tell you a lot about the number of rides.

4. *Model Training*

AND FURTHER FEATURE OPTIMIZATION

AIM: To forecast demand for a given latitude-longitude.

Metric: RMSE, how close we are able to predict ride demand to true value

We can use models like Decision Trees, or boosting/bagging classifiers easily here. Let's try with Random Forrest (Bagging) and XGBoost (Boosting).

This is a regression problem.

Sample of data:

	ts	pickup_cluster	mins	hour	month	quarter	dayofweek	request_count
0	2020-03-26 00:00:00	0	0	0	3	1	3	0
1	2020-03-26 00:30:00	0	30	0	3	1	3	0
2	2020-03-26 01:00:00	0	0	1	3	1	3	0
3	2020-03-26 01:30:00	0	30	1	3	1	3	0
4	2020-03-26 02:00:00	0	0	2	3	1	3	0
...	...	...	...	...	...	...	...	...
878395	2021-03-26 21:30:00	49	30	21	3	1	4	11
878396	2021-03-26 22:00:00	49	0	22	3	1	4	13
878397	2021-03-26 22:30:00	49	30	22	3	1	4	13
878398	2021-03-26 23:00:00	49	0	23	3	1	4	10
878399	2021-03-26 23:30:00	49	30	23	3	1	4	7

4.1 Splitting into Train and Test

We can choose first 24 days of a month into train set and rest of the days into test set. Other approaches include Incremental Training (Tuning the model incrementally).

```
df_train = df[df.ts.dt.day <= 23]
df_test  = df[df.ts.dt.day > 23]
```

```
X      = df_train.iloc[:,1:-1]
y      = df_train.iloc[:,-1]
X_test = df_test.iloc[:,1:-1]
y_test = df_test.iloc[:,-1]
```

4.2 Random Forrest Regressor

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators = 300, random_state = 42, n_jobs =
-1, verbose=True)
regressor.fit(X,y)
```

```
feature_importances = pd.DataFrame(regressor.feature_importances_,mindex =
X.columns, columns=['importance']).sort_values('importance',ascending=False)
```

	importance
pickup_cluster	0.461499
hour	0.252366
month	0.118886
quarter	0.077498
dayofweek	0.074334
mins	0.015416

```
RMSE TRAIN: 1.9343965753661254, RMSE TEST: 4.439438294448992
```

We clearly see that the pickup region has a strong impact on number of rides. Then the hour of the day also has an important impact. Quarter isn't really that impactful thus this shows there isn't much impact of **seasonality** in our data.

4.3 XGBoost Regressor

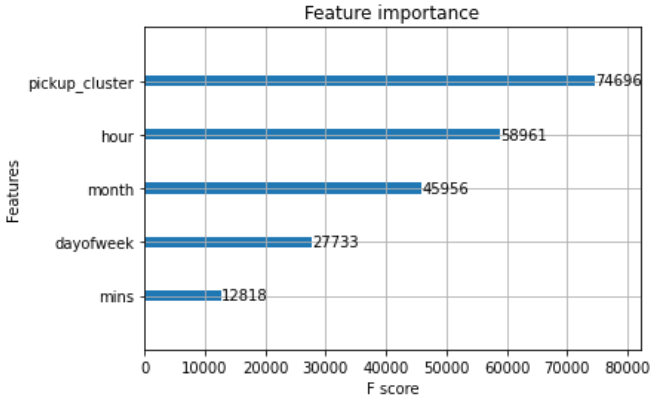
```
import xgboost as xgb
```

```
model=xgb.XGBRegressor(learning_rate=0.01, random_state=0, n_estimators=1000,
max_depth=8, objective="reg:squarederror")
eval_set = [(X_test, y_test)]
model.fit(X,y,verbose=True, eval_set=eval_set,
early_stopping_rounds=15,eval_metric="rmse")
```

```
print("Model Score:", model.score(X,y))
```

RMSE TRAIN: 2.6808759094911205, RMSE TEST:4.326231614641142

This shows that random forrest is overfitting. XGBoost is performing better here.



4.4 Boosting Results with Time Series Analysis

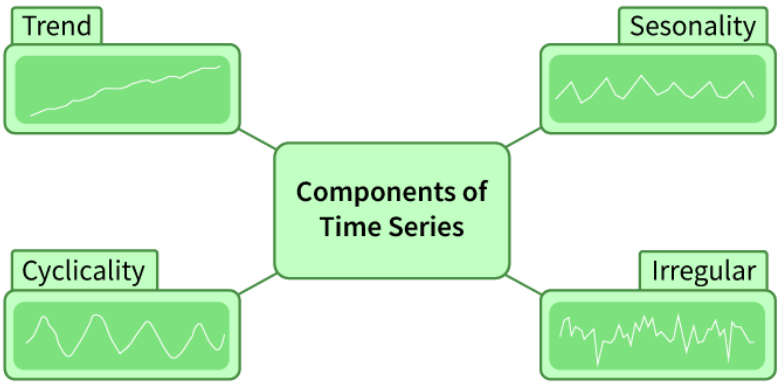
Since our data is time-series based, let's look at tools that help us mine information from time series datasets.

Time series analysis examines data collected at regular intervals over a defined timespan. This method focuses on studying patterns in sequentially ordered observations, rather than analyzing sporadic or arbitrarily gathered data points. By consistently recording measurements at set time intervals, analysts can identify trends, cycles, and other temporal characteristics within the dataset.

Components of a Time Series:

We can see a time series as composition of following:

- **Trend**: In which there is no fixed interval and any divergence within the given dataset is a continuous timeline. The trend would be Negative or Positive or Null Trend.
- **Seasonality**: In which regular or fixed interval shifts within the dataset in a continuous timeline. Would be bell curve or saw tooth.
- **Cyclical**: In which there is no fixed interval, uncertainty in movement and its pattern.
- **Irregularity**: Unexpected situations/events/scenarios and spikes in a short time span.



We may try to model these components individually wherein we split these components using techniques like local regression or differencing to perform the splits. Other techniques include modelling lagged features and average lags.

Autoregression in time series states that the current value of a time series is dependent on previous values in the time series. These are called the "lagged features". This is more relevant to our problem statement.

Suppose, the ride requests in a region at Timestamp 7.30 AM is 5, at 7AM is 4 and 6.30 AM is 6. So, the first lagged feature of 7.30 AM is value at 7AM (4) and second lagged feature equals 6.

Speaking of averages, we also provide the average of previous k lagged values.

But how many lagged values to incorporate?

This is answered by **Autocorrelation** (ACF) and **Partial Autocorrelation** (PACF).

Auto-Correlation Function (ACF)

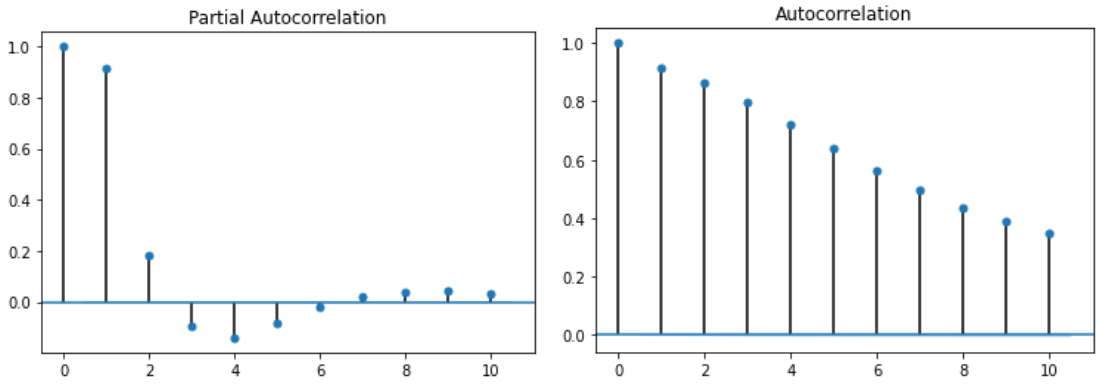
ACF tells us how much a given value at a given time is "influenced" by previous k values in the time series. Choose the value of k such that the influence is maximized. The formula for the same is pretty intuitive.

$\rho(k) = Cov(X_t, X_{t-k}) \;/\; [\sigma(X_t) \cdot \sigma(X_{t-k})]$

Partial Auto-Correlation (PACF)

Partial Auto-Correlation tells us how much the value at current time stamp is influenced by the ***kth*** lagged value. ACF measures influence of previous k values combined but PACF measures the standalone influence of ***kth*** previous value.

```
plot_acf(df_train['request_count'], lags=10)
plot_pacf(df_train['request_count'], lags=10)
```



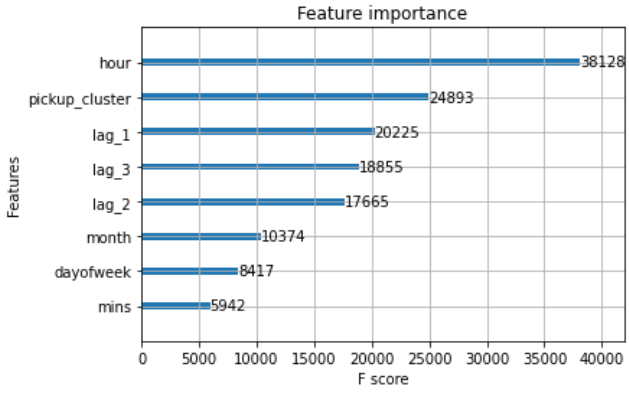
This shows that the past 3 values are really important when discussing the current value. **In other words, the trend in previous 1.5 hours impacts current value.**

So, we will add 3 lagged features to each row.

Upon using XGBoost with same configuration again, new results obtained are:

RMSE TRAIN: 2.4530548280046203, RMSE TEST:2.518347268292163

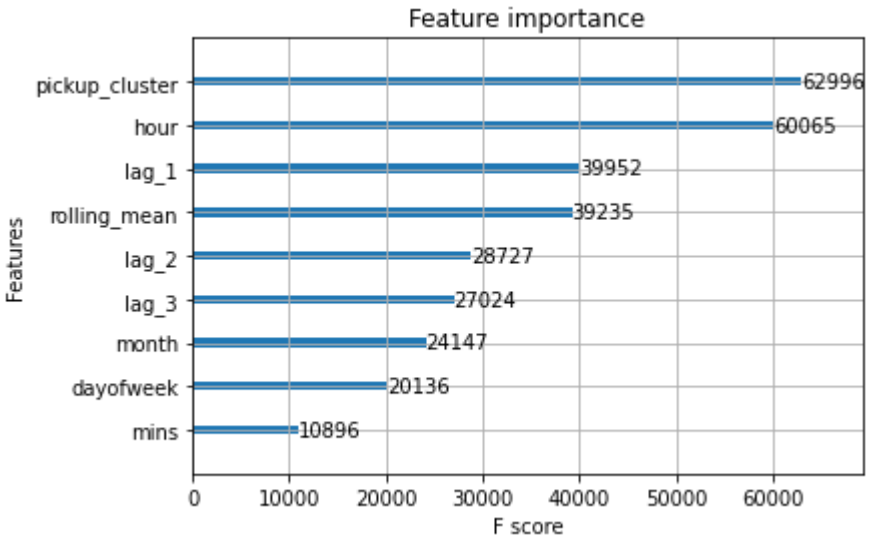
Notice the huge improvement!



Let's now include mean of previous 3 lagged values to further boost crucial information.

New scores obtained are:

RMSE TRAIN: 2.2876961962667974, RMSE TEST:2.4566977172952758



Code and more text is available in the notebooks (4). **Joblib** is used to save the models to disk.

Final features become :

['pickup_cluster', 'mins', 'hour', 'month', 'dayofweek', 'quarter', 'lag_1', 'lag_2', 'lag_3', 'rolling_mean']

RMSE Test: 2.28

RMSE Train: 2.45

5. *Future Scope*

Time Series data is sequential in nature thus allowing for recurrent Deep Learning techniques to be applied like RNNs, LSTMs and GRUs. They can be trained on GPUs on the same data.

More train and test splitting methods can be tried on.