<u>Solution Approach</u>

Step 1: Setup completion to run the pipeline completely

- Ran "***make -f Makefile***" command
- Ran "***make run***" command
- Added a dummy column "***accept_freq***" and populated it with some random value in "***driver_historical_completed_bookings***" function of "***transformations.py***". Just to run the pipeline.
- Added the "***accuracy***" metric in "***evaluate***" function of "***classifier.py***". Just to run the pipeline, no logic behind adding accuracy metric for judging the performance of model.
- By making the above changes, I was able to run the pipeline and generate the file in submission folder.

Step 2: Change in *driver_historical_completed_bookings* to add a column with values of drivers acceptance history.

- Idea was to get the percentage of orders accepted out of all sent.
- Selected only the ACCEPTED,REJECTED and IGNORED related rows.
- For current_order_id of a driver, filtered the corresponding previous ordered data based on current timestamp. Then calculated the percentage of acceptance by dividing total accepted orders divided by total order request sent.
- One row operations are given below.

current_time	driver_id	current_order_id	Status
2:00 AM	d1	120	Accepeted
3:00 AM	d1	121	Rejected
1:00 AM	d1	122	Rejected
5:00 AM	d2	123	Accepeted
6:00 AM	d2	124	Accepeted

Current_time data			all prev data for that driver		
current_time	driver_id	current_order_id	prev_time	prev_order_id	Prev Status
3:00 AM	d1	121	1:00 AM	122	Rejected
3:00 AM	d1	121	2:00 AM	120	Accepeted

results

current_time	driver_id	current_order_id	prev accepted	prev req sent	accept_freq
3:00 AM	d1	121	1	2	0.5

- For the first order of the driver, I am filling it with 0.4(40% acceptance percentage). Ideally change is 50% for both acceptance and rejection but to start with, given 0.4. If the driver performs well in its first few orders it will improve.
- Since no historical data is present in the test data so added the ***try and except*** block to run this script only in training cases.
- Adding ***try-except block*** is a bit risky. Just did it for assignment completion purposes. Better we should pass a parameter in ***apply_feature_engineering*** function that will let us know whether it is a train or test transformation.
- If I need to do it in production for live cases. I would have integrated this logic with ETL so that for every driver accept_freq will get updated at the end of the day and we will use it for the next day.
- Performing these historical operations is a time taking task and this we should do in our sql ETL script.
- Better do it for the last 10 orders or last one month orders. For example how many he accepted out of the last 10 requests sent to him.

Step 3: Training the base model

- Filtered the resulted data (Accepted,Ignored and Rejected) for training purposes.
- Used the ROC_AUC_SCORE metrics instead of Accuracy because the data was highly imbalanced for training. Only ~18k cases are present of not-acceptance in training data.

```
df.is_completed.value_counts()
```

```
is_completed
1    180703
0     18782
Name: count, dtype: int64
```

- With the initial parameters mentioned in the config file I got the below mentioned results on the unseen data. It's predicting almost every record as 1 (acceptance). We definitely need to improve the model performance.

```
[[   54  3736]
 [   93 36014]]
              precision    recall  f1-score   support

           0       0.37      0.01      0.03      3790
           1       0.91      1.00      0.95     36107

    accuracy                           0.90     39897
   macro avg       0.64      0.51      0.49     39897
weighted avg       0.85      0.90      0.86     39897

Accuracy: 0.9040278717698073
Recall: 0.997424322153599
Precision: 0.9060125786163522
AUC/ROC: 0.5058361716308892
```

Step 4: Hyperparameter tuning

- From initial observations we are already aware that the data was highly imbalanced. So definitely we need to take care of that somehow. To work upon this, I have used the "***class_weight"*** parameter,I am setting it to "***balanced***". This means the classes will be weighted inversely proportional to how frequently they appear in the data.
- I have used the "***RandomizedSearchCV"*** to find the best parameters with 5 fold validation.
- Final Parameters used are: n_estimators=250,max_depth=15,n_jobs=-1,random_state=33,bootstrap=false, class_weight="balanced"
- Final model performance on unseen data is mentioned below. ROC_AUC_SCORE has improved from ~0.50 to ~0.61 and also started getting the balanced results of rejection and acceptance both.

```
[[ 1827  1951]
 [ 9591 26528]]
              precision    recall  f1-score   support

           0       0.16      0.48      0.24      3778
           1       0.93      0.73      0.82     36119

    accuracy                           0.71     39897
   macro avg       0.55      0.61      0.53     39897
weighted avg       0.86      0.71      0.77     39897

Accuracy: 0.7107050655437752
Recall: 0.7344610869625405
Precision: 0.9314933810878191
AUC/ROC: 0.6090251437988987
```

Step 5: Further improvement thoughts

- We should definitely include the drivers acceptance percentage of last x orders.
- We should include additional information.
    - Average distance of orders accepted/rejected in the last x days: Few drivers do not like to go too far so they reject it.
    - Average monetary value of rejected and accepted order: Accept only orders of high amount.
    - Ratings
    - Current Weather condition - Cloudy, Raining etc
- Some enhancements on execution level
    - Numbers of drivers available under 2km radius: We should definitely send the request if very few drivers are available even if the model is telling he will reject.
    - A/B testing before deployment to enhance it further and cover the corner cases.