BIKE COUNT PREDICTION PROJECT

Report - December 2021

First, we **explored the dataset**, then we thought about the **most relevant additional features** to feed our model. We then looked for the best way to **preprocess** the chosen data and for **suitable regressors**. We tested several of them. Finally, we deepened the **tuning of the three regressors** that proved to be the most efficient: XG Boost, Catboost and LightGBM.

EXPLORING THE DATASET

The first two parts intertwined, and we went back and forth as we added more features. We explore here the dataset with all the features.

After getting the dataset, we needed to find which features we should keep.
So, we transformed the dates, added weather data and additional features (lockdown and curfew) and then used the *Pandas Profiling* module, which is very useful for data exploration.

Thanks to it we could observe which features had a significant, positive or negative, correlation on the target value: *log_bike_count*. The features with the highest squared correlation are *counter_name*, *hour,* and *curfew* [1.1]. Although correlation does not necessarily indicate causality, we can safely say that these variables are worth keeping because log_*bike_count* is logically correlated with *counter_name* (busier lanes), time of day (more traffic at peak hours) and curfew (prohibition of traffic over a long period of the dataset).

The data frame profiling also enabled us to worry about missing values. As 8.8 % of the *log_bike_count* values are zero[1.2]. Although these are not necessarily outliers, as it is possible that no bikes passed in a street for two hours, we decided to check by plotting the *log_bike_count* of counters that had many zero.

It turns out that two of the fifty-six counters were indicating zero for a long time [1.3]. Two possible explanations, a breakdown or public work on the bike path.

By visualizing the scatter plots like the one *hour/log_bike_count*, it also gave us ideas of features engineering that we will develop in the preprocessing part of the report[1.4].

Finally, we also decided to keep variables such as temperature (*t*) and humidity (*u*) which are slightly correlated with the *log_bike_count* and which improve our model.

ADDING FEATURES

Considering the permutation importance plot analysis [2.1], we wanted to add new features which would better fit the high volatility and seasonality of the *log_bike_count* variable.

First, by plotting one of the main counters [2.2], we realized that both confinements due to Covid-19 might have had an impact on the *bike_count*. Thus, we decided to add a binary feature *'confinement'* to our training set. We encoded this variable using a function with a datetime mask and adding it to our final pipeline [2.3]. This feature had a dramatic effect on our RMSE improving it by almost 0.4 (with XG Boost).

We also tried to add a binary feature accounting for the French public holidays. However, it only added complexity without improving our overall precision, so we did not use it.

Because we have had such good results on the confinement feature, we decided to further explore the restrictions linked to Covid-19, by plotting the count on a day with and without curfew [2.4], we realized that adding a binary variable *'curfew'* might have a positive impact on the accuracy of our model. We encoded it using a ColumnTransformer and added it at the beginning of our pipeline [2.5]. This once again had good results on the accuracy of our model.

We also explored the features of the '*external_data.csv*' file. We plotted a permutation importance graph with our models on the features [2.6]. It is thus observed that the most important data among the meteorological data are t, tx12, u, tend24, rr24 and td. We can therefore choose to add to the model the data that have an impact on the model but are not directly correlated to the temperature: u, tend24 and rr24.

Finally we tried to explore other databases like opendata.paris.fr but we couldn't find data that was very relevant and easy to merge with our training dataset so we didn't go any further.

PREPROCESSING

We explored several ways of preprocessing our data. For the categorical data: i.e., counter name, we stayed with a OneHotEncoder.

The main issue was the date columns. First, we tried another way of encoding the dates: by considering the first date as date_0, and then computing the difference between the date and this date_0 [3.1]. By fitting a numerical encoder to this new date column, we had worse results than by using the original date_encoder function so we

stayed with this one. We then tried several categorical encoders and found we had better results with an OrdinalEncoder than a OneHotEncoder.

For the numerical columns (t, u, rr24), we had good results using a StandardScaler.

Finally, we explored a bit of feature engineering on one of our date columns. By visualizing the scatter plot of *hour/log_bike_count* [1.4], we thought that the variable hour and log_bike_count might have a quadratic relation, so we preprocessed the column 'hour' using a PolynomialFeature with degree two [3.2].

TESTING MODELS

We tried seven types of regressors and kept the three most efficient for our dataset: **XGBoost, Catboost** and **LightGBM** (all tree-based).

First, we will talk about a failure: Prophet.

We tried to implement **Prophet**, a regression model specialized in Time Series problems, but we quickly realized that this was not going to work four our dataset as Prophet is relevant when the target value is mostly conditioned by the TimeSeries, but in our project the *counter_name* is central.
Moreover, if Prophet allows via *'add regressor'* to add numerical features, it does not allow categorical features.

We tried Lasso and Ridge regression with parameter tuning but we did not get very good results, so we proceeded to tree-based regressors.

After testing **RandomForest** which was slow for only correct results, we decided to turn to faster tree-based methods.

First **LightGBM**, it first allowed us to solve the time issue with RandomForest. This is due to the construction of the algorithm which is 'leaf_wise' (and not 'level-wise'). The split is made based on the contribution to the global loss and not only the loss on the branch. We also read that it was well-adapted to large datasets (+ 10 thousand rows). The first test, before tuning, validated our expectations since we obtained an RMSE of 0.770 (local test), a score that we had not reached with the previous models.

Second, we also tried another library for the Gradient Boosting method with **XGBoost**, which we read often got good results for regression prediction problems; we had very good results using a XGBoostRegressor model. Before tuning, we had a very similar score of 0.78 without any parameter tuning.

Third, **CatBoost** because we read that it was suitable for categorical data (stands for Category Boosting) and that it was particularly fast for datasets with more than a hundred thousand rows. CatBoost seems to be the most efficient of the three without any parameter tuning as the RMSE score without parameter tuning was 0.748.

Finally, we tried to implement a **deep neural network model** thanks to TensorFlow and Keras. The RMSE results on the neural network validation sets are quite difficult to interpret as they are really different according to the different folds. If they perform better than the LGBM and XGBOOST on some folds, with the activation function ('elu') on the two hidden layers, they are much less efficient on other folds. Having tested this model last, we were unable to interpret these variations.

TUNING HYPERPARAMETERS

Since an exhaustive *GridSearch* is excessively time consuming to run locally, we tried two methods to tune our models.

Firstly, after having read an interesting documentation [5.1] on the most important parameters of the model, we tried to group the parameters influencing the same aspect of the model. This was done by trial and error to try to get a frame for each key parameter.

For the LightGBM, we grouped the hyperparameters affecting the structure of the trees (*max_depth, min_data_in_leaf, num_leaves*). Then those allowed to improve the accuracy (*learning_rate, n_estimators, max_bin*) and finally, the parameters controlling the overfitting *(lambda_1, lambda_2, min_gain_to_split*).

We had the same approach for the XGBoost, by first finding the optimal number of estimators, then tuning the structure parameters (*max_depth, min_child_weight*), then *gamma*, *colsample_bytree* and *subsample*, then finishing with regularization parameters to avoid overfitting (l*ambda and alpha*).

The second approach we tried, apparently less methodical, was to perform a RandomizedGridSearch on the key hyperparameters, each associated with a rather large panel of values (values consistent with the parameter in question [5.2]). For the LGBM, we set the number of iterations to 200 (190 minutes).

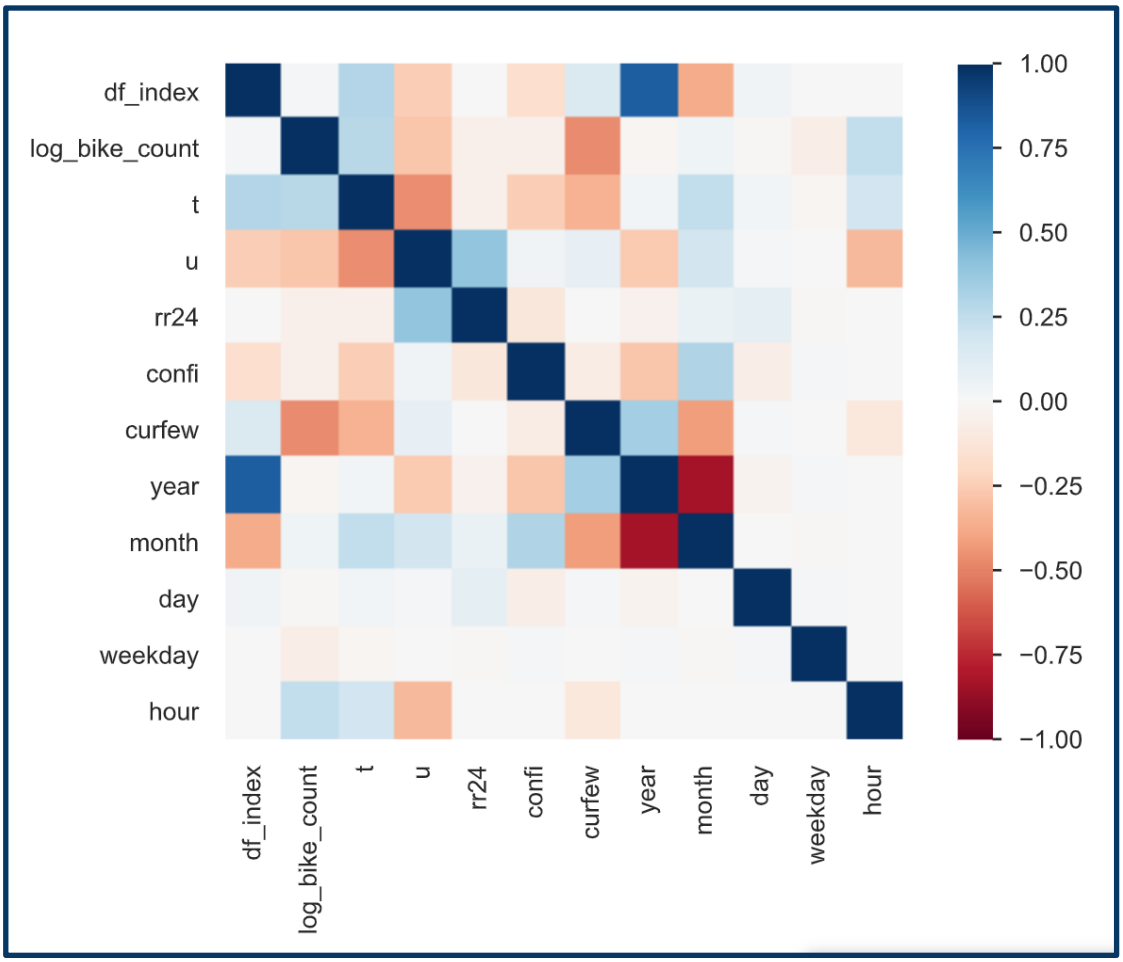
This method proved to be efficient because it allowed us to obtain the best tuning of our LightGBM with a local RMSE of 0.738 (0.750 on RAMP [5.3]).

For the XGBoost, we also had better results using a RandomizedGridSearch with a final local RMSE of 0.703 on the validation set (0.741 on RAMP [5.4]).

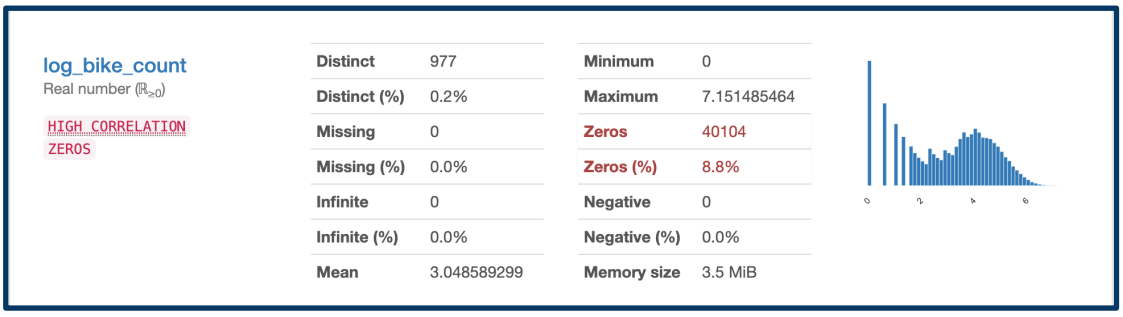
Same method for Catboost but with fewer iterations as they were longer. Though, we only get an improvement of 0.006 points of RMSE thanks to the parameter tuning (0.741).

APPENDIX

Appendix 1.1 :



Appendix 1.2 :



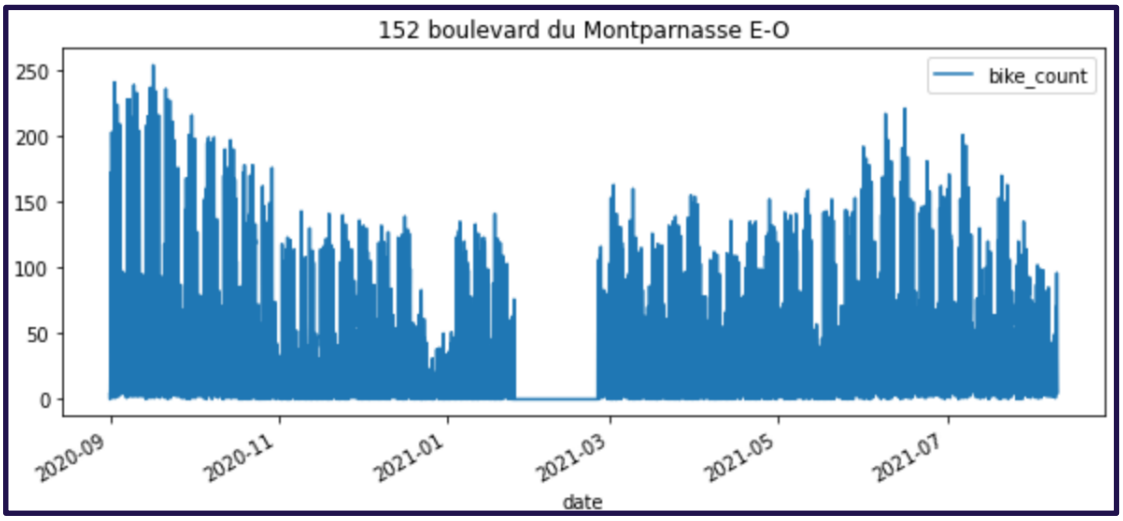
log_bike_count

Real number ($\mathbb{R}_{\geq 0}$)

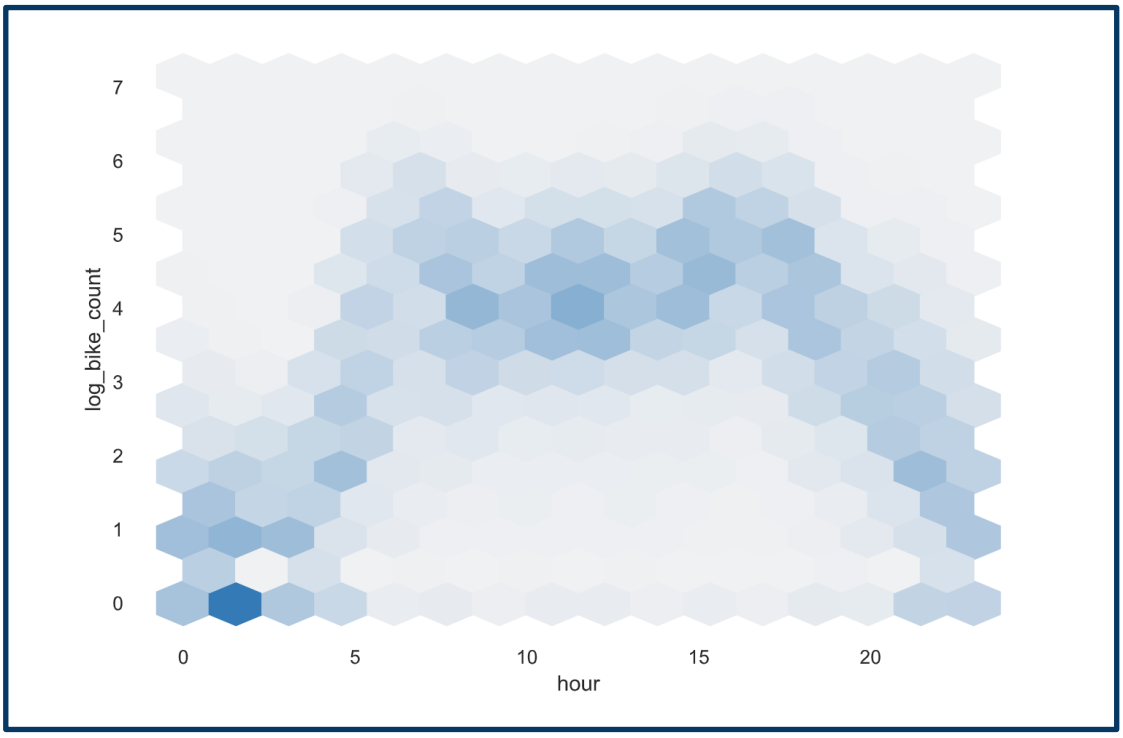
HIGH CORRELATION
ZEROS

Distinct	977	**Minimum**	0
Distinct (%)	0.2%	**Maximum**	7.151485464
Missing	0	**Zeros**	40104
Missing (%)	0.0%	**Zeros (%)**	8.8%
Infinite	0	**Negative**	0
Infinite (%)	0.0%	**Negative (%)**	0.0%
Mean	3.048589299	**Memory size**	3.5 MiB

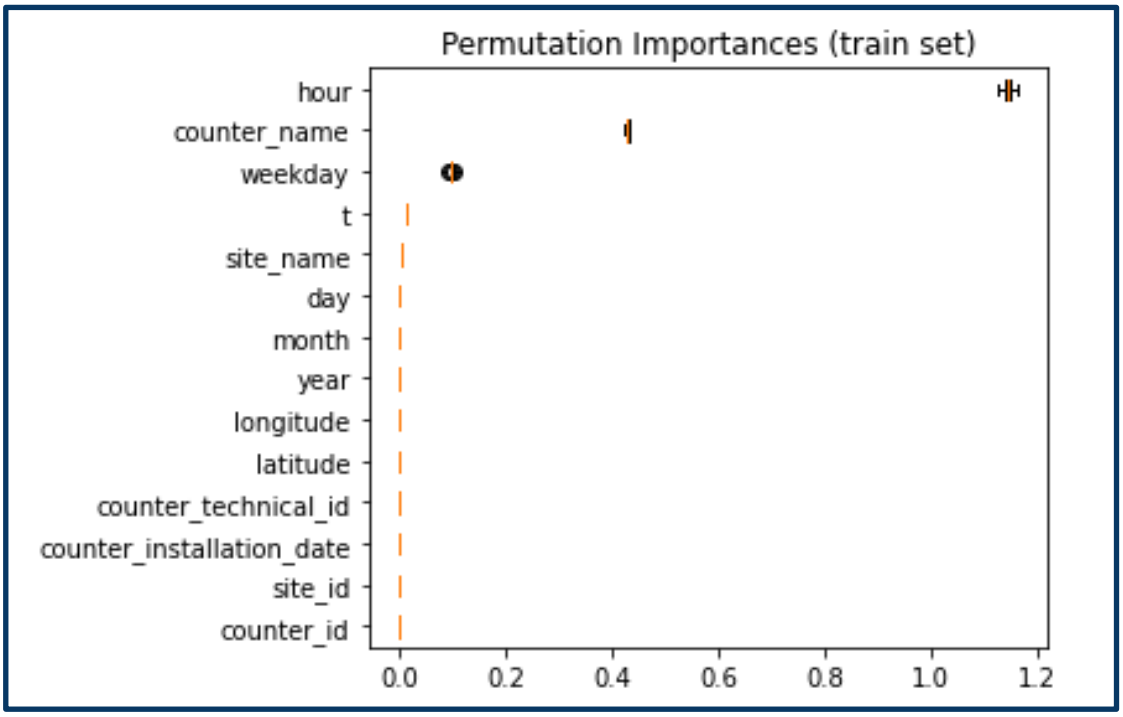
Appendix 1.3 :



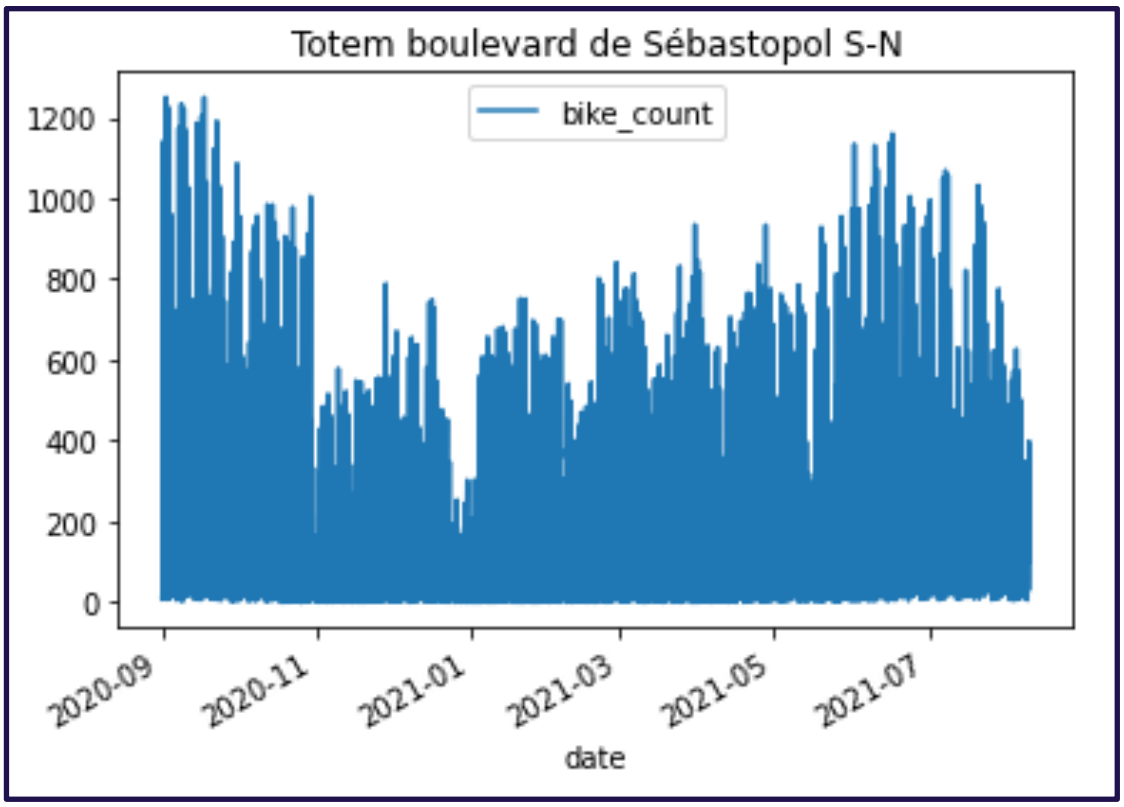
Appendix 1.4 :



Appendix 2.1 :



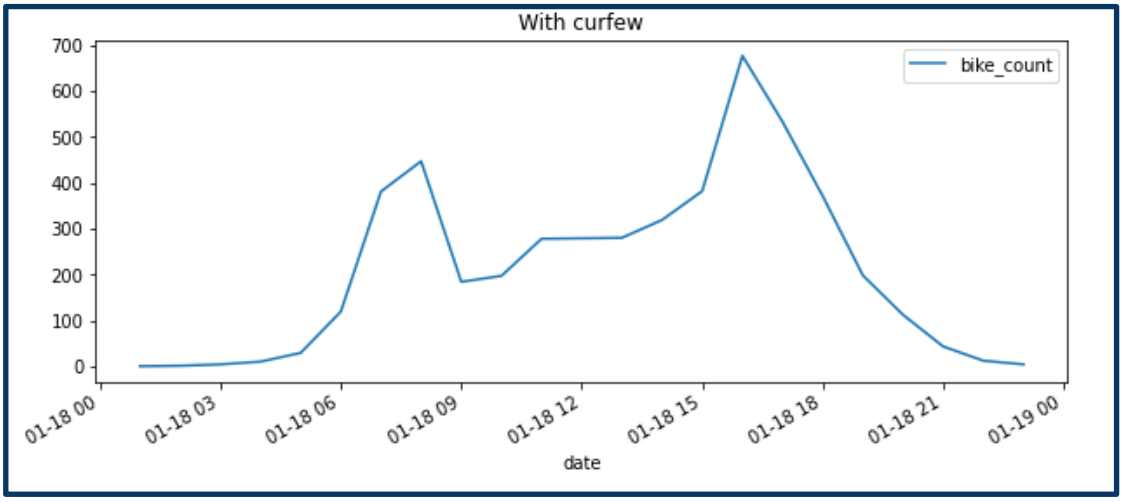
Appendix 2.2 :



Appendix 2.3 :

```
def confinement(X):

    date = pd.to_datetime(X['date'])
    X.loc[:, ['date_only']] = date
    new_date = [dt.date() for dt in X['date_only']]
    X.loc[:, ['date_only']] = new_date
    mask = ((X['date_only'] >= pd.to_datetime('2020/10/30'))
        & (X['date_only'] <= pd.to_datetime('2020/12/15'))
        | (X['date_only'] >= pd.to_datetime('2021/04/03'))
        & (X['date_only'] <= pd.to_datetime('2021/05/03')))
    X['confi'] =np.where(mask, 1, 0)
    return X.drop(columns=['date_only'])
```

Appendix 2.4 :



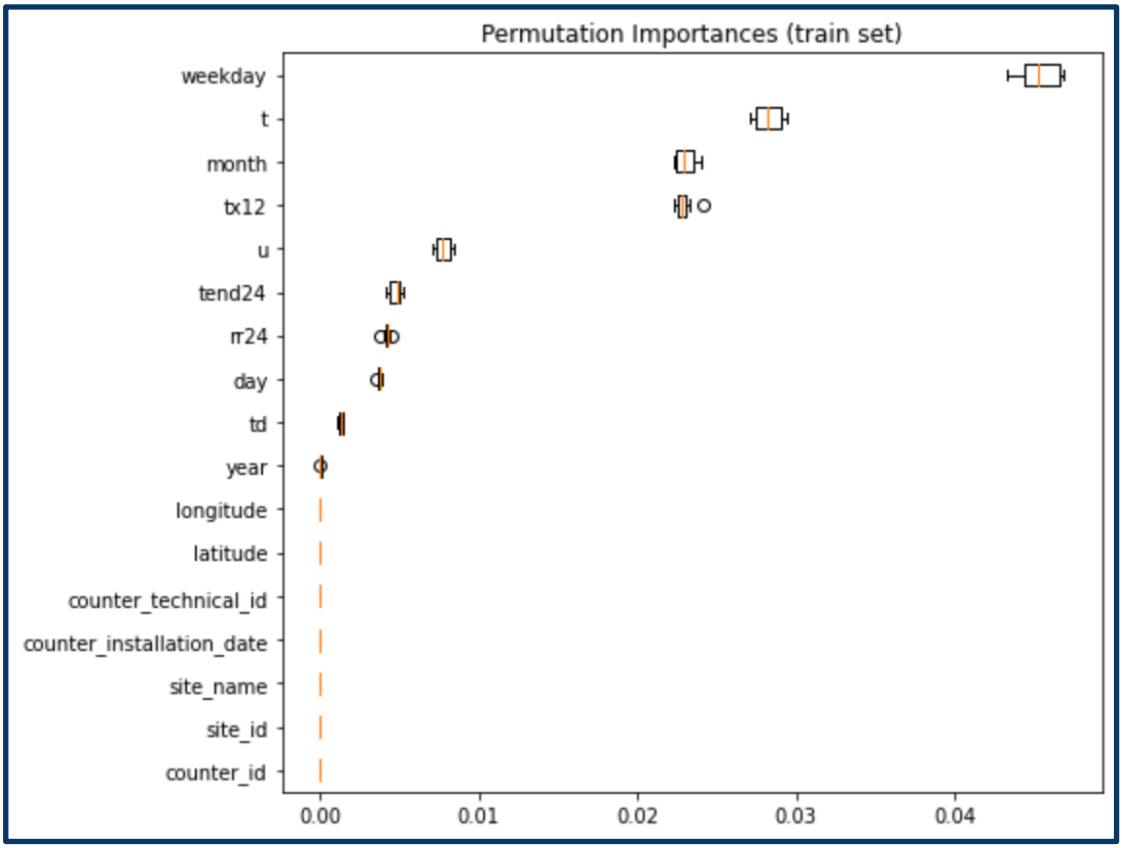
Appendix 2.5 :

```
def curfew(X):
    date = pd.to_datetime(X['date'])
    X.loc[:, ['date_only']] = date
    new_date = [dt.date() for dt in X['date_only']]
    X.loc[:, ['date_only']] = new_date
    X.loc[:, ['hour_only']] = date
    new_hour = [dt.hour for dt in X['hour_only']]
    X.loc[:, ['hour_only']] = new_hour
    mask = (
        #First curfew
```

```
        (X['date_only'] >= pd.to_datetime('2020/12/15'))
        & (X['date_only'] < pd.to_datetime('2021/01/16'))
        & ((X['hour_only'] >= 20) | (X['hour_only'] <= 6))

        |

        # Second curfew
        (X['date_only'] >= pd.to_datetime('2021/01/16'))
        & (X['date_only'] < pd.to_datetime('2021/03/20'))
        & ((X['hour_only'] >= 18) | (X['hour_only'] <= 6))

        |

        # Third curfew
        (X['date_only'] >= pd.to_datetime('2021/03/20'))
        & (X['date_only'] < pd.to_datetime('2021/05/19'))
        & ((X['hour_only'] >= 19) | (X['hour_only'] <= 6))

        |

        # Fourth curfew
        (X['date_only'] >= pd.to_datetime('2021/05/19'))
        & (X['date_only'] < pd.to_datetime('2021/06/9'))
        & ((X['hour_only'] >= 21) | (X['hour_only'] <= 6))

        |

        # Fifth curfew
        (X['date_only'] >= pd.to_datetime('2021/06/9'))
        & (X['date_only'] < pd.to_datetime('2021/06/20'))
        & ((X['hour_only'] >= 21) | (X['hour_only'] <= 6))
        )
    X['curfew'] =np.where(mask, 1, 0)

    return X.drop(columns=['hour_only', 'date_only'])

def _encode_dates(X):
    X = X.copy()  # modify a copy of X
    # Encode the date information from the DateOfDeparture columns
    X.loc[:, "year"] = X["date"].dt.year
    X.loc[:, "month"] = X["date"].dt.month
    X.loc[:, "day"] = X["date"].dt.day
    X.loc[:, "weekday"] = X["date"].dt.weekday
    X.loc[:, "hour"] = X["date"].dt.hour
```

```
    # Finally we can drop the original columns from the dataframe
    return X.drop(columns=["date"])
```

Appendix 2.6 :



Appendix 3.1 :

```
def _encode_dates_2(X):
    X = X.copy()  # modify a copy of X
    # Encode the date information from the DateOfDeparture columns
    X.loc[:, 'new_time'] = X['date'] - min(data['date'])
    X.loc[:, '#nb_of_hours'] = X.loc[:,'new_time'].dt.seconds//3600


    # Finally we can drop the original columns from the dataframe
    return X.drop(columns=["date",'new_time'])
```

Appendix 3.2 :

```
preprocessor = ColumnTransformer([
        ('date', OrdinalEncoder(), date_cols),
```

```
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_cols),
        ('numeric', StandardScaler(), numeric_cols),
        ('period', 'passthrough', period_cols),
        ('hour', PolynomialFeatures(degree=2), hour_col)
        ]
    )
```

Appendix 5.1 :

https://www.kaggle.com/bextuychiev/lgbm-optuna-hyperparameter-tuning-w-understanding (first part of the article, tuned without Optuna)

Appendix 5.2 :

```
params = {

'lgbmregressor__learning_rate' : [0.01, 0.02, 0.03, 0.04, 0.05, 0.08,         0.1,
0.2, 0.3, 0.4],
'lgbmregressor__n_estimators' : [100, 200, 300, 400, 500, 600, 800, 1000, 1500,
2000],
'lgbmregressor__num_leaves': sp_randint(6, 50),
'lgbmregressor__min_child_samples': sp_randint(100, 500),
'lgbmregressor__min_child_weight': [1e-5, 1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3, 1e4],
'lgbmregressor__subsample': [0.2, 0.5, 0.7, 0.9],
'lgbmregressor__max_depth': [-1, 1, 2, 3, 4, 5, 6, 7],
'lgbmregressor__colsample_bytree': [0.2, 0.5, 0.7, 0.9],
'lgbmregressor__reg_alpha': [0, 1e-1, 1, 2, 5, 7, 10, 50, 100],
'lgbmregressor__reg_lambda': [0, 1e-1, 1, 5, 10, 20, 50, 100]

}
```

Appendix 5.3 :

Appendix 5.4 :