RAMP PROJECT - FINAL REPORT

Sarah Mayer and Camille Epitalon-de Guidis - December 2021

Exploring the dataset

The main part of the dataset exploration is done in the starting_kit file. There is one uncovered topic we have dealt with: how to treat the out-of-order counters (counted 0 bikes over several following days).

By plotting the number of bikes by day over the year for each counter, we discovered that 8 of them had a huge number of "0" values in the ***bike_count*** and ***log_bike_count*** features. More intriguing, those null values would last over several following days (Fig. 1) in the training paquet.

Fig.1 - *log_bikes_count* per day for 20 Avenue de Clichy NO-SE

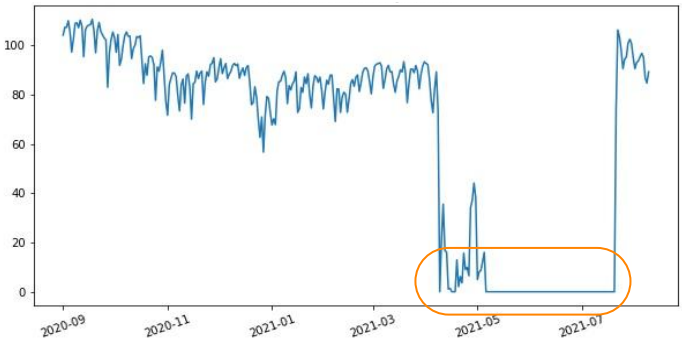


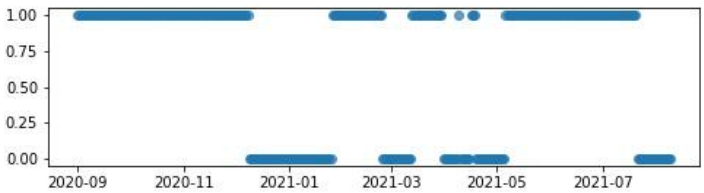
Fig.2 - Number of days without counting bikes by counter

Counter	Days with missing values
152 boulevard du Montparnasse E-O	29
152 boulevard du Montparnasse O-E	29
20 Avenue de Clichy NO-SE	80
20 Avenue de Clichy SE-NO	79
254 rue de Vaugirard NE-SO	99
254 rue de Vaugirard SO-NE	99
Voie Georges Pompidou NE-SO	18
Voie Georges Pompidou SO-NE	18

On closer inspection, we realized that this phenomenon was not marginal. Indeed, over the 343 days present in the training paquet, 66% of them have at least one counter with ***bike_count*** set to 0 during the whole day. The graph below (Fig. 3) illustrates the breakdown between days with all counters working (y=1) and days with at least one counter with null values.

We interpreted them as missing data rather than "0 bike counted per day" which makes no sense in Paris. And the test paquet does not contain any.

Fig.3 - Days with (1) and without (0) missing values



We thus decided to create a first model to approximate the actual bikes circulation on these counters during the bugging period in order to enrich the training dataset and therefore to improve the precision of the end prediction model.

Preliminary model to estimate null values

First, we had to reformat the data to well predict the missing values: 1 row per date and hour with the *log_bike_count* for each counter (Fig. 4). Then, we looped over our the bugging counter to:

1. split the dataset between rows with unjustified null values and other ones
2. train a Ridge regression model
3. predict the missing values.

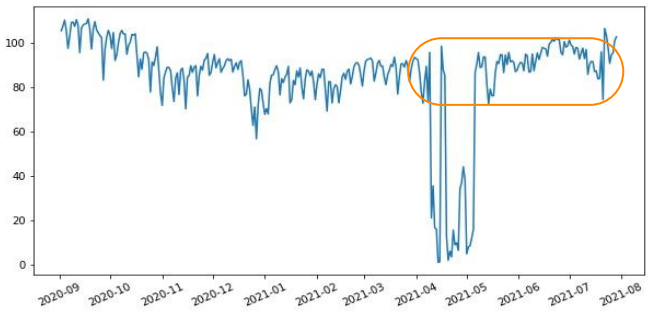
Fig.4 - Reformatted training dataset for preliminary prediction

year	month	day	weekday	hour	log_0	log_1	...	log_46	log_47	log_48	lo
2020	9	1	1	1	1.609438	1.386294	...	0.693147	2.708050	1.098612	2.39
2020	9	1	1	2	0.000000	0.693147	...	1.791759	1.791759	1.609438	2.07
2020	9	1	1	3	1.098612	1.945910	...	2.197225	2.639057	2.995732	2.30

We chose a Ridge regression model at random at first, just to see where it would lead us. Just visually, the results looked quite satisfying (Fig. 5) compared to Fig. 1.

As we were starting to challenge the model and tune the parameters, we realized that the response variable (log_bike_count) was not available in the dataset passed in the ramp project.

Fig.5 - Simulated *log_bikes_count* per day for Clichy NO-SE



Pretty hard then to train our model without this feature. After discussing it with Roman Yurchak in a ramp session, we decided to drop this idea.

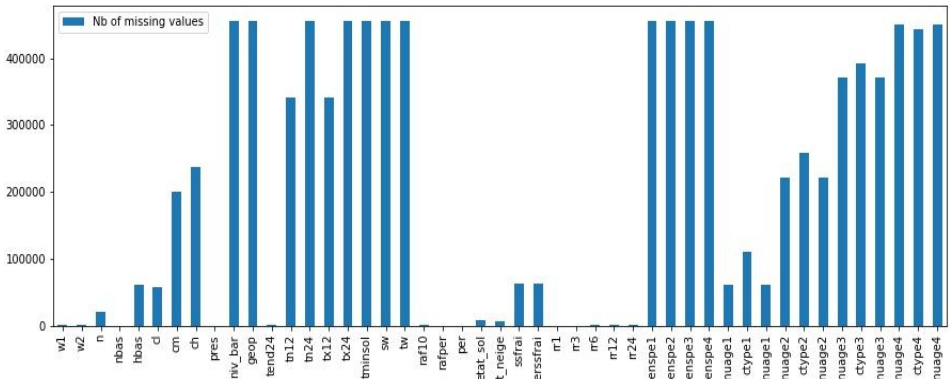
Merging external data

The external_data dataset

From external_data, we first decided to discard all variables with too many missing data (Fig. 6).

Then, after having split the *date* feature in different variables (year, month, day, hour…), we explored the feature importance.

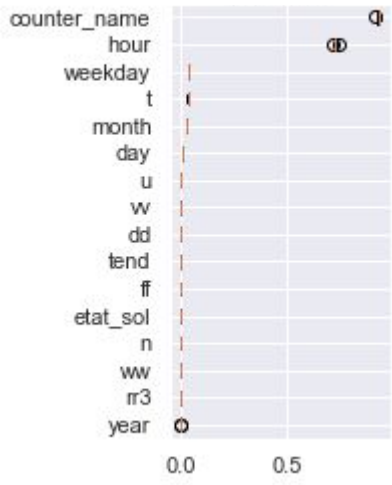
Fig. 6 - Number of missing values for each weather feature



Considering the graph (Fig. 7) and the meaning of each variable, we first decided to only keep the temperature and humidity features (ie. 't' and 'u').

Later in our work, on different models, we tried to add other variables in our final model and some of them ended up in a slight improve of the RMSE ('rr3' for instance).

Fig. 7 - Feature importance in our model



The two packages to spot holidays in Paris

We decided to import the packages **SchoolHolidayDates** and **JoursFeries** as the cyclist behaviour may change on those typical days. After testing the importance of these new feature, we decided to eliminate them. As an element of interpretation, we thought they may not be representative of the trend, as thus don't increase our prediction (cf doc 1).

COVID impact

Intuitively, we reckon adding the quarantine and curfews biases to our model would be valuable as those COVID measures did impact our daily behaviours. After looking for packages, we decided to manually generate lockdown and curfews periods and add them to our training dataset (cf. doc 2). Same as with the holidays, adding those variables did not turn out to improve our model score. We chose not to add them. You can still find it in our python file, in the function *merge_covid_data()*.

Data preprocessing

Encoding choices

As said previously, we decided early to split the *date* feature in separate variables (*year*, *month*, *day*, *weekday*, *hour*). Actually, we asked our Regression professor (Katia) about using a Time Series model (such as Sarimax) and she advised us against it (very hard to tune, not very performant when many other features). How to encode them, then?

Well first, when using linear regression models, we had decided to OneHotEncode them as they could be considered as categorical factors. It did look really promising and as we soon opted for a non linear model (a tree-based one), OneHotEncoding them was a mistake. We rather used a StandardScaler for all the numeric features (ie temperature, humidity and hour, day, weekday, month and year), even if the impact of the scaling is very light on the final RMSE.

Regarding the categorical variables (*counter_name*, *site_name*), we decided to use a OneHotEncoder, based on our results (better than OrdinalEncoder).

Model selection

Initially, we decided to test the whole set of models we had already used in our different courses, to discard the poorest ones and keep the ones with the best RMSE (using cross-validation) and training time (both being computed on a Jupyter notebook).

Model	RMSE	Training time
Ridge Regressor	1.55± 0.122	1.471
Random Forest Regressor	1.24± 0.229	259.222
Lasso	1.62± 0.105	1.694
XGBoost	1.18± 0.22	61.186
ElasticNet	1.61± 0.106	1.313
HistGradientBoostingRegressor	1.16± 0.23	19.022
LightGBM	1.16± 0.226	14.292

But fairly quickly, we understood that most machine learning models do not make statistical sense with time-series data since we assume our data are independent which doesn't hold in time series.

What's more, plotting the pairplot and the "profile_report" of the 7 variables we had kept (cf. doc 3) showed that the linearity between the different feature was very questionable (except maybe between temperature and humidity). That's why we decided to discard all linear models and to opt for regression trees.

After some research and many tests, we identified 3 models to best predict these time series:

Prophet
Well fitted when time-series is the central factor. However, "countername" being the most important factor, this model was not appropriated.

XGBoost
This model is well appropriated for datasets composed of a mixture of categorical and numeric features.

LightGBM
As we are in the case of a non linear model (as shown previously), those models seemed appropriated.

After many iterations, we got our best results with LightGBM and decided to stick to it.

Parameter tuning

Tuning the parameters turned out to be a very time consuming part of our work, the idea being to increase the accuracy of the model while preventing high variance or overfitting problems.

With the parameters set by default, the model provides the RMSE showed on Fig. 8. Using *GridSearchCV*, we started by tuning *num_leaves*, *max_depth*, *min_data_in_leaf*. Then we tuned other ones (*reg_alpha*, *min_child_samples*, etc).

Fig. 8 - Results before parameter tuning

```
----------------------------
Mean CV scores
----------------------------
       score             rmse          time
       train  0.457 ± 0.0271  2.3 ± 1.25
       valid   0.898 ± 0.186  4.0 ± 0.69
       test   0.757 ± 0.1188  0.5 ± 0.17
----------------------------
Bagged scores
----------------------------
       score   rmse
       valid  0.917
       test   0.706
```

```
params3 = {
    'colsample_bytree': 0.7,
    'learning_rate': 0.01,
    'max_depth': 11,
    'min_child_samples': 198,
    'min_child_weight': 0.1,
    'n_estimators': 2000,
    'num_leaves': 99,
    'reg_alpha': 1,
    'reg_lambda': 0.1,
    'subsample': 0.5
}
```

Fig. 9 - Final set of parameters

By raising the hyperparameter n_estimators and num_leaves, we drastically reduced the RMSE of the model, but it has cost: the training and validation time both increased.

The best set we found is the one on the left (Fig. 9).

Feature engineering

After some research and discussions with our professors, we decided to test some feature engineering on the date/hour variable, adding polynomial and trigonometric transforms (cf doc 4).

Cosinus & sinus transformations of dates

The encode the time-related variables (ie hour, month, day) with cos and sin, in order to represent the time circularity and to avoid considering time as linear. However, as we chose the LightGBM model, and it builds its split rules according to one feature at a time. This means that this model fails to process these two features simultaneously whereas the cos/sin values are expected to be considered as one single coordinates system.

Polynomial combinations

Similarly, we tried to implement polynomial feature engineering, raising the date/hour features to the power of 2 and 3 and to represent the interaction between features multiplying the variables together. After some iterations, we failed in improving the RMSE and decided not to use them.

Results

The final model is available on the ramp platform, under the submission name: “pantac_sur_tarmac19” and the team “ceptln”.

It gives the following outputs:

- RMSE: 0.731
- Train time: 161.27 sec
- Validation time: 374.28 sec

```
----------------------------
Mean CV scores
----------------------------
       score             rmse          time
       train  0.378 ± 0.0123  14.0 ± 6.12
       valid  0.723 ± 0.1268  14.1 ± 1.37
       test   0.651 ± 0.0691   1.4 ± 0.13
----------------------------
Bagged scores
----------------------------
       score   rmse
       valid  0.734
       test   0.560
```

Appendix

Doc 1: SchoolHolidayDates and JoursFeries modules

```
def _encode_dates(X):
    d = SchoolHolidayDates()
    holidays_list = list(set(list(d.holidays_for_year_and_zone(2020, 'C').keys()) +
list(d.holidays_for_year_and_zone(2021, 'C').keys())))
    public_holiday = list(JoursFeries.for_year(2020).values()) +
list(JoursFeries.for_year(2021).values())
    X = X.copy()
    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
    X.loc[:, 'holiday'] = pd.to_datetime(X['date']).dt.date.isin(holidays_list).astype(int)
    X.loc[:, 'public_holiday'] = pd.to_datetime(X['date']).dt.date.isin(public_holiday).astype(int)
    X= X.drop(columns=["date"])
    return X
```

RAMP PROJECT - FINAL REPORT

Sarah Mayer and Camille Epitalon-de Guidis - December 2021

Appendix

Doc 2: Confinement

```
def merge_covid_data(X):

    '''
    Enriches the input dataframe with quarantine and curfew data (merges on 'date')

    Parameters:
        X (pd.DataFrame): the dataframe to enrich

    Returns:
        X (pd.DataFrame): the initial dataframe
    '''

    # Create 2 columns: date_only and hour_only
    date_and_time = pd.to_datetime(X['date'])
    X['date_only'] = date_and_time
    new_date = [dt.date() for dt in X['date_only']]
    X['date_only'] = new_date
    X['hour_only'] = date_and_time
    new_hour = [dt.hour for dt in X['hour_only']]
    X['hour_only'] = new_hour

    # Create a mask to spot the days under quarantine in Paris
    is_quarantine = (
        (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'))
    )

    # Create a mask to spot the days under curfew in Paris
    is_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)) |
        (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)) |
        (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)) |
        (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))
    )

    # Create the associated columns
    X['quarantine'] = np.where(is_quarantine, 1, 0)
    X['curfew'] = np.where(is_curfew, 1, 0)

    # Clean the new dataframe and return it
    X.drop(columns=['hour_only', 'date_only'], inplace=True)
    return X
```

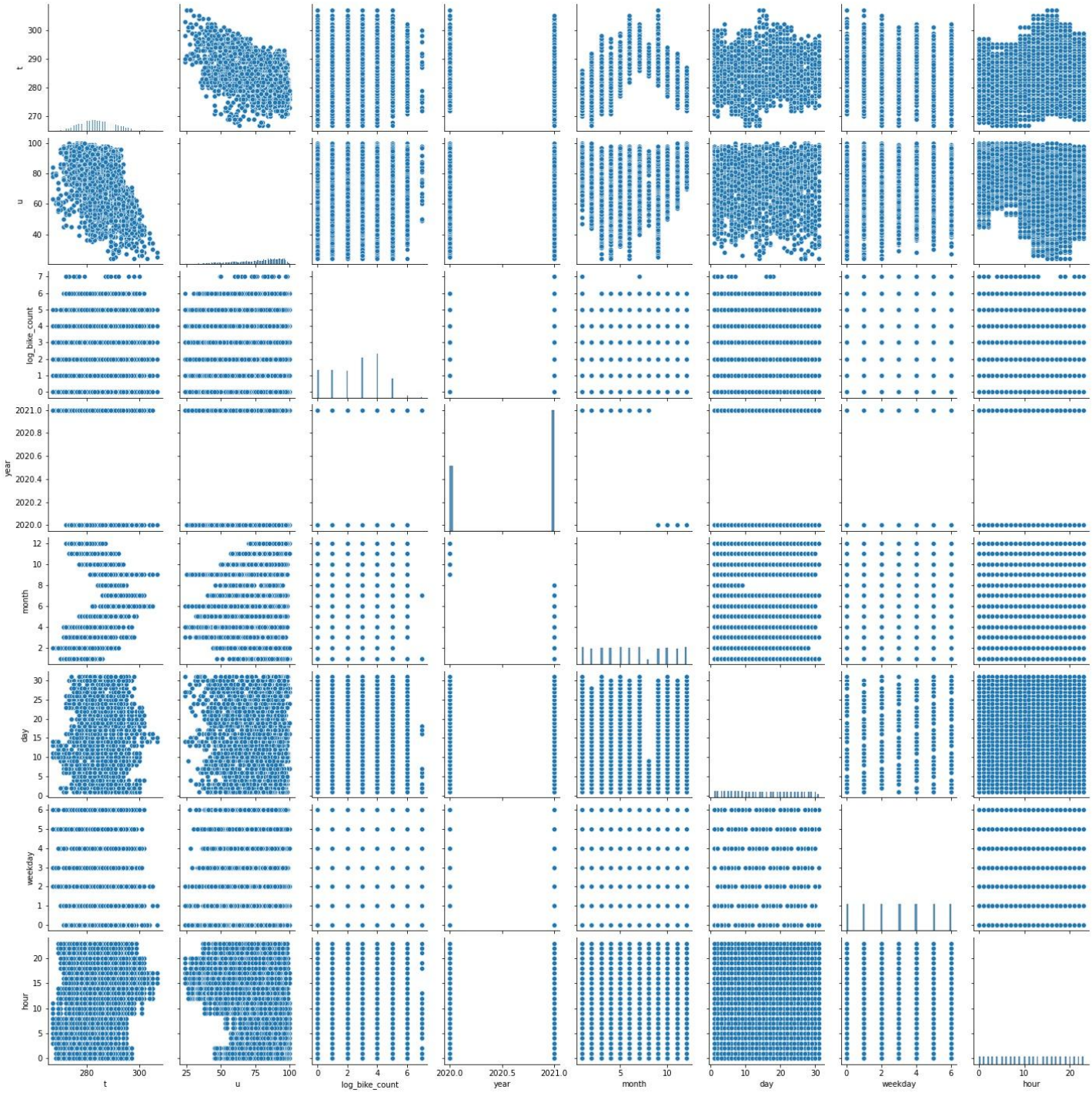
RAMP PROJECT - FINAL REPORT

Sarah Mayer and Camille Epitalon-de Guidis - December 2021

Appendix

Doc 3: Pairplot to check linearity assumption

```
import seaborn as sns
sns.pairplot(data=datatest)
plt.savefig("linearity.jpg")
```



RAMP PROJECT - FINAL REPORT

Sarah Mayer and Camille Epitalon-de Guidis - December 2021

Appendix

Doc 4: Polynomial combinations - feature engineering

```
def _encode_dates_poly(X):
    X = X.copy()

    # Create new columns with date parts from X.date
    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

    # Clean new X dataframe
    X = X.drop(columns=["date"])

    # Adding polynomial features from date variables
    numeric_encoder = StandardScaler()
    X_encoded = numeric_encoder.fit_transform(X[['month', 'day', 'weekday', 'hour']])
    X_encoded = pd.DataFrame(X_encoded, columns=['month', 'day', 'weekday', 'hour'])

    poly_encoder = PolynomialFeatures(degree=3)
    dates_poly = poly_encoder.fit_transform(X_encoded)
    dates_col = poly_encoder.get_feature_names(X[['month', 'day', 'weekday', 'hour']].columns)
    X_encoded.drop(columns=['month', 'day', 'weekday', 'hour'])
    dates_poly = pd.DataFrame(dates_poly, columns=dates_col)
    dates_poly.drop(columns=['1', 'month', 'day', 'weekday', 'hour'], inplace=True)
    X.reset_index(drop=True, inplace=True)
    X = X_encoded.join(dates_poly).join(X.drop(columns=['year', 'month', 'day', 'weekday','hour']))

    # Adding cosinus and sinus features from date variables to enhance the date periodicity
    X['cos_hour'] = np.cos(X['hour']*(2.*np.pi/24))
    X['sin_hour'] = np.sin(X['hour']*(2.*np.pi/24))
    X['cos_day'] = np.cos(X['day']*(2.*np.pi/30))
    X['sin_day'] = np.sin(X['day']*(2.*np.pi/30))
    X['cos_month'] = np.cos(X['month']*(2.*np.pi/12))
    X['sin_month'] = np.sin(X['month']*(2.*np.pi/12))
    X['cos_weekday'] = np.cos(X['weekday']*(2.*np.pi/7))
    X['sin_weekday'] = np.sin(X['weekday']*(2.*np.pi/7))

    return X
```