Machine Learning II Data Challenge - Final Report

This document is our report for the ML2 final project, March 2020. It presents the steps we went through and the results we obtained, from data exploration and vizualization to feature engineering, model testing and model tuning. Our team includes:

- Tommy Tran
- Constantin Vodé
- Thomas de Mareuil

Table of Contents

Introduction

Challenge goals

The goal of this project is to train an algorithm to replace usual house energy consumption monitoring systems which are too intrusive and too expensive. This challenge is known as NILM (Nonintrusive load monitoring) or NIALM (Nonintrusive appliance load monitoring). The aim of the challenge is to predict the proportion of electric consumption in one household dedicated to 4 appliances (washing machine, fridge_freezer, TV, kettle) based on time data. The information of disaggregation consumption could help reduce electricity consumption and electricity bill with customized advice or control of client appliances.

This challenge is provided by ENS (Ecole Nationale Supérieure) and BCM Energy (start-up based in Lyon operating on the whole value chain of renewable electricity, from production to energy trading and supply to final consumers), with data available through this link (https://challengedata.ens.fr/participants/challenges/29/).

Data description

The input data includes:

- global consumption (measured every minute)
- time steps
- 7 variables related to weather (temeperature, humidity, humidity index, visibility, wind speed, windchill and pressure)

The output data includes consumption data at each time step for 4 household appliances:

- washing machine
- fridge-freezer
- TV
- kettle.

Data exploration

In [2]:
```
# import libraries and set document params
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
import seaborn as sns
%matplotlib inline
sns.set_style('darkgrid', {'xtick.bottom': True})
sns.set_palette('deep')
import warnings
warnings.filterwarnings('ignore')
```

Let's load and print the first rows of our input and output training datasets, to get a first view of our data:

In [3]:
```
input_train = pd.read_csv('X_train.csv')
input_train = input_train.drop('Unnamed: 9', axis=1)
output_train = pd.read_csv('y_train.csv')
```

In [138]:
```
input_train.head()
```

Out[138]:

	time_step	consumption	visibility	temperature	humidity	humidex	windchill	wind	pressure
0	2013-03-17T00:01:00.0	550.4000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	2013-03-17T00:02:00.0	548.6000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	2013-03-17T00:03:00.0	549.3000	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	2013-03-17T00:04:00.0	549.3667	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	2013-03-17T00:05:00.0	548.8909	NaN	NaN	NaN	NaN	NaN	NaN	NaN

In [139]:
```
output_train.head()
```

Out[139]:

	time_step	washing_machine	fridge_freezer	TV	kettle
0	2013-03-17T00:01:00.0	0.0	79.2000	7.0	0.0
1	2013-03-17T00:02:00.0	0.0	78.0000	7.0	0.0
2	2013-03-17T00:03:00.0	0.0	76.9000	7.0	0.0
3	2013-03-17T00:04:00.0	0.0	76.1111	7.0	0.0
4	2013-03-17T00:05:00.0	0.0	75.2727	7.0	0.0

We can already see that there are lots of missing data in the input dataset, we'll deal with it in the next sections. We convert the `time_steps` to datetime objects for future analysis.

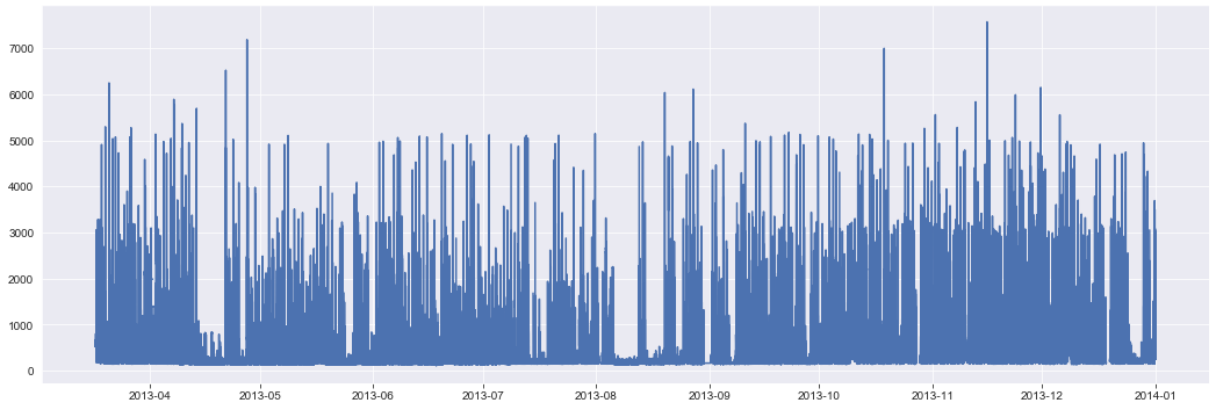
In [140]:
```
input_train['time_step'] = pd.to_datetime(input_train['time_step'])
output_train['time_step'] = pd.to_datetime(output_train['time_step'])
```

Data visualisation

Total consumption

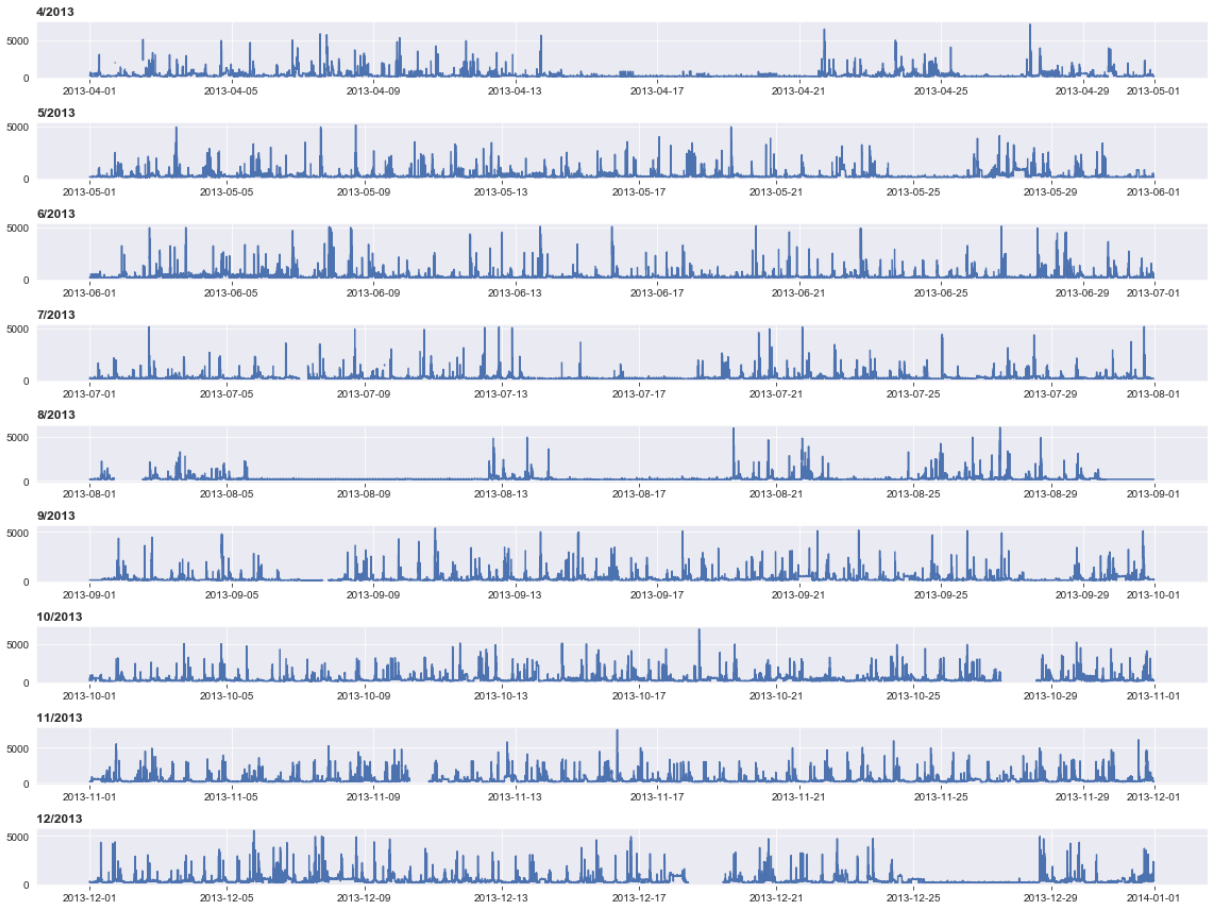
Let's first plot total consumption to see if we can identify any pattern.

In [141]:
```
total = input_train[['consumption','time_step']]
total.set_index(['time_step'], inplace=True, drop=True)

rcParams['figure.figsize'] = (18, 6)
plt.plot(total.index, total.consumption)
plt.show()
```



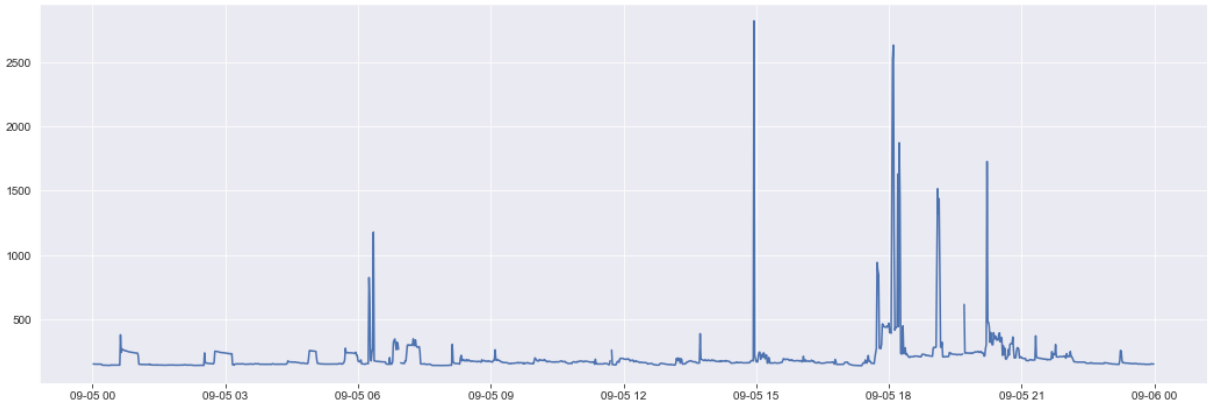
We can already see peaks in consumption, but the data is very messy. Let's zoom in and vizualize all months in rows (starting in April 2013, the first full month), to see if we can better detect any pattern.

In [142]:
```
plt.figure(figsize = (16,12))
for i in range(4,13):
    # prepare subplot
    ax = plt.subplot(9, 1, i-4+1)
    # get all observations for the month
    month = total[(total.index.month == i)]
    # plot the active power for the month
    plt.plot(month.consumption)
    # add a title to the subplot
    plt.title("{}/2013".format(i), y=1, loc='left', fontweight='semibold')
    plt.tight_layout()
plt.show()
```



This view of all months in rows allows us to see periods with consumption close to 0, which may correspond to holidays (i.e. people out of house). However it's still hard to spot any regularity. Let's zoom into juste one day.

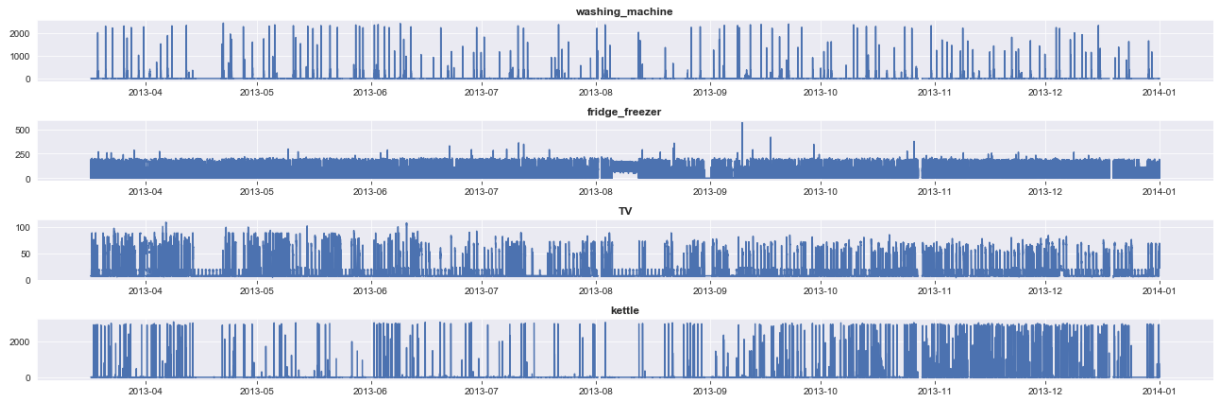
In [143]:
```
total_sept05 = total[(total.index>'2013-09-05') & (total.index<'2013-09-06')]
plt.plot(total_sept05.index, total_sept05) # I chose sept05 because it's my birthday
plt.show()
```



Based on this day (and other days that we plotted), consumption seems to be higher in the evening, with more consumption peaks from 18:00 to 21:00, but it's still hard to see any regularity / patterns... Let's take a look at consumption by appliance to try to sort this out.

Appliances consumption

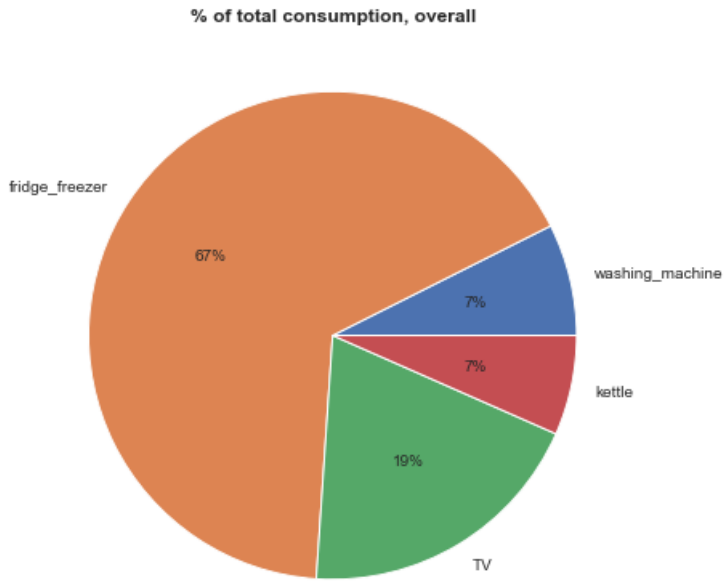
In [144]:
```
appliances = output_train.set_index(['time_step'], drop=True)

# line plot for each variable
plt.figure()
for i in range(len(appliances.columns)):
    plt.subplot(len(appliances.columns), 1, i+1)
    name = appliances.columns[i]
    plt.plot(appliances[name])
    plt.title(name, y=1, fontweight='semibold')
    plt.tight_layout()
plt.show()
```



This view of the 4 appliances is already interesting. We see that the kettle and washing machine work by peaks, with high consumption value (>2000 for each peak). We cannot see a pattern yet, but his could explain the peaks observed earlier in the total consumption graphs.

We can also see that consumption by the fridge-freezer is low (<250) but constant, which explains why in the end it constitutes the bulk of household consumption (see pie plot below for total proportions over the whole training dataset).

In [145]:
```
# plot pie plot of proportions of consumption

sums = appliances.sum(axis = 0, skipna = True)

sums.plot(kind='pie', labels=appliances.columns, figsize=[7,7],
          autopct=lambda p: '{:.0f}%'.format(p))
plt.title('% of total consumption, overall', fontsize='large', fontweight='semibold')
plt.ylabel('')
plt.show()
```

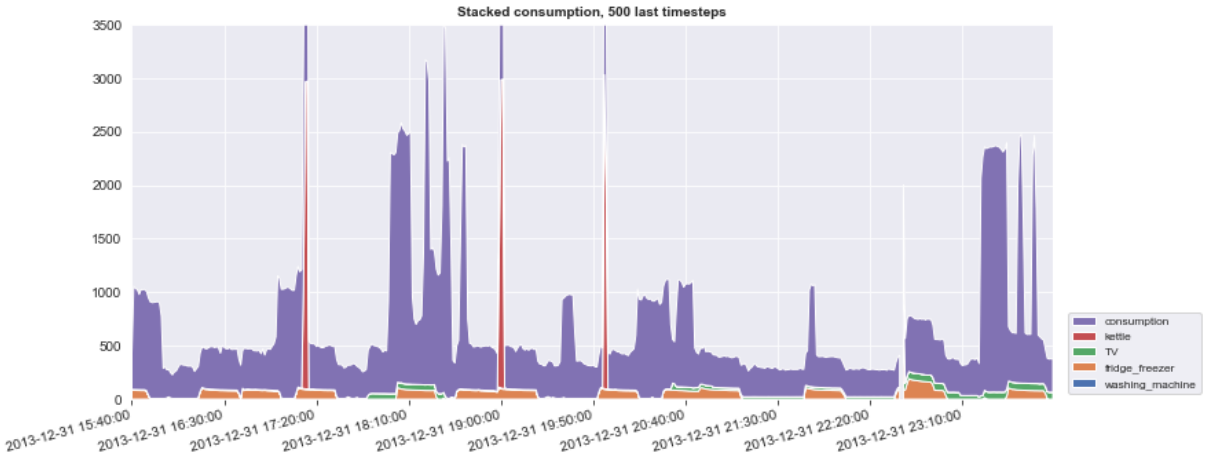


In [146]:
```
print("The maximum consumption of our 4 appliances summed together is", appliances.sum(axis
=1).max())
print("This maximum is reached on", appliances.sum(axis=1).idxmax())
```

```
The maximum consumption of our 4 appliances summed together is 3629.0
This maximum is reached on 2013-12-23 16:07:00
```

Let's plot the total consumption and the consumption of the 4 appliances in parallel, to check how total consumption reacts to variations in appliance consumption.

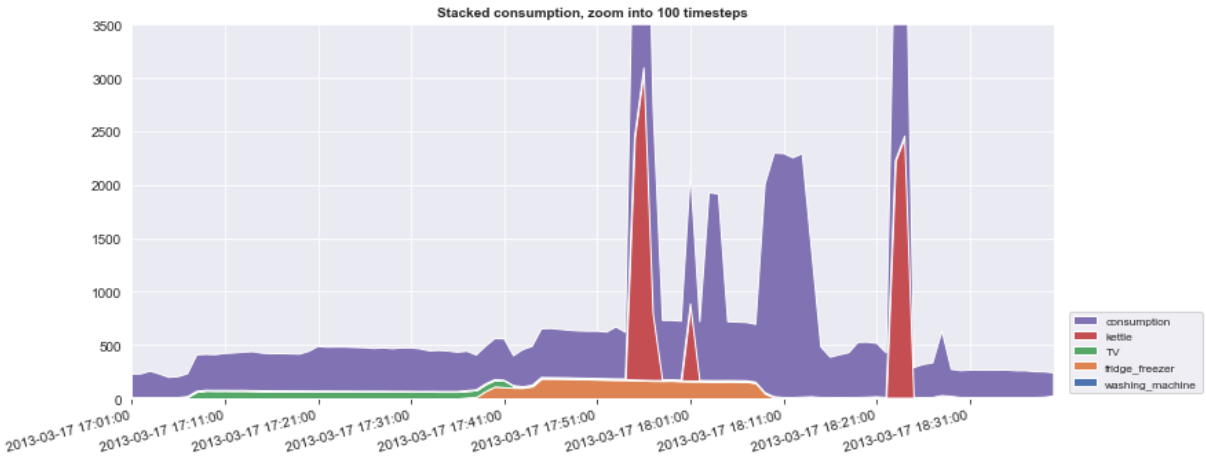
In [147]:
```
# Stackplot, large view

length = 500
appliances2 = appliances.tail(length).transpose()
total2=total.tail(length).transpose()
full = pd.concat([appliances2, total2])

labels = np.asarray(full.index)
xticks = [full.columns[i] for i in range(0,length) if i % (length/10) == 0]

fig, ax = plt.subplots(figsize=(12,5))
ax.stackplot(range(1,length+1), full, labels = labels)
ax.set(xlim=(1, length), ylim=(0, 3500), autoscale_on=False, title='Zoom window')
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles[::-1], labels[::-1], loc='lower left', bbox_to_anchor= (1.01,0.0), fontsi
ze='small')
plt.xticks(np.arange(1, length+1, step=length/10), xticks, rotation=15, ha="right")
plt.title('Stacked consumption, {} last timesteps'.format(length), fontweight="bold", fonts
ize = 'medium')
plt.show()
```



It looks like total consumption roughly follows variations in appliance consumption, but with additional variations that aren't explained by our 4 appliances - maybe it corresponds to other appliances that we don't have data for.

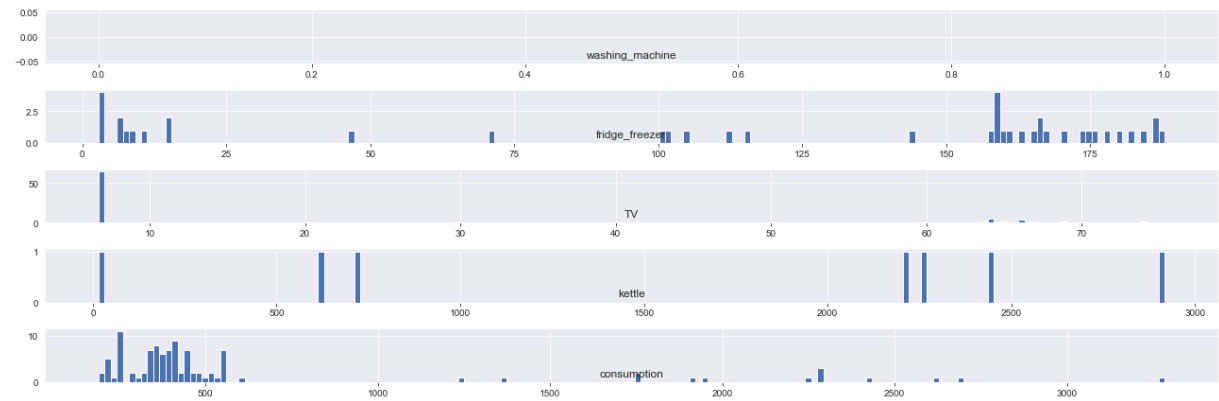
Let's zoom into another period to see closer and confirm this observation.

In [148]:
```
# Stackplot, zoomed view

appliances2 = appliances[1020:1120].transpose()
length = len(appliances2.columns)
total2=total[1020:1120].transpose()
full = pd.concat([appliances2, total2])

labels = np.asarray(full.index)
xticks = [full.columns[i] for i in range(0,length) if i % (length/10) == 0]

fig, ax = plt.subplots(figsize=(12,5))
ax.stackplot(range(1,length+1), full, labels = labels)
ax.set(xlim=(1, length), ylim=(0, 3500), autoscale_on=False,title='Zoom window')
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles[::-1], labels[::-1], loc='lower left', bbox_to_anchor= (1.01,0.0), fontsi
ze='small')
plt.xticks(np.arange(1, len(full.columns), step=len(full.columns)/10), xticks, rotation=15,
ha='right')
plt.title('Stacked consumption, zoom into {} timesteps'.format(length), fontweight="bold",
fontsize = 'medium')
plt.show()
```



We can also plot the distribution of values for each appliance and for total consumption, to see if it follows any known distribution law (based on the graphs below, it looks like it doesn't...).

In [149]:
```
full2 = full.transpose()
plt.figure()
for i in range(len(full2.columns)):
    plt.subplot(len(full2.columns), 1, i+1)
    name = full2.columns[i]
    full3 = full2[full2[name] != 0]
    full3[name].hist(bins=175)
    plt.title(name, y=0)
plt.tight_layout()
plt.show()
```

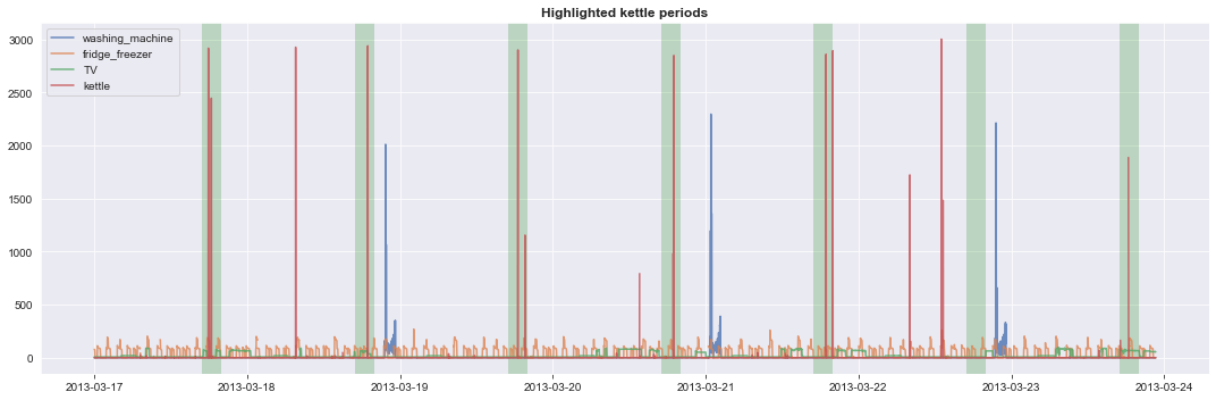


Before going into preprocessing and modeling, let's just zoom into consumption for each appliance, to get some more insights.

Kettle

For the kettle, we already saw in previous plots that consumption is very close to 0 most of the time, with bursts up to 3000 when the kettle is used. It also looks like the kettle is used mostly in the evening around 18:30. Below, we highlight the period around 18:30 every day to confirm this observation.

In [150]:
```
def plot_highlight(data, hours, minutes, title, facecolor='green', alpha_span=0.2):

    #draw line plots corresponding to columns of data
    fig, axes = plt.subplots(nrows=1, ncols=1, sharex=True, figsize=(15,5))
    for v in data.columns.tolist():
        axes.plot(data[v], label=v, alpha=.8)

    #find indexes corresponding to hours:minutes
    indices = []
    time_steps = data.index
    for i in range(len(time_steps)):
        if (pd.to_datetime(time_steps[i]).hour == hours) & (pd.to_datetime(time_steps[i]).m
inute == minutes):
            indices.append(i)

    #highlight correct periods (3-hour span, and we iterate over correct indices)
    i = 0
    while i < len(indices):
        axes.axvspan(data.index[indices[i] - 90], data.index[indices[i] + 90],
                     facecolor=facecolor, edgecolor='none', alpha=alpha_span)
        i += 1

    # add graph settings
    axes.set_title(title, fontsize = 12, fontweight = 'semibold')
    axes.legend()
    plt.tight_layout()
    plt.show()

plot_highlight(appliances.head(10000), hours = 18, minutes = 30, title = "Highlighted kettl
e periods")
```



However, we observe some exceptions, e.g. additional use of kettle at other times than 18:30 for some days.

Washing-machine

As we can see in the plot above, the washing machine also works by peaks, and seems to be set mostly at night (perhaps when the person goes to bed and lets it wash overnight). However there is no repetitive pattern (it can be every day for a few days, then 3 days without washing, it varies a lot in the dataset).

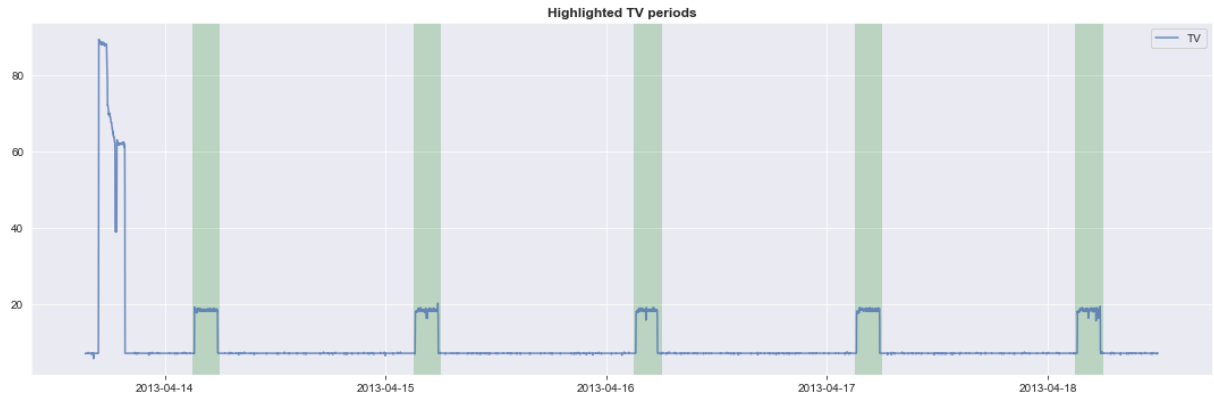
TV

The TV has the most repetitive consumption pattern. This explains why it will be the appliance we best predict in the next sections.

While TV consumption remains very low (it never goes over 110, vs. kettle over 3000), there are some peaks in the evening, probably when the person watches TV, but not only. Those big peaks don't seem to be regular, but we can observe smaller bursts in consumption (at ≈20), everyday, from 3 to 6 am. Maybe it corresponds to some regular automatic TV connection at night for updates / maintenance.

See below one peak (TV on) on April 13th, and then a series of very regular "update bursts", highlighted in green.

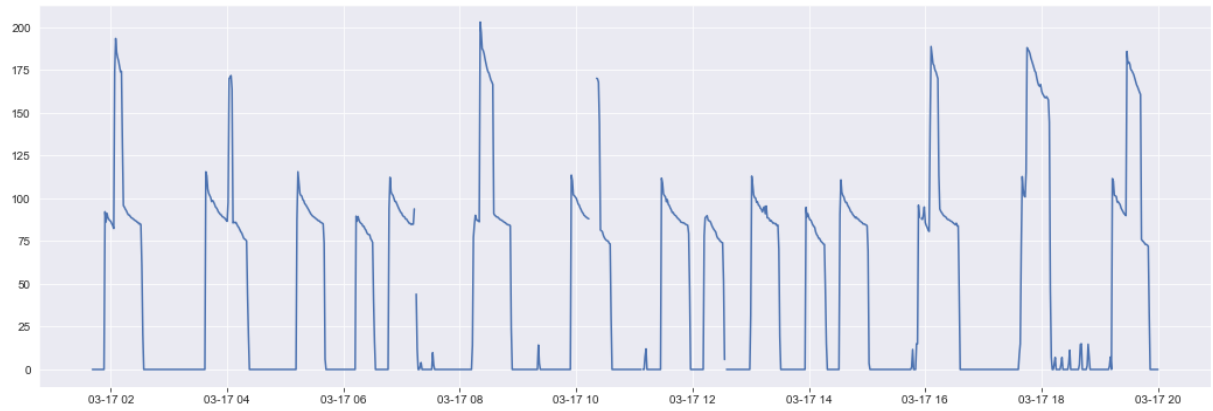
In [151]:
```
plot_highlight(pd.DataFrame(appliances.TV[39800:46800]), hours = 4, minutes = 30, title =
"Highlighted TV periods")
```



Fridge-freezer

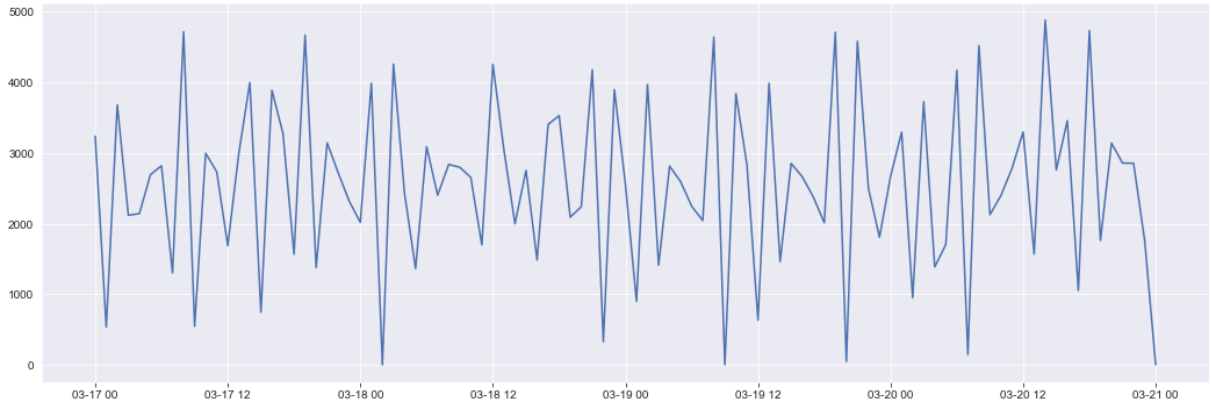
Last, for the `fridge_freezer`, we observe peaks of variable height, which seem often separated by ≈1 hour, but with no real regularity.

In [152]:
```
plt.plot(appliances.fridge_freezer[100:1200])
plt.show()
```

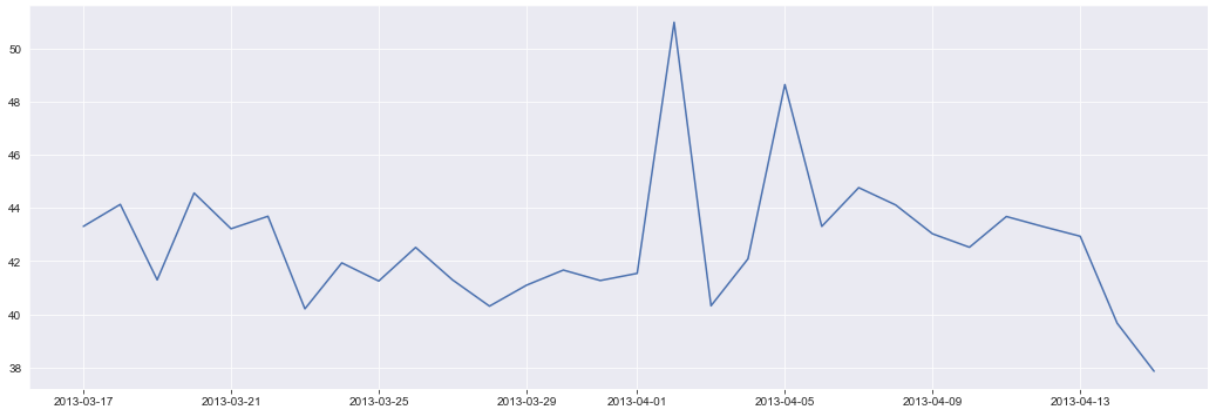


We finally tried to group values by hour and by day using pandas' `Grouper`. The plots below show respecively 4 days of fridge consumption, grouped by hour, and 30 days (1 month), grouped by day. But this shows there doesn't seem to be any regularity: all hours have different consumptions, and days are very different from one another (be it in terms of mean consumption or total consumption of the fridge)... Let's hope our models will be able to predict such unregular data!

In [153]:
```
fridge_hours = appliances.fridge_freezer[:5760].groupby(pd.Grouper(freq='H')).sum()
plt.plot(fridge_hours)
plt.show()
```



In [154]:
```
fridge_days = appliances.fridge_freezer[:43000].groupby(pd.Grouper(freq='D')).mean()
plt.plot(fridge_days)
plt.show()
```

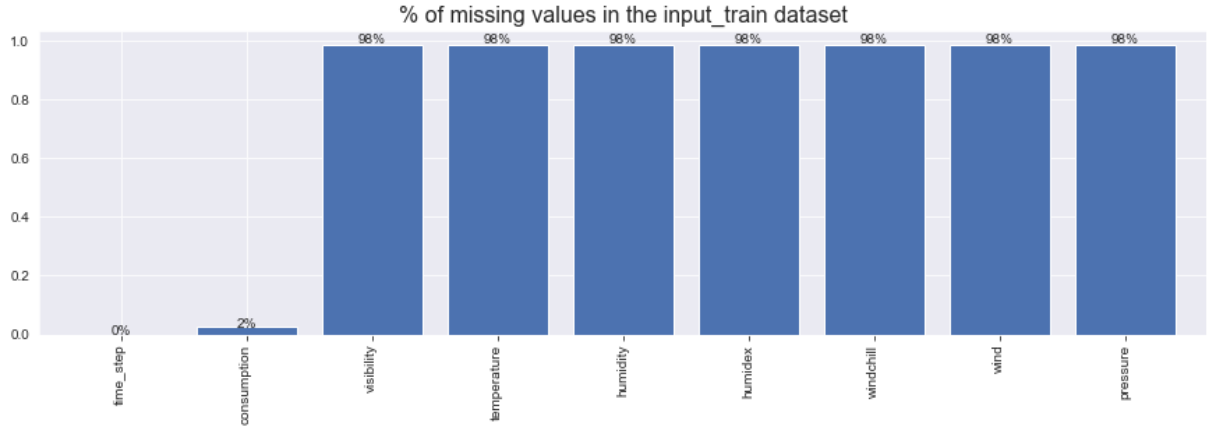


Now we vizualized our data, we will preprocess it so it can be fed to our models. The plots sometimes often had discontinued lines, which correspond to missing values: in the next section, this will be the first issue we address.

Missing values

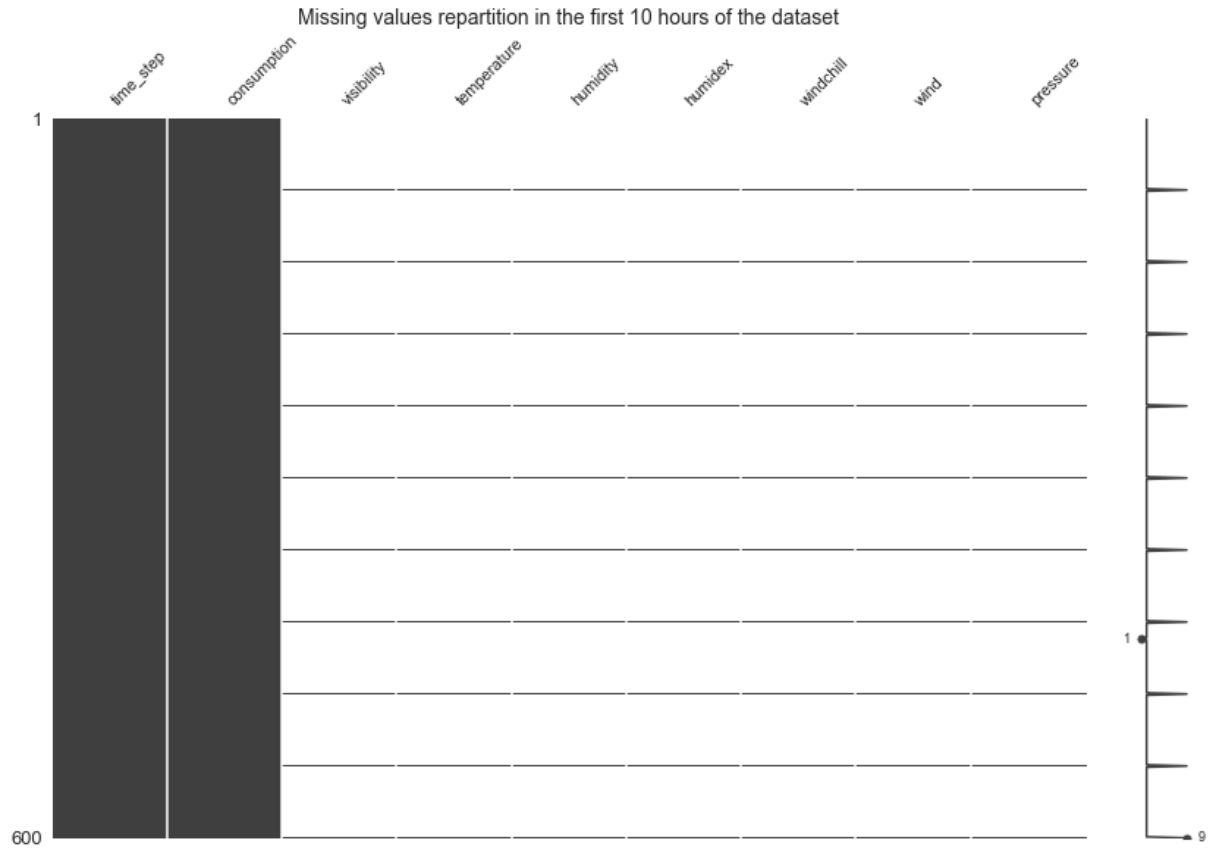
The train set contains 417 599 values with 10 231 missing values (2.44%) for `consumption`, `washing_machine`, `fridge_freezer`, `TV` and `kettle`, and the test set contains 226 081 values with 24 719 missing values (10.93%). Let's visualise them:

In [155]:
```
missing = input_train.isna().sum()
missing = missing / len(input_train)

ax = missing.plot(kind='bar',figsize=(15,4),width = 0.8,edgecolor=None)
plt.title("% of missing values in the input_train dataset",fontsize= 16)

for p in ax.patches:
    width, height = p.get_width(), p.get_height()
    x, y = p.get_xy()
    ax.annotate('{:.0%}'.format(height), (p.get_x()+.5*width, p.get_y()+1.01*height), ha =
'center')
```



In the graph below, missing values are in white. We can see that all meteorological features are measured only once per hour (they correspond to the 10 horizontal lines, 1 for each of the 10 first hours plotted in this graph), which explains the high proportion of missing values. Missing values in consumption are too scarce to be visible with this graph - we will deal with them later.

In [156]:
```
import missingno as msno
msno.matrix(input_train[:600], figsize=(14,9), fontsize=11)
plt.title("Missing values repartition in the first 10 hours of the dataset", fontsize=14)
plt.show()
```



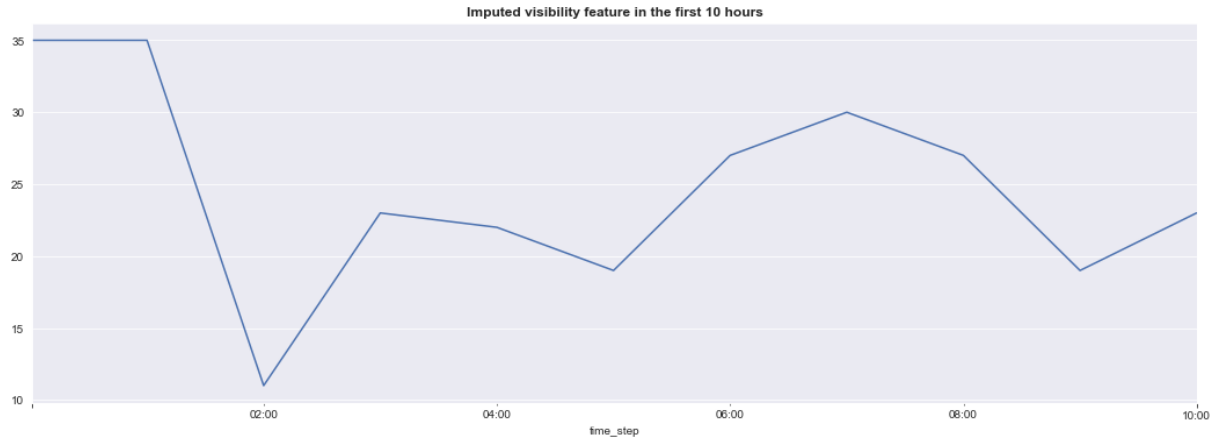
To impute the missing meteorological values, we will consider that variations between hourly measurements are **linear**. We will therefore use a linear interpolation method.

In [157]:
```
to_interpolate = input_train.columns[-7:] # selects weather features
input_train[to_interpolate] = pd.Series.interpolate(input_train[to_interpolate],
                                                    method='linear', # linear interpolation
                                                    limit=None, limit_direction='both')
input_train[55:65] # display just a few of them
```

Out[157]:

	time_step	consumption	visibility	temperature	humidity	humidex	windchill	wind	pressure
55	2013-03-17 00:56:00	642.6060	35.0	8.9	86.000000	8.9	6.000000	19.000000	1017.300000
56	2013-03-17 00:57:00	601.3000	35.0	8.9	86.000000	8.9	6.000000	19.000000	1017.300000
57	2013-03-17 00:58:00	594.9556	35.0	8.9	86.000000	8.9	6.000000	19.000000	1017.300000
58	2013-03-17 00:59:00	586.2000	35.0	8.9	86.000000	8.9	6.000000	19.000000	1017.300000
59	2013-03-17 01:00:00	586.9000	35.0	8.9	86.000000	8.9	6.000000	19.000000	1017.300000
60	2013-03-17 01:01:00	616.5000	34.6	8.9	86.033333	8.9	6.003333	18.966667	1017.293333
61	2013-03-17 01:02:00	541.0667	34.2	8.9	86.066667	8.9	6.006667	18.933333	1017.286667
62	2013-03-17 01:03:00	541.2000	33.8	8.9	86.100000	8.9	6.010000	18.900000	1017.280000
63	2013-03-17 01:04:00	540.5000	33.4	8.9	86.133333	8.9	6.013333	18.866667	1017.273333
64	2013-03-17 01:05:00	540.8000	33.0	8.9	86.166667	8.9	6.016667	18.833333	1017.266667

In [158]:
```
input_train2 = input_train.set_index(['time_step'], drop=True)
input_train2['visibility'][:600].plot()
plt.title('Imputed visibility feature in the first 10 hours', fontweight='semibold')
plt.show()
```



In [159]:
```
# check nb of missing values after imputation
input_train.isna().sum()
```

Out[159]:
```
time_step          0
consumption    10231
visibility         0
temperature        0
humidity           0
humidex            0
windchill          0
wind               0
pressure           0
dtype: int64
```

Now that we don't have any more missing weather values, let's focus on the missing `consumption` values.

We tried to print these values (with code line: `print(input_train[input_train.isna().any(axis=1)])`) to see if we could find any pattern. Missing values happen mostly at the end of the dataset, but they seem to be random. Maybe it corresponds to defaults in measurement systems.

We also observed that the missing `consumption` values we also missing in `output_train` for the 4 appliances. Therefore we had a choice: either drop all of these missing time steps and train without them (as it only represents less than 2% of our training data), or impute them and train our models using a full dataset.

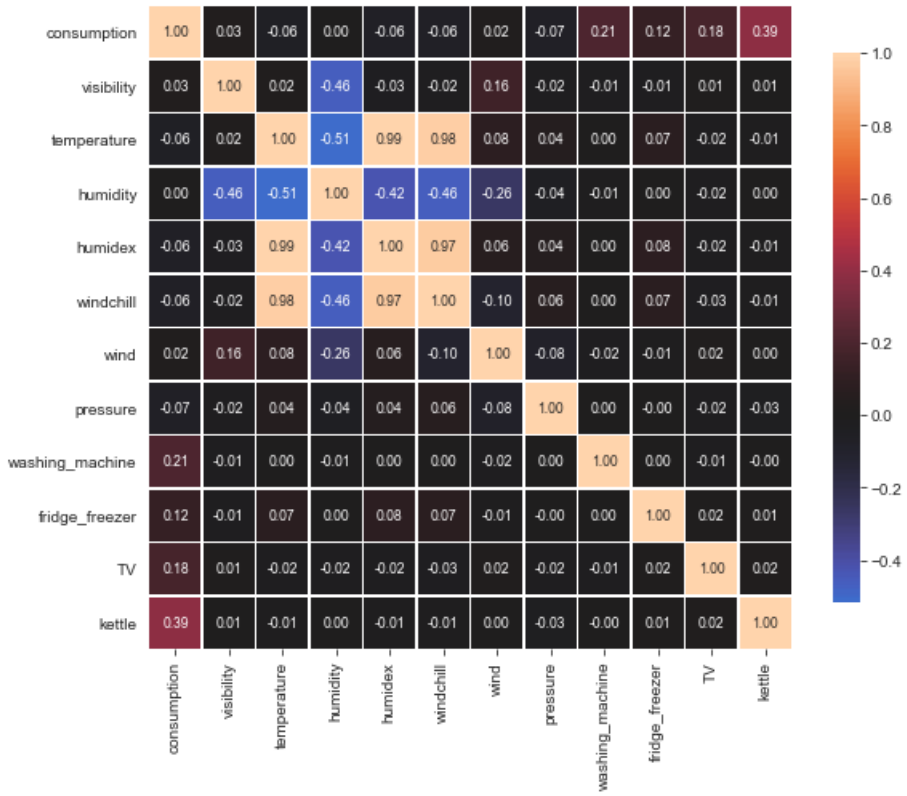
We tried several imputation methods (linear interpolation, replacement with same measurement the day before) but it did not improve our predictions. As a consequence we finally chose to just drop the missing values.

Correlation

We see in the correlation heatmap that some weather variables are a little correlated together: we will drop the repetitive ones in preprocessing (next section).

But what's most important is that there is **no relationship between the target variables** (see the 4 last lines in the heatmap, corresponding to `washing_machine`, `fridge_freezer`, `TV`and `kettle`). This is why in the Modeling section we will predict them independantly, with 4 separate models.

In [231]:
```
total_data = pd.DataFrame()
total_data[input_train.columns] = input_train
total_data[output_train.columns] = output_train

def correlation_heatmap(train):
    correlations = train.corr()
    fig, ax = plt.subplots(figsize=(9,9))
    sns.heatmap(correlations, vmax=1.0, center=0, fmt='.2f',
                square=True, linewidths=.5, annot=True, cbar_kws={"shrink": .70}, annot_kws
={"fontsize":8.5})
    plt.show();

correlation_heatmap(total_data)
```



Feature engineering

Basic feature encoding

For pre-processing / feature encoding, our first step was:

- add **dummy variables** (one-hot encoded) corresponding to months, weeks, weekends and days of the week
- add sinus and cosinus **circular time encoding**, in order to take into account time circularity (i.e. 23pm is close to 1am)
- **drop the repetitive** weather-related variables (cf. correlation) and keep the other ones, even though modeling showed us that their importance was low for regression (still better than nothing!)

In [4]:
```
def preprocess(X):
    X.drop(["Unnamed: 9"], axis=1, inplace=True)

    # fill-in missing weather values
    to_interpolate = X.columns[-7:]
    X[to_interpolate] = pd.Series.interpolate(X[to_interpolate], method='linear',
                                              limit=None, limit_direction='both')
    # add dummy time variables
    X.rename(columns = {X.columns[0]:'time_step'}, inplace=True)
    X['time_step'] = pd.to_datetime(X['time_step'])
    X['weekend'] = ((pd.DatetimeIndex(X['time_step']).dayofweek) // 5 == 1).astype(float)
    X['weekday'] = X['time_step'].dt.weekday
    X = X.join(pd.get_dummies(X['weekday'], prefix='wd'))
    X['month'] = X['time_step'].dt.month
    X['week'] = X['time_step'].dt.week
    X['hour'] = X['time_step'].dt.hour
    X['day_of_year'] = X['time_step'].dt.dayofyear

    # add circular time variables
    X['sin_time'] = np.sin(2*np.pi*X['time_step'].dt.hour/24)
    X['cos_time'] = np.cos(2*np.pi*X['time_step'].dt.hour/24)

    # drop repetitive features and NaNs
    X.drop(['time_step','weekday',
            'windchill', 'visibility', 'pressure', 'humidity', 'humidex'],
            axis=1, inplace=True)
    X = X.dropna()
    return X
```

In [5]:
```
# preprocess input_train data
input_train = pd.read_csv('X_train.csv')
input_train = preprocess(input_train)

input_test = pd.read_csv('X_test.csv')
input_test = preprocess(input_test)

# preprocess output_train data
output_train = pd.read_csv('y_train.csv')
output_train['time_step'] = pd.to_datetime(output_train['time_step'])
output_train.drop(columns=['time_step'], inplace=True)
output_train = output_train.dropna()
```

After several back-and-forths with modeling, we tested more complex feature encodings and added more efficient features. We finally came up with the pre-processing function presented below.

More complex feature engineering for our final model

In our final pre-processing function, we notably added:

- **average values** for the hour, month, week and day of the year
- **delta in consumption** compared to the previous time step
- **lag features** (consumption 1 hour before and 1 day before each time step)
- **rolling statistics** (hourly mean, daily mean)
- **polynomial features** (e.g. consumption *x* hour of the day, see code below)
- dummy variables for **consumption magnitude**

This last new feature is of particular importance: it corresponds to one-hot encoded indicators for each bucket of 250 in consumption, i.e. a column with ones if consumption is between 2500 and 2750, another for consumption between 2750 and 3000, and so on. This gives more flexibility to the model: instead of having just 1 coefficient (total consumption) to predict 4 target variables, it now has other coefficients (the magnitude indicators) to adjust predictions depending on consumption magnitude. This helps notably the model to better translate consumption peaks into high kettle values.

In [6]:
```
def advanced_preprocess(X):

    #Average values
    groupcollist = ['hour', 'month', 'week', 'day_of_year', 'weekend']
    aggregationlist = [('consumption',np.mean,'avg')]
    for type_id in groupcollist:
        for column_id,aggregator,aggtype in aggregationlist:
            mean_df = X.groupby([type_id]).aggregate(aggregator).reset_index()[[column_id,t
ype_id,]]
            mean_df.columns = [type_id+'_'+aggtype+'_'+column_id,type_id]
            X = pd.merge(X,mean_df,on=type_id,how='left')

    #Delta compared to previous time step
    X['delta_cons'] = X.consumption.diff()

    ##Polynomial Features
    X['cons^2'] = X.consumption.apply(lambda x: x*x)
    X['cons_X_hour'] = X.consumption*X.hour
    X['delta_cons^2'] = X.delta_cons.apply(lambda x: x*x)
    X['cons_X_sin'] = X['consumption']*X['sin_time']
    X['cons_X_temp'] = X['consumption']*X.temperature
    X['delta_cons^2'] = X.delta_cons*X.delta_cons

    #Lag features
    X['day-1'] = X.consumption.shift(1440)
    X['hour-1'] = X.consumption.shift(60)

    #Rolling statistics
    X['hourly_mean'] = X.consumption.rolling(60).mean()
    X['daily_mean'] = X.consumption.rolling(1440).mean()

    #Dummy variables for consumption magnitude
    X['magnitude'] = (X.consumption//250)+1
    X = X.join(pd.get_dummies(X['magnitude'], prefix='magn'))

    X.fillna(0, inplace=True)
    return X

input_train = advanced_preprocess(input_train)
input_test = advanced_preprocess(input_test)
```

In [7]:
```
input_test.drop(columns = ['magn_26.0','magn_27.0','magn_29.0','magn_30.0'], inplace = True
)
input_train.drop(columns = ['magn_27.0','magn_28.0','magn_29.0','magn_31.0'], inplace = Tru
e)
```

Below are the preprocessed datasets ready for training:

In [325]:
```
print(input_train.shape)
input_train.head()
```

```
(407368, 58)
```

Out[325]:

	consumption	temperature	wind	weekend	wd_0	wd_1	wd_2	wd_3	wd_4	wd_5	wd_6	month	week	hour
0	550.4000	8.9	19.0	1.0	0	0	0	0	0	0	1	3	11	0
1	548.6000	8.9	19.0	1.0	0	0	0	0	0	0	1	3	11	0
2	549.3000	8.9	19.0	1.0	0	0	0	0	0	0	1	3	11	0
3	549.3667	8.9	19.0	1.0	0	0	0	0	0	0	1	3	11	0
4	548.8909	8.9	19.0	1.0	0	0	0	0	0	0	1	3	11	0

In [383]:
```
print(output_train.shape)
output_train.head()
```

```
(407368, 4)
```

Out[383]:

	washing_machine	fridge_freezer	TV	kettle
0	0.0	79.2000	7.0	0.0
1	0.0	78.0000	7.0	0.0
2	0.0	76.9000	7.0	0.0
3	0.0	76.1111	7.0	0.0
4	0.0	75.2727	7.0	0.0

Modeling

We want to predict, for each minute, the power used by the washing machine, the fridge, the TV and the kettle.

In a regular time series problem, you usually have past data (e.g. stock prices over the past year) and you want to predict the next values (e.g. stock price for tomorrow, or for the next week). For prediction, you use methods such as Autoregression (VAR), ARIMA, etc. You can also create lag features (i.e. use the data from previous days as a feature) to transform the problem into a supervised learning problem.

We tried these regular time series methods (Autoregression/VAR, ARIMA) and we tried to use past time steps as features (lag features), but this did not yield good results (see the corresponding sub-section below). In fact, the difference here is that we don't predict the 4 appliances based on their past data. We have to predict 4 full time series (the 4 appliances), based on a different time series (total consumption), and we need to train a model to link inputs to outputs. It's in fact **a regression problem, with 1 input (total consumption, + a few weather-related data) and 4 outputs (the 4 appliances**. We train a regressor on a training dataset, and we use another dataset for predictions.

Therefore, as a baseline we used 4 separate linear regressions. We then tried more elaborate regression methods: random forests, boosted trees (gradient boosting) and XGBoost. We also tried deep learning methods: fully-connected neural networks, convolutional networks (CNN) and recurrent networks (RNN using LSTM cells). Our best model finally uses XGBoost. Below, you will find a summary of our explorations and results.

Train-test split

First, for local model testing, we split our training data using `train_test_split` and we define the RMSE metric. The built-in `train_test_plit` function shuffles data before splitting, which we cannot afford with ordered time data. Therefore we have to do the split by hand, with the first 80% of data kept for training and the last 20% for testing (on both input and output data, X and y).

In [258]:
```
def rmse(y_true, y_pred):
    return np.sqrt(np.mean(np.square(y_true - y_pred)))
```

In [301]:
```
train_size = int(len(input_train)*0.8)
test_size = int(len(input_train)-train_size)

X_train = input_train[:train_size]
X_test = input_train[train_size:]

y_train = output_train[:train_size]
y_test = output_train[train_size:]
```

Baseline models: 4 linear regressions

The benchmark we used is 4 univariate linear regressions (one per appliance).

In [189]:
```
from sklearn.linear_model import LinearRegression
```

In [194]:
```
def regressor(model, X_train, y_train, X_test, y_test):
    preds = []
    scores = []
    for i in range(len(y_train.columns)):
        appliance = y_train.columns[i]               # select appliance to train on
        reg = model.fit(X_train, y_train[appliance]) # train model
        pred_i = model.predict(X_test)               # predict using X_test
        preds.append(pred_i)                         # store predictions
        score_i = rmse(y_test[appliance], pred_i)    # compute model score
        scores.append(score_i)                       # save scores
    return preds, scores
```

In [199]:
```
baseline_model = LinearRegression()
base_preds, base_scores = regressor(baseline_model, X_train, y_train, X_test, y_test)
print(base_preds)
print(base_scores)
```

```
[array([9.19721154, 8.59050119, 8.78618371, ..., 8.34698019, 8.40484506,
       8.34483533]), array([47.01168678, 46.7130305 , 46.81092957, ..., 49.04371246,
       49.0723102 , 49.04265244]), array([23.45157971, 23.33676246, 23.38908689, ..., 18.9
3711574,
       18.94916923, 18.93666896]), array([8.06708192, 6.47850118, 7.01455432, ..., 2.66354
416, 2.81676295,
       2.65786485])]
[54.563536873321446, 51.23345954232711, 16.545863495169044, 138.7133055076153]
```

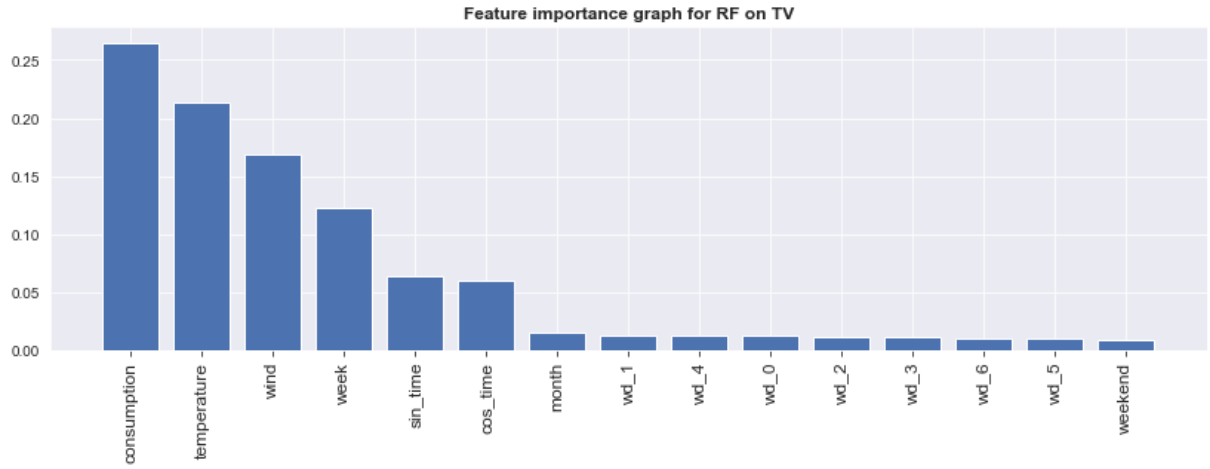
To improve this baseline model, we could cross-validate with different splits (e.g. train on the last 80% of data and test on the first 20%, etc.), but let's rather move on to more complex models to improve predictions.

More elaborate regressions: Random forests and Gradient boosting

Random Forests

In [260]:
```
from sklearn.ensemble import RandomForestRegressor

RF_model = RandomForestRegressor()
RF_preds, RF_scores = regressor(RF_model, X_train, y_train, X_test, y_test)
```

In [268]:
```
RF = RF_model.fit(input_train, output_train['TV'])

X_columns = input_train.columns
ordering = np.argsort(RF.feature_importances_)[::-1]
importances = RF.feature_importances_[ordering]
feature_names = X_columns[ordering]
x = np.arange(len(feature_names))

plt.figure(figsize = (14, 4))
plt.bar(x, importances)
plt.xticks(x, feature_names, rotation=90, fontsize=12)
plt.title('Feature importance graph for RF on TV', fontsize=12, fontweight='semibold')
plt.show()
```



In this feature importance graph we can see that for the Random Forest to predict TV, `consumption` is the most important variables, as well as `week`, `sin/cos-time` and `month`. Weather variables seem also to be somehow important, but their importance is lower with other model types.

Gradient Boosting (within MultiOutputRegressor)

For Gradient Boosting implementation, we found a built-in sklearn class called `MultiOutputRegressor`, which can train 4 models in 1 command. The only problem with this method is that it's harder to tune (grid search becomes actually much more complex to implement than with 4 separate models). We used it with default Gradient Boosting parameters, as shown below.

In [ ]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.multioutput import MultiOutputRegressor

def train(X, y):
    model = MultiOutputRegressor(GradientBoostingRegressor(), n_jobs=-1).fit(X, y)
    return model

model_MTR = train(input_train, output_train)
```

These models performed a little better than baseline.

For submission, we trained these models on the full dataset and made predictions for the 4 appliances based on the `input_test` data. We then wrote a custom function (not included in this report, for simplicity) that concatenates 4 predictions and writes a csv document for online submission. Online score was around 45, vs. baseline at ≈48.

Visual comparison of machine learning models

In the cell below, we test several models and plot a comparison graph.

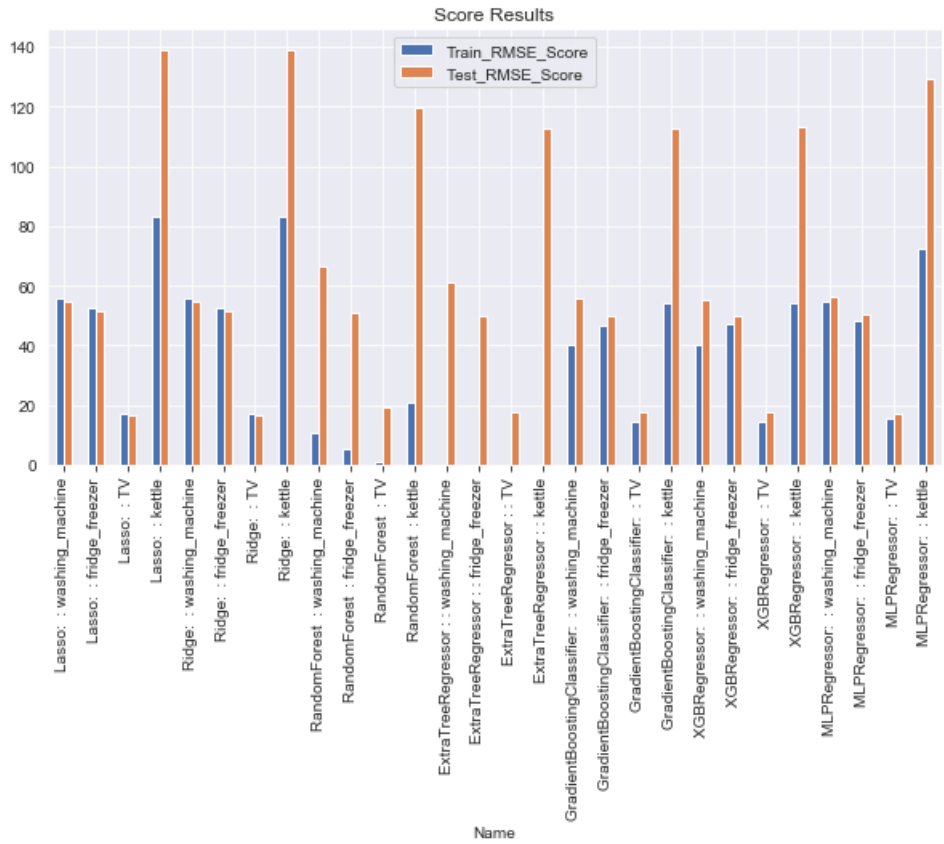
In [ ]:
```
from sklearn.linear_model import Ridge, Lasso
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.neural_network import MLPRegressor
import xgboost as xgb
from xgboost import XGBRegressor
import time
from math import sqrt
from sklearn.metrics import mean_squared_error

models = [ ['Lasso: ', Lasso()],
           ['Ridge: ', Ridge()],
           ['RandomForest ',RandomForestRegressor()],
           ['ExtraTreeRegressor :',ExtraTreesRegressor()],
           ['GradientBoostingClassifier: ', GradientBoostingRegressor()] ,
           ['XGBRegressor: ', xgb.XGBRegressor()] ,
           ['MLPRegressor: ', MLPRegressor(activation='relu', solver='adam',learning_rate=
'adaptive',max_iter=1000,learning_rate_init=0.01,alpha=0.01)]]

# Run all the proposed models and store train and test scores in a list
model_data = []
for name,curr_model in models :
    for appliance in y_train.columns:
        curr_model_data = {}
        curr_model.random_state = 78
        curr_model_data["Name"] = name+' : '+appliance
        start = time.time()
        curr_model.fit(X_train,y_train[appliance])
        end = time.time()
        curr_model_data["Train_Time"] = end - start
        curr_model_data["Train_RMSE_Score"] = rmse(y_train[appliance],curr_model.predict(X_
train))
        curr_model_data["Test_RMSE_Score"] = rmse(y_test[appliance],curr_model.predict(X_te
st))
        print(curr_model_data)
        model_data.append(curr_model_data)
```

In [251]:
```
model_df = pd.DataFrame(model_data)
model_df[4:12]
```

Out[251]:

	Name	Train_Time	Train_RMSE_Score	Test_RMSE_Score
4	Ridge: : washing_machine	0.314998	55.824032	54.563537
5	Ridge: : fridge_freezer	0.075344	52.498158	51.233461
6	Ridge: : TV	0.075227	16.967621	16.545856
7	Ridge: : kettle	0.076840	82.891175	138.713306
8	RandomForest : washing_machine	446.700031	10.579237	66.549605
9	RandomForest : fridge_freezer	395.131883	5.250862	50.879147
10	RandomForest : TV	437.495346	0.994132	19.452610
11	RandomForest : kettle	368.117551	20.848427	119.639223

In [255]:
```
model_df.plot(x="Name", y=[ 'Train_RMSE_Score' , 'Test_RMSE_Score'],
              kind="bar", title = 'Score Results', figsize= (10,5))
plt.show()
```



In the graph above, we can visually compare the performance of each model on the 4 appliances. What we look for is the lowest test error possible (in orange). We can see that the best scores are obtained with Gradient Boosting and its variant XGBoost (which we will later select as our final model - see last section of this report).

The problem with the linear models and the random forests/gradient boosting that we used until now is that **they do not efficiently take into account the time nature of data**. This is why we then tried deep learning methods.

Deep learning models: ANN, CNN, RNN

Vanilla NN

We first tried a standard fully-connected neural network (using Keras) with 2 dense layers (ReLu activation), and a final output layer comprising 4 cells with linear activation, in order to output 4 values (the 4 appliances). For training we chose the Adam gradient optimizer (state-of-the-art) and 6 epochs (in order to stop training before it overfits).

In [35]:
```
from keras.models import Sequential
from keras.layers import Dense, Activation
from sklearn.metrics import mean_squared_error

ann = Sequential()
ann.add(Dense(32, input_dim=input_train.shape[1], activation='relu'))
ann.add(Dense(16, activation='relu'))
ann.add(Dense(4, activation='linear'))
ann.compile(loss='mean_squared_error', optimizer='adam')

ann.summary()
```

```
Using TensorFlow backend.

WARNING:tensorflow:From /Users/netflix/opt/anaconda3/lib/python3.7/site-packages/tensorflo
w_core/python/ops/resource_variable_ops.py:1630: calling BaseResourceVariable.__init__ (fr
om tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be
removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                608
_________________________________________________________________
dense_2 (Dense)              (None, 16)                528
_________________________________________________________________
dense_3 (Dense)              (None, 4)                 68
=================================================================
Total params: 1,204
Trainable params: 1,204
Non-trainable params: 0
_________________________________________________________________
```

In [36]:
```
ann.fit(input_train, output_train, batch_size=32, epochs=6, verbose=1)
```

```
WARNING:tensorflow:From /Users/netflix/opt/anaconda3/lib/python3.7/site-packages/keras/bac
kend/tensorflow_backend.py:422: The name tf.global_variables is deprecated. Please use tf.
compat.v1.global_variables instead.

Epoch 1/6
417599/417599 [==============================] - 23s 54us/step - loss: 3453.0641
Epoch 2/6
417599/417599 [==============================] - 21s 51us/step - loss: 3318.6284
Epoch 3/6
417599/417599 [==============================] - 24s 57us/step - loss: 3263.6336
Epoch 4/6
417599/417599 [==============================] - 23s 56us/step - loss: 3249.1618
Epoch 5/6
417599/417599 [==============================] - 23s 54us/step - loss: 3240.0047
Epoch 6/6
417599/417599 [==============================] - 24s 56us/step - loss: 3224.4089
```

Out[36]: <keras.callbacks.callbacks.History at 0x13e123950>

For readability of this report, we did not include the function to compute predictions and write a csv for submission. After submission, this model had a performance around 45, i.e. a little above baseline (47.6) and equivalent to the random forest and gradient boosting regressions. The loss after 6 epochs of training (≈3224, corresponding to the total mean-squared error) is a good benchmark for the more complex networks in the next sections (which will be a little more performant - see below).

CNN

To better take into account the structure of data, we then tried CNNs.

CNNs have to receive as input a 3-dimensional object, so we started by reshaping the input values to include a 3rd dimension of size 1, as evidenced below.

In [203]:
```
print(input_train.values.shape)
input_train_cnn = input_train.values.reshape((input_train.shape[0], 1, input_train.shape[1
]))
print(input_train_cnn.shape)
```

```
(407368, 18)
(407368, 1, 18)
```

The first CNN below has 1 convolutional layer and 1 dense layer, followed by the output layer with linear activation. The convolutional layer takes as input each row of the `input_train` data, and "reads" it with a kernel of size 2 (i.e. length of the 1D convolutional window is 2). It has "same" padding, which means that it adapts padding so that output has the same length as the original input. Batch normalization and Dropout layers (to avoid overfitting) did not significantly improve performance, so we did not add them.

In [40]:
```
from keras.layers import MaxPooling1D, GlobalMaxPooling1D, Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.normalization import BatchNormalization

cnn = Sequential()

cnn.add(Conv1D(64,input_shape=(1, 18),kernel_size=2,padding='same',activation='relu',stride
s=1))
cnn.add(GlobalMaxPooling1D())
#cnn.add(BatchNormalization())
cnn.add(Dense(64,activation='relu'))
#cnn.add(Dropout(0.2))
cnn.add(Dense(4, activation='linear'))
cnn.compile(loss='mean_squared_error',optimizer='adam')

cnn.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 1, 64)             2368
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_5 (Dense)              (None, 4)                 260
=================================================================
Total params: 6,788
Trainable params: 6,788
Non-trainable params: 0
_________________________________________________________________
```

In [41]:
```
cnn.fit(input_train_cnn, output_train, batch_size=32, epochs=6, verbose=1)
```

```
Epoch 1/6
417599/417599 [==============================] - 30s 72us/step - loss: 3448.4684
Epoch 2/6
417599/417599 [==============================] - 26s 63us/step - loss: 3315.1157
Epoch 3/6
417599/417599 [==============================] - 25s 61us/step - loss: 3264.1828
Epoch 4/6
417599/417599 [==============================] - 26s 61us/step - loss: 3221.9133
Epoch 5/6
417599/417599 [==============================] - 25s 60us/step - loss: 3208.3612
Epoch 6/6
417599/417599 [==============================] - 25s 61us/step - loss: 3210.2061
```

Out[41]:
```
<keras.callbacks.callbacks.History at 0x147bd39d0>
```

As you can see with the training above, performance is a little better than the dense network from the previous section (3210 vs. 3224 after 6 epochs of training), but remains very close. Score for this model after submission was almost the same, around 45.

To help the model learn from past time steps, and not just from separate rows of data (separate time steps, with no link between each other), we tried another CNN that takes as input 3-dimensional objects than over several time steps. That is to say we reshaped the input data so that it is now constituted of blocks of 4 time steps (using the function `split_sequences` below).

In [211]:
```
def split_sequences(input_train, output_train, n_steps):
    sequences = np.hstack((input_train, output_train))
    X = list()
    y = list()
    for i in range(len(sequences)):
        # define breaking points every n_steps
        end_ix = i + n_steps
        if end_ix > len(sequences):
            break
        # split sequences, both for input and corresponding output data
        seq_x, seq_y = sequences[i:end_ix, :-4], sequences[end_ix-1, -4:]
        X.append(seq_x)
        y.append(seq_y)
    return np.asarray(X), np.asarray(y)

input_train_cnn2, output_train_cnn2 = split_sequences(input_train.values, output_train.valu
es, 4)

print(input_train_cnn2.shape)
print(input_train_cnn2)
print("\n",output_train_cnn2.shape)
print(output_train_cnn2)
```

```
(407365, 4, 18)
[[[ 5.50400000e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.48600000e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.49300000e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.49366700e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]]

 [[ 5.48600000e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.49300000e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.49366700e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.48890900e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]]

 [[ 5.49300000e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.49366700e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.48890900e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]
  [ 5.49633300e+02  3.50000000e+01  8.90000000e+00 ...  1.00000000e+00
    0.00000000e+00  1.00000000e+00]]

 ...

 [[ 4.07875000e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.42666700e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.11555600e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.10500000e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]]

 [[ 3.42666700e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.11555600e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.10500000e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.12000000e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]]

 [[ 3.11555600e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.10500000e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.12000000e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]
  [ 3.10444400e+02  8.00000000e+00  9.60000000e+00 ...  0.00000000e+00
   -2.58819045e-01  9.65925826e-01]]]

 (407365, 4)
[[ 0.      76.1111  7.      0.    ]
 [ 0.      75.2727  7.      0.    ]
 [ 0.      75.      7.      0.    ]
 ...
 [ 0.       0.     65.5     0.    ]
 [ 0.       0.     65.7778  0.    ]
 [ 0.       0.     65.3333  0.    ]]
```

In [47]:
```
cnn2 = Sequential()

cnn2.add(Conv1D(64,input_shape=(4, 18),kernel_size=2,padding='same',activation='relu',strid
es=1))
cnn2.add(MaxPooling1D(pool_size=2))
# as the input now spans over 4 rows of data (shape (4,18)), we have to flatten it before t
he dense layer:
cnn2.add(Flatten())
cnn2.add(Dense(64,activation='relu'))
cnn2.add(Dense(4, activation='linear'))
cnn2.compile(loss='mean_squared_error',optimizer='adam')

cnn2.summary()
```

```
WARNING:tensorflow:From /Users/netflix/opt/anaconda3/lib/python3.7/site-packages/keras/bac
kend/tensorflow_backend.py:4070: The name tf.nn.max_pool is deprecated. Please use tf.nn.m
ax_pool2d instead.

Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_2 (Conv1D)            (None, 4, 64)             2368
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 2, 64)             0
_________________________________________________________________
flatten_1 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_7 (Dense)              (None, 4)                 260
=================================================================
Total params: 10,884
Trainable params: 10,884
Non-trainable params: 0
_________________________________________________________________
```

In [48]:
```
cnn2.fit(input_train_cnn2, output_train_cnn2, batch_size=32, epochs=6, verbose=1)
```

```
Epoch 1/6
417596/417596 [==============================] - 1130s 3ms/step - loss: 3245.0590
Epoch 2/6
417596/417596 [==============================] - 31s 75us/step - loss: 3072.9655
Epoch 3/6
417596/417596 [==============================] - 24s 57us/step - loss: 2974.0088
Epoch 4/6
417596/417596 [==============================] - 24s 57us/step - loss: 2902.7288
Epoch 5/6
417596/417596 [==============================] - 24s 57us/step - loss: 2829.1071
Epoch 6/6
417596/417596 [==============================] - 27s 65us/step - loss: 2818.4827
```

Out[48]:
```
<keras.callbacks.callbacks.History at 0x13b606410>
```

After 6 epochs of training, loss seems to be much better than with previous networks, at 2818 (vs. over 3200 before). Unfortunately, when sumitting online, we got very bad scores, around 80... which seems to reveal that somthing is wrong with this model. Our reasonning (using several rows of data as input to help the model learn the time nature of data) seemed interesting to us, but practice revealed it didn't work out as good as expected!

As a last try for deep learning, we implemented recurrent neural networks.

RNN with LSTM

Recurrent neural networks, and especially LSTM cells, are supposed to be very effective with sequential data, as they have the ability to "remember" about the earlier steps in the sequence. Each cell is connected not only with the next layer, but with the other cells in its own layer, which allows for information to flow and for the network to learn sequential patterns.

In [212]:
```
from keras.layers.recurrent import LSTM
from sklearn.preprocessing import MinMaxScaler
```

```
Using TensorFlow backend.
```

After some sequence pre-processing and rescaling to adapt input to a LSTM recurrent neural network, we were able to train our model:

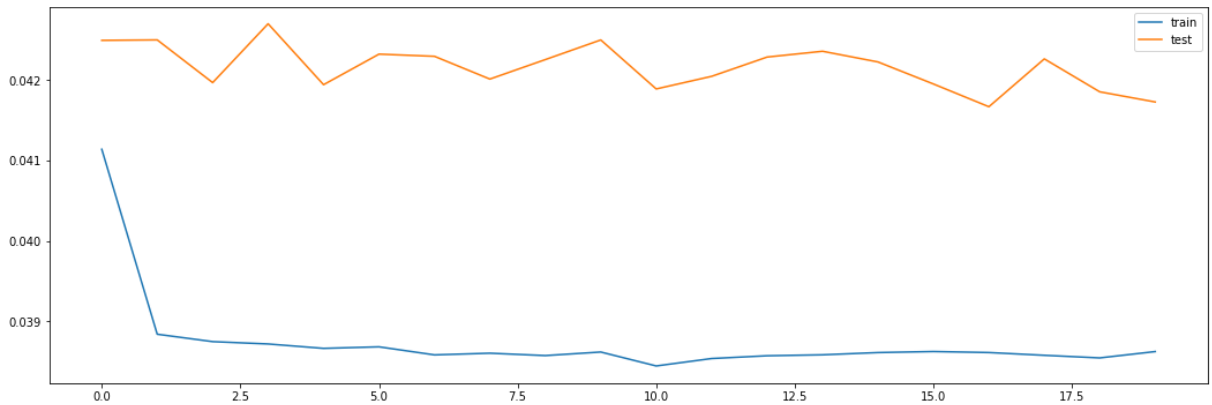
In [56]:
```
model = Sequential()
model.add(LSTM(100, input_shape=(train_X.shape[1], train_X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(4))
#model.add(Activation('softmax'))
model.compile(loss='mae', optimizer='adam')
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_3 (LSTM)                (None, 50)                13800
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 204
=================================================================
Total params: 14,004
Trainable params: 14,004
Non-trainable params: 0
_________________________________________________________________
```

In [80]:
```
history = model.fit(train_X, train_y, epochs=5, batch_size=64,
                    validation_data=(test_X, test_y), verbose=2, shuffle=False)
```

```
Train on 300000 samples, validate on 117597 samples
Epoch 1/5
 - 16s - loss: 0.0414 - val_loss: 0.0425
Epoch 2/5
 - 16s - loss: 0.0387 - val_loss: 0.0422
Epoch 3/5
 - 15s - loss: 0.0385 - val_loss: 0.0422
Epoch 4/5
 - 16s - loss: 0.0384 - val_loss: 0.0418
Epoch 5/5
 - 16s - loss: 0.0383 - val_loss: 0.0415
```

In [58]:
```
import matplotlib.pyplot as pyplot
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```



As evidenced in the graph above, train and test errors are close to each other (0.039 and 0.042, which means very few overfitting), but test error doesn't improve a lot with training.

In fact, results were disappointing. An interesting thing we tried to do was to clip unrealistic predictions, i.e. set to 0 when predictions where negative (as the RNN sometimes outputted negative values). This improved performance a little bit, but in the end results of RNN/LSTM were below our expectations, with online scores no better than the benchmark.

This is why, after exploring deep learning, **we refocused on machine learning methods that worked better, and especially our final model: XGBoost**. Below are 2 very short sections about other model explorations (NILM and time series forecasting), and the final section with our final XGBoost model.

NILM disaggregation algorithms

While learning online about the particular problem this challenge tackles, we tumbled upon a Python API designed specifically for Non-Intrusive Load Monitoring problems, the NILMTK (https://github.com/nilmtk/nilmtk) library (Non Intrusive Load Monitoring Toolkit). See from documentation: disaggregation and metrics (https://github.com/nilmtk/nilmtk/blob/master/docs/manual/user_guide/disaggregation_and_metrics.ipynb), data grouping and basic statistics (https://github.com/nilmtk/nilmtk/blob/master/docs/manual/user_guide/elecmeter_and_metergroup.ipynb).

This library provides 4 disaggegration algorithms (implemented as python classes) to predict appliance consumption based on total building consumption: **Combinatorial optimisation (CO)**, **FHMM - factorial hidden Markov model** (using exact inference), an implementation of **George Hart's 1985 disaggregation algorithm**, and a **Maximum Likelihood Estimation** algorithm.

We tried to use these tools, but unfortunately (apart from the fact that these algorithms are completely new to us), the NILMTK methods were very specific to a few datasets used in research (especially the REDD dataset, which has a specific data format, with complex classes and objects). Therefore the library mostly serves to replicate results from research on these public datasets.

To adapt the library and disaggregation algorithms to our problem would require a very significant amount of work, so we - sadly - abandoned the idea to use this powerful-looking API... but if we have more time in the future, this adaptation work looks very interesting, and we would definitely look further into it!

Time series forecasting using only y_train (Vector Autoregression, SARIMAX)

We tried to take the problem as a standard time series problem, i.e. predict the 4 appliances (`y_train`) based on past time steps with VAR (Vector Auto-Regression model, i.e. autoregression with several time series) and SARIMAX (an efficient variant of ARIMA, the Autoregressive Integrated Moving Average).

The train set has data from March 2013 to January 2014, and the test set has data from January to June 2014. These periods follow each other, so we thought that we could start from the train data, and iteratively predict each following time step in the test data with autoregression.

Unfortunately, results were disappointing, and we turned back to machine learning and to the more efficient XGBoost model, presented in the next (and last) section of this report.

In [89]:
```
from statsmodels.tsa.vector_ar.var_model import VAR

#fit the model with y_train data
model = VAR(endog=y_train)
model_fit = model.fit()

#make predictions for submission (226081 steps to predict ahead of train data)
y_hat = model_fit.forecast(model_fit.y, steps=226081)
print(y_hat)

# format to dataframe and add time_step index for submission
y_hat = pd.DataFrame(y_hat, columns = y_train.columns)
y_forecast = pd.concat([full_timesteps, y_hat], axis=1)
```

```
/anaconda3/lib/python3.6/site-packages/statsmodels/tsa/base/tsa_model.py:162: ValueWarnin
g: No frequency information was provided, so inferred frequency T will be used.
  % freq, ValueWarning)
/anaconda3/lib/python3.6/site-packages/statsmodels/base/wrapper.py:36: FutureWarning: y is
a deprecated alias for endog, will be removed in version 0.11.0
  obj = getattr(results, attr)
```

```
[[ 1.47666319  2.02130243 64.81298882  5.34615283]
 [ 2.35464732  3.96201444 64.29889264  7.57182252]
 [ 2.8803253   5.8246052  63.79061119  8.49245995]
 ...
 [ 5.77773947 49.93682391 14.49599786  5.0003242 ]
 [ 5.77773947 49.93682391 14.49599786  5.0003242 ]
 [ 5.77773947 49.93682391 14.49599786  5.0003242 ]]
```

In [304]:
```
from statsmodels.tsa.statespace.sarimax import SARIMAX

for appliance in data.columns:
    # train-test split
    train = data[:train_size][appliance]
    test = data[train_size:][appliance]

    #fit the model
    model = SARIMAX(endog=train, exog=exog) #endog is the target, exog the df of additional
features
    model_fit = model.fit()

    # make prediction on validation
    prediction = model_fit.forecast(len(test), exog=exog_2)
    yhat = pd.DataFrame(prediction, columns = [appliance])
    yhat = yhat.reset_index().rename(columns={'index':'time_step'})

    # print scores for each appliance
    score_fore = rmse(np.array(test), yhat[appliance])
    print('score '+appliance+': ',score_fore)
```

```
score washing_machine:  60.22075798740973
score fridge_freezer:  51.72706696764762
score TV:  22.54438893640721
score kettle:  125.7318290000122
```

We submitted these predictions and we also tried to **average them with previous models**, but results weren't better than XGBoost, which we selected as our final model.

Our final model: XGBoost

The function below trains an XGBoost model for each of the 4 appliances and outputs a dataframe with all predictions (based on `input_test`). We tuned the model hyperparameters with grid search to find the best-performing ones. During training we use early stopping to stop training when validation error hasn't improved in 6 iterations.

In [326]:
```
def final_model(input_train, output_train, input_test):

    # train-test split
    train_size = int(len(input_train)*0.8)
    test_size = int(len(input_train)-train_size)
    X_train = input_train[:train_size]
    X_test = input_train[train_size:]
    y_train = output_train[:train_size]
    y_test = output_train[train_size:]

    # XGB training and predicting for each variable
    score = []
    pred = []
    for i in range(len(output_train.columns)):
        name = output_train.columns[i]
        print('****************** Training for {} ******************'.format(name))

        XG = XGBRegressor(max_depth=8,
                          objective='reg:squarederror',
                          n_estimators=1000,
                          min_child_weight=300,
                          colsample_bytree=0.8,
                          eta=0.3,
                          seed=42).fit(X_train, y_train[name],
                                       eval_metric="rmse",
                                       eval_set=[(X_train, y_train[name]), (X_test, y_test[
name])],
                                       verbose=True,
                                       early_stopping_rounds = 6)
        # store score
        pred_XG = XG.predict(X_test)
        score_XG = rmse(y_test[name], pred_XG)
        score.append(score_XG)

        # predict on input_test dataset, and store prediction for submission
        pred_XG = XG.predict(input_test)
        pred.append(pred_XG)

    pred = pd.DataFrame(np.asarray(pred).transpose(), columns=['washing_machine','fridge_fr
eezer','TV','kettle'])
    print('The best performance is on {} with RMSE at {}.'.format(output_train.columns[scor
e.index(min(score))], min(score)))

    return pred, score

pred, score = final_model(input_train, output_train, input_test)
pred.head()
```

```
****************** Training for washing_machine ******************
[0]     validation_0-rmse:55.9138       validation_1-rmse:54.3284
Multiple eval metrics have been passed: 'validation_1-rmse' will be used for early stoppin
g.

Will train until validation_1-rmse hasn't improved in 6 rounds.
[1]     validation_0-rmse:54.6153       validation_1-rmse:53.4407
[2]     validation_0-rmse:53.8567       validation_1-rmse:52.8793
[3]     validation_0-rmse:52.7394       validation_1-rmse:52.3557
[4]     validation_0-rmse:51.8213       validation_1-rmse:52.1161
[5]     validation_0-rmse:51.1267       validation_1-rmse:51.8688
[6]     validation_0-rmse:50.5732       validation_1-rmse:51.5028
[7]     validation_0-rmse:50.1731       validation_1-rmse:51.3472
[8]     validation_0-rmse:49.7662       validation_1-rmse:51.1123
[9]     validation_0-rmse:49.3418       validation_1-rmse:50.9585
[10]    validation_0-rmse:48.9785       validation_1-rmse:50.8118
[11]    validation_0-rmse:48.7253       validation_1-rmse:50.8052
[12]    validation_0-rmse:48.3464       validation_1-rmse:50.569
[13]    validation_0-rmse:47.9791       validation_1-rmse:50.3499
[14]    validation_0-rmse:47.765        validation_1-rmse:50.3702
[15]    validation_0-rmse:47.5332       validation_1-rmse:50.3317
[16]    validation_0-rmse:47.4228       validation_1-rmse:50.4024
[17]    validation_0-rmse:47.2707       validation_1-rmse:50.3913
[18]    validation_0-rmse:47.0585       validation_1-rmse:50.3204
[19]    validation_0-rmse:46.8671       validation_1-rmse:50.3499
[20]    validation_0-rmse:46.6337       validation_1-rmse:50.2191
[21]    validation_0-rmse:46.5563       validation_1-rmse:50.3021
[22]    validation_0-rmse:46.3795       validation_1-rmse:50.3418
[23]    validation_0-rmse:46.1393       validation_1-rmse:50.4022
[24]    validation_0-rmse:45.8183       validation_1-rmse:50.4235
[25]    validation_0-rmse:45.6707       validation_1-rmse:50.3623
[26]    validation_0-rmse:45.5512       validation_1-rmse:50.4108
Stopping. Best iteration:
[20]    validation_0-rmse:46.6337       validation_1-rmse:50.2191

****************** Training for fridge_freezer ******************
[0]     validation_0-rmse:67.8615       validation_1-rmse:67.241
Multiple eval metrics have been passed: 'validation_1-rmse' will be used for early stoppin
g.

Will train until validation_1-rmse hasn't improved in 6 rounds.
[1]     validation_0-rmse:63.8372       validation_1-rmse:63.5974
[2]     validation_0-rmse:60.3684       validation_1-rmse:60.6967
[3]     validation_0-rmse:57.3809       validation_1-rmse:58.2303
[4]     validation_0-rmse:54.7744       validation_1-rmse:56.1366
[5]     validation_0-rmse:52.5362       validation_1-rmse:54.3877
[6]     validation_0-rmse:50.6573       validation_1-rmse:52.8197
[7]     validation_0-rmse:49.0203       validation_1-rmse:51.5189
[8]     validation_0-rmse:47.6695       validation_1-rmse:50.5078
[9]     validation_0-rmse:46.5053       validation_1-rmse:49.6436
[10]    validation_0-rmse:45.6037       validation_1-rmse:48.8145
[11]    validation_0-rmse:44.7908       validation_1-rmse:48.2932
[12]    validation_0-rmse:44.071        validation_1-rmse:47.8243
[13]    validation_0-rmse:43.469        validation_1-rmse:47.4422
[14]    validation_0-rmse:42.9648       validation_1-rmse:47.1618
[15]    validation_0-rmse:42.5292       validation_1-rmse:46.866
[16]    validation_0-rmse:42.1368       validation_1-rmse:46.662
[17]    validation_0-rmse:41.8276       validation_1-rmse:46.4508
[18]    validation_0-rmse:41.5569       validation_1-rmse:46.3019
[19]    validation_0-rmse:41.2902       validation_1-rmse:46.1387
[20]    validation_0-rmse:41.0693       validation_1-rmse:46.012
[21]    validation_0-rmse:40.8524       validation_1-rmse:45.9204
[22]    validation_0-rmse:40.6567       validation_1-rmse:45.8654
[23]    validation_0-rmse:40.4833       validation_1-rmse:45.7873
[24]    validation_0-rmse:40.3405       validation_1-rmse:45.7275
[25]    validation_0-rmse:40.1734       validation_1-rmse:45.6982
[26]    validation_0-rmse:40.0436       validation_1-rmse:45.6603
[27]    validation_0-rmse:39.9225       validation_1-rmse:45.6117
[28]    validation_0-rmse:39.812        validation_1-rmse:45.605
[29]    validation_0-rmse:39.7075       validation_1-rmse:45.6195
[30]    validation_0-rmse:39.6243       validation_1-rmse:45.6099
[31]    validation_0-rmse:39.5452       validation_1-rmse:45.6367
[32]    validation_0-rmse:39.4567       validation_1-rmse:45.7138
[33]    validation_0-rmse:39.372        validation_1-rmse:45.6906
[34]    validation_0-rmse:39.3022       validation_1-rmse:45.6855
Stopping. Best iteration:
```

```
[28]    validation_0-rmse:39.812        validation_1-rmse:45.605

****************** Training for TV ******************
[0]     validation_0-rmse:21.1037       validation_1-rmse:21.0481
Multiple eval metrics have been passed: 'validation_1-rmse' will be used for early stoppin
g.

Will train until validation_1-rmse hasn't improved in 6 rounds.
[1]     validation_0-rmse:19.8302       validation_1-rmse:20.005
[2]     validation_0-rmse:18.7744       validation_1-rmse:19.1618
[3]     validation_0-rmse:17.8207       validation_1-rmse:18.4544
[4]     validation_0-rmse:17.0454       validation_1-rmse:17.81
[5]     validation_0-rmse:16.3025       validation_1-rmse:17.2887
[6]     validation_0-rmse:15.683        validation_1-rmse:16.874
[7]     validation_0-rmse:15.1628       validation_1-rmse:16.5279
[8]     validation_0-rmse:14.7038       validation_1-rmse:16.2731
[9]     validation_0-rmse:14.292        validation_1-rmse:16.0514
[10]    validation_0-rmse:13.9219       validation_1-rmse:15.8569
[11]    validation_0-rmse:13.6019       validation_1-rmse:15.7424
[12]    validation_0-rmse:13.3469       validation_1-rmse:15.6125
[13]    validation_0-rmse:13.1077       validation_1-rmse:15.5441
[14]    validation_0-rmse:12.8416       validation_1-rmse:15.512
[15]    validation_0-rmse:12.6661       validation_1-rmse:15.4674
[16]    validation_0-rmse:12.4791       validation_1-rmse:15.4134
[17]    validation_0-rmse:12.3201       validation_1-rmse:15.3907
[18]    validation_0-rmse:12.1568       validation_1-rmse:15.3576
[19]    validation_0-rmse:12.0058       validation_1-rmse:15.3541
[20]    validation_0-rmse:11.9123       validation_1-rmse:15.3542
[21]    validation_0-rmse:11.7688       validation_1-rmse:15.3826
[22]    validation_0-rmse:11.6597       validation_1-rmse:15.4041
[23]    validation_0-rmse:11.5787       validation_1-rmse:15.437
[24]    validation_0-rmse:11.5081       validation_1-rmse:15.4383
[25]    validation_0-rmse:11.4153       validation_1-rmse:15.452
Stopping. Best iteration:
[19]    validation_0-rmse:12.0058       validation_1-rmse:15.3541

****************** Training for kettle ******************
[0]     validation_0-rmse:85.5601       validation_1-rmse:146.079
Multiple eval metrics have been passed: 'validation_1-rmse' will be used for early stoppin
g.

Will train until validation_1-rmse hasn't improved in 6 rounds.
[1]     validation_0-rmse:82.4064       validation_1-rmse:141.069
[2]     validation_0-rmse:79.7531       validation_1-rmse:136.722
[3]     validation_0-rmse:77.4385       validation_1-rmse:132.926
[4]     validation_0-rmse:75.5076       validation_1-rmse:129.658
[5]     validation_0-rmse:73.855        validation_1-rmse:126.727
[6]     validation_0-rmse:72.4227       validation_1-rmse:124.296
[7]     validation_0-rmse:71.2245       validation_1-rmse:122.27
[8]     validation_0-rmse:70.2751       validation_1-rmse:120.524
[9]     validation_0-rmse:69.3497       validation_1-rmse:118.87
[10]    validation_0-rmse:68.622        validation_1-rmse:117.552
[11]    validation_0-rmse:67.9895       validation_1-rmse:116.409
[12]    validation_0-rmse:67.3888       validation_1-rmse:114.968
[13]    validation_0-rmse:66.9828       validation_1-rmse:114.055
[14]    validation_0-rmse:66.5305       validation_1-rmse:113.37
[15]    validation_0-rmse:66.1789       validation_1-rmse:112.786
[16]    validation_0-rmse:65.9301       validation_1-rmse:112.444
[17]    validation_0-rmse:65.6753       validation_1-rmse:111.902
[18]    validation_0-rmse:65.4471       validation_1-rmse:111.407
[19]    validation_0-rmse:65.2403       validation_1-rmse:110.912
[20]    validation_0-rmse:65.0913       validation_1-rmse:110.458
[21]    validation_0-rmse:64.9199       validation_1-rmse:110.11
[22]    validation_0-rmse:64.7128       validation_1-rmse:109.693
[23]    validation_0-rmse:64.5824       validation_1-rmse:109.305
[24]    validation_0-rmse:64.4609       validation_1-rmse:109.145
[25]    validation_0-rmse:64.3404       validation_1-rmse:108.886
[26]    validation_0-rmse:64.2591       validation_1-rmse:108.607
[27]    validation_0-rmse:64.029        validation_1-rmse:108.1
[28]    validation_0-rmse:63.9065       validation_1-rmse:107.769
[29]    validation_0-rmse:63.8053       validation_1-rmse:107.593
[30]    validation_0-rmse:63.7011       validation_1-rmse:107.297
[31]    validation_0-rmse:63.6326       validation_1-rmse:107.152
[32]    validation_0-rmse:63.5748       validation_1-rmse:107.017
[33]    validation_0-rmse:63.4989       validation_1-rmse:106.851
[34]    validation_0-rmse:63.4124       validation_1-rmse:106.603
[35]    validation_0-rmse:63.3412       validation_1-rmse:106.407
```

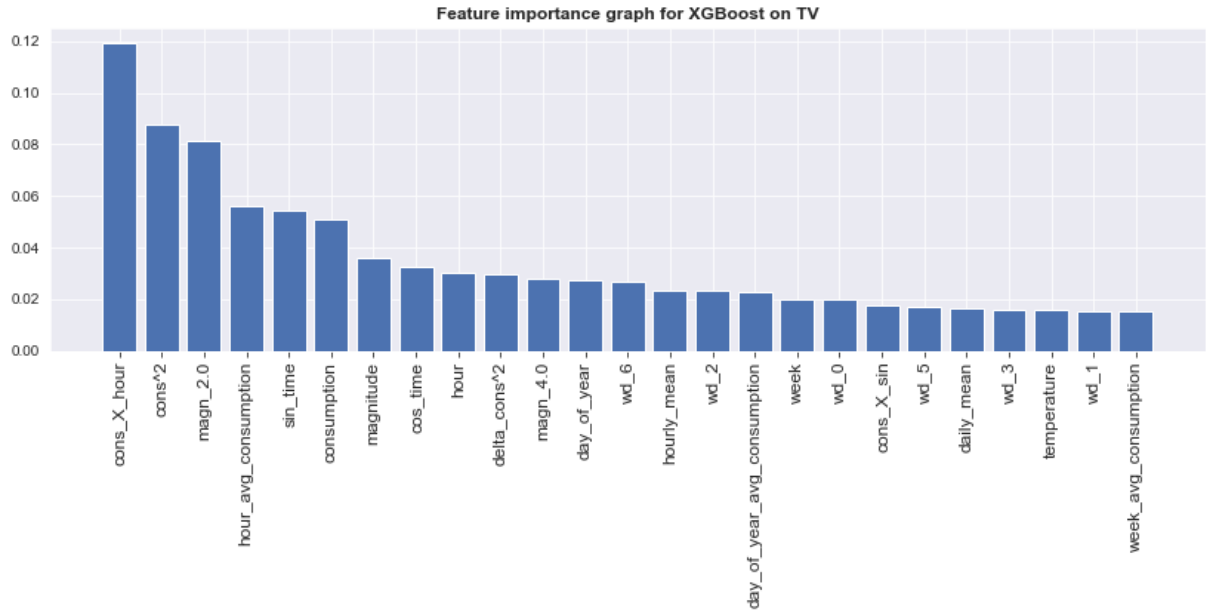
```
[36]    validation_0-rmse:63.2855       validation_1-rmse:106.302
[37]    validation_0-rmse:63.2109       validation_1-rmse:106.057
[38]    validation_0-rmse:63.1598       validation_1-rmse:105.912
[39]    validation_0-rmse:63.1053       validation_1-rmse:105.847
[40]    validation_0-rmse:63.0522       validation_1-rmse:105.906
[41]    validation_0-rmse:63.0047       validation_1-rmse:105.889
[42]    validation_0-rmse:62.9664       validation_1-rmse:105.814
[43]    validation_0-rmse:62.9052       validation_1-rmse:105.71
[44]    validation_0-rmse:62.8589       validation_1-rmse:105.663
[45]    validation_0-rmse:62.8138       validation_1-rmse:105.709
[46]    validation_0-rmse:62.7446       validation_1-rmse:105.717
[47]    validation_0-rmse:62.6705       validation_1-rmse:105.598
[48]    validation_0-rmse:62.6173       validation_1-rmse:105.481
[49]    validation_0-rmse:62.5591       validation_1-rmse:105.308
[50]    validation_0-rmse:62.4998       validation_1-rmse:105.162
[51]    validation_0-rmse:62.452        validation_1-rmse:105.09
[52]    validation_0-rmse:62.3909       validation_1-rmse:104.934
[53]    validation_0-rmse:62.3497       validation_1-rmse:104.889
[54]    validation_0-rmse:62.3063       validation_1-rmse:104.794
[55]    validation_0-rmse:62.2593       validation_1-rmse:104.743
[56]    validation_0-rmse:62.2263       validation_1-rmse:104.774
[57]    validation_0-rmse:62.1743       validation_1-rmse:104.824
[58]    validation_0-rmse:62.1365       validation_1-rmse:104.791
[59]    validation_0-rmse:62.0654       validation_1-rmse:104.811
[60]    validation_0-rmse:62.0285       validation_1-rmse:104.735
[61]    validation_0-rmse:61.9944       validation_1-rmse:104.548
[62]    validation_0-rmse:61.9597       validation_1-rmse:104.628
[63]    validation_0-rmse:61.9158       validation_1-rmse:104.534
[64]    validation_0-rmse:61.8861       validation_1-rmse:104.379
[65]    validation_0-rmse:61.8692       validation_1-rmse:104.418
[66]    validation_0-rmse:61.8252       validation_1-rmse:104.381
[67]    validation_0-rmse:61.7926       validation_1-rmse:104.289
[68]    validation_0-rmse:61.7648       validation_1-rmse:104.143
[69]    validation_0-rmse:61.731        validation_1-rmse:104.135
[70]    validation_0-rmse:61.7003       validation_1-rmse:104.197
[71]    validation_0-rmse:61.6655       validation_1-rmse:104.26
[72]    validation_0-rmse:61.6325       validation_1-rmse:104.102
[73]    validation_0-rmse:61.6019       validation_1-rmse:104.06
[74]    validation_0-rmse:61.5407       validation_1-rmse:104.088
[75]    validation_0-rmse:61.5107       validation_1-rmse:104.044
[76]    validation_0-rmse:61.4831       validation_1-rmse:103.928
[77]    validation_0-rmse:61.3558       validation_1-rmse:103.703
[78]    validation_0-rmse:61.3166       validation_1-rmse:103.68
[79]    validation_0-rmse:61.2966       validation_1-rmse:103.75
[80]    validation_0-rmse:61.258        validation_1-rmse:103.729
[81]    validation_0-rmse:61.2317       validation_1-rmse:103.795
[82]    validation_0-rmse:61.2072       validation_1-rmse:103.823
[83]    validation_0-rmse:61.1808       validation_1-rmse:103.753
[84]    validation_0-rmse:61.1454       validation_1-rmse:103.805
Stopping. Best iteration:
[78]    validation_0-rmse:61.3166       validation_1-rmse:103.68

The best performance is on TV with RMSE at 15.354849851166408.
```

Out[326]:

	washing_machine	fridge_freezer	TV	kettle
0	7.257164	99.082115	8.982498	0.021016
1	7.278738	108.829414	8.982498	0.021016
2	7.278738	111.149956	8.982498	0.021016
3	7.257164	111.149956	8.982498	0.021016
4	7.257164	111.149956	8.982498	0.021016

This last plot allows us to see the importance of the 25 most relevant features when predicting `TV` consumption. Interestingly, the most important feature appears to be the `consumption * hour_of_the_day` feature that we built in the Feature Engineering section.

In [20]:
```
X_columns = input_train.columns
ordering = np.argsort(XG.feature_importances_)[::-1][:25]
importances = XG.feature_importances_[ordering]
feature_names = X_columns[ordering]
x = np.arange(len(feature_names))

plt.figure(figsize = (14, 4))
plt.bar(x, importances)
plt.xticks(x, feature_names, rotation=90, fontsize=12)
plt.title('Feature importance graph for XGBoost on TV', fontsize=12, fontweight='semibold')
plt.show()
```



With this final model, the validation scores we obtained locally are:

```
score washing_machine:  50.21
score fridge_freezer:  45.60
score TV:  15.35
score kettle:  103.68
```

And our score in the online challenge was: `41.24`.

Conclusion

In this project, we learned a lot about manipulating time series, preprocessing data, testing models and how all these steps interact together. An interesting part was also the data vizualization part, particularly relevant for sequential data. We allocated a fair amount of time to it in order to produce an in-depth visual analysis of the data.

To improve our final model, we see several potential paths. First, we realized that the most effective step in the workflow was feature engineering. With other features based on total consumption, and other rounds of testing, we could probably improve performance. We could also go through more parameter tuning, with new grid searches in order to push performance a little forward again. Finally, we could also allocate more time to testing deep learning models, because the disappointing results we got in this area could probably be improved with different preprocessing, or different network structures - with more time we could, for sure, further explore this domain.

All in all, we feel like in this project we were able to apply knowledge from the Machine Learning II course and to learn a lot personnaly. We spent a lot of time browsing the internet to learn about various methods and to find solutions to our problems, and this made us progress a lot - in coding, there's nothing better than learning by doing!