Online News Popularity

January 9, 2021

1 Table of contents

3.2 Formating the dataset

3.2.1 Dataset's description

The dataset's description specifically considers the first two features ***url*** and ***timedelta*** as non-predictive. Let us remove them from the main dataset. Nevertheless, we'll store them elsewhere for a potential alternative model to compare.

[ ]:
```
data_url = data['url']
data_timedelta = data[' timedelta']
data = data.drop(['url', ' timedelta'], 1)
```

3.2.2 Dataset's columns

[ ]:
```
data.columns
```

We rename the columns because they have a space in it. It will be easier to handle further the line.

[ ]:
```
data.columns = data.columns.str.replace(' ', '')
```

[ ]:
```
data.columns = data.columns.str.replace('ss', 's')
```

3.2.3 Dataset's missing and misc. values

Let us check for any missing or duplicate data.

[ ]:
```
data_url.nunique() == len(data_url)
```

[ ]:
```
any(data.isna().sum() != 0)
```

It seems that there are neither *NA* nor duplicated values. However, when transforming some features, the dataset revealed some values that could be considered *NA* in spirit. For example, let us examine the ***rate_positive_words*** and ***rate_negative_words*** case :

[ ]:
```
data[['rate_positive_words',
      'rate_negative_words']].head(10)
```

We can observe that the sum of the two columns is always equal to 1. This is no coincidence as the following command showcases the $y = 1 - x$ relationship that the pair of features respect :

[ ]:
```
(data["rate_positive_words"] + data["rate_negative_words"] == 1).value_counts()
```

There is a non-negligeable error rate on our presumed relationship. Let us check the indexes that are allegedly to be exceptions to our rule :

[ ]:
```
data[data['rate_positive_words'] + data['rate_negative_words'] !=␣
↪1][['rate_positive_words', 'rate_negative_words']]
```

Undeniably, $0.827957 + 0.172043 = 1.000000$. So this must be a mistake on either Python's or the source's end. Let's suppose that Python returns a correct result with 1% error :

```
[ ]: data[data['rate_positive_words'] + data['rate_negative_words'] < 0.
     ↪99][['rate_positive_words', 'rate_negative_words']]
```

We can easily deduce that the remaining indexes were not correctly set up by the algorithm that put up the dataset. In case one might still believe in an article that has 0% rate of positive words and negative words, let's check the other features present for these indexes :

```
[ ]: data[data['rate_positive_words'] + data['rate_negative_words'] < 0.99]
```

As we can see, most of the values are absent. When checking one of the urls, we were fully certain that this was a mistake that needed to be dropped from the dataset.

```
[ ]: index = data.index[data['rate_positive_words'] + data['rate_negative_words'] ==␣
     ↪0.00].tolist()
     data = data.drop(index, 0)
```

3.2.4 Dataset's outliers

Let's have an overall look at the distribution of the features as well as the target ***shares***.

```
[ ]: plt.rcParams["figure.figsize"] = 20,25
     fig = data[data.columns].hist()
```

Some histograms are less compressed while other remain the overall the same. This source of error might came from outliers in our dataset. We can easily check in some of the histograms above, for example ***n_tokens_title***, that the x-axis is extended for invisible values on the y-axis. Before discussing the most appropriate reaction, let's examine the values that have 1 in 10^50 chance to exceed the mean, that is to say, the D values such as $|D - \mu_{feature}| > 15 \cdot \sigma_{feature}$.

```
[ ]: potential_outliers = ['n_tokens_content', 'num_hrefs', 'num_self_hrefs',␣
     ↪'num_imgs', 'num_videos',
                           'kw_min_min', 'kw_max_min', 'kw_avg_min', 'kw_min_max',␣
     ↪'kw_max_max', 'kw_max_avg', 'kw_avg_avg',
                           'self_reference_min_shares', 'self_reference_max_shares',␣
     ↪'self_reference_avg_shares']

     for column in potential_outliers :
         mean = data[column].mean()
         sigma = data[column].std()
         D = list()
         for d in data[column] :
             if abs(d - mean) > 15 * sigma :
                 D.append(d)
         print(column, "mean =", round(mean, 2), "number of outliers =", len(D))
```

The outlier criteria is intrinsically related to its standard deviation. Thus, we won't consider the features who are already in the $[-1, 1]$ interval but only those whose histograms are compressed to surprising extreme values. After that, we'll check that the number of outliers in ***shares*** hasn't significatively change, this would mean that the extreme values were not necessary.

[ ]:
```
def outliers_indexes(column, degree) :
    mean = data[column].mean()
    sigma = data[column].std()
    indexes = list()
    for i in data.index :
        if abs(data[column][i] - mean) > degree * sigma :
            indexes.append(i)
    return indexes

data = data.drop(outliers_indexes('n_tokens_content', 15), axis = 0)
data = data.drop(outliers_indexes('num_hrefs', 15), axis = 0)
data = data.drop(outliers_indexes('num_self_hrefs', 15), axis = 0)
data = data.drop(outliers_indexes('num_imgs', 15), axis = 0)
data = data.drop(outliers_indexes('num_videos', 15), axis = 0)
data = data.drop(outliers_indexes('kw_min_min', 15), axis = 0)
data = data.drop(outliers_indexes('kw_max_min', 15), axis = 0)
data = data.drop(outliers_indexes('kw_avg_min', 15), axis = 0)
data = data.drop(outliers_indexes('kw_min_max', 15), axis = 0)
data = data.drop(outliers_indexes('kw_max_max', 15), axis = 0)
data = data.drop(outliers_indexes('kw_max_avg', 15), axis = 0)
data = data.drop(outliers_indexes('kw_avg_avg', 15), axis = 0)
data = data.drop(outliers_indexes('self_reference_min_shares', 15), axis = 0)
data = data.drop(outliers_indexes('self_reference_max_shares', 15), axis = 0)
data = data.drop(outliers_indexes('self_reference_avg_shares', 15), axis = 0)
```

Let's plot once more the histograms.

[ ]:
```
plt.rcParams["figure.figsize"] = 20,25
fig = data['num_hrefs'].hist()
```

Undeniably, we have an easier time to estimate the different distributions, for example ***num_hrefs*** appears to be following an exponential distribution whereas in the precedent histogram it looked like one range of values dominated the rest outside that range. Let's check if we've significantly changed the explained variable after all these changes :

[ ]:
```
data.shares.describe()
```

[ ]:
```
pd.read_csv("OnlineNewsPopularity.csv")[' shares'].describe()
```

We can see that the original dataset doesn't really differ to the transformed one. We've lost 3% of the dataset and 5% of standard deviation which was likely the result of noise as we've explained.

3.3 Transforming the dataset

3.3.1 Generic methods for visualization and transformation

We write the different methods that enable us to plot histograms, density plots, probability-probability plots as well as to transform data as we please.

[ ]:
```
from scipy import stats
from scipy.stats import norm, skew
from sklearn.preprocessing import StandardScaler

# examine() plots a column's density plot as well as prints out the skewness␣
 ↪and kurtosis of its curve
def examine(column) :
    sns.distplot(data[column])
    print('Skewness of the curve :', data[column].skew())
    print('Kurtosis of the curve :', data[column].kurt())

# transform() plots the before and after transformation of a column according␣
 ↪to a specified model.
# The different models correspond to the inverse of the law the column is␣
 ↪supposed to follow so as to normalize it.
# "Normal" standardizes the column, "Log" return the logarithm of the column,␣
 ↪"Log+1" and "Log+2" also but prevent log(0) values.
# (a, b) returns the inverse of the Beta distribution whose parameters are␣
 ↪alpha = a and beta = b.
def transform(column, method = "Normal") :
    fig = plt.figure(figsize = (15, 5))
    plt.subplot(1,2,1)
    sns.distplot(data[column], fit = norm);
    (mu, sigma) = norm.fit(data[column])
    plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu,␣
 ↪sigma)], loc = 1)
    plt.ylabel('Frequency')
    plt.title(str(column) + ' distribution')
    plt.subplot(1,2,2)
    res = stats.probplot(data[column], plot = plt)
    plt.suptitle('Before transformation')

    if method == "Normal" :
        data[column] = (data[column] - data[column].mean()) / data[column].std()
    elif method == "Log" :
        data[column] = np.log(data[column])
    elif method == "Log+1" :
        data[column] = np.log1p(data[column])
    elif method == "Log+2" :
        data[column] = np.log(data[column] + 2)
    else :
        data[column] = stats.beta.pdf(data[column], a = method[0], b =␣
 ↪method[1])
        data[column] = data[column] / data[column].max()

    fig = plt.figure(figsize = (15, 5))
```

```
    plt.subplot(1,2,1)
    sns.distplot(data[column], fit = norm);
    (mu, sigma) = norm.fit(data[column])
    plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu,␣
 ↪sigma)], loc=1)
    plt.ylabel('Frequency')
    plt.title(str(column) + ' distribution')
    plt.subplot(1,2,2)
    res = stats.probplot(data[column], plot = plt)
    plt.suptitle('After ' + str(method) + ' transformation')

# hist() returns the histogram of one or more columns
def hist(columns) :
    plt.rcParams["figure.figsize"] = (20, 20)
    fig = data[columns].hist()

# corrplot() returns the corrplots of two or more columns based on the␣
 ↪Pearson's or Spearman's correlation coefficient.
def corrplot(columns, method = 'pearson') :
    sns.heatmap(data[columns].corr(method),
                vmin = -1, vmax = 1, cmap = 'coolwarm',
                annot = True, square = True);
```

3.3.2 Examining the target *shares*

Let's examine the dependent variable ***shares***.

[ ]:
```
data['shares'].describe()
```

The target ***shares*** is expressed in the hundreds with some exceptions that are in the hundred thousands. However, most of the features' values are contained in the $[-1, 1]$ interval. To avoid overfitting and underfitting certain features, we'll transform all features whose values surpass the range of the $[-1, 1]$ domain. As for the target variable, we'll keep in mind the possibility to scale the values but won't at the moment, as it is rarely significant to do so (memory overflow).

3.3.3 Transforming the feature *n_tokens_title*

Let's examine the feature ***n_tokens_title***.

[ ]:
```
examine('n_tokens_title')
```

It seems fair to assume the probability distribution of ***n_tokens_title*** follows a normal law. Let's standardize it to follow $\mathcal{N}(0, 1)$ law in order to restrain it as much as possible to the $[-1, 1]$ interval.

[ ]:
```
transform('n_tokens_title')
```

3.3.4 Transforming the feature *n_tokens_content*

Let's examine the feature ***n_tokens_content***.

[ ]:
```
examine('n_tokens_content')
```

We can observe a probability distribution whose skewness, or asymmetry of the curve, and kurtosis, or tail of the curve, remind of an exponential law. By applying the exponential inverse function to our target ***shares***, the probability distribution shall be normalized. In turn, coefficients regulation methods such as lasso or ridge have an easier time dealing with this variable. Finally, through documentation the use of the **np.log1p** method which transforms the variable through the function $log(1+x)$ rather than the conventionnal **np.log** $log(x)$ method seemed more satisfying and prevented the undefined $log(0)$ value error.

[ ]:
```
transform('n_tokens_content', "Log+1")
```

3.3.5 Transforming the feature *n_non_stop_words*

Let's examine the feature ***n_non_stop_words***.

[ ]:
```
examine('n_non_stop_words')
```

The feature ***n_non_stop_words*** follows what appears to be an inverse exponential law. The results when normalizing through **np.log1m** and $\frac{X-\mu}{\sigma}$ were unsatisfying. We came to the conclusion that the distribution follows the Beta $\beta(\alpha,\beta)$ distribution whose parameters we shall estimate via la méthode des moments. Indeed, if $X \hookrightarrow \beta(\alpha,\beta)$, then $E(X) = \frac{\alpha}{\alpha+\beta}$ and $V(X) = \frac{\alpha\cdot\beta}{(\alpha+\beta)^2\cdot(\alpha+\beta+1)}$. If we estimate $E(X)$ and $V(X)$ by applying respectively **mean()** and **std()** to ***n_non_stop_words***, then we have $\alpha = k\cdot\beta$ and $\beta^2 + \frac{1}{k+1}\beta - \frac{k}{(k+1)^3\cdot V(X)} = 0$ where $k = \frac{E(X)}{1-E(X)}$. Let's first determine the parameters.

[ ]:
```
mean = data['n_non_stop_words'].mean()
var = data['n_non_stop_words'].std() ** 2

k = mean / (1 - mean)
delta = (1/(k+1))**2 - 4 * 1 * (- k / ((k+1)**3 * var)) # delta > 0
b1 = (-1/(k+1) + np.sqrt(delta)) / (2 * 1) # b1 > 0, possible solution
b2 = (-1/(k+1) - np.sqrt(delta)) / (2 * 1) # b2 < 0, impossible solution

beta = b1
alpha = k * beta
(alpha, beta)
```

When plotted on a casio the distribution fits. Let's now apply the beta density function **beta.pdf()** and the **max** scaling $x_{scale} = \frac{x}{max(x)}$.

[ ]:
```
transform('n_non_stop_words', (alpha, beta))
```

The probability has been scaled in a shape that we hope neither affects the model (overfitting/underfitting) nor the feature.

3.3.6 Transforming the feature *num_hrefs*

Let's examine the feature ***num_hrefs***.

[ ]:
```
examine('num_hrefs')
```

The feature ***num_hrefs*** follows what appears to be an exponential law.

[ ]:
```
transform('num_hrefs', "Log+1")
```

3.3.7 Transforming the feature *num_self_hrefs*

Let's examine the feature ***num_self_hrefs***.

[ ]:
```
examine('num_self_hrefs')
```

The feature ***num_self_hrefs*** follows what appears to be an exponential law.

[ ]:
```
transform('num_self_hrefs', "Log+1")
```

3.3.8 Transforming the feature *num_imgs*

Let's examine the feature ***num_imgs***.

[ ]:
```
examine('num_imgs')
```

The feature ***num_imgs*** follows what appears to be an exponential law.

[ ]:
```
transform('num_imgs', "Log+1")
```

3.3.9 Transforming the feature *num_videos*

Let's examine the feature ***num_videos***.

[ ]:
```
examine('num_videos')
```

The feature ***num_videos*** follows what appears to be an exponential law.

[ ]:
```
transform('num_videos', "Log+1")
```

3.3.10 Transforming the features *rate_positive_words* and *rate_negative_words*

Let's examine the pair of features ***rate_positive_words*** and ***rate_negative_words***.

[ ]:
```
corrplot(['rate_positive_words', 'rate_negative_words'])
```

[ ]:
```
corrplot(['rate_positive_words', 'rate_negative_words'], 'spearman')
```

Through both Pearson's and Spearman's correlation coefficient methods, the linear dependancy between ***rate_positive_words*** and ***rate_negative_words*** is undeniable. To prevent the model to overfit by adjusting the weight he gives to both features, we shall remove the latter one. Indeed, if we know the value of the first feature we know exactly the value of the other one, the explained variance from both features is the same.

[ ]:
```
data = data.drop('rate_negative_words', 1)
```

3.3.11 Transforming the features *keywords*

Let's examine the features ***kw_avg_avg***, ***kw_avg_max***, ***kw_avg_min***, ***kw_max_avg***, ***kw_max_max***, ***kw_max_min***, ***kw_min_avg***, ***kw_min_max*** and ***kw_min_min***.

[ ]:
```
corrplot(['kw_min_min', 'kw_max_min', 'kw_avg_min', 'kw_min_max',
          'kw_max_max', 'kw_avg_max', 'kw_min_avg', 'kw_max_avg', 'kw_avg_avg'])
```

[ ]:
```
corrplot(['kw_min_min', 'kw_max_min', 'kw_avg_min', 'kw_min_max',
          'kw_max_max', 'kw_avg_max', 'kw_min_avg', 'kw_max_avg',␣
 ↪'kw_avg_avg'], 'spearman')
```

For the same reasons as before, we'll remove the ***kw_avg_...*** features, as they are intrinsically defined by ***kw_min_...*** and ***kw_max_...***. Moreover, the average is likely estimated from the arithmetic mean $m = \frac{a+b}{2}$. On the contrary to the geometric mean $m = \sqrt{a \cdot b}$, the arithmetic mean is heavily penalized by extreme values, thus the weights of ***kw_min_...*** and ***kw_max_...*** on the average is significant, as showcased by the linear correlation plots above.

[ ]:
```
data = data.drop(['kw_avg_min', 'kw_avg_max', 'kw_avg_avg'], 1)
```

[ ]:
```
hist(['kw_min_min', 'kw_max_min', 'kw_min_max', 'kw_max_max', 'kw_min_avg',␣
 ↪'kw_max_avg'])
```

Let us apply **np.log** to all features.

[ ]:
```
transform('kw_min_min', 'Log+2')
```

[ ]:
```
transform('kw_max_min', 'Log+1')
```

[ ]:
```
transform('kw_min_max', 'Log+1')
```

[ ]:
```
transform('kw_max_max', 'Log+1')
```

[ ]:
```
transform('kw_min_avg', 'Log+2')
```

[ ]:
```
transform('kw_max_avg', 'Log+1')
```

Most of the P-P plots are disfigured. This can be the consequence of a poor variable defintion or the persistence of outliers but only applied to the ***keywords*** attributes.

3.3.12 Transforming the features *self_reference*

Let's examine the features ***self_reference_min_shares***, ***self_reference_max_shares*** and ***self_reference_avg_shares***.

[ ]:
```
corrplot(['self_reference_min_shares', 'self_reference_max_shares',␣
 ↪'self_reference_avg_shares'])
```

[ ]:
```
corrplot(['self_reference_min_shares', 'self_reference_max_shares',␣
 ↪'self_reference_avg_shares'], 'spearman')
```

For the same reasons as before, we'll remove the ***self_reference_avg_shares*** feature.

[ ]:
```
data = data.drop('self_reference_avg_shares', 1)
```

[ ]:
```
hist(['self_reference_min_shares', 'self_reference_max_shares'])
```

[ ]:
```
transform('self_reference_min_shares', 'Log+1')
```

[ ]:
```
transform('self_reference_max_shares', 'Log+1')
```

3.3.13 Transforming the features *data_channel*

Unfortunately, the ***data_channel*** features are not all present. Indeed, we can see that some articles are classified under none of the categories :

[ ]:
```
data[data['data_channel_is_lifestyle'] + data['data_channel_is_entertainment']␣
↪+ data['data_channel_is_bus'] +
      data['data_channel_is_socmed'] + data['data_channel_is_tech'] +␣
↪data['data_channel_is_world'] != 1][[
   'data_channel_is_lifestyle', 'data_channel_is_entertainment',␣
↪'data_channel_is_bus',
   'data_channel_is_socmed', 'data_channel_is_tech', 'data_channel_is_world']]
```

[ ]:
```
corrplot(['data_channel_is_lifestyle', 'data_channel_is_entertainment',␣
↪'data_channel_is_bus',
      'data_channel_is_socmed', 'data_channel_is_tech',␣
↪'data_channel_is_world'])
```

Evidently, all features are negatively correlated to each other, but there are no real $y = -x$ relations. We shall restrain ourselves to transform these features.

3.3.14 Transforming the features *weekday* and *is_weekend*

[ ]:
```
box = data.boxplot(column=['shares'], by=['weekday_is_sunday',␣
↪'weekday_is_saturday', 'weekday_is_friday',
                                          'weekday_is_thursday',␣
↪'weekday_is_wednesday', 'weekday_is_tuesday',
                                          'weekday_is_monday'], return_type =␣
↪None, grid = False, rot = 45, fontsize = 15)
```

Let's put a log10 scale on the y-axis.

[ ]:
```
datalog10 = data.copy()
datalog10.shares = np.log1p(datalog10.shares)
datalog10.boxplot(column=['shares'], by=['weekday_is_sunday',␣
↪'weekday_is_saturday', 'weekday_is_friday',
                                          'weekday_is_thursday',␣
↪'weekday_is_wednesday', 'weekday_is_tuesday',
```

```
                                                'weekday_is_monday'], return_type =␣
↪None, grid = False, rot = 45, fontsize = 15)
```

We can see that the tendancy to share during the end of the week (Friday to Sunday) is higher than the rest of the week. Moreover, the standard deviation of monday and wednesday is higher than tuesday and thursday. This all correlates to what we would expect in actuality. Evidently, ***weekday_is_sunday*** is correlated to the remaining ***weekday*** features. ***is_weekend*** is also correlated to both ***weekday_is_saturday*** and ***weekday_is_sunday***. For equivalent reasons to the ***rate_positive_words*** / ***rate_negative_words*** case, we'll omit these two features of the dataset.

[ ]:
```
data = data.drop(['weekday_is_sunday', 'is_weekend'], 1)
```

3.3.15 Transforming the features *LDA*

The Latent Dirichlet allocation (LDA) is a statistical method that allows the "gentrification" of the individuals following different topics. Here we suppose that every article is described by its "allegiance" to 5 different topics. These topics are mathematically established. A more known example in the academic field would be the Principal Component Analysis, which constructs artificial variables that suffice to explain the dataset, most of the time, the PCA reveals that most of the variance is explained by fewer number of variables than there are. As it is a probability, the law of total probability applies. Thus, we expect $LDA_{topic\,0} = 1 - \sum_{k=1}^{4} LDA_{topic\,k}$. Let's examine the set of features closely.

[ ]:
```
corrplot(['LDA_00', 'LDA_01', 'LDA_02', 'LDA_03', 'LDA_04'])
```

There isn't enough evidence to claim that two topics are related. It isn't a surprise since the LDA method constructs topics that are independent. Nevertheless, we'll remove ***LDA_00*** as it's a linear combination of the rest :

[ ]:
```
data['LDA'] = data['LDA_01'] + data['LDA_02'] + data['LDA_03'] + data['LDA_04']
corrplot(['LDA', 'LDA_00'])
```

[ ]:
```
data = data.drop(['LDA_00', 'LDA'], 1)
```

3.3.16 Transforming the features *global*

[ ]:
```
corrplot(['global_subjectivity', 'global_sentiment_polarity',␣
↪'global_rate_positive_words', 'global_rate_negative_words'])
```

[ ]:
```
corrplot(['global_subjectivity', 'global_sentiment_polarity',␣
↪'global_rate_positive_words', 'global_rate_negative_words'], 'spearman')
```

The ***global_sentiment_polarity*** is correlated to ***global_rate_positive_words*** and ***global_rate_negative_words***. So, we decide to remove the former and keep the two latters.

[ ]:
```
data = data.drop('global_sentiment_polarity', 1)
```

3.3.17 Transforming the features *title*

[ ]:
```
corrplot(['title_subjectivity', 'title_sentiment_polarity',␣
↪'abs_title_subjectivity', 'abs_title_sentiment_polarity'])
```

For the same reasons, we remove the ***abs_title_subjectivity*** and ***abs_title_sentiment_polarity***.

[ ]:
```
data = data.drop(['abs_title_subjectivity', 'abs_title_sentiment_polarity'], 1)
```

3.3.18 Transforming the features *polarity*

[ ]:
```
corrplot(['avg_positive_polarity', 'min_positive_polarity',␣
↪'max_positive_polarity',
      'avg_negative_polarity', 'min_negative_polarity',␣
↪'max_negative_polarity'])
```

We remove ***avg_positive_polarity*** and ***avg_negative_polarity***.

[ ]:
```
data = data.drop(['avg_positive_polarity', 'avg_negative_polarity'], 1)
```

[ ]:
```
import statsmodels.regression.linear_model as sm
import statsmodels.api as s
data1 = data[['num_imgs', 'num_videos', 'n_tokens_title']]
data1 = s.add_constant(data1)
data2 = data[['num_imgs', 'num_videos', 'n_tokens_title',␣
↪'data_channel_is_lifestyle',
      'data_channel_is_entertainment', 'data_channel_is_bus',
      'data_channel_is_socmed', 'data_channel_is_tech',
      'data_channel_is_world', 'weekday_is_monday', 'weekday_is_tuesday',
      'weekday_is_wednesday', 'weekday_is_thursday', 'weekday_is_friday',
      'weekday_is_saturday']]
data2 = s.add_constant(data2)
ols1 = sm.OLS(exog = data1, endog = data['shares']).fit()
#pred1= ols1.predict("rentrer le dataframe de l'utilisateur ici")
ols2 = sm.OLS(exog = data2, endog = data['shares']).fit()
#pred2= ols2.predict("rentrer le dataframe de l'utilisateur ici")

from sklearn.linear_model import LinearRegression
reg1 = LinearRegression().fit(data1, data['shares'])
reg2 = LinearRegression().fit(data2, data['shares'])
#pred1= reg1.predict("rentrer le dataframe de l'utilisateur ici")
#pred2= reg2.predict("rentrer le dataframe de l'utilisateur ici")
```

[ ]:
```
ols1.summary2().tables[0][3][0]
```

4 Regression models

Our problem is of the **regression** type. We shall compare the following models : **Forward selection with 10-fold cross-validation** on **transformed data** and **raw data Random Forest** on **transformed data** and **raw data Lasso regularization** on **transformed data** and **raw data**

4.1 Metrics

We'll use two metrics for our prediction error : **Mean Absolute Error** : $MAE = \frac{1}{n}\sum_{k=1}^{n}|y_i - \hat{y}_i|$ **Root Mean Square Error** : $RMSE = \sqrt{\frac{1}{n}\sum_{k=1}^{n}(y_i - \hat{y}_i)^2}$. These metrics are expressed in the same units as the target ***shares***, their values should give more insight than other metrics such as the **Mean Square Error** as well as the **Mean Percentage Error**. Moreover, the **MAE** gives the easier metric to interpret but is less sensitive to outliers as opposed to the **RMSE**. We don't know the requirements of the prediction problem, thus, for safety measures we'll only keep the **MAE** for insight but not for feature selection.

[ ]:
```
def MAE(pred, real) :
    return (pred - real).abs().mean()

def RMSE(pred, real) :
    return np.sqrt(((pred - real)**2).mean())
```

4.2 Forward selection with 10-fold cross-validation

4.2.1 Data splitting for 10-fold cross-validation

We'll use **cross-validation** by spliting the dataset in 10 sets of same length. By default, the **cross-validation score** is the R^2 of the model. Nevertheless, our aim is to return the best predictions, not the most linear model, especially since we're in no position to confirm such hypothesis. We chose the **CV score** to be the **arithmetic mean** $\frac{a_1+...+a_n}{n}$ where a_k is the **RMSE** of the model where set_k is the subset used of testing, while the rest is used for training. The **geometric mean** $\sqrt[n]{a_1 \cdot ... \cdot a_n}$ is better at representing the average of a sample whose elements are not independent, which is obviously our case thanks to the cross validation, but the **arithmetic mean** is the most sensitive to outliers, which will be the most valuable information in our minimization of the prediction error.

[ ]:
```
import random
set1 = list(); set2 = list(); set3 = list();
set4 = list(); set5 = list(); set6 = list();
set7 = list(); set8 = list(); set9 = list();
set10 = list()
sets = data.copy().index.tolist()

while len(sets)//10 > 0 :
    set1.append(sets.pop(random.randint(0, len(sets)-1)))
    set2.append(sets.pop(random.randint(0, len(sets)-1)))
    set3.append(sets.pop(random.randint(0, len(sets)-1)))
    set4.append(sets.pop(random.randint(0, len(sets)-1)))
    set5.append(sets.pop(random.randint(0, len(sets)-1)))
```

```
    set6.append(sets.pop(random.randint(0, len(sets)-1)))
    set7.append(sets.pop(random.randint(0, len(sets)-1)))
    set8.append(sets.pop(random.randint(0, len(sets)-1)))
    set9.append(sets.pop(random.randint(0, len(sets)-1)))
    set10.append(sets.pop(random.randint(0, len(sets)-1)))
set1 = set1 + sets
```

4.2.2 Linear Regression on transformed data

[ ]:
```
import statsmodels.regression.linear_model as sm
import statsmodels.api as s
import time
features = data.drop('shares', axis = 1)
current_features = []
linearselection = pd.DataFrame(columns = ['feature_selected', 'rsquared_adj',␣
↪'rmse_train_score', 'rmse_test_score', 'mae_train_score',
           'mae_test_score'])
while len(features.columns) > 0 :
    t = time.time()
    for feature in features.columns :
        optimum = (None, None)

        set1train = data.loc[set2+set3+set4+set5+set6+set7+set8+set9+set10,␣
↪current_features + [feature]]
        set1train = s.add_constant(set1train)
        set2train = data.loc[set1+set3+set4+set5+set6+set7+set8+set9+set10,␣
↪current_features + [feature]]
        set2train = s.add_constant(set2train)
        set3train = data.loc[set2+set1+set4+set5+set6+set7+set8+set9+set10,␣
↪current_features + [feature]]
        set3train = s.add_constant(set3train)
        set4train = data.loc[set2+set3+set1+set5+set6+set7+set8+set9+set10,␣
↪current_features + [feature]]
        set4train = s.add_constant(set4train)
        set5train = data.loc[set2+set3+set4+set1+set6+set7+set8+set9+set10,␣
↪current_features + [feature]]
        set5train = s.add_constant(set5train)
        set6train = data.loc[set2+set3+set4+set5+set1+set7+set8+set9+set10,␣
↪current_features + [feature]]
        set6train = s.add_constant(set6train)
        set7train = data.loc[set2+set3+set4+set5+set6+set1+set8+set9+set10,␣
↪current_features + [feature]]
        set7train = s.add_constant(set7train)
        set8train = data.loc[set2+set3+set4+set5+set6+set7+set1+set9+set10,␣
↪current_features + [feature]]
        set8train = s.add_constant(set8train)
```

```
        set9train = data.loc[set2+set3+set4+set5+set6+set7+set8+set1+set10,␣
↪current_features + [feature]]
        set9train = s.add_constant(set9train)
        set10train = data.loc[set2+set3+set4+set5+set6+set7+set8+set9+set1,␣
↪current_features + [feature]]
        set10train = s.add_constant(set10train)

        set1trainvalidation = data.
↪loc[set2+set3+set4+set5+set6+set7+set8+set9+set10, 'shares']
        set2trainvalidation = data.
↪loc[set1+set3+set4+set5+set6+set7+set8+set9+set10, 'shares']
        set3trainvalidation = data.
↪loc[set2+set1+set4+set5+set6+set7+set8+set9+set10, 'shares']
        set4trainvalidation = data.
↪loc[set2+set3+set1+set5+set6+set7+set8+set9+set10, 'shares']
        set5trainvalidation = data.
↪loc[set2+set3+set4+set1+set6+set7+set8+set9+set10, 'shares']
        set6trainvalidation = data.
↪loc[set2+set3+set4+set5+set1+set7+set8+set9+set10, 'shares']
        set7trainvalidation = data.
↪loc[set2+set3+set4+set5+set6+set1+set8+set9+set10, 'shares']
        set8trainvalidation = data.
↪loc[set2+set3+set4+set5+set6+set7+set1+set9+set10, 'shares']
        set9trainvalidation = data.
↪loc[set2+set3+set4+set5+set6+set7+set8+set1+set10, 'shares']
        set10trainvalidation = data.
↪loc[set2+set3+set4+set5+set6+set7+set8+set9+set1, 'shares']

        set1test = data.loc[set1, current_features + [feature]]
        set1test = s.add_constant(set1test)
        set2test = data.loc[set2, current_features + [feature]]
        set2test = s.add_constant(set2test)
        set3test = data.loc[set3, current_features + [feature]]
        set3test = s.add_constant(set3test)
        set4test = data.loc[set4, current_features + [feature]]
        set4test = s.add_constant(set4test)
        set5test = data.loc[set5, current_features + [feature]]
        set5test = s.add_constant(set5test)
        set6test = data.loc[set6, current_features + [feature]]
        set6test = s.add_constant(set6test)
        set7test = data.loc[set7, current_features + [feature]]
        set7test = s.add_constant(set7test)
        set8test = data.loc[set8, current_features + [feature]]
        set8test = s.add_constant(set8test)
        set9test = data.loc[set9, current_features + [feature]]
        set9test = s.add_constant(set9test)
```

```
        set10test = data.loc[set10, current_features + [feature]]
        set10test = s.add_constant(set10test)

        set1testvalidation = data.loc[set1, 'shares']
        set2testvalidation = data.loc[set2, 'shares']
        set3testvalidation = data.loc[set3, 'shares']
        set4testvalidation = data.loc[set4, 'shares']
        set5testvalidation = data.loc[set5, 'shares']
        set6testvalidation = data.loc[set6, 'shares']
        set7testvalidation = data.loc[set7, 'shares']
        set8testvalidation = data.loc[set8, 'shares']
        set9testvalidation = data.loc[set9, 'shares']
        set10testvalidation = data.loc[set10, 'shares']

        ols1 = sm.OLS(exog = set1train, endog = set1trainvalidation).fit()
        ols1train = ols1.predict(set1train)
        ols1test = ols1.predict(set1test)
        ols2 = sm.OLS(exog = set2train, endog = set2trainvalidation).fit()
        ols2train = ols2.predict(set2train)
        ols2test = ols2.predict(set2test)
        ols3 = sm.OLS(exog = set3train, endog = set3trainvalidation).fit()
        ols3train = ols3.predict(set3train)
        ols3test = ols3.predict(set3test)
        ols4 = sm.OLS(exog = set4train, endog = set4trainvalidation).fit()
        ols4train = ols4.predict(set4train)
        ols4test = ols4.predict(set4test)
        ols5 = sm.OLS(exog = set5train, endog = set5trainvalidation).fit()
        ols5train = ols5.predict(set5train)
        ols5test = ols5.predict(set5test)
        ols6 = sm.OLS(exog = set6train, endog = set6trainvalidation).fit()
        ols6train = ols6.predict(set6train)
        ols6test = ols6.predict(set6test)
        ols7 = sm.OLS(exog = set7train, endog = set7trainvalidation).fit()
        ols7train = ols7.predict(set7train)
        ols7test = ols7.predict(set7test)
        ols8 = sm.OLS(exog = set8train, endog = set8trainvalidation).fit()
        ols8train = ols8.predict(set8train)
        ols8test = ols8.predict(set8test)
        ols9 = sm.OLS(exog = set9train, endog = set9trainvalidation).fit()
        ols9train = ols9.predict(set9train)
        ols9test = ols9.predict(set9test)
        ols10 = sm.OLS(exog = set10train, endog = set10trainvalidation).fit()
        ols10train = ols10.predict(set10train)
        ols10test = ols10.predict(set10test)

        rmsetrain = 0
        rmsetest = 0
```

```
        maetrain = 0
        maetest = 0
        rsquared = 0

        rmsetrain += (RMSE(ols1train, set1trainvalidation))
        rmsetest += (RMSE(ols1test, set1testvalidation))
        maetrain += (MAE(ols1train, set1trainvalidation))
        maetest += (MAE(ols1test, set1testvalidation))
        rsquared += float(ols1.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols2train, set2trainvalidation))
        rmsetest += (RMSE(ols2test, set2testvalidation))
        maetrain += (MAE(ols2train, set2trainvalidation))
        maetest += (MAE(ols2test, set2testvalidation))
        rsquared += float(ols2.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols3train, set3trainvalidation))
        rmsetest += (RMSE(ols3test, set3testvalidation))
        maetrain += (MAE(ols3train, set3trainvalidation))
        maetest += (MAE(ols3test, set3testvalidation))
        rsquared += float(ols3.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols4train, set4trainvalidation))
        rmsetest += (RMSE(ols4test, set4testvalidation))
        maetrain += (MAE(ols4train, set4trainvalidation))
        maetest += (MAE(ols4test, set4testvalidation))
        rsquared += float(ols4.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols5train, set5trainvalidation))
        rmsetest += (RMSE(ols5test, set5testvalidation))
        maetrain += (MAE(ols5train, set5trainvalidation))
        maetest += (MAE(ols5test, set5testvalidation))
        rsquared += float(ols5.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols6train, set6trainvalidation))
        rmsetest += (RMSE(ols6test, set6testvalidation))
        maetrain += (MAE(ols6train, set6trainvalidation))
        maetest += (MAE(ols6test, set6testvalidation))
        rsquared += float(ols6.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols7train, set7trainvalidation))
        rmsetest += (RMSE(ols7test, set7testvalidation))
        maetrain += (MAE(ols7train, set7trainvalidation))
        maetest += (MAE(ols7test, set7testvalidation))
        rsquared += float(ols7.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols8train, set8trainvalidation))
```

```
        rmsetest += (RMSE(ols8test, set8testvalidation))
        maetrain += (MAE(ols8train, set8trainvalidation))
        maetest += (MAE(ols8test, set8testvalidation))
        rsquared += float(ols8.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols9train, set9trainvalidation))
        rmsetest += (RMSE(ols9test, set9testvalidation))
        maetrain += (MAE(ols9train, set9trainvalidation))
        maetest += (MAE(ols9test, set9testvalidation))
        rsquared += float(ols9.summary2().tables[0][3][0])

        rmsetrain += (RMSE(ols10train, set10trainvalidation))
        rmsetest += (RMSE(ols10test, set10testvalidation))
        maetrain += (MAE(ols10train, set10trainvalidation))
        maetest += (MAE(ols10test, set10testvalidation))
        rsquared += float(ols10.summary2().tables[0][3][0])

        rmsetrainscore = rmsetrain / 10
        rmsetestscore = rmsetest / 10
        maetrainscore = maetrain / 10
        maetestscore = maetest / 10
        rsquared = rsquared / 10

        if optimum[1] == None :
            optimum = (feature, rmsetestscore)
        elif optimum[1] > rmsetestscore :
            optimum = (feature, rmsetestscore)

    features = features.drop(optimum[0], 1)
    current_features.append(optimum[0])
    linearselection = linearselection.append({'feature_selected' : optimum[0],␣
↪'rsquared_adj' : rsquared, 'rmse_train_score' : rmsetrainscore,
                                              'rmse_test_score' :␣
↪rmsetestscore, 'mae_train_score' : maetrainscore,
                                              'mae_test_score' : maetestscore},␣
↪ignore_index = True)
    print(time.time() - t)
```

[ ]:
```
linearselection
```

[ ]:
```
plt.plot([i for i in range(len(linearselection))],␣
 ↪linearselection['rmse_train_score'], 'b')
plt.plot([i for i in range(len(linearselection))],␣
 ↪linearselection['rmse_test_score'], 'r')
plt.show()
plt.plot([i for i in range(len(linearselection))],␣
 ↪linearselection['mae_train_score'], 'b')
```

```
plt.plot([i for i in range(len(linearselection))],␣
 ↪linearselection['mae_test_score'], 'r')
plt.show()
plt.plot([i for i in range(len(linearselection))],␣
 ↪linearselection['rsquared_adj'], 'g')
plt.show()
```

4.3 Random Forest

[ ]:
```
import time
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
datatrain, datatest, sharestrain, sharestest = train_test_split(data.
 ↪drop('shares', 1), data['shares'], train_size = 0.67)

forestselection = pd.DataFrame(columns = ['trees', 'rmse_auto_train',␣
 ↪'rmse_auto_test', 'autoscore', 'rmse_sqrt_train',
            'rmse_sqrt_test', 'sqrtscore', 'rmse_log_train', 'rmse_log_test',␣
 ↪'logscore'])

for t in range(1, len(data.columns)) :
    timer = time.time()

    treeauto = RandomForestRegressor(n_estimators = t, criterion = 'mse',␣
 ↪max_features = 'auto')
    treeauto.fit(datatrain, sharestrain)
    rmseautotrain = RMSE(treeauto.predict(datatrain), sharestrain)
    rmseautotest = RMSE(treeauto.predict(datatest), sharestest)
    autoscore = treeauto.score(datatest, sharestest)

    treesqrt = RandomForestRegressor(n_estimators = t, criterion = 'mse',␣
 ↪max_features = 'sqrt')
    treesqrt.fit(datatrain, sharestrain)
    rmsesqrttrain = RMSE(treesqrt.predict(datatrain), sharestrain)
    rmsesqrttest = RMSE(treesqrt.predict(datatest), sharestest)
    sqrtscore = treesqrt.score(datatest, sharestest)

    treelog = RandomForestRegressor(n_estimators = t, criterion = 'mse',␣
 ↪max_features = 'log2')
    treelog.fit(datatrain, sharestrain)
    rmselogtrain = RMSE(treelog.predict(datatrain), sharestrain)
    rmselogtest = RMSE(treelog.predict(datatest), sharestest)
    logscore = treelog.score(datatest, sharestest)

    forestselection = forestselection.append({'trees' : t, 'rmse_auto_train' :␣
 ↪rmseautotrain, 'rmse_auto_test' : rmseautotest, 'autoscore' : autoscore,
```

```
                                                'rmse_sqrt_train' : rmsesqrttrain,␣
↪'rmse_sqrt_test' : rmsesqrttest, 'sqrtscore' : sqrtscore,
                                                'rmse_log_train' : rmselogtrain,␣
↪'rmse_log_test' : rmselogtest, 'logscore' : logscore}, ignore_index = True)

    print(str(t), str(time.time() - timer))

forestselection
```

[ ]:
```
plt.plot([i for i in range(len(forestselection))],␣
↪forestselection['rmse_auto_test'], 'b')
plt.plot([i for i in range(len(forestselection))],␣
↪forestselection['rmse_sqrt_test'], 'g')
plt.plot([i for i in range(len(forestselection))],␣
↪forestselection['rmse_log_test'], 'r')
plt.show()
plt.plot([i for i in range(len(forestselection))],␣
↪forestselection['rmse_sqrt_train'], 'g')
plt.plot([i for i in range(len(forestselection))],␣
↪forestselection['rmse_log_train'], 'r')
plt.show()
```

[ ]:
```
names_and_feature_imp = pd.DataFrame({'names': data.drop('shares', 1).columns,
                                      'importance': treelog.
↪feature_importances_})
names_and_feature_imp.sort_values(["importance"], ascending=True, inplace=True)
names_and_feature_imp.plot(kind='barh', title="Features' importances of the log␣
↪tree",figsize=(20,10))
```

4.4 LinearRegression

[ ]:
```
from sklearn.ensemble import RandomForestRegressor
regr = LinearRegression()

regr.fit(datatrain, sharestrain)

s_pred = regr.predict(datatest)


print('Coefficients: \n', regr.coef_)

print("Mean squared error:", mean_squared_error(sharestest, s_pred))

print('R2 :', r2_score(sharestest, s_pred))
```

4.5 Decision Tree

[ ]:
```
regressor = DecisionTreeRegressor()
regressor.fit(datatrain,sharestrain)

y_pred = regressor.predict(datatest)

print("RMSE: " + str(round(sqrt(mean_squared_error(sharestest,y_pred)),2)))
print("R_squared: " + str(round(r2_score(sharestest,y_pred),2)))
```

5 Flask regression

[ ]:
```
dataapi =  data[['n_tokens_title', 'num_imgs', 'num_videos']]

yy = data['shares']

dataapi =  data[['n_tokens_title', 'num_imgs', 'num_videos']]

yy = data['shares']

from sklearn.model_selection import train_test_split

Xtrain, Xtest, Ytrain, Ytest = train_test_split(dataapi, yy, train_size=0.8)

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
regr = LinearRegression()

model_saved1 = regr.fit(Xtrain, Ytrain)

s_pred = regr.predict(Xtest)


print('Coefficients: \n', regr.coef_)

print("Mean squared error:", mean_squared_error(Ytest, s_pred))

print('Variance score:', r2_score(Ytest, s_pred))

!pip install joblib
import joblib
joblib.dump(model_saved1, "./model_saved1")
```