Python for Data Analysis Project

NICOLAS Kevin & YAYA-OYE Ikhlass – DIA1

Introduction

We had to work with **Spam Email Database**, where we had to find a model to be able to **predict** if an email is a **spam** or not.

We should be aware that the data we used to build our model had been collected in **1999** and nowadays, a spam email is much more **discreet** and **harder** to detect.

The **collection** of emails, spam or not, had been collected from the donor of the data, **George Forman**, and from the **team** who worked on it.

Import Dataset

As the **spambase.data** does not contain columns' name, we built a **function** which stock all features' name (in **spambase.names**) in a list.

Then we use this list when **reading** the data file.

Import Dataset

```
]: # A little function to recuperate the columns names in spambase.names document
with open('spambase.names','r') as f:
    line = f.readline()
    while 'word_freq_make' not in line : line = f.readline()
    features_names = []
    while(line):
        features_names.append(line.split(':')[0])
        line = f.readline()
    f.close()

features_names.append('spam')
```

```
: df = pd.read_csv('spambase.data', names=features_names)
```

Dataset Analysis: the variables

The dataset has **58 variables** that are :

- **48** continuous real attributes that count the **percentage** of specifics **words** that **are** in the email.
- **6** continuous real attributes that count the **percentage** of specifics **characters** that **are** in the email.
- **3** continuous integer attributes that respectively count the **average length** of **uninterrupted** sequences of **capital** letters, the **length** of **longest uninterrupted** sequence of **capital** letters and the **total** number of **capital** letters in the email.
- Finally, one nominal class attribute of type spam that takes the value **1** for a **spam** and **0** for a **regular** email.

Data Analysis: the instances

There is **no missing** value in the dataset.

We have **4601 instances** collected and the repartition is **1813 spams (39,4%)** and **2788 non-spam emails (60,6%)**.

We can see for example that the capital letters' variables should be **indicators** to tell if an email is a spam or not :

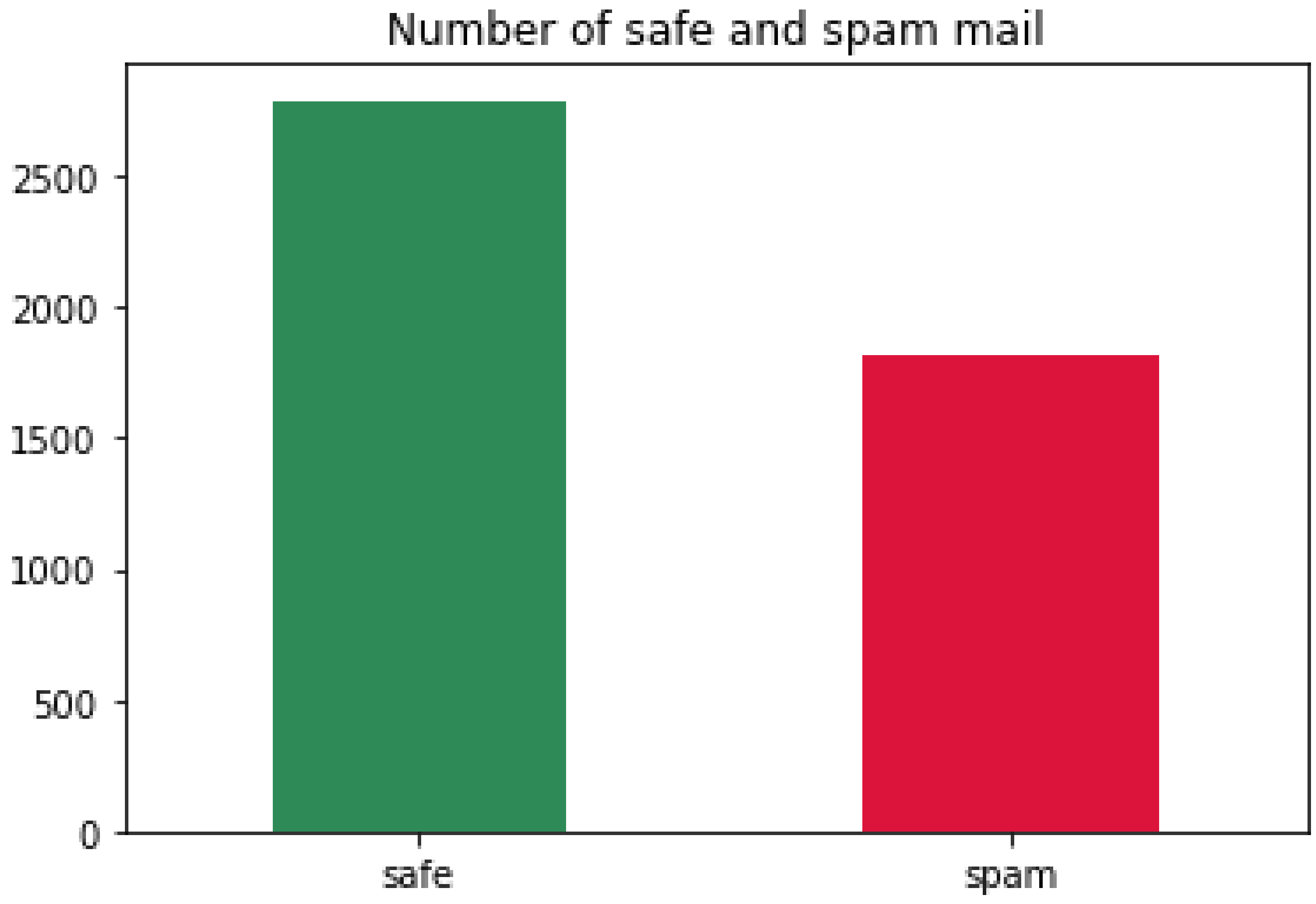
```
df[df.spam == 1].describe().T.drop('count', axis=1)
```

capital_run_length_average	9.519165	49.846186	1.0	2.324	3.621	5.708	1102.500
capital_run_length_longest	104.393271	299.284969	1.0	15.000	38.000	84.000	9989.000
capital_run_length_total	470.619415	825.081179	2.0	93.000	194.000	530.000	15841.000
spam	1.000000	0.000000	1.0	1.000	1.000	1.000	1.000

```
df[df.spam == 0].describe().T.drop('count', axis=1)
```

capital_run_length_average	2.377301	5.113685	1.0	1.384	1.8570	2.5550	251.000
capital_run_length_longest	18.214491	39.084792	1.0	4.000	10.0000	18.0000	1488.000
capital_run_length_total	161.470947	355.738403	1.0	18.750	54.0000	141.0000	5902.000
spam	0.000000	0.000000	0.0	0.000	0.0000	0.0000	0.000

Some plots and visualizations

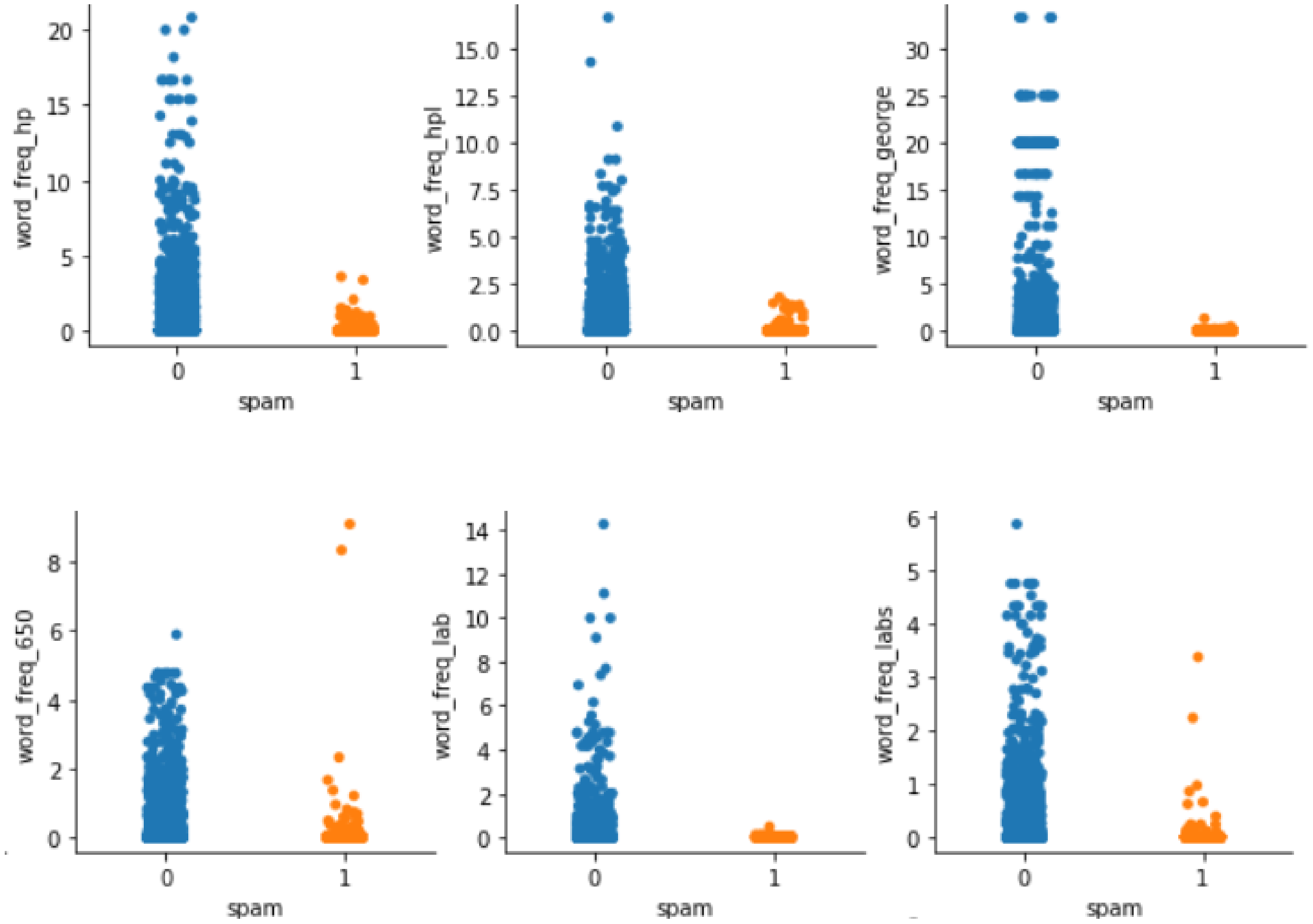


Scatterplot

We made a **scatterplot** to have a first visualizations of the **individuals' distributio**n according to their **type** (safe or spam). For this we tried the **seaborn**'s pair plot function, but it fit better for linear relationship (so for regression problems).

Here it's a **classification** problem with a categorical target (0 or 1) so we use **strip** plot, which is a scatterplot with a nicer look.

Scatterplot

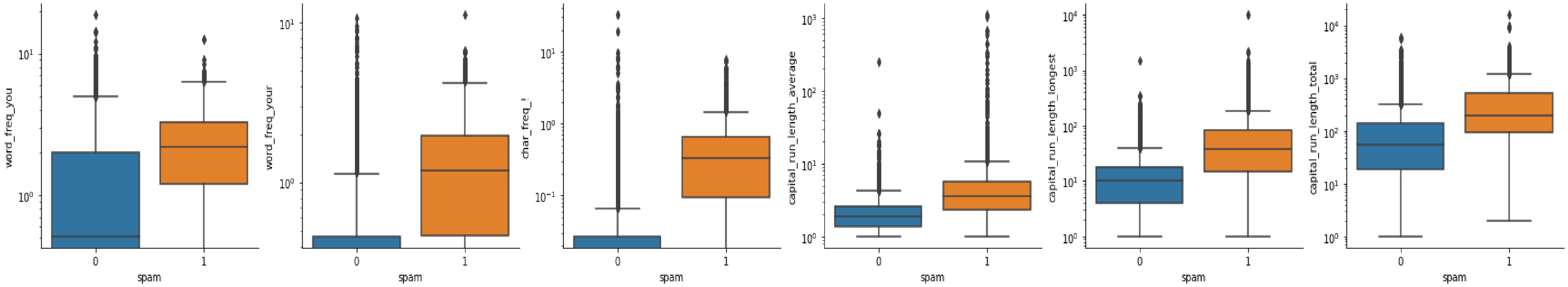


For example, we can imply that for the **6** features represented on those plots, their **frequency** is **higher** in **non-spam mail** than in spam.

Moreover, for the word ***"650"*** if frequency is **higher** than **8%**, according to our dataset a model only based on this feature should predict the input as a **spam**.

Boxplot

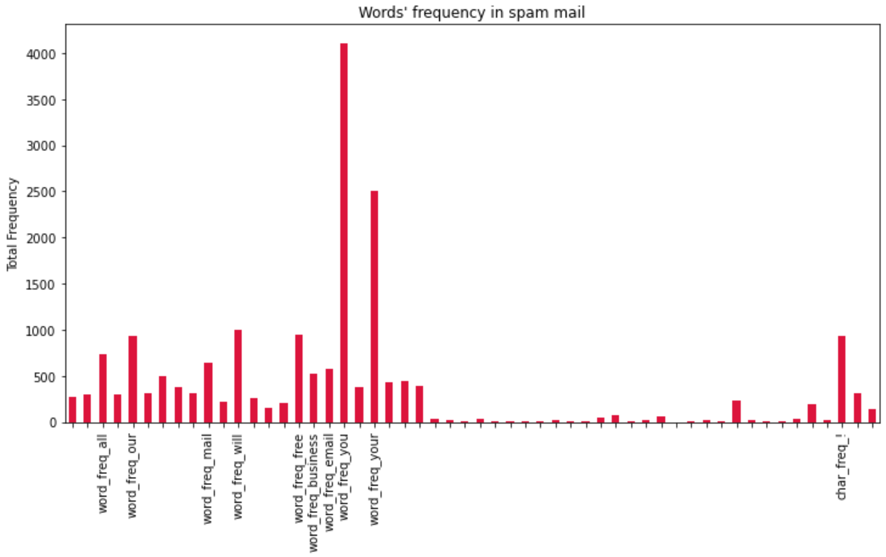
For a **better visualization** of our data (because features are frequencies) we use a **logarithmic** scale for our boxplot. Here are the **6** nicer and more interesting boxplots which show **features** for which **spam** mail generally have **higher** frequencies.

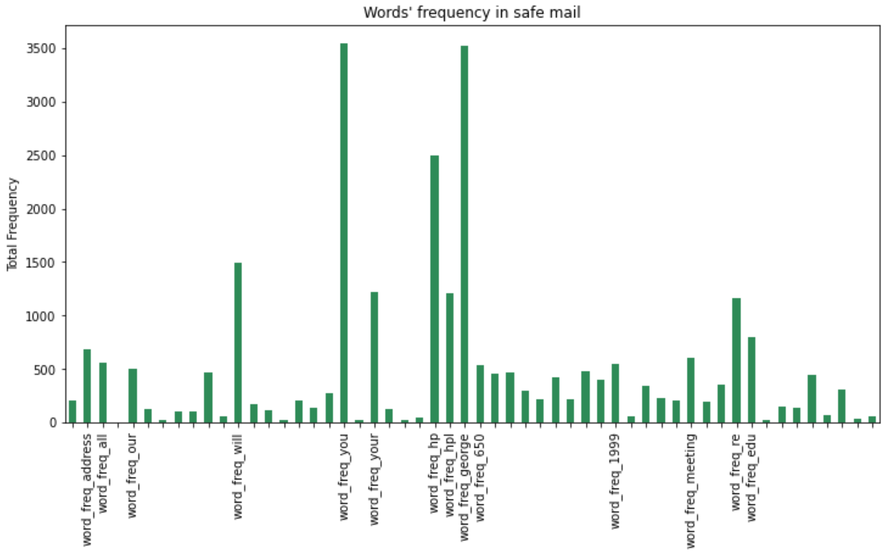


Barplot

Then we plot the **sum of frequency** for each features for a **safe mail** and for **spam** and only plot features names for those which have a total frequency **higher than 500**.

The word ***"you"*** is the more present in both kind of mail but more in **spam** even if they are fewer in our dataset than safe mail. While ***"George"*** (the donor of the data) and ***"hp"*** (meaning ridiculous) are also very present in **safe mail**.

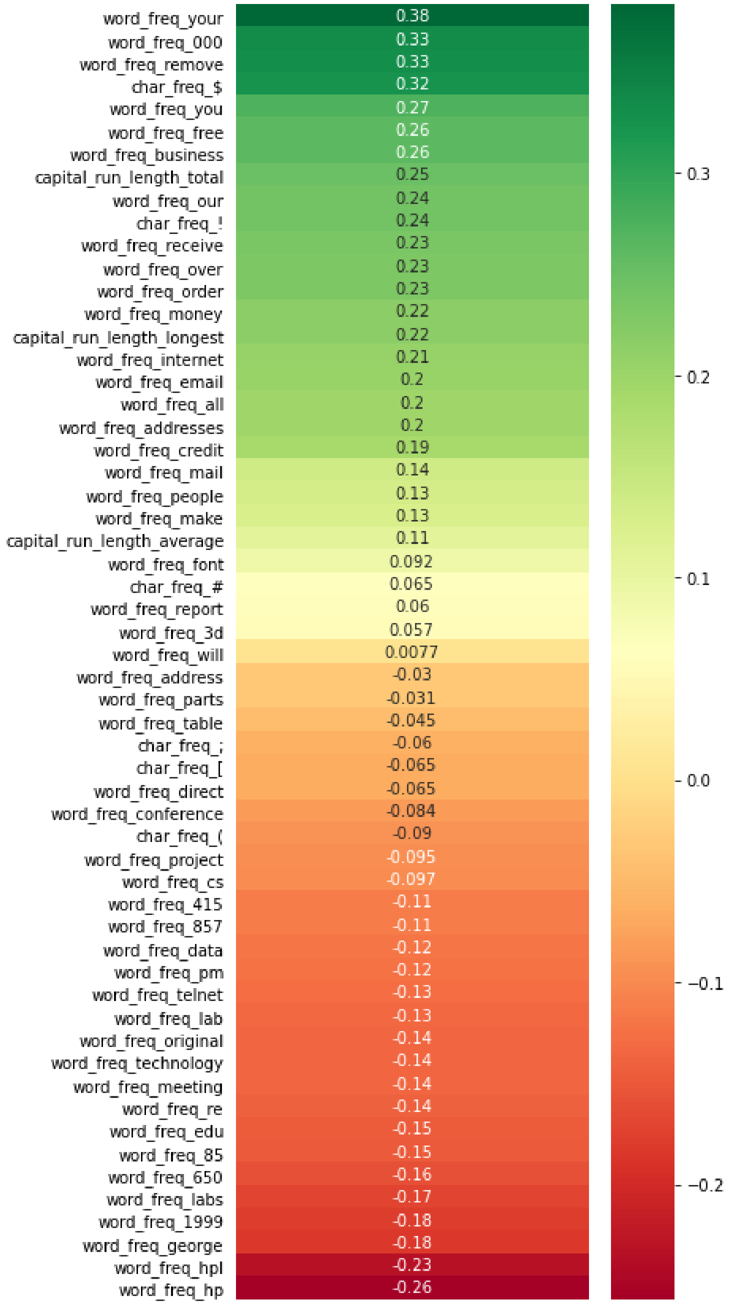




Correlation

Finally, because the correlation matrix plot is not well **interpretable**, we only plot the **correlation** of features to the **target**. A positive correlation means a **high frequency of the feature** implies the mail is a **spam** and vice versa for a negative correlation.

We can see that according to our previous plots, *hp*, *hpl* and *George* presence tends to predict a **non-spam** while *your* or *remove* are more present in **spam.**



Data Preprocessing: selected features

According to the correlation between some features and the target, we define a **threshold** and **select features** which have a **higher** correlation with the target than the **positive** threshold or a **lower** correlation than the **negative** threshold.

```
]: threshold = 0.11
   selected_features = corr_spam[(corr_spam < -threshold) | (corr_spam > threshold)].index
   len(selected_features)

41
```

This gave us **41** features. (originally, we had 57 features)

Data Preprocessing: Splitting

Then we split our dataset in **half** : a **training set** with **75%** of the dataset and a **test set** with the last **25%**.

And we make the same operation with the **selected** features.

```
Entrée [24]: X_train, X_test, Y_train, Y_test = train_test_split(X,Y, random_state=10)
             X_train_s, X_test_s, Y_train_s, Y_test_s = train_test_split(X[selected_features],Y, random_state=10)
             print('Training set with all features shape: {}'.format(X_train.shape))
             print('Testing set with all features shape: {}\n'.format(X_test.shape))
             print('Training set with selected features shape: {}'.format(X_train_s.shape))
             print('Testing set with selected features shape: {}\n'.format(X_test_s.shape))
             print('Training set with target value shape: {}'.format(Y_train.shape))
             print('Testing set with target value shape: {}'.format(Y_test.shape))
```

```
Training set with all features shape: (3450, 57)
Testing set with all features shape: (1151, 57)

Training set with selected features shape: (3450, 41)
Testing set with selected features shape: (1151, 41)

Training set with target value shape: (3450,)
Testing set with target value shape: (1151,)
```

Data Preprocessing: Scaling

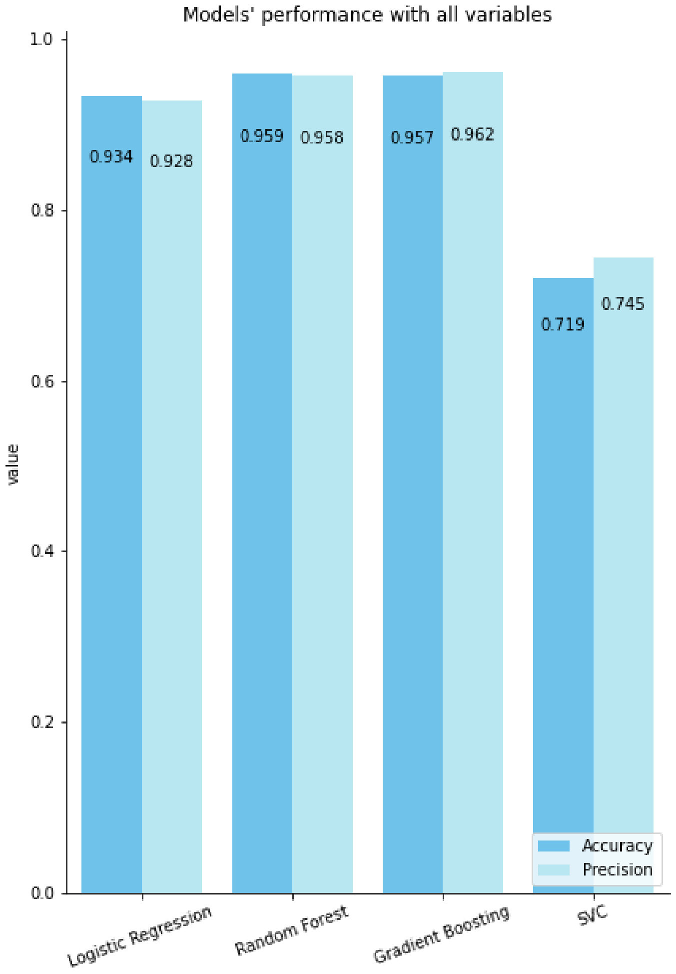
In our first attempts we **scaled** our data. However, at the end of the project we saw that it will **not** be beneficial for us because we would like to apply our model on **real world data**.

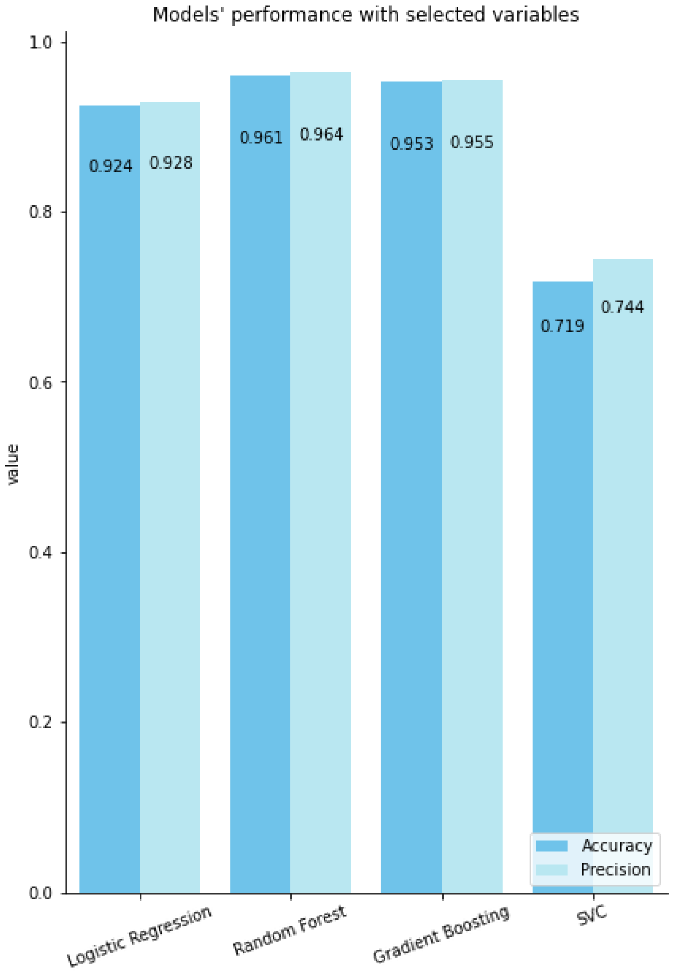
So, we decided not to scale it and we constated it was not a big deal for us except for the **Logistic Regression** which stopped when the number of iterations reached the **limit**.

Evaluation metric choice

- According to our study, we are making a model to define a safe mail or a spam. Obviously, we are going to look at the **accuracy** of our model. However, the biggest risk for this problem is that a model defines a safe mail as a spam: If an **important mail** is sent in **spam directory**, the user may not see it.
- Then the metric on which we focused is the **precision** which is equal to TruePositive/(TruePositive + FalsePositive). For us, the precision is explained by the **number of true spam among all instances predicted as spam**.

First models





We fitted **4 models** on the training set with all features and the training set with the 41 selected features : Logistic Regression, Random Forest, Gradient Boosting and Support Vector Classifiers.

The two best models are ensemble classifiers **Random Forest** and **Gradient**. Use **selected variables** also seems to be an interesting idea to use a lighter model with **good performances**.

Tuning of the Random Forest model

After several **Grid Search**, we decided to work with the **Random Forest** which is **faster** and performs slightly **better** than Gradient Boosting.

We defined some basic parameters as the use of **bootstrap** and **out-of-bag** samples, the **number of features** considered for the **best split**.

The **hyperparameters** on which we worked were the split quality **criterion**, the **number of trees** used and the **maximum depth** for each tree.

Then, we use **5 cross validation folds** and tried to get a model which optimizes **precision**.

Entrée [47]:
```
param_grid = {
    'bootstrap': [True],
    'criterion': ['gini','entropy'],
    'oob_score': [True],
    'max_depth': [20, 35, 50, 70],
    'max_features': ['sqrt'],
    'n_estimators': [75, 100, 200, 300, 500]
}
```

Entrée [48]:
```
grid_search = GridSearchCV(estimator = random_model, scoring = ['accuracy','precision'], param_grid = param_grid,
                          cv = 5, n_jobs = -1, verbose = 2, refit='precision')
```

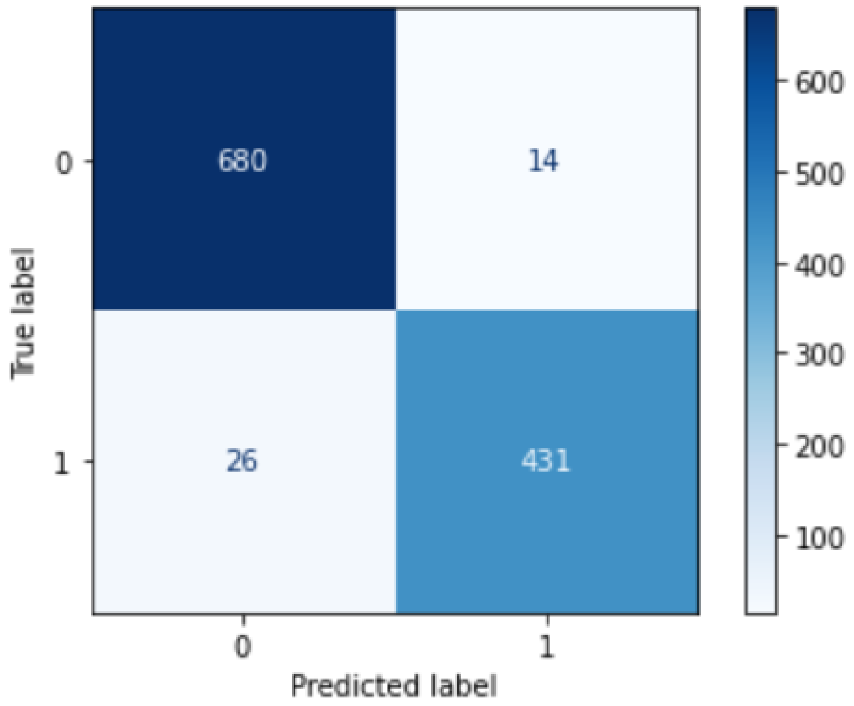
Entrée [49]:
```
grid_search.fit(X_train, Y_train)
```

```
Fitting 5 folds for each of 40 candidates, totalling 200 fits
```

Best Random Forest

Here is the **best Random Forest** model using **57 features** with the hyperparameters selected by GridSearch function.

```
Entrée [52]: params = grid_search.best_params_
             params

 Out[52]: {'bootstrap': True,
           'criterion': 'entropy',
           'max_depth': 35,
           'max_features': 'sqrt',
           'n_estimators': 500,
           'oob_score': True}

Entrée [56]: best_rf.score(X_test, Y_test)

 Out[56]: 0.9652476107732406

Entrée [55]: precision_score(Y_test,best_rf.predict(X_test))

 Out[55]: 0.9685393258426966
```

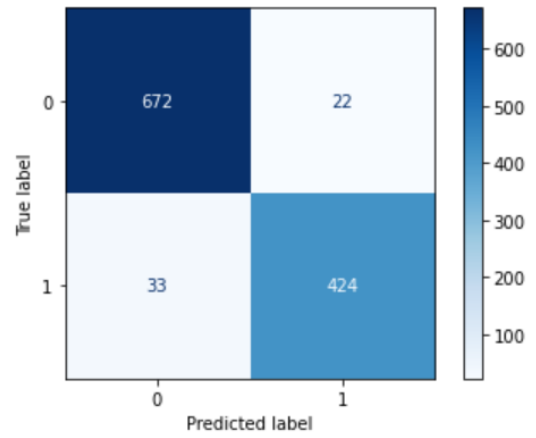


Tuning: SelectFromModel function

As using less features can lighten and improve models, we used a scikit-learn function which selected relevant features of a model based on their importance.

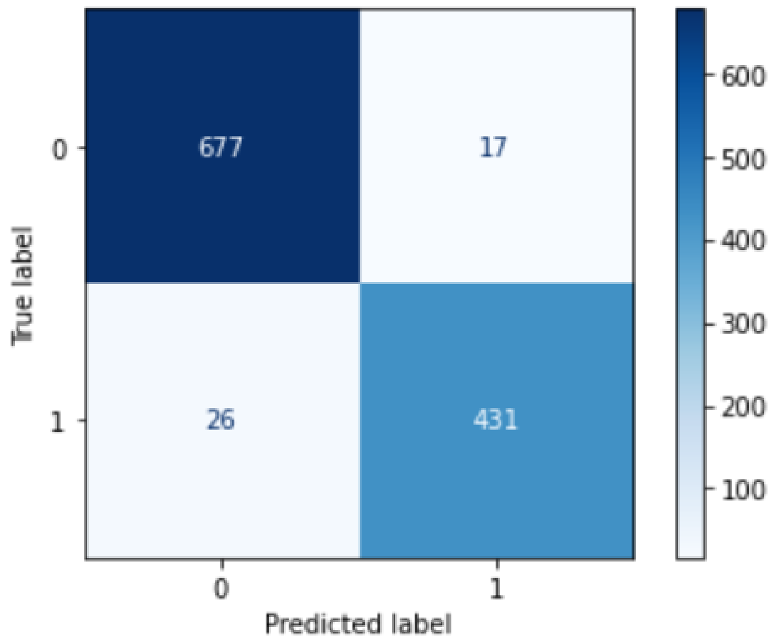
Here the function chose **16 features** and **lightened** a lot the model, but we **lost** almost **2%** on the precision.

```
Entrée [60]: from sklearn.feature_selection import SelectFromModel

Entrée [61]: selector = SelectFromModel(estimator=best_rf).fit(X_train, Y_train)

Entrée [62]: important_features = np.array(df.columns[:57][selector.get_support()])

Entrée [63]: final_X_train = X_train[important_features]
             final_X_test = X_test[important_features]

Entrée [81]: print(important_features)
             print("We are now using {} features.".format(len(important_features)))

['word_freq_our' 'word_freq_remove' 'word_freq_free' 'word_freq_you'
 'word_freq_your' 'word_freq_000' 'word_freq_money' 'word_freq_hp'
 'word_freq_hpl' 'word_freq_george' 'word_freq_edu' 'char_freq_!'
 'char_freq_$' 'capital_run_length_average' 'capital_run_length_longest'
 'capital_run_length_total']
We are now using 16 features.
```




```
Entrée [66]: precision_score(Y_test,best_rf.predict(final_X_test))
Out[66]: 0.9506726457399103
```

Tuning: Selected features with threshold




```
Entrée [69]: precision_score(Y_test,best_rf.predict(X_test_s))
```

```
Out[69]: 0.9620535714285714
```

We also fit our model with the **41 features** selected before with the defined **threshold**.

This model is a little **lighten** and only loses **0.6%** compared to the model fitted with all features.

Final model chosen

After some thought we chosen the model using **all features** because even if it uses more features it is neither overweight nor too complex.

We preferred to **limit the risk** to predict a **safe mail as a spam** and the eventual consequences resulting from it.

Saving of the model

Saving of the **model** to implement it into the **Flask API**.

Save the model

Entrée [70]:
```
from joblib import dump, load
best_rf.fit(X_train,Y_train)

dump(best_rf, 'model_saved.joblib')
```

Out[70]:
```
['model_saved.joblib']
```

Convert an email example to an individual

We created a function that will use an **input text** entered by the **user** to convert it as an **individual** to apply it on our **saved model** that will return if the text is a **spam** or not.

To do so, we need 2 libraries that are ***re*** and the ***Counter*** function from the *collections* library.

```
import re
from collections import Counter
```

The function

We put all the **words** used as **variables** in the dataset in a **list**.

We **split** the text put as a **parameter** in the function and we **count** the number of **word or character** that we have in the text.

For the **48 firsts** variables, we simply do the **frequency** of **appearance** of the **word** variable in the text.

For the **next 6** variables, we do the **frequency** of **appearance** of the **character** variable in the text.

After that, we **find** all the **capital letters** in the text, we **count** the number of them and with those 2 variables, we have the **3 lasts** variables about capital letters.

At the end, we **return** all the **information** about our text as a **line** that our **model** will read to **predict** if the text is a spam or not.

The function will be used in our **API Flask**.

The function

```
def text_to_ind(text):
    all_variables = ['make', 'address', 'all', '3d', 'our', 'over', 'remove', 'internet', 'order', 'mail', 'receive', 'will', 'p
    dic = {}
    text_split = re.split(r'\W+', text)
    text_split_count = Counter(text_split)
    for i in range(48):
        if all_variables[i] in text_split_count.keys() : dic[all_variables[i]] = 100*text_split_count[all_variables[i]]/len(text_
        else : dic[all_variables[i]] = float(0)

    for i in range(48,54):
        dic[all_variables[i]] = 100*text.count(all_variables[i])/(len(text)- text.count(' '))

    all_uppercase_sequence = re.findall(r"[A-Z]+", text)
    sum_uppercase = 0

    for sequence in all_uppercase_sequence:
        sum_uppercase += len(sequence)

    dic['capital_run_length_average'] = sum_uppercase/len(all_uppercase_sequence)
    dic['capital_run_length_longest'] = len(max(all_uppercase_sequence, key=len))
    dic['capital_run_length_total'] = sum_uppercase

    return np.array(list(dic.values())).reshape(1,-1)
```

Flask Application: libraries and classes

To transform our **model** into an **API**, we chose to use **Flask** that we saw in class.

First, we need to import Flask **classes** that we will use in our API

```
from flask import Flask, render_template, request
```

The ***render_template*** class allows us to use our **own html templates** for better display. The ***request*** class is used because we use a **form style** to send information to our **model**.

We also need to import the ***load*** class from the ***joblib*** library to use our model

```
from joblib import load
```

Flask Application: routes

We defined **6 routes** in our Flask application.

The first one is the **root** of the application ***('/')*** where we have **2** methods : the ***home()*** method that just returns the ***home.html*** template and the ***text_to_ind(text)*** method that we explained earlier.

The second one is our **Prediction page** route ***('/prediction', methods = ['POST'])***. The method of this route is a **POST method** because we want to **send** the individual to our **model**. In this route, we have the ***page_pred()*** method that **collects** the **text** entered by the user, **loads** the **model**, **converts** the text as an **individual** and **returns** the ***results.html*** template that **keep** the **data** of the individual.

Flask Application: routes

The **4** other routes are very **similar** to the root's route, but they have a **prefilled text** to show **examples** for our model. Each of them return a specific **html template**. In those routes, we displayed respectively a well detected **spam**, a well detected **safe mail**, a **spam** interpreted as a **safe mail** and a **safe mail** interpreted as a **spam**.

```
@app.route('/TrueSpam')
def true_spam():
    return render_template('true_spam.html', title = "True Spam")

@app.route('/FalseSpam')
def false_spam():
    return render_template('false_spam.html', title = "False Spam")

@app.route('/TrueSafeMail')
def true_mail():
    return render_template('true_mail.html', title = "True Safe Mail")

@app.route('/FalseSafeMail')
def false_mail():
    return render_template('false_mail.html', title = "False Safe Mail")
```

Flask Application: templates

2 templates will be used in **all** the other templates of our API.

```
{% include 'icon_web.html' %}

{% include 'navigation.html' %}
```

The first one is the ***icon_web.html*** template where we put the **icon** of our application.

The second one is the **navigation bar** of the application. The root's route and the 4 examples routes are available **directly** via the navigation bar. We put some **CSS style** to the template to have a beautiful tool.

We also add a **copyright footer** to all our templates.

Flask Application: home template

The **home** template has a **CSS style** in the header to have a beautiful interface.

There is a **text area** where the user enters his **email example**, and this will be used and transformed as an **individual** for the **model**.

To send it for **prediction**, the user just has to press the **Predict button** below the text area.

Flask Application: results template

The **results** template also has a **CSS style** in the header. The general **display** of the template depends on the **data** sent by the **home** route.

If the email is detected as a **spam (data sent = 1)**, the screen **blink** from **red to white** to insist on the **danger** of the spam. We also display an image to fill the screen and makes it more pleasant.

If the email is detected as a **safe mail (data sent = 0)**, the screen **blink slowly** from **green to white** to **reassure** the user about the nature of his mail. We also display an image to fill the screen and makes it more pleasant.

At the end of the template, we put a **go back home button** to access the application's root.

Flask Application: examples template

The 4 templates lefts, **true_spam**, **false_spam**, **true_mail** and **false_mail** are very similar to the **home** template.

The only difference is that for **each template**, we **prefilled** the text area with examples that we found to show the **limits** of our **model**.

When we click the **Predict** button on those routes, we are also redirected to the **results** route.

Thank you !

All our code and application are available on our GitHub :

www.github.com/ikhlo

www.github.com/kevinnclas

For any question, contact us :

ikhlass.yaya-oye@edu.devinci.fr

kevin.nicolas@edu.devinci.fr