Fouille de Données et Apprentissage

Rapport du devoir

Auteur :
David Condaminet
(21306226)

Responsable :
M. Alexis Lechervy

Mars 2017

Table des matières

1 Introduction

1.1 Objectif

L'objectif de ce projet est de réaliser un bon détecteur de spams pour SMS et pour emails. Pour ce faire, il va falloir comparer des méthodes de classification supervisée sur les données des bases de SMS et d'emails.

Pour exploiter les méthodes de classification supervisée, on utilise la librairie *scikit-learn* sur Python 3.

1.2 Résumé du tutoriel sur la classification de textes

La tutoriel disponible à la page `http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html` présente des outils de *scikit-learn* dont j'ai eu besoin pour réaliser ce projet.

1.2.1 Extraire des caractéristiques de fichiers textes

Avant de faire de l'apprentissage automatique sur des documents, il faut calculer des descripteurs à partir de leur texte, de manière à obtenir une représentation vectorielle numérique de ces derniers.

La représentation la plus intuitive est : *bags of words*. Elle consiste à affecter dans un dictionnaire un entier naturel comme identifiant à chaque occurence de mot de tous les documents. Puis, pour chaque document i, le nombre d'occurences de chaque mot w est compté et stocké dans un dictionnaire X en $X[i, j]$ où j est l'indice du mot w.

Pour transformer les textes en vecteurs dans ce projet, j'ai utilisé deux outils de *scikit-learn* :
- ***CountVectorizer*** qui permet de calculer les vecteurs de chaque texte en construisant un dictionnaire du nombre d'occurences de chaque mot de ce texte.
- ***TfidfTransformer*** qui permet de transformer ces vecteurs en calculant la fréquence d'apparition dans un texte de chaque mot à partir de son nombre d'occurrences, en divisant ce dernier par le nombre total de mots que contient le document.

1.2.2 Entraîner le classifieur

Maintenant que les textes sont représentés sous forme de vecteurs, on peut effectuer de l'apprentissage automatique en utilisant un classifieur. Pour pouvoir prédire la classe de données test, il faut d'abord entraîner ce dernier sur une base d'apprentissage train.
Dans le tutoriel, le classifieur *Naïf Bayésien* ***MultinomialNB*** est utilisé comme exemple. Dans ce projet, j'utilise les principaux classifieurs, dont celui-ci.

Pour faire apprendre un classifieur, il faut appeler sa méthode *fit()* en lui donnant en arguments :
- la base de données d'apprentissage X
- la liste des classes associées aux données de cette base y

Ensuite, on peut lui faire prédire les classes d'un certain ensemble de données test grâce à sa méthode *predict()* en lui donnant simplement la base de test.

1.2.3 Construire un *Pipeline*

Pour combiner la vectorisation via ***CountVectorizer***, la transformation via ***TfidfTransformer*** et la classification, *scikit-learn* propose un outil très pratique et très simple d'utilisation nommé ***Pipeline***. Il se comporte comme un classifieur, c'est-à-dire qu'on peut l'entraîner en appelant sa méthode *fit()*.

Pour créer un ***Pipeline*** de classification de textes, il suffit de lui donner en paramètre de son constructeur une liste contenant le vectoriseur, le transformeur, et le classifieur. J'utilise dans ce projet cet outil pour chacun de mes classifieurs.

1.2.4 Évaluer les performances d'un classifieur sur une base de test

Il est important d'évaluer les performances d'un classifieur. Pour ce faire, il faut analyser la prédiction de celui-ci. Il existe différentes fonctions de *scoring* comme par exemple le *f1-score*, ou encore la matrice de confusion.

1.2.5 Trouver le meilleur jeu de paramètres d'un classifieur en utilisant *GridSearchCV*

La librairie *scikit-learn* fournit un excellent outil nommé ***GridSearchCV*** permettant de trouver le meilleur jeu de paramètres pour une grille de jeux de paramètres, un classifieur et une mesure de performance (fonction de *scoring*) donnés.
A partir de ce dictionnaire, il va tester toutes les combinaisons de paramètres possibles en évaluant la performance du classifieur paramétré comme il se doit, par rapport à la mesure.

Dans la grille de jeux de paramètres, le nom des paramètres du classifieur doit impérativement être précédé du préfixe «*clf__*» où *clf* est le nom donné au classifieur dans le ***Pipeline*** lors de sa création.
Lors de l'instanciation du ***GridSearchCV***, on peut affecter la valeur *-1* à son paramètre *n_jobs* pour qu'il teste en parallèle différentes combinaisons en utilisant tous les coeurs de la machine.
On peut aussi lui affecter un nombre de validations croisées différentes (*fold*) à effectuer pour chaque combinaison via le paramètre *cv* qui est par défaut à 3.
Cet outil accepte comme classifieur un ***Pipeline***, et il se comporte également comme un classifieur. Il possède donc les méthodes *fit()* et *predict()*.
Après l'avoir *fit*, on peut accéder à différentes valeurs très intéressantes comme le meilleur score en moyenne (sur toutes les validations croisées) via son attribut *best_score_* et au jeu de paramètres associé via *best_params_*. On peut également récupérer un résumé détaillé de sa recherche via son attribut *cv_results_*.

J'utilise ***GridSearchCV*** tout au long du projet pour comparer mes différents classifieurs sur différentes mesures de performance.

Dans ce projet, on essaie de concevoir un détecteur de spams. La classe recherchée est donc «*spam*». La classe 1 est la classe positive où se trouvent les spams recherchés, tandis que la classe 0 est la classe négative où se trouvent les hams.

On dispose de deux bases de données :
— une pour les SMS
— une pour les emails

2 Base de SMS

2.1 Réalisation

Sous forme d'un fichier texte *SMSSpamCollection.txt*, les SMS sont représentés ligne par ligne, par le type de message (spam ou ham) et le message lui-même, séparés tous deux par une tabulation. Dans cette base, il y a 5574 SMS et parmi ces SMS, 4827 sont des hams et 747 sont des spams. Les deux classes ne sont donc pas équilibrées.

Tout d'abord, j'extrais les messages et les labels de ce fichier. J'obtiens alors deux ensembles : un contenant les messages, l'autre contenant les classes 0 ou 1 remplaçant respectivement ham ou spam. Après avoir identifié les outils que j'allais utiliser en lisant le tutoriel, je me suis rendu compte que pour séparer la base en un sous-ensemble d'entraînement et un sous-ensemble de tests, il fallait utiliser un objet ***ShuffleSplit***. Lors de l'instanciation de cet objet, je lui définis un nombre de cross-validations *n_splits*, une proportion pour l'ensemble de tests *test_size* et un grain pour le générateur pseudo-aléatoire utilisé *random_state*. J'ai respectivement choisi comme valeurs **pour l'intégralité du projet** 5 pour *n_splits* et 42 pour *random_state*. Comme demandé dans l'énoncé, la taille de test *test_size* est 0.7 (70%) pour la base de SMS. Cet objet ***ShuffleSplit*** se passe directement dans le paramètre *cv* du ***GridSearchCV*** vu précédemment. Ainsi, pour chacune des 5 validations croisées, l'ensemble de SMS sera divisé aléatoirement en 30% pour l'entraînement et 70% pour le test.

J'ai choisi d'étudier les algorithmes de classification et de faire varier leurs paramètres suivants :

Type de classifieur	**Paramètres pouvant varier**
Arbre de décision	critère (gini, entropy)
SVM	LinearSVC, kernel (linear, poly, rbf), C, gamma, degree
kPPV	n_neighbors, weights (uniform, distance)
Naïf Bayésien	Gaussian, Multinomial, Bernoulli

Pour évaluer les performances de ces classifieurs, je crée pour chacun d'entre eux un pipeline prenant le vectoriseur et le transformeur vus précédemment et le classifieur lui-même. Puis, j'utilise le GridSearchCV pour faire de la validation croisée à partir de ce pipeline.
Le pipeline du classifieur GaussianNB doit prendre un deuxième transformeur ***DenseTransformer*** pour transformer le texte en matrice.

J'ai choisi d'utiliser plusieurs critères d'évaluation :
— la proportion d'exemples bien classés
— le nombre de spams bien détectés
— le nombre de non spams (hams) bien détectés
— la précision moyenne
— l'aire sous la courbe ROC
Ces cinq fonctions de *scoring* sont définies dans le module ***Metrics.py***. Pour utiliser une de ces mesures de performance dans le GridSearchCV, il faut la transformer en *scorer* pour pouvoir la passer en attribut *scoring* du GridSearchCV à l'aide de la méthode *scikit-learn* : *make_scorer()*.

La proportion d'exemples bien classés est en fait le nombre de spams et de hams bien détectés divisé par le nombre total d'exemples.
Le nombre de spams bien détectés et le nombre de hams bien détectés s'obtiennent à partir de la matrice de confusion suivante, obtenue par la méthode *confusion_ matrix()* :

		Classes prédites	
		0 (Ham)	1 (Spam)
Classes réelles	0 (Ham)	Vrai Négatif (ham bien détecté)	Faux Positif (ham mal détecté)
	1 (Spam)	Faux Négatif (spam mal détecté)	Vrai Positif (spam bien détecté)

La précision moyenne s'obtient grâce à une méthode *average_ precision_ score()* et représente en fait l'aire sous la courbe Rappel/Précision. L'aire sous la courbe ROC s'obtient par la méthode *roc_ auc_ score()*.

Pour chacun des critères d'évaluation, j'effectue l'évaluation de performance des classifieurs définie ci-dessus (GridSearchCV). J'enregistre pour chaque évaluation la moyenne (sur les 5 validations croisées) des 5 scores de chaque combinaison de paramètres de chaque classifieur. Il y a au total 94 résultats pour chaque critère. Ceux-ci sont sauvegardés dans le répertoire *resultats*.

Pour lancer la classification des SMS avec tous ces critères et tous ces classifieurs, il faut taper la commande :

```
python3 classification_SMS.py
```

Ce script extrait la base de données de SMS et envoie la taille de test *0.7* et cette base au module général de classification ***ClassificationDeTexte*** qui se charge de tout le reste. De cette manière, je vais pouvoir réutiliser le module général pour ma base d'emails.

2.2 Interprétations

2.2.1 Arbre de décision (DecisionTree)

Un arbre de décision est une représentation graphique (arborescente) séparant «le plus rapidement possible» le plus d'exemples par rapport à la classe. L'arbre de décision produit un ensemble de règles. Un exemple est classé en feuille de l'arbre par une seule règle car les règles de l'arbre de décision partitionnent l'espace de recherche.
Il peut être visualisé et a l'avantage d'être facilement compréhensible et interprétable.

J'ai choisi de tester deux critères de sélection d'un attribut :

— **L'entropie de Shannon :** c'est une fonction mathématique qui correspond à la quantité d'information contenue ou délivrée par une source d'information. Plus la source émet d'informations différentes, plus l'entropie (ou incertitude sur ce que la source émet) est grande. L'entropie permet de mesurer le désordre. Sa formule est :

$$H_b(X) = -\mathbb{E}[\log_b P(X = x_i)] = \sum_{i=1}^{n} P_i \log_b \left(\frac{1}{P_i}\right) = -\sum_{i=1}^{n} P_i \log_b P_i.$$

— **L'index (ou coefficient) de Gini :** c'est une mesure statistique de la dispersion d'une distribution dans une population donnée. Elle permet de mesurer l'impureté. C'est un nombre variant de 0 à 1, où 0 signifie l'égalité parfaite et 1 signifie une inégalité parfaite.

	Jeu de paramètres	
Mesure de performance	DecisionTree('criterion' : 'gini')	DecisionTree('criterion' : 'entropy')
Pourcentage d'exemples bien classés	0.952	0.950
Nombre de spams bien détectés	400.200	386.200
Nombre de hams bien détectés	3310.600	3304.600
Précision moyenne	0.825	0.813
Aire sous la courbe ROC	0.865	0.865

TABLE 1 – Tableau comparatif pour l'arbre de décision

En comparant les résultats de ce tableau, je remarque que le premier algorithme classe mieux les spams (400.2) ET et les hams (3310.6) que le second, il a un nombre plus élevé pour ces deux scores. Il a aussi une meilleure précision moyenne (0.825), et la même aire sous la courbe ROC que le second : 0.865, ce qui est plutôt une bonne aire.
L'index de Gini semble plus performant que l'entropie de Shannon sur les SMS.

2.2.2 Machine à vecteurs supports (SVM)

Une machine à vecteurs de support (SVM) est un algorithme de classification qui montre de bonnes performances dans la résolution de problèmes variés tels que la reconnaissance ou la catégorisation de textes.
L'objectif du SVM est de maximiser la séparation entre les classes.

J'ai choisi de tester quatre types de SVM :
- **le LinearSVC :** il est similaire au SVM avec le noyau linéaire, sauf qu'il est implémenté dans *liblinear* et non dans *libsvm* comme les autres. Il a plus de flexibilité dans le choix des paramètres et des fonctions de perte et devrait donc mieux évoluer pour un grand nombre d'échantillons.
- **le SVM avec noyau linéaire :** le noyau linéaire est la fonction la plus simple du noyau. Elle est donnée par le produit scalaire <x, y>, plus une constante quelconque. Il peut fonctionner correctement sur de nombreux problèmes de classification.
- **le SVM avec noyau polynomial :** le noyau polynomial est un noyau «non-stationnaire». Il est bien adapté pour des problèmes où toutes les données d'entraînement sont normalisées.
- **le SVM avec noyau gaussien (Radial Basis Function) :** il gère parfaitement les problèmes non linéaires et est un bon noyau par défaut pour la classification.

Je compare maintenant ces quatres SVMs avec leur meilleur jeu de paramètres :

	Jeu de paramètres			
Mesure de performance	LinearSVC('C' : 1000)	SVC('C' : 100, 'kernel' : 'linear')	SVC('C' : 4, 'degree' : 2, 'kernel' : 'poly')	SVC('C' : 1000, 'gamma' : 0.001, 'kernel' : 'rbf')
Pourcentage d'exemples bien classés	0.983	0.983	0.864	0.983
Nombre de spams bien détectés	476.000	475.800	0.000	475.200
Nombre de hams bien détectés	3358.400	3359.000	3371.800	3360.000
Précision moyenne	0.942	0.943	0.568	0.943
Aire sous la courbe ROC	0.947	0.947	0.500	0.946

TABLE 2 – Tableau comparatif pour le SVM

En comparant les résultats de ce tableau, je remarque que le SVM de noyau polynomial détecte plus de hams que les autres (3371.8), mais il ne détecte AUCUN spam. Ce qui fait que sa

proportion d'exemples bien détectés (0.864) est nettement inférieure à celle des autres (0.983). Sa précision moyenne est par conséquent très mauvaise (0.568) et son aire sous la courbe ROC est insuffisante (0.5). On peut dire que c'est un modèle qui prédit au hasard.
En ce qui concerne les trois autres classifieurs, le SVM à noyau linéaire se distingue en ayant la meilleure précision moyenne (0.943) et la meilleure aire sous la courbe ROC (0.947). Ce qui fait de lui un classifieur excellent. Mais les deux autres SVM sont eux aussi excellents et le suivent de très près (de 0.001).

2.2.3 K plus proches voisins (kPPV)

Pour estimer la classe associée à un nouvel exemple x, la méthode des k plus proches voisins consiste à prendre en compte les k exemples d'apprentissage les plus «proches» (sur un plan à n dimensions où n est la taille des descripteurs de données) de ce nouvel exemple x, selon une distance donnée. Le sens de «proche» est défini par le type de pondération des points. La classe la plus représentée parmi les k exemples d'apprentissage voisins de x est alors retenue. Pour connaître la valeur de ce k, on utilise la validation croisée.

J'ai choisi de tester un critère, le poids de l'algorithme :
— **uniform :** tous les points du voisinnage ont le même poids.
— **distance :** les points ont un poids en fonction de leur distance. Les voisins plus proches de x ont une influence plus grande que les voisins qui sont plus éloignés.

Je compare maintenant ces deux types de kPPV avec leur meilleur jeu de paramètres, qui se trouve être composé d'un nombre de voisins de 15 pour les deux :

	Jeu de paramètres	
Mesure de performance	*kPPV('weights' : 'distance', 'n_neighbors' : 15)*	*kPPV('weights' : 'uniform', 'n_neighbors' : 15)*
Pourcentage d'exemples bien classés	0.955	0.950
Nombre de spams bien détectés	357.200	338.400
Nombre de hams bien détectés	3369.400	3368.800
Précision moyenne	0.856	0.840
Aire sous la courbe ROC	0.837	0.819

TABLE 3 – Tableau comparatif pour le kPPV

En comparant les résultats de ce tableau récapitulatif, je remarque que le classifieur travaillant avec des poids de distance est meilleur que celui travaillant avec des poids uniformes sur les cinq mesures de performance.

2.2.4 Classification Naïve Bayésienne (Naive Bayes)

La classification naïve bayésienne est un type de classification Bayésienne probabiliste simple basée sur le théorème de Bayes avec une forte indépendance (dite naïve) des hypothèses. Autrement dit, les descripteurs sont deux à deux indépendants conditionnellement aux valeurs de la variable à prédire.

J'ai choisi de tester trois types de classifieurs naïfs bayésiens différents :
— **Gaussien :** la vraisemblance (probabilité) des caractéristiques est supposée gaussienne. Formule :
$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$ Les paramètres σ_y et μ_y sont estimés en utilisant le maximum de vraisemblance.

— **Multinomial :** implémente l'algorithme de Bayes naïf pour les données distribuées multinomialement et est l'une des deux variantes Bayes naïves classiques utilisées dans la classification des textes. La distribution est paramétrée par les vecteurs et leur nombre de caractéristiques pour chaque classe.
— **De Bernouilli :** implémente les algorithmes naïfs de formation et de classification de Bayes pour les données qui sont distribuées selon des distributions multivariées de Bernoulli. C'est-à-dire qu'il peut y avoir de multiples caractéristiques mais chacune est supposée être une variable binaire (Bernoulli, booléenne), à la différence du multinomial. Par conséquent, cette classe requiert que les échantillons soient représentés sous forme de vecteurs fonctionnels à valeur binaire. Si un autre type de données est transmis, une instance ***BernoulliNB*** peut binariser son entrée (en fonction du paramètre *binarize*). La règle de décision pour cet algorithme est basée sur : $P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$.
Dans le cas de la classification de texte, ***BernoulliNB*** pourrait avoir de meilleurs résultats sur certains ensembles de données, en particulier ceux avec des documents plus courts.

Il est donc intéressant d'évaluer ces trois modèles dans le cadre de ce projet.

Je compare maintenant ces trois types de classifieurs naïfs bayésiens, en laissant par défaut le paramètre alpha à 1 pour le multinomial et le bernouilli :

Naive Bayes	**Jeu de paramètres**		
Mesure de performance	*Gaussian()*	*Bernoulli()*	*Multinomial()*
Pourcentage d'exemples bien classés	0.913	0.958	0.930
Nombre de spams bien détectés	457.400	365.600	255.600
Nombre de hams bien détectés	3105.200	3370.600	3371.800
Précision moyenne	0.757	0.864	0.776
Aire sous la courbe ROC	0.892	0.845	0.741

TABLE 4 – Tableau comparatif pour la classification Naïve Bayésienne

En comparant les résultats de ce tableau comparatif, je remarque que celui de Bernouilli a le meilleur pourcentage d'exemples bien classés (0.958) et la meilleure précision moyenne (0.864). Quant au gaussien, il détecte largement le plus grand nombre de spams (457.4) et a une bonne aire sous la courbe ROC (0.892). Tandis que le multinomial détecte le plus grand nombre de hams (3371.8). Mais ce dernier a aussi une aire sous la courbe ROC de 0.741, correcte, mais nettement inférieure à celle des autres.
J'en déduis que le meilleur classifieur parmi les trois est celui de Bernouilli car il est plus précis que le gaussien et certes, il détecte moins de spams que lui (365.6), mais beaucoup plus de hams (3370.6 par rapport à 3105.2).

De plus, la priorité d'un bon détecteur de spams est de ne pas classer un ham comme un spam. En effet, cela est à éviter absolument car si une personne attend un email urgent et important et qu'elle ne le retrouve jamais car la messagerie l'a placé dans la boîte à spams, c'est très embêtant...
La classe 0 (ou ham) joue alors un rôle majeur, la bonne prédiction de cette classe est estimée plus importante que la détection de l'autre classe, il faut alors privilégier la précision moyenne (rappel et précision) et le nombre de hams bien détectés.

3 Base d'Emails

3.1 Réalisation

La base d'emails est séparée en deux dossiers : *Train* et *Test*. Le dossier *Train* est lui même découpé en trois dossiers contenant les exemples d'apprentissage : *spam*, *ham* et *ham difficile*. Chaque fichier contient le code source d'un email. Les fichiers sont donc classés selon le type de message qu'ils contiennent puisque le nom du dossier dans lequel ils sont représente le nom du label supervisé.

Dans un premier temps, je vais seulement faire de l'apprentissage sur le dossier *Train* (l'autre dossier n'étant pas classé). Le dossier contient 1720 emails au total, dont 1376 hams et 344 spams. Les deux classes ne sont donc pas équilibrées.
Une fois dans ce dossier, pour charger en mémoire les fichiers textes avec comme catégorie le nom du sous-dossier *spam*, *easy_ham* ou *hard_ham*, j'utilise la méthode *load_files()* de *scikit-learn*.

Comme pour les SMS, la classe 1 (positive) est *spam*. J'ai choisi de regrouper *easy_ham* et *hard_ham* afin de n'avoir qu'une autre classe 0 (négative) représentant les hams en général.

Le mail étant constitué d'une entête et d'un message, il a tout d'abord fallu supprimer la partie entête avant de pouvoir les traiter. Les deux parties étant séparées par une ligne vide ne contenant qu'un retour à la ligne, il est facile de nettoyer les emails en effaçant leur entête.

J'ai décidé d'utiliser deux tiers de *Train* pour l'ensemble d'apprentissage, et un tiers pour l'ensemble de test. Je suis obligé de séparer la base de *Train* afin d'avoir un ensemble de tests car, la base *Test* n'étant pas classée, le classifieur ne peut pas et apprendre sur la base *Train* et tester sur le même ensemble car il ferait alors du sur-apprentissage.

Pour faire un détecteur de spams performant pour les emails, il a d'abord fallu utiliser le module général de classification ***ClassificationDeTexte*** (exactement comme pour les SMS) pour évaluer sur cette base de mails les différents classifieurs choisis sur les différentes mesures choisies. L'exécution de la classification pour la base des emails se lance avec la commande suivante :

```
python3 classification_Email.py
```

A la suite de cette éxécution, j'obtiens un fichier de résultats par mesure de performance sur les 94 classifieurs pour la base des emails. Tous ceux-ci sont enregistrés dans le répertoire *resultats/*.

3.2 Interprétations

L'objectif maintenant est de trouver les meilleurs paramètres et le meilleur classifieur pour faire un détecteur de spams performant.

Comme pour les SMS, j'analyse pour chaque type de classifieur les scores de toutes les combinaisons de paramètres, et j'en ressors le meilleur classifieur pour chacun des types :
- **Arbre de décision :** DecisionTree('criterion' : 'entropy')
- **SVM :** SVC(C=1000, gamma=0.01, kernel='rbf')
- **kPPV :** kPPV('weights' : 'distance', 'n_neighbors' : 6)
- **Naïf Bayésien :** Gaussian()

	Classifieur avec son meilleur jeu de paramètres			
Mesure de performance	*Arbre de décision*	*SVM*	*kPPV*	*Naïf Bayésien*
Pourcentage d'exemples bien classés	0.958	**0.982**	0.956	0.945
Nombre de spams bien détectés	108.200	**111.600**	106.200	89.400
Nombre de hams bien détectés	435.800	446.000	436.800	**447.400**
Précision moyenne	0.912	**0.962**	0.905	0.893
Aire sous la courbe ROC	0.939	**0.965**	0.932	0.873

TABLE 5 – Tableau comparatif des meilleurs classifieurs de chaque type de classification pour les mails

J'obtiens alors le tableau comparatif ci-dessus. D'après ce tableau, le meilleur classifieur semble être le SVM. En effet, il est le plus performant sur toutes les mesures sauf le nombre de hams bien détectés. Le classifieur bayésien le dépasse d'1.4 ham en moyenne, ce qui est négligeable sachant que le SVM domine les quatre autres évaluations. Il a notamment une excellente aire sous la courbe ROC (0.965) et la meilleure précision moyenne. De plus, il classe correctement 98.2% des emails de l'ensemble de test (de taille $1720 * 0.33 \simeq 568$ mails).
Par conséquent, je choisis le SVM pour apprendre sur la totalité de la base d'apprentissage (répertoire *Train*) et prédire ensuite la classe des mails du répertoire *Test*.

Pour éxécuter mon détecteur de spams et obtenir ainsi mon fichier *resultats_TEST_Condaminet.txt* contenant mes résultats de la classification des emails de la base *Test*, il suffit de lancer la commande suivante :

```
python3 DetecteurDeSpam.py
```