Projet L3
Ingénierie des langues
Fouille de données

GOEHRY Martial
16711476

10 décembre 2022

Table des matières

1 Introduction

Ce projet a pour but de développer un modèle permettant de catégoriser des emails en spam ou ham. La définition d'un spam dans le dictionnaire *Larousse* est :

> "Courrier électronique non sollicité envoyé en grand nombre à des boîtes aux lettres électroniques ou à des forums, dans un but publicitaire ou commercial."

Il est possible d'ajouter à cette catégorie tous les mails indésirables comme les tentatives d'hameçonnage permettant de soutirer des informations personnelles à une cible.

L'objectif est de travailler uniquement sur les données textuelles issues du corps du mail. Nous avons donc en point de départ les éléments suivants :

- langue : anglais
- corpus : monolingue écrit
- type : e-mail

Déroulé Le développement de ce projet s'articule autour de 3 phases majeurs

- Phase 1 : Récupération des données
- Phase 2 : Analyse des caractéristiques
- Phase 3 : Construction du modèle

Phase 1 La phase 1 concerne la récolte des informations et les traitements minimums nécessaires pour la mise en base. Les objectifs de traitement de cette phase sont :

- Extraire les corps des mails et éliminer les méta-données superflues
- Eliminer les mails non anglais
- Eliminer les mails en doublons
- Eliminer les parties de textes non pertinente (liens, réponses, certaines ponctuations)

Cette phase se termine avec la mise en base des documents dans un index ElasticSearch.

Phase 2 La phase 2 vise à générer les données statistiques et numériques à partir des corps de mails. Lors de cette phase une analyse statistique manuelle des données est réalisée

Phase 3 La phase 3 regroupe tous les opérations d'exploitation des données et vise à développer et à créer un modèle de classement des mails et d'en évaluer les performances.

Afin de conserver une certaine cohérence dans le déroulé entre les phases et au vu du temps que j'ai pris pour réaliser ce projet l'ensemble des étapes est automatisé en langage Python. Seule la récolte initiale des mails a été réalisée à la main.

Le schéma ci-dessous donne une vue synthétique des étapes du projet

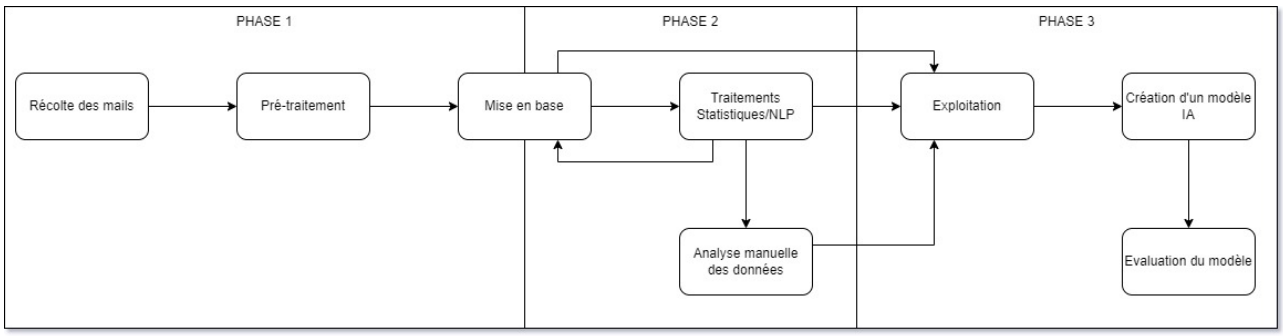

FIGURE 1 – Schéma des grandes étapes

2 Phase 1 : Récupération des données

La phase 1 est une pipeline qui vas permettre d'extraire un maximum d'information d'un email en essayant de pas dénaturer le fond ni la forme. Il pourra ensuite être stocké avec sa catégorie d'appartenance. Durant cette phase nous allons également initialiser les bases de données en créant les index (ES) et les tables (PSQL et SQLITE).

Ci-dessous le schéma général de cette phase.

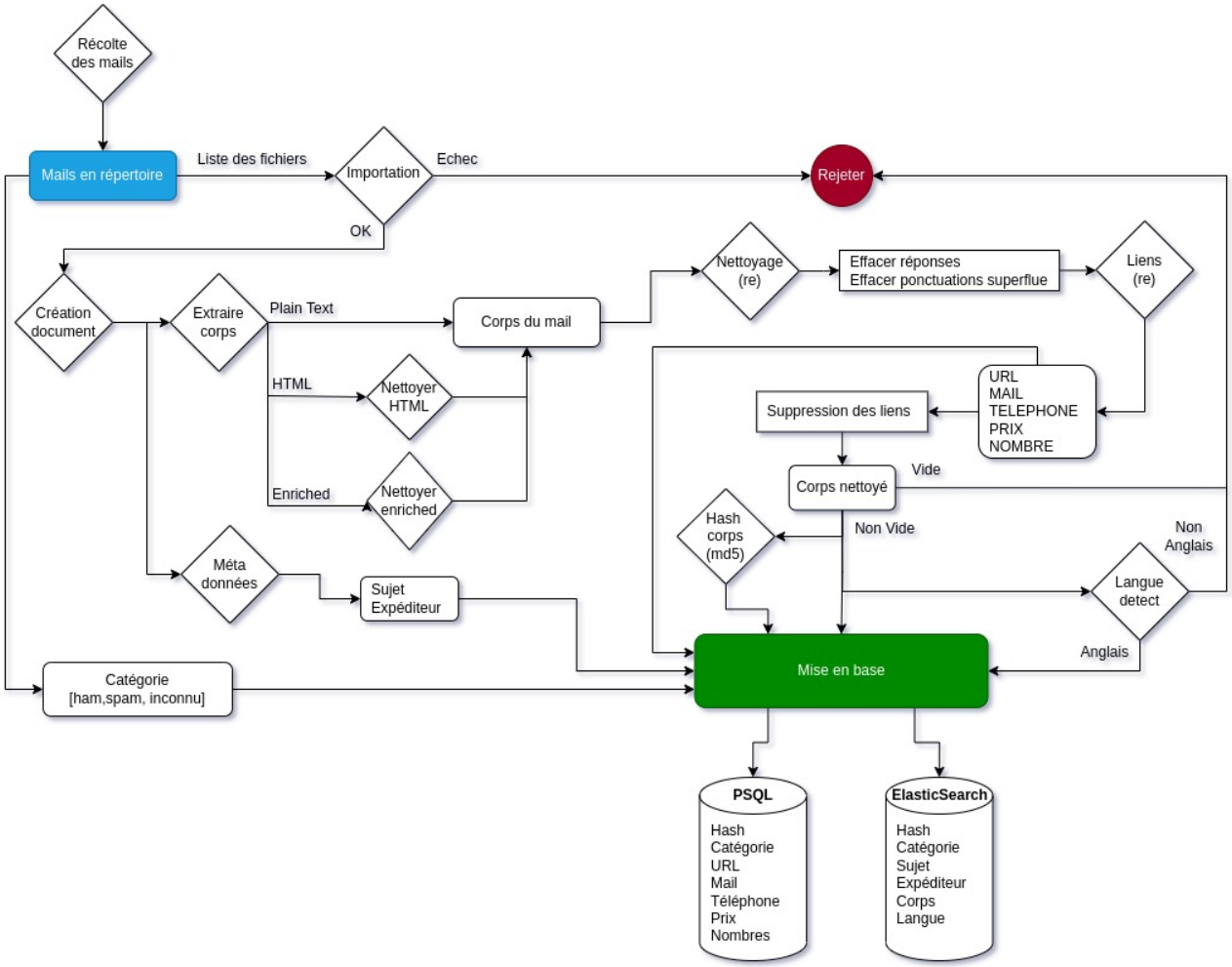

FIGURE 2 – Schéma des étapes de la phase 1

2.1 Récolte des données

Recherche de dataset J'avais dans l'idée de faire la recherche de mails en français. Cependant, je n'ai pas trouvé de dataset dans cette langue. Je me suis donc retourné vers les datasets de mails en anglais.

J'ai pu alors récupérer deux dataset :

- Enron company mails (voir : F.1)
- Dataset SpamAssassin (voir : F.1)

Les mails de SpamAssassin ont l'avantage d'être pré-trié, contrairement aux mails de la compagnie Enron. Ainsi le développement du moteur se fera uniquement avec les mails du SpamAssasin afin de pouvoir vérifier les résultats de l'analyse.

Téléchargement des données Le téléchargement du dataset Enron est possible a partir du moment où l'on possède un compte sur la plateforme Kaggle. Le dataset SpamAssassin est

ouvert, il suffit de télécharger les archives de chaque catégorie.

La récolte des données a été réalisée à la main sans automatisation. Les mails sont alors stockés dans plusieurs répertoires *HAM* et *SPAM* selon leur catégorie.

Format :
— *Enron* - 1 fichier CSV avec tous les mails
— *SpamAssassin* - 1 fichier texte par mail

2.2 Pré-traitement

Les étapes de pré-traitement regroupent toutes les étapes et actions réalisées avant la mise en base. L'objectif de ces étapes est d'extraire le message en retirant les métadonnées du mail. Il va être possible d'effectuer certain traitement de nettoyage et de récupération d'informations sommaires.

Les manipulations de messages dans Python se font principalement à l'aide du module *email* natif.

2.2.1 Importation

La fonction *email.message_from_binary_file* permet de transformer un fichier mail en objet python manipulable :

Fonction d'importation des fichiers

```
1  def import_from_file(chemin):
2      try:
3          with open(chemin, 'rb') as data:
4              msg = message_from_binary_file(data, policy=policy.default)
5              return msg
6
7      except FileNotFoundError:
8          print("Fichier : '{}' non trouve".format(chemin), file=sys.stderr)
9          return None
```

2.2.2 Extraction des corps des mails

Une fois le fichier importé au format *EmailMessage*, il est possible d'en extraire le corps.

Le corps du mail peut être composé de plusieurs parties qui ne sont pas forcément du texte. Les parties non textuelles ne sont pas conservée.

Extraction du corps du mail

```
1  def extract_body(msg):
2      refused_charset = ['unknown-8bit', 'default', 'default_charset',
3                         'gb2312_charset', 'chinesebig5', 'big5']
4      body = ""
5
6      if msg.is_multipart():
7          for part in msg.walk():
8              if not part.is_multipart():
9                  body += extract_body(part)
10         return body
11
```

```
12      if msg.get_content_maintype() != 'text':
13          return ""
14
15      if msg.get_content_charset() in refused_charset:
16          return ""
17
18      if msg.get_content_subtype() == 'plain':
19          payload = msg.get_payload(decode=True)
20          body += payload.decode(errors='ignore')
21
22      if msg.get_content_subtype() == 'html':
23          payload = msg.get_payload(decode=True)
24          body += nettoyage.clear_html(payload.decode(errors='ignore'))
25
26      if msg.get_content_subtype() == 'enriched':
27          payload = msg.get_payload(decode=True)
28          body += nettoyage.clear_enriched(payload.decode(errors='ignore'))
29
30      return body
```

2.2.3 Nettoyage

Le nettoyage du texte utilise principalement les expressions régulières pour retirer un maximum d'éléments indésirables dans le texte. J'utilise 2 modules externes afin de traiter le code HTML et faire la détection des mails qui ne sont pas écrit en anglais.

Par regex J'utilise le module python *re* pour générer les suivantes :

Suppression des réponses Lorsque l'on réponds à un mail, le texte du message précédent est conservé dans le corps du mail. Afin de permettre la distinction avec les mails précédent le caractère '>' est ajouter en début de ligne. Je retire toutes les lignes correspondant à des réponses afin de limiter les doublons.

Nettoyage des réponses

```
1 def clear_reply(texte):
2     pattern = re.compile('^>.*$', flags=re.MULTILINE)
3     return re.sub(pattern, '', texte)
```

Suppression des ponctuations Afin de ne pas surcharger la base de données et pour se concentrer sur le texte, une grande partie des caractères de ponctuation seront retirés. L'idée est de se concentrer sur les ponctuations les plus présentes (., ? !)

Nettoyage des ponctuations

```
1 def clear_ponctuation(texte):
2     pattern_ponct = re.compile('[*#\\-_=:;<>\\[\\]"\'~)(|/$+}{@%&\\\]',
      flags=re.MULTILINE)
3     return re.sub(pattern_ponct, ' ', texte)
```

Suppression des balises pour les enriched text Certaines parties du corps de mail sont de type *enriched text*. Les balises ne sont pas pertinente dans notre analyse et sont donc retirées.

Nettoyage des balises enriched text

```
def clear_enriched(texte):
    pattern = re.compile('<.*>')
    return re.sub(pattern, '', texte)
```

Suppression des liens Certaines informations présentent dans le texte ne peuvent pas être utilisées dans l'analyse textuelle. Cependant il peut être intéressant de conserver une trace de leur présence. Nous allons donc modifier les liens url, mail et les numéros de téléphone qui seront comptabilisé avant d'être retiré du texte.

Nettoyage des liens

```
def change_lien(texte, liens):
    pattern_mail = re.compile('[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+')

    pattern_url1 = re.compile('(http|ftp|https)?:\/\/([\w\-_]+(?:(?:\.[\w\-_]+)+))'
                              '([\w\-\.,@?^=%&:/~\+#]*[\w\-\@?^=%&/~\+#])?', flags=re.MULTILINE)
    pattern_url2 = re.compile('(\\w+\\.)+\\w+', flags=re.MULTILINE)
    pattern_tel1 = re.compile('\\(\\d{3}\\)\\d+-\\d+')  # (359)1234-1000
    pattern_tel2 = re.compile('\\+\\d+([ .-]?\\d)+')    # +34 936 00 23 23

    temp, liens['MAIL'] = re.subn(pattern_mail, ' ', texte)

    temp, liens['URL'] = re.subn(pattern_url1, ' ', temp)
    temp, nb = re.subn(pattern_url2, ' ', temp)
    liens['URL'] += nb

    temp, liens['TEL'] = re.subn(pattern_tel1, ' ', temp)
    temp, nb = re.subn(pattern_tel2, ' ', temp)
    liens['TEL'] += nb

    return temp
```

Suppression des nombres Comme pour les liens, les nombres sont comptabilisés et retirés. Je fais la distinction entre les nombres seuls et les nombres accompagnés de sigle monétaires.

```
monnaie = '€$£'
```

Nettoyage des nombres

```
def change_nombres(texte, liens):
    pattern_prix1 = re.compile(f'[{monnaie}]( )?\\d+([.,]\\d+)? ', flags=re.MULTILINE)
    pattern_prix2 = re.compile(f' \\d+([.,]\\d+)?( )?[{monnaie}]', flags=re.MULTILINE)
    pattern_nb = re.compile('\\d+')
```

```
5
6       temp, liens['PRIX'] = re.subn(pattern_prix1, ' ', texte)
7       temp, nb = re.subn(pattern_prix2, ' ', temp)
8       liens['PRIX'] += nb
9
10      temp, liens['NOMBRE'] = re.subn(pattern_nb, ' ', temp)
11
12      return temp
```

Par module J'ai utilisé deux modules externes plus performant que ce que j'aurais pu faire avec simplement des expressions régulières.

Suppression du code HTML Certaines parties du corps du mail sont de type HTML. J'utilise le module *BeautifulSoup* pour parser le code et récupérer le texte affiché.

Nettoyage des nombres

```
1  from bs4 import BeautifulSoup
2
3  def clear_html(texte):
4      brut = BeautifulSoup(texte, "lxml").text
5      return brut
```

Sélection des mails en anglais Lors de mes tests, je me suis rendu compte que certains mails n'étaient pas en anglais. J'ai donc trouvé le module *langdetect* qui permet de détecter le langage utilisé dans un texte en utilisant un modèle Naïve Bayes avec une précision de 99% (voir F.2).
Je conserve dans les données à mettre en base le langage détecté dans l'idée de pouvoir traité plusieurs langues (en idée d'évolution).

La détection de la langue se fait dans la fonction s'occupant de créer le document pour la mise en base ElasticSearch.

Création d'un document

```
1  import langdetect
2
3  def create_document(mail, categorie):
4      corp = mail_load.extract_body(mail)
5      corp, liens = nettoyage.clear_texte_init(corp)
6      sujet, expediteur = mail_load.extract_meta(mail)
7
8      if not corp:
9          return None
10
11     try:
12         lang = langdetect.detect(corp)
13     except langdetect.lang_detect_exception.LangDetectException:
14         return None
15
16     if lang != 'en':
17         return None
18
```

```
    if categorie.lower() not in ['spam', 'ham']:
        categorie = 'inconnu'

    doc = {
        'hash': hashlib.md5(corp.encode()).hexdigest(),
        'categorie': categorie.lower(),
        'sujet': sujet,
        'expediteur': expediteur,
        'message': corp,
        'langue': lang,
        'liens': liens
    }
    return doc
```

Exemple de traitement Les sections suivantes présentent des exemples de traitement de la phase 1.

Traitement initial

```
message = '''
Message dedicated to be a sample to show how the process is clearing the
    text.

Begin reply :
> He once said
>>> that it would be great
End of reply.

Substitutions :
spamassassin-talk@example.sourceforge.net
https://www.inphonic.com/r.asp?r=sourceforge1&refcode1=vs3390
hello.foo.bar
between $ 25 and 25,21 $

A number is : 2588,8 588
Phone type a : (359)1234-1000
Phone type b : +34 936 00 23 23
Ponctuation : ---## ..
~ ~~~~~
'''
text, liens = clear_texte_init(message)
print(liens)
print(text)

```

Résultat traitement initial :

```
{'URL': 2, 'MAIL': 1, 'TEL': 2, 'NOMBRE': 3, 'PRIX': 2}

Message dedicated to be a sample to show how the process is clearing the text.

Begin reply
```

```
End of reply.

Substitutions



between and

A number is  ,
Phone type a
Phone type b
Ponctuation   ..
```

Traitement HTML

```python
message_html = '''
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>Foobar</title>
</head>
<body>
I actually thought of this kind of active chat at AOL
bringing up ads based on what was being discussed and
other features
  <pre wrap="">On 10/2/02 12:00 PM, "Mr. FoRK"
  <a class="moz-txt-link-rfc2396E" href="mailto:fork_
  list@hotmail.com">&lt;fork_list@hotmail.com&gt;</a>
  wrote: Hello There, General Kenobi !?
<br>
</body>
</html>
'''
print(clear_html(message_html))

```

Résultat traitement HTML :

```
Foobar


I actually thought of this kind of active chat at AOL
bringing up ads based on what was being discussed and
other features
  On 10/2/02 12:00 PM, "Mr. FoRK"
  <fork_list@hotmail.com>
  wrote: Hello There, General Kenobi !?
```

Traitement enriched text

```
message_enriched = '''
<smaller>I'd like to swap with someone also using Simple DNS to take
advantage of the trusted zone file transfer option.</smaller>
'''
print(clear_enriched(message_enriched))

```

Résultat traitement enriched text :

```
I'd like to swap with someone also using Simple DNS to take
advantage of the trusted zone file transfer option.
```

2.2.4 Mise en base

Cette section détaille les éléments relatifs à la mise en base des informations récoltées. Dans ce projet, j'utilise 2 moteurs de bases de données pour stocker les extractions des mails.

1. un index ElasticSearch pour faire le stockage des données textuelles
2. une base PostgreSQL pour le stockage des données numériques

J'utilise des conteneurs *docker* pour héberger les services de bases de données. L'utilisation des conteneurs me permet de partager plus facilement mes configurations et limite les erreurs d'installations.

Pour chaque mail récolté le programme de la phase 1 va générer un *document* avec les informations suivantes :

— hash - signature md5 du texte nettoyé
— catégorie - Ham, Spam ou Inconnu
— sujet - correspond à l'objet du mail
— expéditeur - adresse mail
— corps - corps du mail nettoyé
— langue - la langue détectée du mail (en)
— liens - données non textuelles extraites du corps :
  - URL - liens URL
  - Mail - adresses mail
  - Téléphone - numéros de téléphone
  - Prix - nombres avec un symbole de devise
  - Nombres

Chaque document va générer une entrée dans la base ElasticSearch et une entrée dans la base PostgreSQL.

Création d'un document

```
def create_document(mail, categorie):
    corp = mail_load.extract_body(mail)
    corp, liens = nettoyage.clear_texte_init(corp)
    sujet, expediteur = mail_load.extract_meta(mail)

    if not corp:
        return None

    try:
        lang = langdetect.detect(corp)
    except langdetect.lang_detect_exception.LangDetectException:
        return None
```

```
13
14      if lang != 'en':
15          return None
16
17      if categorie.lower() not in ['spam', 'ham']:
18          categorie = 'inconnu'
19
20      doc = {
21          'hash': hashlib.md5(corp.encode()).hexdigest(),
22          'categorie': categorie.lower(),
23          'sujet': sujet,
24          'expediteur': expediteur,
25          'message': corp,
26          'langue': lang,
27          'liens': liens
28      }
29      return doc
30
```

Ci-dessous le schéma des bases de données avec les relations entre elles.

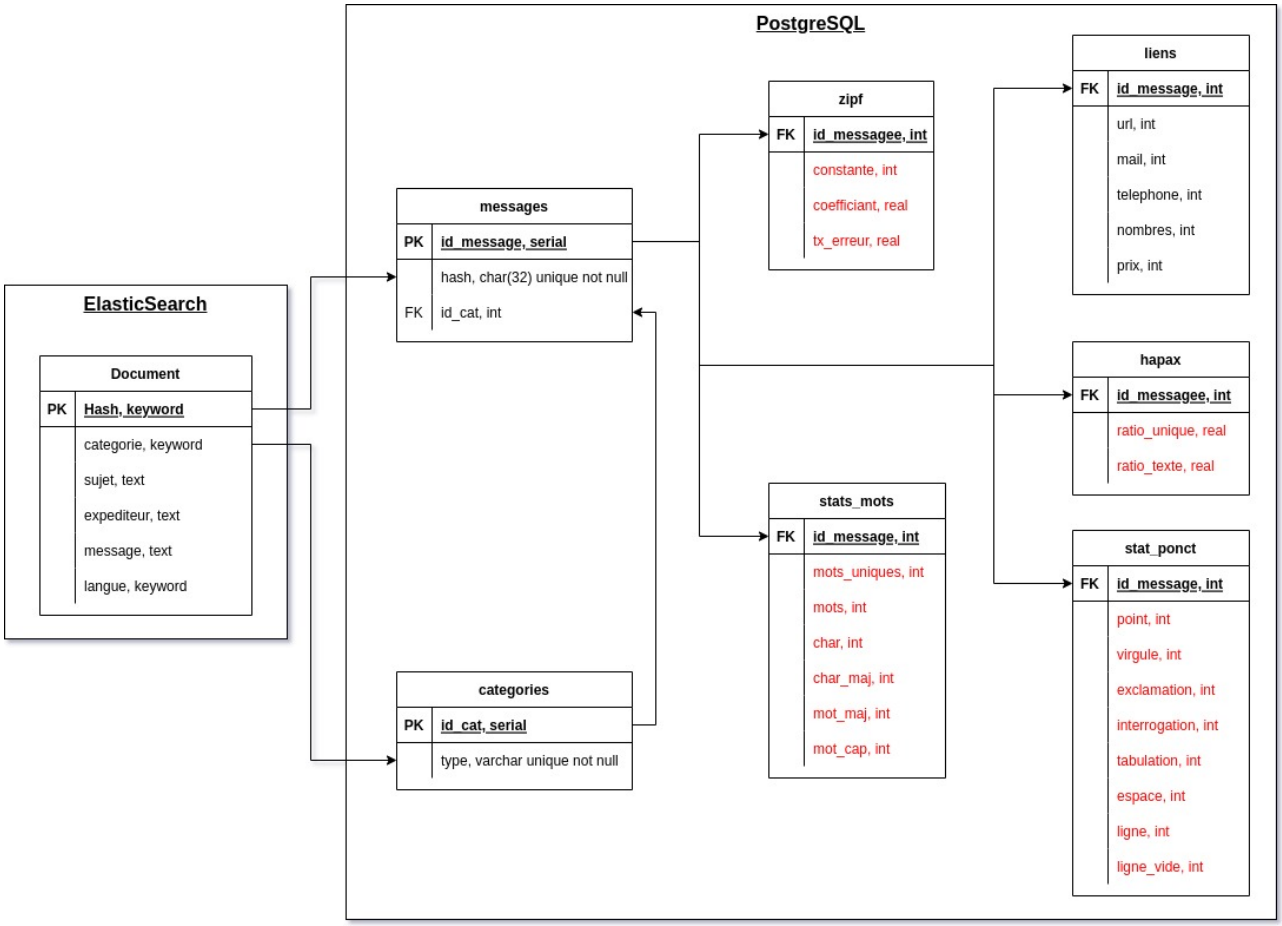

FIGURE 3 – Schéma des bases de données de l'application

Le *hash* calculé lors de la phase de traitement est l'identifiant unique du mail dans toutes les bases. La catégorie du mail est également présente dans les deux bases. Les champs en rouge sont des caractéristiques qui ne sont pas calculées lors de la phase 1.

Stockage des données : ElasticSearch ElasticSearch est un moteur de base de données NoSQL. Il intègre un moteur d'indexation des documents est assez performant pour stocker des données textuelles de taille aléatoire. Il est cependant assez compliqué de modifier le schéma d'un index une fois qu'il a été créé. Pour ces raisons, j'utilise cette technologie pour ne stocker les données textuelles après nettoyage car elles n'ont plus vocation à être modifiées.

Les corps de mail mise en base seront récupérés ultérieurement pour réaliser les opérations d'analyse. Les résultats seront stockés dans la base PostgreSQL plus souple.

Ci-dessous les fonctions principales pour l'ajout d'un document dans la base ES.

Fonctions basique pour la liaison ElasticSearch

```python
def es_connect(server, creds, crt):
    """ Connexion au serveur ElasticSearch """
    client = Elasticsearch(server, api_key=creds, ca_certs=crt)

    try:
        client.search()
        client.indices.get(index="*")
        return client

    except (exceptions.ConnectionError, AuthenticationException,
    AuthorizationException) as err:
        print("ES:conn - Informations client ElasticSearch :\n\t", err)
        client.close()
        return None


def es_index_doc(es_cli, index, doc):
    """ Index un document dans la base ES """
    id_doc = doc['hash']

    if es_document_exists(es_cli, index, id_doc):
        return 1

    es_cli.index(index=index, document=doc)
    es_cli.indices.refresh(index=index)
    return


def es_document_exists(es_cli, index, hash):
    """ Regader dans l'index si le hash du document est deja present """
    try:
        resp = es_cli.search(index=index, query={"match": {"hash": hash}})
    except elasticsearch.NotFoundError as err:
        print("Error : hash", err, file=sys.stderr)
        return None

    return True if resp['hits']['total']['value'] == 1 else False

```

Le déploiement et les configurations de la base ElasticSearch sont disponibles dans l'annexe B.1.

Stockage des premières informations statistiques : PostgreSQL Pour le stockage des données statiques et d'analyse, j'ai décidé de m'orienter vers le système de gestion de base de données PostgreSQL. Une base de données relationnelle est plus flexible qu'un index Elastic-Search pour l'ajout de nouvelles caractéristiques.

Durant la phase 1, j'utilise que 3 tables :
— messages - liste des mails avec le hash permettant de faire le lien avec l'index ES
— categorie - liste des catégories de mails
— liens - chaque ligne contient toutes les nombres de liens retiré lors du nettoyage
Cette base a pour but de stocker les données formatées pour l'analyse, l'exploitation et l'entrainement du modèle.

Fonctions basiques pour la liaison PostgreSQL

```
1  def connect_db(database, user, passwd, host, port):
2      """ Connexion a la base de donnees Postgres """
3      try:
4          client_psql = psycopg2.connect(database=database, user=user,
       password=passwd, host=host, port=port)
5      except psycopg2.Error as e:
6          print("Erreur de connexion : \n{}".format(e), file=sys.stderr)
7          return None
8
9      client_psql.autocommit = True
10     return client_psql
11
12
13 def insert_data(client_psql, table, data):
14     """
15     Insere les donnees d'un dictionnaire dans une table de la base de
       donnees PSQL
16     Les cles du dictionnaire doivent correspondre aux colonnes de la table
       .
17     """
18     cols = ','.join([str(c) for c in data.keys()])
19     vals = ','.join([str(v) if (type(v) != str) else f"'{v}'" for v in
       data.values()])
20     query = f"INSERT INTO {table}({cols}) VALUES ({vals})"
21
22     exec_query(client_psql, query)
23
24
25 def get_data(client_psql, table, champs, clause=None):
26     """ Recupere les donnees de la base. """
27     query = f"SELECT {','.join(champs)} FROM {table}"
28     if clause:
29         query += f" WHERE {clause}"
30
31     result = exec_query(client_psql, query)
32     return [dict(zip(champs, ligne)) for ligne in result]
33
34
35 def insert_document_init(client_psql, data, id_cat):
36     """
```

```
37      Insere un nouveau document dans la base PSQL.
38      """
39      insert_data(client_psql, 'messages', {'hash': data['hash'], 'id_cat':
    id_cat})
40      id_message = get_data(client_psql, 'messages', ['id_message'], f"hash
    LIKE '{data['hash']}'")[0]['id_message']
41
42      liens = data['liens']
43      liens.update({'id_message': id_message})
44      insert_data(client_psql, 'liens', liens)
45
46  def exec_query(client_psql, query):
47      """ Execute une query dans la base PSQL """
48      cursor = client_psql.cursor()
49
50      try:
51          cursor.execute(query)
52          if query.upper().find("SELECT", 0, 6) >= 0:
53              return cursor.fetchall()
54          return []
55      except psycopg2.Error as e:
56          print("Erreur d'execution de la requete : {}".format(e), file=sys.
    stderr)
57          print("requete : {}".format(query), file=sys.stderr)
58          return []
59
```

Le déploiement et les configurations de la base PostgreSQL sont disponibles dans l'annexe B.2.

Stockage des données statistiques du traitement : SQLite Les données présentent dans cette base permettent de suivre l'évolution du traitement lors des différentes étapes de nettoyage. A chaque grandes étapes de la phase 1 (Importation, Nettoyage, Mise en base), je calcule pour les HAM, SPAM et (HAM+SPAM) les éléments suivants :

- mails - nombre de mails
- mots - nombre de mots
- mots_uniques - nombre de mots uniques

Ces données me permettent d'estimer la quantité de données nettoyées durant cette phase.

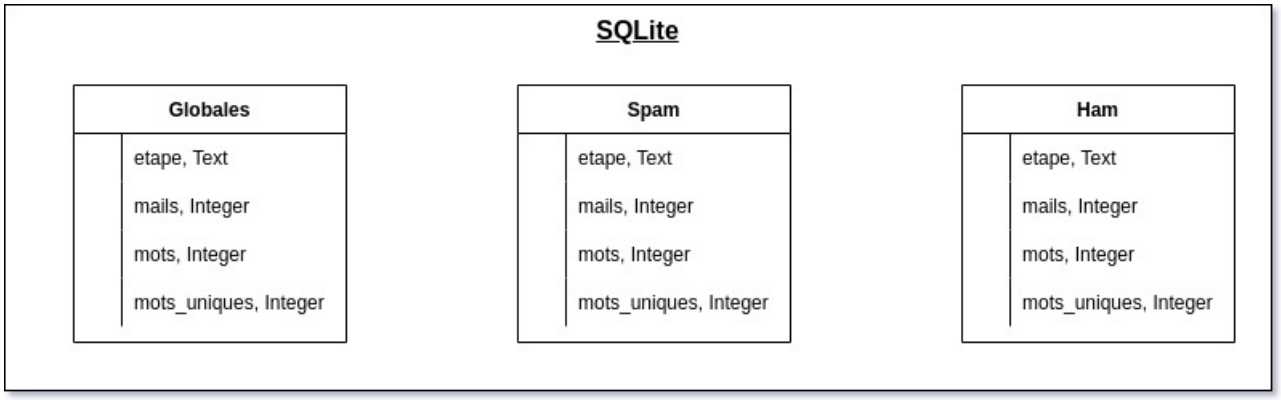

FIGURE 4 – Schéma de la base de données pour lors du traitement

2.3 Données de la phase 1

3 Phase 2

3.1 Traitement

3.1.1 Recherche de caractéristiques

Références

3.1.2 Analyse préliminaire

4 Phase 3

A Développement visualisation distribution de Zipf

Présentation La loi de distribution de Zipf est une loi empirique (basée sur l'observation) qui veut que le mot le plus fréquent est, à peu de chose près, 2 fois plus fréquent que le 2^{eme}, 3 fois plus fréquent que le 3^{eme} etc.

La formulation finale de la 1^{ere} loi de Zipf est la suivante :

$$|mot| = constante \times rang(mot)^{k \approx 1}$$

avec $|mot|$ la fréquence d'apparition d'un mot, *constante* une valeur propre à chaque texte, *rang(mot)* la place du mot dans le tri décroissant par fréquence d'apparition et k un coefficient proche de 1.

Développement Afin de pouvoir utiliser les résultats de cette distribution dans mon analyse, j'ai développé un ensemble de fonctions sur un corpus "*reconnu*". Mon choix s'est porté sur le corpus *Brown* (voir F.1) présent dans la librairie *nltk*. Ce corpus contient environ 500 documents contenant 1 millions de mot en anglais.

Le processus d'analyse se fait sur 2 versions de ce corpus.
— la première version contient tous les mots sans modifications
— le seconde version contient tous les mots sans les *stopwords*
Les *stopwords* sont des mots qui n'ont pas ou peu de signification dans un texte. Ces mots sont retirés dans la 2^{e} version pour voir l'effet d'une réduction sur la distribution de Zipf.

Les paragraphes ci-dessous détaillent les étapes du développement :

Étape 1 - Ordonner les mots La première étape est de compter les occurrences de tous les mots des 2 corpus et de les ranger en fonction de leur nombre d'occurrence.

Triage des mots

```
1  def frequence_mot(bag, freq=None):
2      """
3      Calcule la frequence de chaque mot dans un sac de mot
4      :param bag: <list> - liste de tous les mots d'un texte
5      :param freq: <dict> - dictionnaire avec {<str> mot: <int> frequence}
6      :return: <dict> - dictionnaire avec la frequence par mot {mot:
       frequence}
7      """
8      if freq is None:
9          freq = {}
10     for mot in bag:
11         freq[mot] = freq.get(mot, 0) + 1
12     return freq
13
14 def classement_zipf(dico):
15     """
16     Trie un dictionnaire de mots : occurence et leur assigne un rang en
       fonction du nombre d'occurence
17     :param dico: <dict> dictionnaire de mot: occurences
18     :return: <list> {"rang": <int>, "mot": <str>, "frequence": <int>}
```

```
19      """
20      ranked = []
21      for rang, couple in enumerate(sorted(dico.items(), key=lambda item:
        item[1], reverse=True), start=1):
22          ranked.append({"rang": rang,
23                         "mot": couple[0],
24                         "frequence": couple[1]})
25
26      return ranked
```

On obtient les représentations suivantes :

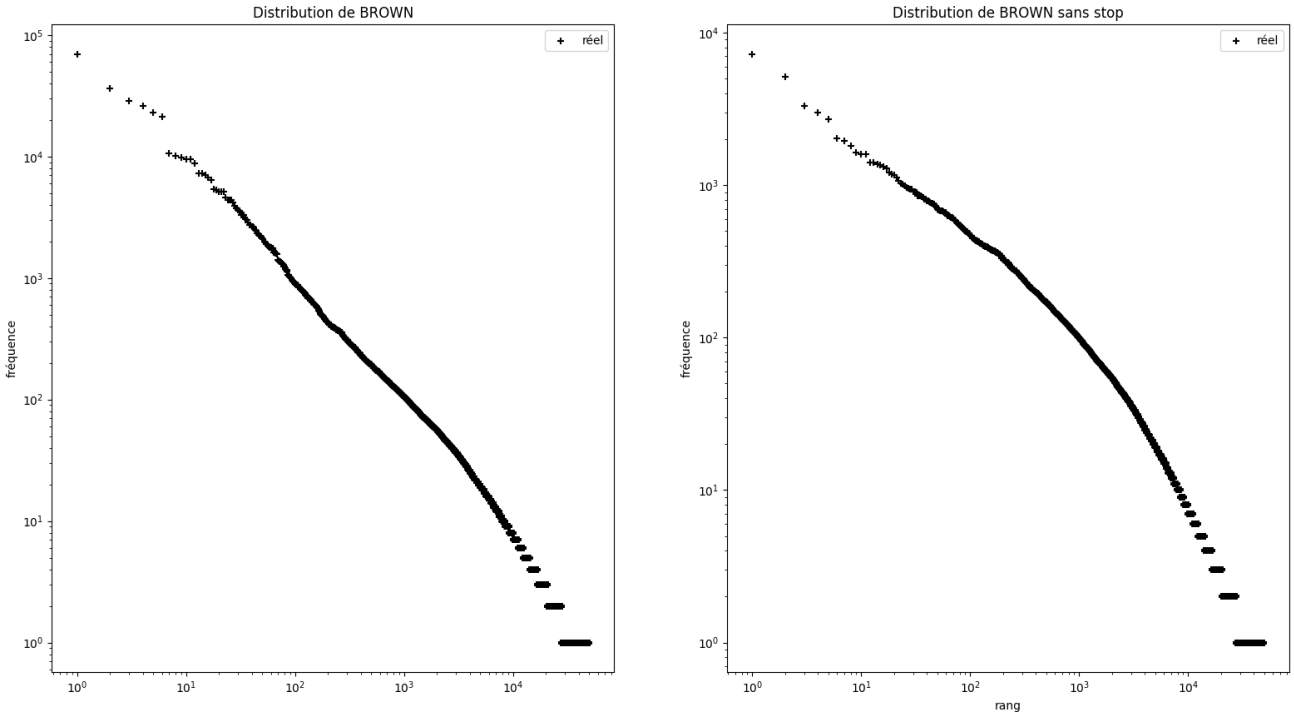

FIGURE 5 – Distribution de Zipf pour les deux corpus

— Nombre de mots dans brown : mots : 49398 occurences : 1012528
— Nombre de mots dans brown stop : mots : 49383 occurences : 578837

La distribution de la version complète du corpus semble à première vue plus fidèle à la représentation classique de la distribution de Zipf.

Etape 2 - calcul de la constante Le premier paramètre que je détermine est la *constante*. Pour ce faire j'effectue le calcul suivant pour tous les mots :

$$constante = |mot| \times rang(mot)$$

On obtient une liste de toutes les constantes théoriques pour chaque mot selon son rang. De cette liste, nous allons extraire la moyenne et la médiane.

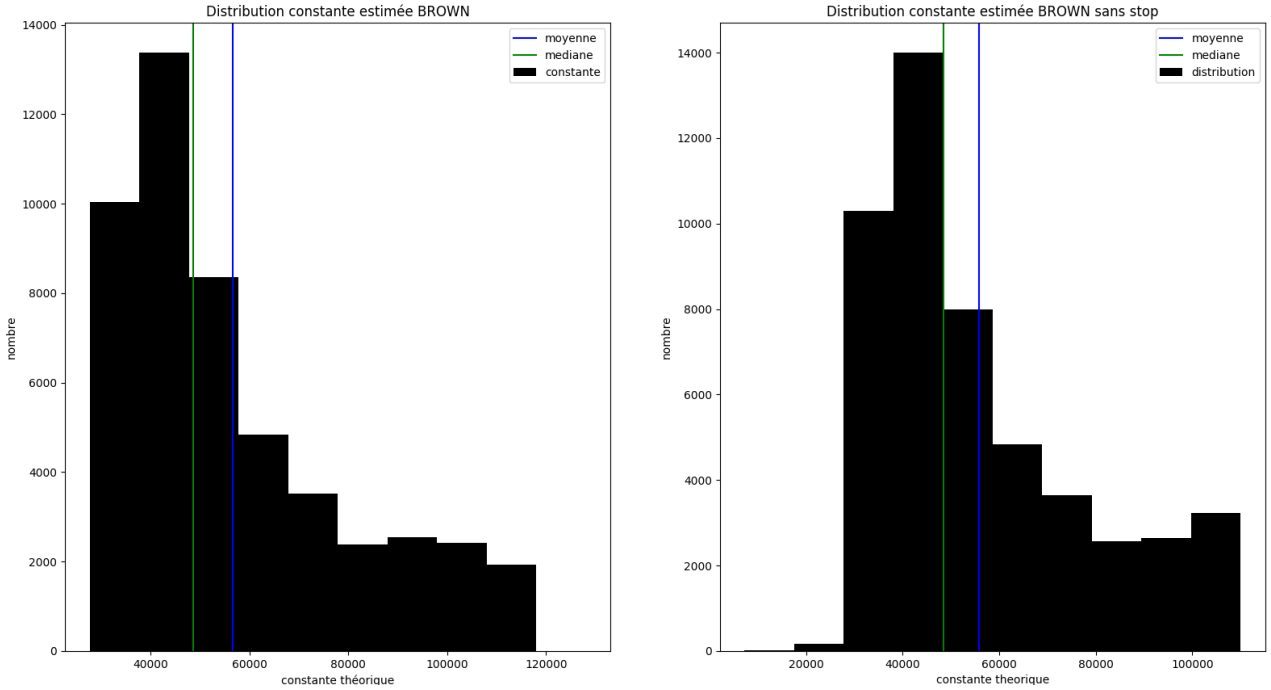

FIGURE 6 – Distribution des constantes théoriques pour les deux corpus

On voit qu'il y a une majorité de mots donnent une constante brute comprise entre 20.000 et 60.000. Dans les deux corpus La différence entre les moyennes et médianes des deux corpus n'est pas flagrante :

— Brown moyenne : 56525.81, médiane : 48601.50
— Brown (- stopwords) moyenne : 55809.97, médiane : 48494.00

Etape 3 - recherche du coefficient Le coefficient k permet d'ajuster le résultat, et pourra éventuellement donner une indication de complexité. La recherche de k se fera sur les deux corpus avec utilisant les moyennes et médianes.

Pour ce faire nous allons :

1. Faire la liste de tous les coefficients possibles dans l'intervalle $[0.86, 1.3]$ avec un pas de 0.01 [1].
2. Calculer toutes la fréquences théoriques de tous les rangs avec tous les coefficients possibles avec en constante la moyenne et la médiane de chaque corpus.
3. Calculer la moyenne des coûts absolus entre les fréquences théoriques par coefficient avec la fréquence réelle observée pour chaque corpus.

Le couple coefficient/constante avec le coup minimal sera retenu pour l'utilisation dans la phase de *feature engineering*.

Fonctions utilisées dans la recherche du coefficient

```
1 def zipf_freq_theorique(constante, rang, coef):
2     """
3     Calcul la frequence theorique d'un mot selon son rang, la constante du
      texte et un coeficiant d'ajustement
```

1. Borne et pas, totalement arbitraire afin d'obtenir un graphique présentable

```
    :param constante: <int> constante determinee par la distribution de Zipf
    :param rang: <int> rang du mot selon sa frequence
    :param coef: <float> variable d'ajustement
    :return: <float> frequence theorique zipfienne
    """
    return constante / (rang ** coef)

def cout(l1, l2, methode):
    """
    Calcul le cout de l'ecart entre les elements de l1 et le l2, place par place
    :param l1: <list> liste d'entier
    :param l2: <liste> liste d'entier
    :param methode: <str> methode de calcul du cout
    :return: <float> cout selon methode
    """
    if len(l1) != len(l2):
        print("Erreur, fonction cout: l1 & l2 de taille differente", file=sys.stderr)
        return None

    if len(l1) == 0:
        print("Erreur, fonction cout: liste vide", file=sys.stderr)

    if methode.lower() not in ['absolue', 'carre', 'racine']:
        print("Erreur, fonction cout - methode '{}' inconnue".format(methode), file=sys.stderr)
        return None

    if methode.lower() == 'absolue':
        return np.mean([abs(x-y) for x, y in zip(l1, l2)])

    if methode.lower() == 'carre':
        return np.mean([(x-y)**2 for x, y in zip(l1, l2)])

    if methode.lower() == 'racine':
        return np.sqrt(np.mean([(x-y)**2 for x, y in zip(l1, l2)]))

    return None
```

Calcul des fréquences par coefficient

```
    ls_coef = list(np.arange(0.86, 1.3, 0.01))
    zbmo_th = {coef: [stats.zipf_freq_theorique(zb_const_moyen, r, coef) for r in zb_rang] for coef in ls_coef}
    zbme_th = {coef: [stats.zipf_freq_theorique(zb_const_median, r, coef) for r in zb_rang] for coef in ls_coef}
    zbmoth_cmoy = [stats.cout(zb_freq, zbmo_th[coef], 'absolue') for coef in ls_coef]
    zbmeth_cmoy = [stats.cout(zb_freq, zbme_th[coef], 'absolue') for coef in ls_coef]

    zbsmo_th = {coef: [stats.zipf_freq_theorique(zbs_const_moyen, r, coef)
```

```
    for r in zbs_rang] for coef in ls_coef}
8   zbsme_th = {coef: [stats.zipf_freq_theorique(zbs_const_median, r, coef
    ) for r in zbs_rang] for coef in ls_coef}
9   zbsmoth_cmoy = [stats.cout(zbs_freq, zbsmo_th[coef], 'absolue') for
    coef in ls_coef]
10  zbsmeth_cmoy = [stats.cout(zbs_freq, zbsme_th[coef], 'absolue') for
    coef in ls_coef]
```

La recherche du coefficient nous retourne les éléments suivants :

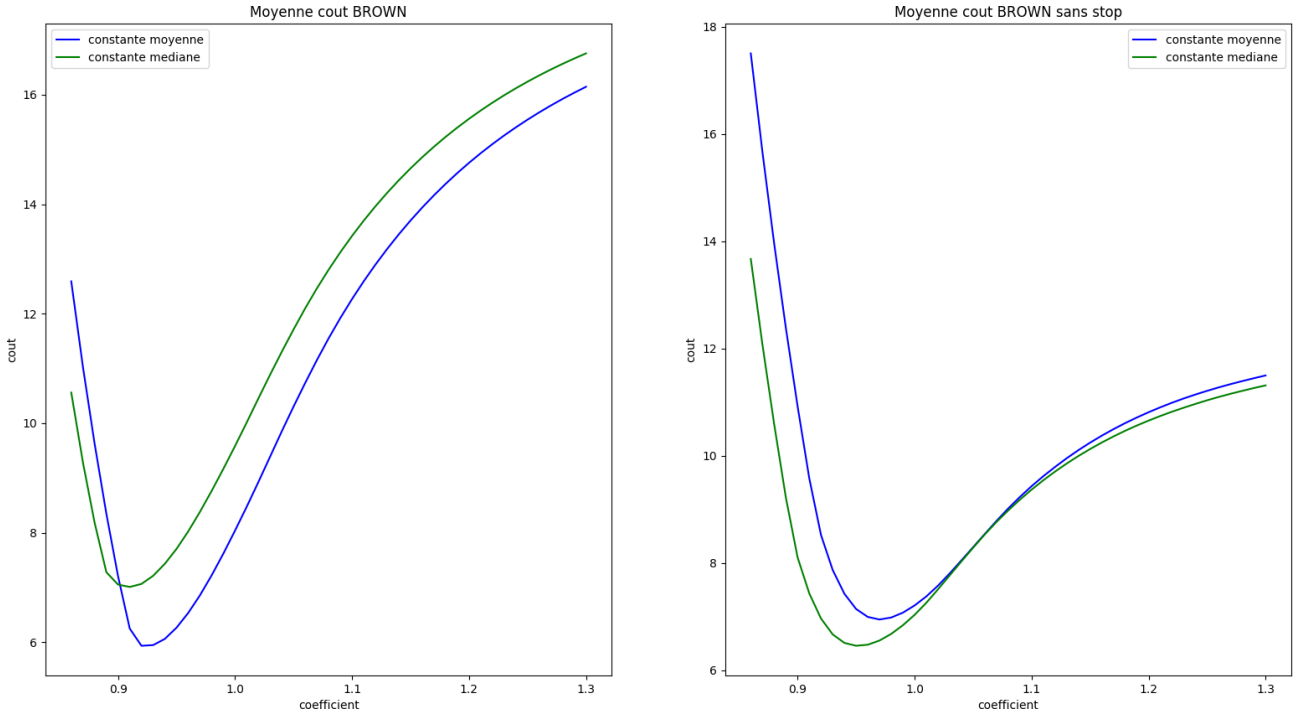

FIGURE 7 – Coût absolu moyen par coefficient

— Coût min brown moyenne : 5.93, median : 7.01
— Coût min brown (- stopwords) moyenne : 6.95, median : 6.46
— Coefficient min brown moyenne : 0.92, median : 0.91
— Coefficient min brown (- stopwords) moyenne : 0.97, median : 0.95

Résultats Le tableaux ci dessous rappelle les données récupérées au long de la recherche :

	BROWN avec stopwords	BROWN sans stopwords
nombre de mots uniques	49398	49383
nombre de mots total	1012528	578837
Constante moyenne	56525.81	55809.97
Constante médiane	48601.50	48494.00
Coefficient avec moyenne	0.92	0.97
Cout du coefficient moyenne	5.93	6.95
Coefficient avec médiane	0.91	0.95
Cout du coefficient médiane	7.01	6.46

D'après les données il est possible de dire que l'on obtient de meilleurs résultats si on conserve tous les mots du corpus. Dans ce cas l'utilisation de la moyenne des constantes génère un taux d'erreur plus faible que la médiane.

Ci-dessous la représentation des fréquences théoriques avec le coefficient optimal pour chaque corpus et chaque méthode. On voit que la courbe de la constante moyenne sur le corpus brute est celle qui suit le mieux les données réelles.

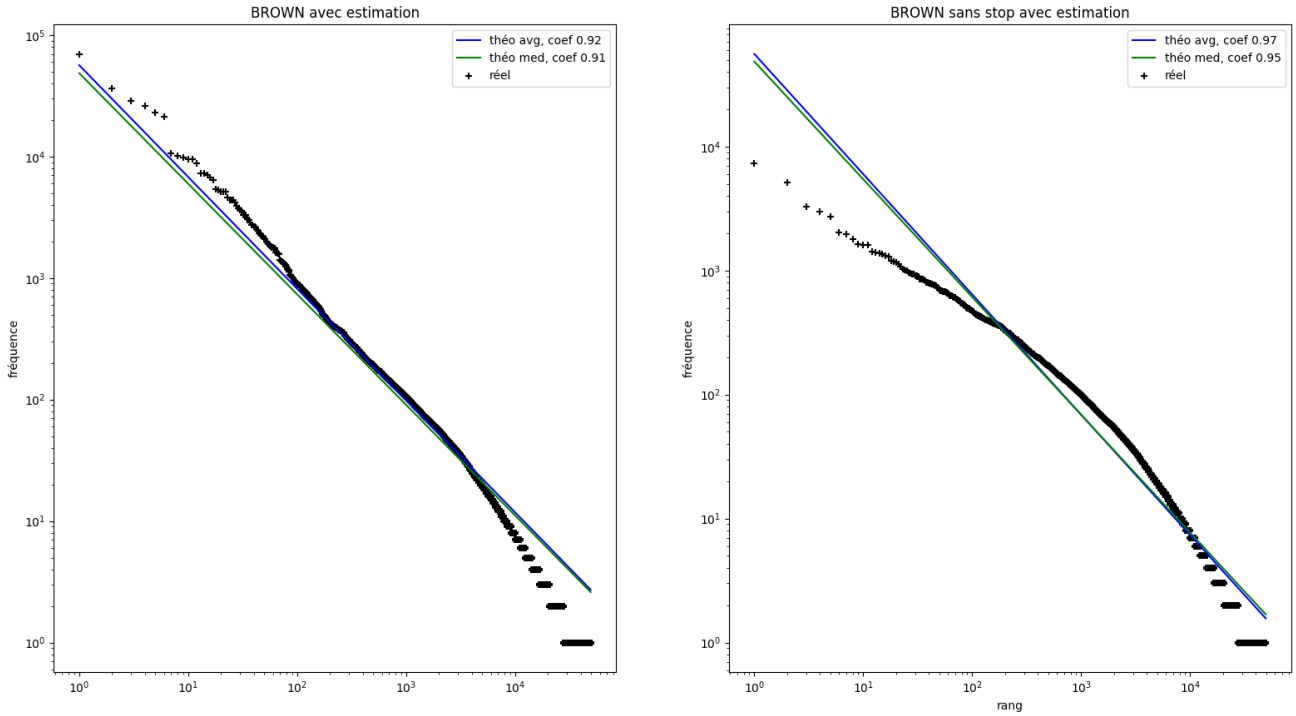

FIGURE 8 – Distribution de Zipf avec les estimations

En conclusion, j'utiliserais la moyenne des constantes sur un document complet afin de déterminer le coefficient dans ma recherche de spam.

Notes : L'ensemble des codes sources pour cette partie est disponible dans les fichiers :

- ./analyse/rech_zipf.py
- ./traitement/stats.py

B Déploiement des bases de données

B.1 ElasticSearch

B.1.1 Conteneurisation

B.1.2 Initialisation de l'index

B.2 PostgreSQL

B.2.1 Conteneurisation

B.2.2 Initialisation de la base de données

C Tableau des choix technologiques

Élément	Retenu	Raisons	Observations
Datasets			
Mail de la compagnie Enron	Non	Mails non classés	Non retenu pour la phase de développement car pas de moyen fiable de contrôler la sortie automatiquement
Mail du projet SpamAssassin	Oui	Mails déjà pré-triés	Mails principalement en Anglais déjà pré-trié en catégorie Spam et Ham
Brown dataset (nltk)	Oui	Corpus d'un million de mots en Anglais publié en 1961	Dataset utilisé pour le développement de la visualisation de la distribution de Zipf
Stopwords (nltk)	Oui	Corpus de mots commun non significatif dans un texte	Utilisation dans le développement de la visualisation de la distribution de Zipf
Langage et Modules			
Python	Oui	Langage polyvalent pour le traitement des données	
Module email	Oui	Module natif pour le traitement des mails	Grande flexibilité pour la lecture des mails
Bases de données			
ElasticSearch	Oui	Technologie utilisée dans mon entreprise. Présence d'une interface de visualisation des données Kibana.	Application dockerisée.
PostgreSQL	Oui	Moteur de base de données relationnelle plus facilement scalable que ElasticSearch pour l'ajout de nouvelle catégorie de données. Il n'est pas nécessaire de ré-indexer toute la base pour ajouter des champs	Application dockerisée
SQLite	Oui	Base de données légère pour stocker uniquement les données statistiques des étapes de la phase 1	Rapide à mettre en place et déjà intégrée

D Modèles

D.1 Naive Bayes

E Bibliographie

F Sitotec

F.1 Corpus

— Enron company mails, fichier CSV contenant l'ensemble des mails d'une entreprise ayant fermée ses portes (33.834.245 mails) [en ligne], https://www.kaggle.com/wcukierski/enron-email-dataset (consulté le 27/01/2022)

— Mails project SpamAssassin, projet opensource de détection de spam (6065 fichiers email déjà trier en ham et spam) [en ligne], https://spamassassin.apache.org/old/publiccorpus/ (consulté le 27/01/2022)

— Brown corpus, ensemble de texte en anglais publié en 1961 qui contient plus d'un million de mots https://www.nltk.org/book/ch02.html (consulté le 20/08/2022)

F.2 Modules

Module langdetect

— Page Github du projet *langdetect* capable de différencier 49 langages avec une précision de 99%, [en ligne] https://github.com/Mimino666/langdetect (consulté le 04/12/2022)

— Language Detection Library, présentation du module (anglais) [en ligne] https://www.slideshare.net/shuyo/language-detection-library-for-java (consulté le 04/12/2022)