Un modelo de análisis estilométrico de correos electrónicos para la redacción personalizada basada en el destinatario
A model for stylometric analysis of emails for recipient-based personalised writing

Trabajo de Fin de Grado
Curso 2019–2020

Autor
Carlos Moreno Morera

Directores
Raquel Hervás Ballesteros
Gonzalo Méndez Pozo

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Un modelo de análisis estilométrico de correos electrónicos para la redacción personalizada basada en el destinatario
A model for stylometric analysis of emails for recipient-based personalised writing

Trabajo de Fin de Grado en Ingeniería Informática
Departamento de Ingeniería del Software e Inteligencia Artificial

Autor
Carlos Moreno Morera

Directores
Raquel Hervás Ballesteros
Gonzalo Méndez Pozo

Convocatoria: Junio 2020

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

6 de junio de 2020

A Pedro Pablo y Marco Antonio, por crear TeXiS
e iluminar nuestro camino

Acknowledgments

A Guillermo, por el tiempo empleado en hacer estas plantillas. A Adrián, Enrique y Nacho, por sus comentarios para mejorar lo que hicimos. Y a Narciso, a quien no le ha hecho falta el Anillo Único para coordinarnos a todos.

Resumen

Un modelo de análisis estilométrico de correos electrónicos para la redacción personalizada basada en el destinatario

Un resumen en castellano de media página, incluyendo el título en castellano. A continuación, se escribirá una lista de no más de 10 palabras clave.

Palabras clave

Máximo 10 palabras clave separadas por comas

Abstract

A model for stylometric analysis of emails for recipient-based personalised writing

An abstract in English, half a page long, including the title in English. Below, a list with no more than 10 keywords.

Keywords

10 keywords max., separated by commas.

Contents

List of figures

List of tables

Chapter 1

Introduction

"Have you ever retired a human by mistake?"
— Rachael - Blade Runner (1982)

Smartphone development meant not only a technological advance but a social revolution too. This intelligent telephones have brought with them countless paradigm shifts in terms of the social sphere. Since then, we are able to speak of a new model of human relationship both between people and with our technology. This current relation standard is due to the easy and quick way of accessing the different information that our mobile devices provide us. Long waits (nowadays the meaning of "long" waits has changed too, people consider more than two or three second too much time) for obtaining anything such as accessing to a website or showing any operation result, are excessively tedious and could be even frustrating for some smartphone users. When we are using our mobile, we want, as fast as possible, the information we are looking for. Precisely because of this, Human-Computer Interaction (HCI) becomes a very important part in the process of development of most applications, not only in terms of speed of response and efficiency of algorithms, but also in how we show different information and the easiness for obtaining it.

As for the relationships between people, as we have said, they have dramatically changed. There is no doubt that the main driving technologies behind this transformation of our relational paradigm are the social networks and the instant messaging. Focusing on the latter, it is necessary to make a breakdown of what consequences to our interpersonal interaction the instant communication have brought with itself. Just as it happens with the HCI, easiness and speed are probably the first features we look for when we are going to send or receive any information to anybody. If we also expect a reply, the ideal would be to obtain it as quickly as possible. Therefore, in most of occasions, in practice we are looking for an "automatic" response from a human, what practically implies that everyone is "obligated" to be connected at any time with the answer we are asking for prepared. This new insight into the relationships between people, that perceives the humans as servers who send a request waiting for a quickly reply with the expected data, has promoted a very fast sending of short messages which intends to substitute and simulate an spoken conversation. These little texts are often concise and summarised, and they form an atomic semantic unit, namely they have their own independent meaning.

1.1. Incentive

Introducción al tema del TFM.

1.2. Objectives

Descripción de los objetivos del trabajo.

1.3. Working plan

Aquí se describe el plan de trabajo a seguir para la consecución de los objetivos descritos en el apartado anterior.

1.4. Explicaciones adicionales sobre el uso de esta plantilla

Si quieres cambiar el **estilo del título** de los capítulos, edita `TeXiS\TeXiS_pream.tex` y comenta la línea `\usepackage[Lenny]{fncychap}` para dejar el estilo básico de LaTeX.

Si no te gusta que no haya **espacios entre párrafos** y quieres dejar un pequeño espacio en blanco, no metas saltos de línea (\\) al final de los párrafos. En su lugar, busca el comando `\setlength{\parskip}{0.2ex}` en `TeXiS\TeXiS_pream.tex` y aumenta el valor de $0.2ex$ a, por ejemplo, $1ex$.

TFMTeXiS se ha elaborado a partir de la plantilla de TeXiS[1], creada por Marco Antonio y Pedro Pablo Gómez Martín para escribir su tesis doctoral. Para explicaciones más extensas y detalladas sobre cómo usar esta plantilla, recomendamos la lectura del documento `TeXiS-Manual-1.0.pdf` que acompaña a esta plantilla.

El siguiente texto se genera con el comando `\lipsum[2-20]` que viene a continuación en el fichero .tex. El único propósito es mostrar el aspecto de las páginas usando esta plantilla. Quita este comando y, si quieres, comenta o elimina el paquete *lipsum* al final de `TeXiS\TeXiS_pream.tex`

1.4.1. Texto de prueba

[1] `http://gaia.fdi.ucm.es/research/texis/`

Chapter 2

State of the Art

"Who controls the past controls the future.
Who controls the present controls the past."
— 1984 - George Orwell (1949)

2.1. Electronic Mail

Electronic Mail (Guide, 2005, Chapter 11) is a communication service which has been used since 1971 (Wikipedia contributors, 2019b) when the first network e-mail with the text "QWERTYUIOP" was sent through ARPAnet (Advanced Research Projects Agency Network, the first network which implements the TCP/IP protocol) with the experimental protocol CYPNET. Nowadays, the messages are delivered by using a client/server architecture. In this way, an e-mail is created by using a client-side mail program. Then, this software sends the message to a server, which will redirect it to the recipient's mail server. From there, the e-mail is delivered to the addressee.

In order to make all this process possible, an Internet standard that extends the format of e-mail messages, and a wide range of network protocols exist for allowing different machines (which often execute distinct operative systems and make use of different mail programs) to share e-mails. In this section, we are going to study this standard, these protocols and the API which is going to be used for reading, sending e-mails and accessing to the user's e-mail data. First of all, we are going to explain the MIME standard (see Section 2.1.1) which specifies the format of e-mail messages. Then we are going to explain the main e-mail management protocols, both electronic mail transmission protocol (such as Simple Mail Transfer Protocol, which is explained in Section 2.1.2) and message access protocol (such as Post Office Protocol and Internet Message Access Protocol, which are studied in Sections 2.1.3 and 2.1.4, respectively).

In spite of being a mail server-independent solution, as we will see, we are going to find security issues which are going to hinder our user's e-mail data access. These trials come from the automatic server access. For this reason, Gmail API is going to be introduced (see Section 2.1.5) and, finally, the assessment of the advantages and disadvantages of making use of the e-mail protocols or the Gmail API is discussed (see Section 2.1.6).

2.1.1. MIME

To be able to automatically create messages and read the body of the e-mails, it is essential to understand what the MIME standard consists in. Hence, in this section we are

going to give a general idea about this.

MIME, whose acronym stands for Multipurpose Internet Mail Extensions (Wikipedia contributors, 2019c), is an Internet standard for the exchange of several file types (text, audio, video, etc.) which provides support to text with characters other than ASCII, non-text attachments, body messages with numerous parts (known as multi-part messages) and headers information with characters other than ASCII. It is defined in a series of Request For Comments (RFC): RFC 2045 (Freed and Borenstein, 1996b), RFC 2046 (Freed and Borenstein, 1996c), RFC 2047 (Moore, 1996), RFC 2049 (Freed and Borenstein, 1996a), RFC 2077 (Nelson and Parks, 1997), RFC 4288 (Freed and Klensin, 2005a) and RFC 4289 (Freed and Klensin, 2005b).

Virtually all e-mails written by people on the Internet and a considerable proportion of these automatically generated messages are transmitted in MIME format via SMTP (see Section 2.1.2). Internet e-mail messages are so closely associated with SMTP and MIME that they are usually called SMTP/MIME messages.

The content types defined by the MIME standard are of great importance also outside the context of e-mails. Examples of this are some network protocols such as HTTP from the Web. HTTP requires data to be transmitted in an e-mail-type message context although the data may not be an e-mail itself.

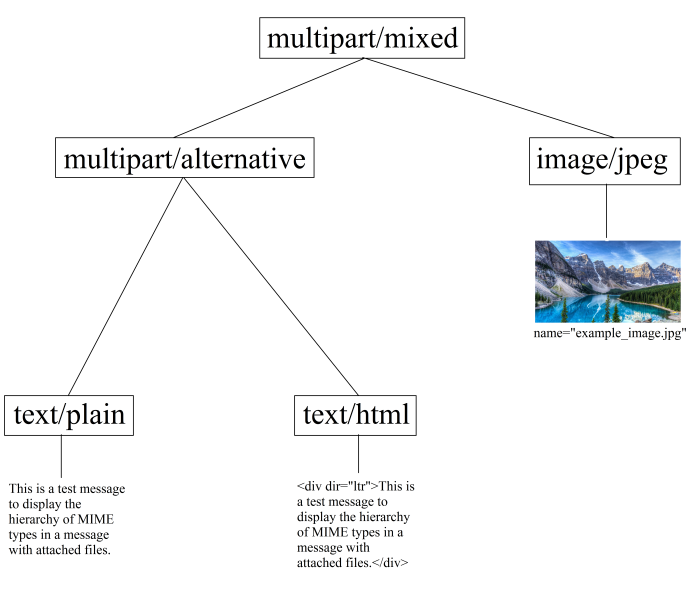
Nowadays, no e-mail program or Internet browser can be considered complete if it does not accept MIME in its various facets (text and file formats).

In this section we will learn about the MIME type nomenclature (see Section 2.1.1.1), which is necessary for being able to exchange several file types. Then, we will illustrate the MIME structure of an e-mail, consisting of MIME headers (see Section 2.1.1.2) and, finally, two common MIME message encoding (base64 and quoted-printable) are explained (see Sections 2.1.1.3 and 2.1.1.4, respectively).

2.1.1.1. Type Nomenclature

Each data type has a different name in MIME. These names follow the format: type-/subtype (both type and subtype are strings), in such a way that the first denotes the general data category and the second the specific type of that information. The values the type can take are:

- *text*: means that the content is simple text. Subtypes like *html*, *xml* and *plain* can follow this type.
- *multipart*: indicates that the message has numerous parts with independent data. Subtypes like *form-data* and *digest* can follow this type.
- *message*: it is used to encapsulate an existing message, for example when we want to reply a e-mail and add the previous message. Subtypes like *partial* and *rfc822* can follow this type.
- *image*: means that the content is an image. Subtypes like *png*, *jpeg* and *gif* can follow this type.
- *audio*: indicates that the content is an audio. Subtypes like *mp3* and *32kadpcm* can follow this type.
- *video*: denotes that the content is an video. Subtypes like *mpeg* and *avi* can follow this type.



MIME-Version: 1.0
From: Sender <sender@gmail.com>
Date: Fri, 4 Oct 2019 22:06:29 +0200
Message-ID: <CADkvQ12BgJJxjqApD6AcNcAhJJEZ5FBkPi-9Pw__XduXgN_3sQ@mail.gmail.com>
Subject: Example MIME message
To: addressee@gmail.com
Cc: addressee2@gmail.com
Content-Type: multipart/mixed; boundary="000000000000abadb805941b3c9d"

--000000000000abadb805941b3c9d
Content-Type: multipart/alternative; boundary="000000000000abadb605941b3c9b"

--000000000000abadb605941b3c9b
Content-Type: text/plain; charset="UTF-8"
Content-Transfer-Encoding: quoted-printable

This is a test message to display the hierarchy of MIME types in a message
with attached files.

--000000000000abadb605941b3c9b
Content-Type: text/html; charset="UTF-8"
Content-Transfer-Encoding: quoted-printable

This is a test message to display the hierarchy of MIME types in a message with attached files.

--000000000000abadb605941b3c9b--
--000000000000abadb805941b3c9d
Content-Type: image/jpeg; name="example_image.jpg"
Content-Disposition: attachment; filename="example_image.jpg"
Content-Transfer-Encoding: base64
Content-ID: <16d985155126855b6e01>
X-Attachment-Id: 16d985155126855b6e01

--000000000000abadb805941b3c9d--

Figure 2.1: MIME types tree structure of an e-mail example

- *application*: it is used for application data that could be binary. Subtypes like *json* and *pdf* can follow this type.
- *font*: means that the content is a file which defines a font format. Subtypes like *woff* and *ttf* can follow this type.

2.1.1.2. MIME headers

MIME has several headers which appear in all e-mails sent with this standard. The most important of them are the following:

- *Content-Type*: the value of this header is the type and subtype of the message with the same structure that we have explained before. For example, if we have the header *Content-Type: text/plain*, it means that the message is a plain text. The use of the type *multipart* makes the creation of messages with parts and subparts organized in a tree structure (in which leaf nodes can belong to any type and the rest of them can belong to any multipart subtype variety) possible (Freed and Borenstein, 1992, Section 7.2). A feasible composition of a message with a part with plain text and other non-text parts could be constructed by using *multipart/mixed* as the root node like in Figure 2.1. Indeed, in the example of Figure 2.1 we can observe the use of *multipart/alternative* for a message which contains the body both in plain text and in html text. Other different e-mails constructions are possible (like forwarding with the original message attached by using *multipart/mixed* with a *text/plain* part and a *message/rfc822* part) thanks to the tree structure of the *Content-type* header.

  Another important detail, that we can observe in the example in Figure 2.1, is the fact that each node of the tree structure of the e-mails is visited and showed following the pre-order traversal.

- *Content-Disposition*: this header is used to indicate the presentation style of the part of the message. There are two ways to show the part: *inline* content-disposition (which means that the content must be displayed at the same time as the message)

and *attachment* content-disposition (the part is not displayed at the same time as the message and it requires some form or action from the user to see it). Furthermore, this header also provides several fields for specifying other type of information about the content, such as the name of the file and the creation or modification date. The following example is taken from RFC 2183 (Troost et al., 1997) and, as we will explain after the example, it does not match with the syntax of this same header in the example that we can see in the last part of the example message of Figure 2.1:

```
Content-Disposition: attachment; filename=genome.jpeg;
modification-date="Wed, 12 Feb 1997 16:29:51 -0500";
```

As we have said, this syntax is different from the one used in the e-mail example of Figure 2.1. This results from the fact that, in HTTP, the header we find in that figure (*Content-Disposition: attachment*) is usually used for instructing the client to show the response body as a downloadable file. As we can observe, it has a *filename* field which is used for establishing the default file name when the user is going to download it.

- *Content-Transfer-Encoding*: when we want to send some files in a message, sometimes they are represented as 8-bit character or binary data, which are not allowed in some protocols. On this account, it is necessary to have a standard that indicates how we should re-encode such data into a 7-bit short-line format. The Content-Transfer-Encoding header (Freed and Borenstein, 1992, Section 5) will tell the client which transformation has been used for being able to transport that data. Therefore, and for lack of a previous standard which states a single Content-Transfer-Encoding mechanism, the possible values which specify the type of encoding are: *'base64'* (see Section 2.1.1.3), *'quoted-printable'* (see Section 2.1.1.4), *'8bit'*, *'7bit'*, *'binary'* and *'x-token'*. All these values are not case sensitive. If this header does not exist, we can assume that the value of this header is *'7bit'*, which means that the body of the message is already in a seven-bit mail-ready representation, in other words, all the body of the message is represented as short lines of US-ASCII data. Despite *'8bit'*, *'7bit'* and *'binary'* indicate that the content has not been transformed, they are useful for knowing the kind of encoding that the data has. This header will generally be omitted when the Content-Type has the *multipart* or *message* type (as it happens in the message example of Figure 2.1), because it also admits the last three types we have mentioned.

  It is common to add another header (as we can see in Figure 2.1) called *charset*, the value of which represents the original encoding of data so the client is able to decode it.

2.1.1.3. Base64 encoding

As we have studied when we learnt how the MIME headers (see Section 2.1.1.2) are, we can find e-mail whose content encoding is base64. Base64 (Wikipedia contributors, 2019a; Josefsson, 2006) is a group of reversible binary-to-text encoding schemes which represent binary data as a sequence of ASCII printable characters. It makes use of a radix-64 to translate each character, because 64 is the highest power of two than can be represented using only printable ASCII characters. Indeed all the Base64 variants (like base64url) utilise the characters range A-Z, a-z and 0-9 in that order for the first 62 digits, but the chosen symbol for the last two digits are very different between them. In particular, the MIME (see Section 2.1.1) specification, established in RFC 2045 (Freed and

Borenstein, 1996b), describes base64 based on Privacy-enhanced Electronic Mail (PEM) protocol (Wikipedia contributors, 2019d; Josefsson and Leonard, 2015), which means that the last two characters are '+' and '/', and the symbol '=' is used for output padding suffix. In the same way, MIME does not stablish a fixed size for the base64 encoded lines, by contrast it specifies a maximum size of 76 characters.

If we try to apply standard base64 in a URL encoder, it will translate the characters '+' and '/' to its hexadecimal representation ('+' = '%2B' and '/' = '%2F'). This will cause a conflict in heterogeneous systems or if we use it in data base storage, because of the character '%' produced by the encoder (it is a special symbol of ANSI SQL). This is why modified Base64 for URL variants exists (such as base64url in Josefsson (2006)), where the '=' character has no usefulness and the '+' and '/' symbols are replaced by '-' and '_' respectively. Besides it has no impact on the size of encoded lines.

2.1.1.4. Quoted-printable enconding

Other reversible binary-to-text encoding that could be used in the content of a MIME message is the quoted-printable encoding (Wikipedia contributors, 2019e; Borenstein and Freed, 1993). Making use of printable characters (such as alphanumeric and '=') proved capable of transmitting 8 bit data over a 7 bit protocol. Unlike base64, if the original message is mostly composed of ASCII characters, the encoded text is readable and compact.

Each byte can be represented via two hexadecimal characters. On this basis, the '=' symbol followed by two hexadecimal digits are enough to encode all the characters except the printable ASCII ones and the end of line. For example, if we want to represent the 12^th ASCII character we can encode it as '=0C' or if the equality symbol (whose decimal value is 61) is in our original message, it could be encoded as '=3D' (note that despite being a printable ASCII character it must be encoded as it is a special character in this encoding). This is how quoted-printable encodes the different characters.

In respect of the maximum line size, as it happens with the MIME specification of the base64 (see Section 2.1.1.3), it has a length of 76 characters each encoded line. To achieve this goal and still be able to decode the text getting the original message, quoted-printable adds *soft line breaks* at the end of the line consisting of the '=' symbol and it does not modify the encoded text.

2.1.2. Simple Mail Transfer Protocol

Simple Mail Transfer Protocol (also known as SMTP) is a network connection-oriented communication protocol used for the exchange of e-mail messages. It was originally defined by Postel (1982) (for the transfer) and by Crocker (1982) (for the message). It is currently defined by Klensin (2008) and Resnick (2008). However, this protocol has some limitations when it comes to receiving messages on the destination server. For this reason, this task is intended for other protocols such as the Internet Message Access Protocol (see Section 2.1.4) or the Post Office Protocol (refer to Section 2.1.3), and SMTP is used specifically to send messages.

Making use of SMTP, an e-mail is "pushed" from one mail server to another (next-hop mail server) until it reaches its destination. The message is not routed according to the message recipients specified during the client's connection to the SMTP server, but from the destination mail server. Thanks to the fact that this protocol has a feature to initiate mail queue processing, an intermittently connected mail server can extract messages from another remote server when necessary.

2.1.3. Post Office Protocol

Post Office Protocol (also known as POP) is an application protocol (in OSI Model) for obtaining e-mails stored in a remote Internet server called POP server. It was originally defined by Reynolds (1984) (it was POP version 1, also known as POP1). Current POP version (POP3, in general when we talk about POP we refer to this version) is detailed by Myers et al. (1996).

POP was designed for receiving e-mails. Using POP, users with intermittent or very slow Internet connections (such as modem connections) can download their e-mail while online and check it later even when offline. The general operation is: a client using POP3 connects, gets all messages, stores them on the user's computer as new messages, deletes them from the server, and finally disconnects. However some mail clients include the option to leave messages on the server. They use the order UIDL (Unique IDentification Listing) which, unlike most POP3 commands, does not identify messages depending on their mail server ordinal number. This results from the fact that the mail server ordinal number creates problems when a client tries to leave messages on the server, since messages with numbers change from one connection to the server to another. Accordingly, a server which makes use of UIDL, assigns a unique and permanent character string to each message. Thus, when a POP3-compatible mail client connects to the server, it uses the UIDL command to map the message identifier. This way the client can use that mapping to determine which messages to download and which to save at the time of downloading.

Like other old Internet protocols, POP3 used a signature mechanism without encryption. The transmission of POP3 passwords in plain text still occurs. Nowadays POP3 has various authentication methods that offer a diverse range of levels of protection against illegal access to users' mailboxes.

The advantage over other protocols is that between server-client you do not have to send so many commands for communication between them. The POP protocol also works properly if you do not use a constant connection to the Internet or to the network that contains the mail server.

2.1.4. Internet Message Access Protocol

Internet Message Access Protocol (also known as IMAP) is an application protocol, designed as an alternative to Post Office Protocol (see Section 2.1.3) in 1986, which allows the access to stored messages in an Internet server. As with the Post Office Protocol, with IMAP you can access your e-mail from any computer with an Internet connection. The current version of IMAP (IMAP version 4 review 1, or IMAP4rev1) is defined by Crispin (2003).

In contrast to Post Office Protocol, IMAP allows multiple clients to manage the same mailbox. This fact results from the main differences between these two protocols: IMAP does not remove e-mail from the server until the client specifically requests it (as POP removes them by default, it is impossible to access them from another device which has not downloaded the messages) and it does not download the messages to the user's computer (clients may optionally store a local copy of them). This last property gives raise to several advantages with regard to Post Office Protocol: the immediate notification of the arrival of a mail (due to it works in permanent connection mode) while POP checks if there are new e-mails every few minutes (which causes an appreciable rise in traffic and in the time the user has to wait to send a request to the server, because it is necessary to complete the download of all new messages first), it is possible to create shared folders with other users (it depends on the mail server), the e-mails do not take up memory in the user's local

device while POP downloads them regardless of whether they are going to be read or not (effectively IMAP has to download a message when it is going to be read, but they are temporary files and only the e-mail headers are downloaded to manage the mailbox) and it allows the user to manage folders, templates and drafts in server in addition to be able to search a mail from keywords.

2.1.5. Gmail API

Gmail is a free e-mail service developed by the company Google. Users can access Gmail on the web itself and through third-party programs that synchronize e-mail content via POP or IMAP protocols. It also has a mobile application to manage the user's e-mail. Gmail began as a limited beta version on April 1, 2004 and completed its testing phase on July 7, 2009. As stated by BBC news (3rd July 2018): "Gmail is the world's most popular e-mail service with 1.4 billion users".

As we will see in Section 2.1.6, due to the automatic server access, directly using the communication protocol for electronic mail transmission (SMTP) and for retrieving e-mail messages from a mail server (POP or IMAP) will cause us security problems in accessing the user's e-mail data. For this reason, we are going to make use of Gmail API, that we will study in this Section. Thus, in Section 2.1.5.1 we are going to study the necessary protocol for accessing the Gmail API and consequently for being able to get into the user's e-mail data. Further on, we will require a resource (like a programming object) we can work with and represent all the Gmail structure (see Section 2.1.5.2). Once we count on this general resource, we have the necessary tools to be able to understand and handle the internal architecture of the Gmail API and the different means it provides in order to achieve our goal. Therefore, in Sections 2.1.5.3 to 2.1.5.6 we are going to delve into the essential resources for our purpose: labels, messages, threads and drafts.

Finally, as this API is not the only means of accessing the user's mail data (we have studied other ways in previous sections), we will end with a brief description about the API usage limits (in Section 2.1.5.7) to assess its use with respect to other methods of e-mail access.

2.1.5.1. OAuth 2.0 Protocol

Open Authorization or OAuth (Cook and Messina, 2019a) is an open standard which allows simple authorization flows for web services or applications. It is a protocol defined in Hardt (2012) which allows the site's users to share their information with another site without providing their full identity. This mechanism is used by companies like Google, Facebook, Twitter and Microsoft to allow users to share information about their accounts with third-party applications or websites.

Gmail API, as it also happens in the case of other Google APIs, uses OAuth 2.0 protocol (Google, 2019f) to handle authentication and authorization. It will provide us a secure and trusted login system to access the user's Gmail data.

The basic working process of OAuth 2.0 protocol can bee seen in Figure 2.2. As we can observe, at first our application carries out a request in which it sends a token. This token includes, among other things, a credential, which helps Google Servers to identify the application, and a list of OAuth 2.0 Scopes (Google, 2019e), which are a "mechanism in OAuth 2.0 to limit an application's access to a user's account. An application can request one or more scopes. This information is then presented to the user in a consent screen, and the access token issued to the application will be limited to the scopes granted" (Cook and

Messina, 2019c). We will use the Gmail API OAuth 2.0 Scope which allows us to read, compose, send, and delete e-mails.

Once the user has logged in the Gmail account (authentication) and accepted all the necessary permissions that our application needs (authorization), our process receives an authorization code which is going to be exchanged for an access token (Cook and Messina, 2019b). Then, we will be in possession of the OAuth 2.0 credentials for the user (Google, 2019d) which we are going to use for accessing the user's Gmail account.

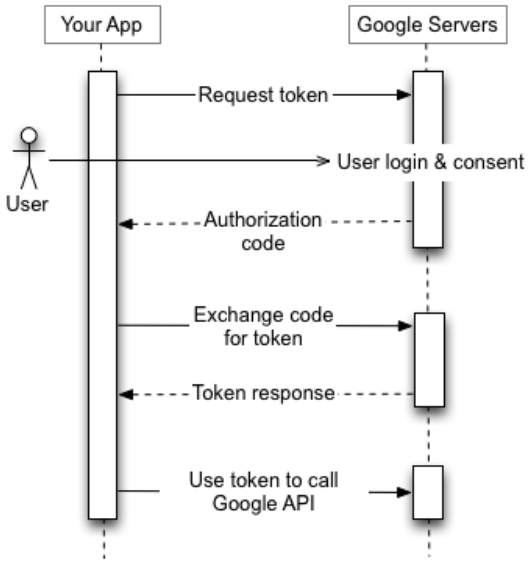


Figure 2.2: OAuth 2.0 for Web Server Applications and Installed Applications. Image extracted from Google (2019f)

2.1.5.2. Users resource

At this point, with the OAuth 2.0 credentials, we are able to call the Gmail API. For this purpose, it is necessary to construct a resource (Google, 2019a, /v1/reference) for interacting with the API. As we will see later, this resource will lead us to manage e-mails, drafts, threads and everything we will like to do with the user's Gmail data.

By using the OAuth 2.0 credentials, we are able to get in contact with the Google Servers and request what is known as users resource (Google, 2019a, /v1/reference/users), which holds all the necessary resources for our task, such as labels (see Section 2.1.5.3), messages (see Section 2.1.5.4), threads (see Section 2.1.5.5) and drafts (see Section 2.1.5.6). In practice, the users resource has instance methods which get in contact with Google Servers and return these other Gmail API resources that we are going to need (the methods' names are *labels()*, *messages()*, *threads()* and *drafts()*, respectively). Now, in next sections, we are going to explain all the resources we can create with the user resource.

2.1.5.3. Labels resource

As we have seen in the explanation of the users resource (Section 2.1.5.2), we can obtain the labels resource (Google, 2019a, /v1/reference/users/labels) by invoking *labels()* instance method of our users resource. It manages the entire set of our e-mail labels, which categorize messages and threads within the user's mailbox.

Labels resource is an object which allows us to access to the different e-mail labels of the user, such as *INBOX*, *UNREAD* and *SENT*. With the labels resource methods, we can obtain each of these "user's labels" which have a dictionary structure and their representation is what we can observe hereunder:

```
{
    'id' : string, # The immutable identifier of the label
    'name' : string, # The display name
    # The visibility of messages in the Gmail web interface
    'messageListVisibility' : string,
    'labelListVisibility' : string, # The visibility of label
    # The owner type of the label ('system' or 'user')
    'type' : string,
    # Total number of messages with the label
    'messagesTotal' : integer,
    # Number of unread messages with the label
    'messagesUnread' : integer,
    # Total number of threads with the label
    'threadsTotal' : integer,
    # Number of unread threads with the label
    'threadsUnread' : integer,
    'color' : {
       # Text color of the label, represented as hex string
       'textColor' : string,
       # Background color represented as hex string #RRGGBB
       'backgroundColor' : string
    }
}
```

The important fields we are going to need are the *name*, the *type* and the number of total messages and threads with the label (which are *messagesTotal* and *threadsTotal* fields, respectively). Labels with *system* type, such as *INBOX*, *SENT*, *DRAFTS* and *UNREAD*, are internally created and cannot be added, modified or deleted.

2.1.5.4. Messages resource

In most of the operations we are going to execute, the correct management of messages will be essential. Therefore, knowing how the e-mails are represented in Gmail API and how to use them is imperative to understand how to work with this API. For this reason, in this section we are going to delve into the messages resource (Google, 2019a, /v1/reference/users/messages) of the Gmail API. As we saw in Section 2.1.5.2, we can access to this resource by invoking the *messages()* instance method when we have a users resource.

As with the labels resource, the messages resource manages the set of all messages of the user's e-mail. With the messages resource methods, we can obtain each of these "user's messages" which, regardless of which programming language is used, have a dictionary structure and their representation is what we can see down below:

```
{
    'id' : string,
    'threadId' : string,
    'labelIds' : [ string ],
    'snippet' : string,
    'historyId' : unsigned long,
    'internalDate' : long,
```

```
    'payload' : {
        'partId' : string,
        'mimeType' : string,
        'filename' : string,
        'headers' : [
        {
            'name' : string,
            'value' : string
        }
        ],
        'body' : {
            'attachmentId' : string,
            'size' : integer,
            'data' : bytes
        },
        'parts' : [ (MessagePart) ]
    },
    'sizeEstimate' : integer,
    'raw' : bytes
}
```

The more important keys of this data structure for this work are:

- *id*: an immutable string which identifies the message.
- *threadId*: we will explain the thread resource in Section 2.1.5.5 and we will see that a thread is composed of different messages that share common characteristics. The value of this field is a string which represent the identifier of the thread the message belongs to.
- *labelIds*: a list of the identifiers of labels (see Section 2.1.5.3) applied to the message.
- *payload*: as we can see in the resource representation above, it has a dictionary data structure. The *payload* field is the parsed e-mail structure in the message parts. The more important keys of the *payload* field are:
  - *mimeType*: the MIME type (see the explanation of *Content-Type* header in Section 2.1.1.2) of the message part.
  - *headers*: a list of headers. It contains the standard RFC 2822 (Resnick, 2001) e-mail headers such as *To*, *From*, *Subject* and *Date*. Each header has a *name* field, which is the name of the header (for example *From*), and a *value* field, which is the value of the header (following the same example as with the *name* field, *example@gmail.com* could be the value).
  - *parts*: a list which contains the different MIME message child parts (we have gone into it in depth in the Section 2.1.1).
  - *body*: a dictionary structure which contains the body data of this part (see Section 2.1.1) in case it does not contain MIME message parts (otherwise it will be empty). This structure should not be confused with an attached file. Each MIME part contains a body property regardless of MIME type of the part.
- *raw*: the entire e-mail message in an RFC 2822 (Resnick, 2001) formatted and base64url (see Section 2.1.1.3) encoded string.

2.1.5.5. Threads resource

When we access to our inbox, we are actually seeing the inbox threads instead of the messages resource. Every message, even if it is an only e-mail without a reply, is enclosed in a thread resource (Google, 2019a, /v1/reference/users/threads) which is essentially a list, perhaps unitary, of messages resources. In fact, as we can observe in the following resource representation, each thread (which can be obtained thanks to the threads resource due to it manages the entire set of threads of a user's e-mail), in its dictionary structure, has a list of messages resources:

```
{
    'id' : string, # The identifier of the thread
    'snippet' : string, # A short part of the text
    'historyId' : unsigned long,
    'messages' : [ users.messages resource ]
}
```

2.1.5.6. Drafts resource

The last Gmail API resource we will study is the most easy to understand after knowing all the structures related with e-mails that we have explained in the above sections: the drafts resource (Google, 2019a, /v1/reference/users/drafts). Its representation is very simple:

```
{
    'id' : string # The immutable identifier of the draft
    'message' : users.messages resource
}
```

As we can observe, a draft is virtually a messages resource with an identifier. Indeed, in order to create a new draft with the *DRAFT* label we must create a MIME message (see Section 2.1.1) as we have to do when we want to send a new e-mail by using the *send* messages resource method.

2.1.5.7. API Usage Limits

One factor to be taken into account is the limitations of the Gmail API (Google, 2019a, /v1/reference/quota) which could become a drawback in the application development. It has a limit on the daily usage and on the per-user rate. In order to measure the usage rate, "quota units" are defined depending on the method invoked (main methods of each resource are explained in Section 3.1). In Table 2.1 we can consult the value of some methods in quota units (we have selected the most important methods for our purpose, for the quota units of other methods it is recommended to refer to (Google, 2019a, /v1/reference/quota)).

However, both daily usage limit and per-user rate limit are acceptable for the type of software we want to build: 1,000,000,000 quota units per day and 250 quota units per user per second. Therefore there are no constraints (for our purpose) that avoid us to use this API.

2.1.6. Advantages and disadvantages of e-mail protocols versus the use of Gmail API

Without using the Gmail API, we may be able to access mail accounts by implementing the different e-mail protocols that we have studied. Indeed, this implementation would

Method	Where the method is explained	Quota units
getProfile	3.1.3	1
labels.get	3.1.4	1
messages.get	3.1.5	5
messages.list	3.1.5	5
messages.send	3.1.5	100
threads.get	3.1.6	10
threads.list	3.1.6	10
drafts.create	(Google, 2019a, /v1/reference/users/drafts)	10

Table 2.1: Main methods' quota units

allow us to access them regardless of the mail server. In other words, we would be able to work with any e-mail account without the need of being a Gmail one. However, when we try to develop an application which is going to access to a user's e-mail account, Google Servers detect it as a non-authorised login and block the authentication process. Then they send to the user a warning titled "A login attempt has been blocked" with the following information: "*Someone just used your password to try to sign in to your account from a non-Google application. Although Google has blocked access, you should find out what happened. Check your account activity and make sure that only you have access to your account*".

Against this background, it is possible to change the user's security settings for allowing the automatic accessing to the account. However, it is not recommended (due to possible security issues) and creates a sense of insecurity for the user of the application that requires this configuration.

On the other hand we have the Gmail API, which facilitates the access to e-mail's data. Besides, its only disadvantage is to limit the daily usage of this technology by imposing quota units. However, this quota units are enough for achieving our aim. For these reasons, and because of the e-mail accounts that we will study belongs to Gmail, the Gmail API has been chosen as the most suitable way for managing the user's e-mail account.

2.2. Computational stylometry

This field of Artificial Intelligence (related with Natural Language Processing and Natural Language Generation) is in charge of studying the writing style in natural language written documents (although it is often used in applications like the detection of plagiarism in programmes). In this section we are going to delve into it in order to know the state of art of this field of study. To achieve this, first a brief introduction is presented (see Section 2.2.1), and then the different applications and techniques used in Computational stylometry are explained (see Section 2.2.2). In addition, it will be necessary to explain the presentation of computational stylometry in the specific field of e-mails (see Section 2.2.3) since, as we can deduce, they present singularities with respect to other types of documents. Finally, various style writing metrics are going to be explained (see Section 2.2.4) for the purpose of calculating and studying them in the extracted dataset (the entire set of e-mails that have been extracted).

2.2.1. Introduction

Stylometry (Wikipedia contributors, 2020b) is the application of the study of linguistic style to written language, although it has also been successfully applied to music and painting. It could be defined as the linguistic discipline that applies statistical analysis to literature in order to evaluate the author's style through various quantitative criteria.

Stylometry is characterized by the assumption that there are implicit features in the texts that the author introduces unconsciously, such as the use of a specific vocabulary that makes up the writer's mental lexicon, the lexical-syntactic structure of the sentences in the document, etc (Burrows, 1992).

According to Holmes (1998), stylometry was born in 1851 when Augustus de Morgan, an English logician, hypothesized that the problem of authorship could be addressed by determining whether one text "does not deal in longer words" (De Morgan and De Morgan, 1882) than another. Following this idea, three decades later, the American physicist Thomas Mendenhall carried out research in which he measured the length of several hundred thousand words from the works of Bacon, Marlowe and Shakespeare (Mendenhall, 1887). However, its results showed that word length is not an effective writing style feature which allows us to discriminate between different authors. Since then, numerous investigations have been carried out to analyse the parameters that define writing style more precisely.

Tweedie et al. (1996) define the writing style as "a set of measurable patterns which may be unique to an author". For this reason, various machine learning and statistical techniques have been used to discover the characteristics that determine it. One of the first and most famous successes was the resolution of the controversial authorship of twelve of the Federalist Papers. These documents, a total of eighty-five papers, were published anonymously in 1787 to convince the citizens of New York State to ratify the constitution. They are known to have been written by Alexander Hamilton, John Jay and James Madison, who subsequently claimed their contributions from each of them. However, twelve were claimed by both Madison and Hamilton. By using the frequency of occurrence of function words, previously used by Ellegard (1962), and employing numerical probabilities adjusted by Bayes' theorem, Mosteller and Wallace (1964) attribute the twelve papers disputed to James Madison. Thereafter, Federalist Papers is a famous example in this area for testing the different solutions, for example Tweedie et al. (1996) make use of neural networks to solve this problem.

2.2.2. Applications and techniques

In addition to the detection and verification of authorship in historical, literary and even forensic investigations, stylometry is used in other areas such as the detection of fraud and plagiarism, the classification of documents according to their genre or audience, etc. Other possible applications of this area are the prediction of the gender, age or personality of the author as Schwartz et al. (2013) studied; inference of the date of composition of texts, which is known as "stylochronometry" (Stamou, 2007; Juola, 2007); and even natural language generation with style (Gatt and Krahmer, 2018, Section 5.1).

To address all these problems, statistical techniques are mostly used. Some of them belong to the field of machine learning such as neural networks (Ng et al., 1997), Support Vector Machines (Abbasi and Chen, 2005), Principal Components Analysis (Binongo and Smith, 1999), decision trees (Apte et al., 1998), Adaboost (Cheng et al., 2011), K-Nearest Neighbors (Kucukyilmaz et al., 2008) and Naive Bayes (Sahami et al., 1998). Others are based on purely statistical approaches (such as cusum in Summers (1999) or Thisted

and Efron (1987)) or merely syntactic-statistical concepts as in the well-known software implementations such as stylo (Eder et al., 2016) and STYLENE (Daelemans et al., 2017). To this last type also belong techniques based on dictionary word counting using Linguistic Inquiry and Word Count also known as LIWC (Pennebaker et al., 2015), while more recent ones which use simple lexico-syntactic patterns, such as n-grams and part-of-speech (POS) tags (Mihalcea and Strapparava, 2009; Ott et al., 2011), belong to the machine learning approach. We can also find techniques outside this paradigm, such as the writing style features driven from Context Free Grammar (CFG), as we can observe in the research of Feng et al. (2012), genetic algorithms (Holmes and Forsyth, 1995) and Markov chains (Tweedie and Baayen, 1998).

In order to address our work, we are going to make use of writing style metrics (which will be explained in Section 2.2.4) based on simple statistics like the mean and easy probabilistic metrics like the entropy.

2.2.3. Style in e-mails

Electronic mails are a very specific type of document in stylometry. Their length, usually quite short, and the level of reliability, in most occasions between the informality of spoken word and the relative formality of an official letter, are two of their characteristic that make them so peculiar. For this reason, a lot of researchers have focused their attention on these type of texts, taking special interest the identification pertaining to the authorship of e-mail messages such as the published thesis by Corney (2003) or Thomson and Murachver (2001), which have investigated the existence of gender-preferential language styles in e-mail communications.

Despite being able to use most of the techniques mentioned above, both the machine learning (such as K-Nearest Neighbors used by Calix et al. (2008) or Support Vector Machines used by De Vel et al. (2001)) and the purely statistical approaches (such as regression algorithms used by Iqbal et al. (2010) for analysing 292 different features in order to verify the e-mail authorship), it is possible to find big differences with other documents such as structural features that pure text lacks. The usage of greeting text, farewell text and the inclusion of a signature are three examples of these structural features that we must take into account.

Due to that e-mail documents have several features which distinguish them from longer formal text documents (such as literary works or published articles), they make any computational stylometry problem challenging compared with others. First of all, as we have previously said, the length of the e-mails is much shorter than other documents, which results in certain language-based metrics not being appropriate (such as *hapax legomena* or *hapax dislegomena*, that is to say, the number or ratio of words used once or twice, respectively). This e-mail's feature also makes contents profiling based on traditional text document analysis techniques, such as the "bag-of-words" representation (for example when Naive Bayes approach is being used) more difficult.

Other electronic mail's particularity is the composition style used in formulating them. That is, an author profile derived from normal text documents (for example published articles) can not be the same as that obtained from a common e-mail document (De Vel et al., 2001). For example, the briefness of the e-mails causes a greater tendency to get to the point without excessive detours on the subject, in other words, they have a concise nature. We may also find that they contain a greater number of grammatical errors or even a quick compositional style that is more similar to an oral interaction, as these can become a dialogue between two or more interlocutors. In this way, the authoring composition

style and interactivity features attributed to electronic mails shares some elements of both formal writing and speech.

The main feature of e-mail against other types of documents that we are interested in is the variation in the individual style of e-mail messages due to the fact that they, as an informal and fast-paced medium, exhibit variations in an individual's writing styles due to the adaptation to distinct contexts or correspondents (Argamon et al., 2003). Many authors such as Allen (1974) and De Vel et al. (2001) support the hypothesis that each writer has certain unconscious habits when writing an e-mail that depend on the target audience. However, we hardly find any research that uses stylometry to set the parameters of writing style according to the recipient of the message.

2.2.4. Style metrics

According to Rudman (1997), at least a thousand stylistic features have been proposed in stylometric research. However, there is no agreement among researchers regarding which "style markers" yield the best results. Chen et al. (2011) (150 stylistic features were extracted from e-mail messages for authorship verification), Gruner and Naven (2005) (sixty-two stylometric measurements applied to pairs of text were calculated and then analysed in order to detect plagiarism in text documents) and Canales et al. (2011) (82 stylistic features extracted from sample exam documents were analysed using a K-Nearest Neighbours classifier for the purpose of authenticating online test takers) are only three examples of a large list of researches which look for appropriate writing style metrics to carry out their work.

As Brocardo et al. (2013) indicate, analysing a huge number of features does not necessarily provide the best results, as some features provide very little or no predictive information. And, as Brocardo et al. (2013) do, our approach is to build on previous works by identifying and keeping only the most discriminating features.

According to Abbasi and Chen (2008) existing stylistic features can be categorised as lexical (word, or character-based statistical measures of lexical variation), syntactic (including function words, punctuation and part-of-speech tag n-grams), structural (especially useful for online text, include attributes relating to text organization and layout), content-specific (are comprised of important keywords and phrases on certain topics) and idiosyncratic (include misspellings, grammatical mistakes, and other usage anomalies) style markers. However, this is not the only existing classification. There are many others like the one proposed by Corney et al. (2001), in which we see how features are divided as character-based, word-based, document-based, function word frequency distribution and word length frequency distribution; or the one proposed by Feng et al. (2012) which use a more simple classification of features in words, shallow syntax and deep syntax. However, hereafter, we are going to use the classification explained in Abbasi and Chen (2008) for referring to the different metrics.

In a vast majority of approaches, stylometrists rely on high-frequency items. Such features are typically extracted in level of (groups of) words, characters or part of speech, called n-grams (Kjell et al., 1994). Whereas token level features have longer tradition in the field, character n-grams have been borrowed from the field of language identification in Computer Science (Stamatatos, 2009; Eder, 2011). However, the most reliably successful features have been function words (short structure-determining words: common adverbs, auxiliary verbs, conjunctions, determiners, numbers, prepositions and pronouns) and word or part of speech n-grams.

A number of successful experiments with function words have been reported, such as

Craig (1999), Koppel et al. (2006) and De Vel et al. (2001). N-grams (word or part of speech ones) to some extent overlap with function words, since frequent short words count higher, but their frequencies also take into account some punctuation and other structural properties of the text. Besides, due to n-gram features are noise tolerant and effective, and e-mails are non-structured documents, many researches working with this specific type of texts, as Brocardo et al. (2013) and Corney et al. (2001), have used them.

Most reports, such as the previously mentioned composed by Kjell et al. (1994) and Corney et al. (2001), indicate that 2 or 3-grams gave good categorisation results for different text chunk sizes but these results were thought to be due to an inherent bias of some n-grams towards content rather than style alone. The effectiveness of n-grams comes from the fact that they are a successful summary marker, which can replace other markers. It is able to capture characteristics about the author's favourite vocabulary, known as word n-grams (Diederich et al., 2003) and are a content-specific feature, as well as sentence structure, known as part of speech n-grams (Baayen et al., 1996; Argamon et al., 1998), which are a syntactic feature. The problem can be found with a small corpus, since, as Baayen et al. (2000) suggest, even successful style markers may not be representative for differentiating gender, theme, author, etc. in these cases.

Another metric based on the frequency of the items is the Probabilistic Context Free Grammar (PCFG) which is used by Feng et al. (2012) in order to detect deception.

All the techniques for setting the parameters of writing style presented so far in this section have a higher level of complexity than others. This may be due to a high level of memory required during calculations (as is the case with n-grams) or a higher algorithmic complexity (as in the case of PCFG). We can also find other simple popular metrics used in other research. A good example is the lexical feature of Burrow's Delta (Burrows, 2002), which is an intuitive distance metric which has attracted a good share of attention in the community, also from a theoretical point of view (Argamon, 2008; Hoover, 2004b,a). Another example is the lexical feature of type-token ratio, which is given by the formula $R = V/N$, where N is the number of units (word occurrences) which form the sample text (tokens) and V is the number of lexical units which form the vocabulary in the sample (types). The behaviour of this style marker was studied in Kjetsaa (1979) and an approximation to Normal distribution of types per 500 tokens in all text analysed was found. Certainly it would seem that the type-token ratio would only be useful in comparative investigations where the value of N is fixed.

In order to study the sentence structure, as part of speech n-grams do, there are many other style markers such as the syntactic feature given by calculating the percentage of part of speech (POS) tags (which have been used in many previous studies in stylometry, such as Argamon-Engelson et al. (1998), Zhao and Zobel (2007), Ott et al. (2011) and Feng et al. (2012)) and the proportion of stop words in a text proposed in Ril Gil et al. (2014). One possible approach consists in the style features which take into account the part of speech tags. The verb-adjective ratio and the article-pronoun ratio belong to this category. The first was proposed in Antosch (1969) and significant results were obtained by showing that this measure is dependent on the theme of the work. For example, folk tales have higher values and scientific works have lower values. The second was studied by Brainerd (1974), where there is evidence of a connection between the number of articles and the number of pronouns in a text.

It is also possible to extract conclusions about the sentence structure through the punctuation features (which belong to the syntactic metrics category), as Baayen et al. (2002) studied. One possibility of metrics is to calculate the amount of commas, periods, semi-colons, ellipsis and brackets as Calix et al. (2008) did.

As we have studied in Section 2.1.1.2, some e-mails use HTML formatting. With this information, De Vel et al. (2001) includes the set of HTML tags as a structural metrics and studies the frequency distribution of them as one of their 21 structural attributes. These also include the number of attachments, position of requoted text within e-mail body, usage of greeting and/or farewell acknowledgement and the inclusion of a signature text. Other structural attributes, including technical features such as the use of various file extensions, fonts, sizes, and colours, have been used in works such as Abbasi and Chen (2005). This is another possibility for studying the sentence structure with an structural feature approach.

In addition to the structural features, De Vel et al. (2001) study other lexical-syntactic metrics based on the amount of blank lines, the total number of lines, count of hapax legomena, the total number of alphabetic, upper-case and digit characters in words and the number of space, white-space and tab spaces in the text.

As for the lexical-syntactic characteristics, we can also mention those defined in Calix et al. (2008), some of which are related to punctuation (such as based on the amount of dollar signs, ampersands, number signs, percent signs, apostrophes, asterisks, dashes, forward slashes, colons, pipe signs, mathematical signs, question and exclamation marks, at signs, backward slashes, caret signs, underscores, vertical lines, etc.), to sentence and paragraph (such as the number of sentences beginning with upper or lower case and the average number of words per paragraph) and to words (such as number of times "well" and "anyhow" appear). Other researchers such as Corney et al. (2001) (184 stylometric measurements were calculated and analysed by using a Support Vector Machine learning method in order to identify the authorship of electronic mails) make use of letter frequencies, distribution of syllables per word, hapax dislegomena, word collocations, preferred word positions, prepositional phrase structure and phrasal composition grammar. As regards frequency distributions of syllables per word, Fucks and Lauter (1965) discovered that it discriminated different languages more than specific authors. However, in Brainerd (1974), it is claimed that a model based on a translated negative binomial distribution was a better fit to such distributions than Fucks and Lauter (1965) translated Poisson distribution. Lastly, Brainerd (1974) concludes that some authors styles are more homogeneous than others with regard to syllable count and it would appear that the distribution of syllables per word in a corpus, being an easily accessible index of its style, is one area that may prove profitable in stylometry studies.

The most famous and ancient (as we have seen in Section 2.2.1) lexical feature is the word length (it is also applied to each part of speech as it is explained by Allen (1974)). However, as Smith (1983) concludes: "Mendenhall's method now appears to be so unreliable that any serious student of authorship should discard it". Besides, it is too strongly influenced by the language used or the subject matter dealt with and, furthermore, cannot always admit enough variance to be significant. A better way to measure style based on this criterion is to construct a graph to show what percentage of words in the text have one letter, two letters, three, and so on up to the length of the longest word; but the influence of the language itself on such measurements cannot be denied (Williams, 1970).

A variation of the word length is the sentence length. It was proposed in Yule (1939) and its major advantage is that there is a much wider range of words per sentence than letters per word. However, the major disadvantage is that it can be easily controlled by an author and it requires more text than is needed for measuring average word lengths.

Other very popular lexical features are those which measure the diversity of a text (such as the Simpson's Index, presented in Simpson (1949), or entropy, used in Holmes (1985)), the richness of its vocabulary (such as the Yule's Characteristic, defined in Yule (2014),

and the definition of richness proposed by Honoré (1979)) and the level of difficulty, such as the proposed in Dale and Chall (1948), the Gunning Fog Index (Wikipedia contributors, 2020a) or the Flesch-Kincaid index (DuBay, 2004).

As for the content-specific features, the most popular metric, a part from the word n-gram, is known as the "bag of words", which consists of storing how many times each word appears. Previous work, such as Mihalcea and Strapparava (2009) and Ott et al. (2011), has shown that "bag of words" are effective in detecting features in different documents. As Allen (1974) claims: "each writer tends to keep relatively constant the distribution of high frequency determiners, such as articles and conjunctions, whose information content is small compared to that of nouns and verbs. The other end of a frequency list is also of use in that sometimes a distinguishing stylistic feature is the avoidance of certain words".

Finally, in respect of idiosyncratic features, they include misspellings, grammatical mistakes, and other usage anomalies (Abbasi and Chen, 2008). Such features are extracted using spelling and grammar checking tools and dictionaries (Chaski, 2001). Idiosyncrasies may also reflect deliberate author choices or cultural differences, such as use of the word "center" versus "centre" (Koppel and Schler, 2003). Besides, we can add the study of features which determine the level of formality of the text, as it happens in Sheika and Inkpen (2012).

Chapter 3

Used technologies

3.1. How to work with Gmail API

In order to be able to read and send emails, it is necessary to access to the user's email data. For this reason, the different ways to obtain this information were studied. One of them is the Gmail API, which allows developers to perform all the actions we need in an easy way.

Gmail API can be used in several programming languages such as Python, PHP, Go, Java, .NET, ... Due to the greater number of examples in the starting guides of the Gmail API (Google, 2019a) and the previous knowledge that was already had of it, Python (version 3.7) was chosen for the first contact with this technology.

The following tries to be a step-by-step explanation of what is necessary to know to access the user's Gmail account, create a message, send an email previously created, create and update a draft, reply a received message (for this it is necessary to know how to create an email) and read important information of message threads and individual emails (such as who is the sender, who has received the message, the subject, the date, the email's body, the attached files, ...). Different methods of Gmail API resources (explained in Section 2.1.5) are studied to achieve this aim.

As we have seen in Section 2.1.5.1, in order to work with Gmail API, it is necessary to obtain the required OAuth 2.0 credentials. For this reason, we are going to developed an implementation which gets them (see Section 3.1.1). Then, with that credentials, we are going to build a Gmail resource (see section 3.1.2), which is necessary for obtaining the rest of the resources that we have explained. Finally, in the rest of this section, we are going to delve into the methods of each resource that we already know.

3.1.1. How to obtain OAuth 2.0 credentials

As we have seen before (see Figure 2.2), to be in possession of OAuth 2.0 client credentials from the Google API Console is required for having the appropriate permissions to use the Gmail API (this credential is the first request token that is sent to the Google Servers in the OAuth 2.0 exchange of information).

The Google API Console, also known as Google Console Developer[1], built into Google Cloud Platform, makes possible an authorized access to a user's Gmail data. In order to achieve it, having a Google account is a prerequisite because accessing to this platform

[1] `https://console.developers.google.com/`

will be necessary. Once this web has been accessed, at first we have to create a new development project by clicking in "New Project" in the control panel (which is the main tab of the Google Console Developer and the one that opens by default when you access it). When we have already created a project, we will enable the API we are going to work with, in this case the Gmail API. To do this we will look for it in the search engine that we can find in the library of APIs of this platform. Now we can apply for the credentials we need. Accessing to the "Credentials" tab and clicking on "Create Credentials" will lead us to an easy questionnaire, about what type of credentials we prefer, that we have to answer by basing on what type of application we are building. Then we must download the .json file and save it in the folder we are going to work in.

Before starting the development of the implementation of the OAuth 2.0 protocol which will provide us a secure and trusted login system to access to the user's Gmail data, we must install the Google Client Library[2] of our choice of language (we will use Python, so we have to install the libraries *google-api-python-client*, *google-auth-httplib2* and *google-auth-oauthlib*).

There are many ways to obtain the necessary permissions for accessing to the user's emails data following the OAuth 2.0 protocol. As this is a first contact with the Gmail API, only with the intention of knowing the possibilities it offers to us and its advantages and disadvantages for future implementations, we are going to develop a simple script which is using a class very useful for local development and applications that are installed on a desktop operating system. The class *InstalledAppFlow*, in *google_auth_oauthlib.flow* (Google, 2019b), is a *Flow* subclass (which belongs to the same library). Thanks to this last class we have mentioned, *InstalledAppFlow* uses a *requests_oauthlib.OAuth2Session* instance at *oauth2session* to perform all of the OAuth 2.0 logic. Besides it also inherits from *Flow* the class method *from_client_secrets_file* which creates a *Flow* instance from a Google client secrets file (this file will be the .json file that we obtained through the Google API Console) and a list of OAuth 2.0 Scopes (Cook and Messina, 2019c).

After constructing an *InstalledAppFlow* by calling *from_client_secrets_file* as we have explained, we can invoke the class method *run_local_server* which instructs the user to open the authorization URL in the browser and will try to automatically open it. This function will start a local web server to listen for the authorization response. Once there is a reply, the authorization server will redirect the user's browser to the local web server. As we can see in Figure 2.2, the web server will get the authorization code from the response and shutdown, that code is then exchanged for a token.

In summary, it is possible to obtain the necessary permissions from the user and to follow the OAuth 2.0 protocol, by executing these instructions (written in Python):

```
from google_auth_oauthlib.flow import InstalledAppFlow

# Create a flow instance
flow = InstalledAppFlow.from_client_secrets_file('credentials.json',
['https://mail.google.com/'])
# Obtain OAuth 2.0 credentials for the user
creds = flow.run_local_server(port = 0)
```

Now, we are able to call Gmail API by using the token (which is stored in the variable *creds*). However, before starting to work on the email data, we should save the OAuth 2.0 credentials since otherwise the user would need to go through the consent screen every time the application is opened. To prevent this from happening, to differentiate access from mail management and consequently to reuse as much code as possible, we have implemented

[2] `https://developers.google.com/gmail/api/downloads`

the following class *auth*, in *auth.py*, with a main method *get_credentials*:

```
1   import pickle
    import os.path
    from google_auth_oauthlib.flow import InstalledAppFlow
    from google.auth.transport.requests import Request
5
    class auth:
    def __init__(self, SCOPES, CLIENT_SECRET_FILE):
    self.SCOPES = SCOPES
    self.CLIENT_SECRET_FILE = CLIENT_SECRET_FILE
10
    def get_credentials(self):
    """
    Obtains valid credentials for accessing Gmail API
    """
15  creds = None
    # The file token.pickle stores the user's access and refresh tokens
    if os.path.exists('token.pickle'):
    with open('token.pickle', 'rb') as token:
    creds = pickle.load(token)
20  # If there are no (valid) credentials available, let the user log in
    if not creds or not creds.valid:
    if creds and creds.expired and creds.refresh_token:
    creds.refresh(Request())
    else:
25  flow = InstalledAppFlow.from_client_secrets_file(
    self.CLIENT_SECRET_FILE, self.SCOPES)
    creds = flow.run_local_server(port=0)
    # Create token.pickle and save the credentials for the next run
    with open('token.pickle', 'wb') as token:
30  pickle.dump(creds, token)
    return creds
```

As we can observe in line 17 within *get_credentials* method, at first we check if the file called *token.pickle* exists, and in that case, it is opened and its information is stored in the variable *creds*. Thus, we avoid to force the user to open the authorization screen. By contrast, as we have seen before, if it does not exists, we obtain the credentials by calling the class methods *from_client_secrets_file* and *run_local_server* (it is written between lines 25 and 30).

There is another case that is also reflected in the code above (in lines 23 and 24): the credentials are expired (it is possible to check it by executing *creds.expired*) and they can be refreshed (the OAuth 2.0 refresh token is *creds.refresh_token*) (Google, 2019d). In this situation, we will refresh the access token by invoking the method known as *refresh* and by giving it a *Request* object (Google, 2019c) from *google.auth.transport.requests* as the function parameter which used to make HTTP requests.

3.1.2. Building a Gmail Resource

At this point, with the OAuth 2.0 credentials, we are able to call the Gmail API. For this purpose, it is necessary to construct a resource (Google, 2019a, /v1/reference) for interacting with the API. The *build* method, from *googleapiclient.discovery* library (Gregorio, 2019), creates that object. As we will see later, this resource will lead us to manage emails, drafts, threads and everything we will like to do with the user's Gmail

data. This is why, using the *auth.py* file explained in Section 3.1.1, we are going to start every user session with the instructions below (or their equivalents in the language we are using):

```
from googleapiclient.discovery import build
import auth

SCOPES = ['https://mail.google.com/']
CLIENT_SECRET_FILE = 'credentials.json'

# Creation of an auth instance
authInst = auth.auth(SCOPES, CLIENT_SECRET_FILE)
# Constructing the resource API object
service = build('gmail','v1',credentials=authInst.get_credentials())
```

Henceforth, we will use the *service* variable to relate it with the resource object created by the *build* method.

3.1.3. Users resource

The *build* method could be called for obtaining any resource of any Google API (by giving it the suitable parameters). Our specific created *service*[3] has an important instance method that we are going to invoke for every execution: the *users()* method. It returns what is known as users resource (Google, 2019a, /v1/reference/users).

The users resource has also instance methods, which return other Gmail API resources that we are going to need, such as *drafts()*, *labels()* (see Section 3.1.4), *messages()* (see Section 3.1.5) and *threads()* (see Section 3.1.6) which return drafts, labels, messages and threads resources respectively. Moreover, it possesses the three methods that we explain hereunder (we must remember that for being able to execute any method that we are going to explain in this and next sections, it is necessary to have the appropriate authorization with at least one of the required scopes that we can look up in its documentation):

- *getProfile(userId)*: it returns an object with a dictionary structure as it follows:

```
{
   # Total number of threads in the mailbox
   'threadsTotal' : integer,
   # User's email address
   'emailAddress' : string,
   # ID of the mailbox's current history record
   'historyId' : string,
   # Total number of messages in the mailbox
   'messagesTotal' : integer
}
```

  The parameter is a string with the user's email address. If we remember the authentication process, at no time we ask the user about the email address because we decided to let the Google API functions to handle all that procedure. Therefore we have no way to know this information. Nevertheless, the special string value *'me'* can be used to indicate the authenticated user. For knowing the required scopes for invoking this function look up in (Google, 2019a, /v1/reference/users/getProfile).

[3] `http://googleapis.github.io/google-api-python-client/docs/dyn/gmail_v1.html`

- *stop(userId)*: stop receiving push notifications for the given user mailbox. As it happens with *getProfile*, the parameter is a string with the user's email address, but it is possible to use the especial string value *'me'*.
- *whatch(userId, body)*: set up or update a push notification watch on the given user mailbox.

As we are going to call only the *getProfile* method, we have described on details this first function and we have just given an idea about what the rest of them do. Now, in next sections, we are going to explain all the resources we can create with the user resource.

3.1.4. Labels resource

As we have studied, we can obtain the mentioned labels resource (Google, 2019a, /v1/reference/users/labels) by invoking *labels()* instance method of our users resource, that is to say, by using our *service* variable, the instruction *service.users().labels()* will return the label resource.

In order to obtain a label object, we will use the methods of this resource: create, delete, get, list, patch and update. In this manner, for example, we can store a label object by calling the next instructions:

```
labels = service.users().labels()
labelList = labels.list(userId = 'me').execute()
label = labels.get(id = labelList[0]['id'], userId = 'me')
```

It is necessary to use the *get* method because, as we can look up in (Google, 2019a, /v1/reference/users/labels/list), the *list* method only contains an *id*, *name*, *messageListVisibility*, *labelListVisibility* and *type* of each label, whereas the *get* method returns the label resource with all the information.

3.1.5. Messages resource

As any other resource, the messages resource has different methods, many of whom we are going to need in this work. Therefore, being aware of these methods and the operations that we are able to do with them is imperative for facing our goals. For this reason, in this section we are going to delve into the messages resource methods. As we saw in Section 3.1.3, we can access to this resource by invoking the *messages()* method when we have a users resource. We will limit ourselves to describing the methods we may need to use:

- *attachments()*: returns the attachments resource (for more information about this resource refer to (Google, 2019a, /v1/reference/users/messages/attachments)).
- *get(userId, id, format = 'full', metadataHeaders = None)*: if successful, this method returns the requested messages resource. Its parameters are:
  - *id*: the identifier string of the message we are looking for.
  - *userId*: the user's email address. As it happens with the *getProfile* method of the users resource (see Section 3.1.3), the special string value *'me'* can be used to indicate the authenticated user.
  - *format* (optional parameter): the format in which we want the message returned. This field can take the following punctual values: *'full'* (returns the entirely email data with body content parsed in the *payload* messages resource

field and the *raw* field is empty), *'metadata'* (returns only an email message with its identifier, email headers and labels), *'minimal'* (returns only an email message with its identifier and labels) and *'raw'* (returns the entirely email message data with the body content in the *raw* messages resource field as a base64url (see Section 2.1.1.3) encoded string and the *payload* field is empty).

- *metadataHeaders* (optional parameter): it is only used when the format parameter takes the punctual value of *'metadata'*. It is a string list where we have to insert the headers we want to be included.

For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/messages/get).

- *list(userId, includeSpamTrash = false, labelIds = None, maxResults = None, pageToken = None, q = None)*: returns a resource with the following structure:

```
{
    'messages' : [ users.messages resource ],
    'nextPageToken' : string,
    'resultSizeEstimate' : unsigned integer
}
```

As it happens with the *list* method of the labels resource (see Section 3.1.4), *'messages'* list does not contain all of a message information (for obtaining the full email data we can use *get* method). Each element of this list only contains the *id* and *threadId* field.

The parameters of this method are:

- *userId*: user's email address (we can use the special string value *'me'*).
- *includeSpamTrash* (optional parameter): boolean parameter which determines if it includes messages with the labels *SPAM* and *TRASH* in the result of the operation.
- *labelIds* (optional parameter): it is a list which let us filter the messages by only returning emails with labels that match all of the identifiers that belong to this list.
- *maxResults* (optional parameter): an integer which determines the maximum number of messages to return.
- *pageToken* (optional parameter): string which specifies a page of results.
- *q* (optional parameter): string which let us do an specific query (with the same query format as the Gmail search box) and filter the messages by only returning emails that match with it.

For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/messages/list).

- *send(userId, body = None, media_body = None, media_mime_type = None)*: it sends the given message to the email addresses specified in the *To*, *Cc* and *Bcc* headers. The first two parameters are the only ones we will use. The first (*userId*) is the user's email address (we can use the special string value *'me'*) and the second is the message we want to send in an RFC 2822 (Resnick, 2001) format. For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/messages/send).

3.1.6. Threads resource

In addition to messages, we will also manage the threads of the user. Because of it, knowing the main operation with them will be necessary. The most important methods of this resource are:

- *get(userId, id, format = 'full', metadataHeaders = None)*: if successful, this method returns the requested threads resource. In respect of the parameters, they are defined in the same way as in *get* messages resource method (see Section 3.1.5) with the exception of the parameter *format*, whose only difference is that it does not accept the *'raw'* value. For knowing the required scopes for invoking this function look up in (Google, 2019a, /v1/reference/users/threads/get).

- *list(userId, includeSpamTrash = False, labelIds = None, maxResults = None, pageToken = None, q = None)*: if successful, it returns a dictionary structure analogous to the view in the *list* message resource method (see Section 3.1.5). Needless to say, instead of returning a messages resource list it will give us a threads resource list, which does not contain the complete information of each thread (for example each element of the list has not a list of messages resource). Full thread data can be fetched using the previous method. The parameters of this method are defined in the same way as the *list* messages resource method. For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/threads/list).

3.2. spaCy

After extracting the user's e-mails, we should be able to analyse the body of the e-mails. To do this we will need a syntactic parser in order to separate the different texts in tokens (in other words, to segment text into words, punctuation marks, etc.) and obtain different characteristics from them (such as their part of speech) for the purpose of being able to calculate the metrics explained in Section 2.2.4. To attain that objective, we are going to use the library spaCy.

3.2.1. spaCy versus others syntactic parsers

We have chosen spaCy as our syntactic parser against others for several reasons that we will explain below.

An evaluation published by *Yahoo! Labs* and Emory University, as a part of a survey of current parsing technologies (Choi et al., 2015), observed that "spaCy is the fastest greedy parser" and its accuracy is within 1% of the best available (as we can see in Figure 3.1). The few systems that are more accurate are 20 times slower or more.

Choi et al. (2015) results and subsequent discussions helped spaCy develop a novel psychologically-motivated technique to improve spaCy's accuracy, which they published in joint work with Macquarie University (Honnibal and Johnson, 2015).

Besides, not only in general but in each particular task (tokenisation, tagging and parsing), spaCy is the fastest if we compare it with other natural language processing libraries. This is shown in Figure 3.2, where we can observe both absolute timings (in ms) and relative performance (normalized to spaCy). Lower is better.

Finally, spaCy has three pretrained model pipelines for Spanish with a very high accuracy (see Figure 3.3). These will help us to tokenise, tag and parse our messages in order to calculate the different style markers defined.

SYSTEM	YEAR	LANGUAGE	ACCURACY	SPEED (WPS)
spaCy v2.x	2017	Python / Cython	**92.6**	*n/a*
spaCy v1.x	2015	Python / Cython	91.8	13,963
ClearNLP	2015	Java	91.7	10,271
CoreNLP	2015	Java	89.6	8,602
MATE	2015	Java	92.5	550
Turbo	2015	C++	92.4	349

Figure 3.1: Benchmarks of different syntactic parsers
Image extracted from `https://spacy.io/usage/facts-figures#benchmarks`

	ABSOLUTE (MS PER DOC)			RELATIVE (TO SPACY)		
SYSTEM	TOKENIZE	TAG	PARSE	TOKENIZE	TAG	PARSE
spaCy	0.2ms	1ms	19ms	1x	1x	1x
CoreNLP	0.18ms	10ms	49ms	0.9x	10x	2.6x
ZPar	1ms	8ms	850ms	5x	8x	44.7x
NLTK	4ms	443ms	*n/a*	20x	443x	*n/a*

Figure 3.2: Per-document processing time of various NLP libraries
Image extracted from `https://spacy.io/usage/facts-figures#benchmarks`

3.3. Flask

3.4. Mongo DB

MODEL	SPACY	TYPE	UAS	NER F	POS	WPS	SIZE
es_core_news_sm 2.0.0	2.x	neural	89.8	88.7	**96.9**	*n/a*	**35MB**
es_core_news_md 2.0.0	2.x	neural	**90.2**	89.0	97.8	*n/a*	93MB
es_core_web_md 1.1.0	1.x	linear	87.5	**94.2**	96.7	*n/a*	377MB

Figure 3.3: Benchmark accuracies for the Spanish pretrained model pipelines Image extracted from `https://spacy.io/usage/facts-figures#benchmarks`

Chapter 4

Style Analyser

In order to generate messages with the user's writing style, it is necessary to define parameters which will determine and describe it. For this purpose, we have developed a style analyser that extracts the messages written by the user and obtains the value of various metrics from them. Then it will be useful for analysing different user's emails and drawing conclusions about what parameters describe the writing style of each person more accurately.

In this section we are going to explain the architecture of this analyser (see Section 4.1) and each of the modules that compose it (they are explained in Sections 4.2, 4.3, 4.4, 4.5 and 4.6). Finally, we are going to present the behaviour of the execution with the Gmail account that is going to be analysed (this discussion can be looked up in Section 4.7).

4.1. Architecture

The first step when we are designing a system's architecture is to know its input and output. In this case, we want to implement a natural language processing system that analyses the writing style of e-mails. As we have previously mentioned, the stylometric analysis will be represented through chosen style markers. Therefore, our system's output is going to be that chosen metrics (they are explained in section 4.5) of each message.

In respect of the system's input, because of the nature of the problem we face, it is reasonable to think that it must be a single e-mail. However, we do not have the corpus of e-mails to analyse. For this reason, our first step will be to extract the e-mails that will be analysed. Hence, our system's input is going to be the Gmail user for accessing to the information that we are interested in. Therefore, we are going to develop a system which receives a Gmail user as input and obtains different metrics of each message sent by the given user as output.

Once we clearly know the input and output of our system, we need to define the different steps that a message have to take for being analysed. In this manner, we are going to design a pipeline architecture with four different phases (extraction, preprocess, typographic correction and measuring) as Figure 4.1 shows. Thus, we divide the original job in 4 different and more simple tasks with distinct inputs and outputs required. This division into phases addresses both the need to atomise each of the steps necessary to obtain the desired output, and to take advantage of benefits that a single indivisible system does not provide. One of these advantages is the possibility of working in parallel with each of the different phases. Another advantage, without a doubt, is the greater facility for

the correction of errors in the pipeline. Thus, if an error of any kind is found in any phase, this will not affect the implementation of subsequent phases and it will not be necessary to modify the entire system. This, together with the fact that each phase stores its corresponding output using different Mongo DB documents (see Section 3.4), allows us to avoid re-running previous phases to the one in which the error occurred. Finally, it is also important to note the advantage of reusing each of the phases separately without having to rely on the others

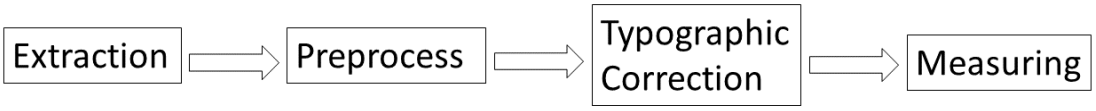


Figure 4.1: Pipeline architecture of the style analyser

As it is easy to deduce, each of these phases is going to be developed as a different module. This implementation will have the advantage that each module is going to be able to work independently from the other modules, which will allow them to work in parallel. That is to say, while a message is being extracted, other e-mails could be being preprocessed, corrected or measured if they have gone through the previous phases. This optimizes the time it takes for a message to be extracted until the respective metrics are calculated (it does not have to wait for the others to move to the next phase of the pipeline). In addition, the last three (preprocess, typographic correction and measuring) have been implemented as web services using Flask (see Section 3.3), which facilitates their reuse even separately in other works and projects. Let us now briefly explain what each of the defined phases consists of.

The first step, extraction phase, as the reader can imagine, consists of the extraction of each one of the sent messages of the given user. In this task, we are going to take advantage of all the studied concepts about the Gmail API (see Section 2.1.5) and make use of every resource it provides us. Besides, we will try to minimize the consumed quota units in each extraction, which means we will only make the requests to the Google Servers that are strictly necessary. This first step is not just the task of extracting the resource that represents each sent message from the user's account, but also the job of transforming it to the format that the preprocessing module needs. Hence, the input of this module will be the same input as that of the complete system (a Gmail user) and its output will be an extracted message ready for being preprocessed.

As for the second step, the preprocessing phase, consists of the modifying the extracted message so that it can be interpreted by the spaCy's natural language processing model to be used. Some of the changes that a message could suffer in this phase are: the removal of the signature, the disposal of the replied messages which appears under the text, the elimination of soft break lines that quoted-printable codification (see Section 2.1.1.4) introduce in some messages, etc. This module also addresses the need to remove characters and structures that do not correspond to those used in a plain text such as bold or italic type styles, font sizes and fonts, enumerations or bulleted lists, etc. Likewise, its output is a message with its body as a plain texts.

In the implemented metrics (as we'll see in section 4.5) we won't take into account typographical errors (such as a spelling mistake). So we will need to fix them as much as possible, and this is the typographic correction module's task. In the same way, it is possible that some tokens do not belong to our spaCy model's vocabulary. Therefore, it will be necessary to know lexical-syntactic information about the token, such as its part of

speech and its lemma. These are the task of the typographic correction module.

Finally, the measuring module is in charge of calculating all the style features chosen for this work. For that purpose, it receives a message (extracted by the extracting module) with a plain text format (thanks to the preprocessing module) free from typographic error (thanks to the typographic correction module) and obtains the result of measuring all the style markers selected in the given message.

As we have explained, the input of the extracting module is a Gmail user and the input of the rest of the modules is an only message. However, each module is independently from each other, which means that it is necessary to have a way of assembling all this modules. For this purpose, the *Analyser* class is developed (see Section 4.6). This entity is in charge of sending to each module the required input in order to obtain its output. Besides it presents the system to the user, communicates the information and captures the user's information (it performs a previous filtering to check that there are no formatting errors), such as the typographic correction of the errors found. In this manner, the architecture of our style analyser system is as shown in the Figure 4.2 (in the following sections we will delve into each module of this system), which represents the UML (Unified Modelling Language) class diagram of it. In this figure we have avoided including both attributes and methods of each class, since with it we want to show the general structure of the system. In the sections corresponding to each of the packages and the *Analyser* class, they attributes and methods will be specified and explained.

As we wanted, the *Analyser* manages the communication between each UML package (which represent each of the mentioned modules). Thus, this UML class will transport the output of each module to the input of the subsequent phase so as to fulfil the pipeline established in Figure 4.1.

If we look at the relationship shared by the *Analyser* class and the *Extraction* package (specifically with the *Extractor* class of this module), we will notice that it is an unidirectional binary association, what it means that the objects of the second are connected with the objects of the first. Besides, we can observe a multiplicity index in the arrowhead (the *Extraction*'s end), which means that the *Analyser* will be related with an only *Extractor* object (it will not be necessary more than one *Extraction* package in order to obtain all user's sent messages).

The rest of packages are "used" by the *Analyser*, through a POST HTTP request, because all of them are implemented as web services. It contacts with the module's app which is related with the module's main class (which is in charge of carrying out its corresponding task).

An important observation to mention is the fact that all packages interact with their corresponding classes, which act as DAO (Data Access Object), with the database used (with Mongo DB technology as it is explained in Section 3.4). Their interaction is based on storing their results in it. The main advantage of this implementation is that it is not required to have enough dynamic memory in order to process every message at the same time. In addition to it, as we have explain, if an error is detected in an specific phase, it is not necessary to execute the previous modules again. With this in mind, it is reasonable to think that each module's main class, of the last three phases, will make use of the corresponding class with the purpose of saving its result, obviously after finishing its execution with the given message.

Below we only have to enter in detail of each of the packages and of the *Analyser*, in order to completely understand the style analyser.

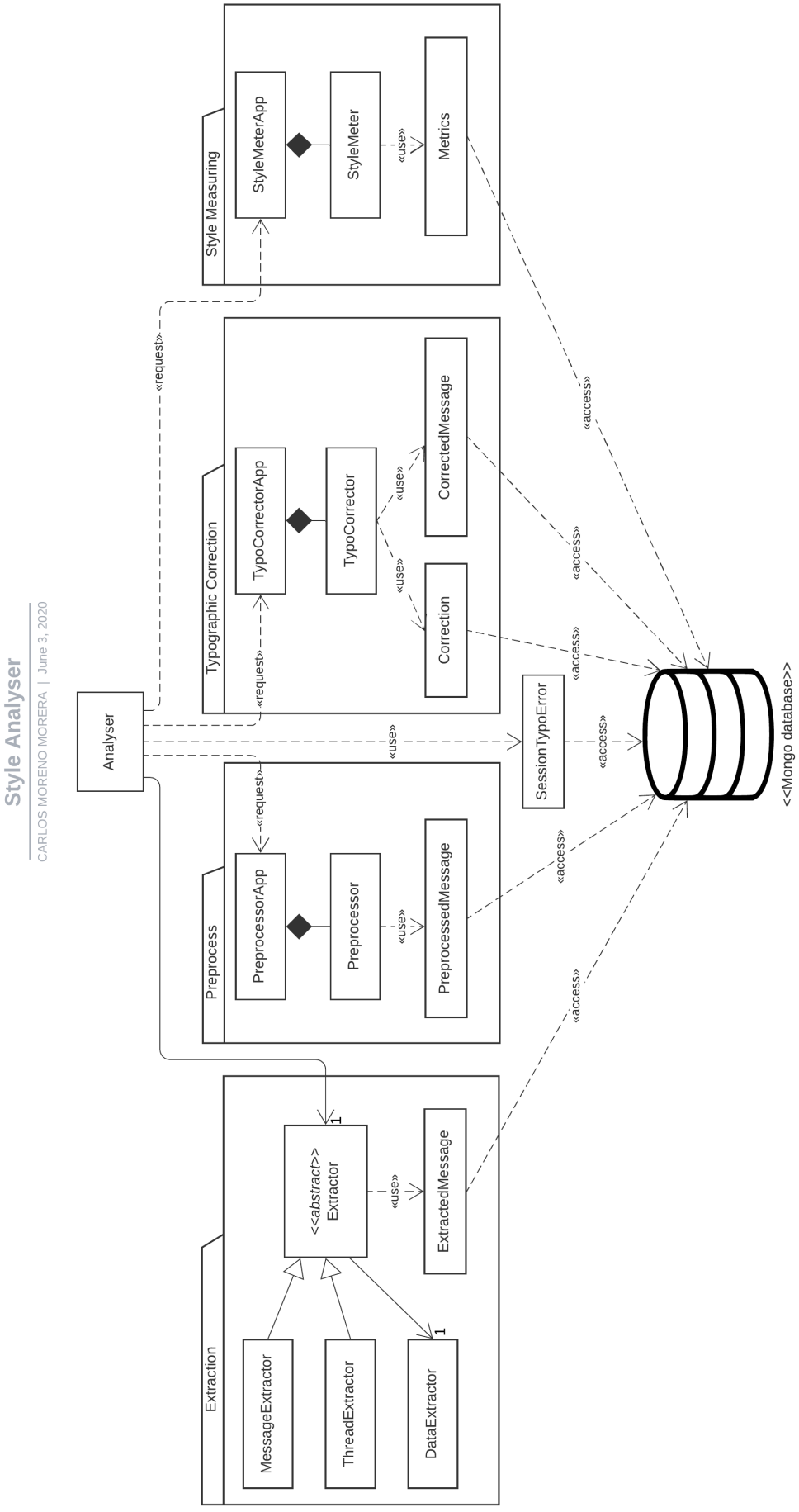


Figure 4.2: UML class diagram of the style analyser

4.2. Extracting module

Extracting module encapsulates all the necessary class in order to extract the given user's sent messages. As it is shown in Figure 4.3, this UML package has five different UML classes: *Extractor*, *MessageExtractor*, *ThreadExtractor*, *DataExtractor* and *ExtractedMessage*.

The main class of the above five is the *Extractor* class, which is an abstract class implemented by *MessageExtractor* and *ThreadExtractor* classes. The reason for implementing it as an abstract class with the abstract methods *get_list*, *get_resource* and *extract_sent_msg*, lies in the desire to minimise the number of quota units used during this process. Let's explain this in detail.

As we have see in the Table 2.1, to carry out the messages resource's operation costs five quota units and to perform the same operations for the thread's resource costs ten quota units. However, when the operation *messages.get* is invoked we get a single message, whereas when the operation *threads.get* is called we get as many messages as there are in the thread. Therefore, minimising the amount of quota units used depends on the number of messages and threads we have.

When we are in the extraction process, at first it is necessary to invoke a *list* method. It will return a list of, at least, 100 identifiers of the resource (message or thread). Then, these identifiers will be used to obtain (by calling the corresponding *get* method) each of the listed resources. If we want to obtain the identifiers of the remaining resources, we will have to invoke *list* again with the *nextTokenPage* obtained in the previous call. With this in mind, we are going to invoke the corresponding *list* method as many times as the result of applying the ceiling function to the division of the number of resources by 100; and we are going to invoke the corresponding *get* method as many times as the amount of resources that the user has (this number is possible to know by calling the *get* method of the labels resource and giving it the string value *SENT* as its parameter called *id*, which only consumes one quota units each time). Hence, the number of quota units inverted in an extraction process will be determined by the following formula:

$$Q = L \cdot \left\lceil \frac{N}{100} \right\rceil + G \cdot N$$

Where L is the cost in quota units of invoking the corresponding *list* method once, N is the number of resources that the user has and G is the cost in quota units of calling the corresponding *get* method once. It is important to remember that the division is the integer and not the real.

Following the previous expression and the quota units cost of the table 2.1, we are able to claim that the number of quota units inverted in a message extraction process will be:

$$Q_M = 5 \cdot \left\lceil \frac{N_M}{100} \right\rceil + 5 \cdot N_M$$

Where N_M is the amount of sent messages that the user has. However, the cost in quota units of a thread extraction process will be determined by:

$$Q_T = 10 \cdot \left\lceil \frac{N_T}{100} \right\rceil + 10 \cdot N_T$$

Where N_T is the number of sent threads that the user has. Consequently, when we obtain that $Q_M < Q_T$, we will save more quota units by executing a message extraction process and, when $Q_T < Q_M$, it will happen by executing a thread extraction process.

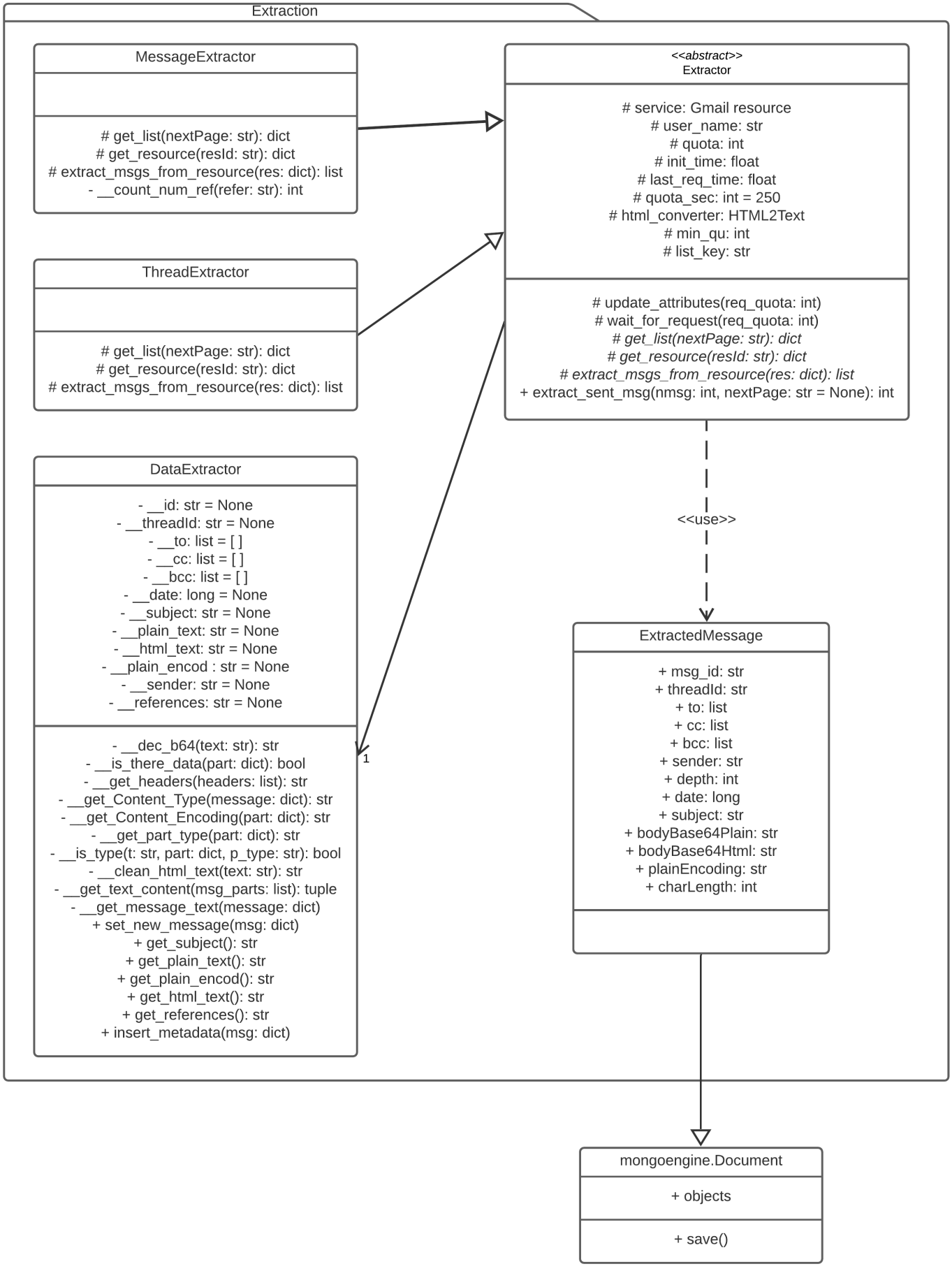


Figure 4.3: UML class diagram of the extracting module

Returning to the extracting module, even though the choice between the two types of extraction is done by the Analyser (see Section 4.6), it is necessary to implement both possibilities. For this reason, *Extractor* class was implemented as an abstract class with the methods related to the *list* and *get* functions of the Gmail API: *get_list* (which calls the corresponding *list* method) and *get_resource* (which invokes the corresponding *get* method). In addition to them, we can find the *extract_msgs_from_resource* as an abstract method. This function is in charge of extract the necessary information from the corresponding resource in order to obtain a list of messages (in the case of the message resource the list has only one item) which are going to be stores in the database. In other words, it transforms a message (or a thread of messages) with the structure explained in Section 2.1.5.4 into the following structure (in the case of the thread resource each message will be transformed to the following structure):

```
{
    'id' : string,
    'threadId' : string,
    'to' : [ string ],
    'cc' : [ string ],
    'bcc' : [ string ],
    'sender' : string,
    'depth' : int,                  # How many messages precede it
    'date' : long,                  # Epoch ms
    'subject' : string,
    # Body as plain text encoded using base64
    'bodyBase64Plain' : string,
    # Body as html text encoded using base64
    'bodyBase64Html' : string,
    # Original encoding of the body as a plain text
    'plainEncoding' : string,
    'charLength' : int
}
```

Once the message has the structure above, it is ready to be saved in the database, because of, as we can observe in Figure 4.3, the dictionary keys are the same as the attributes of the *ExtractedMessage* class (which inherits from the *mongoengine.Document* class, allowing it to insert elements in the database). Indeed, the *extract_msgs_from_resource* method returns a list of *ExtractedMessage* objects.

In addition to the abstract methods, in the *Extractor* class we can find the *extract_sent_msg*, which is in charge of the extraction algorithm that we mentioned before (invoke the corresponding *list* method, the for each identifier get the resource, call the function *extract_msgs_from_resource*, etc.). During this process, the *Extractor* must check that it does not exceed the set limit of quota units, both the daily limit and the secondly limit. Once the daily limit is reached, the process must stop. In the case of the secondly limit, the *Extractor* must stop and wait until it can continue using the Gmail API operations. This is the task of *update_attributes* (which updates the time and quota attributes such as *quota*, *quota_sec*, *last_req_time* and *init_time*) and *wait_for_request* methods.

In respect of the *Extractor* class, there is an only remaining detail that should be mentioned. It has an attribute called *html_converter*. This attribute is an object of the *HTML2Text* class from *html2text* Python's library[1], which is used in order to obtain the message body as a plain text from the e-mail body as HTML text in the event that there is a lack of first one.

[1] `https://pypi.org/project/html2text/`

If we observe the *Extraction* package, we will find the *DataExtractor* class, which is related with the *Extractor* class by an uni-directional binary association with a multiplicity index in the arrowhead (the *DataExtractor*'s end). It performs the task of extracting the information of a given e-mail. To this end, it goes through the headers list (where information as the recipient can be found) and the MIME message parts tree studied in Section 2.1.1.2, in order to get the message body. Likewise, once an e-mail is extracted from Gmail API, the *DataExtractor* class receives it and acquires all the required information from it.

4.3. Preprocessing module

Preprocessing module receives the message with the structure given by the extracting module and modifies the e-mail so that it can be interpreted by the spaCy's pretrained model. As it is shown in Figure 4.4, this UML package has three different UML classes: *PreprocessorApp*, *Preprocessor* and *PreprocessedMessage*.

First of all, we can observe the *PreprocessorApp* class. It inherits from *Flask* class (see Section 3.3), which implements a simple web service. Consequently, if we want to preprocess a message, it will be necessary to execute a POST HTTP request with the e-mail as a *json* in it. Having done so, the *preprocess_message* method will be invoked and send the given message structure to the *Preprocessor* class by calling its only public method: *preprocess_message*. There is also the possibility of transmitting the user's e-mail signature to the *Preprocessor* (so that it can be removed from the different messages) by including it as an string in the sent *json*.

The main class of this UML package is the *Preprocessor* class. It is in charge of modifying the given message. For this reason, it has different methods which implements the distinct tasks that it has to carry out.

The first task this module performs is to filter those e-mails whose message body as plain text is empty, which means that they lack the *bodyBase64Plain* field. As our purpose is analyse the writing style of the user, we are not interested in e-mails without text. Thus, these messages are not preprocessed.

Then, the images inserted in the message body (not as an attachment) are removed by calling the *__removed_pasted_images* method. This function make use of the simple Python's regular expression `r'\[image:[^\]]+\]'`, detects the position of the different images with it and takes them away.

Once pasted images are removed, the *__extract_body_msg* method removes the text of replied e-mail (if it exists) and the soft break lines inserted in the body, as a consequence of the established format in Gellens (1999). When someone replies an e-mail from a Gmail account, the replied message is automatically included under the response (indeed it is possible to intersperse the answer and the responded text). As it is not a written composed by our user, this copied text must be taken away. To this end, the *Preprocessor* class creates an object of the *EmailReplyParser* class of the *email_reply_parser*[2] Python's library. With its *parse_reply* method, only the response is obtained with the replied message's header automatically included (*EmailReplyParser* class does not remove it). However, such header is easy to detect by using regular expressions, due to it has an specific format as the following line (written in Spanish):

El mié., 27 may. 2020 a las 11:11, Name (<example@mailserver.com>) escribió:

The designed regular expression detects this type of sentences with the moment (date

[2] `https://pypi.org/project/email_reply_parser/0.1.0/`

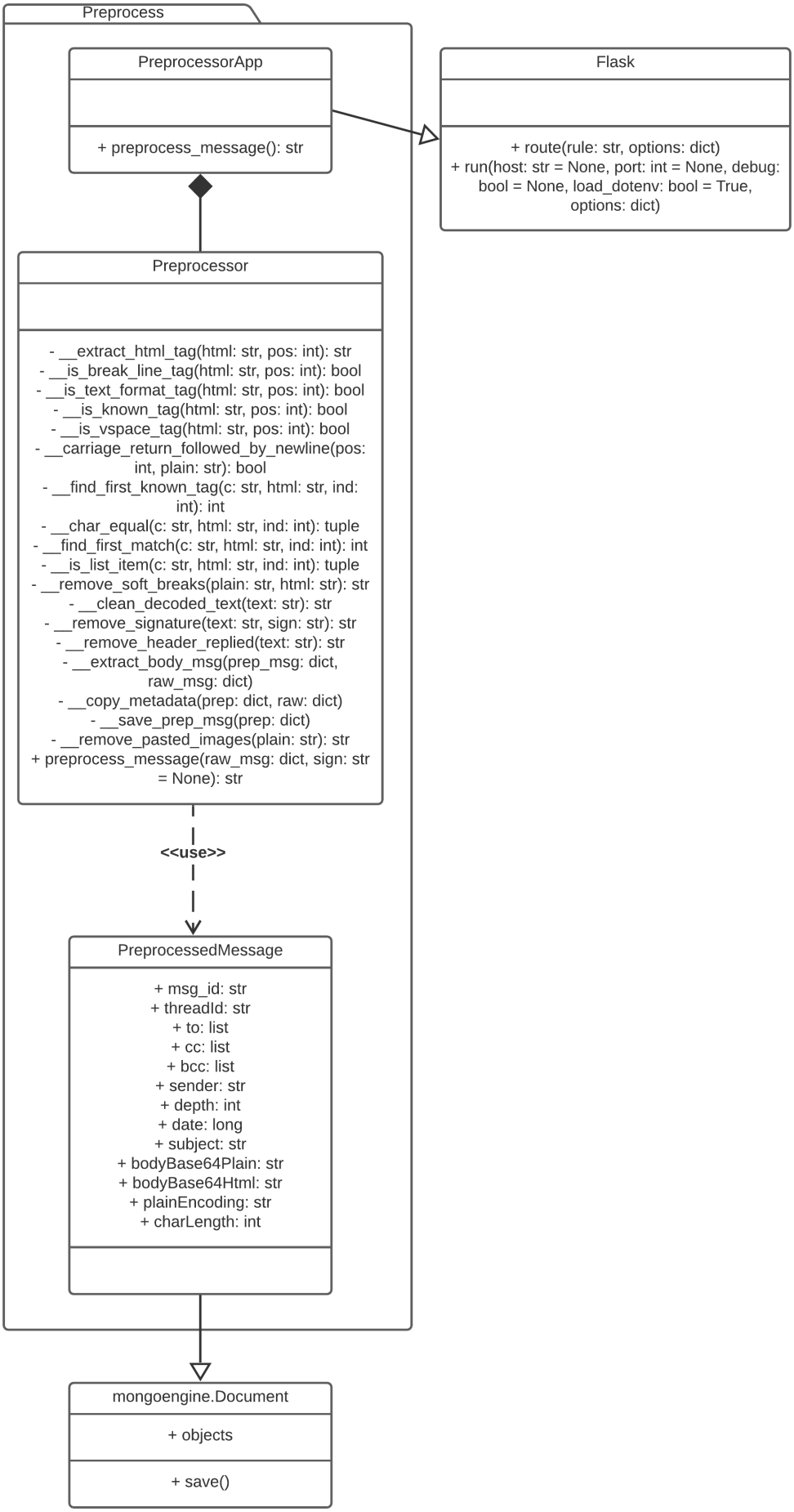


Figure 4.4: UML class diagram of the preprocessing module

and time) and the sender. Once *Preprocessor* knows its position, it is possible to take it away.

A similar problem appears with forwarded messages. Nevertheless, unlike replied e-mails, it is not possible to detect if the user has interspersed new text in the forwarded written. For this reason, *Preprocessor* detects the forwarded header, which indicates the beginning of the resent message, and deletes all the text from it.

In addition to the replied or forwarded text, we find the problem of the inserted soft break lines in order to follow the standard format for sending e-mails. We have implemented two solutions for this issue in *__clean_decoded_text* and *__remove_soft_breaks* methods. The first function deletes all soft break lines in messages encoded with quoted-printable (see Section 2.1.1.4). The second, compares the message body as HTML text and as plain text and removes all soft break lines that do not appear as an HTML tag. Besides, during this process we detects characters that should not appear in the plain text, for instance, Gmail delimits the text in bold with the symbol "*" (there are more examples as the beginning of a bulleted list, an enumeration or the change of font or font size). As in the HTML text it will appear between two tags, we recognise this fact and take the delimiter character away. In this way we are able to obtain a real plain text.

The last modification made by the preprocessor is the removal of the signature. This only happens if the user has provided it to the system, since the recognition of the signature is a complex problem that is not the objective of this work. The *__remove_signature* method is is responsible for carrying out that task.

Once an extracted e-mail is preprocessed, it is ready to be saved in the database. As it happened with the *ExtractedMessage* class, the *PreprocessedMessage* class inherits from the *mongoengine.Document* class, allowing it to insert elements in the database. If we compare the *ExtractedMessage* and *PreprocessedMessage* class, we will realise that they have the same attributes, so a preprocessed message has the same structure as an extracted one.

4.4. Typographic correction module

Typographic correction module receives the message with the structure given by the preprocessing module and detects the typographic errors present in the given e-mail. As it is shown in Figure 4.5, this UML package has four different UML classes: *TypoCorrectorApp*, *TypoCorrector*, *CorrectedMessage* and *Correction*.

As it happened with *PreprocessorApp*, *TypoCorrector* inherits from *Flask* class and, thanks to it, this class implements a simple web service. However, unlike *PreprocessorApp*, *TypoCorrector* has two different methods which carry out distinct task. These two functions correspond to the two *TypoCorrector*'s public method with the same name. Thus, if we want to invoke one of these public methods, it will be necessary to execute a POST HTTP request with an e-mail, in order to be corrected (in the case of the method *correct_msg*), or with the unrecognised token (which has been wrongly classified as "out of vocabulary" by our spaCy's model) that is going to be saved (we will explained both tasks in detail later). Each one of them is going to have a different url address.

The main class of this UML package, as it happens with the rest of packages, is the *TypoCorrector* class. It is in charge of detecting the typographic errors and correcting them if it is possible. For this purpose it makes use (as an attribute) of an spaCy's pretrained model, specifically the one called *es_core_news_md*[3].

The first public method that we are going to explain is *correct_msg*, which receives

[3] `https://spacy.io/models/es`

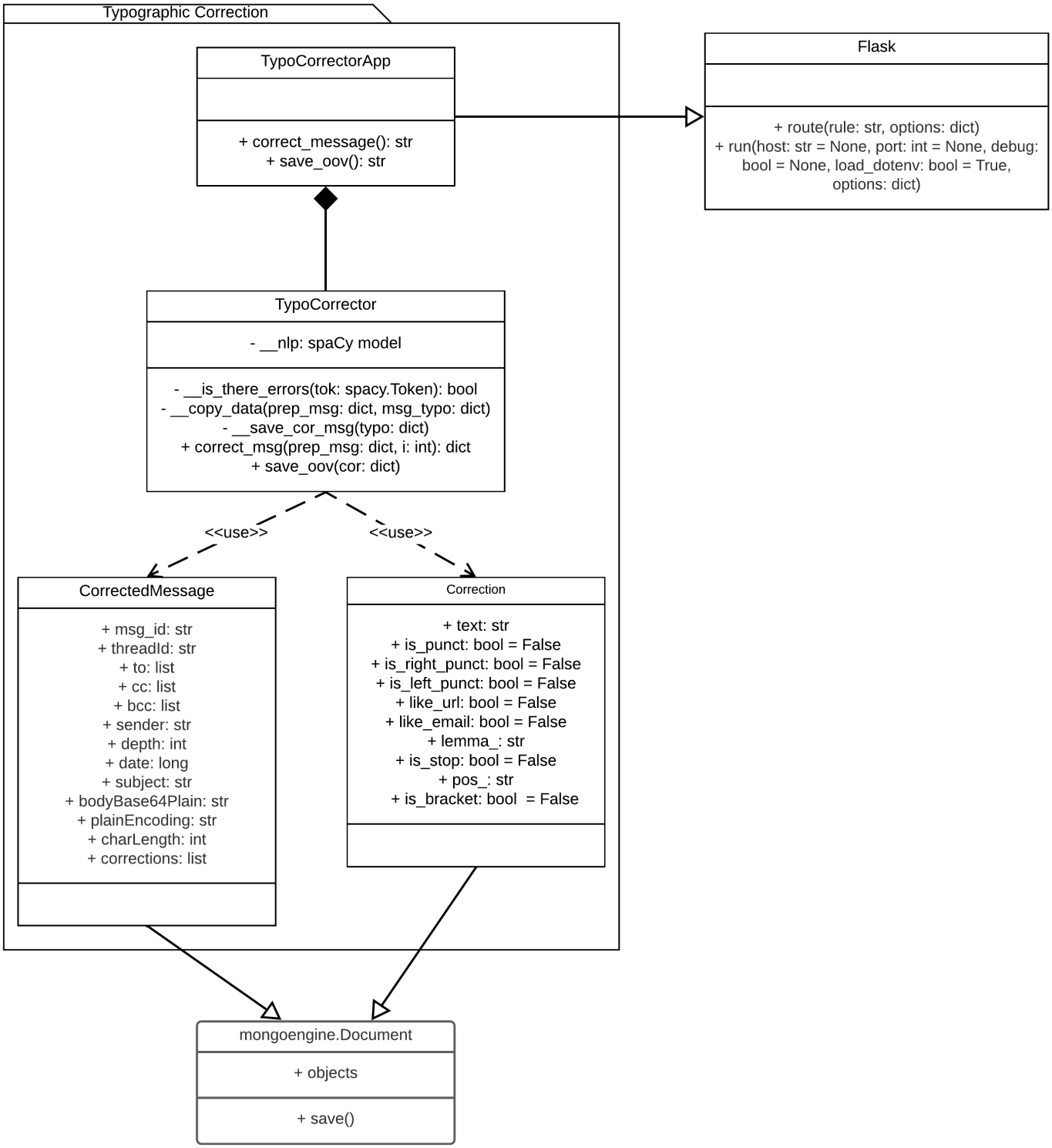


Figure 4.5: UML class diagram of the typographic correction module

as parameters a message and an index. The method's parameters are originally a preprocessed message with its structure and the index as 0, which indicates that the e-mail must be corrected from the beginning, because it points the word from which the typographic correction should be made. When the function finishes its operations, it returns a dictionary with the following structure:

```
{
    'typoCode': <enum 'TypoCode'>,
    'index': int,
    'typoError': str,
    'token_idx': int,
    'message': {
        'id' : string,
        'threadId' : string,
        'to' : [ string ],
        'cc' : [ string ],
        'bcc' : [ string ],
        'sender' : string,
        'depth' : int,                  # How many messages precede it
        'date' : long,                  # Epoch ms
        'subject' : string,
        'bodyPlain' : string,
        'bodyBase64Plain' : string,
        'plainEncoding' : string,
        'charLength' : int,
        'corrections' : [
        {
            'text': str,
            'is_punct': bool,
            'is_right_punct': bool,
            'is_left_punct': bool,
            'like_url': bool,
            'like_email': bool,
            'lemma_': str,
            'is_stop': bool,
            'pos_': str,
            'is_bracket': bool,
            'position': int
        }
        ]
    }
}
```

If no typographic errors are detected in the text or the user's help is not needed, the *message* field is previously saved in the database, by using the *CorrectedMessage* class (its attributes perfectly match with the fields of the *message* field dictionary), the *typoCode* field will take the value *successful* and the *typoError* and *token_idx* fields will take the value *None*. However, in other case, the execution of the system changes.

The first task the *correct_msg* method performs is to filter those e-mails whose message body as plain text is empty, due to the preprocess could produce this result, such as in forwarded messages without new text. In this case, the *typoCode* will take the value *notAnalysed*.

Then, it checks from the given index onwards if one token is recognised by our spaCy's model as "out of vocabulary". If this happens, the word is searched in the database of corrections (which is easy to manage thanks to the *Correction* class) in case it was previously

stored in it as a non-out-of-vocabulary token (in this database we have all words that are not really a typographic error, but they are not recognised by our natural language processing model). If the word appears in it, the information of this "correction" is appended to the *"corrections"* list (each of its elements has the same field as the *Correction* class' attributes) and the execution continues as usual, as if no error has been detected.

In the other hand, if the detected out of vocabulary token is not in our *Correction* database, which means that it could be a real typographic error or a existent word which is not recognised by our spaCy's model and not previously stored, the *Analyser* class (out of this module), with the help of the user, will be in charge of determining if it is a real typographic error and correcting it in that case. For this reason, the *TypoCorrect* will return the mentioned structured with the *typoCode* field taking the value *typoFound*, the *word* one with the text of the detected token and the *token_idx* with the character position of the beginning of the found word. The *index* field will always take the value of the position of the last analysed word, if there are no errors detected it will be the number of words in the message.

Once the *Analyser* has determined if the given word is a real typographic error, and corrected it in that case, it will invoke again the *correct_msg* method, through a POST HTTP request, and it will send as a parameter the returned *message* field (probably with the message body changed or with a new element in *corrections* list) and with the corresponding index, for example if the detected token was not a real typographic error, the index will be the position after the word (due to the previous words has been analysed yet). This is the advantage of this function, it allows us to start a new typographic correction or continue a previously started one, because it admits as the *prep_msg* parameter a preprocessed message or a partially corrected message.

In this Section, we have explained when the *Correction* queries are carried out, but we have not said anything about when its elements are inserted. For this purpose, the *save_oov* method was implemented. If the *Analyser* determines that the returned word is not a typographic error, it can carry out a POST HTTP request in order to save the information of this word in the database for future cases.

4.5. Measuring module

Measuring module is in charge of calculating all the selected writing style metrics. In order to measure these features, it receives from the *Analyser* class a corrected message with the following structure (which matches with the stored messages' structure):

```
{
    'id' : string,
    'threadId' : string,
    'to' : [ string ],
    'cc' : [ string ],
    'bcc' : [ string ],
    'sender' : string,
    'depth' : int,                  # How many messages precede it
    'date' : long,                  # Epoch ms
    'subject' : string,
    'bodyBase64Plain' : string,
    'plainEncoding' : string,
    'charLength' : int,
    'corrections' : [
    {
```

```
            'text': str,
            'is_punct': bool,
            'is_right_punct': bool,
            'is_left_punct': bool,
            'like_url': bool,
            'like_email': bool,
            'lemma_': str,
            'is_stop': bool,
            'pos_': str,
            'is_bracket': bool,
            'position': int
        }
        ]
}
```

As we can see in Figure 4.6, the style measuring package has three different classes with a class diagram similar to that of the preprocess package. These three classes are: *StyleMeterApp*, *StyleMeter* and *Metrics*.

As with the two previous modules, this package implements a Flask web service which can be used through a POST HTTP request with the message as a *json* in it. Having done so, the *measure_style* method of the *StyleMeterApp* class will be invoked and send the given message structure to the *StyleMeter* class by calling its only public method: *measure_style*.

The main class of this UML package is the *StyleMeter* class. It is in charge of calculating the style features. For this reason, it has different methods which implements the distinct style markers that it has to evaluate.

As the reader is able to deduce after presenting the previous Sections, the *Metrics* class will store in the database the results of measuring each message. The *StyleMeter* class will use it once it has calculated all the style characteristics.

We have used 31 lexical-syntactic features (due to previous studies, such as Homem and Carvalho (2011), yield encouraging results with lexical-syntactic features), following the classification of Abbasi and Chen (2008) (which categorised stylistic features as lexical, syntactic, structural, content-specific and idiosyncratic style markers), and we will now divide them into 4 categories in which we have grouped them according to their usefulness in terms of what type of conclusions we can infer from each of them. These categories are: part of speech features (see Section 4.5.1), punctuation features (see Section 4.5.2), vocabulary features (see Section 4.5.3) and structural features (see Section 4.5.4). We must not confuse this latter category (which it belongs to the lexical features of the classification given in Abbasi and Chen (2008)) with the structural metrics explained in Abbasi and Chen (2008).

Some of the popular metrics which are not used in this work, belong to the structural, content-specific and idiosyncratic style markers of Abbasi and Chen (2008), but there are others which belong to the same categories as the explained metrics (lexical and syntactic).

The choice of the metrics presented below, some essentially simple, has been directed by the objective of finding easily explainable characteristics that set the parameters of the style of writing according to the recipient of the e-mail, and then be able to use this study to develop, in future projects, systems of natural language generation of e-mails that take into account this factor. For this reason, some excessively complex metrics, although popular in stylometry, have been avoided and an attempt has been made to prioritize the explainability of the chosen features.

Finally, we are going to relate the explained style markers with their implementation

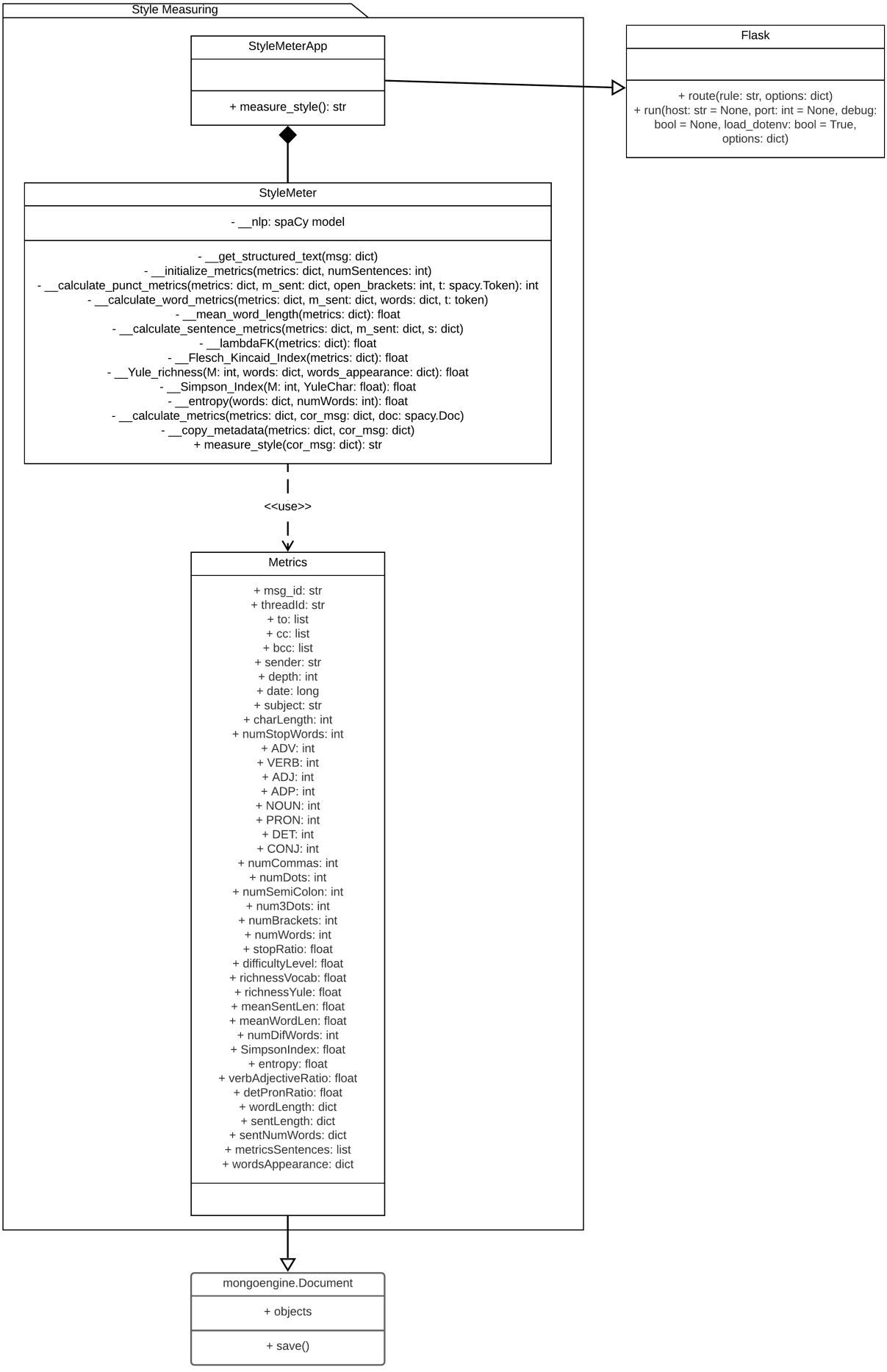


Figure 4.6: UML class diagram of the measuring module

(see Section 4.5.5) in the *Metrics* class.

4.5.1. Part of Speech features

We will call our part of speech metrics as the syntactic features which have to do with the part of speech of each word of the e-mails. Following the suggestion of Holmes (1985), we count the number of nouns, verbs, adjectives, adverbs, pronouns, determinants, conjunctions and prepositions of each text. By calculating this, significant stylistic traits may be found, because as Somers (1966) claims: "A more cultivated intellectual habit of thinking can increase the number of substantives used, while a more dynamic empathy and attitude can be habitually expressed by means of an increased number of verbs. It is also possible to detect a number of idiosyncrasies in the use of prepositions, subordinations, conjunctions and articles".

In adding to this metrics, we calculate the verb-adjective ratio and the determinant-pronouns ratio, extracted from Antosch (1969) and Brainerd (1974), respectively.

4.5.2. Punctuation features

In order to extract conclusions from this syntactic features, and following the example of Calix et al. (2008), we calculate the amount of commas, periods, semi-colons, ellipsis and pair of brackets. With these metrics we can reach conclusions such as the structural complexity of a message (since, for example, juxtaposition structures appear in the presence of some of these scores), the division into sentences of the message or the need for clarification of the text transmitted (for example, by analysing the amount of brackets).

4.5.3. Vocabulary features

In terms of the vocabulary used, we work with the "bag of words" metrics, in other words, we note how many times each different word is used in a message. Of course this is not the only metric that we can categorise as a vocabulary feature and from which we can extract conclusions about the vocabulary used. There are many other which tries to set the parameters of, for instance, the difficult of the vocabulary or its richness. Besides, from the computing of the bag of words, we are able to easily obtain other style marker chosen which also belongs to this category of vocabulary features: the amount of different words in each text, proposed in Ril Gil et al. (2014) and in Corney et al. (2001).

As for the difficulty level, it determines the level of education that someone needs to have if they are to understand the text. There are several indices available to calculate this level, such as the proposed in Dale and Chall (1948), the Gunning Fog Index (Wikipedia contributors, 2020a) or the Flesch-Kincaid index (DuBay, 2004), although the latter is the most commonly documented and cited. The expression which determines the Flesch-Kincaid index is the following:

$$I_{FK} = 1.599\lambda - 1.015\beta - 31.517$$

Where λ is the mean of one-syllable words per 100 words, and β is the mean sentence length measured by the number of words. However, as our spaCy's pretrained Spanish model (see Section 3.2) is not able to divide words by syllables, we determine λ as the mean of words with two or less characters per 100 words.

In respect of the richness of the vocabulary, we have chosen two different metrics. The first that we are going to explain is the one proposed by Honoré (1979), which determines

the richness of te vocabulary based on the total unrepeated words used in the text. The following formula defines it:

$$R_H = \frac{100 \log(M)}{M^2}$$

Where M is the number of different words in the text. However, as Ril Gil et al. (2014) claims, depending on the type of document being analysed, the calculation of R_H has more or less validity (for instance, certain specialist articles, as their nature, requires constant repetition of words). As a consequence of this, another definition of richness of vocabulary is proposed by Yule (2014). This richness marker, that we are going to use as our second richness of vocabulary style marker, is called Yule's characteristic and defined with the following expression:

$$K = \frac{10^4 \left(\sum_{i=1}^{\infty} i^2 V_i - M\right)}{M^2}$$

Where M is the number of different words in the text and V_i is the number of words that appear i times in the document.

From Yule's Characteristic we are able to calculate the Simpson's Index (denoted as D), defined in Simpson (1949). This famous metric is understood as the measurement of diversity based on the change that the two members of an arbitrary chosen pair of word tokens will belong to the same type. To calculate D it is necessary to divide the total number of identical pairs in the sample by the number of all possible pairs, that is to say, what the following expression defines:

$$D = \frac{\sum_{i=1}^{\infty} i(i-1)V_i}{M(M-1)}$$

Where we are maintaining the Yule's Characteristic notation. However, as we have transmitted in advance, it is possible to calculate the Simpson's Index if we know the value of Yule's Characteristic. This relationship is defined by the following expression (and we are going to use it in the implementation in order to speed the computing):

$$10^{-4} K = D\left(1 - \frac{1}{M}\right)$$

Vocabulary distribution can also be measured by using a concept linguists have borrowed from thermodynamics and applied to communication theory: entropy (used in Holmes (1985)). In literary text it is true that with an increase in internal structure, entropy decreases, and with an increase in disorder or randomness, the measure of entropy increases. The expression for the entropy of a system (vocabulary in this case) is:

$$H = -\sum_{i=1}^{\infty} p_i \log(p_i)$$

Where p_i is the probability of appearance of the ith lemma (found by dividing the number of occurrences of that lemma by the total number of words in the text). Due to the value will change according to how much text is analysed, the formula may be refined in order that works of different length may be compared. In this way, as it is proposed in Holmes (1985), the following expression determines absolute diversity for any length text as 100, while absolute uniformity remains zero:

$$H = -100 \sum_{i=1}^{\infty} p_i \frac{\log(p_i)}{\log(M)}$$

In addition to the words distribution features (which are the bag of words and the amount of different words), the level of difficulty, the richness of vocabulary (which is measured by the formula proposed in Honoré (1979) and the Yule's Characteristic), the diversity (represented by the Simpson's Index) and the internal structure of the vocabulary (which is measured by the entropy), we have defined other four style markers which also allow us to extract conclusions about some feature of the vocabulary of the message. First of these is the most popular and old style marker: the mean word length. Researches as Ril Gil et al. (2014) claim that it is "directly connected with the richness of the author's vocabulary and measures his or her ability to use complex words", due to it is considered that complex words are formed by three or more syllables that do not represent proper nouns, prefixes, suffixes or compound words. Thus, Ril Gil et al. (2014) propose an expression similar to the following one in order to calculate it:

$$L_W = \frac{\sum_{i=1}^{\infty} i * C_i}{N} \cdot 100$$

Where C_i is the number of words with i characters and N is the number of words used. This formula is analogous to the expression proposed by Ril Gil et al. (2014), except that with the one that we have presented the punctuation marks are removed from the numerator.

The second of these writing style metrics is the measurement of words length frequency distribution, that is to say, how many words with one character appears in the document, with two characters and so on up to the length of the longest word. Despite of being strongly influenced by the language, it is used in researches as Corney et al. (2001) and Kemp (1976), as it is claimed inAllen (1974), "Each writer, however, will have his own curve, so that although English (and German) texts in general peak at three letters, the writings of John Stuart Mill peak at two and those of Shakespeare peak at four". Our interest will then focus on checking whether, in addition to depending on the author, this metric varies according to the recipient of the e-mail.

The rest of vocabulary features are related to the stop words present in the text. The simplest of those metrics is the style marker which consists of calculating the total number of stop words (denoted as T_S). On the other hand, as it is proposed in Ril Gil et al. (2014), we will calculate the stop words ratio, which is defined with the following expression:

$$S_W = \frac{T_S}{N} \cdot 100$$

4.5.4. Structural features

We will denote by structural features those characteristics that we obtain directly from the construction of the analysed text. Some of these metrics are as simple as the total number of characters in the body of the e-mail or the absolute number of words in the e-mail, both used in researches such as in Corney et al. (2001) and Ril Gil et al. (2014).

Most of these features are sentence length dependent. Both Tallentire (1972) and Kjetsaa (1979) agree that summary measures such as average sentence-lengths are of little use in stylometry studies but distributions of sentence-lengths can be useful, even on their own. Taking into account the above, we will find both the distribution of the length of

the sentences (calculated in number of characters and number of words) and the average length of the sentences in a message found by the number of words, as it is proposed in Corney et al. (2001). For the first one, we are going to store the number of sentence with length one, two, three and so on up to the length of the longest one, by measuring it using both the number of characters and the number of words.

4.5.5. Relationship between metrics and their implementation

Every explained style markers is stored as an attribute of the *Metrics* class. The relationship between them and the presented style features and the categorisation of these metrics, is exposed in Table 4.1.

Feature Category	Field name	Explanation
Part of speech	ADV	Number of adverbs
	VERB	Number of verbs
	ADJ	Number of adjectives
	ADP	Number of prepositions
	NOUN	Number of nouns
	PRON	Number of pronous
	DET	Number of determinants
	CONJ	Number of conjunctions
	verbAdjectiveRatio	Verb-adjective ratio
	detPronRatio	Determinant-pronouns ratio
Punctuation	numCommas	Number of commas
	numDots	Number of periods
	numSemiColon	Number of semi-colons
	num3Dots	Number of ellipsis
	numBrackets	Number of pair of brackets
Vocabulary	wordsAppearance	Bag of words
	numDifWords	Number of different words
	difficultyLevel	Modified Flesch-Kincaid index (I_{FK})
	richnessVocab	Honoré (1979) vocabulary richness (R_H)
	richnessYule	Yule's characteristic (K)
	SimpsonIndex	Simpson's index (D)
	entropy	Entropy (H)
	meanWordLen	Mean word length (L_W)
	wordLength	Word length distribution
	numStopWords	Number of stop words
	stopRatio	Percentage of stop words (S_W)
Structural	charLength	Number of characters
	numWords	Number of words
	sentLength	Sentence length distribution (number of characters)
	sentNumWords	Sentence length distribution (number of words)
	meanSentLen	Average word count per sentence

Table 4.1: Classification of the style markers

There is only one attribute that we have not mentioned and does not appear in Table 4.1: *metricsSentences*. This attribute is a list of as many items as there are sentences in the document and each of the has the following dictionary structure:

```
{
    'numStopWords': int,
    'ADV': int,
    'VERB': int,
    'ADJ': int,
    'ADP': int,
    'NOUN': int,
    'PRON': int,
    'DET': int,
    'CONJ': int,
    'numCommas': int,
    'numDots': int,
    'numSemiColon': int,
    'num3Dots': int,
    'numBrackets': int,
    'wordLength':
    {
        '1': int,
        '2': int,
        ...
    }
    'charLength': int,
    'numWords': int,
    'stopRatio': float,
    'meanWordLen': float
}
```

With this dictionary, we calculate these metrics of each sentence, instead of evaluating them on the entire message.

4.6. Analyser class

The *Analyser* class in charge of manages all the phase in the pipeline, in other words, it sends to each module the required input in order to obtain is output. Besides, as it has been explained in Section 4.4 where the typographic correction module was presented, it asks the user the necessary information for the purpose of detecting and correcting, if it is required, the found typographic errors. In addition to it, as we can see in Figure 4.7, this class is able to store information in the database and extract it through the *SessionTypoError* class.

The *Analyser*'s class constructor has an special interest in this system, due to it chooses the type of extraction that is going to be executed: a message extraction or a thread extraction. With the purpose of making this choice, the *get* method of the labels resource is invoked in order to obtain the value of the fields *messagesTotal* and *threadsTotal* of the *SENT* label structure (see Section 2.1.5.3). With this values tha *Analyser* is able to calculate the quota units cost (as we see in Section 4.2) of each type of extraction (this task is carried out by the *__get_res_cost* method) and choose the one which minimises it. After deciding it, it gives effect to the choice by creating the corresponding object with the type of extraction (*MessageExtractor* or *ThreadExtractor*).

Once an *Analyser* object is created, the entire system can start its execution by calling *analyse* function (the web services of the modules which represent the last three phases in the pipeline have to be running). During the execution of this method, the only module that will require the attention of the *Analyser* is the typographic correction module. The

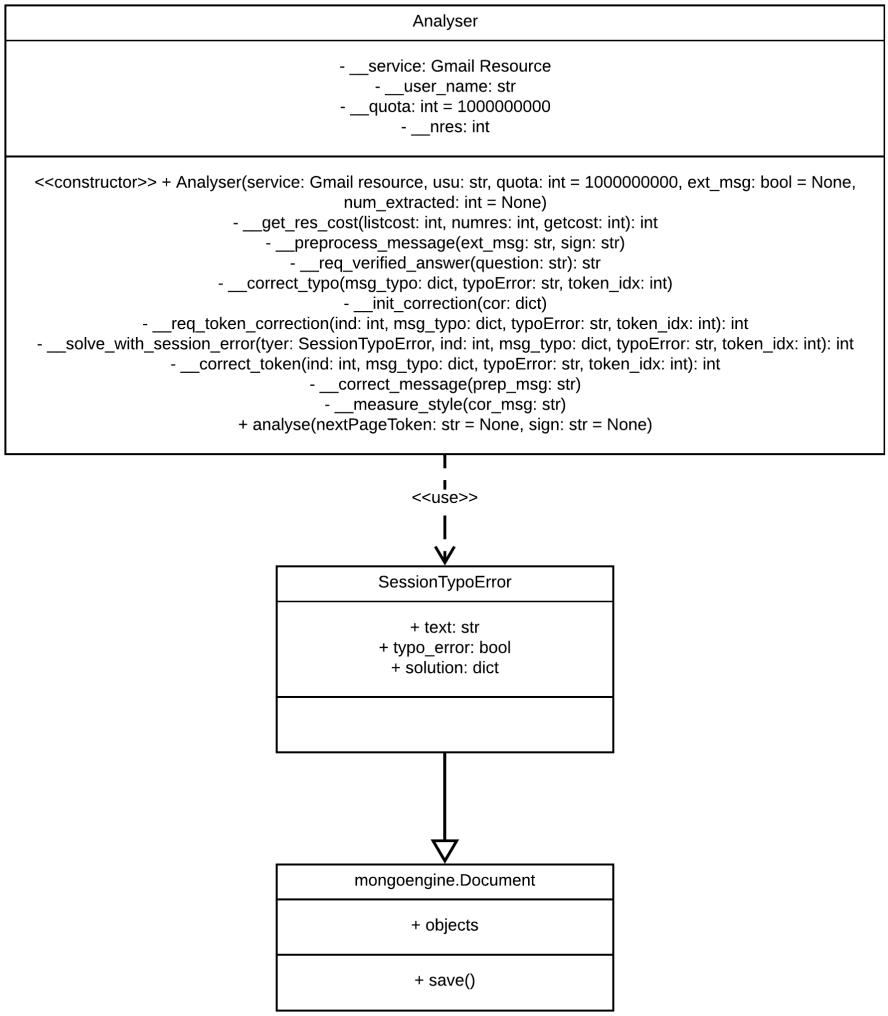


Figure 4.7: UML class diagram of the Analyser

rest of the packages will only need this class to provide the input messages and collect their output structures.

If the *TypoCorrector* detects a typographic error, the first action that the *Analyser* carries out is searching in the database if that word was previously found and corrected it. For this purpose, the *SessionTypoError* stores the common typographic errors that a user made (this collection is dropped at the beginning of each execution) and how to solve it (the *solution* field has the same structure as the items of the list *corrections* of the structure given by the *TypoCorrector*). If it is a real typographic error, the *typo_error* field will take the value *True* and in the *solution* dictionary the text that should replace the word is stored. If it is not, the *solution* dictionary will be appended to the *corrections* list and the message will go on being corrected.

In the case that this typographic error is not stored, the *Analyser* will ask the user whether the message has to be discarded (this option allows the user to remove from the pipeline, for instance, e-mail written in other languages). If it is discarded, the *Analyser* sends then next message ready to be corrected to the typographic correction module. If it is not discarded, the *__req_token_correction* method is invoked. This function will ask

the user whether the word is a real typographic error. If it is, there are two solutions: remove the token from the text or rewrite it. Either way, at the end the user is going to answer the question: Do you want to save this information for this session? It will decide whether the solution is inserted in the collection managed by *SessionTypoError* in case the same error is detected again.

If it is not a real typographic error, the user will answer some questions about the token: such as whether it is an url, an e-mail, a punctuation mark, a stop word, what its part of speech is and what its lemma is. Then, this information is appended to the *corrections* list and ask the user whether this information is stored in *Correction* collection for the future or in *SessionTypeError* collection for this session.

Once the detected word is managed, the *Analyser* resends the message to the typographic correction module in order to go on correcting it.

4.7. Execution behaviour

After executing it with the Gmail account that is going to be analysed (the author's Gmail account), of the 1084 e-mails extracted, 921 were measured, which represents approximately the 84.96% of the total. In this execution, all the 1084 messages were correctly preprocessed, but 163 were discarded in the typographic correction phase. Some of them were discarded because, after the preprocess, they were missing text, whereas the rest of them did not pass this phase due to its language (there were e-mails written in English) or, a minority, because they had a not interested message body for the analysis of the writing style (for example we found some e-mails whose only text was an url).

Chapter 5

Conclusiones y Trabajo Futuro

Conclusiones del trabajo y líneas de trabajo futuro.

Antes de la entrega de actas de cada convocatoria, en el plazo que se indica en el calendario de los trabajos de fin de máster, el estudiante entregará en el Campus Virtual la versión final de la memoria en PDF. En la portada de la misma deberán figurar, como se ha señalado anteriormente, la convocatoria y la calificación obtenida. Asimismo, el estudiante también entregará todo el material que tenga concedido en préstamo a lo largo del curso.

Bibliography

And thus I clothe my naked villany
With old odd ends stolen out of holy writ;
And seem a saint, when most I play the devil.

Richard III, Act I Scene 3
William Shakespeare

Abbasi, A. and Chen, H. Applying authorship analysis to extremist-group web forum messages. *IEEE Intelligent Systems*, Vol. 20(5), 67–75, 2005.

Abbasi, A. and Chen, H. Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Transactions on Information Systems (TOIS)*, Vol. 26(2), 1–29, 2008.

Allen, J. R. Methods of author identification through stylistic analysis. *The French Review*, Vol. 47(5), 904–916, 1974.

Antosch, F. The diagnosis of literary style with the verb-adjective ratio. *Statistics and style*, Vol. 1, 1969.

Apte, C., Damerau, F., Weiss, S. et al. *Text mining with decision rules and decision trees*. Citeseer, 1998.

Argamon, S. Interpreting burrows's delta: Geometric and probabilistic foundations. *Literary and Linguistic Computing*, Vol. 23(2), 131–147, 2008.

Argamon, S., Koppel, M. and Avneri, G. Routing documents according to style. In *First International workshop on innovative information systems*, 85–92. 1998.

Argamon, S., Šarić, M. and Stein, S. S. Style mining of electronic messages for multiple authorship discrimination: first results. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 475–480. 2003.

Argamon-Engelson, S., Koppel, M. and Avneri, G. Style-based text categorization: What newspaper am i reading. In *Proc. of the AAAI Workshop on Text Categorization*, 1–4. 1998.

Baayen, H., van Halteren, H., Neijt, A. and Tweedie, F. An experiment in authorship attribution. In *6th JADT*, Vol. 1, 69–75. 2002.

Baayen, H., Van Halteren, H. and Tweedie, F. Outside the cave of shadows: Using syntactic annotation to enhance authorship attribution. *Literary and Linguistic Computing*, Vol. 11(3), 121–132, 1996.

Baayen, R., Tweedie, F., Neijt, A., Halteren, H. v. and Krebbers, L. Back to the cave of shadows: Stylistic fingerprints in authorship attribution. 2000.

BBC news (3rd July 2018). Gmail messages 'read by human third parties'. *Technology*, 2018. `https://www.bbc.com/news/technology-44699263`.

Binongo, J. N. G. and Smith, M. W. A. The application of principal component analysis to stylometry. *Literary and Linguistic Computing*, Vol. 14(4), 445–466, 1999.

Borenstein, N. and Freed, N. Mime (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. Tech. Rep. RFC 1521, Internet Engineering Task Force (IETF), 1993.

Brainerd, B. *Weighting Evidence in Language and Literature: A Statistical Approach*. University of Toronto Press, 1974.

Brocardo, M. L., Traore, I., Saad, S. and Woungang, I. Authorship verification for short messages using stylometry. In *2013 International Conference on Computer, Information and Telecommunication Systems (CITS)*, 1–6. IEEE, 2013.

Burrows, J. 'delta': a measure of stylistic difference and a guide to likely authorship. *Literary and linguistic computing*, Vol. 17(3), 267–287, 2002.

Burrows, J. F. Computers and the study of literature. *Computers and written texts*, 167–204, 1992.

Calix, K., Connors, M., Levy, D., Manzar, H., MCabe, G. and Westcott, S. Stylometry for e-mail author identification and authentication. *Proceedings of CSIS research day, Pace University*, 1048–1054, 2008.

Canales, O., Monaco, V., Murphy, T., Zych, E., Stewart, J., Castro, C. T. A., Sotoye, O., Torres, L. and Truley, G. A stylometry system for authenticating students taking online tests. *P. of Student-Faculty Research Day, Ed., CSIS. Pace University*, 2011.

Chaski, C. E. Empirical evaluations of language-based author identification techniques. *Forensic Linguistics*, Vol. 8, 1–65, 2001.

Chen, X., Hao, P., Chandramouli, R. and Subbalakshmi, K. Authorship similarity detection from email messages. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, 375–386. Springer, 2011.

Cheng, N., Chandramouli, R. and Subbalakshmi, K. Author gender identification from text. *Digital Investigation*, Vol. 8(1), 78–88, 2011.

Choi, J. D., Tetreault, J. and Stent, A. It depends: Dependency parser comparison using a web-based evaluation tool. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 387–396. 2015.

Cook, B. and Messina, C. OAuth 2.0. https://oauth.net/2/, 2019a. [Online; accessed 27-September-2019].

Cook, B. and Messina, C. OAuth 2.0 Authorization Code Exchange. https://www.oauth.com/oauth2-servers/pkce/authorization-code-exchange/, 2019b. [Online; accessed 27-September-2019].

Cook, B. and Messina, C. OAuth 2.0 Scope. https://oauth.net/2/scope/, 2019c. [Online; accessed 27-September-2019].

Corney, M. W. *Analysing e-mail text authorship for forensic purposes*. PhD thesis, Queensland University of Technology, 2003.

Corney, M. W., Anderson, A. M., Mohay, G. M. and de Vel, O. Identifying the authors of suspect email. 2001.

Craig, H. Authorial attribution and computational stylistics: If you can tell authors apart, have you learned anything about them? *Literary and Linguistic Computing*, Vol. 14(1), 103–113, 1999.

Crispin, M. Internet message access protocol - version 4rev1. Tech. Rep. RFC 3501, University of Washington, 2003.

Crocker, D. H. Standard for the format of arpa internet text messages. Tech. Rep. RFC 822, Dept. of Electrical Engineering, University of Delaware, 1982.

Daelemans, W., De Clercq, O. and Hoste, V. STYLENE: an environment for stylometry and readability research for Dutch. In *CLARIN in the Low Countries*, 195–210. Ubiquity Press, 2017.

Dale, E. and Chall, J. S. A formula for predicting readability: Instructions. *Educational research bulletin*, 37–54, 1948.

De Morgan, S. E. and De Morgan, A. *Memoir of Augustus De Morgan*. Longmans, Green, and Company, 1882.

De Vel, O., Anderson, A., Corney, M. and Mohay, G. Mining e-mail content for author identification forensics. *ACM Sigmod Record*, Vol. 30(4), 55–64, 2001.

Diederich, J., Kindermann, J., Leopold, E. and Paass, G. Authorship attribution with support vector machines. *Applied intelligence*, Vol. 19(1-2), 109–123, 2003.

DuBay, W. H. The principles of readability. *Online Submission*, 2004.

Eder, M. Style-markers in authorship attribution: a cross-language study of the authorial fingerprint. *Studies in Polish Linguistics*, Vol. 6(1), 2011.

Eder, M., Rybicki, J. and Kestemont, M. Stylometry with r: a package for computational text analysis. *R journal*, Vol. 8(1), 2016.

Ellegard, A. A statistical method for determining authorship: the junius letter. *Gothenburg studies in Engilsh*, Vol. 13, 1769–1772, 1962.

Feng, S., Banerjee, R. and Choi, Y. Syntactic stylometry for deception detection. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers-Volume 2*, 171–175. Association for Computational Linguistics, 2012.

Freed, N. and Borenstein, N. Mime (multipurpose internet mail extensions). Tech. Rep. RFC 1341, Internet Engineering Task Force (IETF), 1992.

Freed, N. and Borenstein, N. Multipurpose internet mail extensions (mime) part five: Conformance criteria and examples. Tech. Rep. RFC 2049, Internet Engineering Task Force (IETF), 1996a.

Freed, N. and Borenstein, N. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. Tech. Rep. RFC 2045, Internet Engineering Task Force (IETF), 1996b.

Freed, N. and Borenstein, N. Multipurpose internet mail extensions (mime) part two: Media types. Tech. Rep. RFC 2046, Internet Engineering Task Force (IETF), 1996c.

Freed, N. and Klensin, J. Media type specifications and registration procedures. Tech. Rep. RFC 4288, Internet Engineering Task Force (IETF), 2005a.

Freed, N. and Klensin, J. Multipurpose internet mail extensions (mime) part four: Registration procedures. Tech. Rep. RFC 4289, Internet Engineering Task Force (IETF), 2005b.

Fucks, W. and Lauter, J. Mathematische analyse des literarischen stils.–mathematik und dichtung. versuche zur frage einer exakten literaturwissenschaft. 1965.

Gatt, A. and Krahmer, E. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *Journal of Artificial Intelligence Research*, Vol. 61, 65–170, 2018.

Gellens, R. The text/plain format parameter. Tech. Rep. RFC 2646, Internet Engineering Task Force (IETF), 1999.

Google. Gmail api | google developers. `https://developers.google.com/gmail/api`, 2019a. [Online; accessed 17-October-2019].

Google. google_auth_oauthlib.flow module. `https://google-auth-oauthlib.readthedocs.io/en/latest/reference/google_auth_oauthlib.flow.html`, 2019b. [Online; accessed 27-September-2019].

Google. google.auth.transport package. `https://google-auth.readthedocs.io/en/stable/reference/google.auth.transport.html#google.auth.transport.Request`, 2019c. [Online; accessed 27-September-2019].

Google. google.oauth2.credentials module. `https://google-auth.readthedocs.io/en/stable/reference/google.oauth2.credentials.html#google.oauth2.credentials.Credentials`, 2019d. [Online; accessed 27-September-2019].

Google. OAuth 2.0 Scopes for Google APIs. `https://developers.google.com/identity/protocols/googlescopes`, 2019e. [Online; accessed 27-September-2019].

Google. Using OAuth 2.0 to Access Google APIs. `https://developers.google.com/identity/protocols/OAuth`, 2019f. [Online; accessed 27-September-2019].

Gregorio, J. googleapiclient.discovery. `https://googleapis.github.io/google-api-python-client/docs/epy/googleapiclient.discovery-module.html#build`, 2019. [Online; accessed 23-September-2019].

Gruner, S. and Naven, S. Tool support for plagiarism detection in text documents. In *Proceedings of the 2005 ACM symposium on Applied computing*, 776–781. 2005.

Guide, S. *Red Hat Enterprise Linux 4: Reference Guide*. Red Hat Inc., 2005. `http://web.mit.edu/rhel-doc/OldFiles/4/RH-DOCS/rhel-rg-en-4/index.html`.

Hardt, D. The oauth 2.0 authorization framework. Tech. Rep. RFC 6749, Internet Engineering Task Force (IETF), 2012.

Holmes, D. I. The analysis of literary style—a review. *Journal of the Royal Statistical Society: Series A (General)*, Vol. 148(4), 328–341, 1985.

Holmes, D. I. The evolution of tylometry in humanities. *Literary and Linguistic Computing*, Vol. 13(3), 1998.

Holmes, D. I. and Forsyth, R. S. The federalist revisited: New directions in authorship attribution. *Literary and Linguistic computing*, Vol. 10(2), 111–127, 1995.

Homem, N. and Carvalho, J. P. Authorship identification and author fuzzy "fingerprints". In *2011 Annual Meeting of the North American Fuzzy Information Processing Society*, 1–6. IEEE, 2011.

Honnibal, M. and Johnson, M. An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, 1373–1378. 2015.

Honoré, A. Some simple measures of richness of vocabulary. *Association for literary and linguistic computing bulletin*, Vol. 7(2), 172–177, 1979.

Hoover, D. L. Delta prime? *Literary and Linguistic Computing*, Vol. 19(4), 477–495, 2004a.

Hoover, D. L. Testing burrows's delta. *Literary and linguistic computing*, Vol. 19(4), 453–475, 2004b.

Iqbal, F., Binsalleeh, H., Fung, B. C. and Debbabi, M. Mining writeprints from anonymous e-mails for forensic investigation. *digital investigation*, Vol. 7(1-2), 56–64, 2010.

Josefsson, S. The base16, base32, and base64 data encodings. Tech. Rep. RFC 4648, Internet Engineering Task Force (IETF), 2006.

Josefsson, S. and Leonard, S. Textual encodings of pkix, pkcs, and cms structures. Tech. Rep. RFC 7468, Internet Engineering Task Force (IETF), 2015.

Juola, P. Becoming jack london. *Journal of Quantitative Linguistics*, Vol. 14(2-3), 145–147, 2007.

Kemp, K. W. Personal observations on the use of statistical methods in quantitative linguistics. In *The Computer in Literary and Linguistic Studies (Proceeding, Third International Symposium)*, 59–77. 1976.

Kjell, B., Woods, W. A. and Frieder, O. Discrimination of authorship using visualization. *Information processing & management*, Vol. 30(1), 141–150, 1994.

Kjetsaa, G. And quiet flows the don through the computer. *Association for Literary and linguistic computing Bulletin*, Vol. 7, 248–256, 1979.

Klensin, J. Simple mail transfer protocol. Tech. Rep. RFC 5321, Internet Engineering Task Force (IETF), 2008.

Koppel, M., Akiva, N. and Dagan, I. Feature instability as a criterion for selecting potential style markers. *Journal of the American Society for Information Science and Technology*, Vol. 57(11), 1519–1525, 2006.

Koppel, M. and Schler, J. Exploiting stylistic idiosyncrasies for authorship attribution. In *Proceedings of IJCAI'03 Workshop on Computational Approaches to Style Analysis and Synthesis*, Vol. 69, 72–80. 2003.

Kucukyilmaz, T., Cambazoglu, B. B., Aykanat, C. and Can, F. Chat mining: Predicting user and message attributes in computer-mediated communication. *Information Processing & Management*, Vol. 44(4), 1448–1466, 2008.

Mendenhall, T. C. The characteristic curves of composition. *Science*, Vol. 9(214), 237–249, 1887.

Mihalcea, R. and Strapparava, C. The lie detector: Explorations in the automatic recognition of deceptive language. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, 309–312. Association for Computational Linguistics, 2009.

Moore, K. Multipurpose internet mail extensions (mime) part three: Message header extensions for non-ascii text. Tech. Rep. RFC 2047, Internet Engineering Task Force (IETF), 1996.

Mosteller, F. and Wallace, D. L. *Applied Bayesian and classical inference: the case of the Federalist papers*. Springer Science & Business Media, 1964.

Myers, J., Mellon, C. and Rose, M. Post office protocol - version 3. Tech. Rep. RFC 1939, Dover Beach Consulting, Inc., 1996.

Nelson, S. and Parks, C. The model primary content type for multipurpose internet mail extensions. Tech. Rep. RFC 2077, Internet Engineering Task Force (IETF), 1997.

Ng, H. T., Goh, W. B. and Low, K. L. Feature selection, perceptron learning, and a usability case study for text categorization. In *Proceedings of the 20th annual international ACM SIGIR conference on Research and development in information retrieval*, 67–73. 1997.

Ott, M., Choi, Y., Cardie, C. and Hancock, J. T. Finding deceptive opinion spam by any stretch of the imagination. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, 309–319. Association for Computational Linguistics, 2011.

Pennebaker, J. W., Boyd, R. L., Jordan, K. and Blackburn, K. The development and psychometric properties of liwc2015. Tech. rep., University of Texas at Austin, 2015.

Postel, J. B. Simple mail transfer protocol. Tech. Rep. RFC 821, Information Sciences Institute, University of Southern California, 1982.

Resnick, P. Internet message format. Tech. Rep. RFC 2822, Internet Engineering Task Force (IETF), 2001.

Resnick, P. Internet message format. Tech. Rep. RFC 5322, Qualcomm Incorporated, 2008.

Reynolds, J. K. Post office protocol. Tech. Rep. RFC 918, Information Sciences Institute, 1984.

Ril Gil, Y., Toll Palma, Y. d. C. and Lahens, E. F. Determination of writing styles to detect similarities in digital documents. *RUSC: Revista de Universidad y Sociedad del Conocimiento*, Vol. 11(1), 2014.

Rudman, J. The state of authorship attribution studies: Some problems and solutions. *Computers and the Humanities*, Vol. 31(4), 351–365, 1997.

Sahami, M., Dumais, S., Heckerman, D. and Horvitz, E. A bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the 1998 workshop*, Vol. 62, 98–105. Madison, Wisconsin, 1998.

Schwartz, H. A., Eichstaedt, J. C., Kern, M. L., Dziurzynski, L., Ramones, S. M., Agrawal, M., Shah, A., Kosinski, M., Stillwell, D., Seligman, M. E. et al. Personality, gender, and age in the language of social media: The open-vocabulary approach. *PloS one*, Vol. 8(9), e73791, 2013.

Sheika, F. A. and Inkpen, D. Learning to classify documents according to formal and informal style. *Linguistic Issues in Language Technology*, Vol. 8(1), 1–29, 2012.

Simpson, E. H. Measurement of diversity. *nature*, Vol. 163(4148), 688–688, 1949.

Smith, M. W. Recent experience and new developments of methods for the determination of authorship. *ALLC BULL.*, Vol. 11(3), 73–82, 1983.

Somers, H. Statistical methods in literary analysis. *The computer and literary style*, 128–140, 1966.

Stamatatos, E. A survey of modern authorship attribution methods. *Journal of the American Society for information Science and Technology*, Vol. 60(3), 538–556, 2009.

Stamou, C. Stylochronometry: Stylistic development, sequence of composition, and relative dating. *Literary and Linguistic Computing*, Vol. 23(2), 181–199, 2007.

Summers, K. Analysing for authorship: A guide to the cusum technique. 1999.

Tallentire, D. *An appraisal of methods and models in computational stylistics, with particular reference to author attribution.*. PhD thesis, University of Cambridge, 1972.

Thisted, R. and Efron, B. Did shakespeare write a newly-discovered poem? *Biometrika*, Vol. 74(3), 445–455, 1987.

Thomson, R. and Murachver, T. Predicting gender from electronic discourse. *British Journal of Social Psychology*, Vol. 40(2), 193–208, 2001.

Troost, R., Dorner, S. and Moore, K. Communicating presentation information in internet messages: The content-disposition header field. Tech. Rep. RFC 2183, Internet Engineering Task Force (IETF), 1997.

Tweedie, F. J. and Baayen, R. H. How variable may a constant be? measures of lexical richness in perspective. *Computers and the Humanities*, Vol. 32(5), 323–352, 1998.

Tweedie, F. J., Singh, S. and Holmes, D. I. Neural network applications in stylometry: The federalist papers. *Computers and the Humanities*, Vol. 30(1), 1–10, 1996.

Wikipedia contributors. Base64 — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Base64`, 2019a. [Online; accessed 17-October-2019].

Wikipedia contributors. Email — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Email&oldid=954020305`, 2019b. [Online; accessed 21-September-2019].

Wikipedia contributors. Mime — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=MIME&oldid=916901691`, 2019c. [Online; accessed 23-September-2019].

Wikipedia contributors. Privacy-enhanced mail — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail`, 2019d. [Online; accessed 23-September-2019].

Wikipedia contributors. Quoted-printable — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Quoted-printable`, 2019e. [Online; accessed 23-September-2019].

Wikipedia contributors. Gunning fog index — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Gunning_fog_index&oldid=946173273`, 2020a. [Online; accessed 31-May-2020].

Wikipedia contributors. Stylometry — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Stylometry&oldid=957882878`, 2020b. [Online; accessed 28-May-2020].

Williams, C. B. *Style and vocabulary: numerical studies*. Griffin, 1970.

Yule, C. U. *The statistical study of literary vocabulary*. Cambridge University Press, 2014.

Yule, G. U. On sentence-length as a statistical characteristic of style in prose: With application to two cases of disputed authorship. *Biometrika*, Vol. 30(3/4), 363–390, 1939.

Zhao, Y. and Zobel, J. Searching with style: Authorship attribution in classic literature. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, 59–68. Australian Computer Society, Inc., 2007.

Appendix A

Título del Apéndice A

Contenido del apéndice

Appendix B

Título del Apéndice B

Este texto se puede encontrar en el fichero Cascaras/fin.tex. Si deseas eliminarlo, basta con comentar la línea correspondiente al final del fichero TFMTeXiS.tex.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes