Corrección automática de errores de OCR en documentos semi-estructurados

Trabajo especial de la carrera Licenciatura en Ciencias de la Computación

Autor: Pablo A. Paliza

Directora: Paula Estrella

7 de diciembre de 2016

Facultad de Matemática, Astronomía y Física

Universidad Nacional de Córdoba

Este trabajo se distribuye bajo una Licencia Creative Commons Atribución 2.5 Argentina

Índice General

Resumen **vi**

1 Introducción **7**

2 Trabajos Relacionados **9**

2.1 Antes del OCR 9

2.2 Durante el OCR 9

2.3 Después del OCR 10

2.4 Algunos métodos en detalle 11

2.4.1 Distancia de Levenshtein 11

2.4.2 Distancia de Levenshtein con peso 12

2.4.3 Hash de Anagramas (Anagram Hashing) 13

2.4.4 Clave de similitud (Similarity key) 14

3 Metodología adoptada para documentos del APM **15**

3.1 Análisis de la colección 15

3.2 Propuesta de Procesamiento 17

3.3 Algoritmo Base 19

3.3.1 Cómo Funciona: Teoría de Probabilidad 19

3.3.2 Cómo Funciona: Python 20

3.4 Mejoras sobre el algoritmo Base 22

3.5 Construcción Incremental de Diccionarios 24

3.6 Resultados 25

4 Conclusiones **28**

4.1 Fortalezas y limitaciones 28

4.2 Trabajo futuro 29

Bibliografía xxxii

A Anexo xxxiii

A.1 Programa: Manual de Uso xxxiii

A.2 Programa: Dependencia Archivos xxxiv

A.3 Programa: Ejemplos de Salida: ocr_corrector.py xxxiv

A.4 Programa: Ejemplos de Salida: calculate_wer.py xxxv

A.5 Programa: Funciones Principales xxxviii

A.6 Documentos: Palabras más Frecuentes xxxix

Agradecimientos

Agradezco en primer lugar a mi directora, la Dra. Paula Estrella, quien con dedicación y entusiasmo me incentivó a avanzar constantemente e hizo posible este trabajo.
A mis padres, hermano, familia y amigos que me apoyaron durante mis estudios.

Resumen

En este trabajo se presenta la tarea realizada para corregir automáticamente texto generado por un OCR desde un archivo digital realizado para preservar documentos creados durante la dictadura militar en Argentina entre los años 1976 y 1983, también conocido como el Proceso de Reorganización Nacional.

Estos documentos son bastantes únicos en su estructura, contenido y estado de conservación, haciéndolos una colección desafiante. Se adoptó un enfoque de post-procesamiento, en el que se creó un diccionario especifico y la corrección del texto de salida del OCR se basó en distancias de edición y características tipográficas. En un conjunto de test representativo se logró corregir aproximadamente el 30% de los errores.

Abstract

This paper presents the work done to automatically correct OCRed text from a digital archive setup to preserve documents created during Argentina's 1976-1983 dictatorship, also known as the National Reorganization Process (Proceso de Reorganización Nacional).

These documents are quite unique in their structure, content and state of preservation, making it a challenging corpus. A postprocessing approach was adopted, in which a specific dictionary was created and the correction of the OCRed text was based on edit distances and typographical characteristics of the text. On a representative test set the correction of about 30% of the OCR errors was achieved.

Palabras claves

Corrección automática de OCR, construcción de diccionarios, archivos digitales, evaluación de desempeño.

Clasificación

I.7 [Computing Methodologies]: Document and Text Processing; I.7.5 [Document and Text Processing]: Document Capture—Document analysis, Optical character recognition (OCR)

Capítulo 1

Introducción

La creación y el mantenimiento de archivos digitales es una práctica común para la preservación y la explotación de información en varios campos. La tecnología fundamental que permite la digitalización de documentos, llamada reconocimiento óptico de caracteres, generalmente conocido como OCR por su sigla en inglés (Optical Character Recognition), es un proceso que combina hardware y software, el cual identifica automáticamente a partir de una imagen símbolos o caracteres que pertenecen a un determinado alfabeto, para luego almacenarlos en forma de datos. Así podremos interactuar con éstos mediante un programa de edición de texto, búsqueda, etc. En el presente, cada vez más organizaciones están digitalizando sus archivos (por ejemplo, documentos impresos, fotografías, grabaciones, etc.) en repositorios con el objetivo de lograr un bajo mantenimiento y alto uso de la información luego de un esfuerzo de preparación inicial. Los archivos digitales también incrementan la posibilidad de automatizar varias etapas de procesamiento para facilitar la manipulación de la información que contienen buscando o extrayendo datos específicos, en lugar de manipular los documentos físicos que tienen que procesarse manualmente.

Un archivo de gran importancia en el ámbito local es el "Archivo Provincial de la Memoria de Córdoba" (APM), creado en 2006 por la legislatura de la Provincia de Córdoba a través de la ley 9286, llamada "Ley de la Memoria". Actualmente, el APM contiene una gran cantidad de material digitalizado, que consiste principalmente de documentos producidos por las fuerzas de seguridad y defensa (policía, servicios de inteligencia, fuerzas armadas, servicios de prisión, gendarmería, etc.) durante los periodos no democráticos de las décadas del 70 y 80. Sin embargo la explotación de este material está limitada por los siguientes factores:

- La gran cantidad de documentos existentes, que hacen muy difícil la exploración y búsqueda de información.
- La diversidad de características, que demandan métodos específicos para manipular cada clase de documento en particular.
- El avanzado estado de deterioración de los documentos originales, haciendo difícil el escaneo y almacenamiento.

En orden de poder superar estos obstáculos, se ha comenzado a trabajar bajo un acuerdo de cooperación firmado entre la Universidad Nacional de Córdoba y el APM, donde el objetivo general es desarrollar herramientas para el procesamiento automático del material que sean suficientemente generales para aplicarse a otros tipos de archivos. En este trabajo nos enfocaremos en la post-corrección automática de errores de OCR en una colección específica de documentos preservados en el APM.

La información que puede obtenerse de estos boletines es muy variada, tal como: bajas del personal superior, nombramientos y pases del personal superior y subalterno; comisiones en el exterior; retiros; información vinculada a los diferentes cursos de capacitación del personal militar del Ejército y otras fuerzas armadas y de seguridad. Estos boletines son una de las fuentes principales para poder rastrear el movimiento del personal militar de no contar con otros documentos.

Estructura de la tesis:

El capítulo 2 brinda un repaso sobre los trabajos relacionados que utilizan diferentes estrategias para corregir errores de OCR. Se da una inspección general de las técnicas que se utilizan para corregir estos errores agrupados por los métodos que se realizan antes del OCR, durante el OCR y después del OCR. Además, se explicarán algunos métodos en detalle. En el capítulo 3 se hará un análisis de la colección que se intentará corregir, la discusión del método que se utilizará para realizar la corrección, la construcción incremental de los diccionarios y los resultados obtenidos. Finalmente, la tesis concluye en el capítulo 4, en donde se muestran las fortalezas y limitaciones y finaliza con un vistazo del trabajo futuro.

Capítulo 2

Trabajos Relacionados

Se ha hecho mucho trabajo en esta área, abordando la corrección del OCR en diferentes puntos del procesamiento (antes, durante y después del OCR) y utilizando una amplia variedad de recursos (diccionarios, desambiguación lingüística, etiquetado gramatical, etc.). Debido a esto, este capítulo sólo puede dar un resumen general de los diferentes enfoques disponibles.

2.1 Antes del OCR

El procesamiento de imágenes puede contribuir en gran medida para mejorar la exactitud del OCR, por ejemplo, mejorando la iluminación no uniforme de las imágenes [19]. Este método es muy útil cuando las imágenes a procesar están deterioradas, ya que al mejorar la calidad de la imagen el motor de OCR puede identificar mejor el texto de la imagen. Otro enfoque que se puede realizar antes de procesar la imagen con el OCR es la identificación de texto manuscrito en partes de la imagen [6, 18, 17], ya que en general este tipo de texto no es bien manejado por los programas de OCR.

2.2 Durante el OCR

Si se puede acceder a la programación interna del software de OCR, puede ser posible incrementar la exactitud durante el proceso de reconocimiento. Los motores de OCR de código abierto brindan la posibilidad de estudiar cómo funcionan estos programas y se pueden realizar modificaciones para que se adecúen al corpus que se quiere procesar. Por ejemplo, adaptándolo para alfabetos no estándares o diferentes tipos de fuentes. También es posible entrenar estos motores con textos similares o

una fracción del corpus que se quiere procesar. Un ejemplo de esto es reportado en [7], donde un sistema de código abierto es específicamente entrenado y ajustado para reconocer textos históricos escritos en lenguas bereberes transcritos en latín. Otro ejemplo es [5], en el cual se desarrolló un módulo de corrección de palabras que eran marcadas como sospechosas por el motor del OCR de acuerdo a una característica del sistema de OCR que brinda un nivel de confianza binaria a nivel de caracteres, aprovechando esta información interna para hacer las correcciones necesarias.

2.3 Después del OCR

La situación más común es utilizar un sistema comercial que es generalmente de código no abierto y que no brindan demasiada claridad en cómo realizan el trabajo interno, por lo cual se opta por corregir los errores del OCR a posteriori. Esto se debe a que usualmente estos programas tienen mejor rendimiento que los motores de código abierto. Una manera de corregir los errores de OCR a posteriori es utilizando el modelo del canal ruidoso (noisy channel model) [3], en el cual un mensaje es enviado (texto reconocido por el OCR) y distorsionado debido al ruido en el canal de comunicación (proceso de OCR) y el receptor intenta decodificar o entender qué fue enviado (texto original). Por ejemplo, en [8] el modelo es utilizado en un contexto general de la tarea del procesamiento de lenguaje natural, adquiriendo la traducción del léxico de textos impresos. En [9] el canal ruidoso es usado para lenguajes de bajos recursos mientras en [4] es utilizado para corregir errores en un sistema de traducción de texto embebido en imágenes. Más recientemente, este modelo se ha utilizado para segmentar texto de manera no supervisada para la corrección de textos históricos alemanes.

Otra línea de trabajo, es utilizar la métrica de distancias de edición (comúnmente la distancia de Levenshtein o alguna de sus variaciones) entre las palabras obtenidas por el OCR y las palabras presentes en un diccionario; por ejemplo, en [10] se utiliza la métrica de distancias de edición para corregir textos médicos y en [14, 16] se hace uso de estas métricas para corrección de textos históricos en neerlandés. El algoritmo principal para encontrar palabras similares esta explicado en [13] para adecuarse específicamente a los errores de OCR y fue modificado para implementarse en este trabajo.

Alternativamente las salidas de varios motores de OCR pueden ser comparadas

utilizando un sistema de votación para seleccionar o combinar la mejor salida. La idea general es que diferentes motores de reconocimiento generan diferentes errores; ejemplos de esta línea de investigación pueden encontrarse en [20, 1].

Trabajo reciente ha hecho uso de esfuerzo colaborativo para mejorar los textos procesados por OCR, por ejemplo, haciendo uso de motores de búsqueda de errores ortográficos online [2] o vía Crowdsourcing (colaboración abierta distribuida) para mejorar la digitalización de textos impresos históricos como en el proyecto IMPACT [12].

Cada uno de estos enfoques tiene ventajas y desventajas que fueron evaluadas en vista a las colecciones de texto que se esperan corregir.

2.4 Algunos métodos en detalle

2.4.1 Distancia de Levenshtein

Para medir la similitud entre dos cadenas se puede utilizar la “Distancia de Levenshtein” [11]. Esta se puede definir como el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra. Se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter. Por ejemplo, la distancia de Levenshtein entre “casa” y “carpa” es de 2 porque se necesitan al menos dos ediciones elementales para cambiar uno en el otro.

- casa → cara (sustitución de ’s’ por ’r’)
- cara → carpa (inserción de ’p’ entre ’r’ y ’a’)

Para calcular la distancia de Levenshtein de dos cadenas $x := x_1...x_n$ y $y := y_1...y_m$ de manera eficiente, se puede utilizar el algoritmo de Wagner-Fischer [21], que utiliza programación dinámica. El algoritmo utiliza una matriz de distancia D: $(n+1) \times (m+1)$. Sea $-$ el símbolo neutro (no carácter), entonces cada entrada de la matriz D_i, j se puede calcular recursivamente de la siguiente manera:

$$D_{0,0} = 0$$
$$D_{i,0} = D_{i-1,0} + cost(x_i, -), \quad 1 \leq i \leq n$$
$$D_{0,j} = D_{0,j} + cost(-, y_i), \quad 1 \leq j \leq m$$

$$D_{i,j} = min \begin{cases} d_{i-1,j} + cost(x_i, -) \\ d_{i,j-1} + cost(-, y_i) \\ d_{i-1,j-1} + cost(x_i, y_i) \end{cases}$$

Para la distancia de Levenshtein clásica se utiliza la siguiente función de costo:

$$cost(x_i, y_j) = \begin{cases} 0 & \text{si } x_i = y_j \\ 1 & \text{caso contrario} \end{cases}$$

Tabla 2.1: Matriz de la Distancia de Levenshtein: La distancia entre las palabras computadora y calculadora es 4

$x \backslash y$		C	A	L	C	U	L	A	D	O	R	A
	0	1	2	3	4	5	6	7	8	9	10	11
C	1	0	1	2	3	4	5	6	7	8	9	10
O	2	1	1	2	3	4	5	6	7	7	8	9
M	3	2	2	2	3	4	5	6	7	8	8	9
P	4	3	3	3	3	4	5	6	7	8	9	9
U	5	4	4	4	4	3	4	5	6	7	8	9
T	6	5	5	5	5	4	4	5	6	7	8	9
A	7	6	5	6	6	5	5	4	5	6	7	8
D	8	7	6	6	7	6	6	5	4	5	6	7
O	9	8	7	7	7	7	7	6	5	4	5	6
R	10	9	8	8	8	8	8	7	6	5	4	5
A	11	10	9	9	9	9	9	8	7	6	5	4

2.4.2 Distancia de Levenshtein con peso

La función de costo simple de la “Distancia de Levenshtein” puede adaptarse para atacar diferentes tipos de errores. Cada sustitución de caracteres, como también las inserciones y eliminaciones pueden tener diferentes valores. Con respecto a los errores de OCR, se pueden utilizar funciones de costo complejas, por ejemplo, la sustitución del carácter e por c podría tener menor costo, ya que son muy similares para un procesador de OCR dado que tienen una forma similar. Otro ejemplo de esto es el carácter v con el carácter y si el documento no tiene buena calidad y la parte inferior de la y esta borrosa. Asignándole un menor costo a caracteres similares

se puede hacer que por ejemplo la distancia entre la palabra *Lev* y *Ley* sea menor que la distancia entre *Lev* y *Leo*. Por otra parte, si se apuntase a corregir errores en documentos escritos con teclado, se podría asignar menor costo a los caracteres que sean adyacentes en la disposición del teclado.

2.4.3 Hash de Anagramas (Anagram Hashing)

El método Hash de anagramas fue introducido por primera vez por Reynaert [14] y explicado más en detalle en [15]. En esta sección se explicará de una manera más general que en el trabajo original. El método Hash de Anagramas usa principalmente una tabla de hash y una función de hash; esta función está diseñada para producir un número lo suficientemente grande para poder identificar palabras que tengan los mismos caracteres en común. Tales palabras son llamadas anagramas. En términos generales un anagrama es una palabra que está formada desde otra palabra solamente intercambiando caracteres. Por ejemplo, anagramas de la palabra ROMA son: ARMO, RAMO, AMOR, MORA, y todas las palabras que se puedan formar combinando estas cuatro letras.

Para calcular el valor numérico de una palabra p, se debe calcular el correspondiente valor numérico de cada carácter c_i de la palabra $p = c_1...c_{|p|}$. Para esto se puede tomar, por ejemplo, la codificación ISO latin-1 de cada carácter. Esta codificación asigna un valor entero a cada carácter (por ejemplo, le asigna el valor 97 al carácter a). La función, que mapea cada carácter a su correspondiente valor numérico, se denotará *int*. Para obtener el valor hash de una palabra dada, cada carácter se eleva a una potencia de n y se realiza la sumatoria de los valores obtenidos. Formalmente la función hash se define de la siguiente manera:

$$hash(p) := \sum_{i=1}^{|p|} int(c_i)^n, \quad para \quad p = c_1...c_{|w|}$$

Esta función hash debería producir el mismo valor para las palabras que tengan los mismos caracteres en común. Por lo que se setea empíricamente en 5 a la potencia n . Potencias menores a 5 no tienen las propiedades deseadas de acuerdo con Reynaert [14]. Como esta función de hash produce el mismo número para cada palabra que tengan los mismos caracteres, el nombre hash de anagramas se justifica. El hash calculado para la palabra es llamado clave de anagrama (anagram key). Este método

se utiliza para calcular la clave de similitud explicado en la siguiente sección.

2.4.4 Clave de similitud (Similarity key)

Las técnicas de clave de similitud están basadas en mapear palabras a claves de manera que palabras similares (deletreadas) tienen claves idénticas o similares. El cálculo de la clave de similitud para una palabra está basado en la idea de clasificar sus caracteres respecto a su similitud con el procesador OCR asociado. Esto quiere decir que caracteres que probablemente pueden ser confundidos durante el proceso de OCR, caen en la misma clase de equivalencia. Estas clases de equivalencia se deben adaptar a las diferencias entre los procesadores OCR y los distintos tipos de fuentes. Por lo tanto, estas clases se deben calcular de acuerdo al motor OCR que se esté utilizando y al corpus de datos que se desea procesar.

Las clases de equivalencia deberían contener caracteres que sean similares en cuanto a la forma, ya que en principio es lógico pensar que un motor de OCR puede confundir caracteres parecidos. Por ejemplo, los caracteres *e* y *c* deberían estar en la misma clase de equivalencia. Por otro lado, si se estuviese corrigiendo errores de tipografía en documentos escritos en computadora, las clases de equivalencia se podrían crear de acuerdo a la adyacencia de las teclas con respecto a su disposición en el teclado. Esta técnica en conjunto con el método hash de Anagramas se utilizan y explican en detalle en [13]

Capítulo 3

Metodología adoptada para documentos del APM

En este capítulo se realizará un análisis sobre la colección de documentos del APM (Archivo Provincial de la Memoria) a corregir. En la segunda sección se discutirá la propuesta de procesamiento de los documentos, como también algunos detalles técnicos, en esta sección se encuentra la dirección al repositorio del código realizado en Python. En la tercera sección se explicará el algoritmo base en el que se basó el trabajo. En la cuarta sección se detallarán las mejoras sobre el algoritmo base. En la quinta sección se explicará detalladamente como se realizó la construcción incremental de diccionarios utilizados por el corrector automático. Finalmente, el capítulo termina con la sección de resultados.

3.1 Análisis de la colección

Según consta en la página oficial del APM[1]: desde 2007 el APM ha comenzado una intensa búsqueda de documentos en muchas estaciones policiales principalmente en la ciudad de Córdoba. Durante este proceso se identificó un patrón común para la clasificación o almacenamiento de estos documentos: Usualmente eran encontrados en cuartos remotos de los edificios, que ya no están en uso. En la mayoría de los casos documentos relevantes eran encontrados tirados u olvidados en gabinetes de archivos en desuso o apilados detrás de diferentes objetos (tan diversos como partes de motocicletas, cajas o ropa). Documentos encontrados de esta manera se encuentran en un mal estado de preservación, lo que dificulta el proceso de digitalización

[1]http://www.apm.gov.ar/

y que resulta en imágenes de mala calidad que, a su vez, producen una salida del OCR muy mala. La colección de documentos que nos ocupa eran producidos principalmente por el ejército en intervalos irregulares de acuerdo a su clasificación de seguridad y la cantidad y necesidad variable de información a comunicar al personal. Entre otra información, estos documentos contienen pases y nombramientos del personal, altas y bajas, realización de cursos, información sobre nuevas normativas de servicio interno de carácter administrativo, a veces vinculada a hechos extraordinarios (bonificaciones salariales, etc.); esta información tiene cierta estructura que se puede explotar a la hora de corregir errores. Por ejemplo, pueden identificarse nombres por ser siempre escritos con mayúsculas.

Dado el tipo de información contenida en los documentos, un alto porcentaje de las palabras corresponden a entidades con nombre, en particular nombres de personas y ciudades, nombres de unidades del ejército, número de artículos o leyes, fechas, rangos militares y abreviaciones. Parte del texto está estructurado en tablas de largo y posiciones variables, muchas veces apaisadas ocupando varias páginas. También contienen sellos, firmas y anotaciones manuscritas. Todos estos elementos son en efecto muy desafiantes para cualquier motor de OCR. En el caso particular del APM el motor seleccionado es ABBY Fine Reader, el cual tiene un alto nivel de precisión para documentos relativamente bien preservados. De todas maneras, este motor no es capaz de reconocer confiablemente algunos de los elementos mencionados anteriormente y produce resultados incorrectos o "basura" (secuencia de caracteres en apariencia aleatorios, por ejemplo `&í/C#OSj/A¡S`). Teniendo eso en mente analizamos la colección para estimar cuáles de los documentos son más adecuados y tienen el potencial de ser corregidos automáticamente. El objetivo es focalizarse en los documentos que tienen aparentemente más palabras que "basura". Se definió la métrica *proporción de palabras alfanuméricas* (Alphanumeric Word Ratio, AWR) como la relación entre palabras válidas y la cantidad total de palabras para estimar qué tan adecuado es un documento para ser corregido. Definimos una palabra como una secuencia de caracteres delimitados por espacios.

Del total de la colección de documentos tratados solamente la mitad de sus palabras son alfanuméricas (indicando palabras potencialmente válidas). La distribución de la métrica AWR se muestra en la figura 3.1 y el AWR promedio de todos los documentos es 0,557. Seleccionamos aleatoriamente algunos documentos de cada clase y

realizamos una inspección manual para validar los resultados de la métrica y encontramos que documentos con un AWR entre 0 y 0,5 efectivamente contienen "basura" (que resultan del mal reconocimiento) y documentos con un AWR entre 0,8 y 1 contienen texto de alta calidad (palabras correctamente reconocidas). Por lo tanto decidimos enfocarnos en documentos con un AWR entre 0.5 y 0.8, los cuales contienen una combinación de texto reconocido correctamente y texto mal reconocido. Sin embargo, consideraremos reprocesar el conjunto de documentos con bajo AWR, ya sea por un motor de OCR con mejor efectividad o por el mismo motor con un pre-procesamiento de la imagen original (en un intento de mejorar los resultados del OCR).

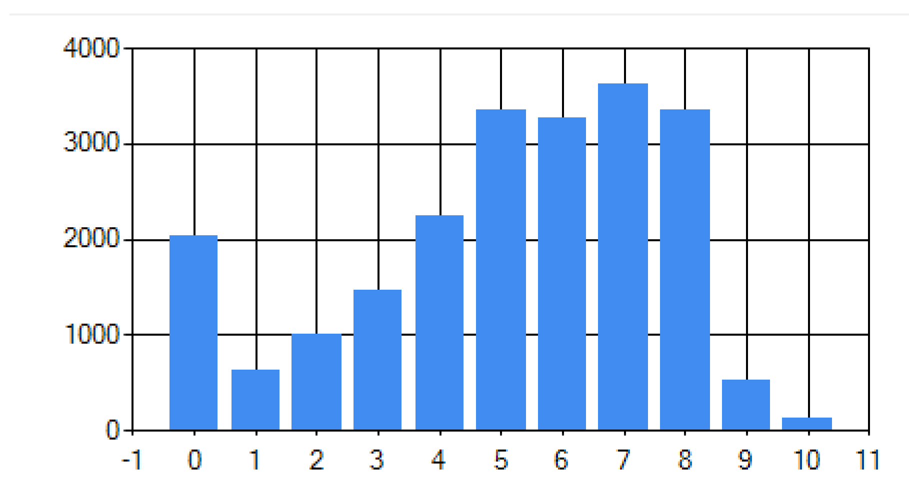

Figura 3.1: Distribución AWR en la colección (multiplicado por 10)

3.2 Propuesta de Procesamiento

Los errores que comete el motor de OCR se pueden separar en dos clases:

- Error de palabra no válida (non-word error): El OCR comete un error de reconocimiento y da como salida una palabra no válida en un diccionario determinado. Por ejemplo, si la palabra original es "otras" y el OCR da como salida "ottas". También caen en esta clasificación las cadenas "basura"

- Error de palabra válida (real word error): El OCR comete un error de reconocimiento y da como salida una palabra válida con respecto a algún diccionario, pero no es la palabra del texto original. Por ejemplo, si la palabra original es "ejército" y el OCR da como salida "ejercicio".

Dada la naturaleza de los documentos a procesar y luego de la inspección manual de una muestra significativa de documentos, decidimos enfocarnos en errores de palabras no válidas ya que los errores que terminan de palabra válida suceden de manera poco frecuente en la muestra aleatoria seleccionada. Por la misma razón decidimos descartar los errores de separación (la unión de varias palabras en una, o la separación de una palabra en varias) y de orden (cuando la salida del OCR es la palabra original pero con algunos caracteres en orden diferente, por ejemplo “sail” en lugar de “sali”).

Con el fin de adoptar un método, consideramos los enfoques discutidos en el capítulo 2 y decidimos que línea de trabajo podríamos seguir considerando nuestras condiciones de trabajo. Un aspecto muy importante que hay que tener en cuenta es el carácter confidencial o privado de los documentos procesados. Esto elimina los enfoques que incluyen la divulgación de los datos, ya sea utilizando “crowdsourcing” (colaboración abierta distribuida) o cualquier tipo de alternativa que implique dar acceso a los datos a través de Internet. Otra limitación es que no tenemos acceso a los documentos originales para re-procesarlos, por lo que el uso de varios motores OCR no es posible. Por razones similares, no es posible experimentar dentro del motor de OCR. Estas limitaciones sugieren que el post-procesamiento del texto de salida del OCR se debería hacer usando una combinación de distancias de edición y diccionarios o usando enfoques más sofisticados (como por ejemplo el modelo de canal ruidoso, máquinas de estado finito, modelos de lenguaje, etc.). Decidimos utilizar la distancia de edición bajo la hipótesis que el limitado y particular vocabulario de los documentos podrían ser capturados en diccionarios personalizados.

Para hacer la corrección se escribió un programa que recibe como entrada un archivo de texto plano (archivo de salida del OCR) y como salida un archivo al cual se le realizan correcciones usando como principales métodos la distancia de edición y la utilización de un diccionario adaptado al corpus de datos. Como lenguaje de programación se optó por utilizar Python, ya que es un lenguaje de programación multiparadigma, multiplataforma, con licencia de código abierto y cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. El código completo se puede acceder desde el siguiente repositorio.

https://sourceforge.net/projects/ocr-corrector/

3.3 Algoritmo Base

Como algoritmo base se utilizó un simple corrector ortográfico como el que describe Peter Norvig en http://norvig.com/spell-correct.html

Código:

```
import re
from collections import Counter
def words(text): return re.findall(r'\w+', text.lower())

WORDS = Counter(words(open('big.txt').read()))

def P(word, N=sum(WORDS.values())):
    #probabilidad de 'word'.
    return WORDS[word] / N

def correction(word):
    #La correccion mas probable de la palabra word.
    return max(candidates(word), key=P)
    def candidates(word):
    #Genera las posibles correcciones para 'word'.
    return (known([word]) or known(edits1(word)) or known(edits2(word)) or [word])

def known(words):
    #el subconjunto 'words' que aparecen en el diccionario de 'WORDS'.
    return set(w for w in words if w in WORDS)

def edits1(word):
    #Todas las ediciones que están a distancia 1 de 'word'.
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)

def edits2(word):
    #Todas las ediciones que estan a distancia 1 de 'word'.
    return (e2 for e1 in edits1(word) for e2 in edits1(e1))
```

3.3.1 Cómo Funciona: Teoría de Probabilidad

La llamada a *correction(w)* intenta elegir la corrección más probable para w. No hay una forma segura de saber si la corrección que vamos a aplicar es la correcta, lo que sugiere que hay que utilizar probabilidades. Estamos intentando encontrar la corrección c, de todos los posibles candidatos de corrección, que maximice la probabilidad de que c es la corrección deseada, dada la palabra original w. Sea *argmax* los puntos del dominio de una función en los cuales los valores de dicha

función son maximizados.

$$argmax_{c \in candidatos} P(c|w)$$

Por el teorema de Bayes esto es equivalente a:

$$argmax_{c \in candidatos} P(c)P(w|c)/P(w)$$

Dado que *P(w)* es siempre igual para cada posible candidato *c*, se puede quitar quedando:

$$argmax_{c \in candidatos} P(c)P(w|c)$$

Las cuatro partes de esta expresión son:

1. Mecanismo de selección: *argmax*

   Se elige el candidato con la probabilidad combinada mas alta

2. Modelo de candidatos: $c \in candidatos$

   Esto nos dice que candidatos de corrección, *c*, hay que considerar

3. Modelo de lenguaje: *P(c)*

   La probabilidad de que *c* aparezca como palabra en un texto.

4. Modelo de Error: *P(w|c)*

   La probabilidad de que la palabra *w* aparezca en un texto como salida del OCR cuando en realidad la palabra original es *c*.

3.3.2 Cómo Funciona: Python

Las cuatro partes del programa son:

1. Mecanismo de selección: En Python, la función *max* hace el trabajo de *argmax*

2. Modelo de candidatos: Una edición simple de una palabra es el borrado, transposición, reemplazo o inserción de un carácter. La función edits1 retorna el conjunto de todas las cadenas (sean palabras o no) que se pueden hacer con una edición simple, es decir todas las cadenas a distancia de Levenshtein 1. Esto puede ser un conjunto muy grande. Si estamos considerando un alfabeto de

tamaño *t*, para una palabra de largo *n*, existen *n* borrados, *n* - 1 transposiciones, $t \times n$ reemplazos, y *t(n+1)* inserciones, dando un total de $(t+2) \times n + t - 1$ cadenas (de las cuales algunas pueden ser duplicados). Sin embargo, si lo restringimos a palabras conocidas (es decir que están en el diccionario), entonces el conjunto es mucho más pequeño. También consideramos correcciones que están a distancia 2 con la función edits2.

3. Modelo de lenguaje: Podemos estimar la probabilidad de una palabra *w*, *P(w)*, contando el número de veces que cada palabra aparece en un archivo de texto que contenga cerca de un millón de palabras (big.txt, que es un archivo de texto formado por la concatenación de varios artículos de Wikipedia). La función *words* separa el texto en palabras, la variable *WORDS* mantiene un contador de cuán frecuentemente cada palabra aparece, y *P* estima la probabilidad de cada palabra, basado en este contador.

4. Modelo de Error: En esta versión del algoritmo se define un modelo de error trivial en el cual todas las palabras conocidas a distancia de edición 1 son infinitamente más probables que las palabras conocidas a distancia de edición 2, y infinitamente menos probables que una palabra conocida a distancia de edición 0. De esta manera podemos hacer que la función *candidates(word)* produzca la primer lista de candidatos no vacía en orden de prioridad.

    - La palabra original, si es conocida; caso contrario
    - La lista de palabras con distancia de edición 1, de existir al menos una; caso contrario
    - La lista de palabras con distancia de edición 2, de existir al menos una; caso contrario
    - La palabra original, aunque sea una palabra no conocida.

    De esta forma no es necesario multiplicar por el factor *P(w|c)*, ya que cada candidato en la prioridad elegida va a tener la misma probabilidad (de acuerdo a este modelo trivial)

3.4 Mejoras sobre el algoritmo Base

Sobre el algoritmo base se realizaron diferentes cambios y mejoras para adaptarse a la clase de errores cometidos por el OCR y al vocabulario utilizado en la colección de documentos. En términos generales los cambios a destacar son:

- Eliminación de transposición de caracteres al generar la lista de candidatos ya que éstos apuntan a corregir errores tipográficos y no de OCR.
- Pre-procesamiento de errores en palabras cortas que suceden con mucha frecuencia en la colección de documentos.
- Implementación de clases de equivalencia.
- Manejo diferenciado para entidades con nombre.
- También se realizó mucho trabajo en la elaboración del diccionario utilizado tratando de capturar el vocabulario específico de la colección de documentos.

Estos cambios son explicados a continuación.

En términos generales, el proceso implementado toma un documento como entrada, extrae las palabras y chequea cada una de ellas contra un diccionario para decidir si son palabras válidas o si se debe generar una lista de correcciones potenciales. Si la palabra examinada está en el diccionario asumimos que es una salida válida del OCR, pero de no estar presente se genera una lista de candidatos, más precisamente una lista de palabras que están a una distancia de edición de 1 o 2 de la palabra que está siendo examinada, de esta lista se selecciona la palabra con más frecuencia en el diccionario como candidata de corrección. La métrica utilizada es la distancia de Levenshtein [11] y algunos test preliminares que se realizaron sugieren que generar candidatos de mayor distancia a 2 introduce errores en lugar de utilizar los reemplazos correctos además de agregar una gran carga computacional. Para generar la lista de candidatos de corrección se utilizaron dos estrategias. La primer estrategia consiste en generar palabras aplicando las operaciones de edición a cada carácter de la palabra examinada y buscando las nuevas palabras en el diccionario. Dado que este proceso automáticamente genera palabras que pueden no ser válidas, se pensó como estrategia limitar a palabras que contengan los trigramas más frecuentes en español, de esta forma recortando el árbol de posibles candidatos y haciendo el proceso considerablemente más eficiente, en principio haciendo posible

explorar más niveles de distancia. Finalmente, esta estrategia fue descartada luego de algunas pruebas al notar que, si se eliminan ciertas palabras posibles por contener algún trigrama no válido en español, se eliminan ramas que luego podrían llevar a palabras válidas si posteriormente se remplaza algún caracter del trigrama no válido.

La segunda estrategia consiste en la personalización del algoritmo presentado en [13] para tomar ventaja de los errores tipográficos que comete el motor de OCR. El algoritmo fue adaptado para el lenguaje español y a los tipos de errores observados en una muestra significativa de la colección de documentos que se están analizando. Las clases de equivalencia resultantes son [fiíjklrtIJLT1!] [abdgoopqOQ690] [éecCG] [vxyVYX] [sS5] [zZ] [EFPRK] [úu], lo cual significa que un carácter de una clase solo puede ser reemplazado por cualquiera de los otros caracteres de la misma clase, y la palabra resultante es buscada en el diccionario. Por ejemplo, la palabra mal reconocida 'Lev' puede ser corregida reemplazando la letra 'v' por una 'y' generando la palabra 'Ley'. De no usarse las clases de equivalencia la palabra 'Lev' se podría reemplazar por 'Leo' (Si 'Leo' tuviera más frecuencia en el diccionario que 'Ley'). Lo que se logra con las clases de equivalencia es análogo a utilizar distancias de edición con peso, dándole un menor peso a los caracteres que están en la misma clase de equivalencia.

Además de una buena estrategia de generación de candidatos, se necesita una diccionario adecuado. Para este proyecto se ha extendido y adaptado un diccionario base, como se explica en la próxima sección.

Antes de correr el algoritmo se hace un pre-procesamiento de las palabras para corregir errores muy frecuentes en palabras cortas. Por ejemplo:

Lev → Ley

Vita → VIta

Iros → 1ros

Ilda → IIda

ei → el

ios → los

1? → 1º

Este pre-procesamiento se realiza porque son errores muy frecuentes y aunque el algoritmo no corrige palabras con largo menor a 3, estas correcciones ayudan a mejorar el rendimiento del algoritmo y la legibilidad de la salida.

El programa también cuenta con un parámetro de entrada para que utilice, o no, una corrección diferenciada para las palabras que puedan ser nombres. Esta funcionalidad se implementó ya que muchos documentos contienen gran cantidad de nombres, los cuales pueden ser datos sensibles, por lo que puede ser muy importante evitar que se realicen correcciones equivocadas. La corrección diferenciada utiliza un diccionario que contiene sólo nombres evitando que se corrija un nombre a una palabra que no es un nombre. Si el parámetro está activo y la palabra a corregir esta en mayúsculas se invoca a un método que intenta corregir la palabra como si fuese un posible nombre. Primero se verifica que la palabra no esté en el diccionario que no incluye nombres y luego verifica el contexto de la palabra que se está intentando corregir chequeando si la palabra anterior o posterior es un nombre, y si este es el caso se utiliza el diccionario que sólo incluye nombres. Este método se podría mejorar utilizando otros métodos de detección de entidades con nombre.

3.5 Construcción Incremental de Diccionarios

El algoritmo de corrección automática hace uso de diccionarios no sólo para saber si es una palabra válida, sino también para elegir la palabra con mayor frecuencia cuando se genera la lista de candidatos de corrección. Por esta razón los diccionarios son una dupla (palabra, frecuencia). Las palabras son únicas y la frecuencia es la cantidad de veces que la palabra aparece en el texto que se utiliza para generar el diccionario. Para la construcción del primer diccionario se tomó un conjunto de artículos de Wikipedia usando un 'web-crawler' enfocándolo a artículos que pudieran tener relación con las palabras utilizadas en el APM. Por ejemplo, se tomaron como semilla los artículos "Proceso de Reorganización Nacional", "Ejército Argentino", "Argentina", y otros de temática similar. Con esto se reunieron 20 MB de texto, donde se encontraron 111.438 palabras únicas, incluyendo palabras de vocabulario en general. Luego de algunas pruebas encontramos que este diccionario no era suficiente ya que, aunque los artículos y palabras tenían relación con las palabras utilizadas en el APM, la cobertura no era suficiente.

Para tratar de solucionar la falta de cobertura decidimos aumentar el diccionario con palabras específicas al corpus de datos. Para esto seleccionamos aleatoriamente 30 documentos del corpus del APM y después de una corrección manual agregamos las palabras al diccionario. A pesar de estos cambios, todavía no obtuvimos

resultados positivos dado que el número de palabras únicas de estos documentos era relativamente baja (propiamente 915 palabras únicas).

Notamos que los nombres propios eran las principales entidades presentes en los documentos, por lo que incluimos una larga lista de nombres de personas, ciudades y provincias obtenidos de una base de datos de nombre de internet y artículos de Wikipedia (ciudades y provincias). Una vez procesada esta lista se obtuvieron 26.505 nombres únicos. También decidimos agregar el diccionario de Aspell (1.250.793 palabras únicas) para lograr mayor cobertura y evitar que se corrijan palabras válidas.

La ultima adición al diccionario es un conjunto de palabras obtenidas de la colección de documentos después de un análisis de errores. El algoritmo se utilizó para procesar todos los documentos y las palabras no encontradas en el diccionario se asentaron. La cantidad de palabras no encontradas fueron 222.648. Estas palabras pueden ser errores de OCR o palabras válidas que nuestro diccionario no contenía, pero asumimos que las 1000 palabras más frecuentes no pueden ser error de OCR porque eso implicaría que el motor comete errores sistemáticos que no es el caso con nuestros datos. La tabla 3.1 muestra una lista de palabras que se añadieron al diccionario. Todas ellas son palabras válidas, mayormente abreviaciones (10/13) y sólo una de ellas tiene un error (comuniqúese), pero eso es un error en el documento impreso y por lo tanto no un error de OCR. Añadiendo estas 1000 palabras al diccionario como palabras válidas del dominio se obtuvieron los mejores resultados hasta el momento.

3.6 Resultados

La primera versión del diccionario estaba basada en artículos de Wikipedia pero se introducían más errores que correcciones ya que hay muchas palabras a distancia 1 o 2 que tienen mayor frecuencia que las palabras específicas. Esto hace que palabras válidas al no estar presentes en el diccionario sean incorrectamente reemplazadas por palabras con mayor frecuencia (por ejemplo, el reemplazo de la palabra revistando por revisando). Esto se ve reflejado en porcentajes negativos de corrección (ver fila denotada Wikipedia en la Tabla 3.2).

Problemas similares sucedieron al crear un diccionario con palabras de documentos corregidos manualmente, ya que, aunque las palabras de este diccionario son específicas al dominio, la falta de cobertura de palabras hacía que se introduzcan

Palabra	Frecuencia
expl	3802
cond	3556
subof	3409
apart	3086
conscripto	2074
comb	2073
mte	1993
sarg	1811
comuniqúese	1723
archívese	1683
agr	1542
gam	1421
iose	1412

Tabla 3.1: Primeras 13 palabras del top 1000 de las palabras más frecuentes de la colección que fueron agregadas al diccionario

más errores (fila específicas-dominio en la Tabla 3.2).

Los primeros resultados positivos aparecieron al combinar estos diccionarios (fila Wikipedia + especificas-dominio). Mejoras considerables se pueden observar cuando se añade la lista de nombres (fila Wikipedia + especificas-dominio + nombres en la Tabla 3.2). Luego se añadieron las palabras de Aspell para lograr mayor cobertura y finalmente se agregaron las palabras especificas-dominio* (1000 palabras más frecuentes del dominio no encontradas en los diccionarios) cuyos resultados se pueden ver en la fila Wikipedia + especificas-dominio + Aspell + especificas-dominio* de la Tabla 3.2. El diccionario de Aspell aporta una contribución muy pequeña en el conjunto de prueba, pero se decidió mantenerlo para trabajos futuros, ya que la solución implementada para la corrección de OCR puede ser aplicada para otras colecciones del APM. Como se puede observar en la Tabla 3.2 las contribuciones más altas corresponden a las palabras derivadas del conjunto de datos que son específicas al dominio y el conjunto de nombres, sugiriendo que se puede realizar más trabajo sobre estos.

Aunque se lograron algunas mejoras, es muy difícil eliminar muchos de los errores dada la heterogeneidad y el estado de conservación de los documentos, lo que genera

una gran variedad de errores en la salida del OCR. Por ejemplo, las palabras DESIGNACION / IMPORTE, aparecen en el mismo documento como DESISKACIOB / na poete, DESIGNACIOH / IMPORTE, D E S I O H A O I O S / IMPORTE, DESIGNACION / IMPOBTE y DESIGNACION / HIFOKTB. Algunos de estos errores pueden ser corregidos, pero otros obviamente quedan presentes.

Los algoritmos y diccionarios creados fueron testeados en un conjunto aleatorio de 10 documentos que fueron corregidos manualmente para llevar a cabo una evaluación. Ese conjunto de documentos contenía un total de 118 errores. Al correr el programa de corrección automática se obtuvieron 3 versiones del archivo. La salida original del OCR, el corregido manualmente y la salida del programa. Mediante un algoritmo automático se calcularon las siguientes métricas:
Cantidad de Errores (CE): Cantidad de modificaciones entre la salida del OCR y el archivo corregido manualmente.
Cantidad de Errores Algoritmo (CEA): Cantidad de modificaciones entre el archivo corregido manualmente y el corregido automáticamente.
Cantidad de Errores Corregidos (CEC): CE-CEA
Ratio (R): CEC/CE

Diccionario	CE	CEA	CEC	R
Wikipedia	118	132	-18	-15
Específicas-dominio	118	147	-29	-24
Wikipedia + especificas-dominio	118	101	17	14
Wikipedia + especificas-dominio + nombres	118	93	25	21
Wikipedia + especificas-dominio + Aspell	118	91	27	23
Wikipedia + especificas-dominio + Aspell + especificas-dominio*	118	84	34	29

Tabla 3.2: Ratio de errores corregidos (R) con las diferentes versiones del diccionario. Resultados negativos indican la introducción de errores

Capítulo 4

Conclusiones

En este trabajo se presentaron las decisiones tomadas para implementar una solución automática para la corrección de errores de OCR para una colección de documentos muy especial. Dadas las limitaciones de acceso al material impreso y la confidencialidad de los documentos se adoptó una metodología basada en diccionarios específicos para realizar una post-corrección de la salida de texto del OCR. El algoritmo principal para realizar la corrección automática se basó en un corrector ortográfico programado en Python. Sobre este algoritmo se realizaron muchas mejoras para adaptarse a la colección de datos y a la salida de un OCR. Las principales mejoras a mencionar fueron:

- Usar clases de similitud para capturar los errores que comete el OCR al identificar erróneamente caracteres similares.
- Pre-procesamiento simple del texto para capturar errores sistemáticos, sobre todo en abreviaciones cortas.
- Tratamiento diferenciado para nombres propios.

Además de estas mejoras, se realizó mucho trabajo en la creación de diccionarios específicos para capturar el vocabulario específico de la colección de documentos a corregir. Con este algoritmo fue posible lograr una corrección aproximada del 30% de los errores encontrados en un conjunto de prueba.

4.1 Fortalezas y limitaciones

El algoritmo propuesto trabaja de a una palabra por vez, por lo que no se utiliza el contexto de las palabras adyacentes para realizar las correcciones. Sin embargo,

dado el acotado y particular vocabulario de los documentos, el uso de diccionarios específicos y el uso de clases de similitud para capturar los errores del OCR lograron resultados prometedores.

4.2 Trabajo futuro

La colección de documentos con la que se trabajó contiene errores muy variados y en muchos casos es difícil que puedan ser sistemáticamente corregidos de manera automática, por lo que queda mucho trabajo por hacer para incrementar más aun la calidad de los documentos.

El trabajo a futuro podría incluir:

Pre-procesamiento de las imágenes de entrada al OCR, como por ejemplo la mejora de la uniformidad de iluminación.

Añadir un modelo de lenguaje entrenado en el dominio de los documentos.

Mejorar los diccionarios con datos aportados por otros miembros del equipo que trabajan en el reconocimiento de entidades (NER) sobre la misma colección de documentos y por personas que utilizan estos documentos, como por ejemplo abogados o personal del APM.

Bibliografía

[1] Ahmad Abdulkader and Matthew R Casey. Low cost correction of ocr errors using learning in a multi-engine environment. In *Document Analysis and Recognition, 2009. ICDAR'09. 10th International Conference on*, pages 576–580. IEEE, 2009.

[2] Youssef Bassil and Mohammad Alwani. Context-sensitive spelling correction using google web 1t 5-gram information. *arXiv preprint arXiv:1204.5852*, 2012.

[3] N Blachman. A mathematical theory of communication. *IEEE Transactions on Information Theory*, 14:27–31, 1968.

[4] Yi Chang, Datong Chen, Ying Zhang, and Jie Yang. An image-based automatic arabic translation system. *Pattern Recognition*, 42(9):2127–2134, 2009.

[5] Siyuan Chen, Dharitri Misra, and George R Thoma. Efficient automatic ocr word validation using word partial format derivation and language model. In *IS&T/SPIE Electronic Imaging*, pages 75340O–75340O. International Society for Optics and Photonics, 2010.

[6] Andreas Fischer, Andreas Keller, Volkmar Frinken, and Horst Bunke. Lexicon-free handwritten word spotting using character hmms. *Pattern Recognition Letters*, 33(7):934–942, 2012.

[7] Marcin Heliński, Miłosz Kmieciak, and Tomasz Parkoła. Report on the comparison of tesseract and abbyy finereader ocr engines. 2012.

[8] Okan Kolak, William Byrne, and Philip Resnik. A generative probabilistic ocr model for nlp applications. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 55–62. Association for Computational Linguistics, 2003.

[9] Okan Kolak and Philip Resnik. Ocr post-processing for low density languages. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 867–874. Association for Computational Linguistics, 2005.

[10] Thomas A Lasko and Susan E Hauser. Approximate string matching algorithms for limited-vocabulary ocr output correction. In *Photonics West 2001-Electronic Imaging*, pages 232–240. International Society for Optics and Photonics, 2000.

[11] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[12] Clemens Neudecker and Asaf Tzadok. User collaboration for improving access to historical texts. *Liber Quarterly*, 20(1), 2010.

[13] Kai Niklas, Rainer Parchmann, M ScÑina Tahmasebi, and Thomas Risse. Unsupervised post-correction of ocr errors. *Unpublished diploma thesis, Hannover University, Germany. Available a t< http://www. l3s. de/˜ tahmasebi/Diplomarbeit_ Niklas. pdf*, 2010.

[14] Martin Reynaert. Text induced spelling correction. In *Proceedings of the 20th international conference on Computational Linguistics*, page 834. Association for Computational Linguistics, 2004.

[15] Martin Reynaert. Corpus-induced corpus clean-up. In *Proc. of the Fifth International Conference on Language Resources and Evaluation, LREC-2006*, 2006.

[16] Martin Reynaert. Non-interactive ocr post-correction for giga-scale digitization projects. In *Computational Linguistics and Intelligent Text Processing*, pages 617–630. Springer, 2008.

[17] José A Rodríguez-Serrano and Florent Perronnin. Handwritten word-spotting using hidden markov models and universal vocabularies. *Pattern Recognition*, 42(9):2106–2116, 2009.

[18] Jose A Rodriguez-Serrano and Florent Perronnin. Synthesizing queries for handwritten word image retrieval. *Pattern Recognition*, 45(9):3270–3276, 2012.

[19] Tolga Tasdizen, Elizabeth Jurrus, and Ross T Whitaker. Non-uniform illumination correction in transmission electron microscopy. In *MICCAI Workshop on Microscopic Image Analysis with Applications in Biology*, pages 5–6, 2008.

[20] Martin Volk, Torsten Marek, and Rico Sennrich. Reducing ocr errors by combining two ocr systems. In *ECAI-2010 workshop on language technology for cultural heritage, social sciences, and humanities*, pages 61–65, 2010.

[21] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

Apéndice A

Anexo

A.1 Programa: Manual de Uso

El código generado en este trabajo final está disponible en https://sourceforge.net/projects/ocr-corrector/

Descripción: Módulo de python para corregir archivos dentro de un directorio basado en diccionarios. El archivo default.cfg tiene la configuración predeterminada del directorio donde se encuentran los archivos a corregir, el directorio donde están los diccionarios, si se va a corregir recursivamente los directorios y el nombre del archivo donde se van a guardar los resultados.

Modo de uso:

python ocr_corrector options

-h, –help Muestra esta ayuda.

-r, –recursive Indica que el directorio de entrada se va a explorar recursivamente.

-d, –dictionaries_folder Recibe como parámetro el directorio donde se encuentran los diccionarios a utilizar.

-f, –folder_to_correct Recibe como parámetro el directorio de entrada de donde se van a leer los documentos a corregir.

-l, –log_filename Recibe como parámetro el nombre del archivo donde se va a guardar los resultados.

-g, –garbage_remove Indica que los archivos corregidos no van a tener las cadenas identificadas como "basura".

-n, –name_detection Detecta nombres propios y los corrige con un diccionario específico.

A.2 Programa: Dependencia Archivos

Dictionaries: Diccionarios utilizados por el algoritmo principal.
ocr_corrector.py: Algoritmo principal para corregir automaticamente errores de OCR
default.cfg: Archivo de configuración con los parámetros predeterminados del algoritmo principal.
dictionary_manager.py: Funciones auxiliares para construir los diccionarios específicos.
calculate_wer.py: Maneja los parámetros de entrada necesarios para calcular la ratio de errores de palabras "Word error rate". Esto se utiliza para medir el rendimiento del algoritmo principal, para esto se utilizan tres archivos La salida del OCR (txt) Salida del OCR corregida manualmente (fixed.txt) Salida del OCR corregida automáticamente por el algoritmo. (corrected.txt) Luego se realizan dos comparaciones para obtener: WER entre el txt y el fixed.txt (Errores del OCR) WER entre el txt y corrected.txt (Errores de OCR que no fueron corregidos automáticamente)
wer.py: Funciones para calcular la ratio de errores de palabras "Word error rate" entre dos archivos.
editdistance.py: Funciones para calcular la distancia de edición entre palabras

Dependencias:

$$ocr_corrector.py \begin{cases} Dictionaries \\ default.cfg \end{cases}$$

calculate_wer.py -> wer.py -> editdistance.py

A.3 Programa: Ejemplos de Salida: ocr_corrector.py

```
Begin file :::: F15N3051989.txt:::::::::
Original -> Correction: 1apartado -> apartado
Original -> Correction: Lev -> Ley
Original -> Correction: Permanetes -> Permanentes
Original -> Correction: dei -> del
Original -> Correction: Avtia -> Aytia
```

```
Original -> Correction: dei -> del
Original -> Correction: 3AVIO -> SAVIO
Original -> Correction: Máy -> May
Original -> Correction: Reser -> Ryser
Original -> Correction: ei -> el
Original -> Correction: liasta -> lista
Original -> Correction: Lev -> Ley
Original -> Correction: Lev -> Ley
Original -> Correction: JELA -> TELA
Original -> Correction: ai -> al
Original -> Correction: habere- -> haberes
Correct_File Time : 0.298372983932
End file :::::: F15N3051989.txt :::::::::
```

A.4 Programa: Ejemplos de Salida: calculate_wer.py

```
F15N3051989.txt
::::::::::::::WER::::::::::
REF: F15N3051989.Fixed.txt   HYP: F15N3051989.corrected.txt
INSERTIONS:
                (Ay:          1
               Ryser          1
                   P          1
                  (A          1
               F.Sí          1
                 pan          1
DELETIONS:
                 1º          1
SUBSTITUTIONS:
         Permanentes ->          -rma-nentes            1
              partir ->                    r            1
                Grl) ->                   D)            1
                  ha ->                   Ra            1
             ESCUELA ->                 TELA            1
```

```
hasta ->      lista     1
al ->         a         1
(Aytia ->     ;i        1
SAVIO", ->    FAVIO",   1
19.101 ->     19^101    1
1º, ->        1º.       1
Grl) ->       G.-O      1
Jefe ->       Jete      1
el ->         ei        1
2º, ->        2',       1
2º ->         2-        1
de ->         cié       1
Grl) ->       Gil)      1
la ->         a         1
Grl) ->       í.iu      1
(Aytia ->     Aytia     1
Reserva" ->   a"        1
la ->         (a        1
referida ->   referid:. 1
(Aytia ->     i.i       1
2º, ->        2-,       1
y ->          v         1
3º, ->        3-,       1
2º, ->        2",       1
(LM-2-I), ->  (LM-2-Í), 1
WRR: 96.163366 % (       777 /         808)
WER: 4.579208 % (        37 /         808)
REF: F15N3051989.Fixed.txt   HYP: F15N3051989.txt
INSERTIONS:
Reser    1
(Ay:     1
F.Sí     1
P        1
```

```
                (A          1
               pan          1
DELETIONS:
               1°          1
SUBSTITUTIONS:
               Ley ->                 Lev            3
               del ->                 dei            2
           ESCUELA ->                JELA            1
       Permanentes ->         -rma-nentes            1
             hasta ->              liasta            1
              2°, ->                2",            1
                al ->                  ai            1
            partir ->                   r            1
              1°, ->                1?.            1
                de ->                cié            1
            19.101 ->              19^101            1
            (Aytia ->                 i.i            1
            (Aytia ->                  ;i            1
       Permanentes ->          Permanetes            1
              Grl) ->              í.i'U            1
                ha ->                  Ra            1
              Grl) ->                G.-O            1
              Jefe ->                Jete            1
                el ->                  ei            1
              2°, ->                2',            1
               2° ->                2-            1
            (Aytia ->             ÍAvtia            1
                la ->                 ¡a            1
              Grl) ->                Gil)            1
          SAVIO", ->            3AVIO",            1
          Reserva" ->                  a"            1
                la ->                  (a            1
          apartado ->           1apartado            1
```

```
          referida ->           referid:.          1
              3º, ->               3?,           1
              Grl) ->               Gd)          1
              2º, ->               2-,           1
                 y ->                 v          1
           haberes ->            habere-         1
              3º, ->               3-,           1
               May ->              Máy           1
        (LM-2-I), ->          (LM-2-Í),          1
WRR: 94.925743 % (       767 /        808)
WER: 5.816832 % (        47 /       808)
```

A.5 Programa: Funciones Principales

```
#Dentro de default.cfg
#[Alphabet]
#alphabet= abcdefghijklmnopqrstuvwxyzñáéíóú123456789
#classes= fiíjklrtIJLT1! abdgoópqOQ690 eécCG vxyVYX sS35 zZ EFPRK úu
#Carga los diccionarios
def load_dictionaries(p):

    json_data = open(os.path.join(p.dictionary_path ,'preProccess.json'))
    p.PRE_PROCCESS = json.load(json_data)

    json_data = open(os.path.join(p.dictionary_path ,'dictClean.json'))
    p.NWORDS = json.load(json_data)

    json_data = open(os.path.join(p.dictionary_path ,'dictCleanNoNames.json'))
    p.NWORDS_NO_NAMES = json.load(json_data)

    json_data = open(os.path.join(p.dictionary_path ,'namesNew.json'))
    p.NNAMES= json.load(json_data)

#Retorna un conjunto de todas las cadenas en que cada letra de word es remplazada por todos los caracteres de
#su clase de similitud. Donde 'a' es el inicio de la palabra, 'b' es el pivot que se remplaza por 'r' que toma todos
#los valores de la clase de similitud del caracter 'b', y 'c' es el final de la palabra (todos los posibles)
def edits_sim(word,p):
    splits     = [(word[:i],word[i],word[i+1:]) for i in range(len(word) )]
    replaces   = [a + r + c for a, b, c in splits for r in class_of(b,p)]
    return set(replaces)
#Retorna un conjunto de todas las cadenas que estan a distancia de edicion 1 de word.
#Donde 'a' es el inicio de la palabra, b es el final y 'c' es el caracter que se utiliza para insertar un caracter
# aleatorio del alfabeto o borrar alguno de los caracteres de la palabra word
```

```
def edits(word,p):
    splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
    deletes    = [a + b[1:] for a, b in splits if b]
    replaces   = [a + c + b[1:] for a, b in splits for c in p.alphabet if b]
    inserts    = [a + c + b     for a, b in splits for c in p.alphabet]
    return set(deletes + replaces + inserts)
#Palabras que estan en el diccionario a 'distancia 2' (solamente usando reemplazos de su
#clase de similitud) de la palabra original
def known_edits2_sim(word,dict,p):
    return set(e2 for e1 in edits_sim(word,p) for e2 in edits_sim(e1,p) if e2 in dict)

#Palabras que estan en el diccionario a 'distancia 2' de la palabra original
def known_edits2(word,pdict,p):
    return set(e2 for e1 in edits(word,p) for e2 in edits(e1,p) if e2 in pdict)
#Palabras que estan en el diccionario a 'distancia 1' de la palabra original
def known(words,pdict): return set(w for w in words if w in pdict)

# Retorna la misma palabra si pword esta en pdict. De no estar busca la palabra con mas frecuencia
# que este a 'distancia 1' usando clases de similitud. Si no hay ninguna utiliza la misma formula sucesivamente para,
# buscar las palabras a 'distancia 1', 'distancia 2' usando clases de similitud,o 'distancia 2'
# De no encontrar ninguna palabra valida retorna pword
def correct(pword,pdict,p):
    word=pword.lower()
    candidates =known([word],pdict)
    if len(candidates)==0:
        candidates= known(edits_sim(word,p),pdict)
        if len(candidates)==0:
            candidates= known(edits(word,p),pdict)
            if len(candidates)==0:
                candidates= known_edits2_sim(word,pdict,p)
                if len(candidates)==0:
                    candidates =known_edits2(word,pdict,p)
                    if len(candidates)==0:
                        candidates=[word]
    ret = max(candidates, key=p.NWORDS.get)
    return ret
```

A.6 Documentos: Palabras más Frecuentes

```
del                              |   152859   |  04.54212   |   004.54212   |
por                              |   45616    |  1.355454   |   05.897574   |
personal                         |   41967    |  1.247026   |   0007.1446   |
los                              |   40457    |  1.202157   |   08.346757   |
que                              |   40404    |  1.200582   |   09.547339   |
```

ejército	33912	1.007676	10.555015
con	31439	0.934192	11.489208
para	28678	0.852151	12.341358
militar	27735	00.82413	13.165488
general	26711	0.793702	13.959191
sargento	23678	0.703579	14.662769
carlos	22878	0.679807	15.342576
cabo	22161	0.658502	16.001078
fecha	21859	0.649528	16.650606
jefe	21309	0.633185	17.283791
alberto	20823	0.618744	17.902535
las	19793	0.588138	18.490673
cdo	19755	0.587009	19.077682
juan	18732	0.556611	19.634292
servicios	18203	0.540892	20.175184
mayor	18037	0.535959	20.711144
ley	16540	0.491477	21.202621
bre	15825	0.470231	21.672852
teniente	15697	0.466428	22.139279
luis	15640	0.464734	22.604013
comando	14955	0.444379	23.048393
desde	14700	0.436802	23.485195
jose	14597	0.433742	23.918937
coronel	13577	0.403433	024.32237
iii	13271	00.39434	024.71671
jorge	13234	0.393241	25.109951
años	12006	0.356752	25.466702
aires	11999	0.356544	25.823246
resolución	11895	0.353453	26.176699
buenos	11857	0.352324	26.529023
antonio	11720	0.348253	26.877277
san	11475	0.340973	027.21825
señor	11446	0.340111	27.558361

```
infantería          |  11446  |  0.340111  |  27.898473  |
comandante          |  11037  |  0.327958  |  28.226431  |
grado               |  11010  |  0.327156  |  28.553587  |
artículo            |  11001  |  0.326889  |  28.880476  |
bpe                 |  10850  |  0.322402  |  29.202877  |
eduardo             |  10569  |  0.314052  |  29.516929  |
oscar               |  10524  |  0.312715  |  29.829644  |
jef                 |  10197  |  0.302998  |  30.132642  |
angel               |  10138  |  0.301245  |  30.433887  |
servicio            |  10064  |  0.299046  |  30.732933  |
inciso              |  9634  |  0.286269  |  31.019202  |
```

"Los abajo firmantes, miembros del Tribunal de Evaluación de tesis, damos Fe que el presente ejemplar impreso, se corresponde con el aprobado por éste Tribunal”