UNED

2017

Práctica 5

TÉCNICAS BASADAS EN GRAFOS APLICADAS AL PROCESAMIENTO DEL LENGUAJENATURAL

ALICIA LARA CLARES

CONTENIDO

1 ENUNCIADO TAREA OPCIONAL

La tarea opcional que se propone para este módulo consiste en completar el desarrollo del algoritmo TextRank a partir de la salida de la tarea obligatoria y utilizarlo para una extracción real de Keyphrases sobre textos. Una Keyphrase consiste en un conjunto de términos (habitualmente entre 1 y 5 términos) que resume la información de la que trata un documento. La sección 3 del artículo de Rada Mihalcea explica este proceso, aunque no es necesario aplicar la evaluación que se propone. La entrada del programa que se deberá desarrollar será la misma que para la tarea obligatoria, es decir, dos versiones (documentos etiquetados gramaticalmente y sin etiquetar) de una colección de documentos. Un ejemplo de documento sin etiquetar será 357.txt. Un ejemplo de documento etiquetado será 357.txt.tagged. La salida consistirá en un fichero de texto por cada documento de la colección, y deberá contener al menos 5 keyphrases con un salto de línea de separación entre ellas. El nombre del fichero de texto deberá seguir la regla de estilo de los documentos proporcionados: IDENTIFICADOR.txt.result. Es decir, para el ejemplo de documento 357.txt, se deberá proporcionar el fichero de resultados 357.txt.result.

Para esta tarea podría resultar de ayuda (aunque no es obligatorio su uso) utilizar una herramienta muy conocida entre los adeptos a Java y a los grafos. JUNG (Java Universal Network/Graph) es una biblioteca de software que proporciona un lenguaje común y extensible para el modelado, análisis y visualización de datos que pueden ser representados como un grafo o una red. Está escrito en Java, lo que permite a las aplicaciones basadas en JUNG hacer uso de las amplias capacidades incorporadas en su API de Java, así como de otras librerías de Java existentes en Internet. La página del proyecto JUNG esta disponible en el enlace: http://jung.sourceforge.net/. Para comenzar, consulta la documentación existente, Wiki, FAQ y ejemplos proporcionados.

2 RESUMEN

Partiendo de la práctica anterior, se reestructura y optimiza el código, además de añadir las nuevas funcionalidades, opciones de configuración y salidas de datos.

La descripción del desarrollo realizado explica no sólo los pasos realizados para resolver la funcionalidad optativa, sino también la nueva implementación desde el principio, ya que supone significativas mejoras con respecto a la versión anterior.

El diseño de esta nueva práctica pretende ser útil para su uso en el futuro, por lo que se implementa como una nueva librería.

Las funcionalidades introducidas son:

1. Uso de **8 parámetros de configuración**, incluyendo el valor de ventana entre ellos.
2. Los documentos de entrada se pueden obtener, además de partiendo de un directorio, a partir de un conjunto de URLs.
    a. En caso de sólo introducir una URL, el **crawler automáticamente extraerá nuevas URLs internas** (que apunten al mismo dominio) desde ésta, hasta obtener un mínimo necesario para trabajar, establecido en 20 URLs.
    b. Para intentar simular el ABSTRACT desde los documentos HTML, se obtiene la información de la etiqueta **meta description**, cuyo XPATH es "'//meta[@name="description"]/@content'"; y el contenido desde la etiqueta <body>.
3. **Soporte para dos idiomas: Inglés y Español**. Esta librería permite realizar todo el proceso a partir de documentos en ambos idiomas.
4. **La estructura de directorios se crea de forma automática**. Sólo es necesario dar la primera URL, o el directorio donde están los archivos. Bajo ese directorio se almacenarán todos los resultados.
5. **Extracción de keyphrases, dependiendo del valor de ventana**.

3 REQUISITOS TÉCNICOS

3.1 HERRAMIENTAS UTILIZADAS

La práctica se ha desarrollado en **Python 2.7** [1] con ayuda de las siguientes librerías:

- **NLTK** [2] para el tratamiento del texto y funciones de procesamiento del lenguaje.
- **Networkx** [3] para la creación, edición y almacenamiento de los grafos. Esta librería permite el cálculo del algoritmo TextRank [4] y la exportación a ficheros Pajek [5].
- **Matplotlib** [6] para la exportación de gráficos, en este caso, de imágenes de grafos (que sirven de ayuda para comprobar el estado y corrección de los mismos)
- **Itertools** [7] para ayudar en el tratamiento de objetos diccionario, vectores y eliminar letras repetidas en palabras.
- **Urllib2** [8], junto con **CookieLib** [9] para obtener el texto HTML del crawler.

- **Lxml** [10] para extraer la información de los nodos HTML.
- **Pickle** [11] para almacenar serializados los resultados de ejecución cuando se trabaja con muchos datos, simulando un sistema de cachés. Sólo útil para el proceso de debug.
- **Cleaner**, de Lxml para eliminar código Javascript de los textos extraídos en HTML.
- **Stanford POS Tagger** , para el cálculo de etiquetas, tanto en español como inglés.

Para la visualización y análisis de los grafos se utiliza **Gephi** [12] para visualizar los grafos resultantes.

4 DESARROLLO DE LA PRÁCTICA

4.1 PARÁMETROS DE CONFIGURACIÓN

El script de Python puede solicitar hasta 8 parámetros de configuración:

```
select_all_default = input("Select default values? (True/False): ")
if not select_all_default:
    language = raw_input("Select language (E english / S spanish): ")
    extract_from_url = input("Extract information from URL? (True/False): ")
    if extract_from_url:
        select_tagged_file = False
        file_selected = False
        print "If you want to extract the information from a list of URLs, " \
              "write the file name that contains that list:"
        file_path = raw_input("Input the path to de file: ")
    else:
        dir_with_data = raw_input("Input the path to the directory that contains the data:
")
        select_tagged_file = input("Select tagged file? (True/False): ")
        file_selected = raw_input("Add file ('False' selects all files): ")
    window = input("Add window number (values 1, ..., 5): ")
else:
    language = 'E'
    extract_from_url = False
    select_tagged_file = True
    file_selected = False
    dir_with_data = 'contags'
    select_tagged_file = False
    file_path = False
```

La idea es:

1. Pregunta si quiere establecer todos los valores por defecto (ejecuta el algoritmo usando los documentos de esta práctica con los tags ya calculados. En caso de 'T', se utiliza también el valor de ventana por defecto, 5.
2. Pregunta el idioma en el que estarán los documentos (o la información de la URL)
3. En caso de no utilizar los valores por defecto, se ofrece la posibilidad de extraer la información de dos formas:
    a. A partir de una lista de URLs. En este caso, pide también la ruta (o el nombre del fichero en caso de estar en el mismo directorio del proyecto) del fichero

que contiene las URLs, separando cada una por un salto de línea o un espacio.

b. A partir de un directorio que contiene los ficheros con los documentos. En este caso, se debe especificar la ruta del directorio, el nombre del fichero (o False para todos) y si el texto de entrada viene ya etiquetado o no.

4.2 EXTRACCIÓN DE INFORMACIÓN A PARTIR DE URLS. CRAWLER

En caso de seleccionar la opción de extraer la información a partir de una lista de URLs, se realizará un paso intermedio en ***Documento.py***:

- El script comienza seleccionando la/s url/s que hay en el fichero que las contiene:

```
def get_urls(self):
    for line in open(self.directory_data, 'r'):
        self.urls.append(line)
```

- Si el fichero contiene sólo una URL, primero se hace un bucle para rescatar nuevas URLs. Para ello, se extrae el contenido HTML de la primera URL, y se recorren todos los atributos "**href**" de los enlaces (etiquetas "a"). Si está bien formado (comprobándolo usando la misma función para esto de Django, con expresiones regulares) y además el dominio es el mismo que el de la primera URL, se añade a la lista. El proceso se repite usando la siguiente URL encontrada **hasta recorrer 5 páginas completas o encontrar 20 enlaces.**

```
if len(self.urls) == 1:
    count_loop = 0
    while len(self.urls) < 21:
        self.add_crawled_urls(self.urls[count_loop])
        count_loop += 1
        if count_loop == 10:
            break
```

(…)

```
def add_crawled_urls(self, url_seed):
    cookiejar = cookielib.LWPCookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))
    opener.addheaders = [('User-agent', 'Mozilla/5.0')]
    page = html.fromstring(opener.open(url_seed).read()) # html de la página
    for atag in page.xpath('//a/@href'):
        regex = re.compile(
            r'^(?:http|ftp)s?://' # http:// or https://
            r'(?:(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.)+(?:[A-Z]{2,6}\.?|[A-Z0-9-]{2,}\.?)|'
#domain...
            r'localhost|' #localhost...
            r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})' # ...or ip
            r'(?::\d+)?' # optional port
            r'(?:/?|[/?]\S+)$', re.IGNORECASE)
        existe = regex.match(atag)
        if not existe:
            print "Encuentro URL '" + atag + "' pero NO está bien formada"
        else:
            #suponemos que si es buena. Pero sólo quiero los enlaces hacia el mismo dominio.
            dominio_del_enlace = self.get_dir_name(atag)
            if dominio_del_enlace == self.dir_output_name:
                self.urls.append(atag)
                with open(self.directory_data, "a") as myfile:
                    myfile.write(atag + "\n")
        if(len(self.urls) > 21):
            break
```

- Una vez seleccionadas todas las URLs, se procede a la tarea de extracción de información. Para hacer el proceso más genérico y aprovechar la estructura dada en los documentos de la práctica, se extrae la meta descripción como un “ABSTRACT” y el contenido de la etiqueta “body” como el contenido general. Y se almacena siguiendo la misma estructura. Un ejemplo:

ABSTRACT

Las meninas - Colección - Museo Nacional del Prado Es una de las obras de mayor tamaño de Velázquez y en la que puso un mayor empeño para crear una composición a la vez compleja y creíble, que transmit...

1. INTRODUCTION

Cargando... El recorrido TITULORECORRIDO se ha creado correctamente. Añade obras desde la página (…)

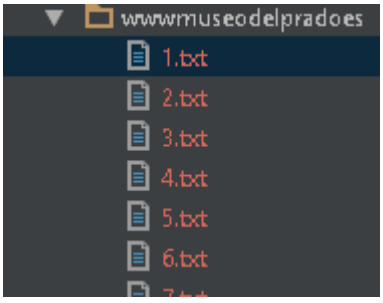
Para evitar el problema de código Javascript, se añade también la librería Cleaner.

- Para almacenar la información sin “entorpecer” otras salidas de programas, se crea automáticamente un directorio con el nombre obtenido a partir del dominio de la primera URL dada. Después, cada documento se almacena con un contador incremental: “1.txt”, “2.txt”, etc. Dentro del mismo directorio se irán almacenando también todos los resultados que se obtengan más adelante.

El código generado en este punto es el siguiente:

```
def get_documents(self):
    "Obtencion del codigo HTML"
    self.dir_output_name = self.get_dir_name(self.urls[0])
    functions_files.create_dir_if_not_exists(self.dir_output_name)
    count_name_files = 0
    if len(self.urls) == 1:
        count_loop = 0
        while len(self.urls) < 21:
            self.add_crawled_urls(self.urls[count_loop])
            count_loop += 1
            if count_loop == 10:
                break
    for url in self.urls:
        count_name_files += 1
        file_name = str(count_name_files) # calculo el nombre del archivo
        cookiejar= cookielib.LWPCookieJar()
        opener= urllib2.build_opener( urllib2.HTTPCookieProcessor(cookiejar) )
        opener.addheaders = [('User-agent', 'Mozilla/5.0')]
        page = html.fromstring(opener.open(url).read()) # html de la página
        title = ''
        for atag in page.xpath('//title'):
            title += atag.text_content() + " "
        title = ' '.join(title.split())
        meta_description = ''
        for atag in page.xpath('//meta[@name="description"]/@content'):
            meta_description += atag + " "
        meta_description = ' '.join(meta_description.split())
        body_content = ''
        for atag in page.xpath('//body'):
            body_content += atag.text_content() + " "
        body_content = ' '.join(body_content.split())

        file_write = self.dir_output_name + '/' + file_name + '.txt'
        with open(file_write, "a") as myfile:
            myfile.write('ABSTRACT' + "\n")
            myfile.write(title + ' ' + meta_description + "\n")
            myfile.write('1. INTRODUCTION' + "\n")
            myfile.write(body_content)
```

- ✓ Hasta este punto, tenemos un directorio con los documentos (HTML) que contienen la información a analizar. En el siguiente ejemplo vemos una captura con los resultados:



Al ejecutar desde ***main.py***, la clase documento contiene el atributo con el nombre del directorio calculado, que se utilizará por la clase ***SimpleTextRank*** para obtener los resultados. Así, el proceso de cálculo de TextRank es independiente del idioma (español o inglés) y de la procedencia de los documentos.

```
if extract_from_url:
    print ("Primer paso. Extracción de información a partir de URLs")
    print ("Fichero seleccionado: " + file_path)
    print ("Idioma de los documentos HTML: " + language)
    documentos = Documento(file_path)
    dir_with_data = documentos.dir_output_name
```

4.3 IMPLEMENTACIÓN DE LA CLASE SIMPLETEXTRANK. CÁLCULO DE TEXTRANK Y OBTENCIÓN DE GRAFOS Y KEYPHRASES PARA UN VALOR DE VENTANA DADO.

4.3.1 Etapa de preprocesado.

La siguiente tarea parte de un directorio que contiene los documentos. El primer paso es inicializar la clase con los valores definidos, que indican: idioma en el que están los documentos, si los documentos ya tienen etiquetas o no, si se quieren analizar todos los ficheros del directorio o uno en concreto y el valor de ventana.

```
keywords = SimpleTextRank()
keywords.select_tagged_file = select_tagged_file
keywords.file_selected = file_selected
keywords.dir_with_data = dir_with_data
keywords.window = window
keywords.language = language
keywords.execute()
```

Los pasos a seguir en esta tarea son

1. **Limpieza: eliminación de etiquetas HTML, conversión a minúsculas, eliminación de tildes**
2. **Tokenización**
3. **Cálculo de etiquetas**
4. **Filtrado**.

En este paso, se rellenarán las variables:

```
self.texts_not_tagged_abstract = {}
self.texts_not_tagged_content = {}
self.texts_tagged_abstract = {}
self.texts_tagged_content = {}
self.texts_filtered_abstract = {}
self.texts_filtered_content = {}
```

- Primero se almacenan en “text_not_tagged_*” los textos, que pueden estar en formato HTML de contenido y, si existe, de abstract.
- Después se tokenizan y calculan las tags utilizando el etiquetador “**Stanford POS Tagger**”, tanto para inglés como español, y se almacenan en las variables “texts_tagged_*”
    - Cada etiquetador (español o inglés) utiliza un modelo distinto, que se explicará más adelante.
- Finalmente, se filtran las etiquetas correspondientes a nombres y adjetivos, y se almacenan en “texts_filtered_*”.

Extraemos sólo las partes del código más destacables.

```
text = text.decode('utf-8').encode('ascii',
errors='ignore').replace("ABSTRACT", "").replace('1. INTRODUCTION', "")
text_abstract = text_abstract.replace("ABSTRACT", "").replace('1.
INTRODUCTION', "")

self.texts_not_tagged_content[file] = text
self.texts_not_tagged_abstract[file] = text_abstract

# calculate tags
self.texts_tagged_content[file] = self.calcula_tags(text)
self.texts_tagged_abstract[file] = self.calcula_tags(text_abstract)
```

El proceso más importante en este caso es el cálculo de etiquetas, que primero limpia, en caso de existir, el código HTML y después utiliza el modelo correspondiente a cada idioma.

```
def calcula_tags(self, token):
    self.model = 'C:\stanford-postagger-full-2016-10-31\models\english-caseless-left3words-
distsim.tagger'
    if self.language == 'S':
        self.model = 'C:\stanford-postagger-full-2016-10-31\models\spanish-distsim.tagger'

    # token = token.decode('utf-8')
    token = self.cleanhtml(token) # elimino etiquetas HTML
    token = re.sub(r'\([^)]*\)', '', token) # elimino paréntesis y su contenido
    tokenizer = RegexpTokenizer(r'\w+')
    token = tokenizer.tokenize(token)
    token = token[:500]
    tags = StanfordPOSTagger(self.model, path_to_jar=self.jar, encoding='utf8').tag(token)
    text_tagged = ''
    for item in tags:
        text_tagged += item[0] + '/' + item[1] + ' '

    return text_tagged
    # token = string.decode('utf-8')
    # token = nltk.word_tokenize(token)
    # tags =  nltk.pos_tag(token)
    # return tags
```

- En caso de la ejecución en español, en la etapa de preprocesamiento **se eliminan las tildes**, para “normalizar” mejor el sistema.
- Para hacer las ejecuciones más rápidas, se utilizan sólo las primeras 500 palabras del contenido de cada documento.

4.3.2 Cáculo de TextRank

Para el cálculo del algoritmo TextRank se utiliza la misma lógica implementada en la práctica obligatoria, siguiendo los pasos descritos en Mihalcea .

La función que realiza todo el cálculo es **self.keywords_extraction(use_abstract)**. La variable use_abstract indica si se está trabajando con ABSTRACT o no.

```
for name, item in loop_tagged_attributes.iteritems():
    for file_name, array_ in item.iteritems():
        words = []  # array con las palabras de cada archivo ya filtradas
        for word in array_:
            words.append(word)
        words = [x.lower() for x in words]

        # word_set_list = list(grams_list)
        unique_word_set = self.unique_everseen(words)
        word_set_list = list(unique_word_set)
        # pprint(word_set_list[:10])

        graph = self.create_graph(words, word_set_list)

        calculated_page_rank = nx.pagerank(graph, weight='weight') # cálculo
de textRank con pesos (los pesos están en graph)

        self.save_textrank_results(calculated_page_rank, name, file_name)
```

Las variables de los bucles contienen:

- **name**: “ABSTRACT/CONTENT”
- **ítem**: diccionario con los datos a analizar:
    - **file_name**: Nombre del archivo actual
    - **array_**: Lista de palabras ya procesadas en ese archivo

Una vez pasadas a minúsculas las palabras, la función “**self.unique_everseen(words)**”, devuelve un array con las palabras únicas del array de entrada, sin perder el orden de aparición.

Después, la función “**self.create_graph(words, word_set_list)**” crea el grafo que servirá de entrada al algoritmo de cálculo de TextRank. Tiene en cuenta, como indica el artículo de Mihalcea, la ventana de entrada de datos.

```
def create_graph(self, words, word_set_list):
    # creo un grafo como explica el paper, teniendo en cuenta el valor de
ventana.
    # si divido words en n-gramas, siendo n la ventana, obtengo la lista de
palabras que comparar. Por ejemplo:
    # si ventana = 2, words = [A, B, C, D, E, B, G] y keywords = [B, C, G]
    # obtengo los bigramas AB, BC, CD, DE, EB, BG.
    # De ahi selecciono los que existen en keywords y los anado: [BC, BG].
    # Para anadirlos, separo las palabras por un espacio: B + " " + G
    # * Si ventana = 3, hago primero ventana=2 y luego ventana=3
    print "Ngrams (window) selected: " + str(self.window)
    grams_ = ngrams(words, self.window)
    grams_ = list(grams_)
    graph = nx.Graph()
    array_valores = {}
    for i, val in enumerate(grams_):
```

```
        ant = 0
        for k in range(1, self.window):
            # print "Anado " + str(val[ant]) + " - " + str(val[k]) +
            # " con posiciones en : " + str(ant) + " - " + str(k)
            key = val[ant] + '/' + val[k]
            if key in array_valores:
                array_valores[key] += 1
            else:
                array_valores[key] = 1
            ant += 1
    # pprint(array_valores)
    for key, value in array_valores.items():
        clave = key.split("/")
        graph.add_edge(clave[0], clave[1], weight=value)


    return graph
```

Para ello, utilizo los n-gramas (con “n”, valor de ventana) para crear el grafo, de manera que las palabras estén interconectadas en el grafo por orden de coaparición.

Una vez creado el grafo, **al que hemos añadido pesos**, se procede al cálculo del algoritmo en sí.

- Podríamos haber obviado los pesos, como ocurre en la implementación por defecto existente en NLTK, pero de esta manera utilizamos el cálculo con pesos, que es más completo.

Los resultados obtenidos se almacenan automáticamente bajo el directorio “**calculated_textrank**”, dentro de, como siempre, el directorio de los documentos.

4.3.3 Extracción de keyphrases

El siguiente paso es la extracción de keyphrases.

```
keywords = sorted(calculated_page_rank, key=calculated_page_rank.get,
reverse=True)
keywords = keywords[0:((len(word_set_list) / 3) + 1)]
# pprint(keywords)

selected_keywords = self.select_keywords_windowed(words, keywords)
```

Ordenando de forma inversa la lista obtenida, como indica en el artículo de Mihalcea, y seleccionando 1/3 de las palabras obtenidas, se obtiene una lista con las palabras clave. Después, se seleccionan las palabras teniendo en cuenta el valor de ventana.

```
def select_keywords_windowed(self, words, keywords):
    # si divido words en n-gramas, siendo n la ventana, obtengo la lista de
palabras que comparar. Por ejemplo:
    # si ventana = 2, words = [A, B, C, D, E, B, G] y keywords = [B, C, G]
    # obtengo los bigramas AB, BC, CD, DE, EB, BG.
    # De ahí selecciono los que existen en keywords y los anado: [BC, BG].
    # Para anadirlos, separo las palabras por un espacio: B + " " + G
    selected_keywords = []
    grams_ = ngrams(words, self.window)
    grams_ = list(grams_)
    for i, val in enumerate(grams_):
        keywords_aux = []
        aux = True
        for k in range(self.window):
```

```
            keywords_aux.append(val[k])
            if val[k] not in keywords:
                aux = False
        if aux:
            new_keyword = " ".join(keywords_aux)
            selected_keywords.append(new_keyword)

    return selected_keywords
```

El funcionamiento es similar al proceso de creación del grafo, pero en este caso la salida es una lista de palabras de ventana dada.

Después, se almacenan las palabras en su directorio correspondiente.

4.3.4 Exportación de grafos

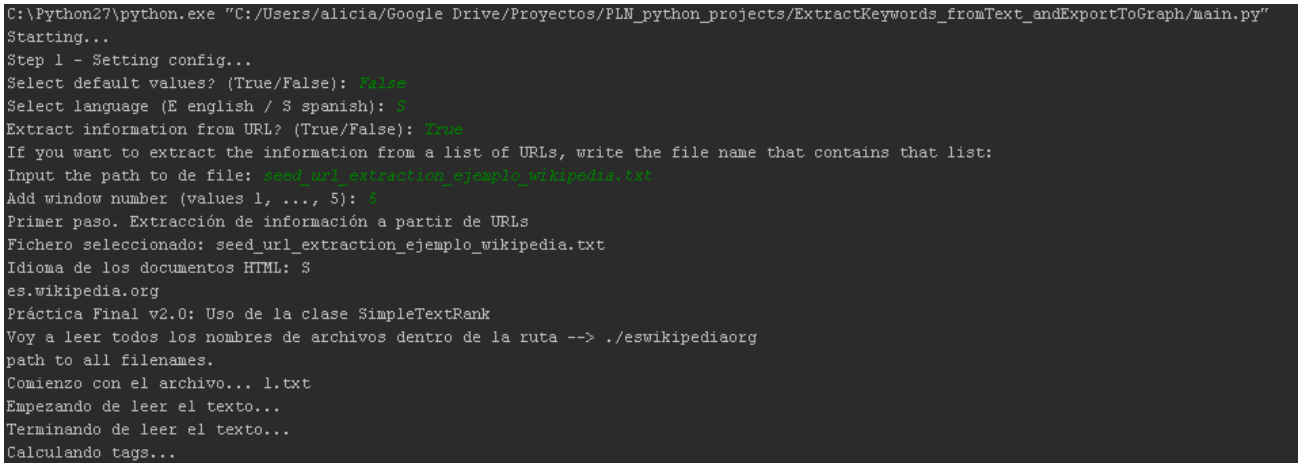
El último paso es la exportación a fichero PAJEK de los resultados.

```
graph_windowed = self.create_graph_windowed(selected_keywords, False)

self.paint_graph(graph_windowed, self.dir_with_data +
'/grafos_sin_pesos/capturas/ventana_' + str(self.window) + '/' + file_name_ +
'.png', False)

nx.write_pajek(graph_windowed, self.dir_with_data +
'/grafos_sin_pesos/ventana_' + str(self.window) + '/' + file_name_ + '.net')

functions_files.replace(self.dir_with_data + '/grafos_sin_pesos/ventana_' +
str(self.window) + '/' + file_name_ + '.net', '0.0 0.0 ellipse', '')


graph_windowed = self.create_graph_windowed(selected_keywords, True)
```

5 RESULTADOS OBTENIDOS Y CONCLUSIONES

5.1 EJEMPLO CON CRAWLER

Vemos un ejemplo de ejecución final:



Aunque en la imagen anterior no se puede visualizar, se pueden obtener conclusiones a partir de los resultados:

C:\Python27\python.exe "C:/Users/alicia/Google Drive/Proyectos/PLN_python_projects/ExtractKeywords_fromText_andExportToGraph/main.py"
Starting...
Step 1 - Setting config...
Select default values? (True/False): False
Select language (E english / S spanish): S
Extract information from URL? (True/False): True
If you want to extract the information from a list of URLs, write the file name that contains that list:
Input the path to de file: seed_url_extraction_ejemplo_wikipedia.txt
Add window number (values 1, ..., 5): 5
Primer paso. Extracción de información a partir de URLs
Fichero seleccionado: seed_url_extraction_ejemplo_wikipedia.txt
Idioma de los documentos HTML: S
es.wikipedia.org
Práctica Final v2.0: Uso de la clase SimpleTextRank
Voy a leer todos los nombres de archivos dentro de la ruta --> ./eswikipediaorg
path to all filenames.
Comienzo con el archivo... 1.txt
Empezando de leer el texto...
Terminando de leer el texto...
Calculando tags...
Comienzo con el archivo... 2.txt
Empezando de leer el texto...
Terminando de leer el texto...
Calculando tags...
Comienzo con el archivo... 3.txt
Empezando de leer el texto...
Terminando de leer el texto...
Calculando tags...
Comienzo con el archivo... 4.txt
Empezando de leer el texto...
Terminando de leer el texto...
Calculando tags...
Comienzo con el archivo... 5.txt
Empezando de leer el texto...
Terminando de leer el texto...
Calculando tags...
Comienzo con el archivo... 6.txt
Empezando de leer el texto...
Terminando de leer el texto...
Calculando tags...
Ngrams (window) selected: 5
graph has 31 nodes with 27 edges

graph has 31 nodes with 27 edges
Ngrams (window) selected: 5
graph has 32 nodes with 38 edges
graph has 32 nodes with 38 edges
Ngrams (window) selected: 5
graph has 26 nodes with 24 edges
graph has 26 nodes with 24 edges
Ngrams (window) selected: 5
graph has 31 nodes with 30 edges
graph has 31 nodes with 30 edges
Ngrams (window) selected: 5
graph has 26 nodes with 23 edges
graph has 26 nodes with 23 edges
Ngrams (window) selected: 5
graph has 39 nodes with 35 edges
graph has 39 nodes with 35 edges
Ngrams (window) selected: 5
graph has 0 nodes with 0 edges
graph has 0 nodes with 0 edges
Ngrams (window) selected: 5
graph has 0 nodes with 0 edges
graph has 0 nodes with 0 edges
Ngrams (window) selected: 5
graph has 0 nodes with 0 edges
graph has 0 nodes with 0 edges
Ngrams (window) selected: 5
graph has 0 nodes with 0 edges
graph has 0 nodes with 0 edges
Ngrams (window) selected: 5
graph has 0 nodes with 0 edges
graph has 0 nodes with 0 edges
Ngrams (window) selected: 5
graph has 0 nodes with 0 edges
graph has 0 nodes with 0 edges

En primer lugar, viendo la salida generada por el programa, la idea de obtener de la meta descripción el ABSTRACT suele devolver muy pocos resultados con este algoritmo. Una mejora futura sería obtener estas palabras clave aplicando un algoritmo distinto.

Podemos ver un ejemplo de salida del programa para el caso de la obra “Las Meninas” de Velázquez:

- Palabras clave para una ventana de 5 (Museo del Prado):

museo meninas velazquez diego rodriguez
meninas velazquez diego rodriguez silva
descargar ficha obra descargar uso
obras tienda prado velazquez diego
tienda prado velazquez diego rodriguez
prado velazquez diego rodriguez silva
diego velazquez diego silva velazquez
compleja crea ble sensacia³n vida
crea ble sensacia³n vida realidad
ble sensacia³n vida realidad tiempo
sensacia³n vida realidad tiempo densa
vida realidad tiempo densa red
realidad tiempo densa red significados
guardadamas fondo puerta josa nieto
fondo puerta josa nieto aposentador
puerta josa nieto aposentador espejo
josa nieto aposentador espejo reflejados
nieto aposentador espejo reflejados rostros

aposentador espejo reflejados rostros felipe
espejo reflejados rostros felipe iv
reflejados rostros felipe iv mariana

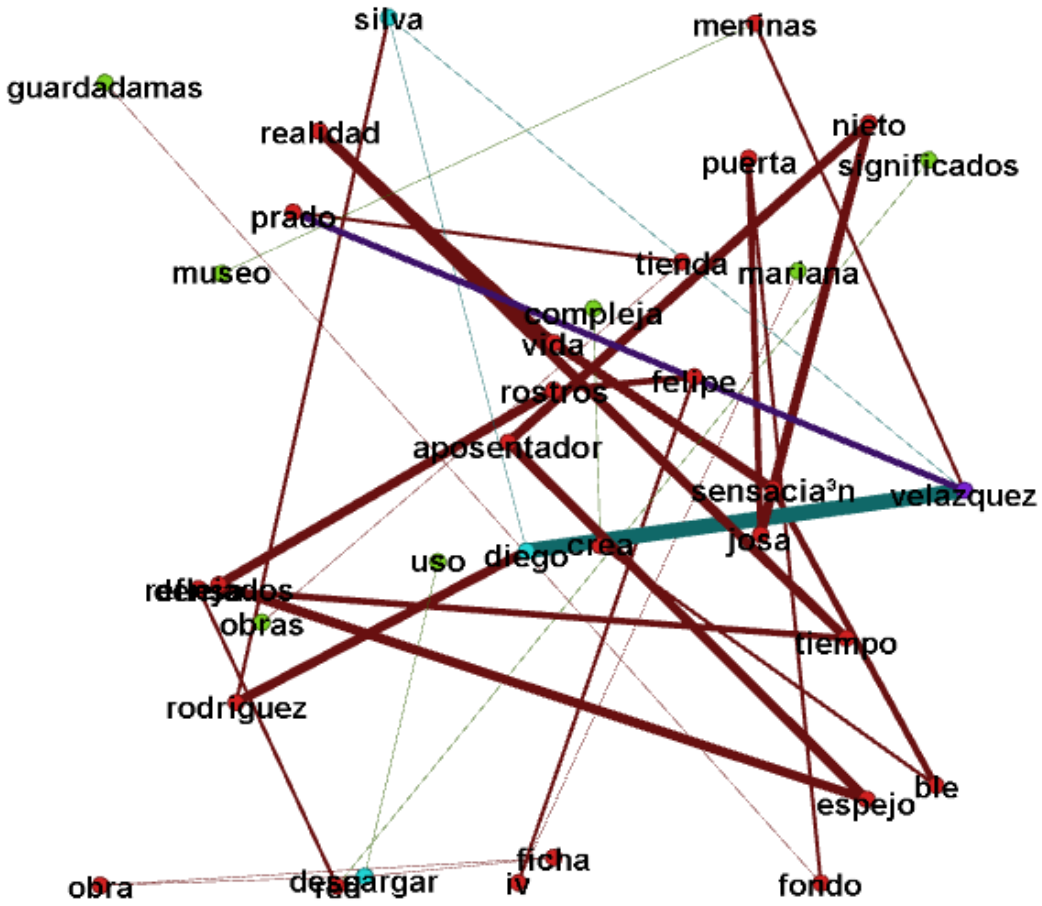
- Palabras clave para una ventana de 5 (Wikipedia):

baºsqueda meninas familia felipe iv
siglo familia felipe iv segaºn
familia felipe iv segaºn inventario
felipe iv segaºn inventario obra
oportunidad palacio alcanza³ auta ntica
primera defendida stirling maxwell carl
defendida stirling maxwell carl justi
stirling maxwell carl justi pona
maxwell carl justi pona acento
parta cipe dina mico representacia³n
cipe dina mico representacia³n tema
dina mico representacia³n tema central
mico representacia³n tema central retrato
representacia³n tema central retrato infanta

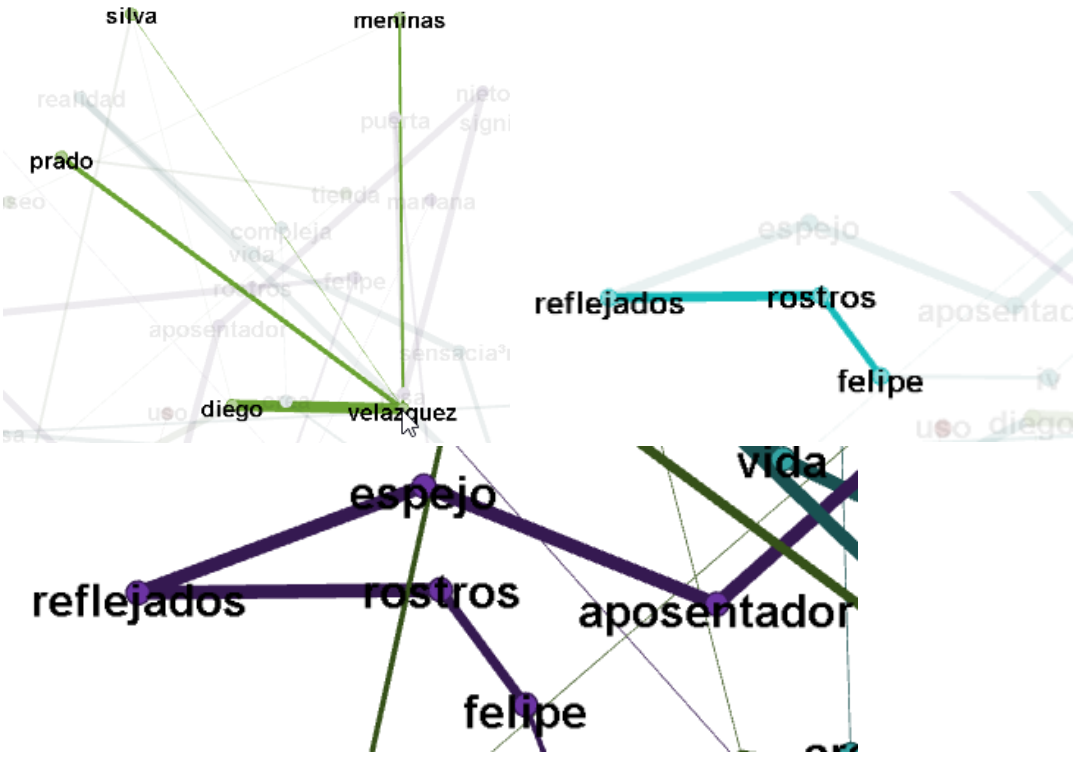
Como se puede comprobar, hay varios problemas latentes:

1. La obtención del texto a partir de la etiqueta <body> devuelve muchos resultados "basura". Quizás una buena idea habría sido obtener el texto a partir de la etiqueta que más contenido en texto plano tuviera.
2. El problema de codificación.

Por otro lado, en Gephi el grafo (con pesos) es el siguiente:



Se puede ver algo muy interesante si particionamos por grado medio: El nombre completo del autor de la obra. Visto de otro modo, están fuertemente conexos. Vemos otras relaciones interesantes:

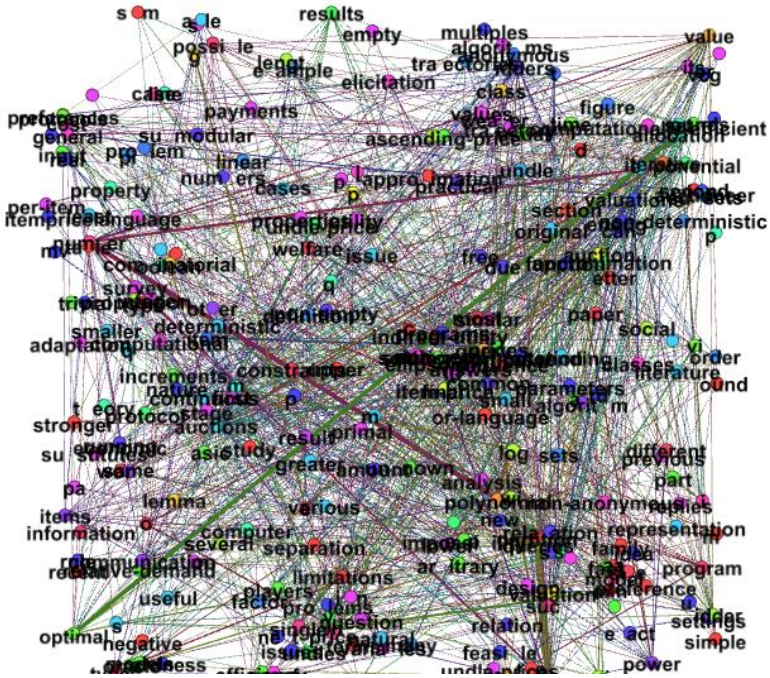


5.2 EJEMPLO CON LOS DOCUMENTOS DE LA PRÁCTICA OBLIGATORIA

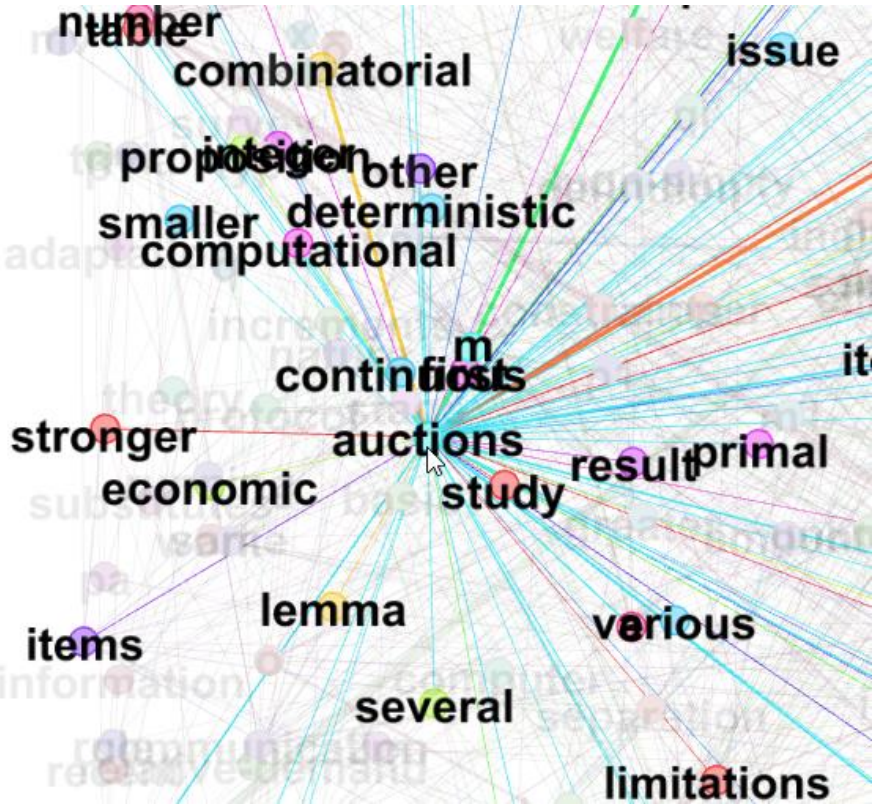
Ejemplo de ejecución:

```
C:\Python27\python.exe "C:/Users/alicia/Google Drive/Proyectos/PLN_python_projects/ExtractKeywords_fromText_andExportToGraph/main.py"
Starting...
Step 1 - Setting config...
Select default values? (True/False): False
Select language (E english / S spanish): E
Extract information from URL? (True/False): False
Input the path to the directory that contains the data: contags
Select tagged file? (True/False): True
Add file ('False' selects all files): False
Add window number (values 1, ..., 5): 5
Práctica Final v2.0: Uso de la clase SimpleTextRank
Voy a leer todos los nombres de archivos dentro de la ruta --> ./contags
path to all filenames.
Comienzo con el archivo... C-41.txt.tagged
Comienzo con el archivo... C-42.txt.tagged
Comienzo con el archivo... C-44.txt.tagged
```

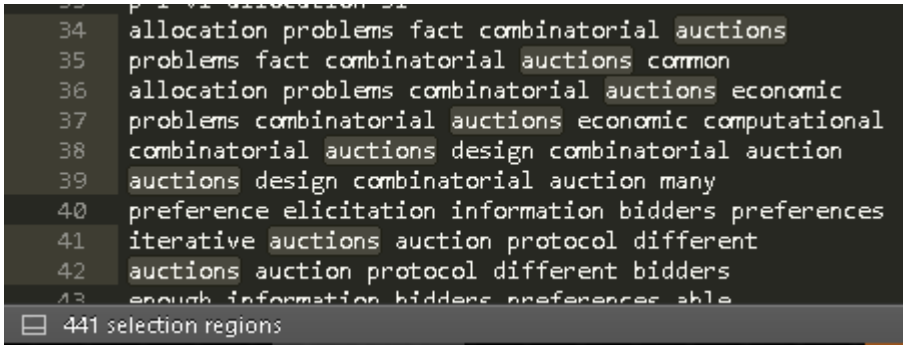
Si seleccionamos entre los resultados, por ejemplo, el archivo J-47, vemos el grafo con pesos que genera:



Echando un vistazo a los resultados que devuelve este grafo en el algoritmo de TextRank calculado, vemos que la palabra con mayor peso es “auctions”. Lo visualizamos en Gephi:



Si buscamos entre las keyphrases, vemos que esta palabra aparece 447 veces:



Por lo que se puede comprobar que los resultados son correctos.

6 TRABAJO FUTURO

El cálculo del algoritmo TextRank puede ser muy útil para la obtención de keyphrases y, a su vez, el estudio visual del contenido de los documentos.

Además de las posibilidades de mejorar el trabajo realizado, se incluyen los problemas que no están solucionados en éste.

- Codificación de archivos. A veces, al trabajar en distintos idiomas y desde distintas fuentes, es necesario comentar/decomentar algunas codificaciones en UTF8 para obtener los resultados correctos.
- Selección de palabras cuando la cantidad es excesiva. Una posibilidad podría ser, por ejemplo, utilizar la matriz TF-IDF para seleccionar sólo las mejores N palabras. En este caso, los resultados no son todo lo precisos que debieran por haber elegido sólo una parte (las primeras 500 palabras) del contenido.
- La obtención del texto a partir de la etiqueta <body> devuelve muchos resultados “basura”. Quizás una buena idea habría sido obtener el texto a partir de la etiqueta que más contenido en texto plano tuviera.

Una **posible aplicación para este trabajo** podría ser la búsqueda de tags, o palabras clave, para páginas Web. Ayudándonos de otros algoritmos (TF-IDF o KLD), se podrían buscar las palabras que mejor definan al documento. Pensando en el análisis en sí de los grafos resultantes, y pensando analizar, por ejemplo, una página Web propia, se podría utilizar para plantearse la diversidad de palabras clave en los documentos, para optimizar los resultados en motores de búsqueda. Si ya se tienen las palabras clave, y se puede obtener la estructura de enlaces internos de las páginas existentes, se podría controlar qué páginas están enlazadas y si su contenido es similar. O encontrar posibles enlaces que aún no existen, lo que resulta un valor añadido en temas de posicionamiento Web (SEO Off-Page).

7 REFERENCIAS

1. Python.org, https://www.python.org/.
2. Natural Language Toolkit — NLTK 3.2.4 documentation, http://www.nltk.org/.
3. NetworkX for Python, https://networkx.github.io/.
4. Mihalcea, R., Tarau, P.: TextRank: Bringing order into texts. Proc. EMNLP. 85, 404–411 (2004).
5. Input Pajek file description.
6. Matplotlib: Python plotting — Matplotlib 2.0.2 documentation, https://matplotlib.org/.
7. 9.7. itertools — Functions creating iterators for efficient looping — Python 2.7.13 documentation, https://docs.python.org/2/library/itertools.html.
8. 20.6. urllib2 — extensible library for opening URLs — Python 2.7.13 documentation, https://docs.python.org/2/library/urllib2.html.
9. 20.21. cookielib — Cookie handling for HTTP clients — Python 2.7.13 documentation, https://docs.python.org/2/library/cookielib.html.
10. lxml - Processing XML and HTML with Python, http://lxml.de/.
11. 11.1. pickle — Python object serialization — Python 2.7.13 documentation, https://docs.python.org/2/library/pickle.html.
12. Gephi - The Open Graph Viz Platform, https://gephi.org/.