Γλωσσική Τεχνολογία

Project

2021 - 2022

Στυλιανός Στυλιανάκης

1059713

Link to code

ΜΕΡΟΣ Α

Προσκομιστής Ιστοσελίδων & Προεπεξεργασία Δεδομένων

Με τη χρήση του **Scrapy** δημιουργήθηκε ο Crawler, με τη βοήθεια του οποίου ανακτώνται άρθρα από τις ειδησεογραφικές ιστοσελίδες *CNN* και *CNBC*. Πιο συγκεκριμένα, μέσα στο αρχείο **spiders.py** ορίστηκαν δύο **spiders**, το **cnbc_spider** και το **cnn_spider,** τα οποία είναι υπεύθυνα για την εξαγωγή άρθρων από τις αντίστοιχες ιστοσελίδες.

Ξεκινώντας από τα προκαθορισμένα **start_urls**, ο crawler ακολουθεί όλους τους συνδέσμους σε άρθρα που βρίσκονται στην διαδρομή του xpath που έχει οριστεί μέσα στη συνάρτηση **parse**, όπως φαίνεται παρακάτω.

```
class CnbcSpider(scrapy.Spider):
    name = 'cnbc_spider'
    start_urls = [
        'https://www.cnbc.com/world',
        'https://www.cnbc.com/business',
        'https://www.cnbc.com/technology',
        'https://www.cnbc.com/politics'
    ]

    def parse(self, response):
        # Get article URLs
        for link in response.xpath('//div[@class="Card-titleContainer"]/a/@href'):
            yield response.follow(link.get(), callback=self.parse_article)
```

Για κάθε έναν από τους συνδέσμους αυτούς τους που ακολουθεί καλείται η **callback function** `parse_article` ώστε να εξαχθούν οι ζητούμενες πληροφορίες.

```
def parse_article(self, response):
    # Extract article paragraphs
    par = response.xpath('//div[@class="ArticleBody-articleBody"]/div[@class="group"]/p').extract()
    # Ignore articles with empty paragraphs
    if par == []:
        return
    paragraphs = ''
    # Extract only the text of the paragraphs and add it to paragraphs string
    for paragraph in par:
        paragraphs += ' ' + extract_text(paragraph)

    # Create article dictionary by extracting required information
    article = {
        'url': response.url,
        'title': response.xpath('//h1/text()').extract_first(),
        'paragraphs': paragraphs
    }

    return article
```

Συγκεκριμένα για την εξαγωγή κειμένου από τις παραγράφους του άρθρου δημιουργείται ένα άδειο string **paragraphs**, καθώς και μία λίστα με όλες τις παραγράφους του άρθρου. Στη συνέχεια, αυτή η λίστα διατρέχεται και από κάθε παράγραφο εξάγεται το καθαρό κείμενο, το οποίο προστίθεται στο string **paragraphs.**

Αντίστοιχα έχει δημιουργηθεί και το **cnn_spider** με αλλαγές φυσικά στις διαδρομές xpath και άλλες μικρές αλλαγές.

Τα spiders αυτά τρέχουν μέσα από τη συνάρτηση **run** του αρχείου **run_Spiders.py**. Στο αρχείο αυτό δημιουργείται ένα subprocess για το κάθε spider και τρέχει ουσιαστικά το shell command “scrapy crawl *spider_name* -o articles_temp.json", δηλαδή κάνει crawl το κάθε spider και αποθηκεύει την έξοδο του στο αρχείο *articles_temp.json*.

```
def run(verbose=0):
    """Run all spiders"""

    cnbc_proc = subprocess.run(
        ["scrapy", "crawl", "cnbc_spider", "-o", "articles_temp.json"],
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True
    )

    cnn_proc = subprocess.run(
        ["scrapy", "crawl", "cnn_spider", "-o", "articles_temp.json"],
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        text=True
    )
```

Στη συνέχεια, επειδή το περιεχόμενο του αρχείου περιέχει **δύο λίστες** με άρθρα (μία από κάθε διεργασία), αντιγράφεται το αρχείο χωρίς τα σημεία «ένωσης» των λιστών. Χρησιμοποιώντας τη συνάρτηση `re.sub("\n\]\[", ",", infile.read())`, αντικαθίσταται η σειρά χαρακτήρων “\n][“ με ένα κόμμα, ενώνοντας έτσι τις λίστες σε **μία** και στη συνέχεια αποθηκεύεται στο καινούριο αρχείο *articles.json*.

```
# Cleanup file by merging article lists into 1
with open('articles_temp.json', 'r', encoding="utf-8") as infile, open('articles.json', 'w', encoding="utf-8") as outfile:
    temp = re.sub("\n\]\[", ",", infile.read())
    outfile.write(temp)
```

Μορφοσυντακτική Ανάλυση

Μετά που θα κατέβουν τα άρθρα και θα δημιουργηθεί το αρχείο *articles.json*, διαβάζονται και φορτώνονται σε ένα **dictionary** μέσω της συνάρτησης `readJSON`, ώστε να υποστούν στη συνέχεια επεξεργασία. Για τον **PoSTagger** χρησιμοποιήθηκε έτοιμη υλοποίηση της βιβλιοθήκης **nltk**, ενώ οι συναρτήσεις που χρησιμοποιούν τη βιβλιοθήκη αυτή βρίσκονται όλες στο αρχείο *nltk_functions.py*.

Παρακάτω φαίνεται πώς γίνονται PoSTag τα άρθρα (χωρίς να αφαιρεθούν τα stop words).

```
def pos_tag(articles):
    """PoS Tag articles"""

    pos_tags = {}
    for article in articles:
        # Split article paragraphs into sentences
        tokenized = sent_tokenize(article['paragraphs'])
        # Send the tokenized words of each sentence for PoS tagging
        pos_tag = process_content(tokenized)
        # Add the tags of each sentence to the pos tags list
        pos_tags[article['url']] = pos_tag
    return pos_tags, filter_stop_words(pos_tags)

def process_content(tokenized):
    """Tokenize each word of the
    input (tokenized) sentence"""

    try:
        tagged = []
        # PoS tag every tokenized sentence
        for sent in tokenized:
            words = word_tokenize(sent)
            tagged.append(nltk.pos_tag(words))
        return(tagged)

    except Exception as ex:
        print(str(ex))
```

Αναπαράσταση ιστοσελίδων στο Μοντέλο Διανυσματικού Χώρου.

Μόλις τελειώσει η διαδικασία του tagging, καλείται η συνάρτηση `filter_stop_words`, η οποία παίρνει σαν όρισμα τα **tags** που έχουν προκύψει από την προηγούμενη συνάρτηση, και από αυτά κρατάει μόνο εκείνα που **δεν είναι stop words**. Στη συνάρτηση αυτή γίνεται και η αρχικοποίηση, αλλά και η συμπλήρωση του λεξικού των **λημμάτων**.

```
def filter_stop_words(pos_tags):
    """Filter stop words from pos tags,
    lemmatize them and create lemmas dictionary"""

    pos_no_stopwords = {}
    lemmas = {}
    articles_w_count = {}
    lemmatizer = WordNetLemmatizer()

    # article[0] -> article url
    # article[1] -> article tags
    for article in pos_tags.items():
        article_pos_no_sw = []
        article_w_count = 0
        for sent in article[1]:
            filtered_pos = []
            for tag in sent:
                # Filter words that have not
                # been tagged with a closed
                # category tag
                if tag[1] not in oc_categories:
                    continue
```

Εδώ να σημειωθεί ότι θα ήταν πιο αποδοτικό αυτό το φιλτράρισμα να γινόταν κατά τη διάρκεια του αρχικού tagging, αλλά έγινε διαχωρισμός των δύο λειτουργιών επειδή στην εκφώνηση του πρότζεκτ παρουσιάζονται ως διαφορετικά υποσυστήματα.

Έπειτα από το **φιλτράρισμα** των stop words, τα εναπομείναντα tags προστίθενται στη λίστα με τα φιλτραρισμένα tags και μετά εξάγουμε το **λήμμα** της λέξης. Ταυτόχρονα, ελέγχουμε και τη **συχνότητα εμφάνισης** του λήμματος **στο άρθρο** μας (με τους κατάλληλους ελέγχους και ενέργειες αρχικοποίησης), καθώς και τον **αριθμό λέξεων** του κάθε **άρθρου**.

Τέλος, από τη συνάρτηση επιστρέφονται τα tags χωρίς τα stop words, το λεξικό του πλήθους λέξεων των άρθρων, αλλά και το λεξικό των λημμάτων.

```
                # Add tag to filtered_pos
                article_w_count += 1
                filtered_pos.append(tag)
                lemma = lemmatizer.lemmatize(tag[0], pos= get_wordnet_pos(tag[1])).lower()

                # If lemma doesn't already exist
                # in the lemmas dict, create it
                # and set the count for the
                # corresponding article to 1
                if lemma not in lemmas.keys():
                    lemmas[lemma] = {
                        article[0]: 1
                    }
                # If lemma has already been added
                # to the dict
                else:
                    # If lemma has previously been found in
                    # the same article, increase its count
                    if article[0] in lemmas[lemma].keys():
                        lemmas[lemma][article[0]] += 1
                    # Otherwise, create a new entry for this
                    # article and initialize its count to 1
                    else:
                        lemmas[lemma][article[0]] = 1
            article_pos_no_sw.append(filtered_pos)
        articles_w_count[article[0]] = article_w_count
        pos_no_stopwords[article[0]] = article_pos_no_sw

    return pos_no_stopwords, articles_w_count, lemmas
```

Δημιουργία του ευρετηρίου

Σε αυτή τη φάση έχει ουσιαστικά φτιαχτεί ένα σημαντικό **μέρος του ευρετηρίου**, απλώς αντί για το βάρος του λήμματος στα αντίστοιχα άρθρα έχουμε προς το παρόν μόνο τη **συχνότητα εμφάνισής** τους σε αυτά. Για τη **μετατροπή** από συχνότητα εμφάνισης σε βάρη υπάρχει η συνάρτηση `calculateTFidf` η οποία παίρνει σαν είσοδο το λεξικό των λημμάτων και το επιστρέφει μετά από τη μετατροπή. Η διαδικασία αυτή φαίνεται στο παρακάτω snapshot του κώδικα:

```python
def calculateTFidf(lemmas, article_w_count):
    """Calculate weights for all lemmas (using tf_idf)"""

    article_count = len(article_w_count)
    for lemma in lemmas:
        # IDF of each lemma is equal to the base 2 log of the
        # article count of our database devided by the count
        # of articles the lemma is present in
        idf = math.log2(article_count/len(lemmas[lemma].keys()))
        for key in lemmas[lemma].keys():
            tf = lemmas[lemma][key]/article_w_count[key]
            tf_idf = tf*idf
            lemmas[lemma][key] = tf_idf

    return lemmas
```

Αποθήκευση και επαναφόρτωση ευρετηρίου

Η **αποθήκευση** του ευρετηρίου γίνεται με τη συνάρτηση `createXML` του αρχείου *functions.py*, η οποία παρουσιάζεται παρακάτω

```python
def createXML(lemmas_dict):
    """Create and save XML file from lemmas dict"""

    root = minidom.Document()
    inv_index = root.createElement('inverted_index')
    root.appendChild(inv_index)

    for word in lemmas_dict.keys():
        new_lemma = root.createElement('lemma')
        new_lemma.setAttribute('name', word)
        for key in lemmas_dict[word].keys():
            new_document = root.createElement('document')
            new_document.setAttribute('url', str(key))
            new_document.setAttribute('weight', str(lemmas_dict[word][key]))
            new_lemma.appendChild(new_document)
        inv_index.appendChild(new_lemma)

    xml_str = root.toprettyxml(indent ='\t')

    with open('lemmas.xml', "w", encoding='utf-8') as file:
        file.write(xml_str)
```

Αντίστοιχα, η **επαναφόρτωση** του ευρετηρίου από το αρχείο γίνεται με τη μέθοδο `readXML` του ίδιου αρχείου.

Αξιολόγηση Ευρετηρίου

Ο **χρήστης** έχει την επιλογή να **εισάγει** ο ίδιος ερωτήματα ή να **δημιουργηθούν** αυτόματα, όπως φαίνεται από τον παρακάτω κώδικα:

```
queries = []
user_in = input("Enter queries. Leave blank to auto generate.\n")
if user_in == "":
    print('Generating queries...')
    queries = functions.generateRandomQueries(lemmas)
else:
    while user_in != "":
        queries.append(user_in)
        user_in = input()
```

Στη συνέχεια, τα ερωτήματα υποβάλλονται και το σύστημα αξιολογείται ως προς το **χρόνο απόκρισης** του. Για αυτόν το σκοπό χρησιμοποιήθηκε η βιβλιοθήκη time και συγκεκριμένα η συνάρτηση `perf_counter`. Ακριβώς πριν την κλήση της συνάρτησης υποβολής ερωτημάτων, αποθηκεύεται ο **χρόνος έναρξης** στη μεταβλητή `start_time` και αντίστοιχα μόλις απαντηθούν όλα τα ερωτήματα στη μεταβλητή `finish_time`. Είναι εμφανές ότι η διαφορά τους είναι και ο **συνολικός χρόνος απόκρισης** του ευρετηρίου μας. Για να βρούμε τον μέσο χρόνο απόκρισης, διαιρούμε τον συνολικό χρόνο με το πλήθος των ερωτημάτων που υποβλήθηκαν.

```
# Make queries and benchmark
# time required to find the article ids
print('Making queries...')
start_time = perf_counter()
query_response = {}
for query in queries:
    query_response[query] = nltk_functions.nltk_query(lemmas, query)
finish_time = perf_counter()
e_time = finish_time - start_time
```

Τέλος, εκτυπώνονται τα ερωτήματα με τις απαντήσεις τους, καθώς και οι μετρικές απόδοσης του ευρετηρίου.

```
# Custom pretty print
for item in query_response.items():
    print(item[0], ': {', sep="")
    for article in item[1].items():
        print('    ', article[0], ': ', article[1], ',', sep="")
    print('}')

print("Number of queries:", len(queries))
print("Total elapsed time: %.6f" % e_time)
print("Average response time: %.6f" % (e_time/len(queries)))
```

Η συνάρτηση των ερωτημάτων `nltk_query` δέχεται ως είσοδο το **λεξικό των λημμάτων**, καθώς και το **ερώτημα** και επιστρέφει τα άρθρα στα οποία βρίσκονται οι λέξεις του ερωτήματος, **ταξινομημένα** ως προς το (συνολικό) βάρος τους.

Αρχικά, «σπάει» το ερώτημα σε **λέξεις** (σε περίπτωση που είναι πολλών λέξεων) και στη συνέχεια τις ψάχνει μία μία σε **τρία στάδια**. Ξεκινάει την αναζήτηση ελέγχοντας αν η λέξη του ερωτήματος υπάρχει **όπως είναι** στο λεξικό των λημμάτων. Σε δεύτερο στάδιο, αν δεν υπάρχει δηλαδή, τη μετατρέπει σε **λήμμα** και έπειτα το αναζητεί στο λεξικό των λημμάτων. Αν ούτε αυτό δουλέψει, ως τελευταία (και πολύ χρονοβόρα) προσπάθεια γίνεται **σύγκριση** (string-matching) **όλων των λημμάτων** με τη λέξη και όποια λήμματα ταιριάζουν αρκετά θεωρούνται ως σωστή απάντηση.

```python
def nltk_query(lemmas, query):
    """Query function that takes as input the lemmas dict as well as a query
    (which can be one or multiple words) and returns the articles in which
    the query word(s) can be found, sorted by their weight sums."""

    lemmatizer = WordNetLemmatizer()
    answer = {}
    articles_containing_query = []

    # Split the query into words and search for them in the articles
    for word in query.split():
        # Try finding answer by directly looking for query word in the dictionary keys.
        if word in lemmas:
            articles_containing_query = lemmas[word].items()
        # If this doesn't work, try finding answer by lemmatizing query words.
        else:
            token = nltk.tag.pos_tag([word])[0]
            qword_lemma = lemmatizer.lemmatize(token[0], pos= get_wordnet_pos(token[1]))
            if qword_lemma in lemmas:
                articles_containing_query = lemmas[qword_lemma].items()
            # If this doesn't work either, try finding answer using string matching
            else:
                for lemmas_key, lemmas_value in lemmas.items():
                    ratio = fuzz.ratio(qword_lemma, lemmas_key)
                    if ratio > 90:
                        articles_containing_query.append(lemmas_value.items())
```

Αφού βρει σε ποια άρθρα περιέχεται η λέξη του ερωτήματος, η συνάρτηση διατρέχει αυτά τα άρθρα και δημιουργεί την **τελική απάντηση**. Τέλος, πριν επιστραφεί η απάντηση, γίνεται **ταξινόμηση** των άρθρων με βάση το βάρος τους, μέσω της συνάρτησης `dict_sort`.

```
    # Add the weight of the word in each article to the answer[article]
    # so that if multiple words of a single query are found in the
    # same articles their weights get summed
    for article, weight in articles_containing_query:
            if article in answer:
                answer[article] += weight
            else:
                answer[article] = weight

# Answer not found by lemmatizing OR string matching
if len(answer.keys()) == 0:
    return {query :'Not found'}

return dict_sort(answer)
```

Μέρος Β

Για το μέρος Β χρησιμοποιήθηκε η βιβλιοθήκη **sklearn**, καθώς διαθέτει υλοποιημένες συναρτήσεις με τις οποίες πραγματοποιήθηκε το κομμάτι αυτό της εργασίας.

Προ-επεξεργασία των συλλογών Ε και Α

Για την φόρτωση των εγγράφων σε τυχαία σειρά χρησιμοποιήθηκε η παρακάτω συνάρτηση του `sklearn.datasets`.

```
# Read dataset and load it into variable
twenty_news_train = fetch_20newsgroups(subset='train', shuffle=True, random_state=21)
```

Δημιουργία χώρου χαρακτηριστικών

Η δημιουργία του χώρου χαρακτηριστικών, καθώς και των διανυσμάτων έγινε με τη συνάρτηση vectorize_Documents, η οποία παίρνει σαν όρισμα το training dataset που φορτώσαμε.

```
# Vectorize dataset
tf_idf_vect = vectorize_Documents(twenty_news_train)
```

Όπως φαίνεται παρακάτω, χρησιμοποιείται ένας `CountVectorizer` και πιο συγκεκριμένα η συνάρτηση `fit_transform` ώστε να γίνει tokenization και φιλτράρισμα των λέξεων των κειμένων. Στη συνέχεια, επιλέγονται τα χαρακτηριστικά και τέλος μετατρέπονται τα κείμενα σε διανύσματα πλήθους.

```
def vectorize_Documents(dataset):
    """Finds features of a document dataset and
    transforms documents to feature vectors, using
    tf-idf"""
    # CountVectorizer does the preprocessing, tokenization and stopword filtering
    # and builds a feature dictionary, transforming documents to feature vectors.
    features_count = count_Vectorizer.fit_transform(dataset.data)
```

Δημιουργία διανυσμάτων χαρακτηριστικών

Επειδή όμως ως συνήθως το απλό πλήθος δεν είναι χρήσιμη μετρική, οι διαστάσεις των διανυσμάτων μετατρέπονται σε **tf_idf** αξιοποιώντας έναν `TfidfTransformer`, όπως φαίνεται παρακάτω.

```
    # TfidfTransformer is used to transform the count of features to tf-idf
    # in each vector.
    features_tfidf = tf_idf_Transformer.fit_transform(features_count)
    return features_tfidf
```

Σύγκριση διανυσμάτων χαρακτηριστικών

Με χρήση Multinomial Classifier

Ένας τρόπος κατηγοριοποίησης είναι να χρησιμοποιήσουμε υλοποιημένο classifier. Πριν τη σύγκριση των διανυσμάτων χαρακτηριστικών από το test dataset, θα πρέπει να φτιάξουμε και να εκπαιδεύσουμε έναν **classifier**.

```
# Train classifier to later predict article category
classifier = train_Classifier(twenty_news_train.target, tf_idf_vect)
```

Αυτό επιτυγχάνεται με τη συνάρτηση fit, όπως φαίνεται παρακάτω.

```
def train_Classifier(d_categories, tf_idf_vector):
    """Trains a model to classify categories using their feature vector"""
    t_classifier = MultinomialNB().fit(tf_idf_vector, d_categories)
    return t_classifier
```

Συγκρίνουμε τα κείμενα μέσω της συνάρτησης `predict_Category_Clf`.

```
# Predict category of test documents from the dataset
twenty_news_test = fetch_20newsgroups(subset='test', shuffle=True, random_state=21)
predict_Category_Clf(twenty_news_test.data, classifier)
```

Η συνάρτηση αυτή, δημιουργεί διανύσματα με τα tf_idf χαρακτηριστικά του κάθε κειμένου και χρησιμοποιεί τον **classifier** που έχουμε εκπαιδεύσει για τη σύγκριση, ώστε να κατηγοριοποιήσει το κάθε κείμενο.

```
def predict_Category_Clf(documents, classifier):
    """Predicts the category of a list of documents"""

    features_count = count_Vectorizer.transform(documents)
    features_tfidf = tf_idf_Transformer.transform(features_count)
    predicted = classifier.predict(features_tfidf)

    for doc, category in zip(documents, predicted):
        print('%r\n%s\n' % (doc, twenty_news_train.target_names[category]))
```

Ο χρόνος που χρειάζεται για να κατηγοριοποιήσει και τα 7532 άρθρα και να εκτυπώσει είναι περίπου 36.5”. Χωρίς την εκτύπωση, ο χρόνος μειώνεται δραματικά στα περίπου 5.5”. Τα πειράματα εκπονήθηκαν σε λάπτοπ με επεξεργαστή Intel Core i7 8550u (15w) με base frequency 1.8GHz.

```
Articles: 7532
Elapsed time: 36.48
```

```
Articles: 7532
Elapsed time: 5.48
```