Università del Piemonte Orientale

Dipartimento di Scienze e Innovazione Tecnologica

Corso di Studi in Informatica

Relazione per la prova finale

Sviluppatori e piattaforme cloud conversazionali: utilizzo e difficoltà riscontrate tramite l'analisi di StackOverflow

Tutore interno:

Prof. Luigi De Russis

Candidato:

Matteo Cacciamali

Anno Accademico 2017/18

Indice

Abstract

Con questo progetto si vuole raccogliere e organizzare i dati relativi alle problematiche degli sviluppatori sull'utilizzo di una piattaforma cloud conversazionale, quale Dialogflow, raccogliendo e analizzando i dati presenti su StackOverflow. Il progetto prevede di implementare una soluzione software adeguata, con l'utilizzo di Python e le dovute API, per raccogliere e organizzare i dati. Successivamente dovrà procedere all'elaborazioni di tali dati per renderli adeguati alla fase di analisi che consisterà nel processare e classificare i testi ottenuti tramite l'utilizzo di librerie dedicate e in fine permettere una visualizzazione grafica dei risultati ottenuti.

1- Introduzione

1.1- Da cosa nasce il progetto

Grandi piattaforme di Natural Language Processing (NLP), come Dialogflow, forniscono agli sviluppatori la possibilità di far comunicare utenti e software in modo sempre più diretto e intuitivo, evolvendo da una comunicazione tramite menu di vario genere all'interno dell'applicazione, in un semplice dialogo tra utente e assistente conversazionale; ne sono alcuni esempi Alexa (Amazon), Cortana (Microsoft) e Siri (Apple). Il settore del software engineering ha tuttavia un panorama talmente ampio e in continua crescita che rende impossibile perfino per un esperto programmatore avere uno scibile talmente ampio da non incombere mai in problematiche durante lo sviluppo di un progetto. Per questa ragione siti di Questions And Answers (Q&A), come StackOverflow, danno la possibilità agli sviluppatori di scambiare le proprie conoscenze, rendendo questi forum una raccolta di informazioni su le problematiche più discusse. Il progetto nasce quindi dalla necessità di effettuare un'analisi automatica di tali problematiche, più nello specifico quelle riguardanti Dialogflow/Apiai.

1.2- Obbiettivo

Il software dovrà essere in grado di raccogliere i feedback degli utenti, prendendo tutti i thread d'interesse prodotti su StackOverflow, in un arco di tempo di circa 2 anni e memorizzarli in un formato di facile lettura e elaborazione. Successivamente prima di passare alla fase di analisi il software dovrà essere in grado di "pulire" il testo, eliminando caratteri e parole che non sono utili ai fini della nostra analisi (vedremo successivamente più nel dettaglio cosa intendiamo con pulizia dei dati) per renderlo adatto a essere analizzato e proseguire quindi all'elaborazione del testo risultante, trovando i topic di maggior interesse e i termini che più compaiono, i quali possono essere indicatori dei temi più discussi. Per concludere il software dovrà anche mostrare tramite grafici i risultati per poterli esaminare in modo semplice e chiaro.

1.3- Fasi di sviluppo

Definiti quindi gli obbiettivi del progetto e cosa dovrà fare il software si è scelto di suddividere il progetto in quattro distinte fasi per tenere separati compiti e problematiche durante l'implementazione. Le quattro fasi sono dunque quelle riportate di seguito nella figura 1.

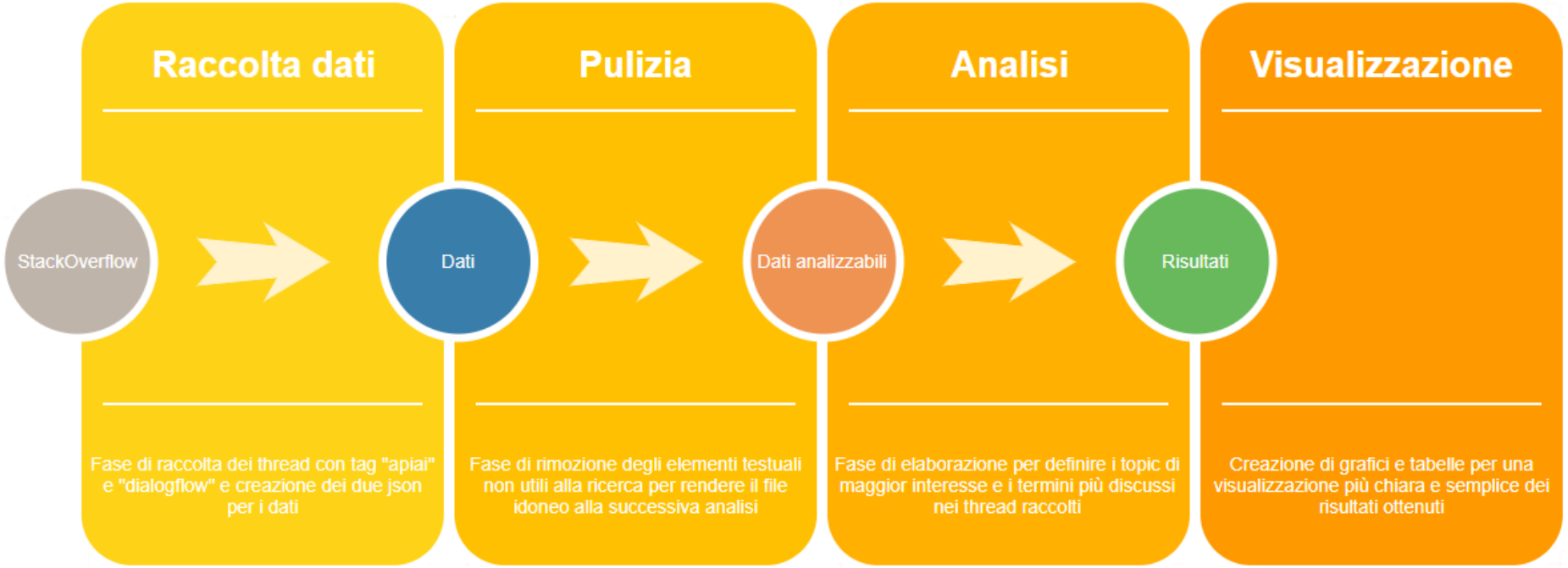

Figura 1. Schema dei passi del progetto

2- Raccolta dati

2.1- StackOverflow

Come tratto dal sito [1], StackOverflow è stato fondato nel 2008, ed è al momento il sito di Q&A con community di fiducia per lo scambio di conoscenze informatiche più grande al mondo. Può vantare infatti più di 50 milioni di utenti, tra professionisti e non, ogni mese per aiutare a risolvere problemi di ogni genere legati al mondo dell'informatica. Per queste caratteristiche è stato scelto come fonte dei dati per la ricerca svolta durante il progetto.

2.2- Fetch dei thread

Una volta scelta la fonte dei dati il problema da affrontare è stato quello di raccogliere tutti i thread in modo automatico. Si è quindi optato per l'utilizzo di API dedicate fornite da StackExchange, con cui mi è stato possibile effettuare la richiesta dei dati a StackOverflow come mostrato in figura 2.

```
from stackapi import StackAPI
import json

SITE = StackAPI('stackoverflow')
SITE.max_pages = 100
SITE.page_size = 100

questions_df = SITE.fetch('questions',
                          order='desc',
                          sort='creation',
                          tagged='dialogflow',
                          filter='!oDlrho48Yi7zQSKSpxpMM2NZTKZx-.nI2()VvYiDi)i')

questions_aa = SITE.fetch('questions',
                          order='desc',
                          sort='creation',
                          tagged='api.ai',
                          filter='!oDlrho48Yi7zQSKSpxpMM2NZTKZx-.nI2()VvYiDi)i')

print(len(questions_df['items']))
print(len(questions_aa['items']))

with open('RetrievedData/dialogflow.json', 'w+') as f1:
    json.dump(questions_df, f1)
with open('RetrievedData/apiai.json', 'w+') as f2:
    json.dump(questions_aa, f2)
```

Figura 2. Codice per la fetch dei dati da StackOverflow

Una volta definito il sito da cui prendere i dati (“stackoverflow” in questo caso) ho impostato max_pages e page_size con cui è stato possibile ottenere tutti i dati possibili altrimenti limitati alle dimensioni di default per la fetch dei dati. Definiti questi parametri per la raccolta dei dati sono passato alla fetch effettiva, ma come si può notare dal codice sono state necessarie due fetch in realtà: una per la tag “dialogflow” e una per “apiai”, questo perché come riportato sul sito ufficiale di API.AI [2], in data 10 Ottobre 2017, è avvenuto un cambio di nome per la piattaforma sopra citata e quindi è stato necessario effettuare una richiesta per entrambi i nomi così da non perdere dati fondamentali. Si può anche notare che compare come ultimo parametro un filtro per la ricerca, quest’ultimo è stato di fondamentale importanza per la raccolta dei dati.

2.2.1- Custom filter

Prima di effettuare l’effettiva fetch ho creato un custom filter, tramite il sito di StackExchange, con cui mi è stato possibile ridurre la mole di dati da richiedere a StackOverflow in modo significativo, permettendo di raccogliere solo le informazioni utili quali titolo e corpo delle domande con le relative risposte e commenti (più altre informazioni utili per lavorare coi dati come “answer_count” e “is_answered”). Ho evitato così di chiedere anche informazioni come il punteggio in badge degli utenti o le loro immagini di profilo fornite dal default filter, che non hanno rilevanza per il progetto ma che rendevano i dati raccolti più pesanti in termini di spazio fisico e più complicati da elaborare.

Il risultato del custom filter creato tramite il sito di StackExchange è quindi quello che possiamo vedere nel esempio riportato qui di seguito, in cui vediamo comparire solo i dati strettamente necessari per la ricerca come: tags, comments, answers e questions con le loro relative informazioni.


```
"tags": [
  "json",
  "android-intent",
  "drag-and-drop",
  "api.ai"
],
"comments": [
  {
    "comment_id": 77910064,
    "body":    "<code>postman</code> is the best tool I have ever seen"
  },
  {
    "comment_id": 77910436,
    "body":    "thank you, will give it a try! @sudhakar"
  }
],
"answers": [
  {
    "comment_count": 0,
    "is_accepted": false,
    "answer_id": 45303999,

    "body":
                "<p>i dont know it would help you or not but try postman
                not entirely for the situation you mentioned but removes
                a lot of pain from your head <a
                href=\"https://www.getpostman.com/docs/postman/collection
                s/creating_collections\" rel=\"nofollow
                noreferrer\">https://www.getpostman.com/docs/postman/coll
                ections/creating_collections</a></p>\n"
                  }
],
"comment_count": 2,
"is_answered": false,
"answer_count": 1,
"creation_date": 1500986979,
"question_id": 45303858,

"title":
                "Is there any way to create a tool with drag and drop
                mechanism to create a JSON file of intents in api.ai?",

"body":
                "<p>I am looking for a tool or a way through which I can
                create a drag and drop mechanism containing intents
                messages for user inputs and the respective response to
                it. It should then be able to create a JSON file of those
                multiple intents so that I can upload it to api.ai, and
                my intents are then visible there.</p>\n"
```


Figura 3. Esempio di thread memorizzato in formato json

2.3- Memorizzazione

Le due fetch di cui abbiamo parlato nel capitolo 2.2 hanno quindi generato due file json, uno per i thread relativi alla tag “dialogflow” e uno per “apiai”. Lavorare su due file separati però non risultava una scelta pratica, ho quindi scelto di scrivere una parte di codice che unisse i due file in un unico json completo e senza ripetizioni di dati (problema dovuto da thread che presentavano entrambe le tag). Inoltre ho scritto anche dei metodi per vedere alcune informazioni sui dati raccolti che riassumo nella tabella di figura 4.

Quantificazione dati di StackOverflow	
Data thread più vecchio	Domenica 1 ° gennaio 2017
Data thread più recente	Mercoledì 17 ottobre 2018
Domande	1736
Risposte	1637
Commenti a risposte	2001
Commenti a domande	2075
Media di commenti per risposta	1.2
Media di commenti per domanda	1.2
Domande marcate come chiuse	887
Percentuale di domande chiuse	51.09%

Figura 4. Tabella delle statistiche sui dati raccolti

3- Pulizia dei dati

A questo punto tutti i dati necessari sono in un unico file json completo e pronto per la fase successiva. Per rendere i dati analizzabili ho dovuto eliminare prima tutto ciò che non ci dà informazioni utili come: tag HTML, punteggiatura e English stop words. Vediamo ora più nel dettaglio questi casi esemplificandoli così da rendere più chiaro il perché è necessario rimuoverli.

3.1- Tag HTML e punteggiatura

Effettuando la richiesta dei dati a StackOverflow tramite le API ci viene restituito il testo dei vari thread ma nella forma che possiamo vedere nel seguente esempio:

* Now that localization is available for Actions on Google, I'm wondering about minimizing latency for user responses. In a stack where an app is using Dialogflow for NLP and then a Google Cloud Function behind that for webhook fulfillment, is it possible to set up cloud functions where a specific localized experience has been established in Actions on Google? In other words, can we set up English-US responses to be provided by a cloud function hosted in us-central1, and English-GB via europe-west1 Google datacenters? \n\n With Alexa & ASK you can set up a language with 'geographical region endpoints' in N America, Europe and India and have Lambda functions fulfill intents from users in those regions. \n\n Dialogflow may not be available in multiple GCP regions, but if it is it would be nice to keep fulfillment local with google cloud functions close to where the user is - if it minimizes conversation delays. Maybe someone has data to suggest it doesn't matter and is fast enough either way. \n"*

Come si può notare a prima vista il testo appare caotico e disordinato e una delle cause principali sono appunto i tag HTML come …. (prima riga del nostro esempio), i quali occupano gran parte del corpo del testo e non forniscono informazioni utili. Quello che vogliamo invece come testo ideale è quello che riporto qui di seguito.

action wonder minimize latency user response stack app nlp cloud function webhook fulfillment set cloud function specific localize experience establish action set english response provide cloud function host central english gb europe west datacenter alexa amp set language geographical region endpoint america europe india lambda function fulfill intent user region multiple gcp region nice keep fulfillment local cloud function close user minimize conversation delay data matter fast

Questo è il risultato della fase di pulizia sul testo precedentemente mostrato.

Per arrivare però al risultato ottimale di pulizia è stato fondamentale la scelta dell'ordine con cui eliminare i vari elementi, perché un ordine sbagliato avrebbe compromesso la eliminazione di altre parti, ad esempio togliendo prima la punteggiatura rispetto ai tag HTML avrebbe creato problemi in quanto sarebbero state eliminate le parentesi angolari su cui invece viene fatto il controllo per capire dove hanno inizio e fine i tag. Inoltre non tutti i tag sono stati gestiti allo stesso modo come ad esempio i tag <code> e dove il primo contiene codice effettivo che un utente mette per mostrare il problema o una possibile soluzione, ma ciò non dà informazioni per la nostra analisi. Il secondo invece serve solo a visualizzare in formato grassetto il proprio contenuto che invece è di nostro interesse, in quanto può contenere termini fondamentali per stabilire il problema del topic in cui è contenuto. Vedremo più nel dettaglio come è stata gestita la cancellazione delle varie parti nel capitolo 3.3 in cui parleremo dell'implementazione scelta per risolvere questo problema.

3.2- English stop words

Come riportato su [3] con English stop words intendiamo tutte quelle parole comunemente usate come "the", "ah", "but" ecc... che come avviene nei motori di ricerca vengono rimosse in quanto non forniscono alcun significato semantico all'interno della frase. Dopo aver eliminato tags e punteggiatura ci concentriamo in fine sull'eliminazione di queste English stop words elencate da Rank NL [4] a cui ho aggiunto, durante vari test sul progetto, alcune parole ricorrenti non presenti originariamente nella lista.

3.3- Espressioni regolari

Per ora ho sempre parlato di cosa eliminare ma mai di come è stata effettuata la pulizia del testo. Tutta l'eliminazione è stata gestita tramite la libreria *"re"* [5] che ha permesso l'uso appunto delle Regular Expressions. Grazie quindi ha re.py ho potuto così eliminare grandi porzioni di testo in modo semplice e veloce, come si può vedere negli esempi seguenti, dove per ogni parte di testo da eliminare è stata creata una lista di parametri di eliminazione, ogni volta il metodo di pulizia viene chiamato controlla per ogni parametro in queste liste se è presente nel testo l'elemento da cancellare e procede in caso affermativo.

```
tags_expressions =
['<code>.+?</code>', '<pre>.+?</pre>', '<a href.+?</a>', '</p>.+?<p>']


for param in tags_expressions:
    text = re.sub(param, ' ', text)
```

Questo estratto di codice mostra l'eliminazione dei tag HTML con contenuto non significativo, infatti `'<code>.+?</code>'` può essere tradotto in linguaggio naturale come "seleziona tutto ciò che si trova tra la stringa `<code>` e `</code>` più questi ultimi compresi" permettendo così di eliminare grandi parti di testo che sappiamo non servire senza saperne il contenuto effettivo. Per quanto riguarda i tag di cui vogliamo invece mantenere il contenuto è bastato usare invece la seguente lista di parametri.

```
tags_words =
['<br>',   '<p>',   '</p>',   '<strong>',   '</strong>',   '<blockquote>',
'</blockquote>', '<li>', '</li>', '<ol>', '</ol>', '<em>', '</em>']


for param in tags_words:
    text = re.sub(param, ' ', text)
```

La stessa procedura di questi ultimi è stata utilizzata per numeri e English stop words. Per la punteggiatura invece la libreria re.py forniva già una soluzione standard che è la seguente:

```
text = re.sub('[^\w\s\\-]', ' ', text)
```

3.4- Metodo di eliminazione

Ora dopo aver mostrato quali elementi è stato necessario rimuovere e come sono stati eliminati spiegherò l'ordine con cui è stato fatto ciò, perché come abbiamo visto nel capitolo 3.1 l'ordine con cui avviene l'eliminazione di alcuni elementi può condizionarne altri. Dopo vari test la procedura scelta è quindi la seguente:

1) Inserimento di spazi vuoti a inizio e fine testo per permettere i controlli anche sulle parole iniziali e finali.

2) Rimozione di tutti i *"carriage return"* cioè dei *"ritorno a capo"*.

3) Conversione di ogni carattere in minuscolo per uniformare il testo e evitare problemi in quanto si sta lavorando in modo case sensitive e quindi avere caratteri sia minuscoli che maiuscoli può causare errori.

4) Rimozione dei tag HTML con contenuto eliminabile.

5) Rimozione dei tag HTML con contenuto da mantenere.

6) Rimozione di caratteri speciali HTML come ad esempio "`&#39;`" (apostrofo), senza mettere spazi al loro posto così da mantenere unite parole come "don't" che diventerà "dont" per i controlli successivi

7) Eliminazione dei numeri

8) Rimozione della punteggiatura

9) Sostituzione di tutti gli eventuali spazzi consecutivi presenti o generati dalle fasi precedenti con uno spazio singolo

10) Rimozione delle English stop words

11) Rimozione degli spazi vuoti a inizio e fine testo inseriti nel passo 1)

3.4- Lemmatizzazione

A questo punto il file json risultante contiene solo parole d'interesse per la nostra ricerca, tuttavia queste parole sono scritte in varie forme diverse mentre invece è necessario che siano tutte uniformate alla loro forma base per permettere un'analisi statistica del testo. Inizialmente si era scelto di eseguire un'operazione di Stemming per normalizzare il testo, come mostrato dal sito *pythonprogramming* nella sezione Natural Language ToolKit (NLTK) [6], ma questa operazione generava parole inesistenti in quanto tende a tagliare la parte finale delle parole per uniformarle. La soluzione finale che ho adottato è stata quindi quella della Lemmatizzazione sempre tramite NLTK [7], che come è possibile vedere nella figura 5, a differenza del processo di Stemming genera solo parole esistenti convertendo parola per parola nel suo lemma.

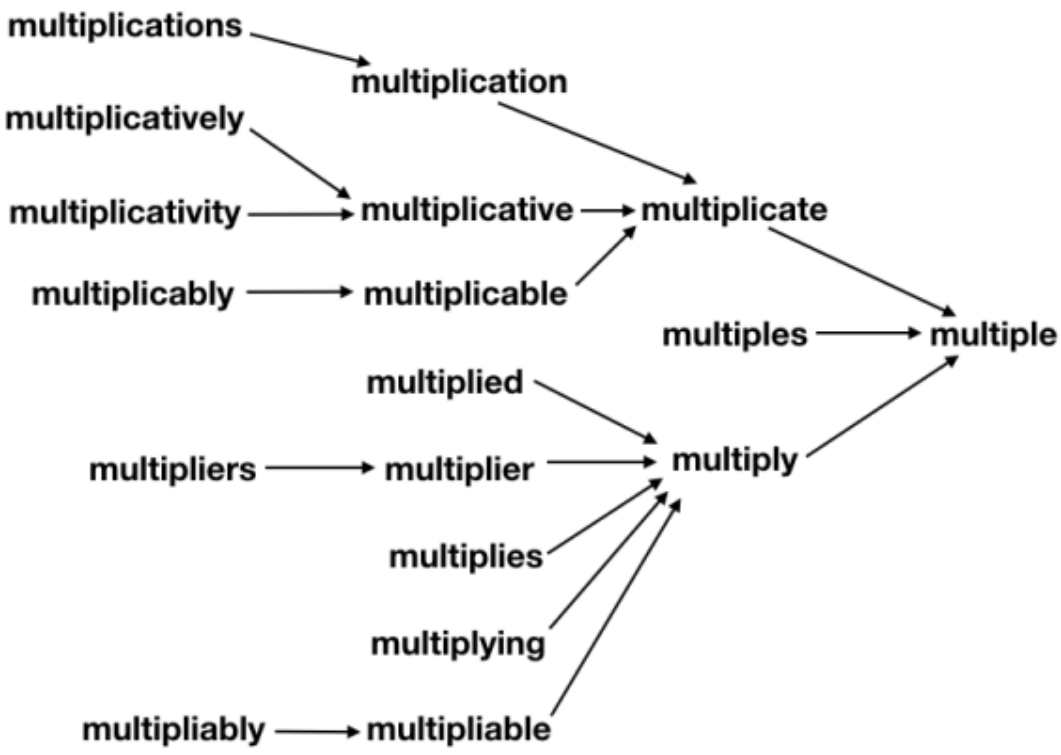

Figura 5. Esempio del processo di lemmatizzazione [8]

Per implementare la Lemmatizzazione ho scritto i metodi *map_words* e *get_wordnet_pos* riportati in figura 6:

```
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return None


def map_words(text):
    lemmatizer = WordNetLemmatizer()
    lemmatized_words = []
    tokens = []

    for word in text.split():
        tokens.append(nltk.word_tokenize(word))

    for t in tokens:
        tagged = nltk.pos_tag(t)
        for word, tag in tagged:
            word_net_tag = get_wordnet_pos(tag)

            if word_net_tag is None:
                lw = lemmatizer.lemmatize(word)
            else:
                lw = lemmatizer.lemmatize(word, pos=word_net_tag)

            lemmatized_words.append(lw)

    mapped = ' '.join(str(e) for e in lemmatized_words)
    mapped = re.sub(r'[^\w\s]', ' ', mapped

    return mapped.lstrip().rstrip()
```

Figura 6. Estratto di codice per la procedura di lemmatizzazione

L'implementazione è stata effettuata, come si può vedere dalla parte di codice appena mostrata, leggendo riga per riga e parola per parola convertendo queste ultime in token per renderle elaborabili da nltk.py. Questo mi ha permesso di ottenere automaticamente il tipo di parola (avverbio, sostantivo, verbo, ecc…) da fornire al metodo di lemmatizzazione che effettua la conversione in lemma della parola. Una volta effettuata questa operazione

per ogni parola del testo, il risultato fornito dalla libreria nltk è una lista di parole convertite in lemma nella forma seguente:

['test', 'error', 'cause', 'format', 'microphone', 'stream', 'data', 'example', 'refer', 'wav', 'file', 'stream', 'format', 'input', 'audio']

A questo punto è bastato riconvertire la lista di parole in forma testuale per inserirle nel nuovo file json che verrà usato per la successiva fase di analisi. Nella figura 7 riporto un esempio di thread per mostrare il risultato ottenuto: il thread in questione è lo stesso mostrato nel capitolo 2.2.1 con cui è possibile vedere la differenza tra prima e dopo l'intero processo di pulizia dei dati.


```
"tags": [
  "json",
  "android-intent",
  "drag-and-drop",
  "api.ai"
],
"comments": [
  {
    "comment_id": 77910064,
    "body": "best tool"
  },
  {
    "comment_id": 77910436,
    "body": "sudhakar"
  }
],
"answers": [
  {
    "comment_count": 0,
    "is_accepted": false,
    "answer_id": 45303999,
    "body": "help postman entirely situation mention remove lot pain
head"
  }
],
"comment_count": 2,
"is_answered": false,
"answer_count": 1,
"creation_date": 1500986979,
"question_id": 45303858,
"title": "create tool drag drop mechanism create son file intent",
"body": "tool create drag drop mechanism intent message user input
respective response create son file multiple intent upload intent
visible"
```


Figura 7. Esempio di thread memorizzato dopo la fase di pulizia

4- Analisi

A questo punto i dati sono pronti per essere analizzati e ho iniziato la fase di ricerca dei topic più discussi e i termini più utilizzati che possono quindi essere indicatori di problematiche. Per fare ciò ho utilizzato la tecnica di modellazione dei topic chiamata *Latent Dirichlet Allocation* (LDA) implementandola con l'uso di gensim.

4.1- LDA

Come tratto da *Journal of Machine Learning Research 3 (2003) 993-1022* [9], *Latent Dirichlet Allocation* è un modello probabilistico generativo di un corpus. L'idea di base è che i documenti sono rappresentati come miscele casuali su argomenti latenti, in cui ogni argomento è caratterizzato da una distribuzione sulle parole. La figura 8 tratta da energies-10-01913-v2 [9] ne mostra una rappresentazione grafica.

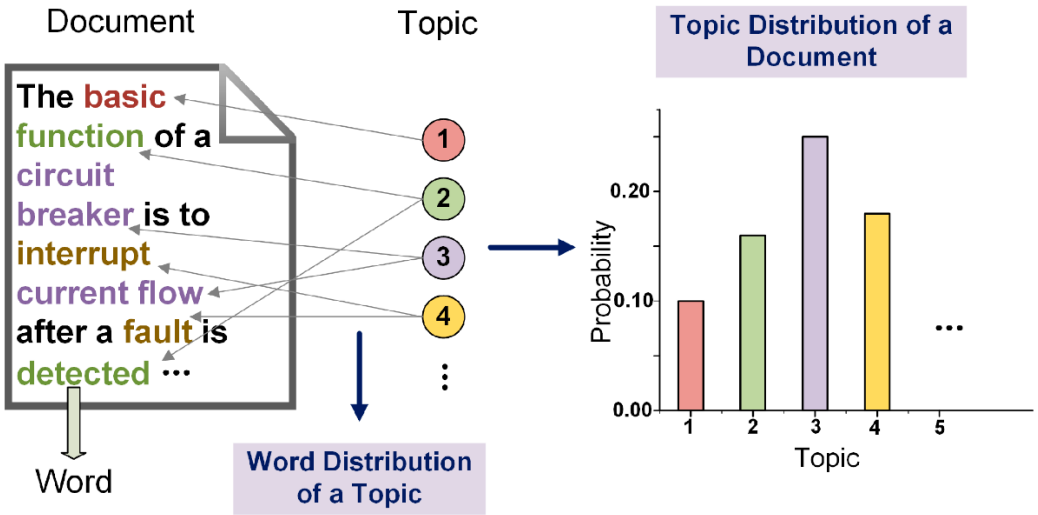

Figura 8. Esempio illustrativo di LDA

4.2- Gensim

Come detto a inizio capitolo per implementare il mio modello LDA sui dati raccolti ho utilizzato gensim [11], nato nel 2008 è iniziato come una raccolta di vari script Python per la libreria di matematica digitale ceca dml.cz nel 2008, in cui è servito a generare una breve lista degli articoli più simili a un determinato articolo (gensim = "generare simili"). ad oggi software per realizzare la modellazione semantica. Importato gensim nel progetto ho potuto quindi trovare i 3 topic maggiormente discussi tramite il codice in figura 9.

```
NUM_TOPICS = 3
lda_model = gensim.models.ldamodel.LdaModel(corpus, num_topics=NUM_TOPICS, id2word=dictionary, passes=15)
lda_model.save('DataAnalysis/model5.gensim')
topics = lda_model.print_topics(num_words=4)
for topic in topics:
    print(topic)
```

Figura 9. Esempio illustrativo di LDA

La ricerca è stata impostata a 3 dopo vari test per la miglior visualizzazione dei dati a cui ho poi dato una forma grafica per poterli studiare più comodamente come mostrerò nel capitolo successivo.

5- Risultati

Generati i risultati dell'analisi effettuata tramite il modello LDA di gensim ho quindi fatto uso della libreria pyLDAvis [12] per una visualizzazione interattiva del modello ottenuto.

```
lda_display = pyLDAvis.gensim.prepare(lda, corpus, dictionary, sort_topics=False)
pyLDAvis.save_html(lda_display, "results.html")
```

Con questa parte di codice ho quindi generato il file HTML con i seguenti risultati.

5.1- Topic più discussi

In questo capitolo riporto le immagini relative al contenuto del file HTML da cui è possibile vedere come l'analisi abbia trovato 3 distinti topic e quali termini sono più usati per ognuno di essi. La figura 10 rappresenta la distribuzione generale di tutti i termini più utilizzati nei 1736 thread raccolti che sono: intent, user, response, fino a parameter e help. Le figure 11, 12, e 13 invece mostrano nel dettaglio come questi termini siano distribuiti nei rispettivi topic 1, 2 e 3.

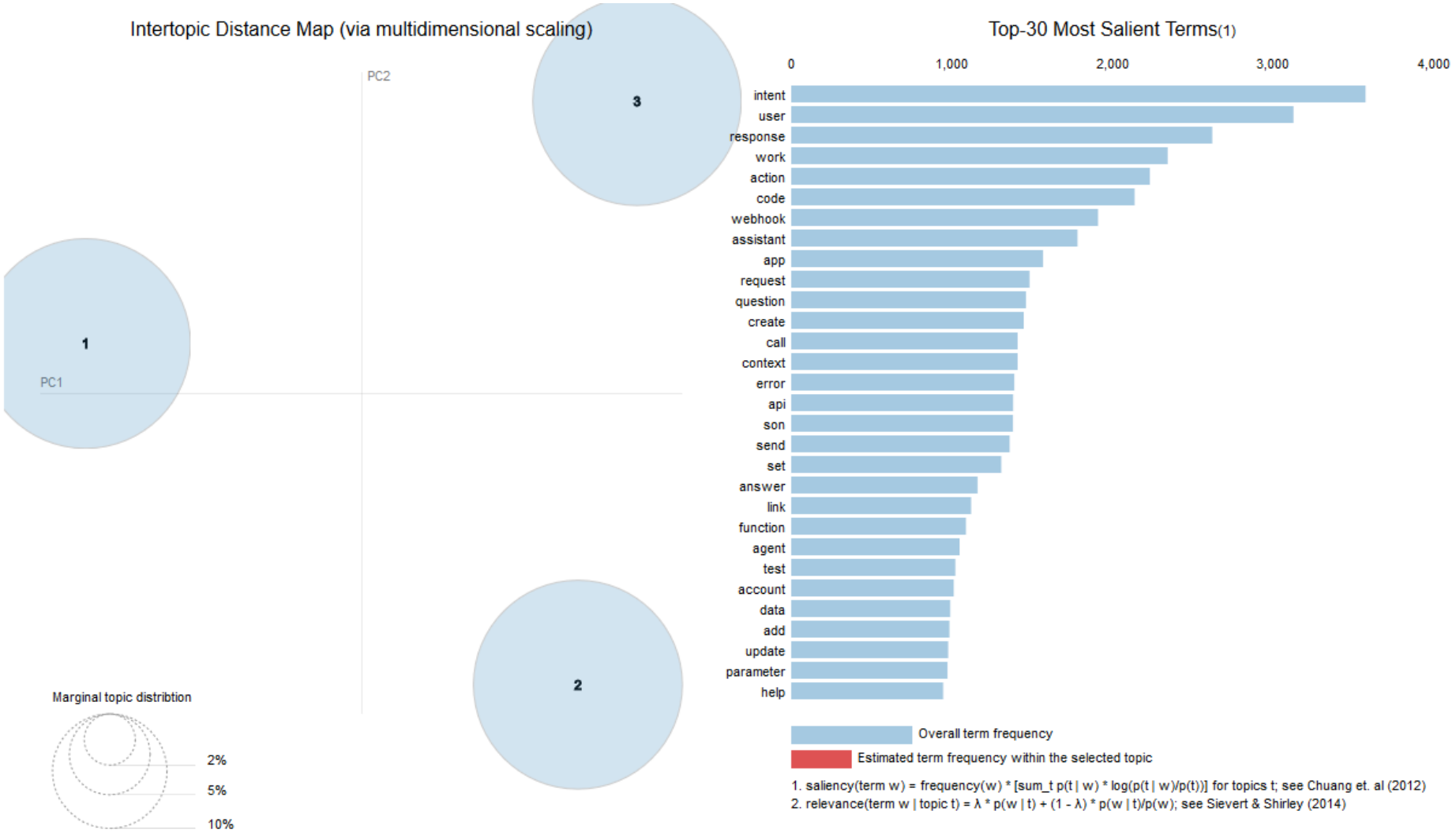

Figura 10. Distribuzione totale delle parole presenti nei 1736 thread raccolti

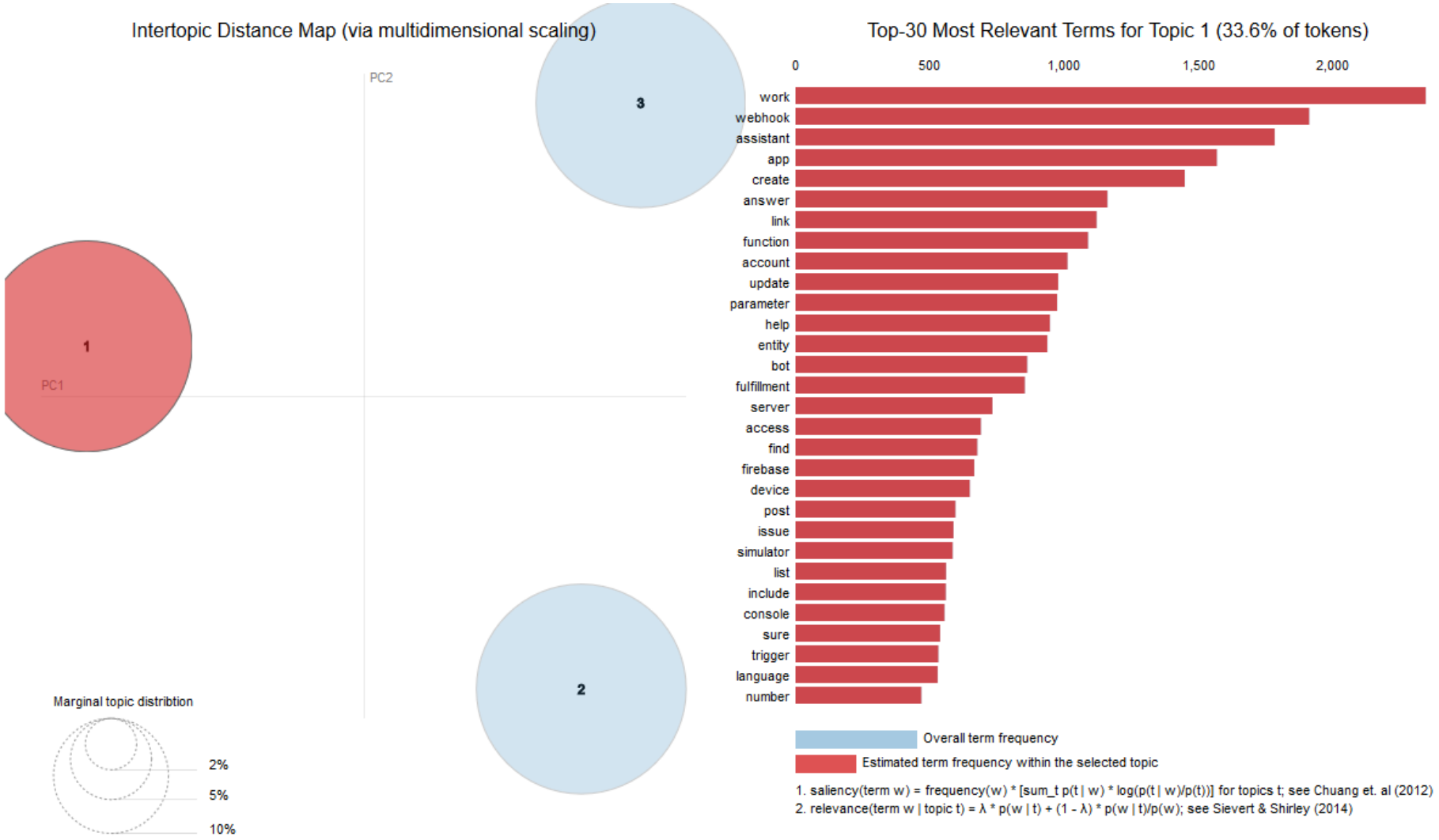

Figura 11. Distribuzione delle parole relativa al topic 1 rispetto al totale

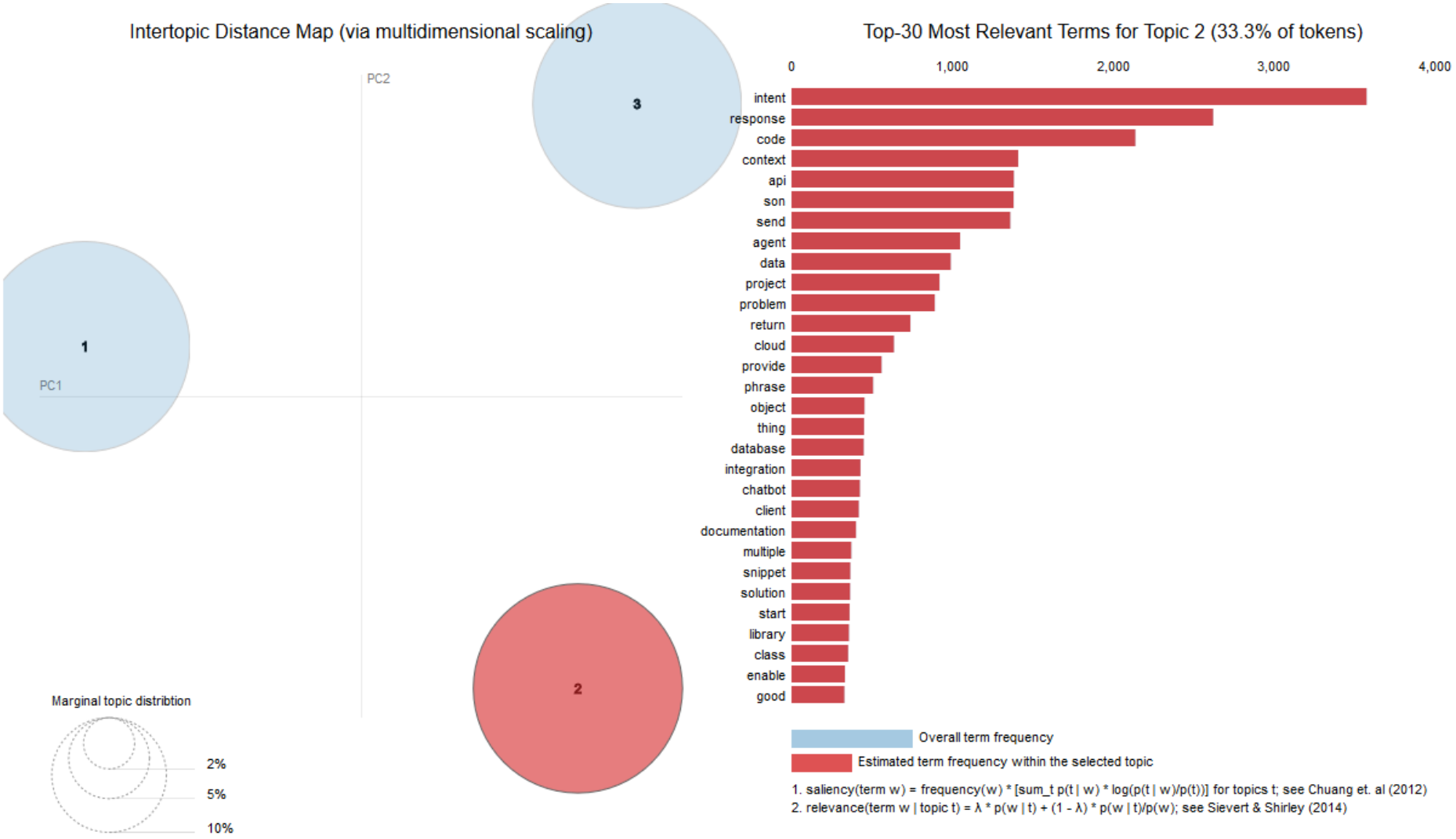

Figura 12. Distribuzione delle parole relativa al topic 2 rispetto al totale

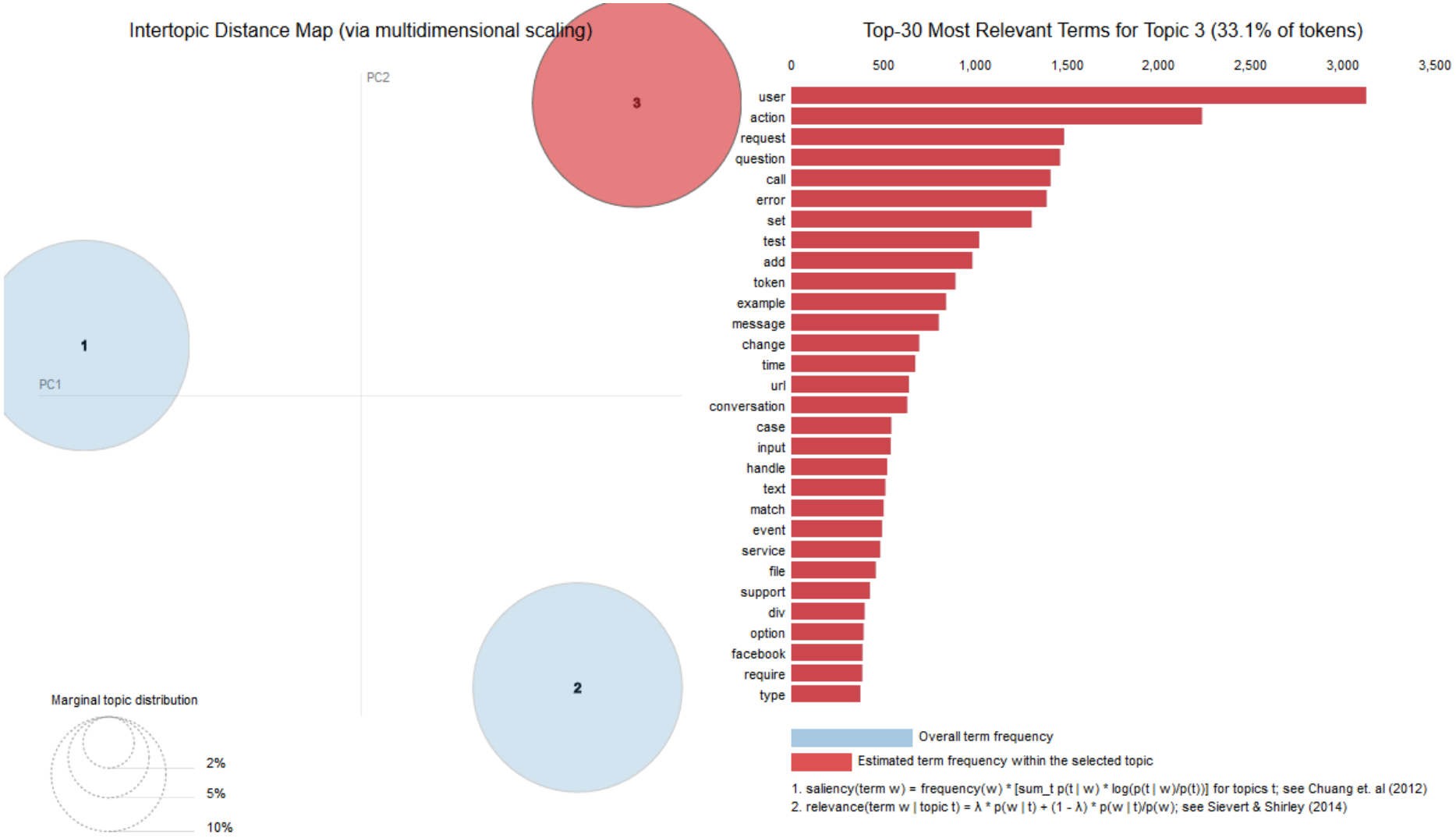

Figura 13. Distribuzione delle parole relativa al topic 3 rispetto al totale

Adesso che abbiamo visto i risultati della ricerca effettuata sappiamo quali sono i termini che più compaiono all'interno dei dati raccolti e essendo divisi in questi 3 topic possiamo vedere come alcuni siano più presenti insieme ad altri. Come nella figura 5 con *"intent"* e *"response"* o nella figura 6 con *"user"* e *"action"*. Andando però manualmente a controllare nel file json e leggendo i thread ho appurato che anche se sono più utilizzati non è sempre indicatore di una problematica relativa a quel termine. Il termine *"response"* ad esempio compare molto ma quasi sempre in frasi come *"thanks for the response"* o per mostrare i risultati di parti di codice scrivendole come *"response: risultati…"*. Anche per *"work"* vale lo stesso discorso dato che viene utilizzato spesso per dire *"i'm trying to start working on…."* Oppure *"it works for me"*. Alla luce di ciò ho preso 10 thread campione in modo casuale per ognuna delle prime due parole di un topic, verificando in quanti casi le problematiche esposte del thread sono effettivamente relative ad esse. Nel grafico che segue in figura 14 ho riportato in verde la percentuale di domande relative alla parola in questione e in rosso la percentuale di quante domande invece contengono il termine ma riguardano un problema di tutt'altra natura.

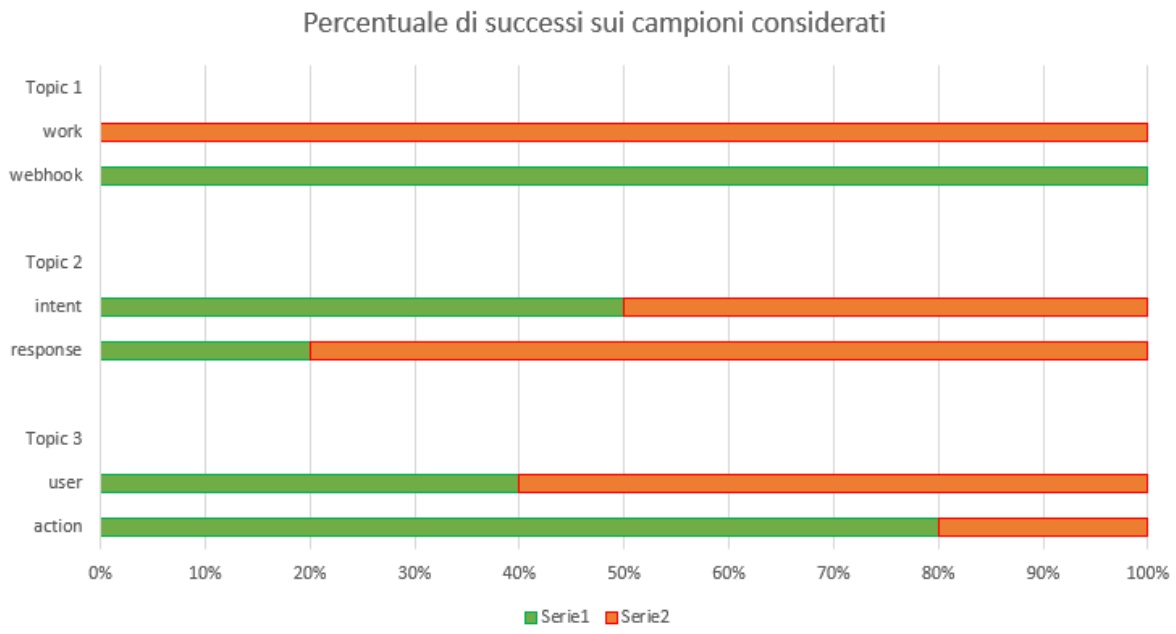

Figura 14. Grafico dei risultati dell'analisi manuale su thread campionati casualmente

5.2- Storico dei thread

Un'altra analisi che ho effettuato sui dati è stata quella di vedere con quale frequenza venivano poste domande su problematiche legate a Dialogflow e APIAI su StackOverflow nell'arco di tempo i cui i dati sono stati raccolti, cioè tra il 1 ° gennaio 2017 e il 17 ottobre 2018. Per fare questo ho analizzato il json e generato un file csv coi dati necessari per questa analisi quali: numero di domande, risposte e commenti per ogni mese nell'arco di questi 2 anni. Una volta creato il file .csv è stato possibile importarlo in un foglio di lavoro di Excel per visualizzarne il grafico risultante mostrato in figura 15 in cui è chiaro notare come nel mese di Ottobre 2017 il numero delle domande relative a Dialogflow/APIAI sia aumentato notevolmente e abbia continuato a crescere nei mesi successivi fino ad un calo di frequenza nell'ultimo periodo. La causa di questo rapido incremento è molto probabilmente associabile al fatto che come abbiamo detto capitolo 2.2, in data 10 Ottobre 2017 è avvenuto il cambio di nome per la piattaforma ma soprattutto sono state introdotte due nuove funzionalità quali [2]:

1. In-line code editor: possibilità di scrivere la logica di realizzazione, testare e implementare un webhook funzionale direttamente nella console.

2. Multi-lingual agent support: possibilità aggiungere ad un nuovo o già esistente *"agent"* lingue e locales che coprono la regione o le versioni specifiche per paese di una lingua di root. Ad esempio, "francese" è una lingua di root che include la Francia e le impostazioni locali in Canada francese.

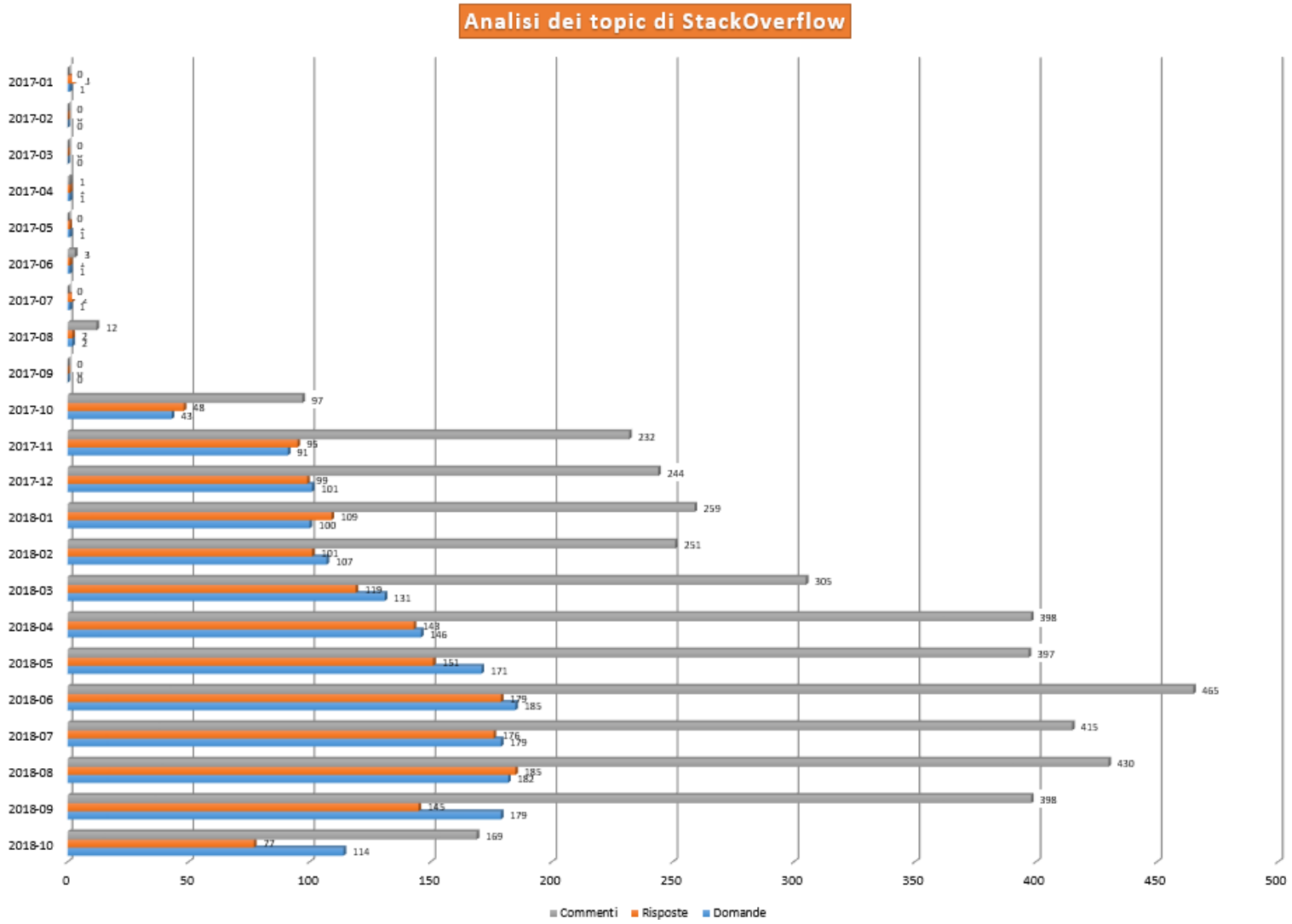

Figura 15. Analisi del posting nel tempo

6- Conclusioni e lavori futuri

Durante questo progetto di ricerca ho potuto utilizzare e apprendere Python come linguaggio di programmazione, oltre a migliorare le conoscenze acquisite durante tutto il corso della triennale. Ho fatto uso anche di molte API e librerie esterne che ho dovuto far interagire e coordinare tra di loro come visto nei capitoli precedenti. Come IDE ho utilizzato pyCharm [13] fornito da jet brains e fondamentale è stato anche l'uso di GitHub [14] per la gestione di backup e versioning del progetto durante tutto l'arco dell'implementazione.

Giunto al termine di questo progetto posso aggiungere che come visto nel capitolo 5.1 ho effettuato una verifica dei dati leggendo i primi 10 thread in cui compaiono le parole più discusse di ogni topic per verificare con certezza se le problematiche erano effettivamente legate a questi termini. Uno sviluppo futuro potrebbe quindi essere quello di automatizzare questa procedura tramite un'analisi, delle sole domande, basata sui dati e i risultati raccolti dal progetto in modo da estendere questa verifica su l'intero file di dati. Altrimenti questa operazione sarebbe impraticabile da svolgere manualmente vista la mole d'informazioni.

7- Ringraziamenti

Desidero ringraziare il mio relatore di tesi, il prof. Luigi De Russis in primis per avermi dato la possibilità di svolgere questo progetto di ricerca e per tutto l'aiuto da lui fornitomi durante il corso dello stage. Un ringraziamento alla mia famiglia che mi ha permesso di arrivare fino a qua in particolare a mio fratello per tutto il sostegno che mi ha sempre dato.

Un ringraziamento a tutta l'università per avermi regalato anni davvero belli e importanti che ricorderò sempre.

Un enorme grazie ai migliori compagni di corso che avrei mai potuto desiderare: Stefano, Cristiano, Andrea, Alice, Paolo e Roberto, ecc…

Un grazie a i miei compagni di viaggio: Francesco, Gabriele, Giada, Antonio, Giulia e Annalisa, ecc….

Un grazie anche a Giada, Lorenza e Lucrezia.

E in fine un ringraziamento speciale a Eric, Matteo e Lorenzo su cui ho sempre potuto contare.

8- Bibliografia

[1] StackOverflow website: https://stackoverflow.com/company

[2] API.AI website: https://blog.dialogflow.com/post/apiai-new-name-dialogflow-new-features/

[3] definizione di stop words: https://searchmicroservices.techtarget.com/definition/stop-word

[4] Rank NL website: https://www.ranks.nl/about

[5] libreria re: https://docs.python.org/3/library/re.html

[6] NLTK – Stemming: https://pythonprogramming.net/stemming-nltk-tutorial/

[7] NLTK – Lemmatizing: https://pythonprogramming.net/lemmatizing-nltk-tutorial/

[8] figura lemmatizzazione: https://searchingforbole.com/2018/01/10/lemmatizer/

[9] Definizione di LDA: http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf

[10] energies-10-01913-v2: https://www.mdpi.com/1996-1073/10/11/1913

[11] genism: https://radimrehurek.com/gensim/about.html

[12] pyLDAvis: https://github.com/bmabey/pyLDAvis

[13] PyCharm website: https://www.jetbrains.com/pycharm/

[14] GitHub website: https://github.com/