Information Retrieval
Search Engine Project

Bedini Alessandro - Palmieri Alessandro - Incerti Marco

A.S. 2019/2020

Obiettivo

Sviluppo di un search engine basato sulle pagine di Wikipedia che restituisca i documenti in ordine di rilevanza

Ciclo di sviluppo

1 **Sviluppo di un search engine di base**
Indicizzazione di una parte del dump di wikipedia tramite whoosh

2 **Valutazione dei risultati**
Creazione di un test set attraverso Google e confronto con search engine di base attraverso MAP e NDGC

3 **Ottimizzazione**
Implementazione di preprocessing, query expansion e algoritmo di scoring

WIKIPEDIA
L'enciclopedia libera

Strumenti

- Python 3 con utilizzo di librerie Whoosh, NLTK, Pyenchant e Tkinter
- Dump XML ridotto da ~16000 documenti
- Test set da 30 query con 900 documenti di "en.wikipedia.org" estratti tramite Google

Index

```
self.schema = Schema(id=ID(stored=True),
                     url=ID(stored=True),
                     title=TEXT(stored=True, analyzer=CustomAnalyzer(), phrase=True),
                     content=TEXT(stored=True, analyzer=CustomAnalyzer(), phrase=True, spelling=True))
```

Utilizzando la prima versione dell'indice, che consiste in:

- una tokenizzazione del testo sugli spazi;
- ricerca solo sul contenuto;
- nessun valore aggiuntivo alla prossimità delle parole;
- nessuna espansione delle query;

MAP: 0.356
NDGC: 0.297

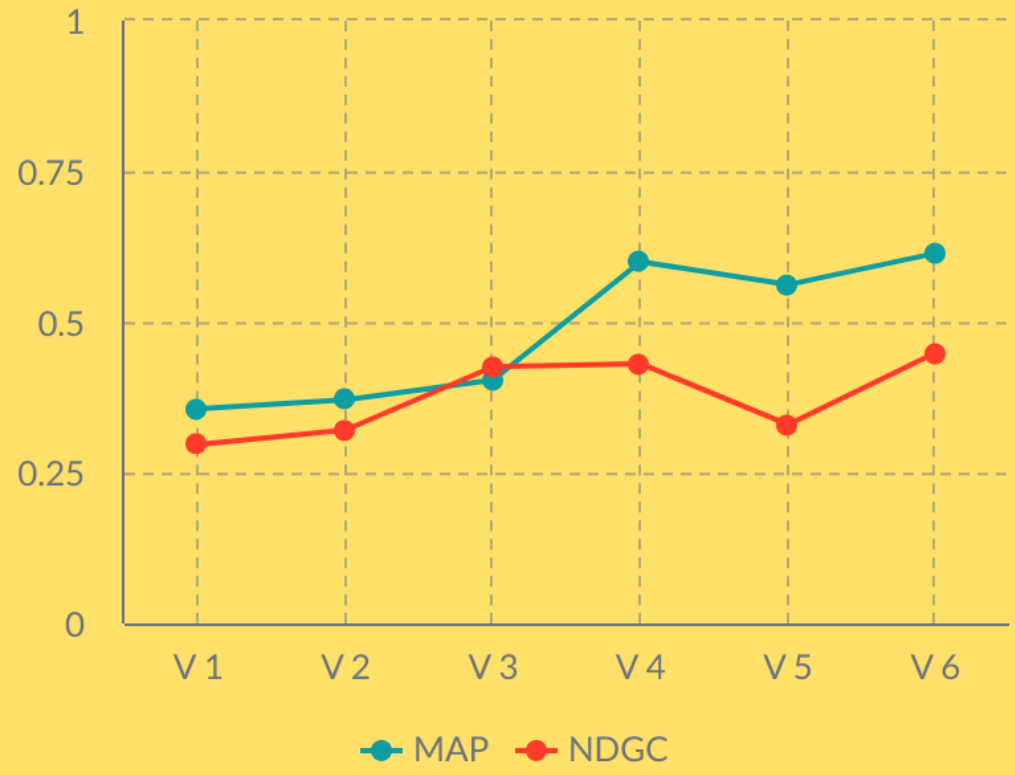
Utilizzando l'ultima versione dell'indice, che consiste in:

- filtro personalizzato per il preprocessing;
- ricerca su titolo e contenuto;
- valore aggiuntivo alla prossimità delle parole;
- espansione delle query utilizzando le keyword;

MAP: 0.740
NDGC: 0.622

Come abbiamo fatto?

Preprocessing



Filtri

- V1: space separeted tokenizer
- V2: regex tokenizer
- V3: regex | lowercase | stop filter
- V4: stemming
- V5: regex | lemmatizer
- V6: stemming | charset | custom normalizer

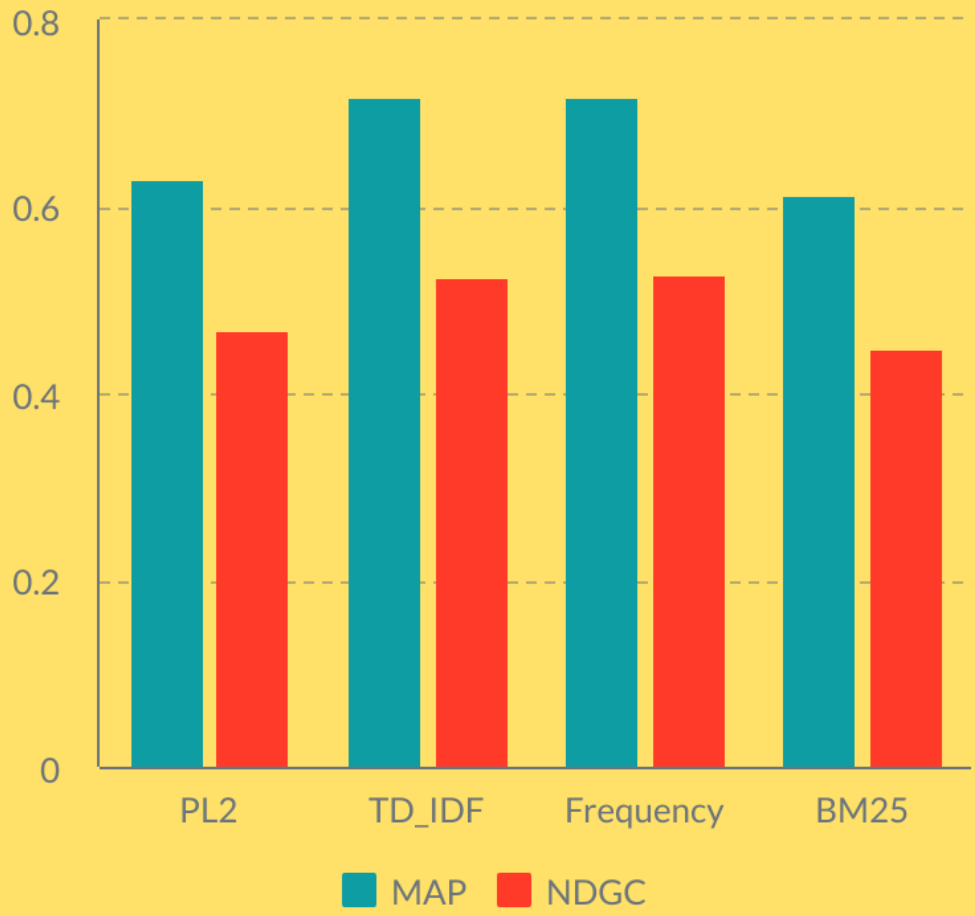
```
def CustomAnalyzer(stopword_list=Custom_StopWord):
    stemming_charset = StemmingAnalyzer() \
                       | CharsetFilter(accent_map)
    myanalizer = stemming_charset | CustomNormalizer()
    print(myanalizer)
    return myanalizer
```

```
class CustomLemmatizer(Filter):
    def __call__(self, tokens):
        for token in tokens:
            token.text = WordNetLemmatizer().lemmatize(token.text)
            yield token
```

```
class CustomNormalizer(Filter):
    def __call__(self, tokens):
        filtri = SubstitutionFilter("-", " ") | SubstitutionFilter("_", " ")
        return filtri(tokens)
```

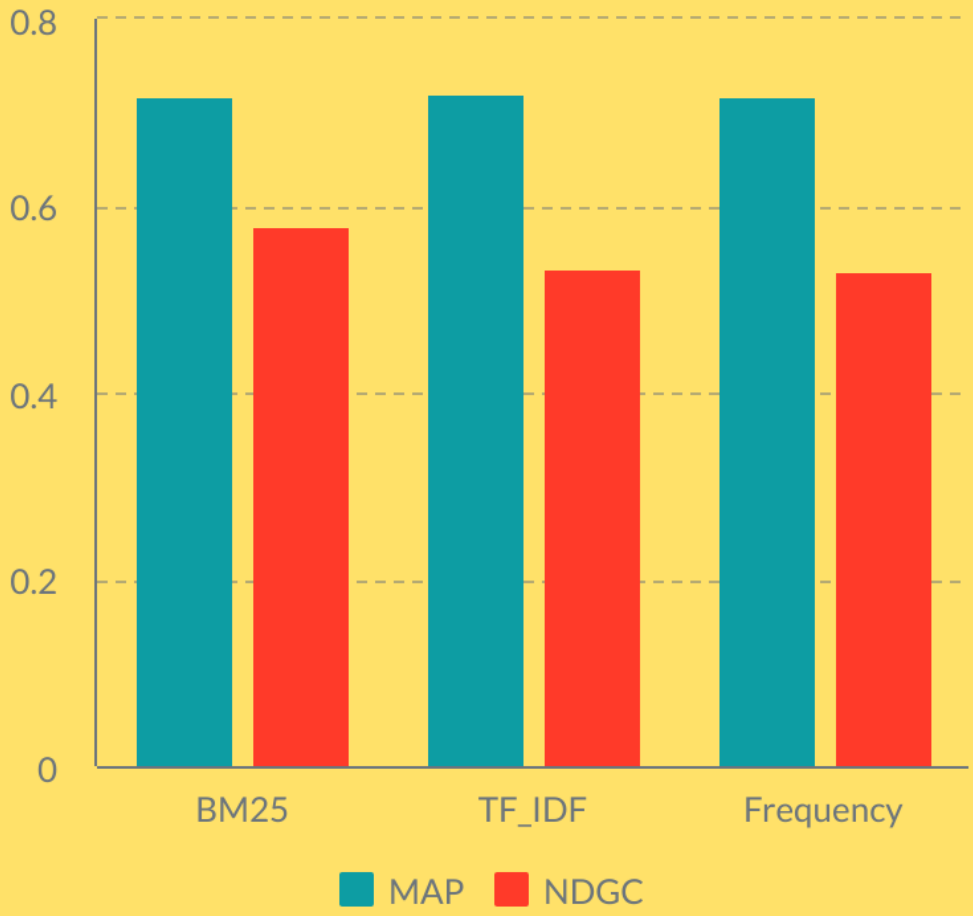
Algoritmo di ricerca

Dal grafico si deduce che gli algoritmi con migliori risultati sono **TF_IDF** e **Frequency**, almeno per quanto riguarda la ricerca solo su contenuto



Questi risultati sono stati ottenuti utilizzando l'index V6

Ricerca MultiField



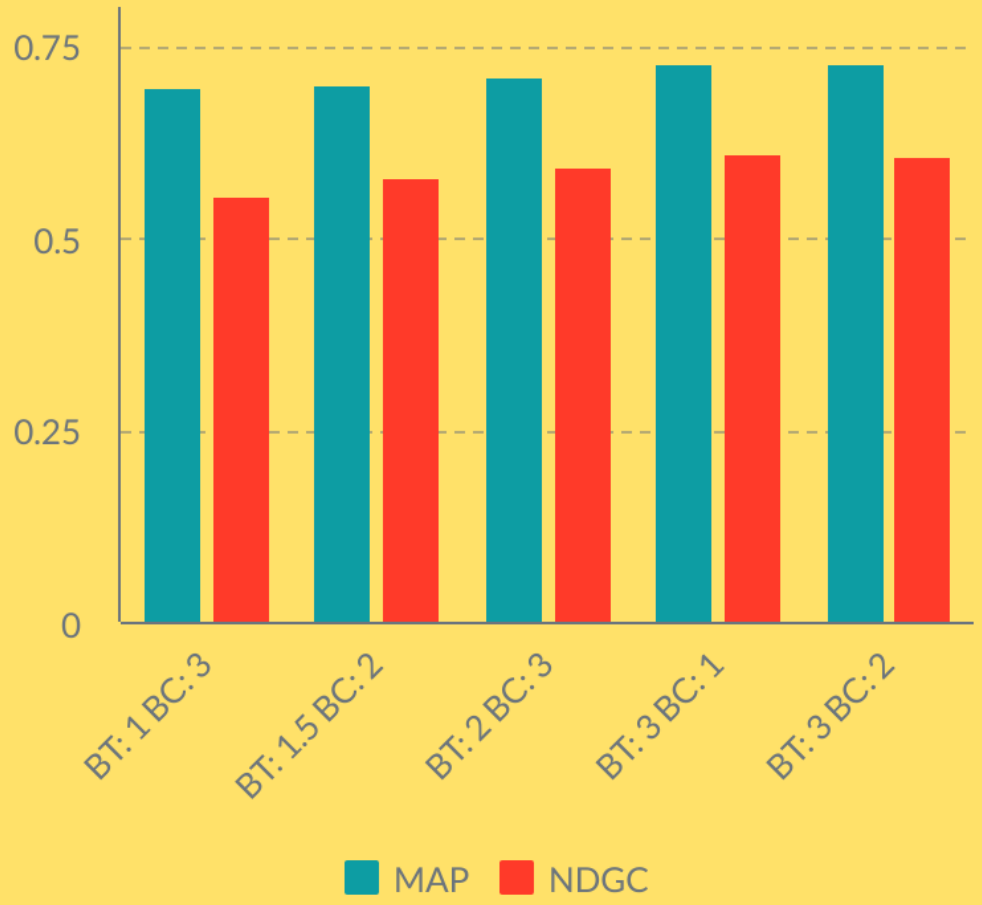
Questi risultati sono stati ottenuti utilizzando l'index V6

Implementando la ricerca su più field, in particolare su titolo e contenuto, abbiamo stabilito che l'algoritmo migliore risulta essere il BM25, ottenendo un aumento del 5% sul NDGC a discapito di una perdita dello 0.3% sulla MAP rispetto a TF_IDF

L'algoritmo scelto è di conseguenza il BM25

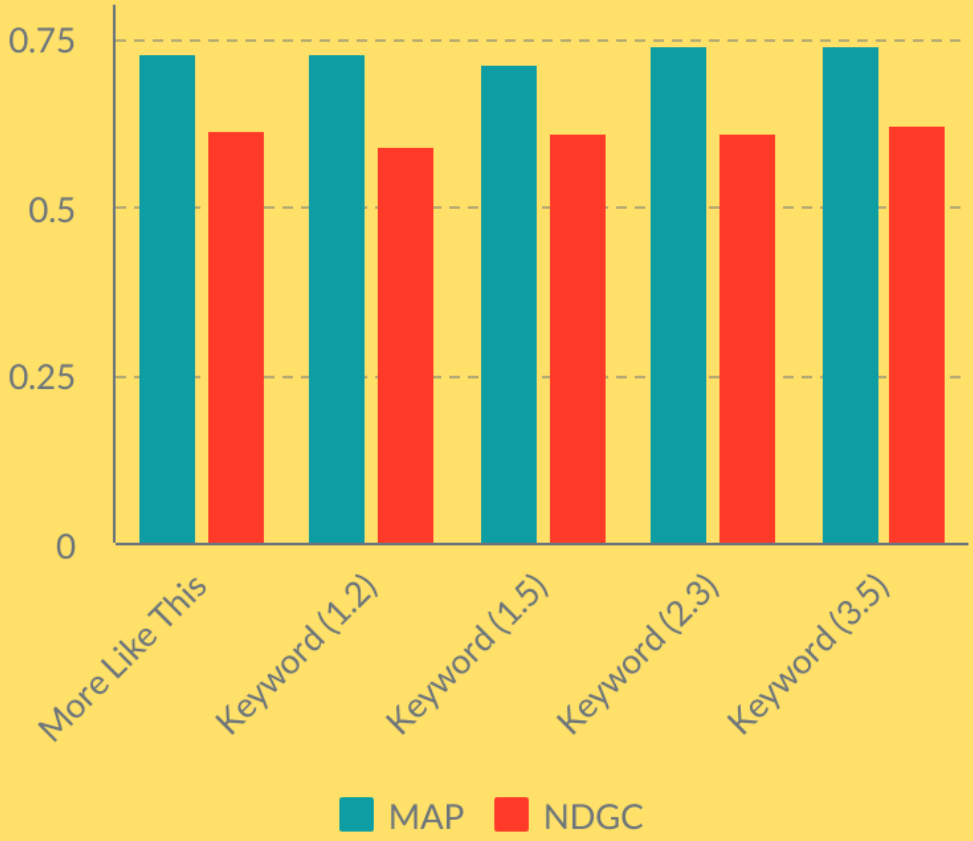
Boost MultiField

Abbiamo ritenunto che un hit sul titolo sia più importante che un hit sul testo. Ad esempio, se l'utente cerca "DNA", una pagina che contiente DNA nel titolo abbia maggiore importanza rispetto ad una che non lo contiene



Questi risultati sono stati ottenuti utilizzando l'index V6
BT: Boost Titolo
BC: Boost Contenuto

Query expansion



Questi risultati sono stati ottenuti utilizzando l'index V6
Keyword (D,N):
D=Documenti da cui estrarre parole chiave
N=Numero parole chiave estratte

```
# QUERY EXPANSION CON MORE LIKE THIS
first_hit = results[0]
more_result = first_hit.more_like_this("content")
results.upgrade_and_extend(more_result)
```

```
# QUERY EXPANSION CON KEYWORD NEI PRIMI 10 DOCUMENTI
keywords= [keyword for keyword, score
            in results.key_terms("content", docs=3, numterms=5)]
query_keyword = self.parser.parse((reduce(lambda a, b: a + ' ' + b, keywords)))
results_keyword = searcher.search(query_keyword, limit=max_query_result, terms=True)
results.upgrade_and_extend(results_keyword)
```

Benchmark

Il search engine permette di eseguire le trenta query di test in un blocco unico in modo da rendere più semplice la valutazione.

Non è stata implementata la possibilità di far scegliere all'utente l'algoritmo di scoring in quanto i search engine in commercio non offrono questa possibilità

Feature aggiuntive

Sono state poi aggiunte **feature** in modo da semplificare l'utilizzo all'utente:

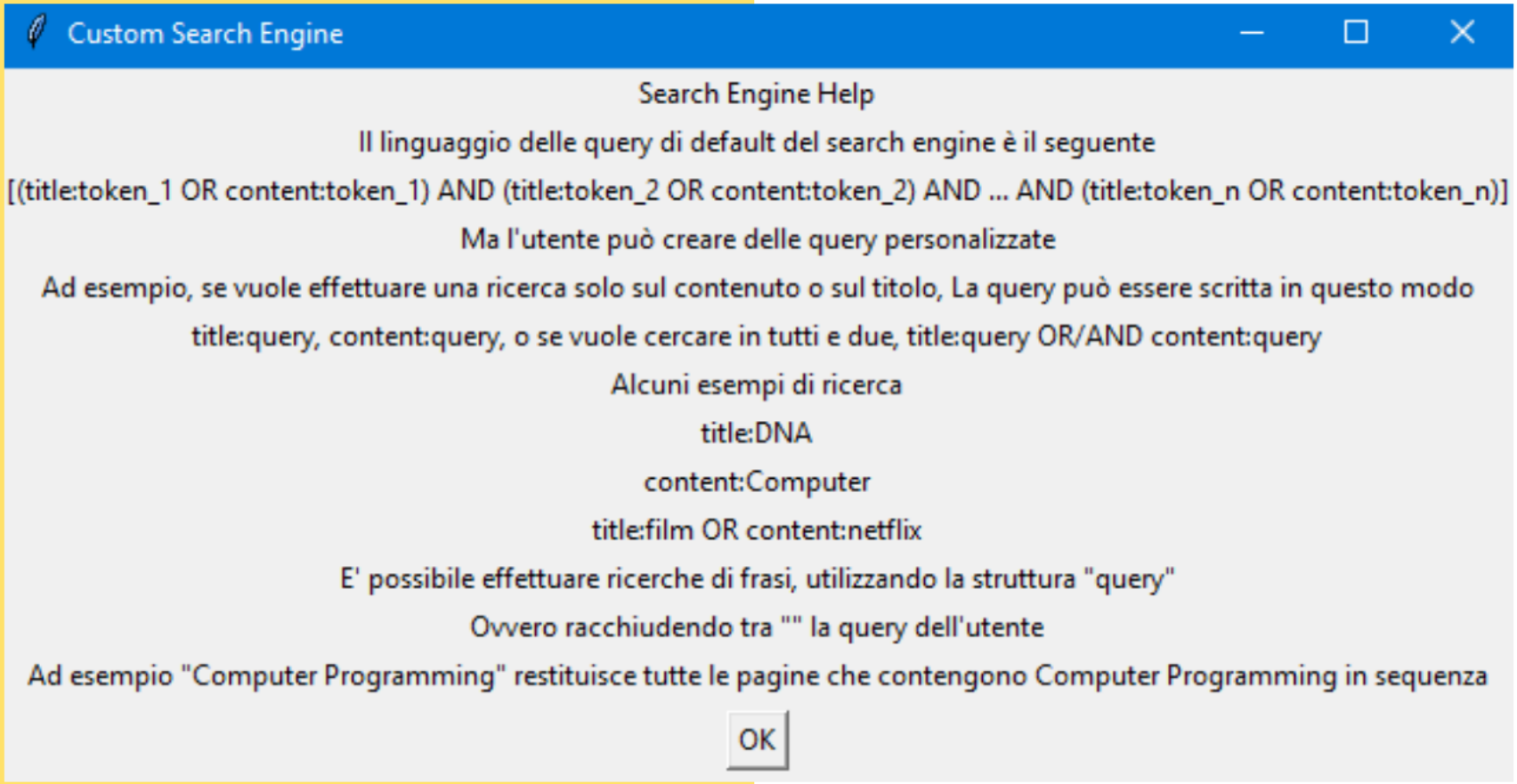
- Possibilità di lettura delle pagine;
- Correttore ortografico;
- Composizione query complesse;

Correttore ortografico

E' data la possibilità all'utente di effettuare un controllo ortografico sulle parole inserite nella query.
Attenzione: il sistema non effettua il controllo e poi la query, ma è l'utente che controlla la query e che dovrà riscriverla in modo corretto.

```
@staticmethod
def suggestion_word(query):
    chkr = enchant.Dict("en_US")
    return [word for word in chkr.suggest(query)]
```

Composizione complesse di query



Suddivisione dei lavori

Abbiamo collaborato insieme per lo sviluppo del search engine suddividendoci il carico di lavoro come riportato di seguito:

Marco Incerti

- Algoritmo di ricerca
- Espansione query

Alessandro Palmieri

- Preprocessing
- Query complesse
- Suggerimento parole

Alessandro Bedini

- Creazione indice
- Multifield

Conclusione

Con il progetto realizzato si è cercato di implementare un search engine di base e cercare di migliorarlo introducendo tecniche di preprocessing, query expansion e algoritmi di ordinamento.
Per quanto riguarda le funzionalità che un utente può utilizzare si è cercato, nei minimi del possibile, di riprodurre quelle dei search engine più diffusi.