Web Information Retrieval - Homework 1

Simone Conia, Marco Favorito, Shani Dana Guetta

April 19 2017

Contents

1 Homework Overview **1**
- 1.1 MG4J . . . 1
- 1.2 Stemmers . . . 1
- 1.3 Scorers . . . 1
- 1.4 The Cranfield Collection . . . 2
- 1.5 The Time Collection . . . 2
- 1.6 Scripts . . . 2

2 Using MG4J **3**
- 2.1 Environment setup . . . 3
- 2.2 Creating a collection . . . 5
- 2.3 Building an inverted index . . . 5
- 2.4 Querying an index and ranking the documents . . . 6

3 Scores Aggregation **8**
- 3.1 Fagin's Algorithm . . . 8
- 3.2 Threshold Algorithm . . . 10

4 Search Engine Evaluation **13**
- 4.1 Average R-Precision . . . 13
  - 4.1.1 Computing R-Precision . . . 13
  - 4.1.2 Computing Average R-Precision . . . 14
  - 4.1.3 Cranfield Collection Results . . . 15
  - 4.1.4 Time Collection Results . . . 16
- 4.2 Average nMDCG . . . 17
  - 4.2.1 Computing MDCG . . . 17
  - 4.2.2 Computing nMDCG . . . 18
  - 4.2.3 Computing Average nMDCG . . . 18
  - 4.2.4 Cranfield Collection Average nMDCG Results . . . 20
  - 4.2.5 Time Collection nMDCG Results . . . 25

5 Conclusions **28**
5.1 Best configuration 28
5.2 Best stemmer 30
5.3 Best scorer function 32

1 Homework Overview

The objectives of this homework are to configure a search engine on two collections of documents, the Cranfield collection and the Time collection, and to find the best configuration in terms of stemming method and scorer function for the search engine.

1.1 MG4J

For this homework, the search engine of choice is MG4J (Managing Gigabytes for Java), a free full-text search engine for large document collections written in Java. It allows to index and query large collection of documents with just few simple steps:

- Create a collection of documents
- Create an inverted index on the created collection
- Query the created index

For more details about MG4J visit:
http://mg4j.di.unimi.it

1.2 Stemmers

MG4J includes some stemmers, but it also provides support for custom stemmers (like homework.EnglishStemmerStopwords). For this homework, the stemmers of interest are the following ones:

- **it.unimi.di.big.mg4j.index.NullTermProcessor:** This is the default stemmer in MG4J: a term processor that accepts all terms and does not do any processing.
- **it.unimi.di.big.mg4j.index.snowball.EnglishStemmer:** A stemmer for the English language.
- **homework.EnglishStemmerStopwords:** A stemmer for the English language able to filter stop words like ”a”, ”an”, ”the”, etc...

1.3 Scorers

The scorers taken into considerations are:

- **CountScorer:** A trivial scorer function that computes the score by adding the number of occurrences within the current document of each query term.
- **TfIdfScorer:** A scorer that implements the TF/IDF ranking formula.

- **BM25Scorer:** A scorer that implements the BM25 ranking scheme. BM25 is the name of a ranking scheme for text derived from the probabilistic model. The essential feature of the scheme is that of assigning to each term appearing in a given document a weight depending both on the count (the number of occurrences of the term in the document), on the frequency (the number of the documents in which the term appears) and on the document length (in words).

1.4 The Cranfield Collection

The search engine is evaluated over the Cranfield Collection. This collection consists of:

- 1400 html documents. These documents contain simple science articles.
- 225 predefined queries.
- 1249 relevance assessments, that is, for each query, a set of relevant documents: the Ground Truth.

For more details about the Cranfield collection visit:
http://ir.dcs.gla.ac.uk/resources/test_collections/cran

1.5 The Time Collection

The search engine is evaluated also over the Time Collection. This collection consists of:

- 423 html documents. These documents contain simple news stories.
- 83 predefined queries.
- 324 relevance assessments, that is, for each query, a set of relevant documents: the Ground Truth.

For more details about the Time collection visit:
http://ir.dcs.gla.ac.uk/resources/test_collections/time/

1.6 Scripts

This homework often mentions some bash or python scripts, but only the most relevant ones are presented and discussed in detail. All the scripts can be found at: https://github.com/MarcoFavorito/wir-hw1.

2 Using MG4J

2.1 Environment setup

Before actually using MG4J, the following configuration script is used to set up the environment:

Listing : main.sh

```
if [ $# != 4 ]
  then
    echo "No argument supplied"
    echo "Usage: main.sh <dataset_path> <collection_name> <title_weight>
     <text_weight>"
    echo -e "Examples:"
    echo -e "\tmain.sh Cranfield_DATASET cran 2 1"
    echo -e "\tmain.sh Time_DATASET time 0 1"
    exit -1
fi

echo "Setting environment..."

export dataset_path=$1
export collection_name=$2
export title_text_weights="$3 $4"

export HW_DIR="$(pwd)"
export script_folder=$HW_DIR/scripts
export collection_path=$dataset_path/$collection_name
export ground_truth=${collection_path}/${collection_name}_all_queries.tsv
export output_path=out/${collection_name}



export stemmers=( it.unimi.di.big.mg4j.index.NullTermProcessor it.unimi.
    di.big.mg4j.index.snowball.EnglishStemmer homework.
    EnglishStemmerStopwords )
export stemmer_names=( default english englishsw )
export scorer_functions=( CountScorer TfIdfScorer BM25Scorer )
export fields=( text title text_and_title )
export OUT_ENGLISHSW_BM25S=${output_path}/scores/englishsw/BM25Scorer

export k_values=( 1 3 5 10 )

echo "Calling the main script..."
source $script_folder/run.sh
```

In order to keep everything as clean as possible, the work has been split into smaller tasks. The following script shows how the work has been organized into separate scripts.

Listing : scripts/run.sh

```
#!/bin/bash

debug_msg="MAIN"

echo
echo $debug_msg - start...

if [ ! -d $script_folder ]; then
    echo $debug_msg - cannot find \"scripts\" folder...
    echo $debug_msg - exiting...
    echo
    exit 1
fi

source $script_folder/0-clean.sh
source $script_folder/1-set-my-classpath.sh
source $script_folder/2-create-collection.sh
source $script_folder/3-build-index.sh
source $script_folder/4-apply-scorer-functions.sh
source $script_folder/5-aggregation.sh
source $script_folder/6-evaluation.sh
source $script_folder/7-make-plots.sh

rm -R $script_folder/__pycache__
rm $script_folder/*.pyc

echo "MAIN" - done!
echo
```

In particular:

- **0-clean.sh** clears any leftover produced by previous executions of the run.sh script.
- **1-set-my-classpath.sh** sets the paths to the MG4J libraries.
- **2-create-collection.sh** creates the document collection to work on.
- **3-build-index.sh** build the inverted indices using various combination of stemmers and scorer functions.
- **4-apply-scorer-functions.sh** queries the indices using a predefined set of queries, and returns the most relevant results.
- **5-aggregation.sh** aggregates the scores of the of some given fields for a given stemmer/scorer configuration.
- **6-evaluation.sh** evaluates the search engines using Average-R-Precision and nMDCG.

- **7-make-plots.sh** draws some plots to show the behavior of the search engine.

2.2 Creating a collection

The following script creates a collection from a set of HTML documents. For example, for the Cranfield Collection, it will create a collection named *cran.collection* using all the HTML documents found in the *Cranfield_DATASET/cran* folder.

Listing : scripts/2-create-collection.sh

```
#!/bin/sh

debug_msg="CREATE_COLLECTION"

echo
echo $debug_msg - start...

if [ ! -d ${output_path} ]; then
    echo $debug_msg - creating \"${output_path}\" directory...
    mkdir ${output_path} --parents
fi

find $collection_path -iname \*.html | \
    java it.unimi.di.big.mg4j.document.FileSetDocumentCollection \
        -f HtmlDocumentFactory -p encoding=UTF-8 $output_path/
     $collection_name.collection

echo $debug_msg - done!
echo
```

2.3 Building an inverted index

The analysis concerns three stemmers: the default one, the English stemmer, and the custom English stemmer that filters stop words. This means that three different indices have to be built. The following script creates three different inverted indices using the three given stemmers, and then it stores them in three separate directories.

Listing : scripts/3-build-index.sh

```
#!/bin/sh

debug_msg="BUILD_INDEX"

build_index () {
    stemmer=$1
    stemmer_name=$2
```

```
8
9       echo $debug_msg - creating new \"${output_path}/indices/$stemmer_name
        \" folder...
10      mkdir ${output_path}/indices/$stemmer_name --parents
11
12      java it.unimi.di.big.mg4j.tool.IndexBuilder -t $stemmer -S ${
        output_path}/$collection_name.collection ${output_path}/indices/
        $stemmer_name/$collection_name
13
14      # create a link for the .collection file into the index folder.
15      ln ${output_path}/$collection_name.collection ${output_path}/indices/
        $stemmer_name/$collection_name.collection
16  }
17
18  echo
19  echo $debug_msg - start...
20
21  if [ ! -d out ]; then
22      echo $debug_msg - cannot find \"out\" directory...
23      echo $debug_msg - exiting...
24      exit 1
25  fi
26
27  if [ -d ${output_path}/indices ]; then
28      rm -r ${output_path}/indices
29  fi
30  mkdir ${output_path}/indices --parents
31
32  for i in {0..2}; do
33      build_index ${stemmers[i]} ${stemmer_names[i]}
34  done
35
36  echo $debug_msg - done!
37  echo
```

2.4 Querying an index and ranking the documents

At this point, a set of predefined queries is given to the search engine. For each query, the search engine will return the most relevant documents and their corresponding scores. These results will be used for score aggregation and search engine evaluation.

Listing : scripts/4-apply-scorer-functions.sh

```
1   #!/bin/sh
2
3   debug_msg="APPLY_SCORER_FUNCTIONS"
4
```

```
echo
echo $debug_msg - start

if [ ! -d ${output_path}/indices ]; then
    echo $debug_msg - cannot find \"${output_path}/indices\" directory...
    echo $debug_msg - exiting...
    exit 1
fi

if [ -d ${output_path}/scores ]; then
    rm -r ${output_path}/scores
fi

for stemmer in ${stemmer_names[@]}; do
    for scorer_function in ${scorer_functions[@]}; do
        mkdir ${output_path}/scores/$stemmer/$scorer_function --parents

        for field in ${fields[@]}; do
            echo "${output_path}/indices/${stemmer}/${collection_name}"
            java homework.RunAllQueries_HW \
                ${output_path}/indices/${stemmer}/${collection_name} \
                ${dataset_path}/${collection_name}_all_queries.tsv \
                $scorer_function \
                $field \
                ${output_path}/scores/${stemmer}/${scorer_function}/${
 field}.tsv
        done
    done
done

echo $debug_msg - done!
echo
```

3 Scores Aggregation

The following Python scripts implement two scores aggregation algorithms: Fagin's and Threshold. Given a set of rankings, these two algorithms return the top-k elements of a new ranking whose scores are computed by a so called aggregation function.

For this homework, it is requested to aggregate and analyze the scores obtained on the *Title* and *Text* fields when the *BM25Scorer* is applied to the stop-word-filtering-English stemmer. The aggregated score of a document here is defined as the sum of its *Title* score and *Text* score, where the *Title* score is twice as important as the *Text* score.

The output of both the scripts are two files with the following format:

```
Query_ID    Doc_ID    Rank    Score
...         ...       ...     ...
```

These files will be used to evaluate the performance of the scores aggregation strategy.

3.1 Fagin's Algorithm

Given a set of rankings, the Fagin's algorithm aggregates these rankings, and then it returns a list containing the top-k document IDs and their corresponding scores.

The Python function takes in input:

- **k** is the number of top documents to return.
- **title_scorings** is a list of tuples (docID, rank, score), sorted by decreasing score.
- **text_scorings** is a list of tuples (docID, rank, score), sorted by decreasing score.
- **title_weight** is the relative importance of the *Title* field.
- **text_weight** is the relative importance of the *Text* field.

The algorithm is the following one:

1. Sequentially access all lists in parallel until there are k document IDs that have been seen in all lists.
2. Insert in the seen dictionary the document IDs with their score considering the *Title* scores are twice as important as *Text* scores.
3. Perform random accesses to obtain the scores of all the entries that have been seen but not in all the lists.

4. Compute the scores for all objects and return the document IDs with the top-k aggregated scores.

Listing : scripts/fagin.py

```
from itertools import zip_longest

def fagin(k, title_scorings, text_scorings, title_weight, text_weight):
  """
  Applies the Fagin's Algorithm on the inputs
  :param k: the parameter that determines the number of docs returned
  :param title_scorings: list of tuples (docId, score), sorted by
     decreasing score
  :param text_scorings: list of tuples (docId, score), sorted by
     decreasing score
  :return: a list of tuples (docId, score), with length k, sorted by
     decreasing score,
  Notice: the lists @title_scorings and @text_scorings can be of
     different length (either empty!)
  """
  rank = {}

  # get data as dict with key: docId and value: score of that docId for
     the query
  title_dict = dict(title_scorings)
  text_dict = dict(text_scorings)

  # computes the seen docs, according to the Fagin Algorithm
  seen = compute_seen_doc_ids(k, title_scorings, text_scorings)

  # compute the score of the seen docs, applying the correct weight
  # retrieve the scores with "random access" through dictionaries
  for doc_id in seen:
    score = title_weight * title_dict.get(doc_id, 0) + text_weight *
     text_dict.get(doc_id, 0)
    rank[doc_id] = score

  # sort the list of (docId, score) by decreasing score and take the
     first k
  sorted_rank = sorted(list(rank.items()), key=lambda x: x[1], reverse=
     True)[:k]

  return sorted_rank
  # get data as dict with key: docId and value: score of that docId for
     the query


def compute_seen_doc_ids(k, title_scorings, text_scorings):
```

```
    # the counter that keeps track of the
    # number of docId seen in both the lists
    # until a given iteration
    counter = 0
    # the set of seen docIds
    seen = set()

    # even if the two list of scorings are of variable length,
    # itertools.zip_longest fills the shorter list with "None" values
    # and make the two list with equal length
    for title_entry, text_entry in zip_longest(title_scorings,
        text_scorings):
      # take the ids if the entry exists, otherwise return None
      title_entry_id = title_entry[0] if type(title_entry) is tuple else
       None
      text_entry_id = text_entry[0] if type(text_entry) is tuple else None

      # if the id is already in the "seen" set, then increase the counter.
      # otherwise add the current docId in the set.
      if title_entry_id in seen: counter += 1
      elif title_entry_id: seen.add(title_entry_id)

      if text_entry_id in seen: counter += 1
      elif text_entry_id: seen.add(text_entry_id)

      # if we have seen for two times at least k elements,
      # then stop and return the "seen" set.
      if counter >= k:
        break

    return seen
```

3.2 Threshold Algorithm

As the Fagin's algorithm, the Threshold Algorithm aggregates these rankings, and then it returns a list containing the top-k document IDs and their corresponding scores.

The Python function takes in input:

- **k** is the number of top documents to return.
- **title_scorings** is a list of tuples (docID, rank, score), sorted by decreasing score.
- **text_scorings** is a list of tuples (docID, rank, score), sorted by decreasing score.
- **title_weight** is the relative importance of the *Title* field.

- **text_weight** is the relative importance of the *Text* field.

The algorithm is the following one:

1. Consider a row in both lists *title_scorings* and *text_scorings*
2. Set the threshold value to be the aggregate of the scores seen in the considered rows.
3. Do random accesses through the dictionaries to compute the score of all seen objects.
4. Sort the scores of the objects seen in the previous step by descending order, and take the top-k objects.
5. If all scores in the top-k list are greater than the threshold value, stop and return this list.
6. Otherwise consider the next row and go back to step 1.

Listing : scripts/threshold.py

```
from itertools import zip_longest

def threshold(k, title_scorings, text_scorings, title_weight, text_weight
    ):
  """
  Applies the Fagin's Algorithm on the inputs
  :param k: the parameter that determines the number of docs returned
  :param title_scorings: list of tuples (docId, score), sorted by
     decreasing score
  :param text_scorings: list of tuples (docId, score), sorted by
     decreasing score
  :return: a list of tuples (docId, rank, score), with length k, sorted
     by decreasing score,
  """

  # a dict with key: docId seen and value: score
  seen = {}

  # get data as dict with key: docId and value: score of that docId for
     the query
  title_dict = dict(title_scorings)
  text_dict = dict(text_scorings)

  # even if the two list of scorings are of variable length,
  # itertools.zip_longest fills the shorter list with "None" values
  # and make the two list with equal length
  for title_entry, text_entry in zip_longest(title_scorings,
     text_scorings):
```

```
23      # take the pair (ids, score)if the entry exists, otherwise return (
         None, 0)
24      title_entry_id, title_entry_score = (title_entry[0], title_entry[1])
         if type(title_entry) is tuple else (None, 0)
25      text_entry_id, text_entry_score = (text_entry[0], text_entry[1]) if
         type(text_entry) is tuple else (None, 0)
26
27      # compute the threshold of the current iteration
28      cur_threshold = title_weight * title_entry_score + text_weight *
         text_entry_score
29
30      # if the id is not None, add it to the "seen" and associate it with
         its aggregated score
31      # retrieve the scores with "random access" through dictionaries
32      if title_entry_id:
33        seen[title_entry_id] = title_weight * title_dict.get(title_entry_id
         , 0) + text_weight * text_dict.get(title_entry_id, 0)
34      if text_entry_id:
35        seen[text_entry_id] = title_weight * title_dict.get(text_entry_id,
         0) + text_weight * text_dict.get(text_entry_id, 0)
36
37      # sort the list of (docId, score) by decreasing score and take the
         first k
38      top_k = sorted(list(seen.items()), key=lambda x: x[1], reverse=True)
         [:k]
39
40      # IF
41      #   the list of best #k element has length at least #k
42      #   AND
43      #   all the scores of these docs are greater than the current
         threshold
44      # THEN
45      #   RETURN that list
46      if len(top_k) >= k and all([score >= cur_threshold for _, score in
         top_k]):
47        return top_k
48
49    raise Exception("There should be at least k elements...")
```

4 Search Engine Evaluation

One of the objectives of this homework is to find the best configuration (in terms of stemming method and scorer function) for the search engine, using the available Ground-Truth data. The following Python scripts implement two metrics used to evaluate the performance of each stemmer/scorer configuration and each aggregation algorithm seen in the previous sections.

4.1 Average R-Precision

The first metric used to evaluate the search engine is the Average R-Precision.

4.1.1 Computing R-Precision

Computing R-precision requires knowing a set of documents that are relevant to a query. The number of relevant documents, R, is used as the cutoff for calculation, and this varies from query to query. For example, if there are 15 documents relevant to "red" in a corpus ($R = 15$), R-precision for "red" looks at the top 15 documents returned, counts the number that are relevant r turns that into a relevancy fraction: $r/R = r/15$[1].

The Python function that implements R-Precision takes in input:

- **retrieved_docs** is a list of tuples (docID, rank, score) sorted by decreasing score, representing the ranking computed by the search engine for query of interest.
- **relevant_docs** is a list that represents the set of relevant documents (docID) specified in the ground-truth file for query of interest.

Listing : scripts/r-precision.py

```
1  def R_precision(retrieved_docs,relevant_docs):
2    """
3
4    :param retrieved_docs: list of tuples (docId, rank, score), sorted by
       score
5    :param relevant_docs: list of tuples (docId) of the ground-truth
6    :return:
7    """
8
9    relevant_docs_number = len(relevant_docs)
10
11   # take only the best |ground truth| docs, according to the R-Precision
12   top_k_retrieved_docs = retrieved_docs[:relevant_docs_number]
13
14   true_positives = 0
```

[1] https://en.wikipedia.org/wiki/Information_retrieval#R-Precision

```
    false_positives = 0

    # count the true positives and the true negatives
    for docId in top_k_retrieved_docs:
      if docId in relevant_docs:
        true_positives += 1
      else:
        false_positives += 1

    return true_positives / (true_positives + false_positives)
```

4.1.2 Computing Average R-Precision

The Average R-Precision for a set of queries is the mean of the R-Precision scores for each query.

$$\text{Average R-Precision} = \frac{\sum_{q=1}^{Q} \text{R-Precision(q)}}{Q}$$

where Q is the number of queries.

The Python function that implements Average R-Precision takes in input:

- **all_retrieved_docs** is a dictionary with key (queryID) and value a list of tuples (docID, rank, score) sorted by decreasing score.
- **all_relevant_docs** is a dictionary with key (queryID) and value a list of documentIDs (docID). This dictionary represents the Ground-Truth.

Listing : scripts/r-precision.py

```
def averaged_R_Precision(all_retrieved_docs, all_relevant_docs):
  """
  Compute the averaged R-Precision
  :param all_retrieved_docs:  dict[key: qId, value: list[(docId, rank,
     score)] sorted by score ]
  :param all_relevant_docs:   dict[key: qId, value: list[(docId)]
  :return: averaged R-Precision among all queries
  """

  n_queries = len(all_relevant_docs)
  precisions = []

  # compute the R-Precision for each query and store in a list
  for qId in all_relevant_docs.keys():
    cur_retrieved = all_retrieved_docs.get(qId,[(None,None,None)])
    cur_retrieved, _, _ = zip(*cur_retrieved)
    cur_relevant = all_relevant_docs[qId]
```

```
      cur_R_precision = R_precision(cur_retrieved, cur_relevant)
      precisions.append(cur_R_precision)

    # compute the averaged R-Precision from the R-Precision of each query
    average = sum(precisions)/n_queries

    return average
```

4.1.3 Cranfield Collection Results

These are the results for the Average R-Precision evaluation on each stemmer-scorer configuration:

Stemmer	Scorer	Field	Average R-Precision
default	CountScorer	text	0.02030036038656728
default	CountScorer	title	0.06077953103815172
default	CountScorer	text_and_title	0.02578652837273527
default	TfIdfScorer	text	0.1943322947633293
default	TfIdfScorer	title	0.17221935023659166
default	TfIdfScorer	text_and_title	0.17932787501753014
default	BM25Scorer	text	0.23548681609026445
default	BM25Scorer	title	0.20043697112662626
default	BM25Scorer	text_and_title	0.2549430381326934
english	CountScorer	text	0.023642988729195624
english	CountScorer	title	0.07690215233318681
english	CountScorer	text_and_title	0.02997251014492394
english	TfIdfScorer	text	0.2083948239120653
english	TfIdfScorer	title	0.17517752560856006
english	TfIdfScorer	text_and_title	0.1893509074543557
english	BM25Scorer	text	0.25755338557062707
english	BM25Scorer	title	0.22468447296033506
english	BM25Scorer	text_and_title	0.26239166583994167
englishsw	CountScorer	text	0.12768598587564103
englishsw	CountScorer	title	0.19686634988359122
englishsw	CountScorer	text_and_title	0.16137814715400925
englishsw	TfIdfScorer	text	0.2081881517226345

```
englishsw   TfIdfScorer     title             0.17298519626105832
englishsw   TfIdfScorer     text_and_title    0.19088393915980117

englishsw   BM25Scorer      text              0.25153428989635895
englishsw   BM25Scorer      title             0.2213059148403976
englishsw   BM25Scorer      text_and_title    0.2664739776808743
```

These are the results for the Average R-Precision evaluation on the aggregations algorithms:

```
Algorithm          Average R-Precision

Fagin              0.24875806987875954
Threshold          0.24875806987875954
```

4.1.4 Time Collection Results

These are the results for the Average R-Precision evaluation on each stemmer-scorer configuration.

Only the *Text* field is taken into consideration because in the Time Collection the *Title* field has no meaning and it is not informative. This is because the *HtmlDocumentFactory Text* and *Title* fields correspond to the <body> and <head> tags of an Html document. For the Time Collection, the <head> tag contains only the title of the document, so it is not semantically relevant.

```
Stemmer        Scorer       Field            Average R-Precision

default     CountScorer     text              0.05
default     TfIdfScorer     text              0.0
default     BM25Scorer      text              0.0

english     CountScorer     text              0.03584532248600286
english     TfIdfScorer     text              0.4446586073977995
english     BM25Scorer      text              0.5511641815149965

englishsw   CountScorer     text              0.2392485556022055
englishsw   TfIdfScorer     text              0.4386345110122573
englishsw   BM25Scorer      text              0.5385187947732243
```

These are the results for the Average R-Precision evaluation on the aggregation algorithms:

```
Algorithm          Average R-Precision

Fagin              0.5385187947732243
Threshold          0.5385187947732243
```

In the case of the Time Collection, the aggregation is not effective at all since the *Title* field contains no relevant information, as discussed above. And even if it contained relevant information, just consider the following values, which are the scores of the documents for the *Title* field using the "EnglishStemmerStopWords - BM25Scorer" configuration (the configuration used for the aggregation):

```
Query_ID  Doc_ID  Rank  Score
13  14  1 2.820362169344532
21  32  1 2.820362169344532
76  31  1 2.820362169344532
76  22  2 2.820362169344532
78  12  1 2.820362169344532
```

Probably, these scores are so few they do not give enough contribution to make the relatives document IDs important enough to be included in the *top k* documents of the aggregation.

Some observations

The reader might have noticed that the Average R-Precision of the Fagin's algorithm output and Threshold algorithm output is the same; this result is not by chance but is due to the fact that both Fagin's and Threshold algorithm, if they are used with the same k, produce equivalent outputs, i.e. given the same input they return the same output[2]. Obviously, from this follows that the R-Precision evaluation applied over the aggregations is the same, since the aggregation outputs are identical.

4.2 Average nMDCG

The evaluation metric nMDCG (normalized Modified Discounted Cumulative Gain) is a modified version of the nDCG (normalized Discounted Cumulative Gain).

4.2.1 Computing MDCG

The MDCG of a particular query and value of k is defined as follow:

$$\text{MDCG}(\text{query}, \text{k}) = \text{relevance}(\text{doc}_1, \text{query}) + \sum_{rank=2}^{k} \frac{\text{relevance}(\text{doc}_{\text{rank}}, \text{query})}{log_2(rank)}$$

where the *relevance* function is defined as follows:

$$\text{relevance}(\text{doc}, \text{query}) = \begin{cases} 1, & \text{doc} \in \text{GroundTruth}\,. \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

[2]The main difference is about performances: Fagin is a simpler algorithm, but at the same time it is not cost optimal for some aggregation functions

4.2.2 Computing nMDCG

The nMDCG of a particular query and value of k is defined as follow:

$$\text{nMDCG}(\text{query}, \text{k}) = \frac{\text{MDCG}(\text{query}, \text{k})}{\text{MaximumMDCG}(\text{query}, \text{k})}$$

where the *MaximumMDCG* function is defined as follows:

$$\text{MaximumMDCG}(\text{query}, \text{k}) = 1 + \sum_{rank=2}^{k} \frac{1}{log_2(rank)}$$

The MaximumMDCG can be seen as the MDCG of a perfect ranking algorithm.

4.2.3 Computing Average nMDCG

The Average nMDCG is simply defined as the average value of the nMDCG for each query.

The Python function takes in input:

- **k** that it is requested to be these 4 values 1, 3, 5, 10.
- **all_retrieved_docs** is a dictionary [key:queryID, value:list[docId, rank, score]] sorted by score.
- **all_relevant_docs** is a dictionary [key:queryID, value:list[docId]]

The functions returns the averaged nMDCG on every query.

Listing : scripts/mdcg.py

```
from math import log2


def maximumMDCG(k, groundTruth):

    # if k is greater equal the number of relevant docs
    # (i.e. the length of @groundTruth) then
    # Compute MDCG with the number of relevant docs.
    max_k = k if k < len(groundTruth) else len(groundTruth)

    res = 1 + sum([1/log2(rank+1) for rank in range(1, max_k)])
    return res

def relevance(docId, groundTruth):
    return 1 if docId in groundTruth else 0

def MDCG(k, retrieved_docs, groundTruth):
    res = 0
    res += relevance(retrieved_docs[0], groundTruth)
```

```
    for i in range(1, k):
        res += relevance(retrieved_docs[i], groundTruth)/log2(i+1)

    return res


def nMDCG(k, retrieved_docs, groundTruth):
    cur_MDCG = MDCG(k, retrieved_docs, groundTruth)
    max_MDCG = maximumMDCG(k)

    res = cur_MDCG / max_MDCG
    return res


def averaged_nMDCG(k, all_retrieved_docs, all_relevant_docs):
    """

    :param all_retrieved_docs:  dict[key: qId, value: list[(docId, rank, score)] sorted by score ]
    :param all_relevant_docs:   dict[key: qId, value: list[(docId)]
    :return: averaged nMDCG on every queries
    """
    nMDCG_values = []
    queryIds = all_relevant_docs.keys()
    for qid in queryIds:
        # if there is no doc for a certain query
        # we cannot compute nMDCG, so set it to 0
        if qid not in all_retrieved_docs.keys():
            cur_nMDCG = 0
        else:
            cur_retrieved_docs = all_retrieved_docs[qid]

            # take only the docIds
            cur_retrieved_doc_ids = [entry[0] for entry in cur_retrieved_docs]
            cur_relevant_doc_ids = all_relevant_docs[qid]

            # handle the case when k is greater than the number of retrieved docs
            # in that case, k cannot be greater than that number
            cur_k = k
            max_k = len(cur_retrieved_docs)
            if cur_k>max_k:
                cur_k=max_k

            # compute nMDCG for the current query
            cur_nMDCG = nMDCG(cur_k, cur_retrieved_doc_ids, cur_relevant_doc_ids)
        nMDCG_values.append(cur_nMDCG)
```

```
65
66      averaged_nMDCG = sum(nMDCG_values)/len(nMDCG_values)
67
68      return averaged_nMDCG
```

4.2.4 Cranfield Collection Average nMDCG Results

These are the nMDCG results for the aggregation algorithms with the different values of K. **Note**: the same considerations stated in Section 4.1.4 holds also in this case.

Algorithm	K	nMDCG
Fagin	1	0.2882882882882883
Threshold	1	0.2882882882882883
Fagin	3	0.30910923604136237
Threshold	3	0.30910923604136237
Fagin	5	0.2870066801783458
Threshold	5	0.2870066801783458
Fagin	10	0.2758719271413992
Threshold	10	0.2758719271413992

The plot in Figure 1 shows a comparison between custom scores aggregation and BM25 Scorer on text and title using the English Stop Words Stemmer.

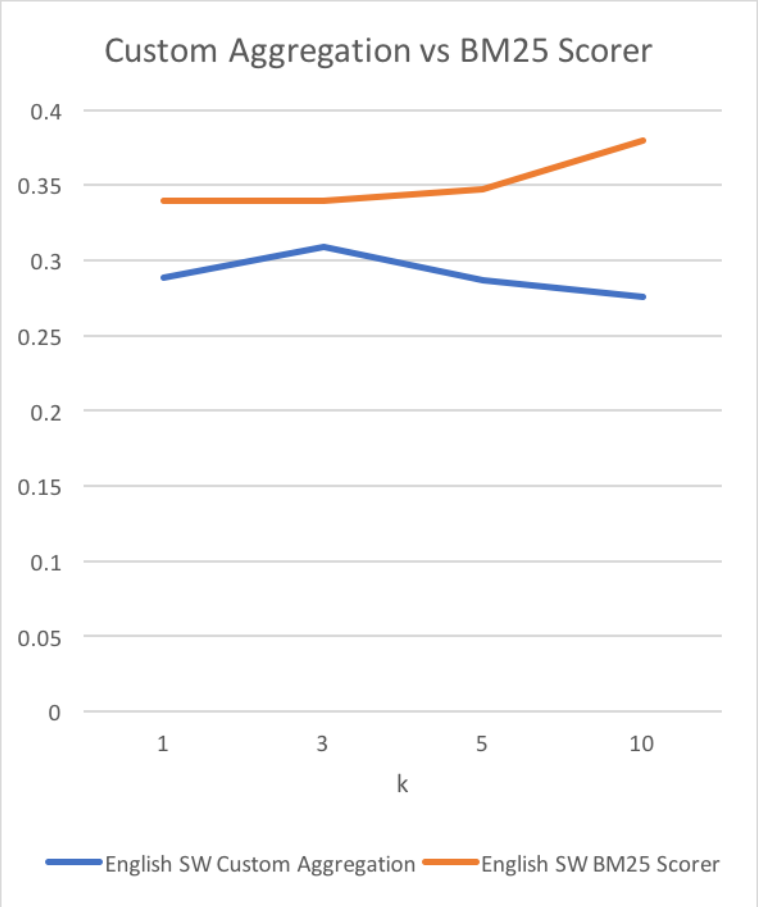

Figure 1: Comparison between custom scores aggregation and BM25 Scorer on text and title using the English Stop Words Stemmer.

Plots in figures 2, 3, and 4 show the Average nMDCG results for the different stemmer-scorer configurations on the Cranfield Collection. On the 'x' axis is represented the value of k (considering only the following values: 1, 3, 5, 10), while on the 'y' axis the average value of nMDCG over all queries.

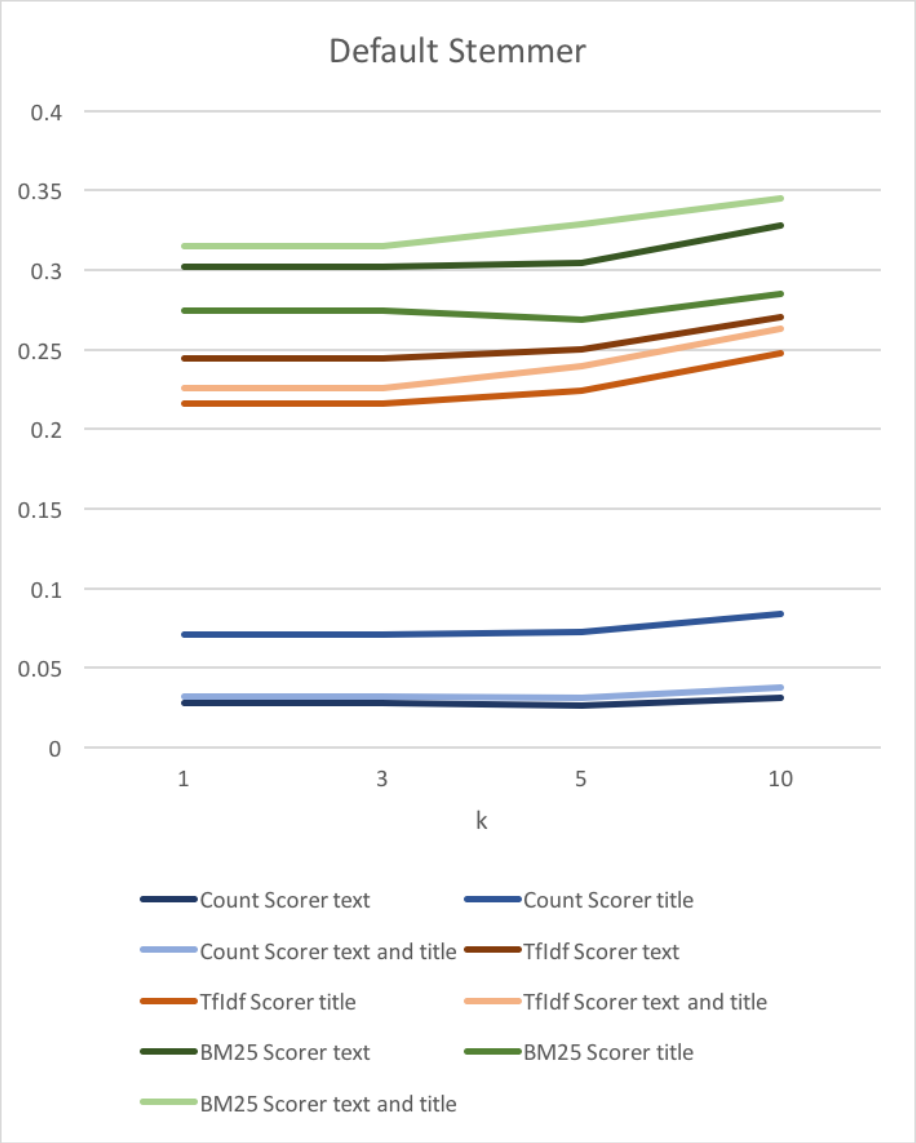

Figure 2: Default Stemmer.

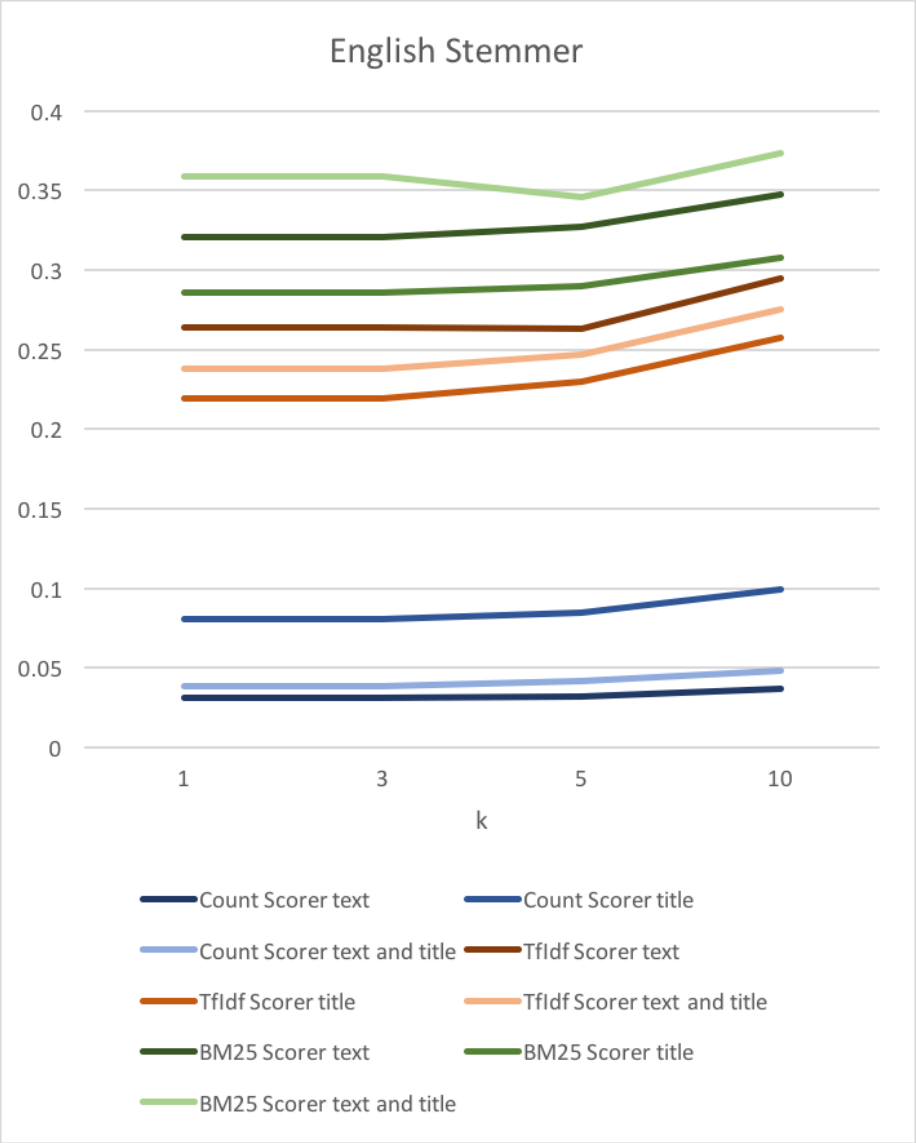

Figure 3: English Stemmer.

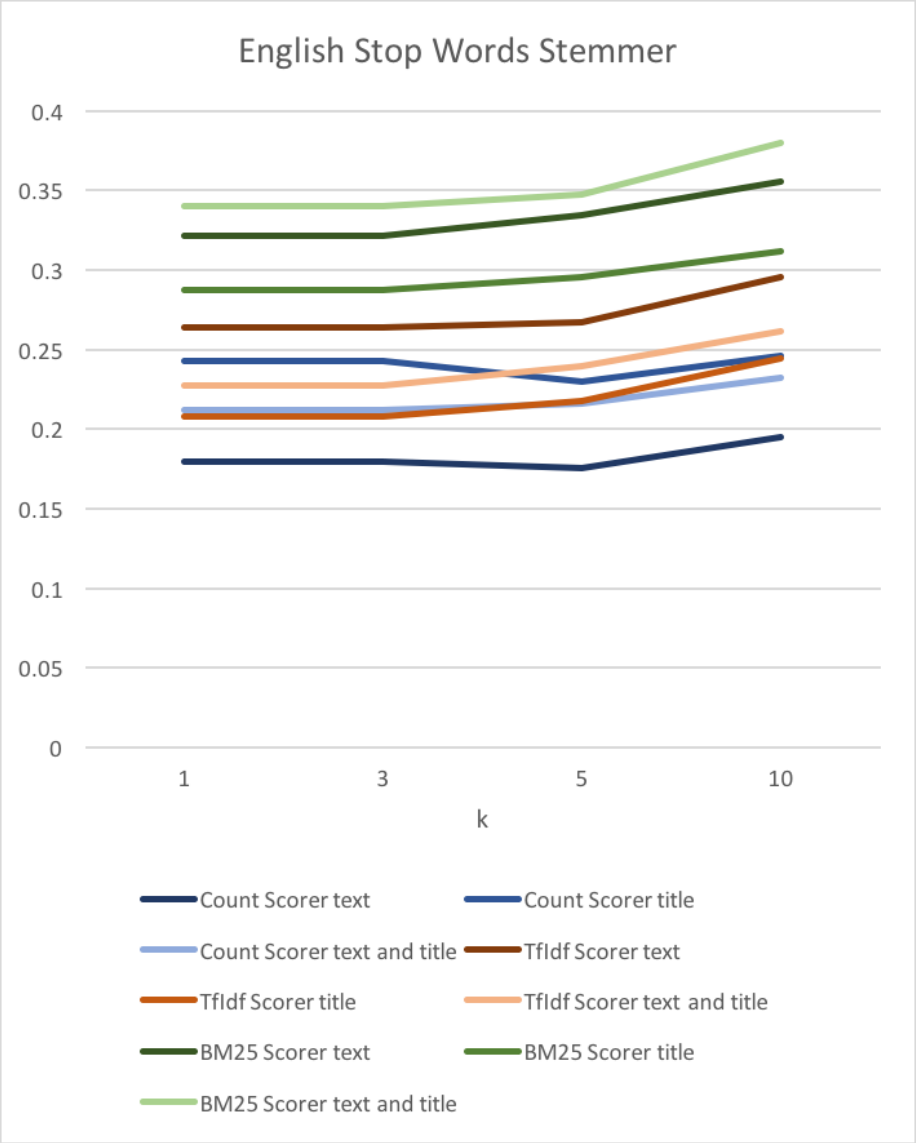

Figure 4: English Stop Words Stemmer.

4.2.5 Time Collection nMDCG Results

These are the nMDCG results for the aggregation algorithms with the different values of K. **Notice**: the same considerations stated in Section 4.1.4 holds also in this case.

Algorithm	K	nMDCG
Fagin	1	0.5903614457831325
Threshold	1	0.5903614457831325
Fagin	3	0.5585508277090627
Threshold	3	0.5585508277090627
Fagin	5	0.5560529519485945
Threshold	5	0.5560529519485945
Fagin	10	0.5474446609447149
Threshold	10	0.5474446609447149

The plot in Figure 5 shows a comparison between custom scores aggregation and BM25 Scorer on text and title using the English Stop Words Stemmer.

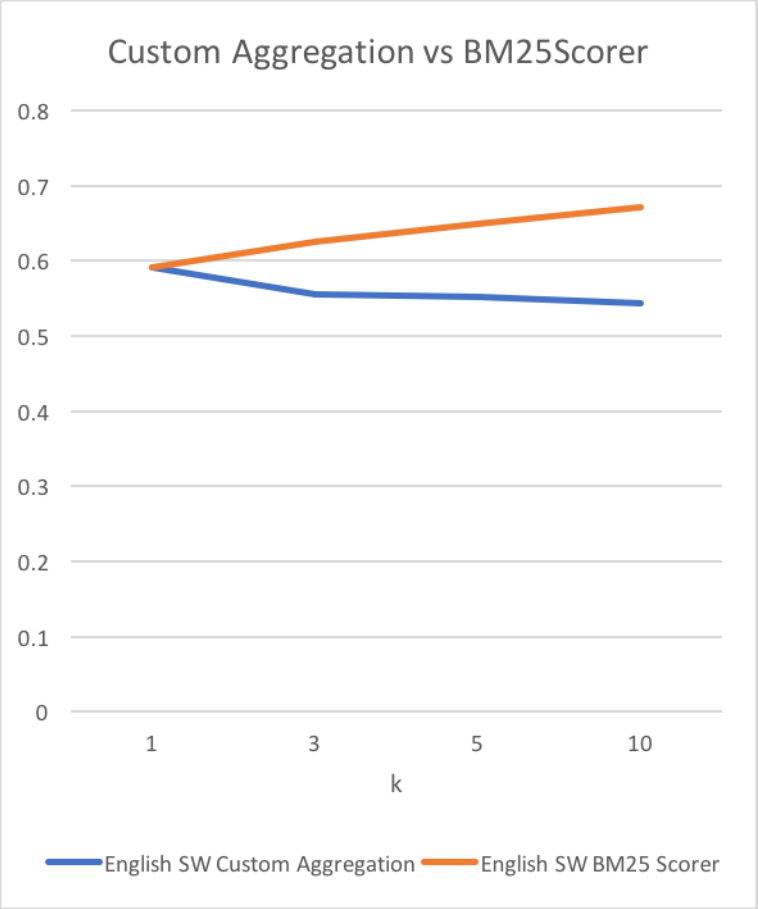

Figure 5: Comparison between custom scores aggregation and BM25 Scorer on text and title using the English Stop Words Stemmer.

Plot in Figure 6 shows the nMDCG results for the different stemmer-scorer configurations on the Time Collection. On the 'x' axis is represented the value of k (considering only the following values: 1, 3, 5, 10) while on the 'y' axis the average value of nMDCG over all queries.

Only the "Text" field is taken in consideration because in Time Collection the "Title" field has no meaning and it is not informative.

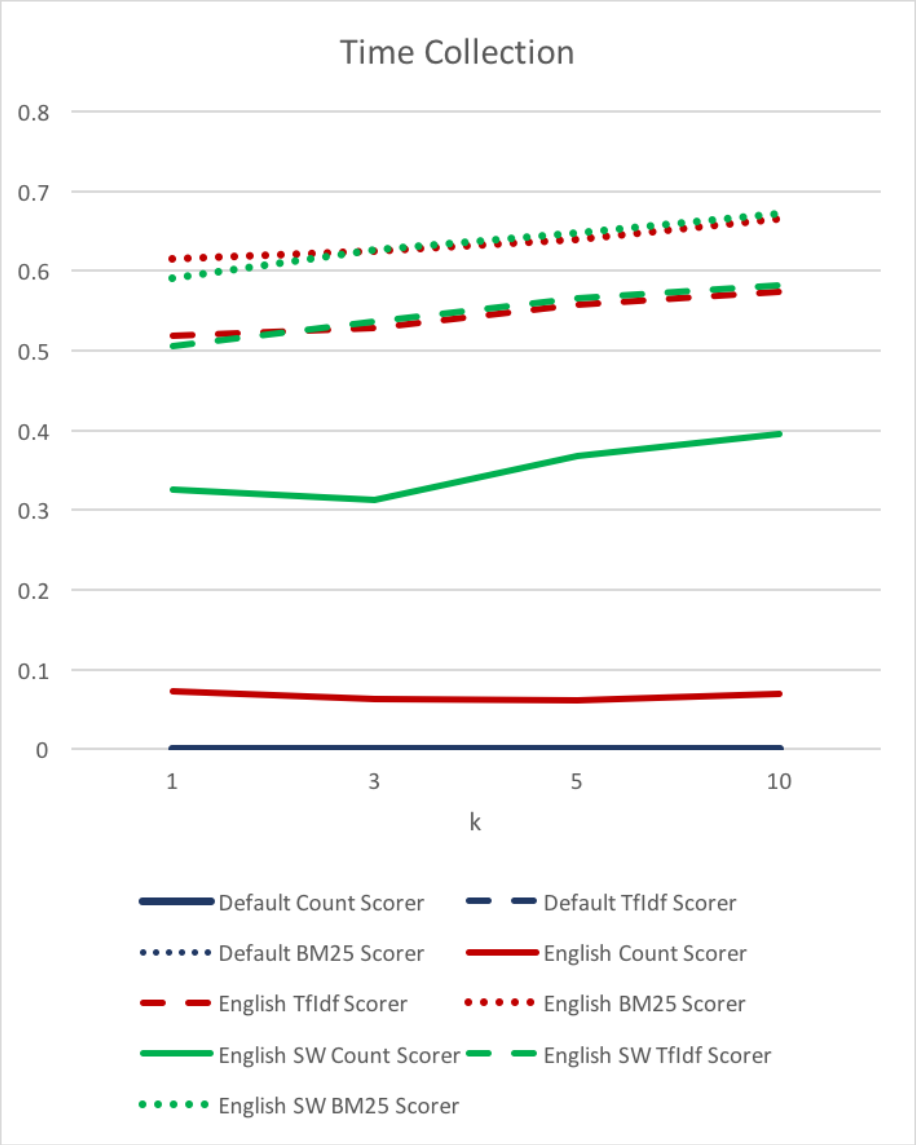

Figure 6: Average nMDCG results on the Text fields for different stemmer/scorer configurations.

5 Conclusions

The main purpose of this section is to analyze the previous results in order to answer questions about the best configuration in terms of stemmer and scorer function, the best stemmer, and the best scorer.

Note: all the following considerations hold for the scores obtained over both "title" and "text" fields.

5.1 Best configuration

The best configuration in terms of stemmer and scorer function is:

- For the Cranfield Collection:
  - According to Averaged R-Precision, "EnglishStemmerStopWords - BM25Scorer" (≈ 0.2665);
  - According to Averaged nMDCG:
    * for $k = 1$: all types of stemmer "NullTermProcessor", "English-Stemmer" and "EnglishStemmerStopWords" with scorer "BM25" (≈ 0.3018);
    * for $k = 3$: the best is "EnglishStemmer - BM25Scorer" (≈ 0.3586), followed by the pair "EnglishStemmerStopWords - BM25Scorer" (≈ 0.3401);
    * for $k = 5$: the best is "EnglishStemmerStopWords - BM25Scorer" (≈ 0.3473), followed by the pair "EnglishStemmer - BM25Scorer" (≈ 0.3456);
    * for $k = 10$: the best is "EnglishStemmerStopWords - BM25Scorer" (≈ 0.3795), followed by the pair "EnglishStemmer - BM25Scorer" (≈ 0.3733);
- For the Time Collection:
  - According to Averaged R-Precision: "EnglishStemmer - BM25Scorer" (≈ 0.5512);
  - According to Averaged nMDCG:
    * for $k = 1$: the best is "EnglishStemmer - BM25Scorer" (≈ 0.6144), followed by the pair "EnglishStemmerStopWords - BM25Scorer" (≈ 0.6144);
    * for $k = 3$: the best score is reached by "EnglishStemmer - BM25Scorer" (≈ 0.6257), followed by "EnglishStemmerStopWords - BM25Scorer" (≈ 0.6244);
    * for $k = 5$: the best is "EnglishStemmerStopWords - BM25Scorer" (≈ 0.6481), followed by the pair "EnglishStemmer - BM25Scorer" (≈ 0.6398);

* for $k = 10$: the best is "EnglishStemmerStopWords - BM25Scorer" (≈ 0.6717), followed by the pair "EnglishStemmer - BM25Scorer" (≈ 0.6648);

Considering the above data, the best configuration seems to be either "EnglishStemmer" with BM25 Scorer or "EnglishStemmerStopWords" with BM25 Scorer: they perform almost in the same way as it can be seen in Figure 7.

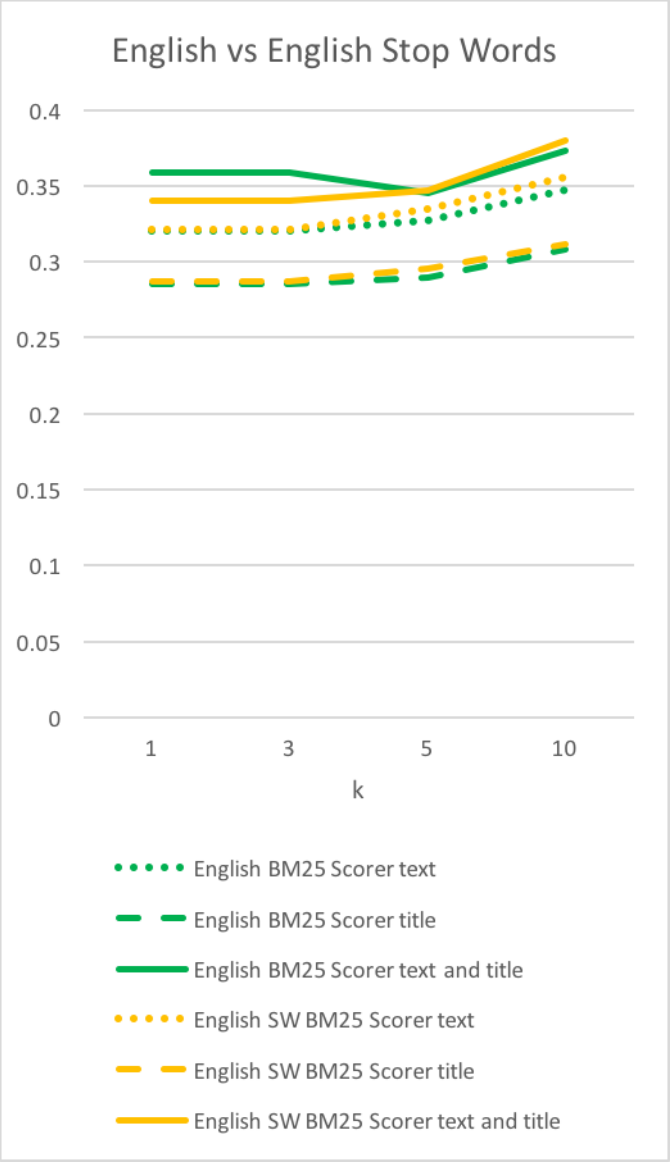

Figure 7: Comparison between English Stemmer and English Stemmer Stop Words both with BM25 Scorer.

5.2 Best stemmer

In order to answer to this question, in the following will be reported some useful results computed from the output of the software in `out/${collection_name}/evaluation/benchmark`, both for the Cranfield and the Time collections. In particular, in order to find

the best stemmer with our results, we aggregated the scores by stemmer, averaging over all the scorers.

Cranfield Collection

```
Evaluating averages for Averaged R-Precision.out
default:    0.153352
english:    0.160572
englishsw:  0.206245

Evaluating averages for Averaged nMDCG with k=1
default:    0.190691
english:    0.201201
englishsw:  0.264264

Evaluating averages for Averaged nMDCG with k=3
default:    0.191278
english:    0.211199
englishsw:  0.260231

Evaluating averages for Averaged nMDCG with k=5
default:    0.200523
english:    0.211233
englishsw:  0.268256

Evaluating averages for Averaged nMDCG with k=10
default:    0.215861
english:    0.231878
englishsw:  0.291918
```

Time Collection

```
Evaluating averages for Averaged R-Precision
default:    0.000803
english:    0.343889
englishsw:  0.405467

Evaluating averages for Averaged nMDCG with k=1
default:    0.000000
english:    0.389558
englishsw:  0.457831

Evaluating averages for Averaged nMDCG with k=3
default:    0.000000
english:    0.400708
englishsw:  0.484140
```

```
Evaluating averages for Averaged nMDCG with k=5
default:    0.000486
english:    0.417359
englishsw:  0.522976

Evaluating averages for Averaged nMDCG with k=10
default:    0.000486
english:    0.432929
englishsw:  0.549836
```

Summary

As we can easily check, in every kind of metric, the best stemmer, on average, is "EnglishStemmerStopWords".

5.3 Best scorer function

Doing the same for the scorers (i.e.: aggregating the scores by scorer, averaging over all the stemmers):

Cranfield Collection

```
Evaluating averages for Averaged R-Precision
CountScorer:  0.072379
TfIdfScorer:  0.186521
BM25Scorer:   0.261270

Evaluating averages for Averaged nMDCG with k=1
CountScorer:  0.106607
TfIdfScorer:  0.247748
BM25Scorer:   0.301802

Evaluating averages for Averaged nMDCG with k=3
CountScorer:  0.094648
TfIdfScorer:  0.230244
BM25Scorer:   0.337816

Evaluating averages for Averaged nMDCG with k=5
CountScorer:  0.097373
TfIdfScorer:  0.242073
BM25Scorer:   0.340566

Evaluating averages for Averaged nMDCG with k=10
CountScorer:  0.107233
TfIdfScorer:  0.266571
BM25Scorer:   0.365854
```

Time Collection

```
Evaluating averages for averaged_r_precision.out
CountScorer:  0.092501
TfIdfScorer:  0.294431
BM25Scorer:   0.363228

Evaluating averages for averaged_nMDCG_1.out
CountScorer:  0.136546
TfIdfScorer:  0.309237
BM25Scorer:   0.401606

Evaluating averages for averaged_nMDCG_3.out
CountScorer:  0.122889
TfIdfScorer:  0.345231
BM25Scorer:   0.416728

Evaluating averages for averaged_nMDCG_5.out
CountScorer:  0.141518
TfIdfScorer:  0.370019
BM25Scorer:   0.429284

Evaluating averages for averaged_nMDCG_10.out
CountScorer:  0.157074
TfIdfScorer:  0.380682
BM25Scorer:   0.445494
```

Summary

The best scorer function is BM25, since in every metric, the aggregated score for BM25 is greater than the other scores. It is an expected result, since it is the state-of-the-art TF-IDF-like retrieval functions used in document retrieval[3].

[3] https://en.wikipedia.org/wiki/Okapi_BM25