# A Simple Search Engine

Benjamin Roth

CIS LMU

# Document Collection for Search Engine

- Now that we have a documents, let's represent a collection of documents for search.
- What does a such a class for representing a document collection need?
  - Information to store?
  - Functionality?

# Document Collection for Search Engine

What does a class need for representing a document collection for search?

- Information to store:
  - Store the **documents**, and access them via an **id**.
  - An **inverted index**: A map from each term to all documents containing that term. (For efficiently finding all potentially relevant documents)
  - The **document frequency** for each terms (number of documents in which it occurs), to be used in similarity computation.
- Functionality:
  - **Read documents** (from directory)
  - **Return (all) documents** that contain (all) terms of a **query**.
  - **Reweight token frequencies** by tf-idf weighting.
  - Compute **cosine-similarity** for two documents.

# Document Collection (Code Skeleton)

```
class DocumentCollection:
    def __init__(self, term_to_df, term_to_docids, \
        docid_to_doc):
        #...
    @classmethod
    def from_dir(cls, root_dir, file_suffix):
        #...
    @classmethod
    def from_document_list(cls, docs):
        #...
    def docs_with_all_tokens(self, tokens):
        #...
    def tfidf(self, counts):
        #...
    def cosine_similarity(self, docA, docB):
        #...
```

# Detail: Constructor

- Set all the required data fields

```
def __init__(self, term_to_df, term_to_docids, docid_to_doc):
    # string to int
    self.term_to_df = term_to_df
    # string to set of string
    self.term_to_docids = term_to_docids
    # string to TextDocument
    self.docid_to_doc = docid_to_doc
```

# Detail: Get all documents containing all search terms

```
def docs_with_all_tokens(self, tokens):
    docids_for_each_token = [self.term_to_docids[token] \
        for token in tokens]
    docids = set.intersection(*docids_for_each_token)
    return [self.docid_to_doc[id] for id in docids]
```

- What does `docids_for_each_token` contain?
- What is contained in `docids`?
- How can we get all documents that contain **any** of the search terms?
- Bonus: What could be (*roughly*) the time complexity of `set.intersection(...)`?

# Detail: Get all documents containing all search terms

- What does `docids_for_each_token` contain?
  *List of set of document ids. (For each search term one set)*
- What is contained in `docids`?
  *The intersection of the above sets. The ids of those documents that contain all terms.*
- How can we get all documents that contain **any** of the search terms?
  *Use set union instead of intersection.*
- Bonus: What could be (*roughly*) the time complexity of `set.intersection(...)`? *A simple algorithm would be:*
  - *For each document id in any of the sets check wether it is contained in all of the other sets.*
  - *If yes, add to result set.*
  - *You can assume that checking set inclusion, and adding to a set takes constant time.*
  - *Complexity: O(nm), where n is number of search terms, m is number of document ids in all sets.*
  - *A more efficient algorithm would use sorted lists of document ids (posting lists).*

# Detail: Tf.Idf Weighting

```
def tfidf(self, counts):
    N = len(self.docid_to_doc)
    return {tok: tf * math.log(N/self.term_to_df[tok]) for \
        tok,tf in counts.items() if tok in self.term_to_df}
```

- Input (dictionary): term $\Rightarrow$ counts of term in document
- Output (dictionary): term $\Rightarrow$ weighted counts
- Remember formulas:
  - Term frequency is just the number of occurrences of the term (we use the simple, unnormalized version).
  - Inverse document frequency:

$$log \frac{N}{df_t}$$

where N is the size of the document collection and $df_t$ is the number of documents term $t$ occurrs in.

# Detail: Cosine Similarity

```
def cosine_similarity(self, docA, docB):
    weightedA = self.tfidf(docA.token_counts)
    weightedB = self.tfidf(docB.token_counts)
    dotAB = dot(weightedA, weightedB)
    normA = math.sqrt(dot(weightedA, weightedA))
    normB = math.sqrt(dot(weightedB, weightedB))
    if normA == 0 or normB == 0:
        return 0.
    else:
        return dotAB / (normA * normB)
```

- Input (dictionaries): term frequencies of two documents.
- Output: Cosine similarity of tf.idf weighted document vectors.
- How would `dot` helper function look like?
- What is the meaning of `normA` and `normB`?
- When can `normA` or `normB` be zero?

# Detail: Cosine Similarity

- How would `dot` helper function look like?

```
def dot(dictA, dictB):
    return sum([dictA.get(tok) * dictB.get(tok,0) for \
        tok in dictA])
```

- What is the meaning of `normA` and `normB`?
  *Vector norm (l2). It is defined as the square root of the dot product of a vector with itself:*

$$|v|_2 = \sqrt{\sum_i v_i^2}$$

  *Intuitively it measures the "length" of a document, and is high if a document contains many terms.*

- When can `normA` or `normB` be zero? *When a query only contains out-of-vocabulary words (`tfidf(...)` filters those words out).*

# Putting it all together: Search Engine

- Most of the functionality is already contained in the `DocumentCollection` class.
- The search engine only has to
  - Preprocess (tokenize) the query.
  - Call the respective methods (e.g. `docs_with_all_tokens`, `cosine_similarity`)
  - Sort the results to put most similar results first.
  - Select some text snippets for displaying to the user.

# Search Engine: Code Skeleton

```
class SearchEngine:
    def __init__(self, doc_collection):
        #...
    def ranked_documents(self, query):
        #...
    def snippets(self, query, document, window=50):
        #...
```

- See full implementation in the lecture repository.

# Summary

- Representing
  - Text documents
  - Document collections
- Factory method constructors
- Retrieving documents
- Computing similarity
- ... Questions?