Contents

[Algorithm 1](#_Toc54734352)

[Preprocessing 1](#_Toc54734353)

[Vectorization 2](#_Toc54734354)

[Matching Score (Cosine Similarity) 3](#_Toc54734355)

[Architecture 4](#_Toc54734356)

[Overview 5](#_Toc54734357)

[Optimizations 6](#_Toc54734358)

# Algorithm

I chose simple algorithm for computing text similarity **TF-IDF** (“Term Frequency — Inverse Document Frequency”).

Let me explain why.

There was not any additional information about which kind of articles we want to check, I had idea to use Machine Learning, but it does not make sense when you do not know the nature of data, which we will have in system. (because we need some dataset for training)

TF-IDF gives nearly same accuracy on generalization as Machine Learning (but ML could give better accuracy/results on specific data, where we can learn similarities/differences from data context). So **TF-IDF** is good point for start.

## Preprocessing

Data pre-processing is very important here. Since we did not know which data will be used in this service, I tried to implement some basic preprocessing steps.

First, each texts are transformed to list of words, and then each word is going through set of transformations.

Text preprocessing contained in TextPreprocessor class

1. *Lowercase* – transforming each word to lowercase
2. *Stop Words* – Stop words are the most commonly occurring words which don’t give any additional value for determining text similarity
3. *Punctuation* - Punctuation are just unnecessary symbols, which could appear in text.
4. *Apostrophe* – removing apostrophe, so “wan’t” will be same as “want”
5. *Converting Numbers* – transforming numbers to their string interpretation. So “100 dollars” will be same as “one hundred dollars”
6. *Lemmatization* – skipped
7. *Stemming* – process of reducing inflected (or sometimes derived) words to their word stem, so words “doing” and “do” will be considered as same

For Stemming process, I implemented porter2 stemming algorithm, which is well known and was introduced long time ago. Again, we don’t know the nature of data for which we are implementing this task, so I guess stemming will be enough for data preprocessing, so I skipped Lemmatization.

Example of data preprocessing:

So the following text “*I wan't to check TeXt 100 times*” will be tokenized to list ["want", "to", "check", "text", "one", "hundr", "time"]

* “*I*” symbol was removed, because it’s stop word
* “*wan’t*” apostrophe was removed
* “100” was converted to words and stemmed “*hundred*” to “*hundr*”

## Vectorization

The main goal of this step is to convert document (list of words) to vector (list of integers), so then we can perform math on it. **TF-IDF** is helping to do this.

Term frequency measures the frequency of word in document, but it really depends on how large document is, therefore we are normalizing each term frequency by dividing it to total number of terms.

Where – term, – document, – number of times when term occurs in document . - total number of terms in document;

Example:

Let’s consider we want to find duplicates for NewArticle: “dogs like people”. And there are following articles in database:

* Article1: “dog can bite your cat”
* Article2: “dogs likes people more than other animals”

So, for TF we will have following table

|  |  |  |
| --- | --- | --- |
| **Term** | **Frequency** | **Frequency normalized** |
| dog | 1 | 0.33 |
| like | 1 | 0.33 |
| people | 1 | 0.33 |

Therefore, Term frequency vector will look like this:

Next step is to compute inverse document frequency (idf), it measures how important word is.

Certain terms, such as “is”, “of”, and “that”, may appear a lot of times but have little importance, therefore we need to weigh down the frequent terms while scaling up the rare ones.

Where - total number of documents, – number of documents where term in it.

As result, we will have following:

|  |  |  |
| --- | --- | --- |
| **Term** | **Num of docs where it appears** | **Idf(t)** |
| dog | 2 | log(2 / 2) = 0 |
| like | 1 | log(2 / 1) = 0.301 |
| people | 1 | log(2 / 1) = 0.301 |

Therefore, Inverse Document frequency vector will look like this:

Then, for computing TF-IDF we need to

Finally our tfidf vector is

Now we have numeric representation of text, and it could be used to find similar texts.

## Matching Score (Cosine Similarity)

Now, when we can vectorize any text to list of number, we can compute similarity between articles and any new text or article.

Moving back to our example, if we want to find duplicates of **NewArticle** we need to have vectors for ALL documents that we have in system.

Here is table of tfidf vectors (calculated by same steps), notice that “dog” word has zero value, because it appeared in ALL documents and it’s value lowered by inverse document frequency factor ☺

|  |  |  |  |
| --- | --- | --- | --- |
| **Term** | **NewArticle** | **Article1** | **Article2** |
| dog | 0 | 0 | 0 |
| like | 0.231 | 0 | 0.099 |
| people | 0.231 | 0 | 0.099 |

On this step I decided to use Cosine Similarity which measures the similarity between two vectors of an inner product space. The formula of it is well known:

And as result we will have that

* NewArticle is similar to Article1 for 0% (it’s obvious because Article1 tf-idf vector is all zeros)
* NewArticle is similar to Article2 for 100%

Note, there is **MatchingThreshold** parameter in appsettings.json, which controls which results should be considered as duplicates

**MatchingThreshold** = 0.75 means that Article1 will be duplicate of Article2 if their similarity result is >= 75%

# Architecture

The system performance has very big dependency on count of articles which we have in system.

It will be really slow to calculate duplicate articles on each GET request, therefore similarity results are stored in Redis cache for faster access.

There are several optimizations which were implemented in order to duplicate checking operations, so the main goal was to increase performance of reading results.

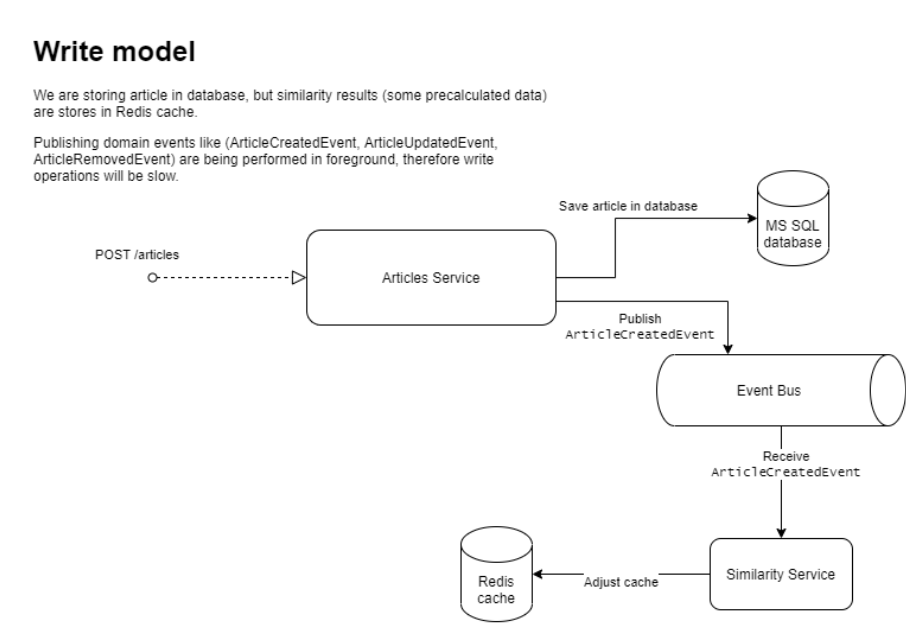
Write model is synchronous, so when adding new item to system, request will wait until all needed recalculations will be made, but we easily can change this to be background work (as result we will have eventually consistency, where request will be completed, but results will be appeared later)

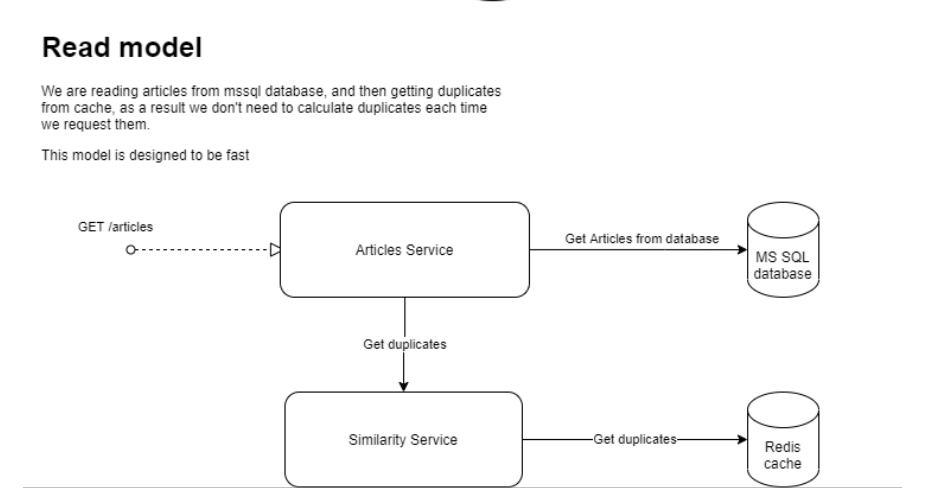
Also, there is initialization logic, which pre-calculates similarity results for articles that are in database.

Ideally, it should be separate service (another physical tier), but for simplicity I added initialization logic on API startup, therefore API will be started after all calculations are done.

Ideally, it should initialize similarity results only for articles which are not calculated yet, but for simplicity service is initializing ALL articles (overwriting previous calculations).

## Overview





## Optimizations

1) I decided to store article pre-processed tokens in database as well as content, so we will not need to perform it again and again

2) I am storing token frequencies in cache in following format, which allows having complexity on getting frequencies from cache

{

"tfIdf:token:day": {

"frequency": 5,

"documentIds": [ 1, 2, 3 ]

}

}

This also give us ability to see which documents will be affected if some of them are changed or removed, so we will recalculate tf-idf only for them (not for WHOLE database).

3) Tf-idf vectors for all documents and their tokens are stored in cache as well, and they are being recalculated for every changes in database (via event bus).

{

"tfIdf:document:1:day": 0.26,

"tfIdf:document:2:day": 0.13,

"tfIdf:document:3:day": 0.54

}

4) I’m storing similarity results for each article (it’s better to use bidirectional graph data structure for this, but I didn’t had time for it)

This cached data allow us to have similarity results for EACH article and return duplicate\_groups very fast ☺

Processed similarities are stores in cache in following structure

{

"articles:1": [ { id: 2, score: 0.8 }, { id: 3, score: 0.4 } ]

"articles:2": [ { id: 1, score: 0.8 }, { id: 2, score: 0.2 } ]

}