NLP Problem “Similarity Scores"

Problem:

You are provided with four documents, numbered 1 to 4, each with a single sentence of text. Determine the identifier of the document D which is the most similar to the first document, as computed according to the TF-IDF scores.
d1: I’d like an apple.
d2: An apple a day keeps the doctor away.
d3: Never compare an apple to an orange.
d4: I prefer scikit-learn to orange.

Theoretical solution (see source)

The TF (*term frequency*) score indicates the importance of a term (word, token, ...) in a given document (here computed as the normalized frequency):

$$\text{tf}(t, d) = \frac{freq_{t,d}}{|t' : t' \in d|}$$

The IDF (*inverse document frequency*) score indicates the importance of a term in a corpus (over all documents):

$$\text{idf}(t, D) = log\left(\frac{|D|}{|\{d : t \in d\}|}\right)$$

The TF-IDF score is computed as:

$$\text{tf-idf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D)$$

The most similar document to *d1* is the one having the maximum sum of TF-IDF scores:

$$\text{sim}(d1, di) = \sum_{t \in d1} \text{tf-idf}(t, di, D)$$

For the given example, the TF, IDF and TF-IDF scores should be computed with respect to the unique words belonging to the *d1* document ("I’d", "like", "an", "apple").

The document most similar to the first document is *d3*.
The output answer for this problem is therefore "3".

Note: it is not stated in the problem whether the terms “I’d" and “scikit-learn" should be separated into two words (i.e. “I", “’d", “scikit", “learn"). Even if that was the case, the result remains the same.

C++ Implementation:

In order to solve this problem, we must compute the TF and IDF scores of the unique words belonging to the first document *d1* with respect to all given documents. The document having the maximum sum of the resulting Tf-IDF scores is the one most similar to the first document.

Several steps are required:

- clean the text of documents: set all words to lowercase and remove punctuation marks (replaced with a white space)

```
string removePunctuation(string input);
```

  We can add the single quote (') and the hyphen (-) characters to the punctuation[] list (declared in the function *removePunctuation*) if we would like to separate words like "I'd" or "scikit-learn". We can also include other characters that might be required.

- get the list of unique words from each document, along with their occurrence frequencies (a map with *string* keys and *int* values)

```
map<string,int> getUniqueWords(string text);
```

- check the presence and number of occurrences of all reference words (from document *d1*) in all documents and compute their TF score

  → we loop through all documents
  → each document is represented by a map of unique words and their occurrences
  → for each document *Docs[i]* we create a map (with *string* keys and *double* values) that stores the TF scores of each reference word *rWord* (from document *d1*) in the current document (*Docs[i]*)

$$TF = (numberOfOccurrences(rWord, Docs[i]))/(numberOfWords(Docs[i]))$$

  → we finally return the array of TF scores of all documents

```
vector<map<string, double> > computeTF(map<string,int> refWords, string
    Docs[], int nDocs);
```

- get the number of documents that include each reference word;
  → the IDF score for a given reference word (from document *d1*) is computed as the logarithm of the ratio between the total number of documents divided by the number of documents including the current word

```
map<string, double> computeIDF(map<string,int> refWords, string Docs[],
    int nDocs);
```

- compute the TF-IDF(t,d,D) score as the product between TF(t,d) and IDF(t,D)

```
vector<map<string, double> > computeTFIDF(map<string,int> refWords,
    vector<map<string, double> > tf, map<string, double> idf);
```

- compute the sum of the TF-IDF(t,d,D) scores of all reference words (from document *d1*) for each document (other than *d1*). The document having the maximum sum is the one most similar to document 1.

```
int getMostSimilarDocument(map<string,int> refWords, vector<map<string,
    double> > tfidf);
```

Similar NLP problem solved in the past:

Problem: add new words into a n-gram language model using a word similarity approach.

Steps:

- use a few examples of sentences for each 'new' word
- use a large textual corpus for 'known' words
- find the similar known words (having similar neighbor distributions) for each new word
  - compute the neighbor distributions for each new word **nW** in each k position $P_k(w|\mathbf{nW}),\ k \in \{..., -2, -1, +1, +2, ...\}$
  - compute the neighbor distributions for each known word **kW** in each k position $P_k(w'|\mathbf{kW}),\ k \in \{..., -2, -1, +1, +2, ...\}$
  - compute the KL divergence between the neighbor distributions of each known word (**kW**) and each new word (**nW**)

    Divergence computed on each k position:

    $$D_{KL}\ (\ P_k(\bullet|\mathbf{kW})\ ||\ P_k(\bullet|\mathbf{nW})\ ) = \sum_{w \in N(\mathbf{nW})} P_k(w|\mathbf{kW})\ log\left(\frac{P_k(w|\mathbf{kW})}{P_k(w|\mathbf{nW})}\right)$$

    Global divergence:

    $$D(\mathbf{kW}, \mathbf{nW})\ =\ \sum_k D_k(\mathbf{kW}, \mathbf{nW})$$
  - select the mots similar words to the new word as those having minimal divergences
- transpose the LM probabilities of known words onto the new words
  - seek the n-grams that contain similar words
  - replace the 'similar word' with the 'new word'
  - add the new n-grams into the new language model