# Document Similarity Checker with Python

## In this article, we will build a system for calculating the similarity between different documents along with making it available as an API and web app.

**Text Similarity** is defined by how close two corpora of text are in comparison to each other. There are a number of ways that could be useful, such as:

1. Search engines use this functionality to show websites containing similar text as the query. Sites like Yahoo Answers and Quora use the same functionality to show related answers
2. Chatbots can use this to provide appropriate answers to the customers, thus enhancing the customer’s experience.
3. It can be used for plagiarism checking to ensure appropriate credits for a given piece of text

This article is divided into 5 sections:

* Cleaning the Data
* Cosine Similarity
* Building the API
* Building the UI
* Containerize the application

If you want to jump to the code, here’s a link to the code repository: [document\_similarity\_checker](https://github.com/fenago/blog/tree/main/Document%20Similarity%20Checker)

**Cleaning the Data**

Before we calculate the similarity between various corpora of text, we need to clean the text to remove the information that is not helpful in calculating the similarity.

**Stopwords**

There are a lot of words in a language that isn’t helpful in the semantic meaning of a sentence. These are known as **stopwords.** These include words such as I, me, myself, etc. To remove these, we will use Python’s library, “nltk” and download the stopwords.

A screen shot of a computer

Description automatically generated with low confidence

Here’s the output from that:

Text, letter

Description automatically generated

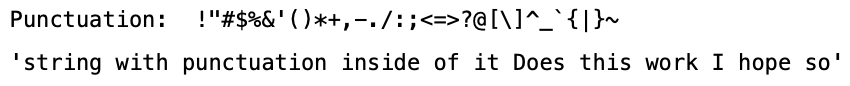
**Punctuation**

Punctuation isn’t really helpful in determining the similarity between two documents and thus we will remove that too using the following code:

Text

Description automatically generated

We create a dictionary that maps each character in punctuation to None, basically removing it. translate method helps us by automatically applying the map to each character in the string. Here’s the output:



**Tokenize**

To calculate similarity, we will tokenize our string which basically means converting it into a string containing the individual words

Text

Description automatically generated

The output:



**Stemming**

Lastly, we need to do stemming. It is basically converting each word into its base form.

Text

Description automatically generated

The output:



**Combining Everything**

Let’s combine everything we have done in this section into a single function

Text

Description automatically generated

**Cosine Similarity**

Let’s start by understanding how cosine similarity works and then implement that in python

**Cosine similarity** is a metric, helpful in determining, how similar the data objects are irrespective of their size. In cosine similarity, data objects in a dataset are treated as a vector. The formula to find the cosine similarity between two vectors, x and y , is:

cos(x, y) = ( x . y ) / (||x|| \* ||y||)

* **x . y** = product (dot) of the vectors ‘x’ and ‘y’.
* **||x||** and **||y||** = length of the two vectors ‘x’ and ‘y’.
* **||x|| \* ||y||** = cross product of the two vectors ‘x’ and ‘y’.

Let’s calculate this in python. Although it looks quite straightforward to implement, we have to take care of various considerations when dealing with text data such as:

* There would be words present in one piece of text and not in the other
* Both will not have the same number of words

For this purpose, we will be using sklearn library to calculate the vectors from the strings and then calculating cosine similarity on them.

Text

Description automatically generated

We need to join the tokens again because the CountVectorizer accepts sentences and outputs a vector equal to the length of unique words in the corpus for each sentence and assigns the count of that word in a particular sentence. Here’s the output from above:

A picture containing text

Description automatically generated

First, we get the cleaned sentences, then the unique words in the corpus, and finally a list of vectors for each sentence. Let’s now calculate cosine similarity:

Text

Description automatically generated

The cosine\_similarity function accepts the vectors in a certain format and thus we have to reshape them as such. If we look at the corpus, we can guess that the first and second sentences are more similar than the second and third. Let’s see what the cosine similarity gives us:

Text

Description automatically generated with medium confidence

It’s indeed confirmed by our program.

**Building the API**

There will be only one endpoint for calculating the similarity using the cosine method. Let’s first create separate files for the above modules. Here’s the name of all the files that we will be using:

1. utils.py (Cleaning Functions)
2. cosine\_similarity.py
3. api.py
4. index.py (for UI)
5. Dockerfile (Containerize the application)
6. templates/index.html (Contains the UI)
7. data/ (contains the text files)

Here’s the directory structure

Text

Description automatically generated

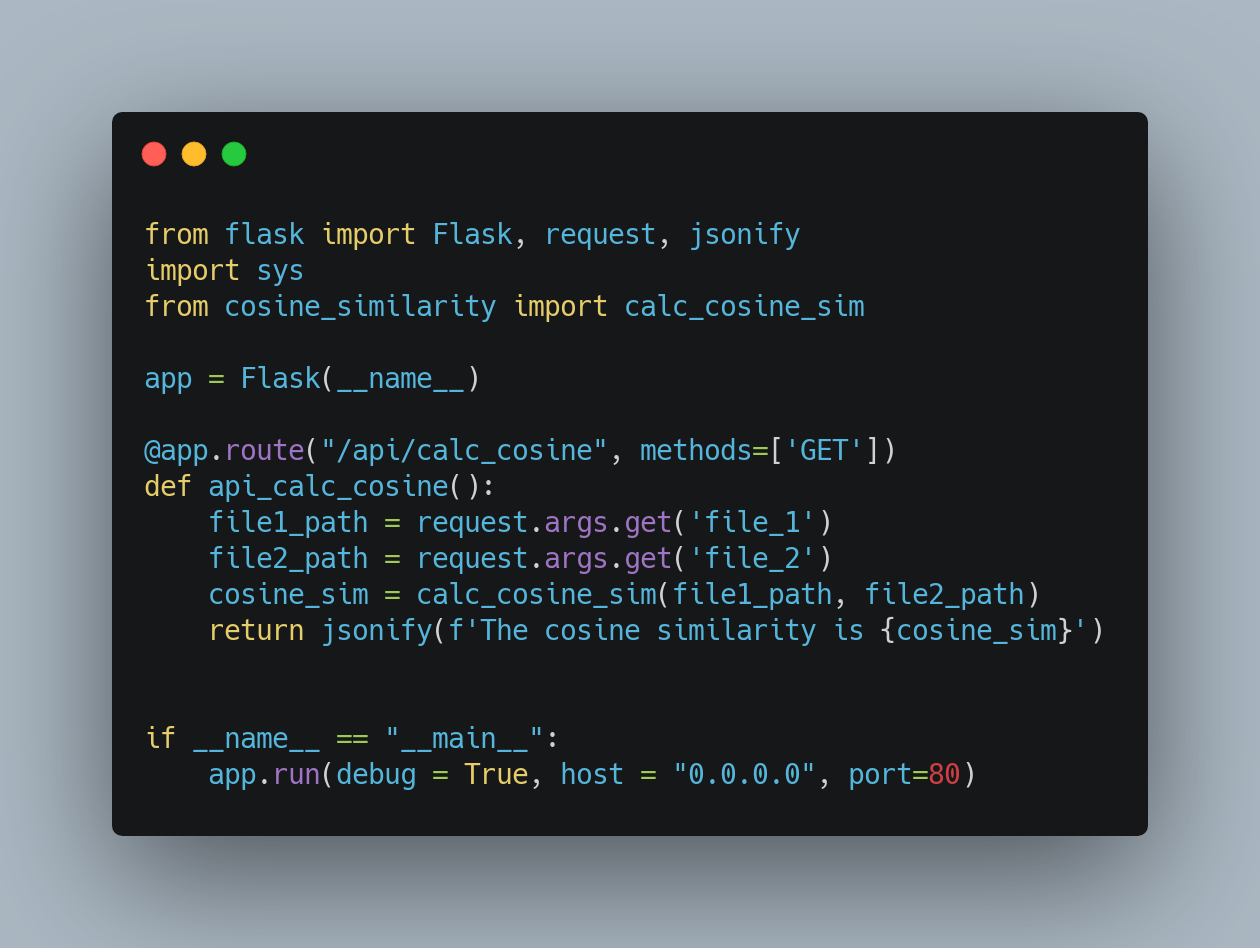
Let’s first group together the above code we wrote into multiple files:

Text

Description automatically generated**utils.py**Text

Description automatically generated**cosine\_similarity.py**

We will be using Flask for our purposes as it helps in building simple APIs really fast. I will introduce you to the whole code of API and then explain it.

**api.py**

Let’s dissect this file:

1. We start by importing the required libraries. These include Flask and functions from the modules above.
2. We define the app variable, which basically represents our web app
3. We then use this syntax, @app.route . This basically is a decorator. All you need to know is whenever we hit this route, the function below that is called.
4. We have only included the method GET as we are not passing any data from a form.
5. We define the route for calculating the similarity.
6. Notice the last line in each of the endpoints, jsonify . JSON is the standard data format in web applications and hence we change our output to JSON before sending.
7. Lastly, we run the app on port 80.

Let’s run this file using the command python api.py :

Text

Description automatically generated

Currently, there is no UI so we can’t view the app on our browser. Instead, we will use curl to access the API endpoints. We need to keep the above terminal running and open a new tab for this. We need to pass the path of both files using the query strings. Note that the names are exactly the same as what we use in the api.py , this is required for the API to function correctly

**Cosine Similarity API Endpoint**

**Building the UI**

As before, let’s first see the code:

Text

Description automatically generated**ui.py**

Let’s dissect the code:

1. We import and define the app as we did in the api.py file but here we also define the Upload folder for saving the files
2. Then we define the routes as before but here we are going to define two methods GET and POST as we are uploading the files from a form.
3. In case of GET , we just render the form.
4. For POST method, we read the files using the names defined in the form and save them in the folder.
5. Finally, calculate cosine similarity using the function we defined above and send it to the template.

Let’s now look into the index.html file. Note that the index.html file should be defined in the templates folder otherwise Flask won’t be able to check it.

Text

Description automatically generated**index.html**

Here, we check if we have the similarity dictionary and show the form or the similarity respectively.

We will then run the app using the following command:

Text

Description automatically generated

Below are the screenshots from the app:

Graphical user interface, application

Description automatically generated**Form Page**Graphical user interface, text

Description automatically generated**Output Page**

**Containerize the Application**

To make our app more useful and easily accessible to people, we would containerize it using Docker. We will create a Dockerfile in the same folder as the index.py. Let’s examine the Dockerfile

Text

Description automatically generatedDockerfile

The commands are executed one by one here.

1. First, we install python and create a new working directory and copy all the contents from the current directory to this one.
2. Then, we run the commands to upgrade pip and install all the necessary libraries for this.
3. Finally, we run the index.py file which runs the server of our app.

To make an image, first, go inside the folder where your Dockerfile exists, then run the following command:

Text

Description automatically generated

This will take some time to execute. After it’s done you can run your app as follows:

Text

Description automatically generated

-p tag defines the port that you want the application to run on. You can view the image running on the docker desktop and also view the application in the browser.

I have also pushed this docker image to docker hub so you can download and play around with this application here: [document\_similarity\_checker](https://hub.docker.com/r/fenago/document-similarity-checker)

**Conclusion**

In this article, we looked into using cosine similarity for calculating document similarity:

1. We started by cleaning the data using various methods such as tokenizing, stemming, removing stop words, etc.
2. We then explored cosine similarity theoretically and implemented it in Python using scikit-learn .
3. After that, we created an API and a UI for our application using Flask
4. Finally, we built a docker image so that anyone can use this application.