Report On

**ARTIFICIAL INTELLIGENCE PROJECT**

**(RECOMMENDATION SYSTEM FOR ACADEMIC RESEARCH)**

**By**

K. Harsha Vardhan, CS21B2036

T. Lakshmi Srinivas, CS21B2045

In the previous report we have seen the problem statement ***“A recommendation system for academic research”?*** Now in this report we will go through the solution of the problem, what are the approaches and tools we used, hurdles we faced and usage of AI in this solution.

**The Initial Setup:**

The first part of any python project is to find the necessary libraries / APIs that are useful and can make our work easier in the code. The ***start of the code*** is importing all these libraries and APIs.

**The hurdles we faced:**

The most important hurdle we had to overcome was the necessity for the recommendation machine’s ability to go through different types of files and compare them to get a better result. Consider, I have to check if an excel spreadsheet has information which is similar or is connected to the information within a PowerPoint and academic PDF journal. It can be solved by reading every file type (i.e., all kinds of file formats) into Python and then we have to transform each object into a single string of words. This *normalizes*all the data and allows for the calculation of a similarity metric.

**First step in process:**

To achieve this task we need to create different *reading classes* for each kind of file format to extract the contents from a given respective file format whether it may be a .doc, .xls, .ppt, .pdf, etc. The ***second step*** of our solution is to create some of the reading classes. Here is the list of reading classes we chose to define in our code:

* *The first class we have here is the****pdfReader****class which is able to format a PDF to be readable in Python. Of all the file formats, one can say that PDFs are one of the most important since many of the journal articles downloaded from research repositories such as Google Scholar are in PDF format.*
* *The****pptReader****class is capable of reading Microsoft Powerpoint files into Python.* *Not many people think about how there is valuable information stored within the bodies of PowerPoint presentations. These presentations are large and created to visualize key ideas and information to the audience.*
* *The****wordDocReader****class can be used for reading Microsoft Word Documents in Python. It utilizes the* ***doc2txt API*** *and returns a string of the text/information located within a given word document.*
* *In some cases, researchers will include excel sheets of their results with their publications. Being able to read the column names, and even the values, could help with recommending results that are like what you are searching for. For example, what if you were researching information on the past performance of a certain stock? Maybe you search for the name and symbol which is annotated in a historical performance excel sheet.*
* *The****csvReader****class will allow for CSV files to be included in your database and to be used in the system’s recommendations.*

This is it for the classes that we used in this project.

***Note*:** There are tons of other file types you can use to enhance your recommendation system. You can also try for images, videos, etc.

**Second step in process:**

**Data Processing:**

Now we will go through the way how we can pre-process this data. This is a recommendation system that we built for a repository of academic research. Hence, we have to break the text down using the pre-processing steps which are followed-up by *Natural Language Processing (NLP)* was important.

We can simply define this class as ***datapreprocessor***and the *first function* within the class is a *word parts of speech tagger*. This function will tag the parts of speech in a word which will be helpful later in the project.

*Second*, we will write a function that conducts the ***normal NLP steps*** which may have already seen before. These steps are:

1. Convert each word into small letters
2. Remove the punctuation
3. Remove digits (*I only wanted to look at non-numeric information. This step could be taken out if desired*)
4. Stopword removal.
5. Lemmanitizaion. **This is where the get\_wordnet\_pos() function comes in handy for including parts of speech!**

***Stopwords*** are a set of commonly used words in any language. In case of “English”language we have *“the”, “is”, “and”, etc.*

***Lemmatization***is the process of grouping together different inflected forms of the same word. It's used in computational linguistics, natural language processing and chatbots. Itlinks similar meaning words as one word.

The next step, we are going to write a function that can ***read all of the files*** into the system.

The next following function is called the ***database\_preprocess()***which can process all the files within your given database. The argument here is a list of the files, each will have its associated string of text (processed already).

The strings of text are then *vectorized*using a special in-built function known as **sklearn’s tfidVectorizer.**What will it do? Basically, it will transform all the text into different feature vectors based on the frequency of each given word. We can go through this in detail in the code documentation. This is done because we can look at how closely documents are related using *similarity formulas* relating to *vector arithmetic*.

**The drawback we faced:**

The reason we created a vectorizer off of the database is that when a user enters a list of terms to search for in the database, those words will be vectorized based on their frequency in said database. This is the *biggest weakness* of the current system. The more we increase the size of the database, the more the time and computational allocation are needed for calculating similarities which will slow down the system.

On reading many articles and browsing deeply through internet we learned a conclusion from the sources that we can use ***Reinforcement Learning*** *for recommending different articles of data.*

Next, we will implement a function named ***input processor*** that can process any word given into a vector. This is similar to that of typing a request into a search engine. Since all of the information within and given to the database will be vectors, we will use ***cosine similarity***to compute the angle between the vectors. The closer the angle is to *00*, the *less similar* the two said vectors will be.

Once we find the similarity score between two vectors, we can create rankings between the words being searched and the documents located within the database.

Here the input *vector file* list is a list of vectors that we created from the files before. The *query vector* is a vector of the words being searched. The *file dictionary* was created earlier which uses file names for the keys and the file’s text as values. Similarities are computed, and then a ranking is created favoring the most similar pieces of information to the queried words being recommended first.

***Note:*** What if the system recommends more than 3 files? By implementing elements of ***Networks and Graph Theory***in our code we will add an extra level of computational benefit to this system and create more confident recommendations.

As another evaluation method for recommendation, we will use one variation of **PageRank** (Theory is documented in the code) called ***eigenvector centrality***. *Eigenvector centrality*is like PageRank in that it measures the connections between nodes of a graph, assigning higher scores for stronger connections. The major difference between them?

“*Eigenvector centrality will account for the importance of nodes connected to a given node to estimate how important that node is*.” This is synonymous with saying, a person who knows lots of important people may be very important themselves through these strong relationships. All-in-all, these two algorithms are very close in the way they are implemented.

For this database, after the vectors are computed, we can place them into a graph where their edge distance is determined by their similarity to other vectors.

What’s the next step? We obtained the recommendations that are created by using the ***cosine similarity*** between each data point in the database, and recommendations computed by the ***eigenvector centrality algorithm***. Which recommendations should we output? **Both!**

The *final function* of this script will weigh the different recommendations produced by ***cosine similarity*** and ***eigenvector centrality***. Currently, **80%** of the weight will be given to the recommendations produced by the *cosine similarity recommendations*, and **20%** of the weight will be given to *eigenvector centrality recommendations*. The final recommendations can be computed based on these weights and aggregated together to produce recommendations that are representative of all the similarity computations in the system. The weights can easily be changed by the developer to reflect which batch of recommendations they feel are more important.

**Conclusion:**

In this project we created a recommendation system for files you collected (especially if you are collecting research for a project). The main feature of this system is that it goes one step further in computing the cosine similarity of vectors by adopting the eigenvector centrality algorithm for more concise, and better recommendations.