CP 423 Text Retrieval & Search Engine – Final Project

Herteg Kohar (190605160), Kelvin Kellner (190668940)

What each member contributed:

- Herteg Kohar (190605160) - Crawling, Indexing, Querying, Training Classifier, Predict Link
- Kelvin Kellner (190668940) - Soundex, Spell Correct, Querying, Story

Crawling:

How we understand the question:

The program collects new documents for each of our topics in our search engine. We use the source.txt file to begin crawling the links for new pages on the website as our trusted source. This is done for the TOPIC_DOCUMENT_LIMIT constant in constants.py. The collected documents are then placed in the data directory in the appropriate subdirectory according to the topic. When a document is collected, we use its URL to generate a hash, which is then used as the filename for the text file containing the document. The hash is then put into another file (hash_to_url.json) which maps the hashes to the URL to be displayed to the user while querying. The crawler works by iterating over all the sources to collect documents from each of the trusted sources, with the aim of achieving a nicely distributed collection of documents.

How we implemented the question:

The crawler uses the sources from source.txt to get reputable documents for each of our topics, which are Astronomy, Economy, and Health. We use a circular queue to rotate the sources for each of the topics to diversify our documents. Each source has its network location as a key in the dictionary. Each crawl extracts all the links from the current page and makes sure it is from our source. If the link is from one of our sources, we then include it in the dictionary where the source's network location is the key and there is a list of URLs to crawl. The text of the page is then collected, and the URL is hashed and made sure that it does not already exist in our collection. If the document is new the text content is saved to a text file with the URL hash as the name. The stopwords are removed before saving as well. Each crawl results in a rotation of the circular queue in order to use the next source. This is done in order to try and keep our documents diverse and not entirely from one source. The hashes URL is also placed in the hash_to_url.json file to display the URL for the user when they query and documents are recommended. This process continues for each of the topics until the TOPIC_DOCUMENT_LIMIT is reached for each topic. To identify the network location and join endpoints to network locations to crawl we used urllib.

Any challenges/issues that we had:

Sites that did not provide any text data were a problem to begin with. After checking to see whether they had text data or not we discarded the ones that use JavaScript to load their site. We tried using a method that could have a timeout to wait for the site to load in cases where JavaScript was mainly being used to load the site. However, we did not use this as implementation would be too time-consuming.

Indexing:

How we understand the question:

The program iterates over the `data` directory and all the topic subdirectories, and after tokenizing, indexes all of the tokens. Along with each token, a list of occurrences is stored, which includes the document hash, the frequency of the token in the document, and the topic of the document. Since the actual hashes are very long a short form is used for each hash which can be found in `mapping.json` located in the `Index Data` directory. The program then saves the inverted index (`inverted_index.json`) and the mapping to the `Index Data` directory. The outline of the occurrences field in the inverted index includes all occurrences of the token within documents, as well as the frequency of the token in the document. The soundex code for each token is also stored in the inverted index, which can be used for spell correction during the querying process. The `hash_to_url.json` file is then checked to see if all the hashes are in the mapping to ensure all files have a corresponding link as well.

How we implement the question:

The first step is to iterate over all of our documents. Then each document's text content is tokenized. Each token is then a key in the inverted index which includes a "soundex" field which stores the token's Soundex code. Another field is the "occurrences" field which houses the number of occurrences as well as the hash to identify which document contained the token as well as the topic. After this we use a short form of the hash to keep the inverted index as readable as possible in the form of $H_1 \ldots H_N$. A map is also saved for this to retrieve the original hash. When the inverted index is being updated, we check to see if the hash is in the mapping before adding it into the inverted index to avoid unnecessary computation.

Any challenges/issues that we had:

Deciding what format to make the inverted index was a task at first. We decided to stick with Json for readability since Python has its built-in json module which makes it much easier to load the inverted index as a Python dictionary when it needs to be used later on for our querying.

<u>Querying:</u>

How we understand the question:

The program prompts the user to enter a search query. The users query will be spell corrected using soundex similarity, edit distance and term frequency are used for fallback conditions. The program will then perform the term-at-a-time algorithm and use the inverted index to display the top 3 highest ranked documents (ranked using cosine similarity) that contain any of the spell corrected query terms. A link to the web page is given for each document, as well as a snippet of matching text blocks with query terms highlighted in unique colours.

How we implement the question:

First, we perform preprocessing on the user's query. We tokenize the query and remove all stop words. Next, the inverted index and mapping files are loaded by the program. Next, the inverted index is used to spell-correct each token in the query. Any words in the query not found in the inverted index are replaced with the closest match or removed from the query if no "close" matches are found. The algorithm chooses the closest match by generating a list of inverted index words with the same soundex code as the target word and using the highest term frequency inverted index word 1 edit distance away,

or 2 edit distance away if there are none, as fallback conditions if multiple comparable matches are found (multiple words with the same soundex code as the target word). After the spell corrected query is generated, all documents that contain any of the query words are retrieved, TF-IDF is used to vectorize the query and documents, and then cosine similarity is computed for all documents. The top 3 most similar documents are chosen, and all sentences from the text containing any of the query words are displayed, with each query word highlighted in a unique colour thanks to the colorama python package.

Any challenges/issues that we had:

For our first attempt at this question, we relied on the textbook section on “term-at-a-time" and tried to implement the provided pseudocode in our own program. We, however, found the pseudocode to be inadequate. We decided to scrap our code and restart, simply following the workflow described in the project outline at face value, rather than overcomplicating it. Things worked much better the second time around as we were able to focus our attention on the key details without getting bogged down.

Training Classifier:

How we understand the question:

We have chosen the KNN model for training as we found it to be the most effective in predicting and classifying new links into their respective topics. We based this decision on our evaluation metrics, including cross-validation, classification metrics, and by using our model outside of training. See graphs and data in the appendix for a better understanding of our reasoning. The program begins by collecting all the documents and placing them into a Pandas dataframe, which includes their text content. The text content is then preprocessed through tokenization and removal of stopwords. Afterward, the data is split into training and testing sets, with 80% used for training and 20% for testing. Next, the TFIDF vectorizer is fitted to the training data and used to vectorize both the training and test sets. The vectorizer is saved for future use in predictions. The KNN model is trained using a grid search to identify the best parameters, in this case the neighborhood size. Once this is done, the model is tested on the test set to make predictions and calculate the relevant metrics.

How we implement the question:

All the documents in the data directory are iterated over and all their text content is then collected and put into a data frame. The text is then preprocessed for each document by tokenizing. The text is then vectorized by using a TFIDF vectorizer fitted on the training data. The vectorizer is saved to be able to make predictions on new text in the future. The dataframe is then broken up into 80% for training and 20% for testing. Before choosing our model, we had set up a cross-validation pipeline to try and identify the best model based on weighted f1 score, precision, recall and accuracy. We also tested our models on our test set as well. KNN turned out to have the best performance every time we crawled for new documents and refitted our models. We decided to use KNN as our final model. So, we then turned to using grid search and cross-validation to find the best neighborhood size for our model. After this the model is then tested on the test set and the metrics are displayed.

Any challenges/issues that we had:

The issue we ran into was a lack of documents at the time of trying to implement our training process as our training and test sets were not very large. This was easily overcome as we continued to crawl our sources for documents and our classifiers continued to improve as our training and test size increased.

Predict Link:

How we understand the question:

"The program prompts the user to input a link, which is then crawled to extract its textual content. The content is then vectorized using the same vectorizer that was utilized during classifier training. The program then loads the saved classifier and uses it to make a classification prediction, printing each of the three program topics along with a corresponding probability/confidence score indicating the likelihood of the link belonging to that topic."

How we implement the question:

First, the program crawls the given URL and proceeds if the link is valid or displays an error message otherwise. Next, the TF-IDF vectorizer and classifier model are loaded from the "4 - Train ML Classifier" option. The program tokenizes the text contents of the link, removes any stop words, and vectorizes it. The model takes the vectorized document and calculates the prediction probabilities for all topics. All topics are printed to the console along with their appropriate probability percentages.

Any challenges/issues that we had:

An issue that occurred was the fact that some sites loaded their content through JavaScript leaving us unable to collect any meaningful textual data from the site to make a classification. In this case, if there is no textual data to be collected the user is then shown a message saying, "Link is not valid, or no text was able to be extracted". This left us able to counteract instances in which no data was available to make a classification as well since we used a try and except block to surround the HTTP GET request for the link.

Our Story:

This search engine program is designed to provide users with the basic functionality for collecting, indexing, classifying, and searching for documents while leveraging our knowledge of search engine architecture and information retrieval gained from this course. We applied course concepts including web and document crawling, text processing, indexing, query processing, classification and retrieval models to develop its various components.

For instance, to collect new documents, we implemented a web crawler algorithm that uses BeautifulSoup to parse text, store documents and extract links from web pages. To index documents, we used an inverted index data structure that stores the frequency of each term in the document to enable fast retrieval of search results. Additionally, we used the term-at-a-time algorithm to retrieve documents related to the query, and applied text processing techniques such as tokenization, stemming, stop-word removal, and spell correction to improve the accuracy of search engine results.

We applied our knowledge of supervised machine learning methods to train a classifier model capable of classifying documents into one of the three program topics. To train the model, we utilized a K-Nearest Neighbors (KNN) classifier based on the text content of both labeled source pages and internal

links found within them. This classifier model is used for the link prediction feature to predict which of the three topics a given link belongs to.

Overall, this simple search engine program is the culmination of our in-depth understanding of search engine architecture and information retrieval techniques that we gained throughout this course. Our ability to effectively implement various algorithms and processes is reflected in the program's performance and utility.

Thank you for taking the time to explore our program,

- Herteg and Kelvin <3

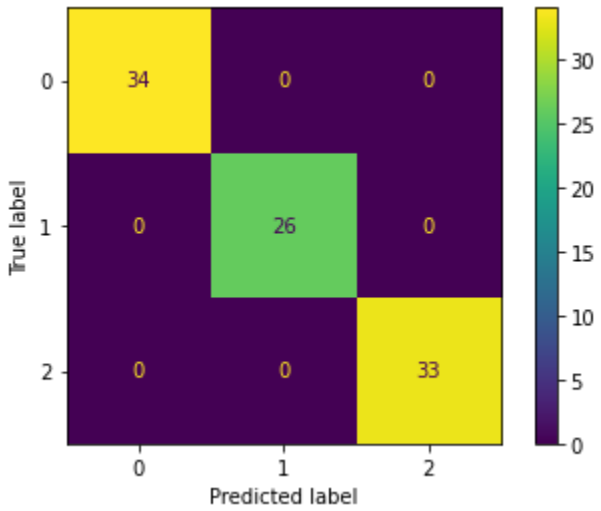
Appendix – Model Comparison

```
Model: SVC
Fold: 1
Accuracy: 0.9594594594594594
Precision: 0.9619753086419753
Recall: 0.9588888888888888
F1 Score: 0.9593534002229654


Fold: 2
Accuracy: 0.9864864864864865
Precision: 0.9871794871794872
Recall: 0.9861111111111112
F1 Score: 0.9863718537060214


Fold: 3
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0


Fold: 4
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0


Fold: 5
Accuracy: 0.9726027397260274
Precision: 0.9727777777777779
```

Recall: 0.9727777777777779
F1 Score: 0.9727777777777779

Mean Metrics
Mean Accuracy: 0.9837097371343948
Mean Precision: 0.9843865147198481
Mean Recall: 0.9835555555555555
Mean F1 Score: 0.9837006063413529
Best Estimator: 2
[0 1 2]
Test Metrics

	precision	recall	f1-score	support
0	1.00	1.00	1.00	34
1	1.00	1.00	1.00	26
2	1.00	1.00	1.00	33
accuracy			1.00	93
macro avg	1.00	1.00	1.00	93
weighted avg	1.00	1.00	1.00	93



Model: KNeighborsClassifier
Fold: 1
Accuracy: 0.972972972972973
Precision: 0.9733333333333333
Recall: 0.9733333333333333
F1 Score: 0.9733333333333333

Fold: 2
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0

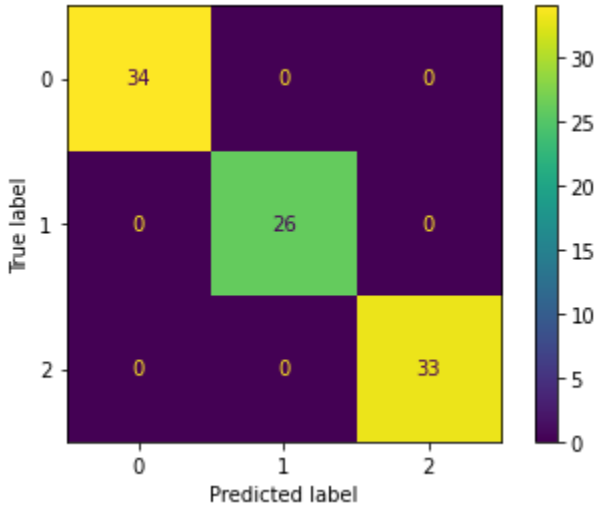
Fold: 3
Accuracy: 0.9864864864864865
Precision: 0.9871794871794872
Recall: 0.9861111111111112
F1 Score: 0.9863718537060214

Fold: 4
Accuracy: 0.9864864864864865
Precision: 0.9866666666666667
Recall: 0.9861111111111112
F1 Score: 0.9861050803300043

Fold: 5
Accuracy: 0.9726027397260274
Precision: 0.9727777777777779
Recall: 0.9727777777777779
F1 Score: 0.9727777777777779

Mean Metrics
Mean Accuracy: 0.9837097371343948
Mean Precision: 0.9839914529914531
Mean Recall: 0.9836666666666666
Mean F1 Score: 0.9837176090294273
Best Estimator: 1
[0 1 2]
Test Metrics

	precision	recall	f1-score	support
0	1.00	1.00	1.00	34
1	1.00	1.00	1.00	26
2	1.00	1.00	1.00	33
accuracy			1.00	93
macro avg	1.00	1.00	1.00	93
weighted avg	1.00	1.00	1.00	93



Model: DecisionTreeClassifier
Fold: 1
Accuracy: 0.8783783783783784
Precision: 0.8799651100375737
Recall: 0.8783333333333333
F1 Score: 0.8782051282051282

Fold: 2
Accuracy: 0.918918918918919
Precision: 0.9242610837438422
Recall: 0.9177777777777778
F1 Score: 0.9178634416729655

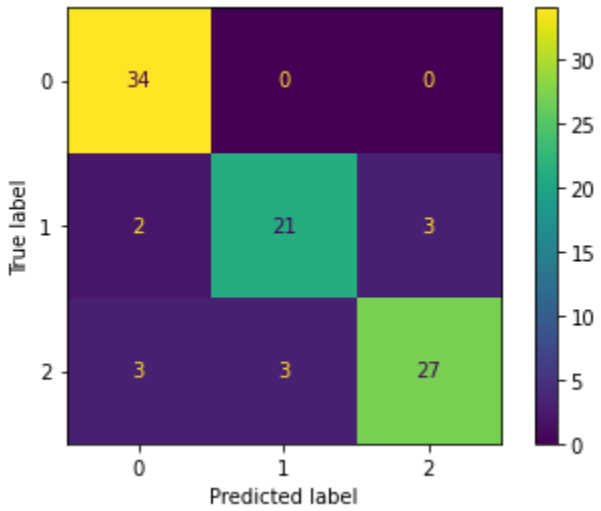
Fold: 3
Accuracy: 0.918918918918919
Precision: 0.9201571268237935
Recall: 0.9188888888888888
F1 Score: 0.9177048888295088

Fold: 4
Accuracy: 0.8783783783783784
Precision: 0.87995337995338
Recall: 0.8771367521367521
F1 Score: 0.8775473801560759

Fold: 5
Accuracy: 0.8493150684931506
Precision: 0.8501486436269046
Recall: 0.8488888888888889
F1 Score: 0.8488232739904472

Mean Metrics
Mean Accuracy: 0.888781932617549
Mean Precision: 0.8908970688370988
Mean Recall: 0.8882051282051282
Mean F1 Score: 0.888028822570825
Best Estimator: 1
[0 1 2]
Test Metrics

	precision	recall	f1-score	support
0	0.87	1.00	0.93	34
1	0.88	0.81	0.84	26
2	0.90	0.82	0.86	33
accuracy			0.88	93
macro avg	0.88	0.88	0.88	93
weighted avg	0.88	0.88	0.88	93



Model: LogisticRegression
Fold: 1
Accuracy: 0.9864864864864865
Precision: 0.9871794871794872
Recall: 0.9866666666666667

F1 Score: 0.9866613311991462

Fold: 2
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0

Fold: 3
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0

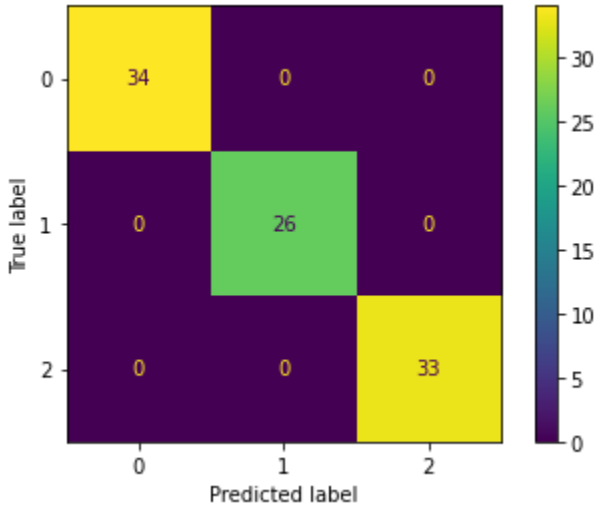
Fold: 4
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0

Fold: 5
Accuracy: 0.9726027397260274
Precision: 0.9727777777777779
Recall: 0.9727777777777779
F1 Score: 0.9727777777777779

Mean Metrics
Mean Accuracy: 0.9918178452425028
Mean Precision: 0.9919914529914531
Mean Recall: 0.9918888888888888
Mean F1 Score: 0.9918878217953848
Best Estimator: 1
[0 1 2]
Test Metrics

	precision	recall	f1-score	support
0	1.00	1.00	1.00	34
1	1.00	1.00	1.00	26
2	1.00	1.00	1.00	33
accuracy			1.00	93

macro avg	1.00	1.00	1.00	93
weighted avg	1.00	1.00	1.00	93



Model: GaussianNB
Fold: 1
Accuracy: 0.9459459459459459
Precision: 0.9487179487179488
Recall: 0.9455555555555555
F1 Score: 0.9462915601023019

Fold: 2
Accuracy: 0.918918918918919
Precision: 0.9292929292929294
Recall: 0.9183333333333333
F1 Score: 0.9194348667059121

Fold: 3
Accuracy: 0.9324324324324325
Precision: 0.9331995540691193
Recall: 0.9322222222222223
F1 Score: 0.9324488944513977

Fold: 4
Accuracy: 0.9594594594594594
Precision: 0.9605128205128205

Recall: 0.9594017094017094
F1 Score: 0.9596791253326208

Fold: 5
Accuracy: 0.9178082191780822
Precision: 0.9173892773892773
Recall: 0.9172222222222223
F1 Score: 0.9165057327278738

Mean Metrics
Mean Accuracy: 0.9349129951869678
Mean Precision: 0.937822505996419
Mean Recall: 0.9345470085470087
Mean F1 Score: 0.9348720358640212
Best Estimator: 3
[0 1 2]
Test Metrics

	precision	recall	f1-score	support
0	0.92	0.97	0.94	34
1	1.00	0.96	0.98	26
2	0.94	0.91	0.92	33
accuracy			0.95	93
macro avg	0.95	0.95	0.95	93
weighted avg	0.95	0.95	0.95	93