Analyzing the Similarity of Essays with Natural Language Processing

Jesse Gempel
Introduction to Artificial Intelligence

February 14, 2026

1 Abstract

The premise of this report is to implement three different Natural Language Processing algorithms to determine the similarities of more than 13,000 essays. Each algorithm will analyze all possible pairs of essays to assign a similarity score. Once these scores are assigned, the algorithms will then select five pairs of essays that it decides are the most similar. Several more pairs of essays will be analyzed with a website called *CopyLeaks* to determine similarity metrics, which will then be compared to the scores of each algorithm. Once these comparisons are made, the limitations of the algorithms and their time complexities will be discussed. Throughout this study, the text similarity problem centers on algorithmic accuracy and speed.

It should be mentioned that all of these algorithms will primarily focus on the similarity and frequency of words appearing among these essays. For the purpose of this study, any aspects related to English syntax or semantics will be ignored. The essays are located in a Kaggle dataset from the Learning Agency Lab entitled *Automated Essay Scoring 2.0*.

2 Preprocessing Data

The Kaggle .csv dataset includes 17,307 rows, each containing three columns: an *essay ID* that serves as a unique identifier for each essay represented by a hexadecimal value, a *full text* that contains an entire essay, and a *score* representing an essay's quality.

To begin cleaning the essay data, any instance of the phrase 'PROPER_NAME' will be deleted. The phrase is only used to anonymize the author of each essay, and removing all its instances will prevent unnecessary similarity detection from each algorithm.

The next step is to use a regular expression to remove all punctuation and uppercase letters from the essays. Since the Python programming language executes these algorithms, case sensitivity and punctuation can cause issues. For instance, the spellings "word", "WORD", and "word?" are all considered unique words to Python, and will result in inaccurate essay comparisons. Once punctuation and uppercase letters are removed from the essays, the cleaned results will be processed in a new column called *cleaned essays*.

To finish preprocessing and cleaning the data, the new column *cleaned essays* will be added to a Python DataFrame. All three algorithms will use this column to analyze the cleaned-up essay text. As this process occurs, deletion of the *full text* and *score* columns is also performed since they are not needed. The original *full text* column will result in inaccurate results, and the *score* column assesses the quality of an essay's semantics and structure–none of these aspects are worth analyzing in this study.

```
1 def clean_essay(text):
2   text = text.replace('PROPER_NAME', '')
3   text = text.lower()                                  # Convert csv file to all lowercase.
4   text = re.sub(r'[^\w\s]', '', text)                  # Remove all punctuation.
5
6   return text
7
8 essay_df['cleaned_essay'] = essay_df['full_text'].apply(clean_essay)
```

Figure 1: A demonstration in Python illustrates how to clean text essay from essays. Line 2 removes the unnecessary *PROPER_NAME* phrase, Line 3 removes all capitalization, and Line 4 deletes all punctuation. Line 8 creates a new data set column that acts as the result of implementing these steps.

```
1 # Removes the third column "score".
2 # The column represents the quality of the essay writing and will be of no use to analyzing text similarity.
3
4 essay_df.drop(columns=['score', 'full_text'], inplace=True)
```

Figure 2: A Python demonstration illustrating the removal of the two unnecessary columns: *score* and *full_text*.

3 Analysis of NLP Algorithms

This study will examine three different algorithmic approaches used for natural language processing: the Count Vectorizer approach, the TF-IDF Similarity approach, and the Hashing Vectorizer approach. These algorithms were chosen to satisfy these criteria: that the initial algorithm serves as the basis for comparison, that the second algorithm is *more accurate* than the initial, and that the third algorithm is *faster* than the initial.

Normalization of Results

Each algorithm computes numerical values using different methods of calculation, which are then stored in a matrix. The matrix will then contain a series of rows that can be interpreted as a single vector. These algorithms work with those vectors to determine the similarity of each pair of essays. However, the vector's values must be normalized to minimize any sensitivity to data outliers, maintaining the output's consistency, and increasing the algorithms' accuracy. To ensure normalization of the data, we use a calculation called *cosine similarity*.

Cosine similarity is described as the measure of an angle's cosine value. The angle in question is between two vectors A and B–in this case, the two vectors hold a series of values that result from the comparison of two essays. The cosine similarity is calculated as demonstrated below:

$$cosine\_similarity(A, B) = \frac{A * B}{||A|| \times ||B||}$$

The numerator represents the *dot product* of vectors A and B, and the denominator involves the multiplication of the two vector's magnitudes, which can be calculated like this:

$$||A|| = \sqrt{a_1 + a_2 + ... + a_n}$$

$$||B|| = \sqrt{b_1 + b_2 + ... + b_n}$$

The cosine similarity is to be calculated with every pair of essays, with the exception of those paired with themselves. The value will always be a number between 0 and 1–any numbers closer to 0 represent a pair of essays with *low* similarity, and *high* essay similarities are illustrated by numbers closer to 1. The cosine similarity values will then be stored in an n x n matrix, with n representing the number of essays in the Kaggle dataset. Once this step is completed for each algorithm, the top 5 cosine similarity values will be determined. The values will represent the essay-pair similarities, and each of the top 5 pairs will be illustrated in tabular format under each algorithm's *Results* sections.

3.1 Word-Frequency-Based Approach

The word-frequency-based approach that will be used is called the Count Vectorization approach. Count Vectorization directly determines the number of occurrences for each word in an essay. For each essay, the word counts are stored into a vector–that way, cosine similarity can be applied to each pair of essays.

3.1.1 Methodology

The algorithm begins by iterating through all 17,307 essays to store any unique words inside a Python dictionary. A dictionary deals with each element containing two parts: a *key* and a *value*. The key stores a unique word, and the value stores an integer that serves as an ordering index. For each essay, new words are added to the dictionary in alphabetical order, and all existing words are skipped.

Once all 17,307 essay iterations are completed, the dictionary holds every single unique word from the Learning Agency Lab's dataset. An m x n matrix is then created, with m rows representing the number of essays and n columns representing the number of unique words. Each matrix element stores the number of occurrences of each word for a specific essay. All rows inside the matrix will then represent vectors that will then be normalized with the cosine similarity function illustrated under *Normalization of Results*.

```
1 from sklearn.feature_extraction.text import CountVectorizer
2 from sklearn.metrics.pairwise import cosine_similarity
3 from scipy.sparse import csr_matrix
4
5
6 count_vectorizer_start_time = time.time()
7 similar_essay_count = 5
8
9 # Create the CountVectorizer.
10 vectorizer = CountVectorizer(stop_words='english')
11 count_vectorizer_matrix = vectorizer.fit_transform(essay_df['cleaned_essay'])
12
13 count_vectorizer_df = pd.DataFrame(count_vectorizer_matrix.toarray(), columns=vectorizer.get_feature_names_out())
14
15 # Convert to sparse matrix to prevent using all available RAM in Google Colab.
16 sparse_matrix = csr_matrix(count_vectorizer_df)
17
18 # Normalize with cosine similarity.
19 cosine_similarity_matrix = cosine_similarity(sparse_matrix)
```

Figure 3: Python code implementation of the Count Vectorization NLP algorithm. Line 10 runs the CountVectorizer() function that executes the algorithm, with a *stop_words* parameter that filters out the most common words for calculation. Examples of stop words include "and", "a", "the", and "is." Line 11 places the results inside of a matrix, and Line 13 creates a DataFrame from the matrix where the results become normalized with a cosine similarity matrix. This final matrix stores similarity values ranging from 0 to 1.

3.1.2 Results

```
5 Most Similar Essay Pairs
        1.) Essay 793 is similar to Essay 15844 (score: 0.99801).
        2.) Essay 14394 is similar to Essay 10969 (score: 0.99676).
        3.) Essay 9014 is similar to Essay 9252 (score: 0.99645).
        4.) Essay 6423 is similar to Essay 11131 (score: 0.99153).
        5.) Essay 2789 is similar to Essay 7284 (score: 0.90399).


5 Least Similar Essay Pairs
        1.) Essay 1085 is similar to Essay 16175 (score: 0.00082).
        2.) Essay 1757 is similar to Essay 13556 (score: 0.00085).
        3.) Essay 14684 is similar to Essay 3277 (score: 0.00096).
        4.) Essay 4727 is similar to Essay 6029 (score: 0.00100).
        5.) Essay 12937 is similar to Essay 14045 (score: 0.00100).

Count Vectorizer Execution Time: 43.245 seconds
Total Execution Time: 101.101 seconds
```

Figure 4: A list of the top 5 most and least similar essay pairs from Count Vectorization. The scores were generated from the cosine similarity calculation. Values close to 0 are considered *not similar*, and values close to 1 are considered *similar*.

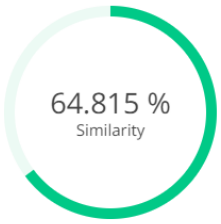
	Essay 1 Index	Essay 1 Key	Essay 2 Index	Essay 2 Key	Essay 1 Sample	Essay 2 Sample
0	793	0c8039c	15844	e9be80d	i think people should participate in the seago...	i think people should participate in the seago...
1	14394	d43da53	10969	a1cde0f	in this essay i will talk about why you should...	in this essay i will talk about why you should...
2	9014	85215b2	9252	88a671e	hi i am and i really recomend that you join t...	hi i am luke and i really recomend that you jo...
3	6423	6017fea	11131	a423c92	i really dont like this new computer thing at ...	i really dont like this new computer thing at ...
4	2789	29aa983	7284	6d25307	a new hom\n\nwhould you send someone to explor...	benefits of researching a new planet\n\nwhould...

Figure 5: Tabular data illustrating the 5 most similar essay pairs according to the Count Vectorization algorithm.

3.1.3 Limitations

An external text similarity tool called Toolsaday.com analyzed many different essay pairings discovered by the Count Vectorization algorithm. Some of the essays were assigned high similarity values (i.e., values close to 1) in the Python code implementation, but these values turned out to be overestimates. In other words, some essay pairs were classified as similar in Python, but at least somewhat different in dedicated text comparison tools. An example of two of these overestimates can be found below in Figure 6.

```
Essay 1:
        Number: 14225
        Key:  d1609f6

  dear state senator

i believe that the electoral college works and that its a problem

Essay 2:
        Number: 16080
        Key:  ed082df

  i think we should keep the electoral college for one reason is its in

Similarity score: 0.7696211668710998
```



dear i think we should keep the electoral college for one reason why electoral college isnt the beat thing for choosing the president of the united states is because the electoral college is unfair to voters for example the winnertakeall system senators only focus on what they care about and not whtas important

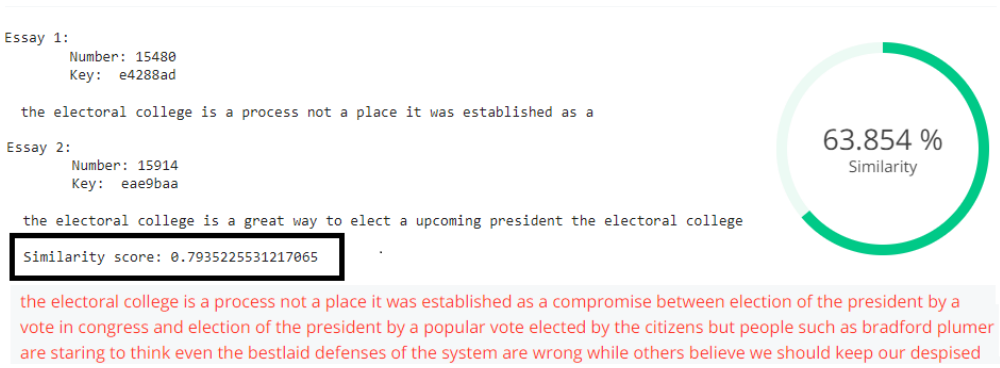
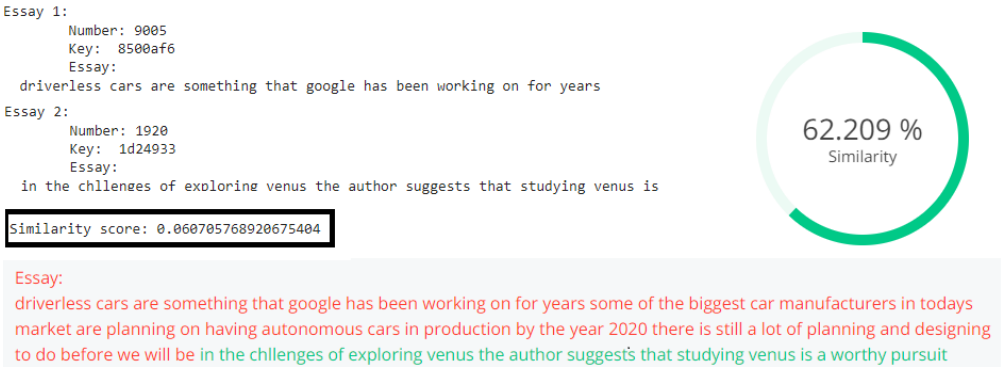


Figure 6: Two essay pairs share high similarity scores (0.770 and 0.793 respectively) under the Count Vectorization algorithm, while Toolsaday.com assigned relatively low similarity percentages. Although the percentages appear high, they are relatively low when considering the tools' comparisons of high-similarity essay pairs.

The use of common words serves as a crucial factor that results in Python's CountVectorizer() function producing excessively high scores. A **common word** is described as one that appears frequently in essays. As shown in Figure 3, a *stop_words* parameter is used to prevent the storage of *some* common words in a Python dictionary–the dictionary was used to store the number of occurrences for each word inside of it. However, not every common word is a stop word. It is difficult to perform two tasks: determine which stop words are included in the CountVectorizer() function, and add a word in a stop word list. Also, a frequently used word such as "dog" may be a *common word*, but Python may not recognize it as a *stop word*. The algorithm's *stop_words* parameter may not prevent "dog" from being stored in the dictionary. This issue causes the similarity score to increase artificially regardless of the context or semantics of an essay. It is worth noting that the removal of stop words somewhat helps improve the algorithm's accuracy. However, more needs to be done to ensure that frequently used words influence the similarity score to a lesser degree.

Similarly, repetitive words and phrases serve as limitations as well. Count Vectorization ignores any context or semantics–it strictly focuses on the commonality of words. A pair of two essays may not be similar at all if their topics (or arguments on a topic) are different. For instance, two argumentative essays in the Kaggle dataset center on the electoral college. One essay supports the abolishment of the electoral college, while another essay wants it to remain in place. If those two essays repeatedly use phrases such as "electoral college" or "political climate," then Count Vectorization will conclude that those two essays are similar. This issue will arise even though they discuss opposing viewpoints about a topic.

The Count Vectorization algorithm also assigned low similarity scores to essays that are actually similar. Figure 7 illustrates two examples of two pairs of essays:



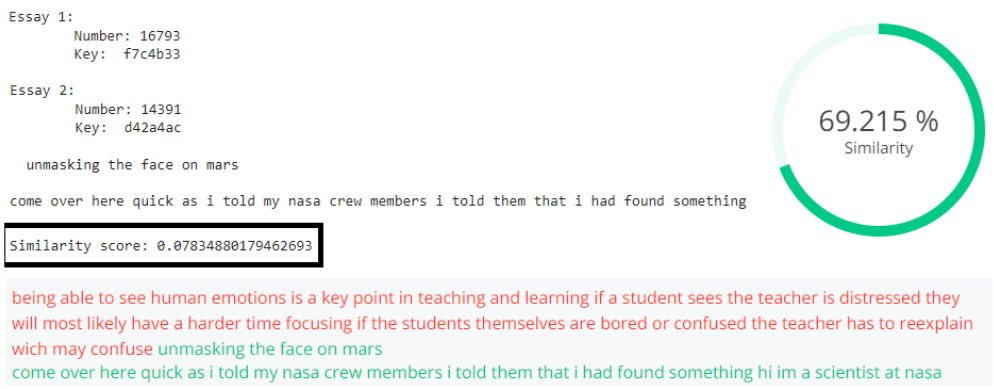


Figure 7: Two essay pairs share very low similarity scores (0.061 and 0.078 respectively) while Toolsaday.com assigned similarity percentages in the 60-70% range. The percentage range is similar to that in Figure 6, but the scores are extremely low in this instance.

The inability to analyze word order serves as yet another example of a practical limitation for Count Vectorization. Two sentences with a different ordering of words can cause the similarity score to diminish. For two sentences with the same exact words, it is true that those words will be placed into a dictionary. It is also true that ordering of words does *not* matter for the algorithm itself. Count Vectorization does assume a high similarity for two sentences or essays that share many of the same words. However, after the algorithm runs, the cosine similarity calculation causes the perceived similarity to drastically decrease. This situation poses an issue because if a person were to copy another essay, he or she could simply change the ordering of sentences to prevent the detection of plagiarism.

Count Vectorization also carries the limitation of failing to recognize the use of synonyms. For instance, the words "stop," "cease," and "abstain" are characterized as synonyms. A person could simply copy another essay and replace all instances of the word "stop" with "abstain" or "cease" to reduce one's similarity score. This issue occurs because the algorithm is solely based on word count. It will not understand that "stop" and "cease" share the exact meaning. Instead, those two words will become separate entries in the Python program's dictionary.

3.1.4 Practical Time Complexity

The Count Vectorization approach is divided into two parts: the actual vectorization and the cosine similarity calculation. The **vectorization process** uses a time complexity of $O(n * d)$, where n represents the number of essays and d illustrates the mean number of words inside each essay. A linear pace is used to create a Python dictionary to store all the unique words in the dataset, then to store word occurrence counts inside a matrix.

The **cosine similarity** process shares a fairly similar time complexity across all three NLP algorithms. For this algorithm, this normalization process has a complexity of $O(n^2 * f)$. n equals the number of essays in the dataset, and f is the number of words inside a Python dictionary. Cosine similarity takes substantial amounts of time because it must calculate the dot product for every possible pairing of essays. For this reason, it takes a quadratic amount of time to navigate n essays.

Combining these two steps, the overall time complexity will become $O(n * d + n^2 * f)$.

3.2 TF-IDF Similarity Approach

The TF-IDF Similarity approach is considered to be more accurate than the Count Vectorization approach. In this approach, it is simply not enough to look at the number of occurrences of each word in an essay. Two values, the *term frequency* (TF) and the *inverse document frequency* (IDF), must be calculated, then multiplied to obtain a TF-IDF value.

3.2.1 Methodology

Similarly to the Count Vectorization algorithm, TF-IDF begins by iterating through every single essay to store unique words in a dictionary. The dictionary's key stores the actual word, and the value stores the index of the word. Once every possible unique word in the dataset is added to the dictionary, a matrix with floating-point values will be created.

An m x n matrix is generated with the same shape as that of the previous algorithm's matrix. m rows still represent the number of essays in the dataset, and n columns still describe the number of unique words in the Python dictionary. However, the value in the matrix represents the result of the TF-IDF calculation, not the number of occurrences of a word.

The calculation begins by calculating the Term Frequency (TF) for each unique word in each essay. The Term Frequency is calculated in this manner:

$$TF(w,d) = \frac{\#\ of\ times\ word\ w\ is\ in\ essay\ d}{total\ \#\ of\ terms\ in\ essay\ d}$$

Next, the Inverse Document Frequency (IDF) must be calculated as shown below:

$$IDF(w) = log\left(\frac{total\#\ of\ essays}{\#\ of\ essays\ with\ word\ w}\right)$$

Once the term frequency (TF) and the inverse document frequency (IDF) are calculated, the two resulting values are multiplied together in this manner:

$$IDF - TF(w,d) = TF(w,d) * IDF(w)$$

The calculation of the TF-IDF value must be repeated for every word in the dictionary for every essay. Each value will then become stored in a matrix–the rows will represent vectors for each essay that will be used for cosine similarity determination.

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2 from sklearn.metrics.pairwise import cosine_similarity
3 from scipy.sparse import csr_matrix
4
5
6 tfidf_start_time = time.time()
7 similar_essay_count = 5
8
9 # Create the TF-IDF vectorier.
10 vectorizer = TfidfVectorizer(stop_words='english')
11 tfidf_vectorizer_matrix = vectorizer.fit_transform(essay_df['cleaned_essay'])
12
13 tfidf_vectorizer_df = pd.DataFrame(tfidf_vectorizer_matrix.toarray(), columns=vectorizer.get_feature_names_out())
14
15 # Convert to sparse matrix to prevent using all available RAM in Google Colab.
16 sparse_matrix = csr_matrix(tfidf_vectorizer_df)
17
18 # Normalize with cosine similarity.
19 cosine_similarity_matrix = cosine_similarity(sparse_matrix)
```

Figure 8: Python code implementation of the Count Vectorization NLP algorithm. Line 10 runs the TfidfVectorizer() function that executes the algorithm, with a *stop_words* parameter that filters out the most common words for calculation. Examples of stop words include "and", "a", "the", and "is." Line 11 places the results inside of a matrix, and Line 13 creates a DataFrame from the matrix where the results become normalized with a cosine similarity matrix. This final matrix stores similarity values ranging from 0 to 1.

3.2.2 Results

```
5 Most Similar Essay Pairs
        1.) Essay 15844 is similar to Essay 793 (score: 0.99816).
        2.) Essay 9014 is similar to Essay 9252 (score: 0.99442).
        3.) Essay 14394 is similar to Essay 10969 (score: 0.98392).
        4.) Essay 11131 is similar to Essay 6423 (score: 0.91546).
        5.) Essay 8406 is similar to Essay 8973 (score: 0.78142).


5 Least Similar Essay Pairs
        1.) Essay 2919 is similar to Essay 4560 (score: 0.00021).
        2.) Essay 2698 is similar to Essay 6992 (score: 0.00022).
        3.) Essay 11555 is similar to Essay 16960 (score: 0.00022).
        4.) Essay 12548 is similar to Essay 6992 (score: 0.00023).
        5.) Essay 13173 is similar to Essay 15858 (score: 0.00024).

TF-IDF Execution Time: 53.640 seconds
Total Execution Time: 112.772 seconds
```

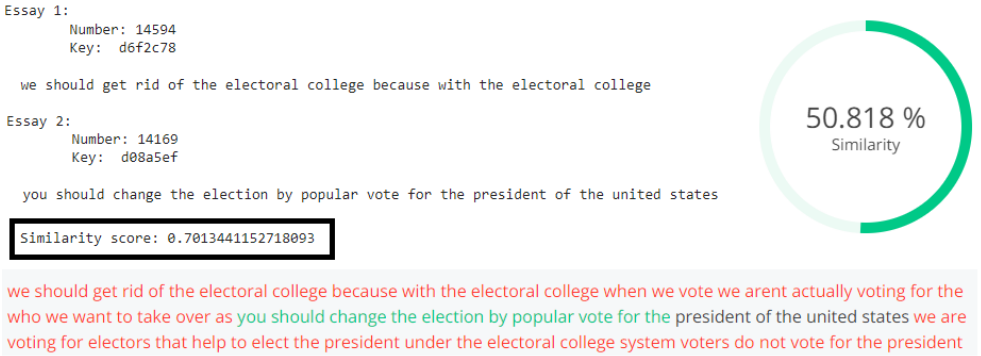
Figure 9: A list of the top 5 most and least similar essay pairs from the TF-IDF Similarity approach. The scores were generated from the cosine similarity calculation. Values close to 0 are considered *not similar*, and values close to 1 are considered *similar*.

	Essay 1 Index	Essay 1 Key	Essay 2 Index	Essay 2 Key	Essay 1 Sample	Essay 2 Sample
0	15844	e9be80d	793	0c8039c	i think people should participate in the seago...	i think people should participate in the seago...
1	9014	85215b2	9252	88a671e	hi i am and i really recomend that you join t...	hi i am luke and i really recomend that you jo...
2	14394	d43da53	10969	a1cde0f	in this essay i will talk about why you should...	in this essay i will talk about why you should...
3	11131	a423c92	6423	6017fea	i really dont like this new computer thing at ...	i really dont like this new computer thing at ...
4	8406	7cdf8b2	8973	84a1b1a	does electoral college work\n\ntoday i am goin...	dear state senator\n\ndo you think that we sho...

Figure 10: Tabular data illustrating the 5 most similar essay pairs according to the TF-IDF Similarity algorithm.

3.2.3 Limitations

The same text similarity tool was used to survey many different essay pairings found by the TF-IDF Similarity algorithm. Some similarity values in the Python code implementation turned out to be artificially high. These observations were conducted by comparing the algorithm's scores with Toolsaday's percentile results. Two examples of TF-IDF Similarity's score overestimates can be found in Figure 11.



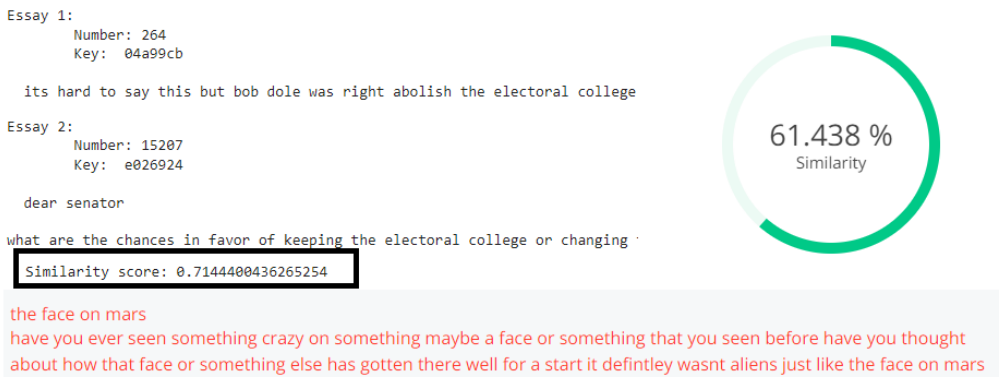
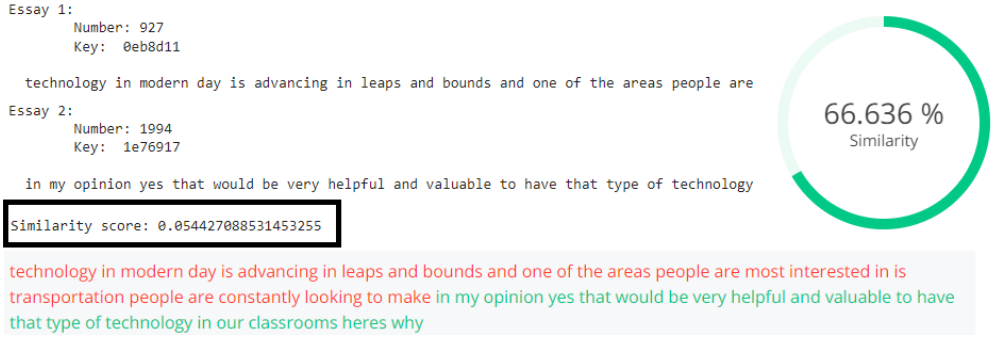


Figure 11: Two essay pairs share high similarity scores (0.701 and 0.714 respectively) under the TF-IDF similarity algorithm. Toolsaday.com's similarity tool assigned relatively low similarity percentages (in the 50-62% range).

Overestimates of similarity scores are partially caused by the algorithm's limitation of inadequately weighing important yet common words. A pair of essays may include many of the same terms, so they appear to lack much difference. However, using a common term such as "not" or "stop" (instead of "go") can drastically change the meaning of a sentence. This observation occurs especially when the two essays attempt to create opposing arguments on the same topic. It also occurs when two essays involve different topics that may share some of the same terminology. For instance, meanings of an essay can drastically change if an author writes about cellular respiration instead of photosynthesis. To describe both of these topics, many of the same biological terms must be used to complete an essay. The presence of these biological terms can inflate the similarity score that the algorithm generates. The TF-IDF Similarity algorithm tends to downplay the use of important words—words that greatly alter a sentence's structure or meaning. For this reason, the algorithm may think that two essays are more identical than they actually are.

TF-IDF also suffers from underestimating its results. Essay pairs with a truly high accuracy are sometimes not represented as such by the algorithm. Figure 12 demonstrates two examples of these types of pairs:



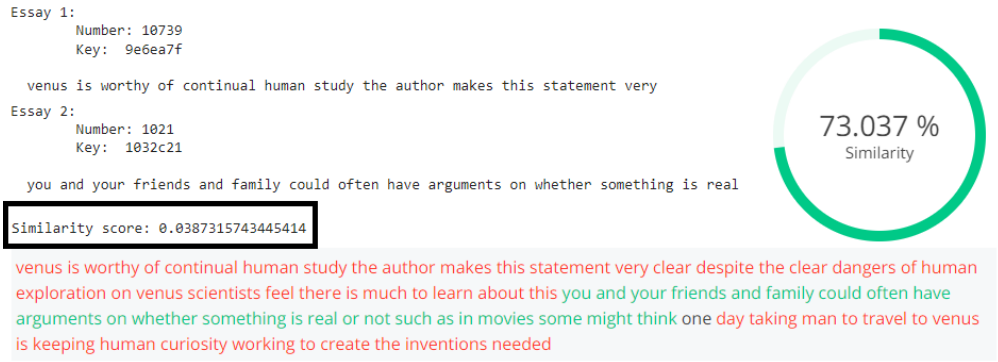


Figure 12: Two essay pairs share very low similarity scores (0.054 and 0.039 respectively) while Toolsaday.com assigned similarity percentages in the 65-75% range. The website obtained slightly lower percentages for essays compared with much higher algorithmic similarity scores.

It was mentioned that under-weighing common words can overestimate similarity scores presented by TF-IDF. The opposite scenario–over-weighing rare words to underestimate scores–serves as yet another limitation to this algorithm. The Inverse Document Frequency (IDF) portion of the TF-IDF calculation increases as the presence of a word decreases. When an uncommon word is present in an essay, it can sometimes cause the word's IDF value to become artificially high. Since the IDF value is directly proportional to the TF-IDF value, a high IDF value can cause an inflated TF-IDF result. This situation especially occurs when Essay 2 does not contain an uncommon word that exists in Essay 1. As a result, the TF-IDF Similarity algorithm can mistakenly mark Essays 1 and 2 as different when they are actually similar.

The algorithm's reaction to two essays with large differences in length acts as a limitation as well. It is possible for a 200 word essay to share high similarity to a 500 word essay. However, the algorithm may not be able to interpret this similarity due to the calculation of the Term Frequency (TF) portion of TF-IDF. The TF calculation for a word depends on two values: 1.) the number of times the word appears in an essay, and 2.) the total number of words in the essay. A short essay with 200 words contains individual words that appear a fewer number of times due to the essay's length. Oftentimes, this scenario can result in a low TF value and thus a low TF-IDF value. A short essay can also affect the IDF value. The Inverse Document Frequency becomes a more reliable metric when an essay contains more words in it.

3.2.4 Practical Time Complexity

Similarly to the Count Vectorization algorithm, the TF-IDF Similarity approach involves the vectiorzation process and the cosine similarity process. The **vectorization process** has a time complexity of $O(n * d)$, where n serves as the total number of essays, and d represents the number of words in an essay. Storing every unique word inside of a Python appears to consume a linear amount of time. However, the algorithm must calculate the TF, IDF, and TF-IDF for every word in every essay. TF-IDF similarity takes longer to complete than the Count Vectorization approach due to those extra steps. Even though that is the case, Count Vectorization and TF-IDF both share the same vectorization time complexity.

The time complexity for **cosine similarity** is $O(n^2 * f)$, which is the same complexity as in the previous algorithm. This process requires the dot product of two essays. Therefore, a quadratic amount of time for n (number of essays) is required for all f unique words, which can be inefficient at times.

These two time complexities add up to form the expression $O(n * d + n^2 * f)$.

3.3 Hashing Vectorizer Approach

The Hashing Vectorization approach is described as a *faster* algorithm than the Count Vectorization approach. It is quite different from the previous algorithms because Hashing Vectorization does not rely on a dictionary to store words and indices. Instead, it utilizes a hashing function to map unique words to a certain number of specified features. It is important to choose the right amount of features for Hashing Vectorization. Too many features will result in the algorithm using too much time and resources, and too few features will result in frequent collisions and poor accuracy.

3.3.1 Methodology

For this study, 20 features will be implemented for this algorithm. Each essay's words in the Kaggle dataset will have the *hash()* function called in Python, which helps calculate an index value in this way:

$$index = hash(unique\_word) \ mod \ n_{features}$$

where $n_{features}$ equals the number of features set as a parameter in Python's *HashVectorizer()* function. When every unique word in the dataset undergoes the above calculation, each word is assigned an index from 0 to $n_{features} - 1$. Since this implementation of the algorithm works with $n = 20$ features, indices of 0 to 24 will be assigned to every unique word (though it is possible for multiple words to have the same index value). 20 columns will also be created in a matrix with m rows representing each essay. Once the matrix is complete, the rows (or vectors) will be normalized by calculating the cosine similarities.

```
1 from sklearn.feature_extraction.text import HashingVectorizer
2 from sklearn.metrics.pairwise import cosine_similarity
3 from scipy.sparse import csr_matrix
4
5
6 hashing_vectorizer_start_time = time.time()
7 similar_essay_count = 5
8
9 # Create the Hashing Vectorizer.
10 vectorizer = HashingVectorizer(n_features=20, norm=None, alternate_sign=False)
11 hashing_vectorizer_matrix = vectorizer.fit_transform(essay_df['cleaned_essay'])
12
13 hashing_vectorizer_df = pd.DataFrame(hashing_vectorizer_matrix.toarray())
14
15 # Convert to sparse matrix to prevent using all available RAM in Google Colab.
16 sparse_matrix = csr_matrix(hashing_vectorizer_df)
17
18 # Normalize with cosine similarity.
19 cosine_similarity_matrix = cosine_similarity(sparse_matrix)
```

Figure 13: Python code implementation of the Count Vectorization NLP algorithm. Line 10 runs the HashingVectorizer() function that executes the algorithm with 20 features. Line 11 places the results inside of a matrix, and Line 13 creates a DataFrame from the matrix where the results become normalized with a cosine similarity matrix. This final matrix stores similarity values ranging from 0 to 1.

3.3.2 Results

```
5 Most Similar Essay Pairs
        1.) Essay 793 is similar to Essay 15844 (score: 0.99993).
        2.) Essay 9252 is similar to Essay 9014 (score: 0.99981).
        3.) Essay 10969 is similar to Essay 14394 (score: 0.99975).
        4.) Essay 11131 is similar to Essay 6423 (score: 0.99951).
        5.) Essay 16819 is similar to Essay 6592 (score: 0.99587).


5 Least Similar Essay Pairs
        1.) Essay 12736 is similar to Essay 10571 (score: 0.44768).
        2.) Essay 11841 is similar to Essay 6389 (score: 0.45081).
        3.) Essay 2383 is similar to Essay 16492 (score: 0.45276).
        4.) Essay 2043 is similar to Essay 10114 (score: 0.46121).
        5.) Essay 10933 is similar to Essay 14670 (score: 0.46448).

Hashing Vectorizer Execution Time: 41.162 seconds
Total Execution Time: 97.954 seconds
```

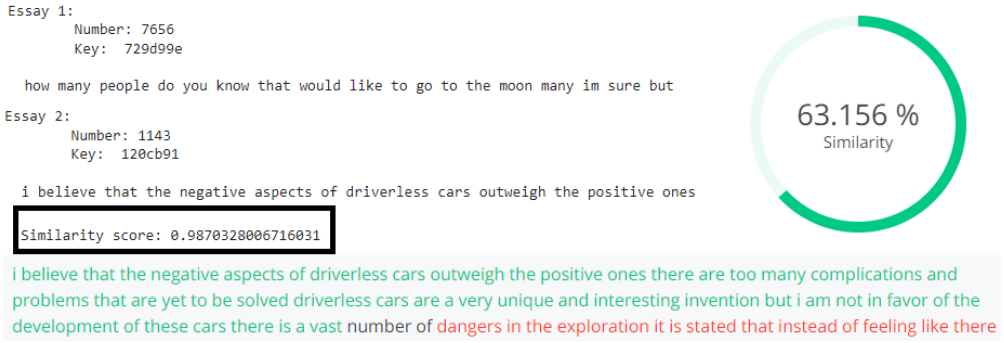
Figure 14: A list of the top 5 most and least similar essay pairs from Hashing Vectorization. The scores were generated from the cosine similarity calculation. Values close to 0 are considered *not similar*, and values close to 1 are considered *similar*.

	Essay 1 Index	Essay 1 Key	Essay 2 Index	Essay 2 Key	Essay 1 Sample	Essay 2 Sample
0	793	0c8039c	15844	e9be80d	i think people should participate in the seago...	i think people should participate in the seago...
1	9252	88a671e	9014	85215b2	hi i am luke and i really recomend that you jo...	hi i am and i really recomend that you join t...
2	10969	a1cde0f	14394	d43da53	in this essay i will talk about why you should...	in this essay i will talk about why you should...
3	11131	a423c92	6423	6017fea	i really dont like this new computer thing at ...	i really dont like this new computer thing at ...
4	16819	f834b3c	6592	6265245	the effect of cars in our world today has grow...	car use all over the world has tried to be red...

Figure 15: Tabular data illustrating the 5 most similar essay pairs according to the Hashing Vectorization algorithm.

3.3.3 Limitations

The results of the Hashing Vectorization algorithm are once again compared with the results from the Toolsaday website. It was discovered that some essay similarity values in the algorithm's Python code turned out to be artificially high. Two examples of TF-IDF Similarity's score overestimates are illustrated in Figure 16 below:



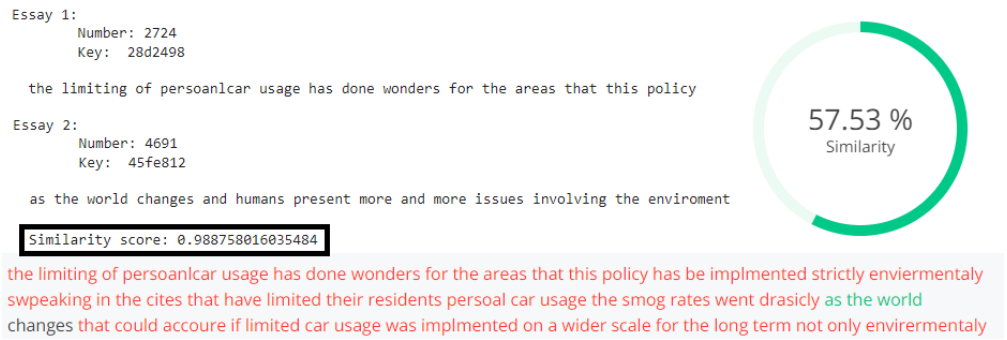


Figure 16: Two essay pairs share high similarity scores (0.987 and 0.989 respectively) under the Hashing Vectorization algorithm. Toolsaday.com's similarity tool assigned relatively lower similarity percentages compared to essay pairs with much lower algorithmic similarity scores. Although Toolsaday's percentages appear somewhat high, they are relatively low compared to the website's comparison of more similar essay pairs.

The lack of an Inverse Document Frequency (IDF) serves as a limitation for the Hashing Vectorization algorithm. Unlike TF-IDF Similarity, Hashing Vectorization does not apply a weight to each word, meaning that every word in an essay shares equal importance–no matter the triviality of the word. Common words that are not contextually important can excessively increase the similarity scores of an essay pair. This situation can occur even if two essays focus on different topics (or viewpoints on topics). For this reason, NLP algorithms such as Count Vectorization and TF-IDF Similarity can include a *stop_words* parameter inside their function. As mentioned in this study's word-frequency-based Approach section, a **stop word** is a common word that does not become stored in a Python dictionary. Hashing Vectorization does not employ a dictionary to store words; it uses a hash function that works with a set number of features. As a result, the *stop_words* parameter does not exist in Python's HashingVectorizer() function. The lack of such a parameter is a limitation because the algorithm does not have a way to filter out frequently used words. These scenarios cause essay pairs to become overestimated in their similarities.

Another limitation with Hashing Vectorization is the presence of hash collisions, which also contributes to the algorithm's tendency to overestimate similarities. **Hash collisions** are the tendency of hash functions to assign indices (or hash values) to two or more words. These collisions commonly occur when the number of features in Python's HashingVectorizer() function is set to an excessively low number. A low number of features can cause an overestimation of essay similarities due to multiple words sharing an identical index. When two or more words share an index, the algorithm treats those words as if they were equal. This situation poses an issue when a hash collision occurs between radically different words, such as "radish" and "buddy." Unrelated words that share an index can cause essays about unrelated topics to achieve an overly high similarity score.

Hashing Vectorization constitutes yet another algorithm that sometimes encounters underestimations of essay similarity. Scenarios occurred where each essay from a pair was distinct from one another. However, external online comparison tools such as Toolsaday appeared to prove otherwise. Figure 17 illustrates two pairings with an underestimation of similarity:

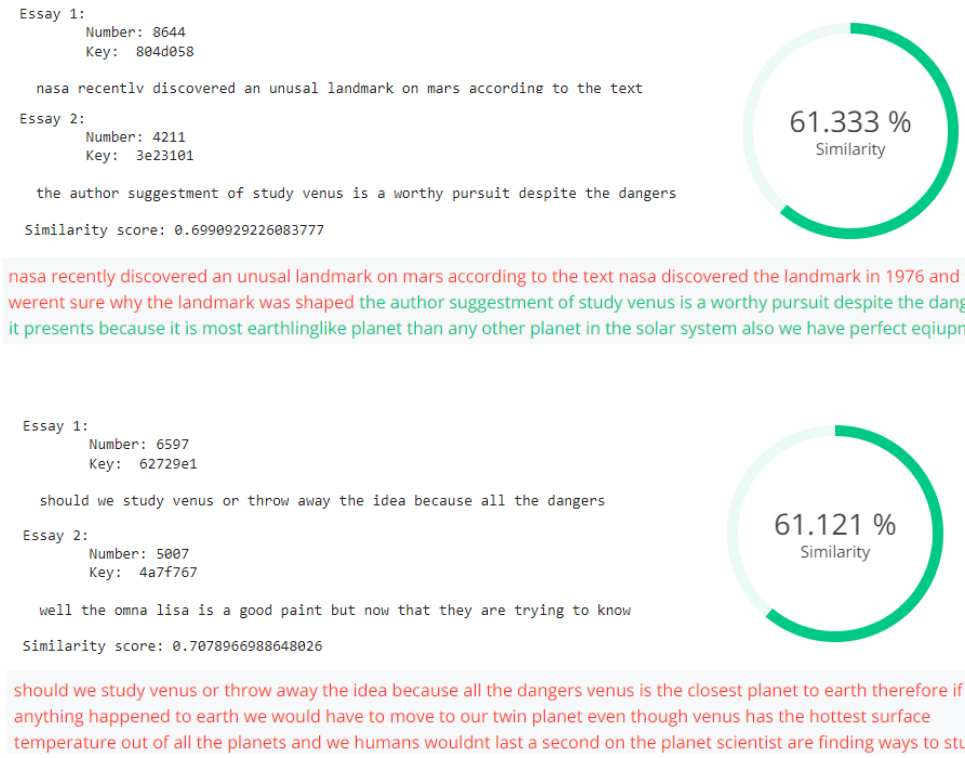


Figure 17: Two essay pairs share relatively low similarity scores (0.700 and 0.708 respectively). Although the scores are still considered high, the range of similarity scores is between 0.46 and 1 across all essay pairs. Toolsaday.com generated similarity percentages, both at around 61%. The website obtained slightly lower percentages for essays compared with much higher algorithmic similarity scores.

A lack of indices in the hashing vectorizer can cause the algorithm to mistakenly conclude an essay pair with similarity as one without similarity. This limitation can be overcome by increasing the number of features in the HashingVectorizer() function's *n_features* parameter. By keeping a low number of features, the algorithm is required to assign identical indices to multiple words. Not only can that situation present an overestimation of essay pair similarity, it can also result in an underestimation. For example, an essay with 500 unique words needs to undergo the Count Vectorization process, which runs with only 20 features. Each of those 500 words is assigned an index, and due to the low feature count, more than 20 words must share a single index. This issue can cause distinct terms to be excluded from a hashing vector, which results in loss of information about which words are to be compared.

Because of this loss of information, it can be difficult to determine which words contribute to the algorithm's essay similarity scores. The difficulty in analyzing the results from the Hashing Vectorization algorithm acts as another limitation. The algorithm sometimes uses the same index number for multiple words for n number of features. As a result, it is nearly impossible to determine which exact words are responsible for the similarity or dissimilarity of two essays. The only information that can be gathered is the index numbers inside the hashing vector.

3.3.4 Practical Time Complexity

Unlike with the previous two algorithms, this approach's **vectorization process** does not require a large Python word dictionary to be created. The lack of such a step makes Hashing Vectorization the fastest algorithm in this study, at least for this step. The time complexity for vectorization is $O(n * d)$ due to the linear speed of executing hash functions, determining the indices of words, and inserting the indices into a matrix.

The time complexity for the **cosine similarity** step is $O(n^2 * k)$. n represents the number of essays in

the Kaggle dataset, and k represents the number of features represented by the *n-features* argument in Python's *HashingVectorizer()* function. A large, quadratic amount of time is required to calculate the dot product of any two essays in every pair combination. Although the algorithm's speed is generally fast, the number of features (k) can greatly affect how quickly normalizing the values can finish.

When these two steps are combined, the overall time complexity of Hashing Vectorization is $O(n * d + n^2 * k)$. The three NLP algorithms appear to share similar complexities, with the exception of k.

4 Discussion

The utilization of resources upon running each NLP algorithm acted as a major challenge for this study. The first algorithm that I attempted to program was the Counting Vectorizer algorithm. For the program to complete its execution, I needed to overcome the barrier of utilizing all available RAM on Google Colab without paying a subscription. At first, I simply followed these three steps: I created a vectorizer, assigned the matrix results to a variable, and applied cosine similarity to the matrix. Those steps were correct, assuming that I had unlimited resources. It turned out that Google Colab crashed every time I ran the program due to excessive RAM consumption. To work around that issue, I needed to assign the vectorizer matrix results to a DataFrame, convert it into a sparse matrix, and apply cosign similarity to the sparse matrix. The inclusion of a sparse matrix allowed complete execution of the first algorithm. Once the code worked, I applied the same concept to the other two algorithms and overcame the RAM utilization issue.

Finding a website to calculate similarity values between two essays acted as another challenge. Between several websites, the essay similarity results had a low variance of quantities. Many websites would present results between 5% and 20%, or between 45% and 60%. The lack of variance for the results would occur, regardless of whether two essays were similar to or different from each other. I also faced the issue of trying to find a text comparison website that did not require a subscription. Ultimately, I ended up using Toolsaday.com to satisfy the "Limitations" section of this study.

Finding pairs of essays to use served as another challenge. For the "Limitations" section, the goal was to find essay pairs that were similar according to one of the three algorithms, but *not* similar according to a website, and vice versa. To find pairs of essays in Google Colab with low and high similarity scores, I programmed an essay pair randomizer. This randomizer was designed using a conditional statement that only presented similarities within a certain range. It would print the two essays' hexadecimal IDs, indices, and texts in their entirety. However, I needed to run the randomizer many times to finally find pairs whose similarity scores were different from Toolsaday's results.

Once I found the pairs of essays that I wanted to visualize, another challenge was determining *how* to visualize them. I needed to show both the results from Google Colab and Toolsaday. However, the visualizations needed to be compact and easy to follow to avoid consuming excess space and confusion in the report. to be easy to follow. The fact that twelve different visualizations needed to be included for all the essay pairings was especially concerning. To resolve this challenge, I combined the use of Windows Snipping Tool and Microsoft Paint to visualize all the information in a compact, easy-to-read manner.

5 Resources

My Google Colab notebook for this report can be accessed with this link:
https://colab.research.google.com/drive/1oqxMUQnl-Sz0i4D8bCYLgJWCM45c1XMR?usp=sharing.

To access my conversation with ChatGPT that influenced this report, the link can be found below:
https://chatgpt.com/c/67117041-c3f8-8000-815c-43a7fc147e6f.

I used an external tool that analyzes the similarity between two essays. It was used to compare

its results to the results of the three algorithms. This tool can be located on https://toolsaday.com/text-analysis/similarity-checker.

6 References

[1] Kaggle. (2024, July 2). *Learning Agency Lab - Automated Essay Scoring 2.0* Retrieved from https://www.kaggle.com/competitions/learning-agency-lab-automated-essay-scoring-2/data