Representing Books as Vectors

Sam Roussel

In collaboration with Jaya Blanchard, Killian Dickson, and Andrew Macdonald

May 2018

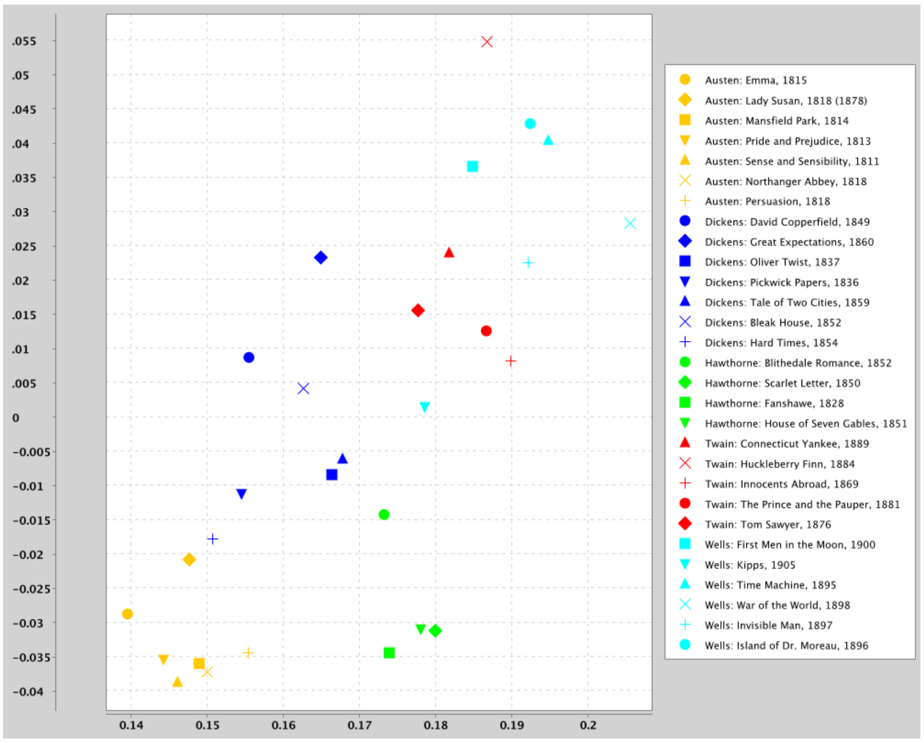

Figure 1: Our results show a strong correlation by author.

Contents

1 Introduction

This paper explains my final project for Math 2301: Intermediate Linear Algebra with Professor Thomas Pietraho. The project was done in collaboration with Jaya Blanchard, Killian Dickson, and Andrew Macdonald.

The original inspiration for this project was a desire to model language in two-space. Similar endeavors include Word2Vec, which uses a neural network to represent words as vectors. We took a more general approach, asking instead how books could be represented as vectors. To do this, we counted the frequencies of the most common words across a corpus of texts. Each book was thus represented as a vector of frequencies. These were normalized, combined into a matrix, and then, using more advanced linear algebra techniques (mainly Singular Value Decomposition), projected onto two-space. This yielded strong correlations by author. We also explored other factors, such as word selection, book selection, and time period. Given further study, there are a number of interesting questions. Do books cluster by genre? Can changes in an author's style be seen over time? Is there an optimal set of words to chose? While we were unable to answer all of these here, we did achieve some useful results through our process.

2 Mathematical Background

Singular Value Decomposition is the key technique to this paper, but some background is necessary before we can truly explore it. In particular, the Spectral Theorem is an important precursor to SVD. It is more specific but also simpler, allowing for the decomposition of certain matrices, and is thus important to discuss. Before we can state this theorem, however, it is necessary to define a few terms.

2.1 Basics and Definitions

Definition 1 (Symmetric Matrix)**.** A matrix A is said to be a **symmetric** matrix if and only if it is equal to its transpose, that is:

$$A = A^T.$$

Definition 2 (Diagonal Matrix)**.** Let the matrix $A = (a_{ij})$, where each a_{ij} is the element of A at the ith row and the jth column. A is said to be a **diagonal** matrix if and only if its entries outside the main diagonal are zero, that is:

$$a_{ij} = 0, \ i \neq j.$$

Definition 3 (Diagonalizable Matrix)**.** A matrix A is said to be a **diagonalizable** matrix if and only if $\exists$ a diagonal matrix Δ and an invertible matrix P such that:

$$A = P\Delta P^{-1}.$$

Definition 4 (Orthogonal Matrix)**.** A matrix A is said to be an **orthogonal** matrix if and only if its transpose is equal to its inverse, that is:

$$A^T = A^{-1}.$$

Definition 5 (Change of Basis Matrix)**.** A matrix is the **change of basis** matrix from A to B if and only if its columns consist of the coordinate vectors of B's basis relative to A's basis.

2.2 Spectral Theorem

Theorem 1 (Spectral Theorem)**.** If A is a symmetric n × n matrix with real-valued entries, there exists an orthogonal matrix U such that $A = U\Delta U^T$, where Δ is diagonal and consists of A's eigenvalues, and the columns of U are A's eigenvectors. In other words, A is diagonalizable.

Importantly, diagonalizable matrices must have a basis of eigenvectors, such that $Av_i = \lambda_i v_i$. Each v_i becomes a column of U, and each λ_i becomes a diagonal entry of Σ. From this perspective, the underlying geometry of the process can be revealed. Essentially, performing the transformation A on a vector maps the vector onto the basis $\{v_i\}$, stretches it along those axes by factors of λ_i, and then undoes the mapping.

This theorem is very useful, as it allows any symmetric, square matrix to be diagonlized. The problem is that it only works for symmetric, square matrices. It would be much better if we could somehow generalize this idea to any matrix. Luckily, this is where Singular Value Decomposition comes in.

2.3 Singular Value Decomposition

Because the Spectral Theorem relies on the symmetry of the matrix in question, it is possible that a non-symmetric matrix is diagonalizable, but not by the Spectral Theorem. The fundamental insight of SVD is to use two different matrices as the "geometric maps" for the transformation. Where in the Spectral Theorem we mapped a vector to a new set of coordinates, stretched it, and then sent it back, in SVD, we map a vector to a new set of coordinates, stretch it, and then send it to another basis.

Essentially, we can write $Av_i = \sigma_i u_i$ as an analog to the normal eignenvalue equation. Here σ_i is a singular value of A, that is, the square root of an eigenvalue of $A^T A$ and AA^T. These matrices are trivially symmetric:

$$(A^T A)^T = A^T A,\ (AA^T)^T = AA^T$$

and are thus diagonalizable. Because of this, we can write:

$$A^T A = V\Delta_1 V^T$$

and

$$AA^T = U\Delta_2 U^T.$$

From the first equation, we have, since V is orthogonal:

$$A^T AV = V\Delta_1$$

For each eigenvalue:

$$A^T Av_i = \lambda_i v_i$$

Multiplying by v^T:

$$v_i^T A^T Av_i = \lambda_i v_i^T v_i$$

By definition of the usual inner product and norm:

$$\|Av_i\|^2 = \lambda_i \|v_i\|^2$$

Since v_i is orthonormal:

$$\|Av_i\|^2 = \lambda_i,$$

$$\|Av_i\| = \sqrt{\lambda_i}.$$

From above, we also know that Av_i is an eigenvector of AA^T, since we can multiply both sides of the earlier equation by A:

$$AA^T Av_i = \lambda_i Av_i.$$

Thus any multiple of Av_i is also an eigenvector, including $\frac{Av_i}{\|Av_i\|}$. Let $U_i = \frac{Av_i}{\|AV_i\|}$. Rearranging and substituting our earlier result:

$$AV_i = \sqrt{\lambda_i} U_i.$$

Let $\sigma_i = \sqrt{\lambda_i}$. Thus $AV_i = \sigma_i U_i$.

Let $\Sigma = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n \end{pmatrix}$. Then $AV = U\Sigma$.

Therefore, finally:

$$A = U\Sigma V^T.$$

This provides the foundation for the Singular Value Decomposition Theorem:

Theorem 2 (Singular Value Decomposition Theorem)**.** Suppose A is an m $\times$ n matrix with real-valued entries. Then $\exists$ orthogonal matrices U and V, such that:

$$A = U\Sigma V^T,$$

where Σ is defined as above.

Each σ_i is a singular value of A, that is, the square root of an eigenvalue of $A^T A$ and AA^T. Crucially, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$.

Geometrically, U and V can be thought of as **change of basis** matrices. V maps the given vector into a new basis, where it is stretched in each dimension by Σ, and then U maps it back to the standard basis.

2.4 Principal Component Analysis

Once a matrix has been decomposed by singular values, it is possible to do some interesting data manipulation. In particular, we can project our vectors onto a smaller space, keeping the most important information, while providing an easier to understand representation of the data.

The most obvious way to does this is to simply truncate our matrix, keeping the first n values of each column vector. For 2-space, this would involve simply taking the first two rows of our matrix. This naïve approach however, would eliminate much of the important data. We should thus represent our matrix in a different basis before truncating it.

Let the basis $B = \{U_i\}_{i=1}^{n}$, and $a_i = \sum_{j=1}^{n} \alpha_{ij} U_j$. Thus the coordinates for a_i in the basis B are each α_{ij}. This yields a new version of A, which we can call Y:

$$Y = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1m} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \dots & \alpha_{nm} \end{bmatrix}.$$

To go from A to Y, we're really multiplying by U:

$$A = UY.$$

This gives:

$$\begin{aligned} U^T A &= Y \\ &= U^T U \Sigma V^T \\ &= \Sigma V^T. \end{aligned}$$

This resulting matrix is very easy to calculate given the SVD. This is the calculation we perform in the code. Once we have $= \Sigma V^T$, we can simply truncate all but the first two coordinates of each column vector. The resulting vectors are thus representative of the most important stretching factors.

3 Process

3.1 Frequency Vectors

The first and arguably most important piece of our process was actually representing a book as a vector. To do this, we decided to count the frequencies of a given set of words, with each frequency becoming a coordinate in the vector. Under this basic system, however, vectors of longer works would inevitably have greater magnitudes. To counteract this, we decided to normalize each vector, allowing for the real difference in word choice to show through the data, regardless of length.

3.2 Choice of Words

An important question arises: which words should be counted? Our first attempt selected 100 random words out of the 10,000 most common words. This yielded little correlation by author, and it also showed no continuity between sets of words, that is, each graph looked different, since our set of words was randomized each time.

We then decided to take a more logical approach: using the most frequent words across our data. This involved counting each word in each book, normalizing those vectors so as to avoid weighing larger works more heavily (as discussed above), and then combining them. This yielded the most used words across our corpus. These words were then counted for each book. Somewhat problematically, we found that using the top 100 words led to a linear trend in our graphed vectors. Removing the top two words, we found a better spread and a less linear relationship between the graphed x and y values.This was likely because of a correlation between the largest two singular values and the most two common words. Thus our final set of words included the third most common word through the hundredth.

3.3 Authors and Books

We decided to first work with authors of overlapping but slightly different eras, who most importantly had many works in the public domain. These authors were Jane Austen (1775-1817), Charles Dickens (1812-1870), Nathaniel Hawthorne (1804-1864), Mark Twain (1835-1910), and HG Wells (1866-1946). Each author had between four and seven books as data. Originally, we also wanted to include Agatha Christie, but her works were too recent to give more than a couple public domain books. We also had included "Discovery of the Futute" as one of H.G. Wells works, but upon further investigation, realized it was both an outlier on the graph and not even a book, but a lecture. Given this, we were careful to ensure that the rest of our data were novels or plays.

All books were in the public domain and downloaded as plain text files from Project Gutenberg. Some cleanup of these files was required, namely deleting the Project Gutenberg specific information at the beginning and end of each file.

In a later trial, we decided to add Shakespeare as an author from a much different time period. This had interesting results. Clustering by author continued, but it was less clear, likely because of the vastly different vocabulary used by Shakespeare.

3.4 Putting it Together

At this point, we have discussed the selection of works, a method of converting them into vectors, and the mathematical background necessary for projecting them onto 2-space. To actually get an image, some programming is required.

Using Java, we created a basic FrequencyMap class to map words to frequencies, as well as providing functionality for normalizing and combining such maps. These maps were essentially vectors with the words stored in addition to the frequencies. Using this structure, we parsed each text file into a string, an array of words, and finally, a FrequencyMap. We then combined these maps (really vectors) into one large map, sorted it by frequency, and stored the top 100 words. We then created new vectors for each book by extracting only the top 100 words from each existing vector. These were again normalized, yielding our final frequency vectors.

At this point, we needed to do some linear algebra. Luckily, there is a package called Jama which allows for various matrix operations, including SVD. After finessing the vector data into a basic two-dimensional array, we performed the SVD operation and the important $= \Sigma V^T$ multiplication. Finally, we truncated down to two columns, leaving a matrix of two-dimensional vectors corresponding to each book.

We then needed another package to do graphics for us. Knowm's XChart package worked well for this. With some styling and a little manipulation from the matrix to lists, we were able to display our data in two dimensions.

4 Results

4.1 Without Shakespeare

Below is the first image (Figure 2) yielded by our process, with Shakespeare not included. As can be seen, books cluster very nicely by author. We can also see a general time trend, with the early Jane Austen (1775-1817) coming before Dickens (1812-1870) and Hawthorne (1804-1864), who were roughly contemporaneous, followed by Twain (1835-1910) and Wells (1866-1946), who also follow chronologically. This pattern makes sense, given the evolution of language over time. The pattern failed, however, to emerge cleanly within an author's own works, suggesting that the correlation may not be as strong as it appears. Then again, the chosen authors may just have been consistent throughout their careers.

The question of consistency is also potentially answered by how closely author's works are clustered. Jane Austen, in particular, is much more clustered than the others. This could be indicative of consistent word choice, but it could also reflect the relative short span (seven years) in which her works were written.

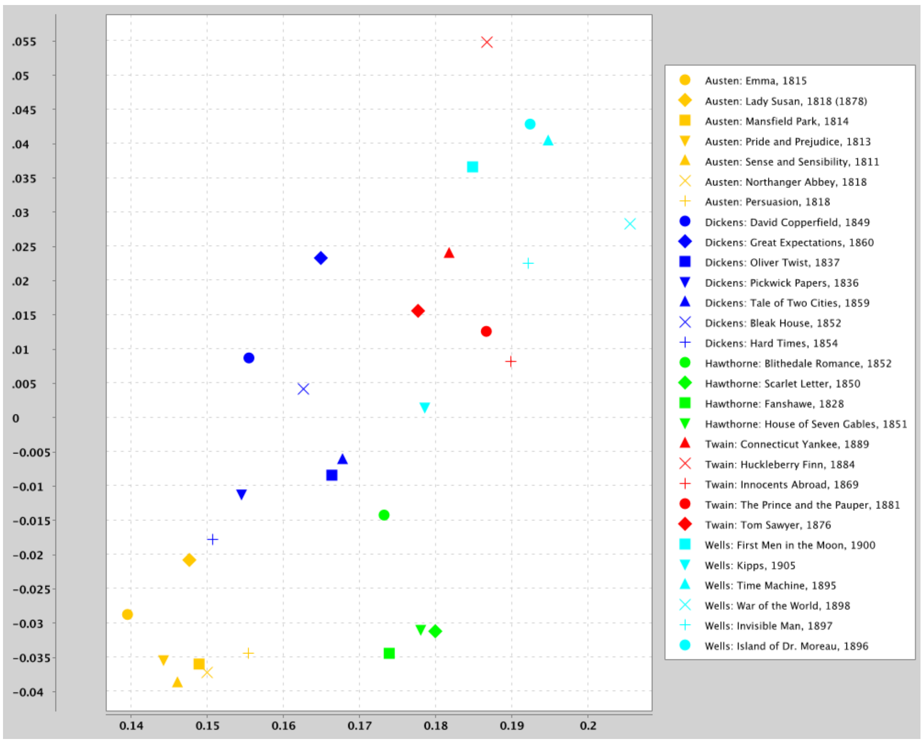

Figure 2: The results show a strong correlation by author.

4.2 With Shakespeare

We then added Shakespeare to our data. This had some interesting effects on the graph as a whole. As can be seen on the next page (Figure 3), the clustering by author is somewhat less clear than before, particularly with Wells, Hawthorne, and Twain being more intermixed. Remarkably, Jane Austen is still the most clustered, excepting the outlier Lady Susan. Nonetheless, there is still significant clustering by author, and perhaps to some extent, time, if one is willing to imagine a parabola through the data.

It makes a lot of sense that adding Shakespeare would weaken our correlations. Shakespeare's vocabulary was vastly different from the other authors given his different era. This meant that the other books had the common trait of containing none of the Shakespeare-specific words, and thus, became more closely related, regardless of author.

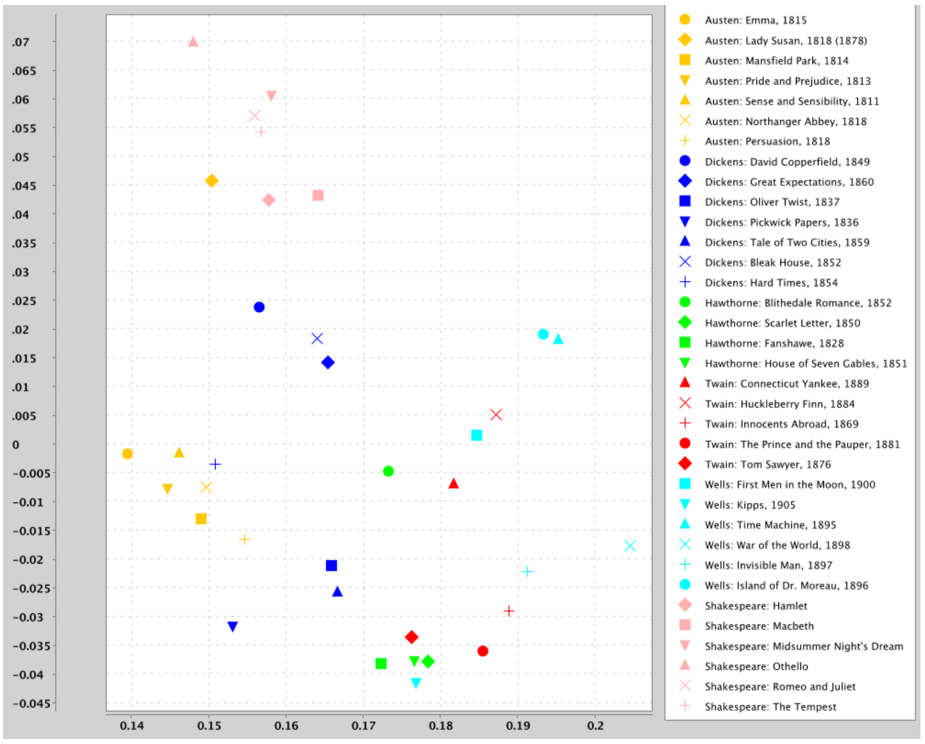

Figure 3: Shakespeare shakes things up.

5 Conclusion

In this project, we sought to represent books in two dimensions, with the hope that correlations would emerge. Our results showed significant clustering by author, and some clustering by time period.

Given the limited sample size and relatively arbitrary selections of books and authors, there is significant room for growth in the future. Natural questions such as genre-clustering and development over an author's career arise. In addition, perhaps there is some optimal selection of words to count instead of the top 100 without the first two. It would be interesting to explore these and other questions given more time and resources.

This project provided a fun and practical application for some of the theoretical math learned in Math 2301. It was very rewarding to be able to see the theory in action. Special thanks to Jaya, Killian, and Andrew for being a great team, and to Professor Pietraho for a great semester.

6 Sources

Project Gutenberg. (n.d.). Retrieved May, 2018, from www.gutenberg.org.

Jama - This software is a cooperative product of The MathWorks and the National Institute of Standards and Technology (NIST) which has been released to the public domain. Neither The MathWorks nor NIST assumes any responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic.

XChart - XChart is developed by Knowm Inc. members and the open-source community. It is covered under an open source Apache License.

7 Appendix (Code)

```
import java.awt.*;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.stream.Collectors;
import java.util.Map;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.Collections;
import org.knowm.xchart.*;
import org.knowm.xchart.demo.charts.ExampleChart;
import org.knowm.xchart.XYSeries.XYSeriesRenderStyle;
import Jama.Matrix;

public class Main
{
    private static Matrix x;
    private static Matrix y;


    public static final String[] filenames =
    {
            "src/austen_emma.txt", "src/austen_lady.txt",
            ↪ "src/austen_mansfield.txt",
            ↪ "src/austen_pride.txt", "src/austen_sense.txt",
            ↪ "src/austen_north.txt",
            ↪ "src/austen_persuasion.txt",
            "src/dickens_copperfield.txt",
            ↪  "src/dickens_great.txt",
            ↪  "src/dickens_oliver.txt",
            ↪  "src/dickens_pickwick.txt",
            ↪  "src/dickens_tale.txt", "src/dickens_bleak.txt",
            ↪  "src/dickens_hard.txt",
            "src/hawthorne_blithedale.txt",
            ↪  "src/hawthorne_scarlet.txt",
            ↪  "src/hawthorne_fanshawe.txt",
            ↪  "src/hawthorne_gables.txt",
            "src/twain_connecticut.txt", "src/twain_huck.txt",
            ↪  "src/twain_innocents.txt",
            ↪  "src/twain_prince.txt", "src/twain_tom.txt",
            "src/wells_first.txt", "src/wells_kipps.txt",
            ↪  "src/wells_time.txt", "src/wells_war.txt",
            ↪  "src/wells_invisible.txt",
            ↪  "src/wells_island.txt",
            //"src/shakespeare_hamlet.txt",
            ↪  "src/shakespeare_macbeth.txt",
            ↪  "src/shakespeare_midsummer.txt",
            ↪  "src/shakespeare_othello.txt",
            ↪  "src/shakespeare_r&j.txt",
            ↪  "src/shakespeare_tempest.txt"
    };

    public static final String[] niceNames =
    {
            "Austen: Emma, 1815", "Austen: Lady Susan, 1818
            ↪  (1878)", "Austen: Mansfield Park, 1814", "Austen:
            ↪  Pride and Prejudice, 1813", "Austen: Sense and
            ↪  Sensibility, 1811", "Austen: Northanger Abbey,
            ↪  1818", "Austen: Persuasion, 1818",
            "Dickens: David Copperfield, 1849", "Dickens: Great
            ↪  Expectations, 1860", "Dickens: Oliver Twist,
            ↪  1837", "Dickens: Pickwick Papers, 1836",
            ↪  "Dickens: Tale of Two Cities, 1859", "Dickens:
            ↪  Bleak House, 1852", "Dickens: Hard Times, 1854",
            "Hawthorne: Blithedale Romance, 1852", "Hawthorne:
            ↪  Scarlet Letter, 1850", "Hawthorne: Fanshawe,
            ↪  1828", "Hawthorne: House of Seven Gables, 1851",
            "Twain: Connecticut Yankee, 1889", "Twain:
            ↪  Huckleberry Finn, 1884", "Twain: Innocents
            ↪  Abroad, 1869", "Twain: The Prince and the Pauper,
            ↪  1881", "Twain: Tom Sawyer, 1876",
```

```java
            "Wells: Discovery of the Future, 1902", "Wells: Kipps, 1905", "Wells: Time Machine, 1895", "Wells: War of the World, 1898", "Wells: Invisible Man, 1897", "Wells: Island of Dr. Moreau, 1896",
            //"Shakespeare: Hamlet", "Shakespeare: Macbeth", "Shakespeare: Midsummer Night's Dream", "Shakespeare: Othello", "Shakespeare: Romeo and Juliet", "Shakespeare: The Tempest"
    };

    private static final int NUMFILES = filenames.length;


    public static void main(String[] args)
    {
        ArrayList<FrequencyMap> maps = new ArrayList<>();

        for(String filename : filenames)
        {
            maps.add(frequence(filename));
        }

        FrequencyMap total = new FrequencyMap();

        for(FrequencyMap map : maps)
        {
            total = FrequencyMap.combine(map, total);
        }

        Map<String, Double> sorted = sortByValue(total.getFreqMap());

        System.out.println(sorted.keySet());

        List<HashMap<String, Double>> list = new ArrayList<>();

        for(FrequencyMap map : maps)
        {
            HashMap<String, Double> newMap = new HashMap<>();

            for(Object key : Arrays.copyOfRange(sorted.keySet().toArray(), 2, 99))
            {
                newMap.put((String)key,
                 ↪  (map.getFreqMap().get(key) == null) ? 0.0 :
                 ↪  map.getFreqMap().get(key));
            }

            list.add(normalize(newMap));
        }


        double[][] array = new double[100][NUMFILES];

        int i = 0;
        int j = 0;

        for(HashMap<String, Double> map : list)
        {
            for(Double value: map.values())
            {
                array[i][j] = value;
                ++i;
            }

            i = 0;
            ++j;
        }

        Matrix m = new Matrix(array);
        Matrix v = m.svd().getV();
        Matrix s = m.svd().getS();

        Matrix fin = s.times(v.transpose());

        x = fin.getMatrix(0, 0, 0, NUMFILES - 1);
        y = fin.getMatrix(1, 1, 0, NUMFILES - 1);

        ExampleChart<XYChart> scatter = new ScatterChart();
        XYChart chart = scatter.getChart();
        new SwingWrapper<>(chart).displayChart();

        try
        {
            BitmapEncoder.saveBitmap(chart, "output",
             ↪  BitmapEncoder.BitmapFormat.PNG);
        }
        catch(Exception e)
        {
            System.out.println("Something went wrong!");
        }
    }

    private static HashMap<String, Double>
     ↪  normalize(HashMap<String, Double> map)
    {
        Double total = Double.valueOf(0);
        HashMap<String, Double> frequencies = new HashMap<>();

        for(Double l : map.values())
        {
            total += l;
        }

        for(String e : map.keySet())
        {
            frequencies.put(e,
             ↪  Double.valueOf(map.get(e)).doubleValue()/total.doubleValue());
        }

        return frequencies;
    }

    private static String readFileAsString(String filename)
    {
        try
        {
            String original = new
             ↪  String(Files.readAllBytes(Paths.get(filename)));
            String noQuotes = original.replace("\"", "");
            String noApost = noQuotes.replace("\'", "");
            String none = noApost.replace("`", "");
            return none;
        }
        catch (IOException e)
        {
            return null;
        }
    }

    private static FrequencyMap frequence(String filename)
    {
        FrequencyMap freq = new FrequencyMap();
        String file = readFileAsString(filename).toLowerCase();

```

```java
        for(String word :
         ↪  Arrays.asList(file.split("[\\p{Punct}\\s]+")))
        {
            freq.addWord(word);
        }

        freq.normalize();

        return freq;
    }

    private static class ScatterChart implements
     ↪  ExampleChart<XYChart>
    {
        @Override
        public XYChart getChart()
        {
            // Create Chart
            XYChart chart = new
             ↪  XYChartBuilder().width(1000).height(800).build();

            // Customize Chart

             ↪  chart.getStyler().setDefaultSeriesRenderStyle(XYSeriesRenderStyle.Scatter);
            chart.getStyler().setChartTitleVisible(true);
            chart.getStyler().setMarkerSize(12);
            chart.getStyler().setHasAnnotations(true);

            // Series
            List<Double> xData =
             ↪  Arrays.stream(x.getArray()[0]).boxed().collect(Collectors.toList());
            List<Double> yData =
             ↪  Arrays.stream(y.getArray()[0]).boxed().collect(Collectors.toList());

            String titles[] = niceNames;

            for(int i = 0; i < 7; ++i)
            {
                XYSeries a = chart.addSeries(titles[i],
                 ↪  Arrays.asList(xData.get(i)),
                 ↪  Arrays.asList(yData.get(i)));
                a.setMarkerColor(Color.ORANGE);
            }

            for(int i = 7; i < 14; ++i)
            {
```

```java
            XYSeries a = chart.addSeries(titles[i],
             ↪  Arrays.asList(xData.get(i)),
             ↪  Arrays.asList(yData.get(i)));
            a.setMarkerColor(Color.BLUE);
        }

        for(int i = 14; i < 18; ++i)
        {
            XYSeries a = chart.addSeries(titles[i],
             ↪  Arrays.asList(xData.get(i)),
             ↪  Arrays.asList(yData.get(i)));
            a.setMarkerColor(Color.GREEN);
        }

        for(int i = 18; i < 23; ++i)
        {
            XYSeries a = chart.addSeries(titles[i],
             ↪  Arrays.asList(xData.get(i)),
             ↪  Arrays.asList(yData.get(i)));
            a.setMarkerColor(Color.RED);
        }

        for(int i = 23; i < 29; ++i)
        {
            XYSeries a = chart.addSeries(titles[i],
             ↪  Arrays.asList(xData.get(i)),
             ↪  Arrays.asList(yData.get(i)));
            a.setMarkerColor(Color.CYAN);
        }

        //for(int i = 29; i < 35; ++i)
        //{
        //    XYSeries a = chart.addSeries(titles[i],
         ↪  Arrays.asList(xData.get(i)),
         ↪  Arrays.asList(yData.get(i)));
        //    a.setMarkerColor(Color.PINK);
        //}

        return chart;
    }
}

// Allows for the sorting of a map by value
public static <K, V extends Comparable<? super V>> Map<K, V>
 ↪  sortByValue(Map<K, V> map)
{
        List<Map.Entry<K, V>> list = new LinkedList<>(map.entrySet());
        Collections.sort(list, (e1, e2) -> -(e1.getValue()).compareTo(e2.getValue()));
        Map<K, V> result = new LinkedHashMap<>();
        for (Map.Entry<K, V> entry : list)
        {
            result.put(entry.getKey(), entry.getValue());
        }

        return result;
    }
}

// NEW FILE BELOW

import java.util.Map;
import java.util.HashMap;

public class FrequencyMap
{
    /**
     * The main data structure of the class; maps Strings to Integers.
     */
    private Map<String, Double> freqMap;

    /**
     * The constructor for the class; simply instantiates freqMap.
     */
    public FrequencyMap()
    {
        freqMap = new HashMap<>();
    }

    /**
     * The addWord() method allows for elements to be added to freqMap. The method utilized .merge() from Map, allowing
     * for the given key to exist and not exist.
     *
     * @param word - the word to be added
     * @return the updated FrequencyMap
     */
    public void addWord(String word)
    {
        freqMap.merge(word, 1.0, Double::sum);
    }

    public void normalize()
    {
        int size = freqMap.size();

        for(String key : freqMap.keySet())
        {
            freqMap.put(key, freqMap.get(key) / size);
        }
    }

    public static FrequencyMap combine(FrequencyMap one,
     ↪  FrequencyMap two)
    {
        one.freqMap.forEach((k, v) -> two.freqMap.putIfAbsent(k,
         ↪  v));

        return two;
    }

    public Map<String, Double> getFreqMap()
    {
        return freqMap;
    }
}
```