Natural Language Document Classification of Philosophy Texts based on High-Level Qualitative Labels

Trent Yarosevich[1]

Abstract
This paper uses document classification with a sample of academic philosophy papers with the aim of exploring whether these techniques are effective at identifying documents based on extremely high-level, qualitative labels that cover extremely large divisions within the philosophy community. In particular, the presumably easier task of distinguishing Continental and Analytic philosophy papers, which draw from very distinct lexicons, is considered first. Following this, we then try to distinguish 'good' from 'bad' philosophy papers, using samples drawn from some of the top Philosophy journals according to google scholar's citation analytics, which are labeled as 'good', and comparing them with papers drawn from graduate and undergraduate journals, which are labeled as 'bad'.
https://github.com/tyarosevich/AMATH_582/blob/master/Project/yarosevich582_project_final.pdf

[1] *Department of Applied Mathematics, University of Washington, Seattle*

Contents

1. Introduction and Overview

Natural language processing (NLP) and machine-learning (ML) have become prevalent topics as algorithms and processing power have become adequate to the extremely complex task of analyzing written documents with thousans of different words. The simple size of the vocabulary is just a small part of the information contained in language, however; it also encodes concepts that are so far-reaching and abstract that even humans can struggle to rigorously define them. Philosophy has, among other things, historically concerned itself with rigorously defining concepts - from Plato's allegory of the cave, to Hegel's introduction to *The Science of Logic*, to Gilles Deleuze's claim that philosophy *is* the study of the concept of concepts[1]. This is by no means an overture to a philosophical investigation, but merely serves as an observation that Philosophy uses language in a very abstracted way that makes use of rich syntactical information.

Classification of documents with NLP, on the other hand, often just considers individual words based on their frequency. This is called the 'bag-of-words' model, and this is the approach I will use here. The question is, can NLP with the bag-of-words approach classify texts whose labels are determined by extremely complex, abstract philosophical categories? To determine this, I will evaluate journal papers in two parts, one that will attempt to distinguish papers from two extremely broad fields within philosophy, and one that will attempt to distinguish between impactful papers that are widely read within the field, and papers that are written by students.

Before moving on to discuss the data acquisition and algorithms, a brief note is in order regarding the two broad fields in question: 'continental' and 'analytic' philosophy. Volumes have been written on what separates these two fields, but it makes for an interesting machine-learning task since they both have exlusive figures, as well as ones that overlap; exlusive lexicons of words, as well as common argumentative language; and exlusive topics as well as differing approaches to the same topics. I, of course, will not engage in any such comparison, but rather simply observe that even within Philosophy, the so-called 'continental-analytic divide' is both widely recognized and debated, and thus the question of whether or not they can be classified with machine-learning is at least

[1] Deleuze, Gilles, et al. What Is Philosophy? Verso, 2015.

interesting, regardless of whether it is a success or failure.

2. Theoretical Background

2.1 Bag-of-words and Text Vectorization

A fair amount of pre-processing was required for this investigation, but the most important overall aspect is the conversion of a text into numbers that faithfully represent that text so that they can be processed by a computer. There are many ways to do this, but the approach used here is called the 'bag-of-words' method, so-called since the arrangement of the individual words is discarded, and their frequency within each document is the only thing that matters. Thus in the preprocessing stage, the text must be reduced to a list of each word, and then a dictionary of all the unique words that occur in all the documents is formed. Each document is then represented by a word-count. For example, suppose we have three documents: "if, then" , "if, or", "if, else". The resulting bag of words would look like this:

	"if"	"then"	"else"	"or"
Document 1	1	1	0	0
Document 2	1	0	0	1
Document 3	1	0	1	0

Table 1. An example bag-of-words matrix

The bag-of-words approach has a few issues, however, that need to be addressed by data-preprocessing, or 'text-cleaning'. One is the presence of 'noise' in the data in the form of what are called stop-words, or words that are so common and necessary for everyday language that they carry no information about the classification of the document. These words, such as 'the' and 'and', can simply be removed from the corpus. Another problem is term-frequency (TF). If word counts alone were used, the very common terms would overshadow the interesting ones that actually characterize the class to which the document belongs, that is to say their term-frequency would be very high. If a table like Table 1 is indexed as a matrix, then each term-frequency would be:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}} \tag{1}$$

To avoid this problem, we use a method known as "inverse-document frequency," which is a weighting adjustment applied to the term-frequencies to give the TF-IDF vectors. This weighting is calculated as follows, where n is the number of documents and $df(t)$ is the number of documents containing the particular term[2]:

$$idf(t) = \log \frac{1+n}{1+df(t)} + 1 \tag{2}$$

These TF-IDF values are then normalized with a Euclidean norm, and the resulting data matrix is then the set of features that will be passed to a classifier.

[2] https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

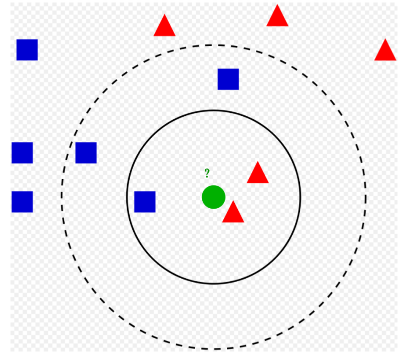

Figure 1. KNN example in which the green dot is new data, and the squares and triangles are two classes in the training set. The inner circle represents $n = 3$, and the outer $n = 5$.

2.2 Classifiers - K-Nearest Neighbors and Discriminant Analysis

The results of pre-processing will be classified with three methods, K-Nearest Neighbors (KNN), Linear/Quadratic Discriminant Analysis (LDA/QDA), and Support Vector Classification (SVC). The latter will be treated as a black-box and will not be discussed here as it is simply presented as another point of comparison for accuracy. KNN and LDA/QDA will be briefly discussed below.

2.2.1 K-Nearest Neighbors

The KNN approach is a very simple, naive supervised machine-learning method that requires labeled data. This algorithm's approach is simple: a training set of labeled data is used to evaluate any test data by finding the distance between each point in the new data-set and the labelled points in the training data-set. The distance d is defined as follows, in which Δx_n represents the difference in each direction between the test point and a particular neighbor:

$$d = \sqrt{\Delta x_1 + \Delta x_2 + ... + \Delta x_n} \tag{3}$$

The test data point is then given the same label as its nearest training data point. This concept can also be extended by increasing the number of neighbors used to label the test data-point, and thus an odd number of neighbors (to avoid ties) can be evaluated to determined the label of the new point. This means that, if $n = 3$, the nearest neighbor might actually be ignored if the next two nearest neighbors belong a different class. Figure 1 provides a simple illustration of the algorithm[3].

2.2.2 LDA and QDA

The other classifying tool I'll discuss is LDA, which largely hinges upon the singular value decomposition, or SVD. The SVD is discussed widely in many textbooks and online resources, so for the sake of brevity I will not discuss it here, and instead focus on the approach taken by discriminant analysis simply noting that the decomposition of a data matrix A

[3] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#/media/File:KnnClassification.svg

has the following notation:

$$A = U\Sigma V^* \tag{4}$$

. With this approach, the orthogonal bases of the SVD are studied in order to try to understand what the data does when projected onto these bases, which eliminates their redundancy (co-variance). In essence, the LDA proceeds by examining the principal orthogonal modes, or POD, which are obtained from the SVD. We can then study how strong these modes are in the data set by looking at a particular sample's projection onto ΣV^*. The hope is then that different datapoints will have common features in a particular mode that can then be used to distinguish them from one another and thereby classify them.

Once a suitable mode is found that distinguishes labeled classes of data, linear discriminant analysis is used to project the data onto a subspace that both (a) maximizes the distance between the labeled classes of data, (b) minimizes the intra-class distance between points within a single labeled class. This is an optimization problem in which the between and within class matrices are given by:

$$\begin{aligned} S_B &= (\mu_2 - \mu_1)(\mu_2 - \mu_1)^T \\ S_W &= \sum_{j=1}^{2} \sum_x (x - \mu_j)(x - \mu_j)^T \end{aligned} \tag{5}$$

These values are then used for the optimzation problem:

$$w = argmax_w \frac{w^T S_B w}{w^T S_W w} \tag{6}$$

Finally, this value is used to solved the eigenvalue problem $S_B w = \lambda S_W w$ in which λ gives the mode of interest and the associated eigenvector is the projection basis. In the subsequent analysis, this method will be handled with built-in methods of the sklearn library.

3. Algorithm Implementation and Development

3.1 Data Selection

The data was downloaded manually, but in a more scaled-up version of this experiment, a web-scraper would be used, and so I will include the details here. The most important aspect of this is choosing a criteria that corresponds to the data labels, which can be difficult when the labels represent such high-level conceptual information - and by 'high-level' I mean the labels represent categories of philosophy that encompass a large number of other categories, and so on.

The principal tool used to select relevant papers published by active academics was the Google Scholar Top Publications resource [4]. To select journals that were explicitly 'continental' or 'analytic', I relied upon a combination of self-identification (some journals explicitly identify themselves), the Leiter Report [5] (a listing of departments and journals principally voted upon by scholars in the field), and personal experience from presenting at continental philosophy conferences myself. In the second part, journals were taken exlusively from the top 10 of the Google Scholar Top Publications in the subject of philosophy, ostensibly to ensure that these are 'good' papers, which is short-hand for 'papers with a relatively large citation impact'. All 'bad' papers were taken from graduate and undergraduate philosophy journals, and it should be noted that authors of exceptional graduate papers are *actively* discourage from publishing in such journals, as many scholars in the field believe publishing in such un-reputable format can have a negative impact on academic success[6]. The journals used were:

Continental: Philosophy and Phenomenological Research, European Journal of Philosophy, Continental Philosophy Review.
Analytic: Mind, Nous, The Philosophical Review, Analysis, The Journal of Philosophy.
Good: Mind, Philosophy and Phenomenological Research, The Philosophical Review, the APA journal, The Australasion Journal of Philosophy.
Bad: Episteme, Dianoia, Sapere Aude, Stance, Aporia, Ergo.

3.2 Data Pre-processing

Since the bag-of-words approach must turn a text into numerical representations of term-frequency, a number of processing steps are required.

(i): The pdf files are read in and coverted to strings using the python library *pdfminer*. While slow, this package is extremely good at extracting all the text from a .pdf file without errors. In every case the first page was omitted, since this is nearly always a title page including noisy data.

(ii): The data must be stemmed (which reduces similar words to a common stem) or lemmatized (which reduces words to their lemma based on a dictionary). Both were used, but lemmatization gave adequate results, and yielded more intellgible n-grams, since stemming tools often produce non-existent stems. Both processes were accomplished with the Natural Language Tool-kit for python, or *nltk*.

(iii): All numbers and names were removed from the data, as this information would generally either be noise, or be a source of unwanted correlation. For example, in part 2, the journals often have information regarding dates in the footer or header of each page, whereas the student journals usually did not. Left in the data, this provided strong correlation with the mere presence of numbers, which showed up in the n-grams. Names were removed using *nltk*, and numbers were removed using a substitution function (numbers were substituted with empty strings).

[4]"Google Scholar Top Publications." `Google.com`, `scholar.google.com/citations?view_op=top_venues&hl=en&vq=hum`.

[5]"Leiter Reports: A Philosophy Blog." Leiter Reports: A Philosophy Blog, leiterreports.typepad.com/.

[6]ibid.

(iv): Stop words, which are common words that don't convey any particular information, were removed using a built-in stop-word dictionary in the Sci-kit Learn python library, or *sklearn*. This dictionary had to be appened, however, with quite a few additional terms, again from the header and footer of the files. These terms created unwanted correlation, as mentioned above, since words like the journal names appear repeatedly in the training set. For this reason, an additional word list was created that removes all words from the headers and footers of each documents. This list includes things like journal names, URL prefixes, and months.
(v): Next, the cleaned text is divided into a training set, and a testing set for cross-validation, with a ratio of 85/15 for train/test.
vi: TF-IDF vectorization is then done using *sklearn*. The number of features used was varied to evaluate effectiveness. The options following options were tuned for accuracy, which will be discussed in the conclusion: max features, n-gram range, and max-df.

3.3 Classification

Compared to the pre-processing, classification was straightforward. The following classifiers are created using the *sklearn* package, trained, and then used on the test sets for cross-validation: K-nearest neighbors, Linear Discriminant Analysis, Quadratic Discriminant Analysis, and Support Vector Classification. Default options were used, and the n value in KNN was tuned for accuracy.

3.4 Visualization

Finally, the training and test sets were concatenated and decomposed using SVD to plot the singular values. For both data sets, a rank-2 representation of the data was plotted using the labels to give a sense of the distribution of the data.

4. Computational Results

The results from the experiment were excellent for KNN, with generally good results among the other methods. These values are collected in Figure 2. Singular values of the vectorized data show dominance of the first value, with a considerably long tail, shown in Figures 3 and 4. Scatter plots of the first two principal components show good distinction between the labeled classes, even with a rank-2 representation of the data, shown in Figures 5 and 6. Finally Table 2 shows the most correlated n-grams. This data was frustrating as I was not able to get the *chi2()* function to properly extract the class-specific n-grams, and so it must be noted that these n-grams are drawn from the entire corpus. Nevertheless, they were an important tool for identifying noise in the data.

5. Summary and Conclusions

The results suggest that we can classify philosophy texts based on high-level labels, in some cases with exceptionally high accuracy, both with KNN and SVC(in which the continental/analytic classification was perfect). This level of

	KNN	LDA	QDA	SVC
Part 1	100 / 96.7	80.8 / 96.7	100 / 50	100 / 100
Part 2	82.9 / 93.3	79.4 / 86.7	100 / 46.7	100 / 86.7

Figure 2. Classification results. Entries correspond to train/test accuracy.

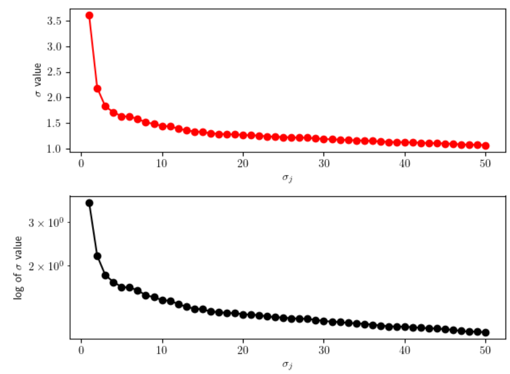

Figure 3. Continental v. Analytic, first 50 singular values.

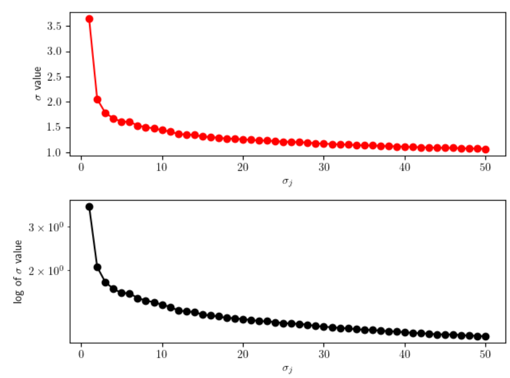

Figure 4. 'Good' v. 'Bad', first 50 singular values.

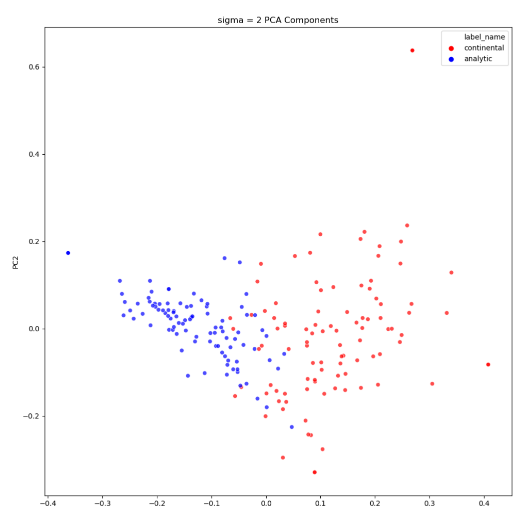

Figure 5. Part 1 $\sigma = 2$ Principal Components.

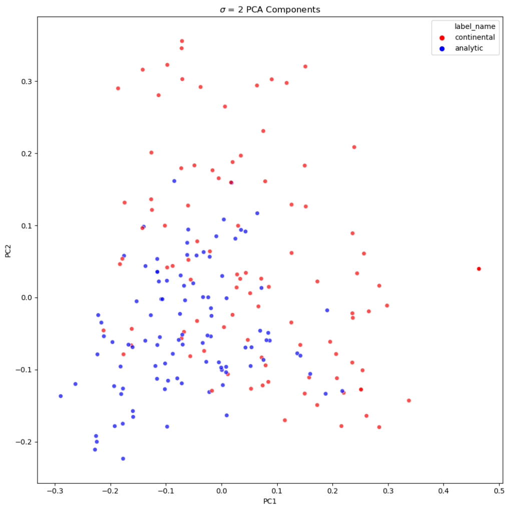

Figure 6. Part 1 $\sigma = 2$ Principal Components.

	Part 1	Part 2
Unigrams	operative first fission tence	luck compatibilists different compatibilist determinism
Bigrams	true context palm tree account moral term body doxastic obligation	acprof oso subject object theory The epistemic reason classical logic

Table 2. Most correlated n-grams.

accuracy requires expert knowledge in a human, and so this result was quite surprising.

Parameter tuning had a significant impact on the accuracy of the results. Specifically, choosing a large number of features, 2000, showed Significant increases in accuracy over an initial trial of 500. Another impactful parameter was max-df, which discards any feature that appears in more than x documents. An initial value of 10 provided good results, but increasing it to 25 improved them substantially, likely because it included more unusual words that correlated strongly with the types of language used by the labeled groups. Finally, in the KNN classification, values of n were varied from 1 to 11. In part 1, a single neighbor provided optimal results, whereas in part 2 a large number of neighbors, 11, worked best.

The biggest issue with the experiment was the isolation of correlated n-grams. As someone with a significant philosophy background, this represented one of the most potentially interesting parts of the entire experiment, as it would have provided some sort of answer to the question: what type of language do the most impactful philosophers use? Future experimentation will focus on clearing up this bug in the code and trying to give concrete n-gram information.

Appendix 1

np.linalg.**svd**(A): Performs a singular value decomposition on the argument matrix.

pdfminder.converter.TextConverter(rsrcmgr, retstr): Creates an object to convert a pdf into a string. The standard resource manager is declared with PDFResourceManager(), and the standard string processor is declared with io.StringIO(). Full documentation for pdfminer.six can be found at: https://pdfminersix.readthedocs.io/en/latest/.

os.walk(**dir**): Returns a string containing the root directory, a list containing the sub-directories as strings, and a list containing the files in the subdirectories as strings. Extremely useful for iterating over folder contents.
nltk.SnowballStemmer("english") and WordNetLemmatizer(): Both objects are used analogously. Declares a stemming/lemmitizing object using the natural language toolkit. A single word can be passed as a string using stemmer.**stem**(), and the object returns the stemmed or lemmatized word.

nltk.word_tokenize(string): Tokenizes the passed string into a list of the individual words contained in the string. Important for stemming and lemmatization.

nltk.tag.pos_tag(): A function from the nltk that tags words based on types such as 'proper nouns'. Useful for processing strings based on word type. In particular it was used to remove all words with the tags for proper nouns and names.

pd.DataFrame(dictionary): One of many methods for creating a pandas data frame object. In this case, a dictionary is passed and the data frame columns correspond to the dictionary values, with the keys serving as column titles.

TfidVectorizer(): Creates an object to vectorize lists of strings and return inverse term-frequency data. A full discussion of the options can be found at https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html.

KNeighborsClassifier(n_neighbors = n): Creates an object to classify data using the k-nearest neighbors method. Data is fitted and predicted using classifier.fit() and classifier.predict(). All the classifiers used have analogous syntax.

(accuracy_score(labels, prediction): Returns the accuracy as a float from the prediction of an already trained classifier object. The 'prediction' parameter is the output of the classifier.predict() method.
PCA(n_components = n): Returns an object that performs a singular value decomposition and returns the *n* principal components as a numpy array. The class method is used as follows: PCA(n_components = n).fit_transform(matrix).

Appendix 2

```
# Journal list:
# continental: philosophy and phenomenological research, european journal of philosophy, continental
    philosophy review
# analytic: mind, nous, the philosophical review, analysis, the journal of philosophy,
# good: mind, philosophy and phenomenological research, the philosophical review, the APA journal, the
    australasion journal of phil,
# bad: Episteme, dianoia, sapere aude, stance, aporia, ergo

#%% Imports
import io
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from pdfminer.pdfpage import PDFPage
import os
import pickle
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
```

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import chi2
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

#%% Functions for the project

# Perform a reduced SVD of the data for Part 1 and plot the singular values on a standard and semi-log axis.
def svd_plot(data):
    A1 = data

    U, S, V = np.linalg.svd(A1, full_matrices=False)
    x = np.linspace(1, 50, 50)

    # Plots the first 50 singular values.
    fig, (ax1, ax2) = plt.subplots(2)
    ax1.plot(x, S[0:50], 'r-o')
    ax1.set_xlabel('$\sigma_j$')
    ax1.set_ylabel('$\sigma$ value')

    ax2.semilogy(x, S[0:50], 'k-o', )
    plt.rc('text', usetex=True)
    ax2.set_xlabel('$\sigma_j$')
    ax2.set_ylabel('log of $\sigma$ value')
    plt.show()

# Function that takes the path of a pdf and returns it as a string object.
def convert_pdf_to_txt(path):
    rsrcmgr = PDFResourceManager()
    retstr = io.StringIO()
    codec = 'utf-8'
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr, codec=codec, laparams=laparams)
    fp = open(path, 'rb')
    interpreter = PDFPageInterpreter(rsrcmgr, device)
    password = ""
    maxpages = 0
    caching = True
    pagenos=set()

    # Skips the first page of the pdf file since it contains data that can be considered noise.
    pages = PDFPage.get_pages(fp, pagenos, maxpages=maxpages,password=password,caching=caching,
        check_extractable=True)
    iter_pages = iter(pages)
    next(iter_pages)
    for page in iter_pages:
        test = page
        interpreter.process_page(page)

    text = retstr.getvalue()
    fp.close()
    device.close()
    retstr.close()
    return text

# A function that iterates through a given root directory and extracts all pdf files as a list of strings.
def pdf_to_strings(path):
    rootdir = path
    corpus_raw = []
    index = 0
    label_list = []

    # Iterates through the base folder with os.walk, which goes through each subfolder, with
    # iterate variables for the files, the subdirectories and the directories.
    for subdir, dirs, files in os.walk(rootdir):

```

```
        # Iterates through each file.
        for file in files:

            # Gets the file path to pass to the extraction function.
            file_path = os.path.join(subdir, file)
            # Appends the name of the current folder, which is the label for the pdf
            label_list.append(os.path.basename(os.path.normpath(subdir)))
            corpus_raw.append(convert_pdf_to_txt(file_path))
    return corpus_raw, label_list;

# Function to generate all the raw lists of strings for each pdf. Significant Processing time.
def process_paths(path1, path2):
    corpus1, label_list1  = pdf_to_strings(path1)
    corpus2, label_list2  = pdf_to_strings(path2)
    return corpus1, label_list1, corpus2, label_list2

# A function to stem a list of strings composed of pdfs. Does so iteratively, then joins them back
# into single strings and returns a list of strings like the argument.
def list_stemmer(doc_list):
    n = len(doc_list)
    stemmed_text_list = []

    # Initialize the stemmer
    stemmer = nltk.SnowballStemmer(”english”)

    for i in range(0, n):
        # According to stackexchange discussions, a list comprehension is much faster for this task
        # than a loop.
        stemmed_text = ’ ’.join(stemmer.stem(token) for token in nltk.word_tokenize(doc_list[i]))
        stemmed_text_list.append(stemmed_text)
    return stemmed_text_list

def list_lemmatizer(doc_list):
    from nltk.stem import WordNetLemmatizer
    n = len(doc_list)
    lemmad_text_list = []

    # Initialize the lemmatizer
    lemmatizer = WordNetLemmatizer()

    for i in range(0, n):
        # According to stackexchange discussions, a list comprehension is much faster for this task
        # than a loop.
        lemmad_text = ’ ’.join(lemmatizer.lemmatize(token) for token in nltk.word_tokenize(doc_list[i]))
        lemmad_text_list.append(lemmad_text)
    return lemmad_text_list

# Removes numbers from the argument, which should be a list of strings.
def num_removal(doc_list):
    for i, list in enumerate(doc_list):
        doc_list[i] = re.sub(r’\d+’, ’’, list)

# Removes most names from the argument, which should be a list of strings
def name_removal(doc_list):
    for i, list in enumerate(doc_list):
        tagged_string = nltk.tag.pos_tag(list.split())
        new_string = [word for word,tag in tagged_string if tag != ’NNP’ and tag != ’NNPS’]
        doc_list[i] = ’ ’.join(new_string)

# Load the desired data. Arguments are the path and whether raw, stemmed, or lemmatized data
# Is desired. Returns part1_data, part1_labels, part2_data, part2_labels. Code has no error
# control, type should be ”lemmed”, ”stemmed” or ”raw
def load_data(path, type):
    import pickle
    with open(path + ”/part1_data_” + type, ’rb’) as f:
        part1_data = pickle.load(f)

    with open(path + ”/part1_labels”, ’rb’) as f:
        part1_labels = pickle.load(f)

```

```
    with open(path + "/part2_data_" + type, 'rb') as f:
        part2_data = pickle.load(f)

    with open(path + "/part2_labels", 'rb') as f:
        part2_labels = pickle.load(f)
    return part1_data, part1_labels, part2_data, part2_labels;

# Save files based on type. Again, no error control, same type as the load function
# must be used.
def save_data(path, type, corpus1, label_list1, corpus2, label_list2):
    with open(path + "/part1_data_" + type, 'rb') as f:
        pickle.dump(corpus1, f)

    with open(path + "/part1_labels", 'rb') as f:
        pickle.dump(label_list1, f)

    with open(path + "/part2_data_" + type, 'rb') as f:
        pickle.dump(corpus2, f)

    with open(path + "/part2_labels", 'rb') as f:
        pickle.dump(label_list2, f)

#%% Download nltk resources and load the various functions written for the project from file. Only really
# needs to be run once.
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

#%% Get the raw strings and pickle the data
path1 = "C:/Users/zennsunni/Dropbox/School Stuff/Winter 2020/AMATH_582/Project/part1"
path2 = "C:/Users/zennsunni/Dropbox/School Stuff/Winter 2020/AMATH_582/Project/part2"
corpus1, label_list1, corpus2, label_list2 = process_paths(path1, path2)

path = "C:/Users/zennsunni/Dropbox/School Stuff/Winter 2020/AMATH_582/Project"
type = "raw"
save_data(path, type, corpus1, label_list1, corpus2, label_list2)

#%% Stem the data and save these files separately

corpus1 = list_stemmer(part1_data_raw)
corpus2 = list_stemmer(part2_data_raw)

save_data(path, type, corpus1, label_list1, corpus2, label_list2)

#%% Lemmatize the data and save these files separately

corpus1 = list_lemmatizer(part1_data_raw)
corpus2 = list_lemmatizer(part2_data_raw)

save_data(path, type, corpus1, label_list1, corpus2, label_list2)

#%% Load the desired data
path = "C:/Users/zennsunni/Dropbox/School Stuff/Winter 2020/AMATH_582/Project"
type = "lemmed"
part1_data, part1_labels, part2_data, part2_labels = load_data(path, type)
#%% Remove numbers and most names from date before setup
num_removal(part1_data)
num_removal(part2_data)

name_removal(part1_data)
name_removal(part2_data)

#%% Setup and Classification
from sklearn.feature_extraction import text

data1 = part1_data
data2 = part2_data

```

```
# This dictionary gives the folder names as keys to the artists inside them.
label_index1 = {"continental": 0, "analytic":1}
label_index2 = {"good": 0, "bad": 1}

# Create numerical labels
part1_labels_int = [label_index1[q] for q in part1_labels]
part2_labels_int = [label_index2[q] for q in part2_labels]

# Put the data in a data frame
prep_dict1 = {'part1_data': data1, 'part1_labels':part1_labels, 'part1_id': part1_labels_int}
prep_dict2 = {'part2_data': data2, 'part2_labels':part2_labels, 'part2_id': part2_labels_int, }
df1 = pd.DataFrame(prep_dict1)
df2 = pd.DataFrame(prep_dict2)

# Split the corpus up into randomized training/testing sets
X_train1, X_test1, y_train1, y_test1 = train_test_split(df1['part1_data'], df1['part1_id'], test_size=0.15, random_state=8)
X_train2, X_test2, y_train2, y_test2 = train_test_split(df2['part2_data'], df2['part2_id'], test_size=0.15, random_state=8)

#%%

# Parameter selection for vectorization of the data. We will look at unigrams, bigrams, and trigrams. We will
# Also ignore words that appear in less than 5% of the documents. Max features is set very high, since a few odd words
# Might be very correllated with a particular label, and philosophy papers tend to use the full spectrum of the
# English Language. This might have to be tuned up to a very high number.
ngram_range = (1,2)
min_df = 1
max_df = 25
max_features = 2000

# Additional stop words, which get added to sklearn's stop word list.
stop_list = ["Authors", "Published", "Permissions", "journal", "journals", "reserved", "Inc", "Philosophical", "Review",
             "org", "This", "content", "downloaded", "https", "All", "Oxford", "New", "York", "university", "University", "Vol", "http", "doi", "Press",
             "And", "org", "UTC", "Mar", "ed", "downloaded", "Sun", "Mon", "Tues", "Wed", "Thur", "Fri", "jstor", "Cornell",
             "DOI", "European", "Continental", "Springer", "Sons", "Wiley", "wileyonlinelibrary","January", "February",
             "March", "April", "May", "June", "July", "August", "September", "October", "November", "December", "Stance",
             "Dianoia", "Undergraduate", "Boston", "College", "Aporia", "Ergo", "no." "no", "Phenomenological", "Research",
             "LLC", "Australasian", "Routledge", "Association", "pp", "In", "html"]
stop_words = text.ENGLISH_STOP_WORDS.union(stop_list)

# Declare the vectorizer. Note this is a TF-ID vectorizer, and thus weights low-frequency words,
# which is perfect for analyzing the very tribal lexicons used by philosophers.
tfidf = TfidfVectorizer(encoding='utf-8',
                        ngram_range=ngram_range,
                        stop_words= stop_words,
                        lowercase=False,
                        max_df=max_df,
                        min_df=min_df,
                        max_features=max_features,
                        norm='l2',
                        sublinear_tf=True)

# Extract the features for the training and test sets for Part 1.
features_train1 = tfidf.fit_transform(X_train1).toarray()
labels_train1 = y_train1
features_test1 = tfidf.transform(X_test1).toarray()
labels_test1 = y_test1

# Extract the features for the training and test sets for Part 2.
features_train2 = tfidf.fit_transform(X_train2).toarray()
```

```python
labels_train2 = y_train2
features_test2 = tfidf.transform(X_test2).toarray()
labels_test2 = y_test2

#%% Plot the singular values

features_full1 = np.concatenate((features_train1,features_test1), axis=0)
features_full2 = np.concatenate((features_train2,features_test2), axis=0)
#%%
svd_plot(features_full2)


#%% Prints the most correlated uni/bi/trigrams

from sklearn.feature_selection import chi2

for field, index in sorted(label_index1.items()):
    features_chi2 = chi2(features_train1, labels_train1 == index)
    indices = np.argsort(features_chi2[0])
    feature_names = np.array(tfidf.get_feature_names())[indices]
    unigrams = [v for v in feature_names if len(v.split(' ')) == 1]
    bigrams = [v for v in feature_names if len(v.split(' ')) == 2]
    # trigrams = [v for v in feature_names if len(v.split(' ')) == 3]
    print("# '{}' category:".format(field))
    print("  . ### Most correlated unigrams:\n. {}".format('\n. '.join(unigrams[-5:])))
    print("  . ### Most correlated bigrams:\n. {}".format('\n. '.join(bigrams[-5:])))
    # print("  . ### Most correlated trigrams:\n. {}".format('\n. '.join(trigrams[-5:])))
    print("")

#%% Classify with KNN, Part 1
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

neigh = KNeighborsClassifier(n_neighbors=1)
neigh.fit(features_train1, labels_train1)
predict1 = neigh.predict(features_test1)

# Training accuracy
print("The training accuracy is: ")
print(accuracy_score(labels_train1, neigh.predict(features_train1)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test1, predict1))

#%% Classify with KNN, Part 2
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

neigh = KNeighborsClassifier(n_neighbors=9)
neigh.fit(features_train2, labels_train2)
predict2 = neigh.predict(features_test2)

# Training accuracy
print("The training accuracy is: ")
print(accuracy_score(labels_train2, neigh.predict(features_train2)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test2, predict2))

#%% SVC Fit for comparison, Part1
from sklearn import svm
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

svm_clf = svm.SVC()
svm_clf.fit(features_train1, labels_train1)
predict1 = svm_clf.predict(features_test1)

# Training accuracy
```

```
print("The training accuracy is: ")
print(accuracy_score(labels_train1, svm_clf.predict(features_train1)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test1, predict1))

#%% SVC Fit for comparison, Part2
from sklearn import svm
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

svm_clf = svm.SVC()
svm_clf.fit(features_train2, labels_train2)
predict2 = svm_clf.predict(features_test2)

# Training accuracy
print("The training accuracy is: ")
print(accuracy_score(labels_train2, svm_clf.predict(features_train2)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test2, predict2))

#%% LDA for comparison Part 1
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA


lda = LDA(n_components=1)
lda.fit(features_train1, labels_train1)
predict1 = lda.predict(features_test1)

# Training accuracy
print("The training accuracy is: ")
print(accuracy_score(labels_train1, lda.predict(features_train1)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test1, predict1))

#%% LDA for comparison Part 2

lda = LDA(n_components=1)
lda.fit(features_train2, labels_train2)
predict2 = lda.predict(features_test2)

# Training accuracy
print("The training accuracy is: ")
print(accuracy_score(labels_train2, lda.predict(features_train2)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test2, predict2))

#%% QDA for comparison Part 1
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA

qda = QDA()
qda.fit(features_train1, labels_train1)
predict1 = qda.predict(features_test1)

# Training accuracy
print("The training accuracy is: ")
print(accuracy_score(labels_train1, qda.predict(features_train1)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test1, predict1))

#%% QDA for comparison Part 2

```

```

qda = QDA()
qda.fit(features_train2, labels_train2)
predict2 = qda.predict(features_test2)

# Training accuracy
print("The training accuracy is: ")
print(accuracy_score(labels_train2, qda.predict(features_train2)))

# Test accuracy
print("The test accuracy is: ")
print(accuracy_score(labels_test2, predict2))



#%% Visualize the features in a plot
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns

features = np.concatenate((features_train1,features_test1), axis=0)
labels = np.concatenate((labels_train1,labels_test1), axis=0)
title = "sigma = 2 PCA Components"
princ_comps = PCA(n_components=2).fit_transform(features)

# Put them into a dataframe
df_features = pd.DataFrame(data= princ_comps,
                           columns=['PC1', 'PC2'])

# Now we have to paste each row's label and its meaning
# Convert labels array to df
df_labels = pd.DataFrame(data=labels,
                         columns=['label'])

df_full = pd.concat([df_features, df_labels], axis=1)
df_full['label'] = df_full['label'].astype(str)

# Makes a new dictionary that is flipped, to unzip the label codes the other
# direction.
new_labels = {"0": "continental", "1": "analytic"}

# And map labels
df_full['label_name'] = df_full['label']
df_full = df_full.replace({'label_name': new_labels})

plt.figure(figsize=(10, 10))
sns.scatterplot(x='PC1',
                y='PC2',
                hue="label_name",
                data=df_full,
                palette=["red", "blue"],
                alpha=.7).set_title(title);

plt.savefig('part1_scatter.png', facecolor = "white")
plt.show()

#%% And for part 2

features = np.concatenate((features_train2,features_test2), axis=0)
labels = np.concatenate((labels_train2,labels_test2), axis=0)
title = "$\sigma$ = 2 PCA Components"
princ_comps = PCA(n_components=2).fit_transform(features)

# Put them into a dataframe
df_features = pd.DataFrame(data= princ_comps,
                           columns=['PC1', 'PC2'])

# Now we have to paste each row's label and its meaning
# Convert labels array to df
df_labels = pd.DataFrame(data=labels,
```

```
                                   columns=['label'])

df_full = pd.concat([df_features, df_labels], axis=1)
df_full['label'] = df_full['label'].astype(str)

# Makes a new dictionary that is flipped, to unzip the label codes the other
# direction.
new_labels = {"0": "continental", "1": "analytic"}

# And map labels
df_full['label_name'] = df_full['label']
df_full = df_full.replace({'label_name': new_labels})

plt.figure(figsize=(10, 10))
sns.scatterplot(x='PC1',
                y='PC2',
                hue="label_name",
                data=df_full,
                palette=["red", "blue"],
                alpha=.7).set_title(title);

plt.savefig('part2_scatter.png', facecolor = "white")
plt.show()
```