An Exploration into Natural Language Processing in Old English

Nathan Snyder

Introduction

Natural Language Processing (NLP) is an exciting field right now, but almost all of the research in this area is dedicated to the processing of modern languages. No new content is being created in dead languages, but there is still use for NLP techniques in the study of ancient and pre-modern languages. NLP can give insight into a language's morphology and syntax as well as aid in automated transcription and translation. The field of NLP combined with modern Machine Learning techniques such as the use of Artificial Neural Networks (ANNs) is very promising and could lead to the discovery of previously unknown patterns in language.

In Modern English, the use of prepackaged software libraries such as NLTK[1] is extremely common in performing NLP analyses, but hardly any of these resources exist for Old English. In fact, very little work in the area of NLP for pre-modern languages has been done before, the most notable of which being the Classical Language Toolkit (CLTK)[2], which aims to provide NLP for a wide range of pre-modern languages. Unfortunately, CLTK is mainly focused on the study of Latin and Greek, and their Old English material is sorely lacking. Other work has been done in the area of NLP for Old English, but these mainly use older statistical techniques to analyze a corpus. There are no public software libraries that offer NLP for Old English using ANNs.

The goal of this exploration was to use ANNs to run NLP analyses on Old English texts. Specifically, this paper will focus on lemmatization and part of speech tagging. Lemmatization is the process of transforming an inflected word into its lemma (also called its headword or its dictionary form). This technique is useful to run before other techniques, such as sentiment analysis, that focus on the semantic value of words instead of their grammatical function. Part-of-speech tagging is the process of identifying the part of speech of a word. This technique is useful for determining the relationships between words and can be used as input to other techniques, such as lemmatization, that can benefit from knowing the part of speech of a word. Both of these techniques fall under the broader category of morphological analysis, as they both aim to learn more about individual words while ignoring the other words in the corpus.

As a deviation from existing software libraries such as CLTK, no hard-coded rules were used in this exploration. Hard-coded rules are often useful for irregular forms in a language (e.g., English ‘be’, ‘am’, ‘is’, …), because these irregular forms do not follow the typical rules of the languages and are thus difficult for a neural network to learn without a lot of training. However, in this exploration, the only source of analysis came from the ANNs. The ultimate goal of this exploration was to see if a simple ANN model could be used for NLP for Old English.

Methods

For all of the analyses performed, data was collected from Dr. Peter Baker’s database of Old English texts. Additionally, all code that was written for this exploration is available in the GitHub repository[3]. The lemmatization script is in Lemmatization.py and the part of speech tagging script is in PartOfSpeechTagging.py. Both scripts use functions that are defined in Utils.py.

For lemmatization, the input data to the ANN was the spelling of a word as it appears in an Old English text and the output was the lemma/headword of that word. For part of speech tagging, the input data was a headword, and the output was its corresponding part of speech. For both of these analyses, the Python MySQL Connector module[4] was used to connect to the Old English database and execute queries that returned the desired columns of inputs and outputs. After retrieving the data, it was cleaned up by removing punctuation characters, replacing certain sequences of characters that had gotten encoded differently in the database with their original Old English equivalents, and turning all capital letters into their lowercase forms.

Then, each word in the dataset was turned into a vector. ANNs work on vectors, not strings, so each string had to be transformed into a vector, and each vector needed to have the same number of components. The number of components in every vector was equal to the length of the longest string in the data. The transformation was performed by finding the integer representation of each character in the string and assigning each integer to one component of the vector in the order of the original string. Any remaining components got filled with 0. The code for transforming a word string into a vector is shown in Code Snippet 1 below.

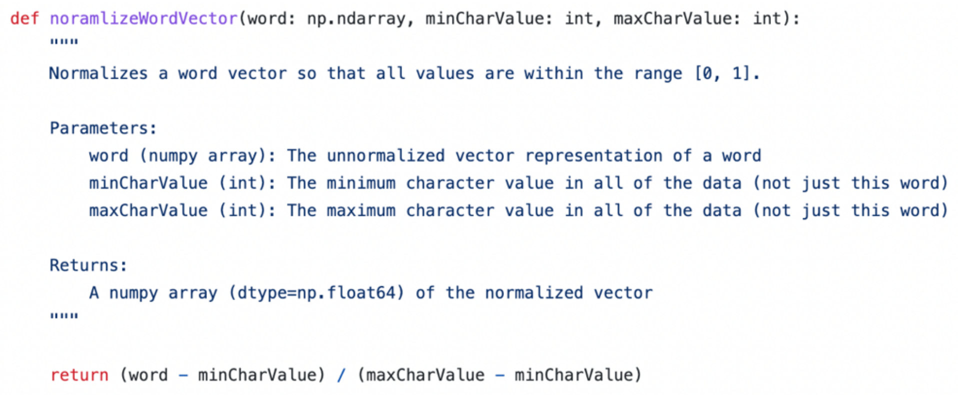
```python
def wordStringToVector(word: str, dim: int):
    """
    Returns the vector representation of a word.

    Parameters:
        word (str): The string representation of the word
        dim (int): The desired dimension of the resulting vector (dim >= len(word))

    Returns:
        vector (list): The vector of length `dim` as a list of ints
    """

    wordLength = len(word)

    if dim < wordLength:
        raise Exception("dim cannot be lower than the length of the word")

    vector = [0] * dim
    for i in range(wordLength):
        vector[i] = ord(word[i])

    return vector
```

Code Snippet 1: the wordStringToVector function from Utils.py

After the string was transformed into a vector, all components of the vector were normalized so that they fell in the range [0, 1]. This step was performed so that the model would perform independent of the range of a dataset's characters. The minimum and maximum values in all of the vectors in the dataset were found, and each component of each vector was normalized using Equation 1 below. The code for this step can be seen in Code Snippet 2 below.

$$x_{\text{norm}} = \frac{x - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

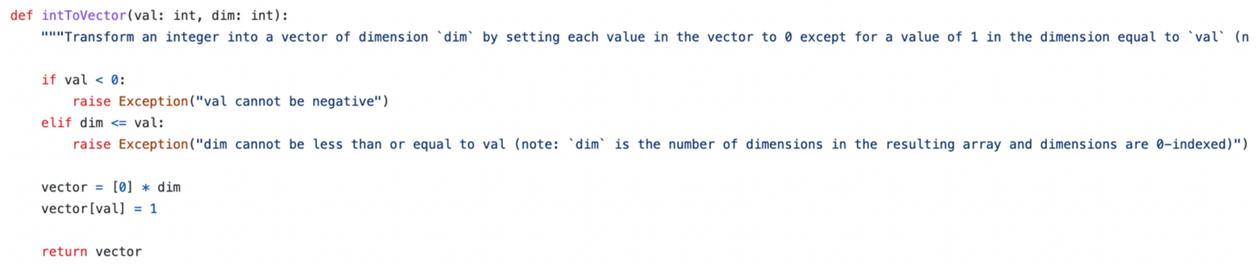
Equation 1: Normalization of the point *x* within the dataset *X*.



```python
def noramlizeWordVector(word: np.ndarray, minCharValue: int, maxCharValue: int):
    """
    Normalizes a word vector so that all values are within the range [0, 1].

    Parameters:
        word (numpy array): The unnormalized vector representation of a word
        minCharValue (int): The minimum character value in all of the data (not just this word)
        maxCharValue (int): The maximum character value in all of the data (not just this word)

    Returns:
        A numpy array (dtype=np.float64) of the normalized vector
    """

    return (word - minCharValue) / (maxCharValue - minCharValue)
```

Code Snippet 2: normalizeWordVector function from Utils.py

The parts of speech for the part of speech tagging were handled differently. Since there are a finite number of parts of speech, all parts of speech were gathered and numbered, and those numbers were represented as vectors. This vector transformation happened by creating a vector of dimension *n*, where *n* is the total number of parts of speech, and setting all vector components to 0 except for a value of 1 in the *i*th component, where *i* is the number of the part of speech. The code for the function that transforms these integers to vectors is available in Code Snippet 3 below. The parts of speech vectors did not have to be normalized, as their components were already in the range [0, 1].



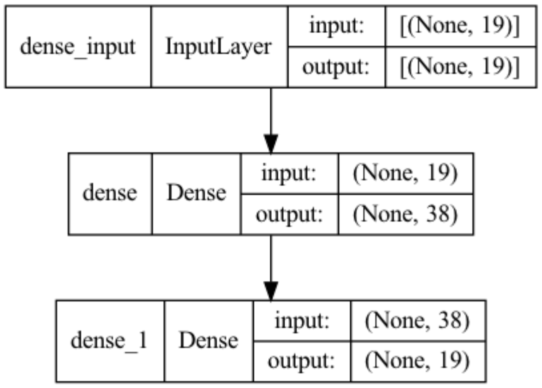
```python
def intToVector(val: int, dim: int):
    """Transform an integer into a vector of dimension `dim` by setting each value in the vector to 0 except for a value of 1 in the dimension equal to `val` (n

    if val < 0:
        raise Exception("val cannot be negative")
    elif dim <= val:
        raise Exception("dim cannot be less than or equal to val (note: `dim` is the number of dimensions in the resulting array and dimensions are 0-indexed)")

    vector = [0] * dim
    vector[val] = 1

    return vector
```

Code Snippet 3: the intToVector function in Utils.py

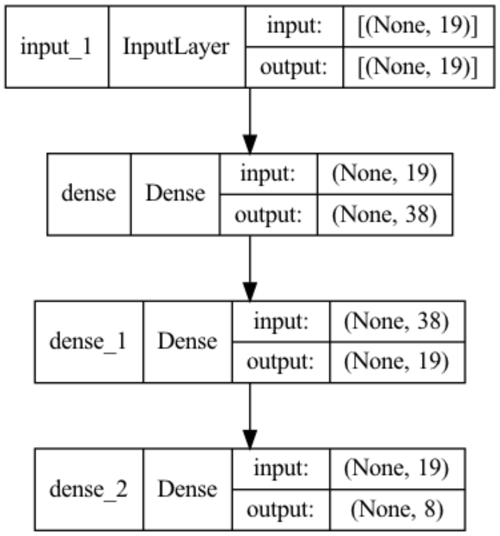
After the vectorization and normalization, the inputs and outputs for the lemmatization ANN are both lists of vectors. Additionally, every vector in both of these lists has the same number of elements. Similarly, after the vectorization and normalization, the inputs and outputs for the part of speech tagging ANN are also both lists of vectors. However, the number of elements of the vectors in the inputs list is determined by the longest word in that list, while the number of elements of the vectors in the outputs list is determined by the number of parts of speech in the list.

The data was then split into three sets: the training set, the validation set, and the testing set. The training set comprised 80% of the data, the validation set 10%, and the training set 10%. The training set was used to train the model, and, with each iteration of training, the model was tuned using the validation set. After the fitting had completed, the model was evaluated with the testing set.

Then, the ANN model was created using Keras[5]. For the lemmatization script, a model with two fully connected dense layers was chosen. The lemmatization model plot can be seen in Plot 1 below. For the part of speech tagging script, a model with three fully connected dense layers was chosen. The part of speech tagging model plot can be seen below in Plot 2 below.


dense_input | InputLayer | input: [(None, 19)] | output: [(None, 19)]
dense | Dense | input: (None, 19) | output: (None, 38)
dense_1 | Dense | input: (None, 38) | output: (None, 19)

Plot 1: Model Plot for the Lemmatization Script

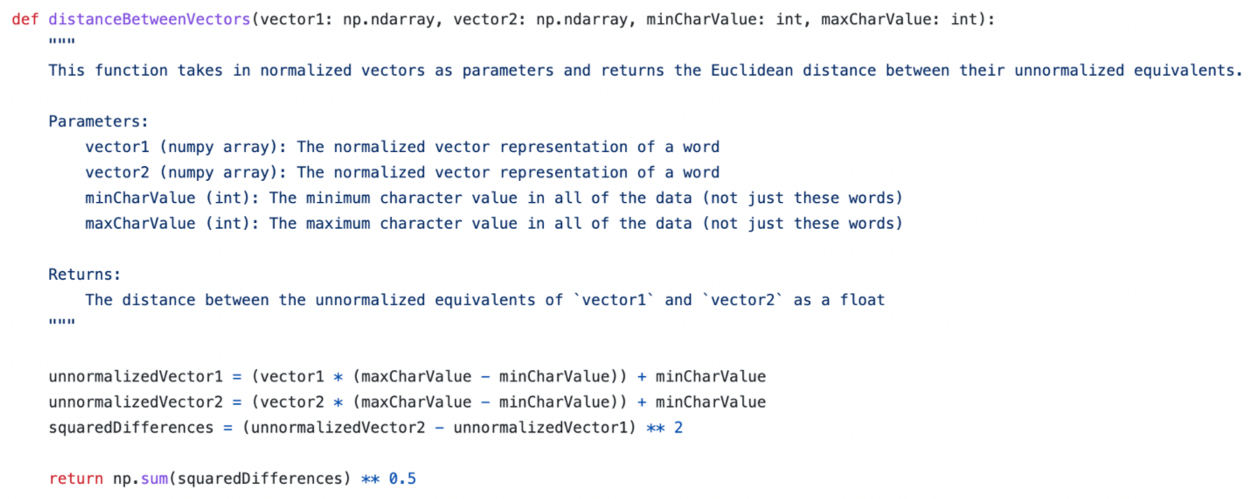

input_1 | InputLayer | input: [(None, 19)] | output: [(None, 19)]
dense | Dense | input: (None, 19) | output: (None, 38)
dense_1 | Dense | input: (None, 38) | output: (None, 19)
dense_2 | Dense | input: (None, 19) | output: (None, 8)

Plot 2: Model Plot for the Part of Speech Tagging Script

The accuracy of the lemmatization model was measured through the Euclidean distance between the unnormalized versions of the expected output vector and the actual output vector. The Euclidean distance was chosen as a metric because it considers the closeness of each component of the vectors instead of just a Boolean condition if they are equal or not. Because the vectors are unnormalized, the distance between two consecutive characters (e.g., ‘a’ and ‘b’) is 1. This method provides a good scale for measuring the closeness of two strings represented as vectors. The formula for the Euclidean distance between two vectors is shown in Equation 2 below, and the code for calculating the Euclidean distance of the unnormalized vectors is shown in Code Snippet 4 below.

$$\text{dist(expected, actual)} = \sqrt{\sum_{i=0}^{n-1} (\text{expected}[i] - \text{actual}[i])^2}$$

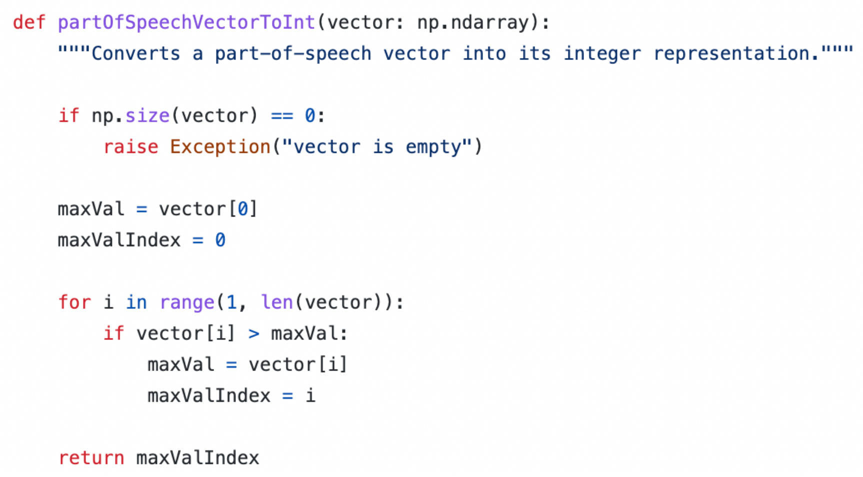
Equation 2: The Euclidean distance between two *n*-dimensional vectors, *expected* and *actual*.



```python
def distanceBetweenVectors(vector1: np.ndarray, vector2: np.ndarray, minCharValue: int, maxCharValue: int):
    """
    This function takes in normalized vectors as parameters and returns the Euclidean distance between their unnormalized equivalents.

    Parameters:
        vector1 (numpy array): The normalized vector representation of a word
        vector2 (numpy array): The normalized vector representation of a word
        minCharValue (int): The minimum character value in all of the data (not just these words)
        maxCharValue (int): The maximum character value in all of the data (not just these words)

    Returns:
        The distance between the unnormalized equivalents of `vector1` and `vector2` as a float
    """

    unnormalizedVector1 = (vector1 * (maxCharValue - minCharValue)) + minCharValue
    unnormalizedVector2 = (vector2 * (maxCharValue - minCharValue)) + minCharValue
    squaredDifferences = (unnormalizedVector2 - unnormalizedVector1) ** 2

    return np.sum(squaredDifferences) ** 0.5
```

Code Snippet 4: The distanceBetweenVectors function from Utils.py

The accuracy of the part of speech tagging model was measured by the percentage of correct guesses. The expected outputs are vectors with all components equal to 0 except for one component which has a value of 1. The actual outputs are vectors with a floating-point number in each component that represents the likelihood that the component is the correct one. The maximum value in an actual output vector is found, and the index of that maximum value is compared to the index of the 1 in the expected output vector. If they are equal, then the guess is correct. This process is then repeated for every pair of an expected vector and an actual vector, and the accuracy is calculated as the percentage of correct guesses. The code to convert an actual output vector to an integer representing the most likely guess is shown in Code Snippet 5 below, and the code for the whole part of speech model evaluation process is shown in Code Snippet 6 below.



```python
def partOfSpeechVectorToInt(vector: np.ndarray):
    """Converts a part-of-speech vector into its integer representation."""

    if np.size(vector) == 0:
        raise Exception("vector is empty")

    maxVal = vector[0]
    maxValIndex = 0

    for i in range(1, len(vector)):
        if vector[i] > maxVal:
            maxVal = vector[i]
            maxValIndex = i

    return maxValIndex
```

Code Snippet 5: The partOfSpeechVectorToInt function in Utils.py

```
# Evaluate the model
modelOutput_vector = model.call(tf.convert_to_tensor(headwordsTestingSet), training=False).numpy()
modelOutput_int = [Utils.partOfSpeechVectorToInt(x) for x in modelOutput_vector]

accuracy = 0
for i in range(lengthOfTestingSet):
    if (partsOfSpeechTestingSet_int[i] == modelOutput_int[i]):
        accuracy += 1
accuracy /= lengthOfTestingSet
```

Code Snippet 6: Model Evaluation in PartOfSpeechTagging.py

Results

For the lemmatization script, 30,576 data points were left after cleaning up the data. The average Euclidean distance between the expected headword and the model's output was 154.227 ± 94.470. To give an idea of what this actually means, Table 1 below shows selected outputs from running the lemmatization script which show the model outputs at various Euclidean distances.

Spelling	Headword	Model Output	Euclidean Distance
from	fram	ftXj	9.186
dēad	dēad	bčgO	22.463
hider	hider	aVF;	71.978
ðā	þā	Đ-!=	117.810
Experimental Average Value			154.227
ġeteld	geteld	ĒYceN	202.399
hwī	hwȳ	pÁ	373.072

Table 1: Selected results from running the Lemmatization script sorted by Euclidean distance

As can be seen from Table 1, the outputted word with a Euclidean distance of 9 from the headword is still unrecognizable from the headword. Even though some words were able to achieve a relatively low Euclidean distance from their headword, no outputted words were recognizably close to their headwords. A Euclidean distance of 0 means that the two vectors are the same, but a distance of just 10 can mean that a word is not close enough to its expected value to be recognized as that value.

Even though the words were not recognizably close to their headwords, the range of characters in the actual outputted vectors is approximately the same as the range of characters in the expected vectors. It was rare for a character to fall outside the normal text characters. Also, the number of characters in each actual outputted vector was usually around the number of characters in the expected vector. When the number of characters in the actual outputted vector was off, the number was always smaller than the number of characters in the expected vector. The ANN was not able to predict the characters exactly, but it was able to get them in the correct region of characters and usually around the correct number of characters.

For the part of speech tagging script, 4,625 data points were left after cleaning up the data. The average accuracy for the part of speech tagging script was 53.607% ± 3.783%. For reference, there were 8 parts of speech represented in the dataset. If the model were to guess randomly, the accuracy would be 12.5%. The model does significantly better than if it were to guess the parts of speech randomly.

However, in the dataset there were 2,275 nouns and 2,350 non-nouns. That means that 49.189% of the dataset was nouns. If the model were to always guess "noun", the accuracy would be around 49.189%, which is not much lower than the model's accuracy. In fact, most of the model's incorrect guesses were when it guessed "noun" or "verb" when a word was actually another part of speech.

Discussion

Both the lemmatization script and the part of speech tagging script produced promising results, but neither of them showed clearly positive results. This suggests that it is likely that one could get better results in the future with the right improvements to the data or model. Overall, it seems likely that ANNs are a good option for NLP for Old English, but this study did not produce good enough results to confirm that. It gave results that suggest that ANNs will be good in the future with the right improvements. There are many reasons why neither of the analyses worked that well, including the software library that was used, the corpus that was used, and Old English itself.

One negative effect of doing NLP for Old English instead of a modern language is that there is no preexisting software for it (besides CLTK, which did not have the right tools for this exploration). Software libraries like NLTK have been specially designed to perform NLP analyses, and they are well tuned for the languages that they support. These software libraries are used widely for NLP but do not support Old English. The methods used in this exploration were not specifically designed for NLP and they were not specifically tuned for Old English. Because of this, the results did not fit as well as they would have if a software library like NLTK supported Old English.

In the lemmatization script, words were represented as vectors, where the beginning components of the vector were the integer values of the characters in the word and the ending components were all 0s. It is possible that the neural network learned that 0s were common and overapplied them in the outputs, which could explain why some of the actual outputted vectors were shorter than the expected vectors.

In the part of speech tagging script, a similar process happened. Almost half of the words in the dataset were nouns. It is likely that the model learned that nouns are common and applied them too often. This explains why most of the errors were when the actual outputted value is “noun” when the expected value is not.

Also, the corpus used in this exploration was relatively small compared to modern corpora. For the lemmatization test, 30,576 spellings and their headwords were tested. For the parts of speech tagging, 4,625 headwords and their parts of speech were tested. A modern corpus can have way more words depending on its purpose. Old English words have many different inflections, and it is difficult for a machine learning model to learn all of the different transformations from inflected words to headwords along with the irregular forms when it only has access to a small corpus. A larger corpus would help to increase the accuracy of the model, because the model would be exposed to more examples.

Another issue that caused problems with the model was that Old English did not have standardized spellings. The same word could be written in many different ways depending on how a scribe decided to write it. This poses a problem for neural network methods because misspellings add an entirely different layer of complexity on top of all of the different possible word inflections. This issue could again be solved with a larger corpus. More examples of these words would give a neural network more material to train with and would most likely lead to better fitting results.

ANNs also have many hyperparameters that have to be tuned in order to produce optimal results. The shape of a neural network, which includes the number and types of layers as well as the number of nodes per layer, can drastically change the results the model produces. Other hyperparameters include activation functions, optimizers, learning rates, loss functions, etc.

Different hyperparameter values were tested briefly, but there was no extensive study into which hyperparameter values produced the best results. It is possible that with a more extensive look into different hyperparameter values, the model could produce better results.

As discussed, there were many limitations to this exploration. Even with those, this exploration produced results that indicate that ANN methods could work well for NLP for Old English. Future studies could standardize Old English spellings, use a bigger corpus, and spend more time adjusting hyperparameters to produce better results. The ultimate goal for Old English NLP would be to produce a software library like NLTK that has the same processing ability for Old English as NLTK does for Modern English.

References

[1] Natural Language Toolkit (NLTK): https://www.nltk.org/

[2] The Classical Languages Toolkit (CLTK): http://cltk.org/

[3] GitHub repository for this exploration: https://github.com/nbsnyder/OldEnglishNLP

[4] Python MySQL Connector module: https://dev.mysql.com/doc/connector-python/en/

[5] Keras: https://keras.io/