MLDS Fall 2018 Contest: Classifying Latin Text as Poetry/Prose

Amit Joshi

February 15, 2018

1 Objective

My main objective for this project was to classify Latin Text as poetry or prose, and to understand which features were most important in making the decision. One of my later objectives was to make deeper distinctions than poetry or prose, by making conclusions about the meaning and genre of the text.

2 Feature Extraction

In addition to the 26 hand engineered features, courtesy of the Dr. Pramit Chaudhuri and Dr. Joseph Dexter, I combined features from two text feature extraction algorithms to have more data to work with.

2.1 N-Grams (TF-IDF)

The first feature extraction algorithm I used was n-grams. The idea of n-grams is that the frequency of terms/phrases found in text is calculated in some way and put into a matrix. Each column of the matrix corresponds to a specific term. To specify, I used TF-IDF, which stands for Term Frequency times Inverse Document Frequency. The term frequency (TF) of a particular term, t, in a given document, d, is equal to the raw count of term t in d divided by the raw count of all terms in d. The Inverse Document Frequency (IDF) is a measure of rarity of a particular term, t. It is calculated by dividing the total number of documents, N, by the number of documents in which t occurs at least once, and then taking the logarithm of the quotient. Then, the TF-IDF value is calculated by multiplying TF and IDF together. A high TF-IDF value for term t and document d means that t occurred frequently inside d, and was rare in documents that were not d. The next step of TF-IDF vectorization is to set the hyperparameters. I decided to use terms of length 2 to 4 words, which occur in between 10% and 90% of all documents. This helps to help prevent overfitting.

2.2 Word2Vec (Skip-Grams)

The second feature extraction algorithm I used was Word2Vec's skip-grams. This algorithm aims to figure out the probability that a word will appear when in close proximity to a center word (probability that a word is a context word of a given center word). To do this, the Neural Network trains to maximize the probability that any context word is found in close proximity to a center word. Not only that, the loss function places a higher weight on context words that are similar to the center word. This loss function reaches a minimum when the expected value of the context word equals the actual observed context word. After the Neural Network is trained, the word vectors of each word in a block of text are averaged, creating 203 new features. These features are useful as they signify the meaning of the text.

3 Data Normalization

An easy way to improve accuracy and train faster is to normalize data. The reason for this is that different features have different ranges if the data is not normalized. One feature might have values on the order of 10^3, while some may have values on the order of 10^{-5}. This means that weight values need to be drastically different for different features. As initial weights are randomly from a normal distribution, the chance of good weights being initially chosen is virtually none. In practice, I found that this causes overfitting on train data, and also took longer to train. To combat this, I implemented z-score normalization, so each feature in the training set is centered at zero and has a standard deviation of one. I used the mean and standard deviation of the features in the training set to approximately normalize the test set features as well. Another positive ramification of this is that it allowed me to rank features by looking at the weight values of my logistic regression model.

4 Dimensionality Reduction/Denoising Data

The next step in the process was to reduce the dimensionality of the training set. I had a total of 2000 features, and I doubted that all of these features are linearly independent. So, I decided to reduce the dimensionality of the input space. In other words, I aimed to represent my all my features with less information. Ideally, the small amount of information lost is random noise. A conventional approach to this problem is using Principal Component Analysis (PCA) and selecting k out of n principal components to represent data, where n is the total number of features and k <n. However, out of curiosity, I decided to use a linear autoencoder. An linear autoencoder is a type of Neural Network whose input layer is the same as the output layer, but contains a smaller hidden layer with a linear activation function. My hidden layer consisted of 1500 nodes, hence, it transformed an input space with 2000 dimensions to a space of 1500 dimensions. This is very nearly equivalent to using PCA to select 1500 principal components. After this, another round of data normalization is necessary, because the denoised data may not be perfectly normalized. Overall, I found that using an autoencoder worked well, allowing my logistic regression model to attain more confidence in its decisions.

5 Logistic Regression

To actually predict whether a piece of latin text was poetry or prose I used logistic regression, with 2000 input nodes and 1 output node with sigmoid activation. Previously, I had used a neural network, but I realized that this does not easily allow me to rank features. In addition, I used L1_L2 regularization to penalize high weight values. A new loss function is created which adds this regularization penalty to sigmoid cross entropy (a loss function which measures accuracy). The model actually attempts to minimize this new loss function, so more balanced weight values are favored. During execution, I was able to assess the confidence of my decisions by seeing how far away from 0.5 the sigmoid activation result is. If the result is close to 1, it is confident that the block of text is poetry. If the result is close to 0, it is confident that it is prose.

6 Feature Labelling and Ranking

An important objective of this project was figuring out which features were most important in deciding whether a block of text was poetry or prose. Before I ranked features, I needed to label the features. For the features I extracted using n-grams, this was simple. The term name was the feature name. For the features I extracted using Word2Vec's skip-grams, naming was a little more complicated. Word2Vec changes each block of text to a 203 by 1 floating point array, where each of these 203 values represents one feature. To explain each feature, I created a vector with all zeros except for the the skip-gram feature that I wanted to name, which I set to one. Then, I found the word that is most similar to this vector, and named the feature after this word. For the most part, this worked very well, giving feature names like "antiqui" and "campi," which are presumably nouns. Most feature names were nouns or adjectives, giving a good description of what each feature meant. To rank features, I simply sorted weights. Highly positive weights indicated positive values for that feature signified poetry, and vice versa for highly negative weights. Below are some important features I discovered. Features prefaced with "similarWord:" indicate features that were generated using Word2Vec, and refer to the meaning of the text, not frequency of any particular word. Features prefaced with "frequency:" refer to features generated with n-grams, and refer to the TF-IDF value of a term. I found that features generated with Word2Vec tended to have very high weightage, while features generated with N-Grams have more medium weightage. The full feature ranking is on my github repository in a file called "feature_ranks.txt"

6.1 Top 5 Features where high inputs indicate Poetry

1. similarWord: vectura
2. similarWord: cognitiones
3. similarWord: auriculis
4. similarWord: ipso
5. similarWord: litteras

6.2 Top 5 Features where high inputs indicate Prose

1. similarWord: verecundis
2. similarWord: soleo
3. similarWord: angiportum
4. similarWord: gaudet
5. Interrogative Sentences

7 Deeper Distinctions

Using Word2Vec to provide a measure of similarity between words allowed me to make more distinctions than poetry/prose. I wanted to measure the mood of the text (happy/sad) as well as if the text was religious or not. To accomplish this, I compared the vectors for the Latin words for “felix” (happy), “tristis” (sad), and “deus” (god) to the vector for each piece of text. A useful measure of similarity is the cosine similarity (the greater the cosine similarity between two vectors is, the more similar the two vectors are). I outputted my predictions for these deeper distinctions in the “deeper_distinctions_train.txt” and “deeper_distinctions_test.txt” files. The larger each value is, the more that specific block of text corresponds with that category.

8 Citations

- Dr. Pramit Chaudhuri and Dr. Joseph Dexter from the Quantitative Criticism Lab
- Tensorflow
- Sci-kit Learn
- Gensim