Machine Learning Engineer Nanodegree

Capstone Project - "Who Wrote This?"

Stefan Dittforth
April 14th, 2018

I. Definition

Project Overview

When writing, authors leave distinctive marks in their stories influenced by the style of their writing. Well know writers are famous for their techniques with which they express ideas and manipulate language. Erika Rasso published a well written introduction to Famous Authors and Their Writing Styles. We learn, for example, that Ernest Hemingway "pioneered concise, objective prose in fiction—which had, up until then, primarily been used in journalism.". Another author discussed in Rasso's article is Franz Kafka. His stories present "surrealist, nightmarish writing in contemporary settings" which invoke feelings of confusion and helplessness. Agatha Christie's style was influenced by "mentions of war". Furthermore, "she utilized a variety of poisons to carry out the murders in her stories". And being interested in archarology "resulted in ancient artifacts and archaeologists being heavily featured in her novels". A last author worthwhile to highlight here is Zora Neale Hurston. Her style is quite unique: "She wrote in colloquial Southern dialects that mimicked the language she grew up hearing.".

Photo by Patrick Tomasso on Unsplash

Our intuition and experience as readers tells us that the style of writing is like a "finger print" that differentiates authors. If we come across a text with no information about the author or a text written under a preudonym, would we be able to tell the author based of the style of writing? An interesting article in this direction has been published by Carole E. Chaski: "Who's At The Keyboard? Authorship Attribution in Digital Evidence Investigations" . The article discusses the question to what extend text can be attributed to an author as part of crime investigations.

Problem Statement

In this project we will explore to what extend machine learning techniques can learn the style of writing for a set of authors. After a learning phase the system predicts which author wrote a given article. The system will not have seen the given article before. The likelihood of an article being written by a particular author will be expressed as a probability value between 0 (unlikely) and 1 (very likely).

We approach the problem with the following strategy:

1. Find a suitable data set that contains several articles from several authors. In order to have enough data to learn characteristics of individual authors we should seek a data set with a few hundred articles per author. Each article should be a few hundred words long.
2. Investigate the data to see whether data cleaning activities are required (e.g. does the text contain hypertext, header information or similar no content related information?).
3. Perform data cleaning activities if required.
4. Split the data set into training and test sets.
5. Select and train various models with different vectorization and classification algorithms with the training data set. Measure prediction accuracy of the models with the test data set.
6. Select best performing model for an application that allows a user to predict authorship of freely selected text.

Such an application can be of help, for example, for historians that try to establish authorship of text fragments. Another use case might be the analysis of how similar or dissimilar a text from one author is to other authors.

Metrics

The problem addressed in this project represents a multi-class classification problem. For a given text the system needs to predict the most likely author of that text out of a finite list of trained authors. As we will outline in section "Data Exploration" further down, we will build our own data set. This gives us the luxury to build a well balanced data set. We will have an equal number of articles for each author. This in turn allows us to use the classification accuracy metric. Classification accuracy expresses the number of correct predictions as a ratio of all predictions made on the unseen test dataset. Accuracy will be expressed in values between 0 and 1. Whereby 1 represents the perfect result meaning the model predicts all data sets correctly.

$$accuracy = \frac{true\ positives + true\ negatives}{total\ size\ of\ test\ data\ set}$$

Classification accuracy is well suited for data sets with an equal (or close to equal) number of observations in each class.

An alternative to accuracy is to express the model performance as log loss (good introduction can be found here). Log loss measures the performance of a classifier using the probabilities of the predicted labels. A perfect model would have a log loss of 0. The key difference to the accuracy is that log loss puts a penalty on incorrect classifications. This penalty is significantly higher the further away the probability of the predicted label is from the true label. Accuracy takes a "black & white" view on the predictions. A label has either been predicted correctly of not. Log loss takes into account the certainty for a prediction, expressed as probability, into account. In other words log loss is a better indicator for how confident a model is.

For our problem log loss will not be a suitable performance metric. For a given text our model will simply report the author with the highest probability as the predicted label. The probabilities for each author need to be seen relative to each other. In this scenario it is of less relevance if the probability of the predicted author is small if the probabilities of all other authors is even smaller.

Other metrics that could be used are precision and recall. A high value for precision indicates a low rate of false positives. High recall values indicate a low rate of false negatives. Both values need to be evaluated together. A classifier performs better when high values for precision and recall are achieved. The interrelationship between precision and recall can be expressed in a single metric: the F_1 score. The F_1 score is the harmonic mean of precision and recall (more details about precision, recall and F_1 score see Precision-Recall). Precision and recall and the F_1 score are valuable metrics for unbalanced data sets as they provides a better consideration of false positives and false negatives. However, as mentioned above we will work with a fairly well balanced dataset and therefore use classification accuracy as our key metric.

Where appropriate we will visualize confusion matrices to get better insight into where a model does well and where it fails. The confusion matrix allows us to see where exactly the misclassifications happening.

II. Analysis

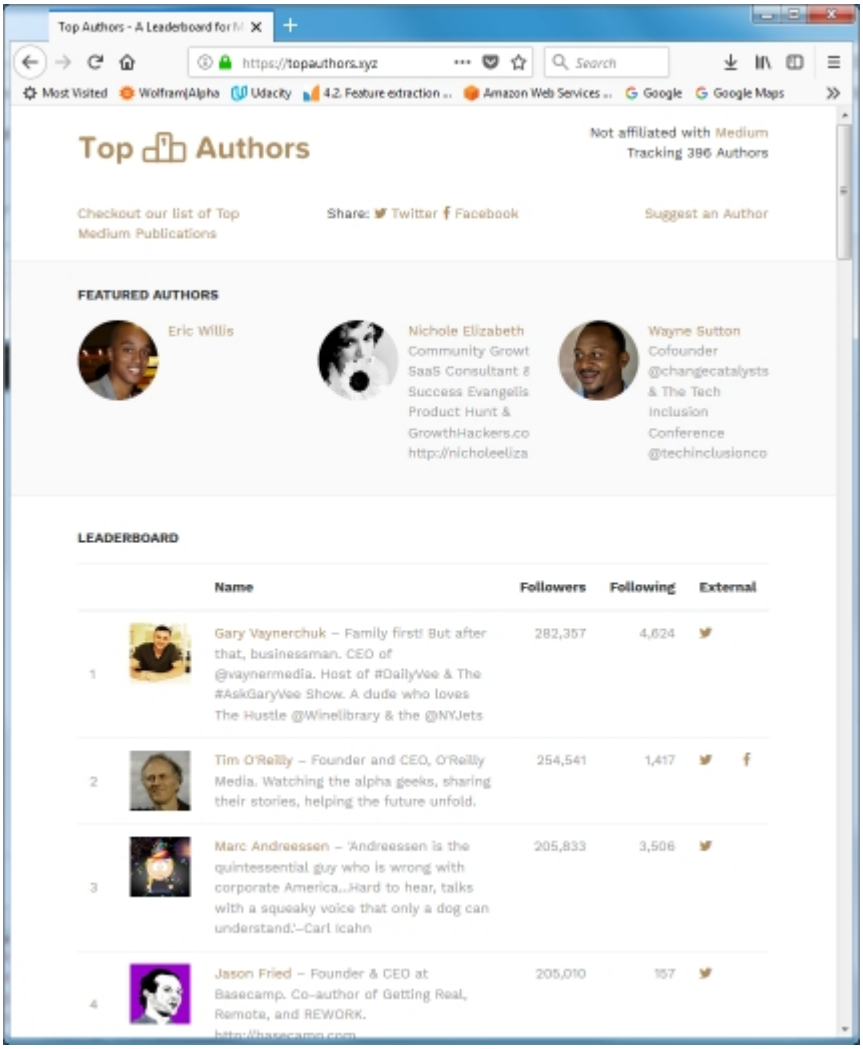
Data Exploration

In order to allow the system to learn the writing characteristics of different authors we require a dataset that provides a large number of articles for individual authors. There are rich datasets for NLP research available in the public domain. A list, as an example, can be found here. However, as part of this project we will build our own dataset. We will develop a web scraper that will collect articles from the publishing platform Medium. The articles on Medium seem to be reasonably long (at least several hundred words). There are enough authors that have published several hundreds articles. With this, it appears feasible to acquire a large enough data set to learn patterns in the writing characteristics to distinguish between individual authors.

This approach has been chosen as an opportunity to develop practical experience not only in machine learning but also around data acquisition. In data science and machine learning the acquisition and preparation of high quality data is often the bigger challenge than the actual development of the machine learning system itself. In "Datasets Over Algorithms" author Alexander Wissner-Gross notes that

> *"the average elapsed time between key [machine learning] algorithm proposals and corresponding advances was about eighteen years, whereas the average elapsed time between key dataset availabilities and corresponding advances was less than three years, or about six times faster, suggesting that datasets might have been limiting factors in the advances."*.

Conveniently the website Top Authors has published a list of 300+ top Medium authors. The project folder contains the short script `get_list_of_Medium_authors.py` that has been used to extract the Medium URL for each author. The initial list of 300+ authors has been reduced to 25. The criteria for this reduction was the number of published articles. For the 25 authors there are at least 300 articles available. The Medium URLs for these authors can be found in file `Medium_authors_25.txt`.

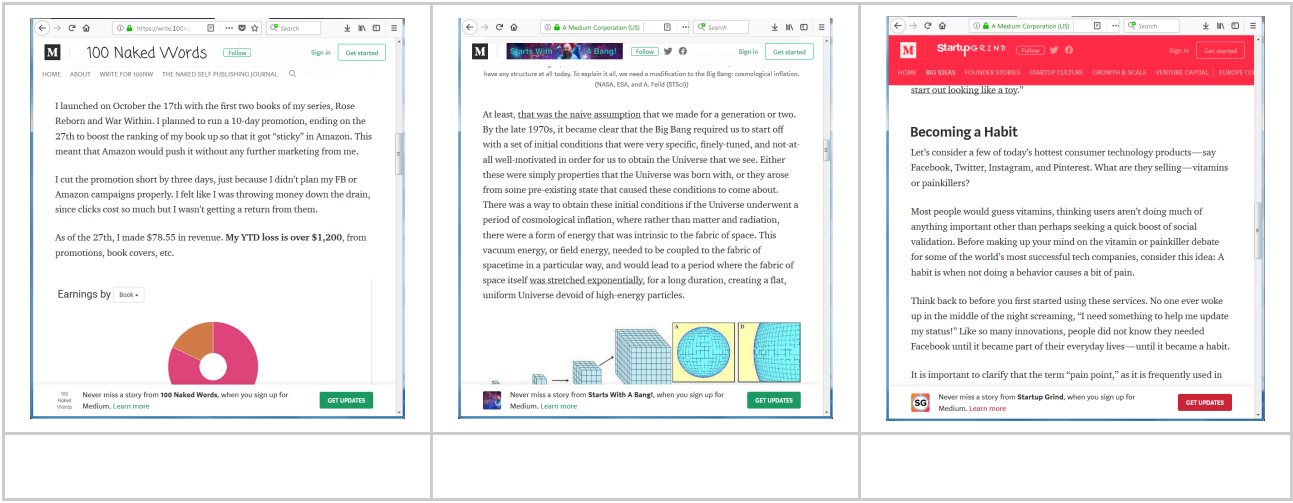


The actual collection of the articles is done with the script `pull_Medium_articles.py`. The script performs two steps. First, it builds a list of all article URLs and for each article saves author URL and article URL in JSON format in the file `Medium_article_urls.json`. Below is an example how the entries for three articles look like.

```
{"author_URL": "https://medium.com/@tedr/latest\n",
 "article_URL": "https://medium.com/season-of-the-witch/etiquette-and-the-cancer-patient-630a50047448?source=user_profile---------1----------"}
{"author_URL": "https://medium.com/@esterbloom/latest\n",
 "article_URL": "https://medium.com/the-billfold/failing-at-shoplifting-life-with-kesha-bc2600b1f440?source=user_profile---------789----------"}
{"author_URL": "https://medium.com/@gassee/latest",
 "article_URL": "https://mondaynote.com/the-ny-times-un-free-at-last-df2eddba360b?source=user_profile---------281----------"}
```

The second part performs the actual download of the articles. The script reads the article URL saved in `Medium_article_urls.json`, navigates to the website and reads the text information from the html code. Each article is saved in text format in its own file. For each author a folder is generated that contains the articles for that author. Initially it was intended to store all articles in JSON format in one file. This turned out to be very cumbersome when troubleshooting the `pull_Medium_articles.py` script. Having a folder structure that allows to do quick visual inspections over the list of files in a file manager proved very helpful.

In addition, the smaller article files made it easier to spot check the downloaded text information in a text editor.

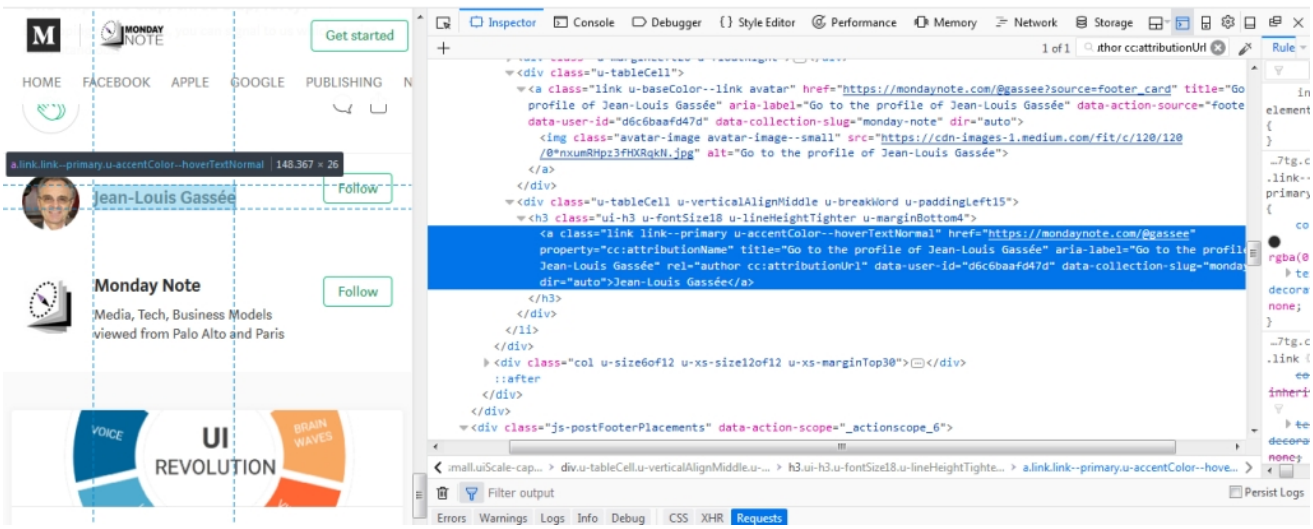


During research for this project several Python libraries for interacting with websites have been explored: mechanize, BeautifulSoup, scrapy and the Selenium WebDriver. Eventually the decision was made to use the Selenium WebDriver. The key reason for this was: the Medium website uses a two step login process. The users provides its email address and then receives a one time login link via this email. That made it difficult to automate the login via script and ruled out all the libraries that don't allow user interaction with the website.

Once an article website is loaded, the required information can be pulled from the text attribute of specific html elements. The code snipped below shows the commands used to get the author name and the article text.

```
author = self.browser.find_element_by_xpath('//a[@rel="author cc:attributionUrl"]').text
body = self.browser.find_element_by_xpath('//div[@class="postArticle-content js-postField js-notesSource js-trackedPost"]').text
```

As shown in the code snippet above the right elements are addressed by their respective xpath. Finding these xpaths required a bit of trail and error. A valuable tool for this is the FireFox Inspector. It allows to inspect the code and structure of a website and to find the right path to the right html element.



After the `pull_Medium_articles.py` script completed, the folder `Medium_articles` containing all article files has been compressed into a ZIP archive (`Medium_articles.zip`) to preserve storage. With `zipfile` Python provides a library to work with ZIP archives. Going forward in this Notebook we will make use of this library to work with the files directly within the ZIP archive without the need to extract the archive.

Developing a web scraper script poses its own challenges. The initial idea is pretty straightforward: here is a list of URLs, go to each website, download the text part and save it in a file. As always, the pitfalls are discovered during implementation. Some time had to be invested to understand the structure of the Medium article websites and figure out the best way to find the right html elements that contain the required information. The Selenium WebDriver is not the most effective tool when it comes to scraping several thousand websites. The time to render each and every website adds up. An attempt has been made by parallelising the article download with multi-threading and spawning of several instances of the Firefox browser. This failed. It turned out that the fast sequence of websites caused Firefox to slowly consume all available memory and eventually Firefox stopped fetching new websites. In a parallelised version of the script the problem was only exaggerated. Finally, a pragamatic approach was taken and the script has been amended with the capability to continue the work where it has left off from a previous run. Over the course of several days the script has been restarted several times and eventually saved all articles.

In defense for Selenium, it needs to be noted that Selenium first and foremost is a tool to automate testing of websites and not a tool for scaping several thousand websites. The primary goal behind the `pull_Medium_articles.py` script was to get the data for this capstone project and not to develop a sophisticated web scraper. In this respect Selenium did the job. Despite the challenges, developing the web scraper script has been a worthwhile learing experience. It provided an opportunity to develop practical experience around data acquisition.

Our web scaper has downloaded the articles in one file for each article. Each article is encoded in JSON format with four attributes: "url", "author", "headline" and "body". Below is an example for one article.


```
{"url": "https://medium.com/the-billfold/1-is-the-loneliest-number-that-you-ll-ever-do-
406f496b87c0?source=user_profile---------1193----------------",
 "author": "Ester Bloom",
 "headline": "1 is the Loneliest Number That You\u2019ll Ever Do \u2026",
 "body": "1 is the Loneliest Number That You\u2019ll Ever Do \u2026\nThursday is a great day to
do that 1 thing you don\u2019t want to do but also don\u2019t want to continue thinking about
doing.\nThe thing I really should do is go to my stupid health insurance webpage and find a
stupid doctor to take care of a small but annoying problem. It\u2019s the kind of problem that
surfaced LAST YEAR just before my family\u2019s 7-week Great Escape and I waited for it to go
away on its own as we made our way through two or three different countries, until I finally
limped into a Spanish hospital and begged for help. A punctual, kind, and handsome, though sadly
not very English-speaking, doctor gave me a temporary fix, and then said I\u2019d need to get
real attention once I returned to the States.\nOf course, I didn\u2019t. Now that it\u2019s
summer again the problem is getting harder to ignore. I hate bureaucracy and the thought of
maybe having to have surgery (!) and not being able to walk (!!) so I just keep putting it off.
But. Today I will change all that. Today I will find and call a doctor. Today I will DO 1
THING.\nHow about you?"}
```


All articles have been downloaded in individual text files and into folders for each author. This folder and file structure has been archived into `Medium_articles.zip`. For an an overview about the data set the table below list the number of articles for each author.

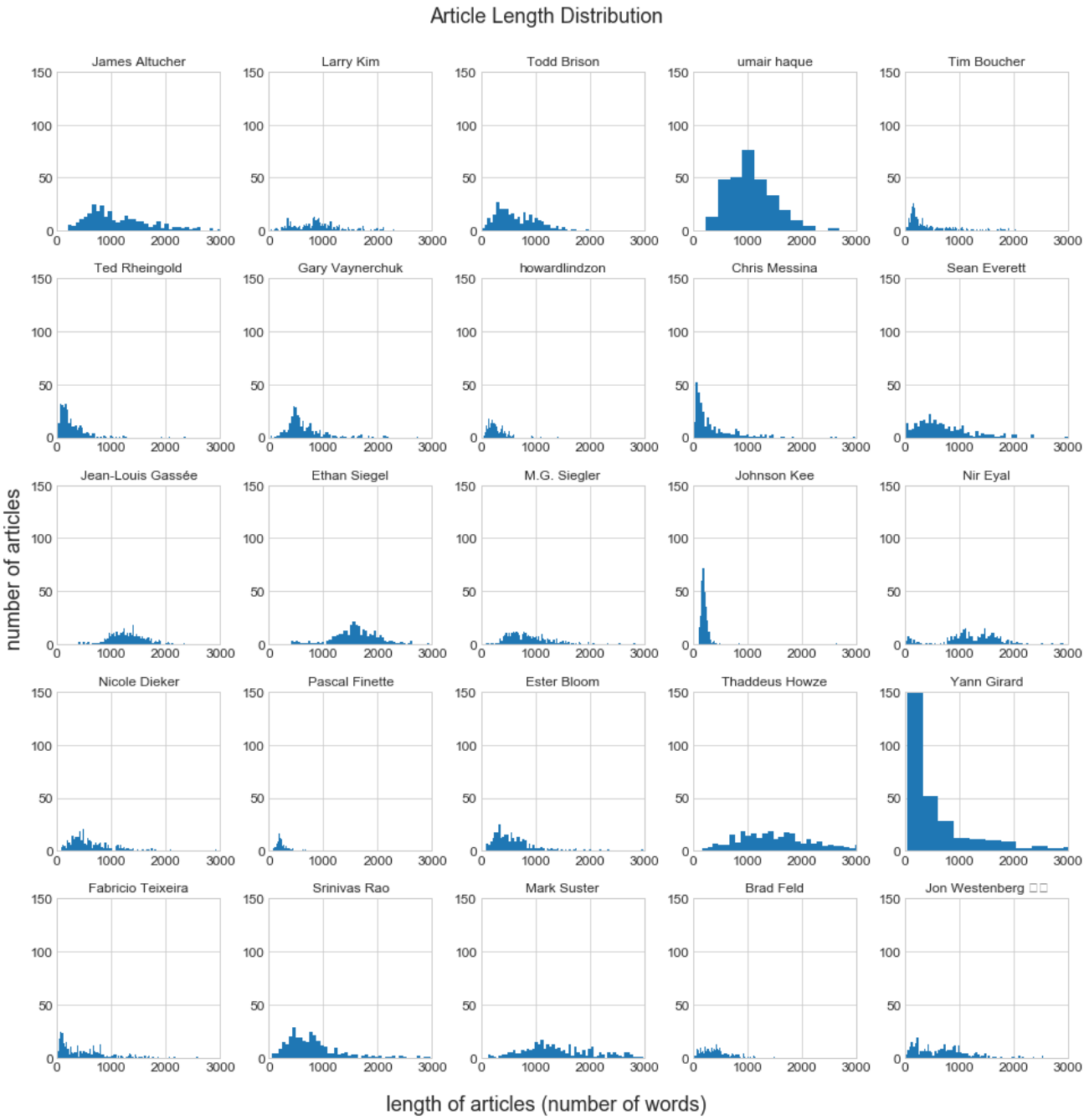
author	number of articles
Nicole Dieker	1948
Fabricio Teixeira	1733
Ester Bloom	1328
Ethan Siegel	1323
Yann Girard	1042
Pascal Finette	1020
Jon Westenberg	932
Chris Messina	872
Mark Suster	825
howardlindzon	795
umair haque	688
James Altucher	680
Johnson Kee	660
Larry Kim	619
Sean Everett	558
Tim Boucher	451
M.G. Siegler	377
Nir Eyal	366
Srinivas Rao	355
Brad Feld	350
Todd Brison	340
Thaddeus Howze	339
Jean-Louis Gassée	337
Ted Rheingold	319
Gary Vaynerchuk	307

The number of articles per authors in the data set is skewed. The number of articles ranges from 1,948 for Nicole Dieker to 307 for Gary Vaynerchuk. To avoid that our system develops a bias towards authors with a high number of articles we will balance the data set. This will be done by keeping the number of articles for each author equal to the author with the lowest number of articles.

Going forward we will work with 307 articles for each author.

Exploratory Visualization

To get some first insight into the data set we will plot a histogram for each author that shows the distribution of article lengths.



From the chart above we can already see differences between the various authors. We see, for example, Yann Girard and Chris Messina have a preference for shorter articles (less then 250 words) whereas the majority of articles of Ethan Siegel and Thaddeus Howze has a length between 1,000 and 2,000 words.

Algorithms and Techniques

We will implement the following models that will utilize various combinations of algorithms for vectorizing the raw text and then learning the classifications. The models represent different approaches to capture the "uniqueness" of writing of the different authors.

- **Model 1 Baseline Model - Just Guess the Author:** This will be our benchmark (or baseline) model. See discussion in section Benchmark.
- **Model 2 Basic Article Metrics:** We will come up with some simple metrics as features that might help us to predict the author of an article. The metrics that come to mind are: the total, mean, median, min and max number of words in paragraphs, sentences and the article itself. When looking at different authors it appears that these features might be useful differentiators. Some authors have a tendency to longer articles. Others use longer sentences or shorter paragraphs. We will test prediction accuracy when learning with decision tree and the logistic regression classifier.
- **Model 3 Bag-of-Words - Word Count:** In this model we will use the vocabulary to build our prediction model. We will implement a the Bag-of-Words algorithm. Bag-of-Words is a very simple but often surprisingly effective algorithm. It takes all the different words in a training set and uses them as features. Each article is then transformed into a feature vector by marking the occurrence of each feature (=word). There are a few variations of Bag-of-Words. In it's basic form each vector counts the number of occurrences for each word. This is what we are going to use in this model 3. The vectorized text will be used to train three classifiers: decision tree, logistic regression and support vector machines (SVM).
- **Model 4 Bag-of-Words - TFIDF:** A model that utilizes the term frequency–inverse document frequency (TFIDF) approach. In TFIDF, similar to word count, each word represents a feature. Each word is assigned a weight factor that is higher the more often a word occurs in an article. However, at the same time the weight of a word is reduced the more often it occurs across other articles. The idea behind TFIDF is that words that occur more across multiple articles carry less meaning then specialized words that occur within a smaller number of articles. The vectorized text will be used to train three classifiers: decision tree, logistic regression and support vector machines (SVM).
- **Model 5 Bag-of-Words - Reduced Vocabulary:** Using the Bag-of-Words algorithm usually results in very large feature vectors. Large feature vectors require more time during the learning phase. In this model we will investigate to what degree a reduced vocabulary impacts our prediction accuracy. We only keep words in the vocabulary that occur at a minimum rate. The vectorized text will be used to train three classifiers: decision tree, logistic regression and support vector machines (SVM).
- **Model 6 Bag-of-Words - Bigrams:** The previous models 3 - 5 only utilized the frequency or count of individual words. These approaches do not try to represent any form of "meaning" in the text. Would our prediction accuracy increase if we find a way to represent relationships between words? One popular way to represent word relationships in NLP are bigrams (or more generalized n-grams). Bigrams count the occurrence of word pairs. The hypothesis is that certain word combinations used by individual authors in their writings will provide a strong indication of authorship. The vectorized text will be used to train three classifiers: decision tree, logistic regression and support vector machines (SVM).
- **Model 7 Bag-of-Words - Reduced Bigrams:** This will be a variation of model 6. The use of bigrams during vectorization creates significantly large feature vectors. This is a problem as it demands a significantly larger volume of training data for the classifier to learn something meaningful. In this model we will investigate the impact of reducing the length of the feature vector by only keeping

bigrams with a minimum number of occurrences. The vectorized text will be used to train three classifiers: decision tree, logistic regression and support vector machines (SVM).

- **Model 8 Learn Word Embeddings & CNN:** Another approach to numerically represent meaning in text is offered by using word embeddings. Word embedding algorithms such as Word2Vec and GloVe aim to find similar representations for words with similar meanings. Individual words are represented by vectors with tens or several hundred dimensions. These vectors are learned by processing a large amount of text through a neural network. During learning, the algorithm learns the embedding either by predicting the current word based on its context or by predicting the surrounding words given a current word. The result of the embeddings learning phase is a set of vectors where words with similar meanings have similar vectors. Essentially "similarness" is represented by the distance between word vectors. The vector representations of words with similar meanings have shorter distances between them.

  In recent years word embedding approaches have been one of the key breakthroughs in natural language processing.

  Based on this promising outlook we will test the word embeddings approach on our problem. Model 8 will learn word embeddings as part of the convolutional neural network (CNN).

- **Model 9 GloVe Word Embeddings & CNN:** In this model we will utilize pre-trained word embeddings from the GloVe algorithm. This is done based on the hypothesis that the GloVe word embeddings will provide much more accurate representations of word relationships than what we can train with our limited training data set.

As mentioned in the model descriptions above we will utilize decision tree, logistic regression, support vector machine (SVM) and convolutional neural network (CNN) algorithms to train a classifier. In the following a description of how these algorithms allow to learn the uniqueness of individual authors.

Decision Tree

A decision tree is a structure in which nodes represent decision points. At each decision point a test is being performed. The branches originating from the decision point represent the possible outcomes of the test. Nodes with no further branches are called leafs. Leafs represent a classification class. The paths from the root to the leafs represent the set of decision rules of a model.

When using decision tree for our classification problem the model learns which features to choose and what conditions to use to split into branches. In our models the features are article metrics (model 2), word counts and bi-grams (models 3 - 7). Learning happens by looking at each feature and testing different split points. Each split is then tested against a cost function to determine the split with the lowest cost. The split point with the lowest cost is the used to split the data into groups. The approach is recursively applied to each group generated until the data cannot be split any further because the group contains only one element or only elements of one class.

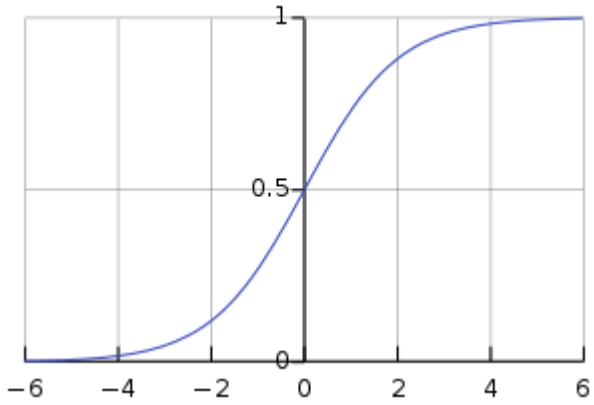
For the cost function we will use the Gini coefficient, which is the default cost function when using decision tree with the scikit learn Python library. The Gini coefficient provides an indication for how "pure" a group is after a split. A group containing only elements of one class will result in a Gini coefficient of 0 (lowest cost). A group with the "worst purity" will result in a Gini coefficient of 0.5. The Gini coefficient for a group is calculated as follows:

$G = \sum_k p_k \cdot (1 - p_k)$, pk represents the proportion of class k in the group.

Left unconstrained decision trees have a tendency to overfitting. This can be controlled by setting the minimum number of elements in a group or setting a maximum depth that a tree is allowed to grow.

Logistic Regression

The logistic regression algorithm is named after the logistic function, also called the sigmoid function, which provides a way to express the probability that a data point belongs to a particular class. The sigmoid function has a characteristic "S" shape curve with results in the range between 0 and 1.



source: Wikipedia

In its basic form the sigmoid S for a particular value x is calculated as follows:

$$S(x) = \frac{e^x}{e^x + 1}$$

Input values are combined with coefficients, denoted a β, to predict output values:

$S(x) = \frac{e^{\beta_0 + \beta_1 x}}{e^{\beta_0 + \beta_1 x} + 1}$ where β_0 is the intercept or bias and β_1 the coefficient for an input value x with only one dimension.

In our problem of predicting the author from a given text the input value will be a vector with several hundreds, if not thousands, of dimensions (= features). This requires a sigmoid function generalized for n dimensions:

$$S(x) = \frac{e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n}}{e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n} + 1} = \frac{e^{\beta_0 + \sum_{i=1}^{n} \beta_i x_i}}{e^{\beta_0 + \sum_{i=1}^{n} \beta_i x_i} + 1}$$

The β values are what our model needs to learn from the training data. Once these β values have been trained the model can be used to make predictions about who the authors is for a given text.

The β values are estimated using the maximum-likelihood estimation (MLE) algorithm. This algorithm seeks β values that minimizes the error for the probabilities predicted by the model to those in the training data.

Support Vector Machine

The support vector machine (SVM) algorithm seeks to find a line that best separates the data points by their class. This separator line is referred to as the hyperplane. In our problem articles will be transformed into multi-dimensional feature vectors and we need to find the hyperplane that best separates articles from various authors.

During the learning phase the SVM algorithm tries to find the hyperplane where the perpendicular distance between the hyperplane and the closest data points is the largest. This distance is called the maximal-margin hyperplane. The data points that are used in the distance calculation are called support vectors.

The maximal-margin hyperplane assumes that there is no overlap between the different classes, which in reality is seldom the case. To loosen the constraint of maximizing the margin additional coefficients are introduced to allow to tolerate some points of the training data to violate the separation hyperplane. This approach is referred to as soft margin.

When using the SVM algorithm different types of hyperplanes can be utilized. The different types are called kernels. Kernels are mathematical representations of hyperplanes with varying degrees of complexity. The scikitlearn toolkit used for this project offers linear, polynomial, rbf and sigmoid kernels. We will use the default kernel 'rbf' - radial basis function kernel.

$$K(x, x') = exp(-\frac{||x-x'||^2}{2\gamma^2})$$

The strength of RBF is that it can create very complex regions within the feature space of the training data which will allow a more accurate separation of classes.

Convolutional Neural Network

For models 8 & 9 we will utilize a convolutional neural network (CNN) to train the model to differentiate between authors. In recent years CNNs were behind many breakthroughs in computer vision. The main reason for these successes is the pattern learning capability of CNNs. CNNs consist of an input layer followed by multiple convolution, pooling and fully connected layers. As the training data flows through the CNN the different layers learn different pattern. In general the complexity of learned patterns increases with each layer in the CNN. In image recognition the first convolutional layer usually learns edges. Subsequent layers learn shapes made out of edges and deeper layers learn to recognize more complex objects from basic shapes.

For our project the hypothesis is that the pattern recognition capability of CNN's will also be applicable to our NLP problem. Our CNN will learn differences between authors that manifest themselves in the patterns of occurrences of individual words and words combinations.

To use a CNN for a NLP problem it is required to transform the text in articles into a numerical matrix representation. Each row in the matrix will represent a word. Words are represented as word embeddings vectors. Word embeddings are vectors with tens or several hundred dimensions. These vectors are learned by processing a large amount of text through a neural network. During learning, the algorithm learns the embedding either by predicting the current word based on its context or by predicting the surrounding words given a current word. The result of the embeddings learning phase is a set of vectors where words with similar meanings have similar vectors. Essentially "similarness" is represented by the distance between word vectors. The vector representations of words with similar meanings have shorter distances between them. The image below depicts a sample text transformed into a word embeddings representation using the pre-trained GloVe vectors.

GloVe word embeddings
vector with 300 dimensions

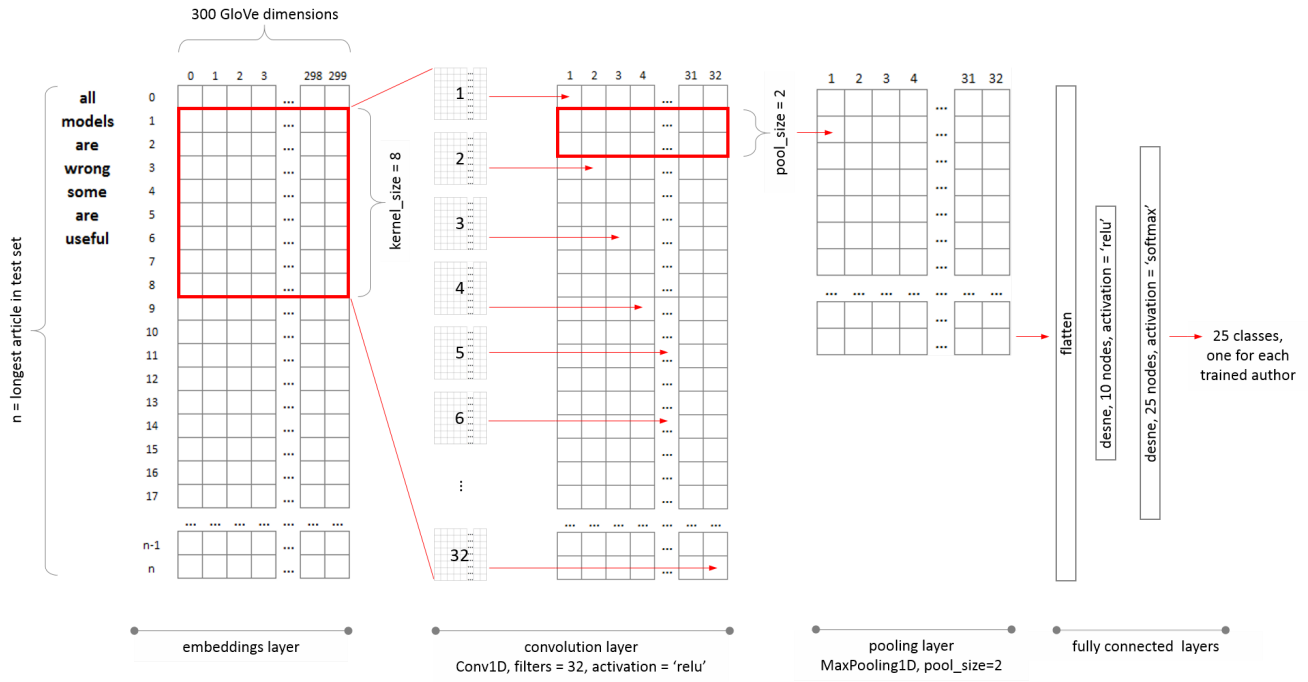
	0	1	2	...	298	299
all	-0.07	0.18	0.22	...	-0.09	-0.23
models	-0.14	0.58	-0.10	...	0.94	0.31
are	-0.24	0.38	0.11	...	-0.09	-0.15
wrong	0.23	-0.16	0.00	...	0.07	0.37
some	-0.21	0.18	-0.16	...	-0.06	-0.29
are	-0.24	0.38	0.11	...	-0.09	-0.15
useful	-0.20	0.06	0.65	...	0.06	0.11

The pre-trained GloVe vectors are available as 50, 100, 200 and 300 dimensional word vectors. For our model 9 we will use the 300 dimension version.

When configuring a CNN's we are requires to specify the dimensions of the input layer. Once defined these dimensions will be fixed throughout the lifetime of the CNN. This requires the input data to be normalized to the dimensions of the input layer. In computer vision, if the training images are of varying dimensions, this proves less of an issue. Images can be resized to the required dimensions of the input layer and will still preserve key information in the images. In our case, where the input data are articles of varying length, the fixed dimensions of the input layer poses a problem. Unfortunately there is no equivalent to "image resizing" for text.

One approach to address this would be to arbitrarily define a maximum length for articles. All words beyond that length are discarded. This obviously has the disadvantage that key information for learning an author's style of writing is lost with the discarded portion. A second approach might be to split articles into fixed length "sub-articles". These sub-articles then replace the original long article in the data set. The advantage of this approach is that information is not lost. However, a side effect is that this might skew the training data set. As outlined in the Data Exploration section above we deliberately prepared the data set so that there is an equal number of articles for each author to avoid that our model develops a bias towards authors with a high number of articles. If we are now splitting long articles into smaller sub-articles we might reintroduce the imbalance. The last approach, which we will use in our CNN models, is to use the length of the longest article in the training data set as the number of rows for the input layer. This ensures all training articles will fit into the embeddings layer.

The image below visualizes the complete CNN architecture used in our CNN models. The CNN architecture is based on a model shown in "Deep Learning for Natural Language Processing, Develop Deep Learning Models for Natural Language in Python" by Jason Brownlee, page 163. The model has been modified to a multiclass classifier (original model was a binary classifier for sentiment analysis).



The embeddings layer is followed by one convolution layer. The convolution layer will only have one dimension as the text has been transformed into a one dimensional list of word vectors. The kernel size for the convolution will be 8. This means the "sliding window" that will pass across the embeddings layer will include 8 words. This way the 32 filters in the convolution layer learn patterns across 8 words. A window that wide might help to learn pattern in word combinations. The result of the convolution layer is a feature map with 32 columns (one for each feature) and $n - k - 1$ rows, where n is the length of the longest article in the training data set and k the kernel size.

As a rule of thumb a higher number of filters allows the model to learn more patterns. However, a large number of filters mans that the dimensionality of the convolutional layer gets large. Higher dimensionality leads to a higher number of parameters which can lead to over fitting. Therefore we will add a pooling layer that helps to reduce the dimensionality of the convolutional layer. In our CNN models 8 & 9 we will work with pool size of 2. This will effectively reduce the number of rows by half (number of columns stay the same).

The model finishes with two fully connected layers. The first takes the flattened matrix output from the max pooling layer as input. The last layer consist of 25 nodes representing the 25 authors we are trying to predict. As we deal with a classification problem we will specify categorical cross entropy as our loss function for the last layer. The output is a 25 element vector where each element represents the probability how likely an article was written by a particular author.

The hyper parameter such as kernel size, number of filters, pool size have been taken as is from the CNN used in "Deep Learning for Natural Language Processing, Develop Deep Learning Models for Natural Language in Python". For a future project it might be worthwhile to investigate whether variations in these hyper parameters allow to achieve better prediction results. It will also be worthwhile to investigate to what extend additional convolution and pooling layers can improve results. If deeper CNNs where successful in detecting patterns in computer vision then they might also be useful for finding patterns in text.

Benchmark

To assess the quality of the predictions of the models described above we will need to compare against some baseline. In our case we will simply do a random guess of who the author of an article is. This should get us in the order of $\frac{1}{n} \cdot 100$ percent accuracy, where n represents the number of authors in the dataset.

```
Just guess the author:
----------------------

number of authors in data set: 25
expected prediction accuracy around: 4.00%
prediction accuracy score on data set: 3.77%
```

III. Methodology

Data Preprocessing

The articles were downloaded by extracting the 'text' attribute from html elements that contain the body text of the article. Many articles contain images, URLs to other pages, etc. There is the concern that the text extracts still contain html fragments which might need to clean out. Utilizing Python code all articles have been searched for the occurrence of the '<' and '>' characters. Those two characters enclose HTML tags. It was found that There are only 156 occurrences for the '<' and '>' characters in over 18,500 articles. Doing some spot checks it appears that the tag brackets are genuine parts of the article text (for example the text is about HTML coding). Based on this it has been decided to leave the html code fragments in the text corpus and not to clean them out.

Implementation

The project has been implemented in the form of the Jupyter notebook `WhoWroteThis.ipynb`. The following sub sections discuss the approaches developed for each of the models outlined in section Algorithms and Techniques above.

Model 1 Baseline Model - Just Guess the Author

This model implements the benchmark model against we will measure the subsequent models. Please see section Benchmark above for a discussion.

Model 2 Basic Article Metrics

We engineer a number features that might help us to predict the author of an article. For each article we calculate the following metrics:

- number of paragraphs in article (article_num_paragraphs)
- number of sentences in article (article_num_sentences)
- number of words in article (article_num_words)
- number of sentences from the paragraph with the largest number of sentences (paragraphs_max_sentences)
- average number of sentences across all article paragraphs (paragraphs_mean_sentences)
- median number of sentences across all article paragraphs (paragraphs_median_sentences)
- number of sentences from the paragraph with the smallest number of sentences (paragraphs_min_sentences)
- number of words from the sentence with the largest number of words (sentences_max_words)

- average number of words across all article sentences (sentences_mean_words)
- median number of words across all article sentences (sentences_median_words)
- number of words from the sentence with the smallest number of words (sentences_min_words)

The data pre-processing functions tokenize articles into paragraphs, sentences and words and count the frequency of each element. We remove punctuation from the data set. The counts are stored in a pandas data frame. These will be the feature vectors. The table shows an extract for three features from three articles.

	author	article_num_paragraphs	article_num_sentences	article_num_words
0	Brad Feld	3	5	67
1	Brad Feld	3	4	68
2	Brad Feld	13	45	708

With the data in a pandas data frame structure we can now train classification algorithms. We first move column 'author' into a separate table as our data labels. As the data is ordered by authors we will first shuffle the data set and then split it into training and test data sets. We will use 25% of the data for testing.

In this model we uses the decision tree classifier and achieved the following results:

```
decision tree classifier:
-------------------------

prediction accuracy score on training data set: 99.98%
prediction accuracy score on test data set: 33.51%
```

On unseen data the accuracy is around 33% (the value will vary slightly with each run). That is not too bad for a model that tries to guess the author of an article only based on the number of words in articles, paragraphs and sentences. The model does not take anything of the article content into account. The 100% accuracy on the training data means the model is extremely biased.

In the next step we performed a grid search to find a better set of parameters that reduces the extreme bias. For the grid search we search for optimal parameters for the maximum depth (`max_depth`) in the range 1 - 30 and the minimum number of samples in the split groups (`min_samples_split`) in the range 2 - 50. We achieved the follow results:

```
GridSearch optimized decision tree classifier:
----------------------------------------------

prediction accuracy score on training data set: 50.56%
prediction accuracy score on test data set: 36.22%
best parameters:
    max_depth:10
    min_samples_split:23
```

The optimization of two parameters with grid search gains us three more percentage points and gets us to 36%. Furthermore, the bias, the difference between the training and test performances, has been reduced.

For comparison we will run the logistic regression classifier and see if that yields better results.

```
logistic regression classifier:
-------------------------------

prediction accuracy score on training data set: 34.69%
prediction accuracy score on test data set: 32.20%
```

With 32% accuracy the result is 4% below what has been achieved with an (grid search) optimized decision tree algorithm.

Model 3 Bag-of-Words - Word Count

In our next model we will use the vocabulary to build our prediction model. We will implement a the Bag-of-Words algorithm. Bag-of-Words is a very simple but often surprisingly effective algorithm. It takes all the different words in a training set and uses them as features. Each article is then transformed into a feature vector by marking the occurrences of each feature (=word). There are a few variations of Bag-of-Words. In it's basic form each vector counts the number of occurrences for each word. This is what we are going to use in this model 3. In the next section (model 4) will use the term frequency–inverse document frequency (TFIDF) approach to build the Bag-of-Words vectors.

The code for this model is separated into a number of functions. The approach to define functions vs. directly executed code cells has been chosen to make the code reusable for the next model in the following section.

Besides the training, predicting and scoring functionalities the code implements two additional capabilities: 1.) it allows trained classifiers to be saved to or loaded from disk. This saves time when the notebook is run multiple times. The model needs to be trained only once. 2.) The model can be trained with different classifiers. During experimentation very often different classifiers (e.g. logistic regression, decision tree, SVM) are tested to evaluate which one is most suitable for a problem.

The data pre-processing for this model does the following:

1. tokenize articles into words
2. make all words lowercase
3. remove stop words
4. reduce words to its base using a Porter stemmer
5. remove punctuation

Next we will utilize the Python functions to build our model. A model will be represented as a Python class. This design decision was driven by a view that our model should be "productized" to some degree. A user of our WhoWroteThis model will want to use a simple and straightforward interface. She should be able to pass in a list of articles in human readable text format and all the activities around text preprocessing, cleaning and vectorizing the data should be hidden.

Running the model yields the following results:

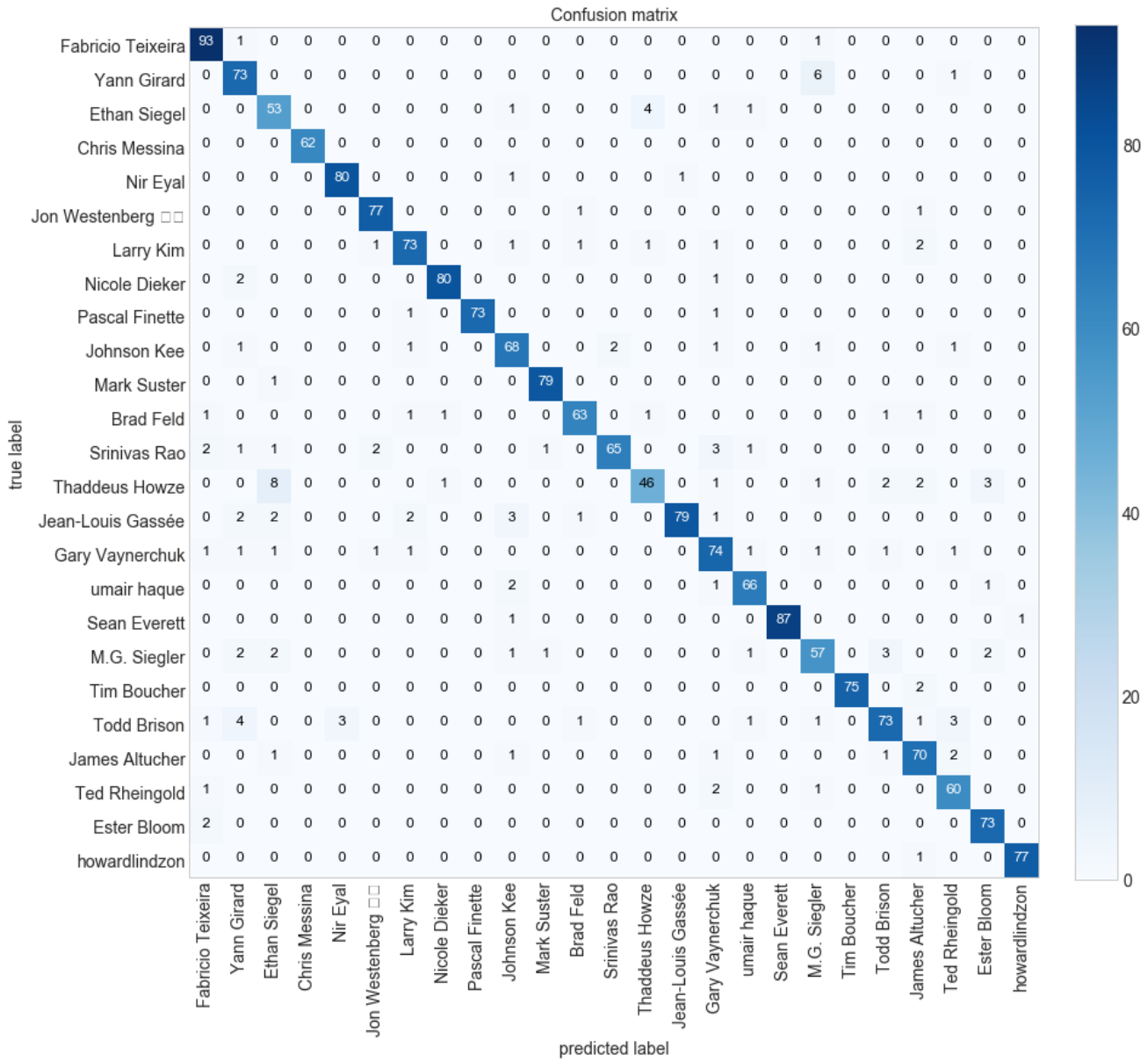
```
training the model

number of training articles: 5,756
size of the vocabulary = length of the feature vector: 63,021


scoring the training data:
```

```
logistic regression classifier prediction accuracy score : 100.00%
decision tree classifier prediction accuracy score : 100.00%
SVM classifier prediction accuracy score : 33.10%

scoring the test data:

logistic regression classifier prediction accuracy score : 92.55%
decision tree classifier prediction accuracy score : 76.76%
SVM classifier prediction accuracy score : 29.55%
```

Using the logistic regression classifier gets us well above 92% prediction accuracy on the test data set. It appears that simply counting the word frequency is a very strong differentiator between authors. It seems to confirm our intuition that a person is able to differentiate between authors by what they write about. Different topics will naturally use different vocabulary. For future research (not pursued in this project) it might be interesting to investigate how the classifier works for authors that write about the same or similar topics.

From the accuracy scores on training and test data we can see that the SVM classifier doesn't perform well on this particular problem. This comes a bit as a surprise. According to the scikit learn documentation SVM is a classifier that generally is effective in high dimensional space and, more importantly, is "still effective in cases where number of dimensions is greater than the number of samples". In our case the number of 63,021 dimensions (= features) is significantly higher than the 5,756 samples (= number of training articles). Perhaps, because in our case the number of dimensions is almost ten times the number of samples make SVM unsuitable for our problem. The SVM library in scikit learn offers options for optimization such as various classifiers (SVC, NuSVC and LinearSVC) and different kernel functions. For this project SVM has not been pursued further as the initial results are way below what the logistic regression classifier achieved.

The accuracy score as a single value doesn't give us much insight into what the model does well and where it fails. For this we will visualize the confusion matrix.



The confusion matrix allows us to see where exactly the misclassifications happening. We can see that Tim Boucher has been misclassified eight times as howardlindzon. Also, John Westenberg has been misclassified six times as Gary Vaynerchuk. The examples show that misclassification can be concentrated to a few classes. With this, confusion matrices can provided valuable cues that are helpful for further fine tuning models.

Model 4 Bag-of-Words - TFIDF

With the Python code for model 3 developed with re-usability in mind, we can now quickly implement our next model that utilizes the term frequency–inverse document frequency (TFIDF) approach. In TFIDF, similar to word count, each word represents a feature. Each word is assigned a weight factor that is higher the more often a word occurs in an article. However, at the same time the weight of a word is reduced the more often it occurs across other articles. The idea behind TFIDF is that words that occur more across multiple articles carry less meaning then specialized words that occur within a smaller number of articles.

To run this model we only need to replace the `CountVectorizer()` vectorizer with the `TfidfVectorizer()` vectorizer in our Python code from model 3. We achieved the following results:

```
training the model

number of training articles: 5,756
size of the vocabulary = length of the feature vector: 63,021

scoring the training data:

logistic regression classifier prediction accuracy score : 98.33%
decision tree classifier prediction accuracy score : 100.00%
SVM classifier prediction accuracy score : 4.29%

scoring the test data:

logistic regression classifier prediction accuracy score : 91.04%
decision tree classifier prediction accuracy score : 75.09%
SVM classifier prediction accuracy score : 3.13%
```

The performance of the TFIDF model with logistic regression is almost identical to the word count model described in the previous section. The TFIDF of words doesn't help to increase the accuracy in predicting the author. This can be explained by the fact that authors most probably write about specific topics (e.g. entrepreneurship, astrophysics). With this we can expect a high frequency of key topic words within the articles from the same author but less so across all articles in the training set. This in turn means that the IDF part of TFIDF has less impact on the learning. Which then means the TFIDF gets closer to what the word count does.

Model 5 Bag-of-Words - Reduced Vocabulary

During pre-processing model 3 extracts a vocabulary of over 63,000 words from 5,756 training articles. As the vocabulary is used as the feature vector this results in very large vectors. Large feature vectors require more computational resources during the learning phase. In this model we test to what degree a reduced vocabulary impacts our prediction accuracy. We only keep words in the vocabulary that occur at a minimum rate (specified by variable `min_occurrence`). The following results were achieved:

```
training the model

number of training articles: 5,756
size of the vocabulary = length of the feature vector: 1,954

scoring the training data:

logistic regression classifier prediction accuracy score : 99.98%
decision tree classifier prediction accuracy score : 100.00%
SVM classifier prediction accuracy score : 84.07%

scoring the test data:

logistic regression classifier prediction accuracy score : 88.59%
decision tree classifier prediction accuracy score : 70.09%
SVM classifier prediction accuracy score : 73.16%
```

A number of different thresholds for the minimum occurrence have been tested:

minimum occurrence	length feature vector	logistic regression accuracy
4	18,285	92.55%
20	7,825	92.13%
100	3,102	90.78%
150	2,395	89.32%
200	1,954	88.59%

With a moderate threshold of four the logistic regression prediction accuracy remains at 92.55% at the same performance level as model 3. However, the length of the feature vector has been reduced dramatically from 63,021 to 18,285 tokens. Even with a drastic threshold of 200 the model still performs well above 88% prediction accuracy but with the benefit of a much smaller feature vector. This result shows that the feature vector can be reduced significantly without a large impact on the prediction accuracy.

Model 6 Bag-of-Words - Bigrams

The previous models 3 - 5 only utilized the frequency or count of individual words. These approaches do not try to represent any form of "meaning" in the text. Would our prediction accuracy increase if we find a way to represent relationships between words?

One popular way to represent word relationships in NLP are bigrams (or more generalized n-grams). Bigrams count the occurrence of word pairs. The hypothesis is that certain word combinations used by individual authors in their writings will provide a strong indication of authorship.

This model tests this hypothesis. With `CountVectorizer(ngram_range=(2, 2))` we specify a count vectorizer that vectorizes a given text into bigrams. We achieved the following results:

```
training the model

number of training articles: 5,756
size of the bigram vocabulary = length of the feature vector: 1,205,446

scoring the training data:

logistic regression classifier prediction accuracy score : 100.00%
decision tree classifier prediction accuracy score : 100.00%
SVM classifier prediction accuracy score : 4.29%

scoring the test data:

logistic regression classifier prediction accuracy score : 87.60%
decision tree classifier prediction accuracy score : 73.48%
SVM classifier prediction accuracy score : 3.13%
```

The results above show one significant disadvantage of the bigram approach: the length of the feature vector literally "explodes" to 1,205,446 features. Given our 5,756 training articles we need to be aware that we are now cursed by dimensionality. In order for a classifier to learn effectively the number of training examples needs to grow with the dimensionality of the feature vector. The Wikipedia article about the Curse of dimensionality mentions an interesting rule of thumb: "there should be at least 5 training examples for each dimension in the representation". Applying this rule to our bigram model would mean to have somewhat over six million training articles. This will be impossible to get for our problem.

Model 7 Bag-of-Words - Reduced Bigrams

As discussed in the results for the previous model, the use of bigrams during vectorization creates significantly larger feature vectors. This is a problem as it demands a significantly larger volume of training data for the classifier to learn something meaningful. In this model we will reduce the length of the feature vector by only keeping bigrams with a minimum number of occurrences. We achieved the following results:

```
training the model

number of training articles: 5,756
size of the bigram vocabulary = length of the feature vector: 31,169

scoring the training data:

logistic regression classifier prediction accuracy score : 99.81%
decision tree classifier prediction accuracy score : 99.90%
SVM classifier prediction accuracy score : 11.71%

scoring the test data:

logistic regression classifier prediction accuracy score : 86.71%
decision tree classifier prediction accuracy score : 72.75%
SVM classifier prediction accuracy score : 9.69%
```

A number of different thresholds for the minimum occurrence have been tested:

minimum occurrence	length feature vector	logistic regression accuracy
4	65,839	87.81%
7	31,169	86.71%
10	19,258	86.35%
20	7,174	84.37%
100	657	70.24%

With a threshold of four we can see that the prediction accuracy of the logistic regression classifier increases slightly compared to model 6. A possible explanation for this behavior is offered by the Hughes phenomenon. This phenomenon states that "the predictive power of a classifier or regressor first increases as number of dimensions/features used is increased but then decreases". It appears that in model 6 we are

far on the side of decreased performance. By reducing the length of the feature vector we have increased the power of the classifier.

Model 8 Learn Word Embeddings & CNN

Another approach to numerically represent meaning in text is offered by using word embeddings. Word embedding algorithms such as Word2Vec and GloVe aim to find similar representations for words with similar meanings. Individual words are represented by vectors with tens or several hundred dimensions. These vectors are learned by processing a large amount of text through a neural network. During learning, the algorithm learns the embedding either by predicting the current word based on its context or by predicting the surrounding words given a current word. The result of the embeddings learning phase is a set of vectors where words with similar meanings have similar vectors. Essentially "similarness" is represented by the distance between word vectors. The vector representations of words with similar meanings have shorter distances between them.

In recent years word embedding approaches have been one of the key breakthroughs in natural language processing.

Based on this promising outlook we will test the word embeddings approach on our problem in the following two models. Model 8 will learn word embeddings as part of the convolutional neural network (CNN). In model 9 we will use pre-trained word embeddings.

A very helpful resource for this topic was the book *"Deep Learning for Natural Language Processing, Develop Deep Learning Models for Natural Language in Python"* from Jason Brownlee. From this book (page 163) the CNN used in this model has been taken. The model has been modified to a multiclass classifier (original model was a binary classifier for sentiment analysis).

A key learning from implementing this model: The `one_hot` function from the `keras.preprocessing.text` library was originally used to convert the words in articles to integer values. This was done as preparation to train the CNN. However, `one_hot` does not encode collision free. During testing it has been found that against a text corpus with 54,619 unique words `one_hot` caused 22,793 collisions. Meaning 22,793 integers were not uniquely mapped to single words. It is believed that this high proportion of collisions significantly distorts the learning of the embedding CNN. To resolve this issue function `learn_vocabulary` has been implemented.

The data pre-processing for models 8 & 9 varies in one step from what has been used for models 3 - 7: we will not reduce words to its base using a Porter stemmer. This is because the word vectors provided with Word2Vec and GloVe work with "un-stemmed" words.

For an extensive discussion of the CNN used in models 8 & 9 please see section Convolutional Neural Network above.

The following results were achieved:

```
scoring the training data:

5756/5756 [==============================] - 245s 42ms/step
accuracy score : 15.55%

scoring the test data:

1919/1919 [==============================] - 86s 45ms/step
accuracy score : 13.18%
```

When predicting authors based on word embeddings learned from scratch we achieve a prediction accuracy on the test data of 13.18% (please note, results vary significantly with different runs of this notebook). This is a very low prediction accuracy. This can be explained by the fact that the word embeddings were learned from only 5,756 articles (around 3.5 million words/tokens). Word embeddings are algorithms that require large amount of training data to properly learn relationships between words. Available pre-trained word embeddings were trained on significantly larger text corpora. For example the GloVe word embeddings file glove.6B.zip contains 6 billion tokens and a vocabulary of 400,000 words and was trained on the Wikipedia corpus.

Model 9 GloVe Word Embeddings & CNN

In our last model we will utilize pre-trained word embeddings from the GloVe algorithm. This is done based on the hypothesis that the GloVe word embeddings will provide much more accurate representations of word relationships than what we can train with our limited training data set.

The following results were achieved:

```
scoring the training data:

5756/5756 [==============================] - 627s 109ms/step
accuracy score : 97.79%

scoring the test data:

1919/1919 [==============================] - 206s 107ms/step
accuracy score : 55.24%
```

Compared to a self trained embeddings layer we achieve a significantly better prediction accuracy of 55.24% by using the pre-trained GloVe word vectors (prediction rate will vary with different runs of the notebook). However, we are still far-off from the the prediction performance of model 3 with 92.55% (count vectorizer and logistic regression classifier).

It appears that word embeddings are not the right approach for our problem. Word embedding algorithms put emphasis on representation of meaning in text. The approach might be more useful if we want to classify articles by their content (e.g. astronomy, programming, finance, etc.).

Refinement

In model 2 the grid search approach was used to perform a refinement of the model. With default settings for the scikit-learn functions we achieved the following results using a decision tree classifier:

```
prediction accuracy score on training data set: 99.98%
prediction accuracy score on test data set: 33.51%
```

We can see a significant difference between the accuracy achieved on the training data (nearly 100%) and the test data (33.51%). The default settings means the model is extremely biased. A grid search was performed to find a better set of parameters that reduces the extreme bias and possibly increases accuracy of the model. For the grid search we searched for optimal parameters for the maximum depth (`max_depth`) in the range 1 - 30 and the minimum number of samples in the split groups (`min_samples_split`) in the range 2 - 50. We achieved the follow results:

```
prediction accuracy score on training data set: 50.56%
prediction accuracy score on test data set: 36.22%
best parameters:
    max_depth:10
    min_samples_split:23
```

The optimization of two parameters with grid search gains us three more percentage points and gets us to 36%. Furthermore, the bias, the difference between the training and test performances, has been reduced.

However, it was found that model 3 utilizing word count for text vectorization and a logistic classifier performed significantly better. The prediction accuracy against the test data set for model 3 is 92.55%. All other models performed way below this performance mark. A visualization of the performance of the individual models and a detailed discussion can be found in the Conclusion section.

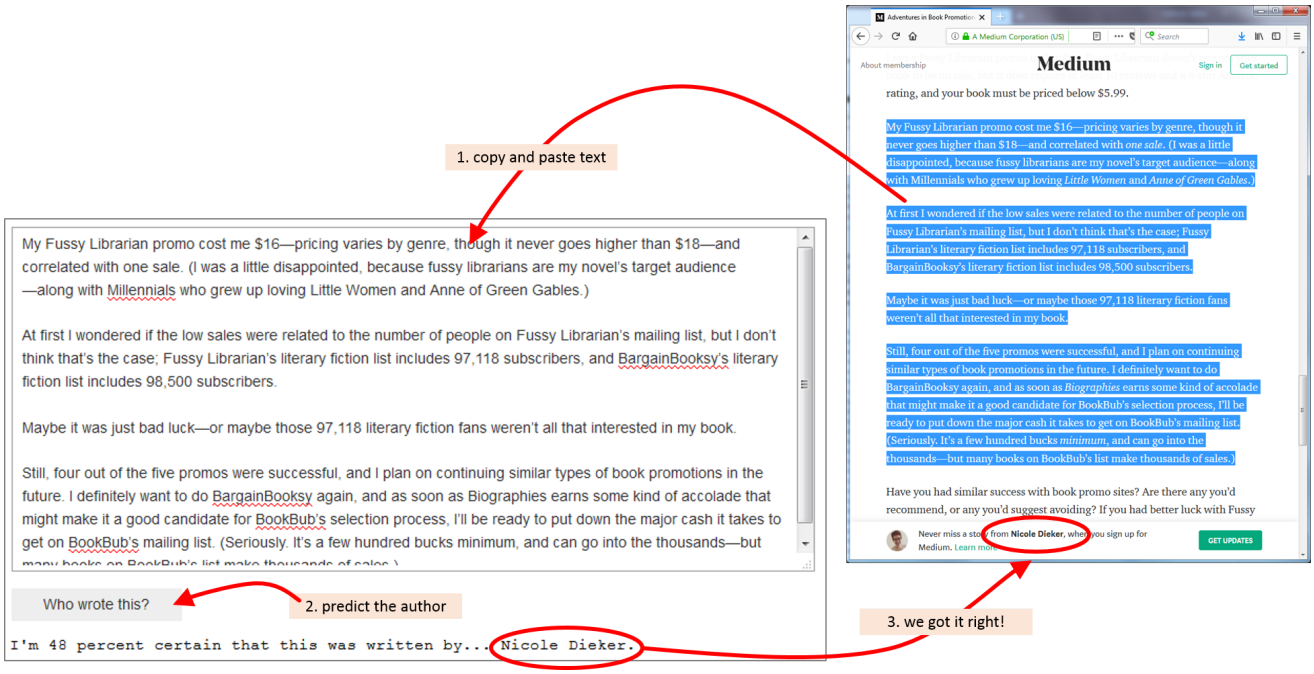
IV. Results

Model Evaluation and Validation

The successful model in this project is model 3. This model utilized the Bag-of-Words algorithm with a word count vectorizer. For each word in an article the number of occurrences is counted. Three classification algorithms have been tested: logistic regression, decision tree and SVM. The logistic regression classifier was the best performing classifier with a prediction accuracy of 92.55% on the test data set.

```
training the model

number of training articles: 5,756
size of the vocabulary = length of the feature vector: 63,021

scoring the training data:

logistic regression classifier prediction accuracy score : 100.00%
decision tree classifier prediction accuracy score : 100.00%
SVM classifier prediction accuracy score : 33.10%

scoring the test data:

logistic regression classifier prediction accuracy score : 92.55%
decision tree classifier prediction accuracy score : 76.76%
SVM classifier prediction accuracy score : 29.55%
```

The model has been used in a small application at the end of the notebook to demonstrate a possible real world application use case. The user can copy an article (or a part of an article) in a text field and let the model predict who the author is.



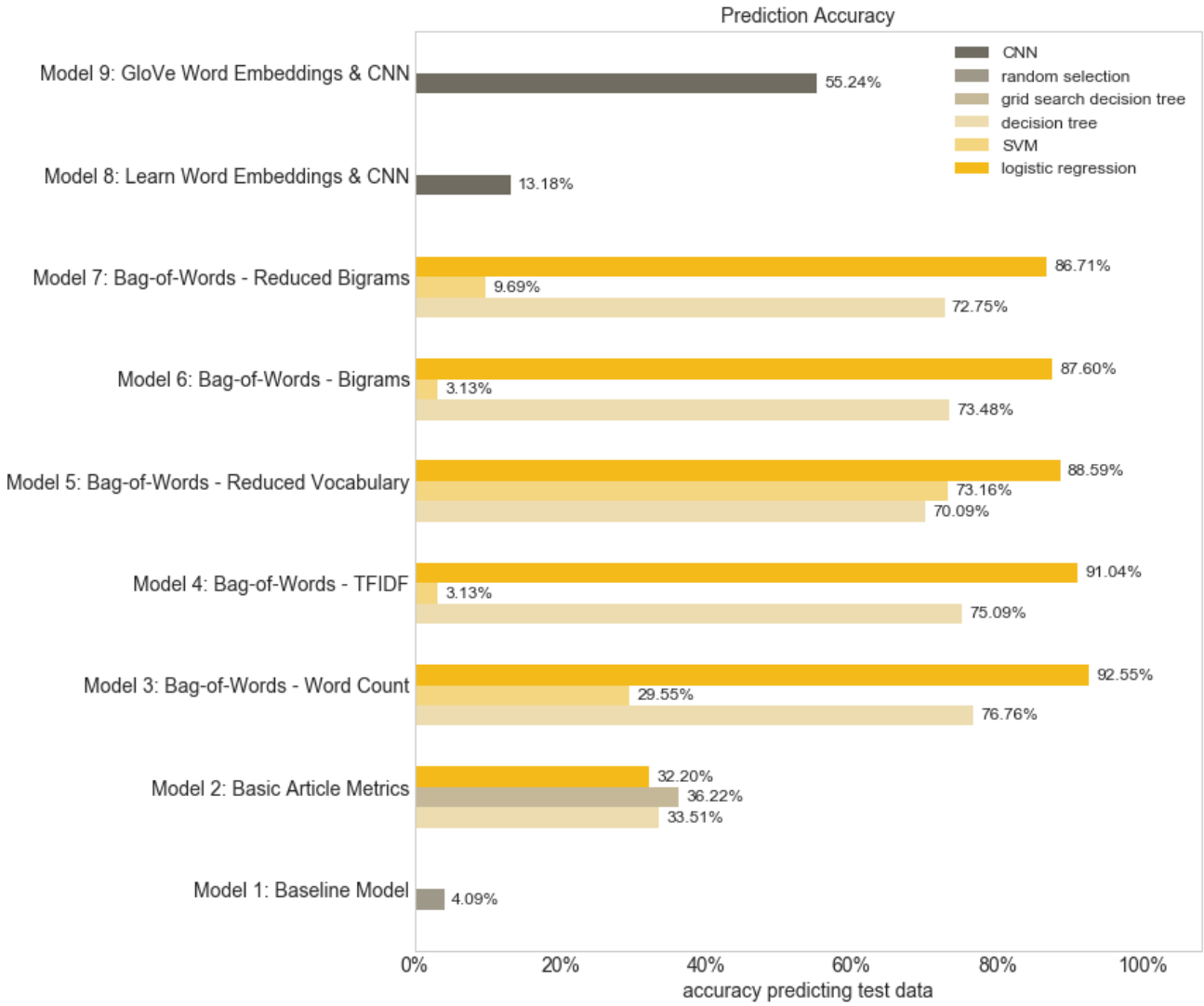
Justification

The prediction capability of model 3 with logistic regression classifier performed significantly better then the benchmark model. Model 3 achieved a prediction accuracy of 92.55% vs. the 3.77% by the benchmark model. The benchmark model performs just a random guess of who the author of an article is.

V. Conclusion

Free-Form Visualization

The chart below summarizes the test data prediction accuracy of the various models we have have investigated in this project.



It is interesting to see that the good old logistic regression classifier in model 3 has achieved a prediction accuracy of 92.55%. With that it has significantly outperformed all other classification approaches. This is a good reminder that, even with todays state of the art algorithms, it's always worth to look into simpler algorithms.

The reason why the more sophisticated models 8 & 9 with word embeddings and CNNs didn't perform better, is probably because of the small size of training data available. Word embeddings require large volumes of training data to pick up the relationship between individual words. For the problem addressed in this project, predicting the authorship of a given text, the amount of training data will always be very limited. As shown in section Data Exploration the most active authors have produced a few hundred articles. Only six have produced 1,000 and more articles. If an application needs to be developed that needs to classify articles into topic categories (sport, gardening , science, ...) then the amount of training data will definitely be much higher as much more authors will write about the same topic. This would make the word embeddings approach worthwhile to revisit.

It is worthwhile to note that model 2 achieved a prediction accuracy above 30%. This model was build only on features reflecting the number of words in articles, paragraphs and sentences. The model does not take anything of the article content into account. In future improvements this model might be used as part of an ensemble learner to improve the prediction accuracy.

Reflection

As mentioned in the previous section it was interesting to see that a model with a simple logistic classifier performed significantly better that more complex models.

Throughout the development of this project some effort has been put into refactoring the code with re-usability in mind. Jupyter notebooks are a great tool for testing approaches and algorithms. Once a promising model has been identified, it is expected that a notebook like this will be handed over to an implementation developer who will transform the code into something that can be shipped to production. Models 3 - 9 have been implemented as Python classes with a consistent interface providing `train`, `predict` and `score` methods. Pre-processing activities have been "packaged" into separate functions which supports reuse and, hopefully, code readability. It is believed that instilling a "production usable" mindset in machine learning engineers will benefit the quick transition of new ideas into usable applications. The last section in the `WhoWroteThis.ipynb` notebook demonstrates the simplicity with which the successful model 3 can be used in a simple application.

An important learning from this project is to keep an eye on testing the different stages of the machine learning pipeline. For all of the models discussed above the raw text from the articles had to be transformed in several stages in order to be used with a classifier or a neural network. The initial text cleaning tasks (removal of punctuation and stop words, stemming, transform all words to lowercase) are fairly straightforward and can be validated by a simple visual inspection of a few samples. It gets a bit more tricky when text is transformed into numbers such as word counts or, even more so, integers that act as pointers to word vectors. Are the numbers I now see are integer indexes or still the word counts from the model I tried earlier? Am I sure that each word/token has now been encoded with its unique integer value? The challenge with machine learning implementation is that bugs not necessarily manifest themselves in hard stops of the code. How do you know whether the reported prediction accuracy of X% is because of the limits of the model or because of a bug in your vectorization code? To address this it is worthwhile for machine learning engineers to be mindful about testability of the different stages of a machine learning system.

Improvement

Model 3 already achieved a remarkable prediction accuracy of 92.55% and increasing this even further might be difficult. Future projects that might want to look into different approaches for vectorization and classification. One approach might be the use of ensemble learning. This approach would require to come up with a series of weak learners that have low individual prediction accuracy but combined might be able to increase the overall prediction accuracy.

Sources

A very helpful resource for this project was the book *"Deep Learning for Natural Language Processing, Develop Deep Learning Models for Natural Language in Python"* from Jason Brownlee. From this book (page 163) the CNN used in models 8 & 9 has been taken. The CNN has been modified to a multiclass classifier (original model was a binary classifier for sentiment analysis).

The following sources served as inspiration for this project and provided valuable guidance for implementing NLP systems.

1 Famous Authors and Their Writing Styles

2 "Who's At The Keyboard? Authorship Attribution in Digital Evidence Investigations"

3 "How a Computer Program Helped Show J.K. Rowling write A Cuckoo's Calling"

4 How to solve 90% of NLP problems: a step-by-step guide

5 scikit learn documentation: confusion matrix

6 Deep Learning for Natural Language Processing, Develop Deep Learning Models for your Natural Language Problems

7 Deep Learning for NLP Best Practices

8 Multi-Class Classification Tutorial with the Keras Deep Learning Library

9 Understanding Convolutional Neural Networks for NLP

10 Logistic Regression for Machine Learning