Derek Pyne
SCS 3252
December 5, 2017

Identifying An Authors Writing DNA?

Predicting horror authors based on their writing style

Motivation

I work for a video messaging company called Helpful (https://www.helpful.com/). We have a web, iOS and android product that lets coworkers communicate with short video messages in the workplace. It is best suited as an engagement tool with teams using it for:

- **Daily updates**: Short videos outlining tasks that were completed yesterday, items outstanding today, and announcements the team should be aware of. Usually done in a group.
- **Acknowledgement**: Managers highlighting a job well done with a direct message.
- **Project discussion**: Using the camera to record a quick explanation or visual description of a problem. Sparks back and forth communication.

For all of these videos, our product automatically generates transcriptions for each video. These transcriptions are tied to the context around them such as the video sender, video recipients, linked media, and past videos in the discussion or group. This is providing us with a large and growing text dataset that we are looking to extract product features from.

Also, other features are being built into the app that would let us classify individuals based on their skillset and we would like to build a model that uses these labeled transcriptions (text tied to the tagged skills of the video sender) to be able to predict associated skills given a users transcriptions.

For this reason, learning how to build a scalable text classification model using a simple dataset will be a great step towards handling future projects at Helpful. Although this dataset is small enough that it could easily be modelled in a traditional Python machine learning stack (Pandas, Numpy, Scikit-learn), I have found this stack hard to scale. In particular, when it comes to larger datasets and hyper-parameter tuning. Using Spark let's us scale nicely to solve both of these problems and is why it was chosen for this project.

The Problem

This classification dataset is provided by Kaggle (https://www.kaggle.com/), an online data science competition community recently acquired by Google. Although this particular competition is categorized as a 'Playground' competition, there is still $25,000 for the best performing submission!

The data consists of text samples for three well known horror authors in the public domain. The labels for each author, as well as the number of observations, is shown in the table below.

The provided text was produced by chunking larger texts into sentences using CoreNLP's MaxEnt sentence tokenizer. This is not perfect so some odd sentences are expected in the dataset.

The goal of the competition is to produce a classifier that predicts the author given a sentence of text.

Author	Label	Count
Edgar Allan Poe	EAP	7900
HP Lovecraft	HPL	5635
Mary Wollstonecraft Shelley	MWS	6044
		19579

Pipeline

To tackle this dataset we will break down our efforts into the OSEMN pipeline. This is common standard for data science problems and includes five steps:

1. Obtain
2. Scrub
3. Extract
4. Model
5. iNterpret

This approach will help keep us organized and focused on the big picture.

Obtain & Scrub

Data is provided by Kaggle in the form of a csv. The first row is a header and the rest are the samples. An extract from the dataset is shown below.

```
“id","text","author"
"id21164","Mein Gott do you take me vor a shicken?"" ""No oh no"" I replied, much alarmed, ""you are no chicken certainly not.""","EAP"
"id08820","Perpetual fear had jaundiced his complexion, and shrivelled his whole person.","MWS"
"id17407","The sun set; the atmosphere grew dim and the evening star no longer shone companionless.","MWS"
"id05166","The rain ceased; the clouds sunk behind the horizon; it was now evening, and the sun descended swiftly the western sky.","MWS"
"id07779","Nothing, however, occurred except some hill noises; and when the day came there were many who hoped that the new horror had gone as swiftly as it had come.","HPL"
"id14749","The state rooms were sufficiently roomy, and each had two berths, one above the other.","EAP"
"id10634","It was getting dark, and the ancient roofs and chimney pots outside looked very queer through the bull's eye
```

We can load this into our DataBricks cluster using the ‘Create Table’ UI. DataBricks provides a database alongside the cluster and automatically puts it into our spark context. This lets any tables added to the database be directly queryable from our notebook.

However, one major limitation with this step was the limited options for csv uploads. Although the interface happily took the provided file, because our data contained comma’s embedded within the text column, text was leaking from the text column into the author column. In the sample provided, the first line shows a case where the quotations and commas in the text were not able to be parsed by the DataBricks tool.

As a workaround, the data was read into a Pandas DataFrame using Pandas `read_from_csv` function. All quotations were then removed from the text column and the dataframe was written back to disc as a csv using pandas `to_csv` function. This produces a file that then loaded nicely into our DataBricks cluster.

Extract

In this situation all of our information is contained in our single text column. We need to extract features suitable for our classification algorithm.

The first step will be tokenization. This is where we break down our text into a list of words. Spark provides a `Tokenizer` class to help with this. It converts all strings to lowercase and then splits into words on whitespace. This is a good place to start but we will have to think whether or not extra normalization is needed. For instance, the tokenizer will split these sentences as follows:

Text	Word
And one part,' too, 'was the hem of the frock' Another piece was 'part of the skirt, not the hem,' that is to say, was torn completely out through the agency of thorns,	[“'and","one","part,'","too,","'was","the","hem","of","the","frock'","another","piece","was","'part","of","the","skirt,","not","the","hem,'","that","is","to","say,","was","torn","completely","out","through","the","agency","of","thorns,"]
How can I thank you, my best and only benefactor?	["'how","can","i","thank","you,","my","best","and","only","benefactor?"]

One thing that becomes immediately clear here is that punctuation is playing a large role. We will remember this and add additional processing later if needed.

Although we now have a sequence of words, this is still not in a form ready for our classifier. We will build our classifier based on the relative frequency of different words in the text. This is a hypothesis that the vocabulary in the text will provide a signature for each author.

To start, we use a `CountVectorizer` to convert our list of words into a vector of token counts. This first builds a dictionary of a size specified by `vocabSize` that consists of the `vocabSize` most common terms. Each list of words is then converted into a term frequency vector of length `vocabSize` by counting the occurrences of each token in the list of words. This creates a large number of sparse features.

However, we have an intuition that some words may contain more information then others. For instance, words like ‘the’, ‘I’, and ‘a’ are so common in the english language that they contain very little information for our classifier. These words are called Stop Words and are often filtered out of the data. We can add a stop word remover into our pipeline and check it’s affect on accuracy.

In addition to the stop words filter, The relative frequency of a word in both in a particular sentence and our corpus as a whole, contains valuable information. Words that occur less frequently tend to be more specific and unique and thus often hold more information. To give these words a boost in our classifier we use Inverse Document Frequency. IDF combines the frequency a term *t,* a document *d*, and the corpus *D*. The document frequency is given by *DF(t, D)* and is the number of documents that contain a particular term. The inverse document frequency is then defined as follows and denotes how much information a given term contains relative to the corpus.

$$IDF(t, D) = log \frac{|D| + 1}{DF(t, D) + 1},$$

We see that because of the logarithm if a term occurs in all documents it is given a value of 0. The additional ‘+1’ terms avoid division by zero for terms that are outside our corpus.

We know have cleaned and extracted features ready for our model.

Model

Bayesian Classifier

For our model we will use a Multinomial Naive Bayes classifier. Naive Bayes classifiers work on Bayes Theorem which determines the probability of a given outcome given the prior probability of the outcome and the likelihood of the outcome given the particular data. Bayes Theorem can be thought of as taking a prior belief, and mixing it with new observations, to produce a posterior probability.

This is close to the way us humans interact with the world around us. We have an intuition with what is normal (our prior), and are constantly faced with new information which we use to adjust our prediction of a given outcome. Bayes Theorem is expressed as,

$$P(A \mid B) = \frac{P(B \mid A)}{P(B)} P(A)$$

where *P(B|A)* is the conditional probability of event B given event A. In our case, it would be the probability of a particular sentence given a particular author. This is the opposite probability of what we want but Bayes Theorem allows us to reverse this probability into P(A|B) which would be the probability of a specific author given a particular sentence. In our situation, Bayes Theorem would be,

$$P(author \mid sentence) = \frac{P(sentence \mid author)}{P(sentence)} P(author)$$

However, we have broken down our sentences into term vectors, this leaves us with,

$$P(author \mid \mathbf{t}) = \frac{P(\mathbf{t} \mid author)}{P(\mathbf{t})} P(author)$$

Since the denominator is not a function of the author we can ignore it. However, it is the conditional probability in the numerator that poses a problem. In general, the probability of a term vector given an author involves not only the probability of each term given the author, but also all the relative probabilities of each term given each other term. This can be expanded with the chain rule but would be unfeasible to compute given our data.

The naive Bayes classifier sidesteps this problem with a strong assumption; that each term in our vector is independent. In other words, the occurrence of a given term has no affect on the probability of another term showing up. To make this concrete, this would assume that the presence of the word 'hockey' in a sentence has no affect on the probability of the word 'ice' appearing. Although this seems unreasonable, it greatly simplifies the equation as our complicated conditional probability now becomes the product of the probability of each individual term given an author, which just so happens to be something that we are very close to producing with our IDF feature vector.

With this assumption the probabilities produced by our model are no longer exact. However, since we are really interested in picking the most likely author of our possibilities, and not the exact probability of each author, we can just compare the size of the probabilities produced by our model to predict an author and not worry too much about the true probability. Historically, although this assumption may seem unreasonable, Bayesian classifiers have a long history of very accurate models and are still widely used in industry today.

The Bayesian classifier provided by MLLib also continues a smoothing parameter. We will tune this parameter in our pipeline.

Hyperparameter Tuning

To take advantage of the distributed nature of Spark, we will adjust some of the parameters in our model and have MLLib chose the best combination. To do this we use a `ParamGridBuilder` which helps manage all the possible combinations of values for our chosen parameters. In our pipeline we will vary the smoothing parameter of our bayesian classifier and the vocabulary size of our count vectorizer.

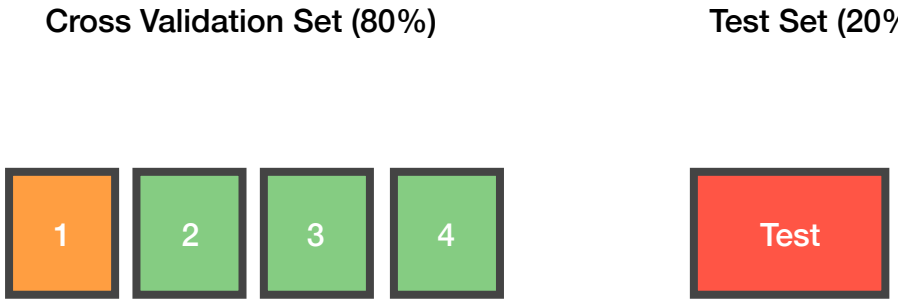
In order to choose the best set of parameters, an evaluation metric is needed. Since our dataset contains only three categories, each of which occurs with similar proportion, a simply accuracy should suffice. We will use the `MulticlassClassificationEvaluator` with the 'accuracy' metric for this.

However, we also want to make sure that our classifier is not overfitting the data and giving us an inflated accuracy metric. We can use cross-validation to combat this by using a separate portion of our dataset for training and testing. All of our parameter

tuning will be done exclusively on the training set with the test set saved for a final unbiased estimate of our classifier. The data was split 80/20.

Within the training set, we still want to make sure that we are not overfitting the data and picking suboptimal parameters. We can use cross-validation to similarly split our data into training and evaluating by splitting it into *k* different sections (also known as folds). We will then reserve one of these sections for testing a given parameter set, and use the rest of the data to fit the model. This gives us a single estimate of our accuracy for this iteration. We can then repeat this process a total of *k* times, switching which section we use for testing. These accuracy estimates are then averaged to produce a less variable estimate of the quality of each set of parameters. The data partitioning is outlined below and 4-folds was chosen.

The diagram show the data partitioning between the cross-validation and test sets as well as the 4 fold split for cross validation. The highlighted orange partition would be the fold chosen to test a set of parameters in this iteration.



Interpret

Overall, our optimal model gave us an accuracy of 76.1% on our test set. The optimal parameters were a smoothing factor of 1.0 and a vocabulary of 3000.

This is a great starting point but opens the door for future improvements such as:

- Bigram (ngram) features: Instead of using single words, we could use pairs (bigrams) or triples (trigrams) of words. This could help pull out signature phrases of each author giving us extra information.
- Sentence structure: The structure of the sentences themselves can tell us about the grammatical style of an author. The length of sentences, average word length, and grammatical devices used could all add additional information.
- Part of Speech Tagging (POS): By highlighting the proportions of adjective, verbs, nouns, etc. in a given sentence, this could identify the type of flow in the authors writing.

Appendix: Sample Code

Below is the code used in the final pipeline. A link to the notebook is also:

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/4275649547278487/894804701504368/6214483029726302/latest.html

```
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.feature import StringIndexer, Tokenizer, HashingTF, IDF, CountVectorizer
from pyspark.ml import Pipeline
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml import Pipeline

labelIndexer = StringIndexer(inputCol="author", outputCol="label")

tokenizer = Tokenizer(inputCol="text", outputCol="words")

countv =  CountVectorizer(inputCol="words", outputCol="rawFeatures", vocabSize=3000, minDF=2.0)

idf = IDF(inputCol="rawFeatures", outputCol="features")

nb = NaiveBayes(smoothing=1.0, modelType="multinomial")

evaluator = MulticlassClassificationEvaluator(metricName="accuracy")

paramGrid = ParamGridBuilder()\
  .addGrid(nb.smoothing, [0.4, 0.8, 1.0, 2.0])\
  .addGrid(countv.vocabSize, [1000, 3000, 5000, 700])\
  .build()

cv = CrossValidator(estimator=nb, evaluator=evaluator, estimatorParamMaps=paramGrid, numFolds=4)

pipeline = Pipeline(stages=[labelIndexer, tokenizer, countv, idf, cv])

train, test = df.randomSplit([0.8, 0.2])
model = pipeline.fit(train)
predictions = model.transform(test)
accuracy = evaluator.evaluate(predictions)

print "Accuracy on our test set: %g" % accuracy
```