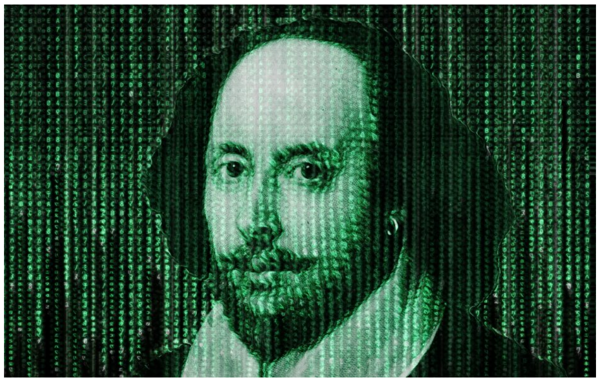
Lab 10: Authorship Detection with Machine Learning



Last week you made a rudimentary author detector. You turned each text into a vector of word counts, and then computed the distance between those two vectors. It even actually worked!

Today we will use the same data and even some of your code from last week, but instead of just straight-up comparing vectors, you will now use machine learning to learn how to put an **importance weight** on each of those vector dimensions.

The goal today is not to teach machine learning, but rather to practice using a really useful Python library. You'll get more comfortable calling other people's code, and hopefully also learn how easy it is to use some machine learning tools.

Step 0: Directory Setup

1. Create a new folder for this lab (lab10)
2. Download the training novels and testing novels.
3. Install the scikit-learn package for machine learning:

```
pip3 install scikit-learn
```

Step 1: From Dictionaries to Matrices

Copy your part3 code from last week to **authorML.py**.

Today we use **scikit-learn**, one of the most widely used machine learning libraries for Python. If you do any Python programming after this class, you will likely run across this toolkit. It provides *many* machine learning algorithms for you, all using the same input/output format for easy use.

Like with any library, if you want to use it, you need to figure out how to **convert your current data into their format**.

Our Current Format: from last week, we have two variables that represent one piece of text:

1. A string label for the author (e.g., 'AUSTEN')
2. A Dictionary of word counts. The Dictionary keys are strings and the values are integers.

Scikit Format: scikit-learn requires its own input format for each input example:

1. An integer label for the 'class' you want to learn (e.g., 3)
2. A vector of numbers (ints or floats) where the vector indices are 'features'

In order to use scikit-learn, our task is to map your (1) to their (1), and your (2) to their (2).

Let's start with the first one: the class. When you want to "learn" something, what are you learning? These are often called classes -- what classes do you hope to predict? If you want to predict the stock market, the classes might be "up" and "down". If you want to identify the object in an image, the classes are "cat", "dog", "horse", etc. Hopefully you see that our author detection task has classes "dickens", "austen", "shakespeare", etc.

Ok so our classes are author names (strings). In order to use scikit, we see it wants **integer** classes. What do we do? Well we just convert our strings to unique integer IDs (e.g., "dickens" is 1, "austen" is 2, etc.). We could program this ourselves and map authors to integers, but Scikit has an object class that does this for you! Here's how it works:

```python
from sklearn.preprocessing import LabelEncoder

# Create an instance of the label encoder class.
LE = LabelEncoder()
# Tell the encoder to 'fit' your author strings (create unique IDs)
# Just give it your full list of authors, repetition allowed!
# This is just an example, don't copy/paste this hard-coded example:
LE.fit( ["dickens", "dickens", "dickens", "shakespeare", "conrad", "eliot"] )

# The encoder has a mapping now, so now transform your list of strings.
# Give it the same list you did to fit()
# This example will produce int IDs: [0, 0, 0, 1, 1, 2]
train_Y = LE.transform( ["dickens", "dickens", "dickens", "conrad", "conrad", "eliot"] )
```

That's it! Why did we do BOTH fit() and transform()? You will only do fit() once in your real program. That saves the mapping from unique strings to unique IDs. The transform() function is called each time you get new data because it uses the unique ID mapping from fit() to then map your data into those IDs.

Hopefully you see that you first need to make a List of your authors in order of the texts that you input. Then call fit() and transform() to make the mapping.

Now for (2). Your Dictionary of word counts is pretty close to a vector of values already. Just like with the author labels, scikit-learn can map Dictionaries to int vectors for us. It's very similar to the label encoder we just did above:

```python
from sklearn.feature_extraction import DictVectorizer

# Create an object instance.
v = DictVectorizer()
# This is just an example List of Dictionaries...
counts = [ {'the':9, 'a':6, 'person':2}, {'the':4, 'a':6, 'person':1}, ... ]
# Find all string keys and create unique integer indices for each.
v.fit(counts)

# Now transform your List of Dictionaries to a matrix using the integer mapping it just 'fit'.
train_X = v.transform(counts)
```

The trick here is that **counts** is a List of Dictionaries. It's all your Dictionaries from your training texts from last week. You want to give scikit all of them so it can create the entire string to int mapping at once. The resulting **train_X** variable is the input that scikit-learn needs.

Ok, I know the above might feel like a lot of details. Please read it again slowly right now.

Required: Your job is to put the above pieces together. You must open the file training-snippets.tsv. Create a List of all Dictionaries from it like last week. Also create a List of authors that align with your Dictionaries. Those two Lists should be the same length, and in the same order. Finally, convert those two Lists to the scikit format as shown above.

Sanity Check: you should now have **train_Y** and **train_X** when done.

Step 2: Train a Machine Learner

You have your training data in the correct format now, right?

Great, let's train a probabilistic classifier called Multinomial Naive Bayes. It computes probabilities of words for each author, and does some fancy probability mass smoothing to generalize the distributions. You don't even need to know what this means! Let's just create one:

```
from sklearn.naive_bayes import MultinomialNB

classifier = MultinomialNB()
classifier.fit(train_X, train_Y)     # "fit" means train the classifier, build the probabilities
```

train_X is your matrix from scikit. **train_Y** is your scikit list of author IDs. They should be the same length, right?

Done! You just trained your first machine learner! It's the **classifier** variable, and now you can use it to *predict* the author on any new text that you might have. How?

```
predictions = classifier.predict(test_X)
print(predictions)
print(LE.inverse_transform(predictions))
```

The variable **test_X** is the same format from our **train_X** above. Your task is thus to read the testing file **test-snippets.tsv** just like you did the training file, convert it to the correct format, and run predict().

Required: open and read in **test-snippets.tsv**, make predictions with MultinomialNB, and print them out with the two print statements above.

```
[5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 2 2 2 2 2 2 6 2 2 2 2 2 2 2 2 2 2 2 2
 2 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 9 1 9 9 9 9 9 9 9 9 9 9 9 9 9 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 8 8 8 8 8 8 8
 8 8 8 8 8 8 3 8 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6]
['HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE'
 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE'
 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE' 'HAWTHORNE'
 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'JAMES'
 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS'
 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'DICKENS' 'SHAKESPEARE'
 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE'
 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE'
 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE' 'SHAKESPEARE'
 'SHAKESPEARE' 'SHAKESPEARE' 'TWAIN' 'CONRAD' 'TWAIN' 'TWAIN' 'TWAIN'
 'TWAIN' 'TWAIN' 'TWAIN' 'TWAIN' 'TWAIN' 'TWAIN' 'TWAIN' 'TWAIN' 'TWAIN'
 'TWAIN' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD'
 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD' 'CONRAD'
 'CONRAD' 'CONRAD' 'CONRAD' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT'
 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT' 'ELIOT'
 'SHAW' 'SHAW' 'SHAW' 'SHAW' 'SHAW' 'SHAW' 'SHAW' 'SHAW' 'SHAW' 'SHAW'
 'SHAW' 'SHAW' 'SHAW' 'ELIOT' 'SHAW' 'HARDY' 'HARDY' 'HARDY' 'HARDY'
 'HARDY' 'HARDY' 'HARDY' 'HARDY' 'HARDY' 'HARDY' 'HARDY' 'HARDY' 'HARDY'
 'HARDY' 'HARDY' 'HARDY' 'HARDY' 'HARDY' 'AUSTEN' 'AUSTEN' 'AUSTEN'
 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN'
 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'AUSTEN' 'JAMES'
 'JAMES' 'JAMES' 'JAMES' 'JAMES' 'JAMES' 'JAMES' 'JAMES' 'JAMES' 'JAMES'
 'JAMES' 'JAMES' 'JAMES' 'JAMES' 'JAMES']
```

Step 3: Compute Performance Accuracy

Now that you have predictions, let's compute the accuracy of your learner. You downloaded si286.py last week, so we will import that again and just call the accuracy() function. **test_Y** are the authors from test-snippets.tsv:

```
si286.accuracy(predictions, test_Y)
```

You should get an author prediction accuracy of 98.2%.

Step 4: (Optional, 5-10% extra credit) Explore Other Learners

The scikit-learn library is extensive and includes many machine learners. Naive Bayes is actually quite basic, but it generalizes well to small datasets like this lab. In this Step, explore the library's webpage and find another type of learner ... use that to learn and make predictions instead of Naive Bayes. Put your solution in a separate extraML.py file.

What to turn in

Visit the submit website and upload authorML.py