Project 1 – Naive Bayes Classification

Evan A Bonsignori

10/6/2017

**1. Classification:**

We seek to classify books into two similar and often difficult to distinguish genres: Science Fiction and Fantasy. Features will be word occurrences (occurs or doesn’t occur) among the 2000 most frequently occurring words in each book (our training example).

**2. Corpus:**

We have a library of over 200 science fiction and fantasy eBooks and have organized them by genre (at least 110 of each) and converted them into utf-8 formatted .txt files to ease preprocessing. We will use 100 of each genre for training and an additional 10 (10%) from each genre for testing.

**3. External Corpora:**

We will be using the NLTK corpus of common English stop words for cleaning the data in the preprocessing phase.

**3. Architecture:**

The project will be implemented in Python 3.6.1 and use the following standard libraries: os, tkinter, random, and time. The appendixes located at the end of the document provide the projects implementation. See Appendix I: Systems Diagram for a block diagram of the initial classifier training and classifier testing programs; Appendix II: Directory Structure for the project directory tree; and Appendix III: Source Code for the project source code.

**4. Functionality**

The four Python programs in the project are:

* rename\_and\_shuffle\_books.py
* initial\_train.py
* performance\_test.py
* classify\_new.py

rename\_and\_shuffle\_books.py organizes the book files (.txt training datasets) by name and number, and shuffles them in their directories. It doesn’t need to be run in the current project.

initial\_train.py trains the Naive Bayesian classifier using 200 genre-labelled books and saves the resulting data structure that represents the classifier in a binary file using *pickle*. Execution of initial\_train.py can take upwards of five minutes depending on your systems processing power, but it only needs to be run once. For ease of use, the project includes an already calculated classifier located in the *saved* directory.

performance\_test.py loads the saved classifier and calculates various performance metrics using labelled test data and labelled book excerpt data. See **7. Performance Analysis** for an elaboration on why book excerpt testing is done. After calculating sensitivity, specificity, and accuracy, the performance metrics are printed to the console.

classify\_new.py prompts the user to either open a text file and classify its genre, or to manually enter text to the console for classification. After choosing the option and following instructions, the program outputs the resulting classification (corresponding genre) and the percent accuracy that the classifier prescribed to the classification.

**5. API’s:**

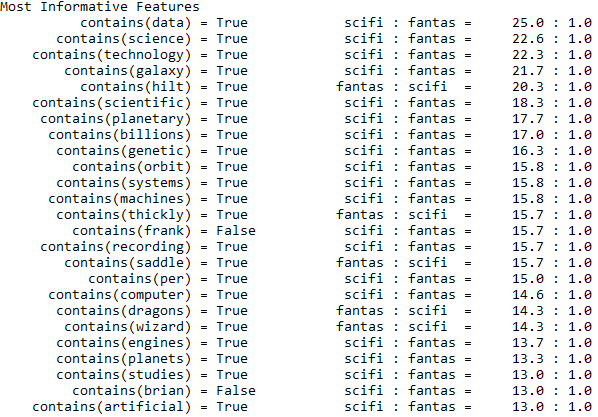
In addition to removing stop words, **NLTK’s** method *word\_tokenize* is used to tokenize books and the class *FreqDist* to find the tokenized word frequencies. The **Pickle** module is used to save Python objects in memory into to a binary formatted file with the extension “.pickle”. This allows us to save the classifier and cleaned data arrays into binary files, and reduces unnecessary computation.

**NLTK’s** *NiaveBayesClassifier* class will be used for classification. The *NiaveBayesClassifier* class works by finding the prior probabilities of each word, given that they are part of one of two genres, and then performances Bayes Formula on each set of data to find the posterior probability of one of the two genres given that a certain word occurs. The machine learning inference is made when the maximum posterior probabilities are applied to a new set of words and the genre with the highest percent of posterior probabilities is selected, thus classifying the book with its (hopefully) correct genre.

**6. Performance:**

The performance specification for our machine learning scenario is genre classification accuracy. We defined the genre “fantasy” to be a positive example and “scifi” to be negative example for purposes of replicating a confusion matrix. Then we counted the corresponding true/false positives/negatives that resulted from classifying our labelled test set. Using these values, we calculated the sensitivity, specificity, and accuracy of our test set and found them to all be 100%. See **7. Performance Analysis** for an explanation of these results.

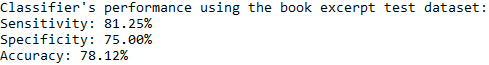
The classifier uses the most informative words (with highest information gain) to classify new documents. There are a total of 88600 features in the currently trained classifier (saved in a .pickle binary file), but the 25 features found in Figure 1 are the most significant (largest posterior probabilities) for classification. The words and their corresponding ratios act as weights for determining if a new book falls into one genre rather than another.

**Figure 1 – Classifier’s Most Informative Features**

**7. Performance Analysis:**

Since books contain thousands of words, the classifier has a lot of data available when classifying, thus providing great accuracy. Given that there are unique words that appear in science fiction or fantasy books and not the other, genre classification is within the scope of a Naive Bayesian classifier. Word independence can be assumed, because extracting frequent words seen in Figure 1, like “robot”, “technology”, and “galaxy” can easily classify a book as sci-fi without needing additional context (i.e. the words to the left or right of each word).

To test my theory that classification accuracy improves with the number of words that occur in a book, I created a separate testing data set. This dataset consists of one to three-word sentences extracted from unclassified sci-fi and fantasy books. I tried to choose strings of sentences that were ambiguous of genre. For instance, if a sentence contained the word “science” and “galaxy” I did not include it for a sci-fi labelled test example. After creating 32 book excerpts, I ran the excerpts through performance\_test.py and performed the same accuracy calculations. The classifier accuracy dropped from 100% to 78.12% (Figure 2).



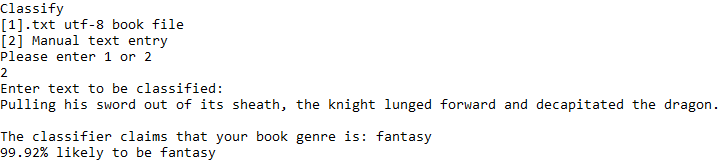
**Figure 2 – Book Excerpt Test Performance**

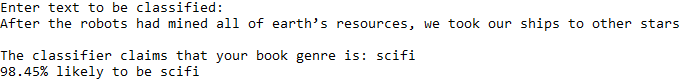
These results are not surprising. The classifier was trained on the 2000 most frequent words in entire books, so using three sentence excerpts is a small subset of the domain that the classifier was trained for. However, since the classifier is just a word-based classifier, sentences are acceptable input. The purpose of the excerpt test set is not to show the inefficiency of the classifier when classifying beyond its domain, but to explain the classifier’s 100% classification accuracy. The classifier isn’t perfect, but it is almost perfect when functioning within its domain. I conjecture that the classifier would yield more than a 95% accuracy if classifying the domain of all science fiction and fantasy books.

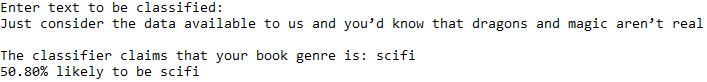
To defend this conjecture, let us assume that since human beings are the writers of sci-fi and fantasy books, human beings can read and correctly identify any book into one of the two genres. Now let us consider the more specific subset of single sentences from a sci-fi or fantasy book. I’ve compiled a list of 15 sentences that can be classified as “likely sci-fi”, “likely fantasy”, or “ambiguous”. Consider just three of them and guess each of their classifications:

1. “Pulling his sword out of its sheath, the knight lunged forward and decapitated the dragon.”
2. “After the robots had mined all of earth’s resources, we took our ships to other stars”
3. “Just consider the data available to us and you’d know that dragons and magic aren’t real.”

If we classify these sentences using classify\_new.py’s manual text entry option, we get the following results shown in Figure 3.



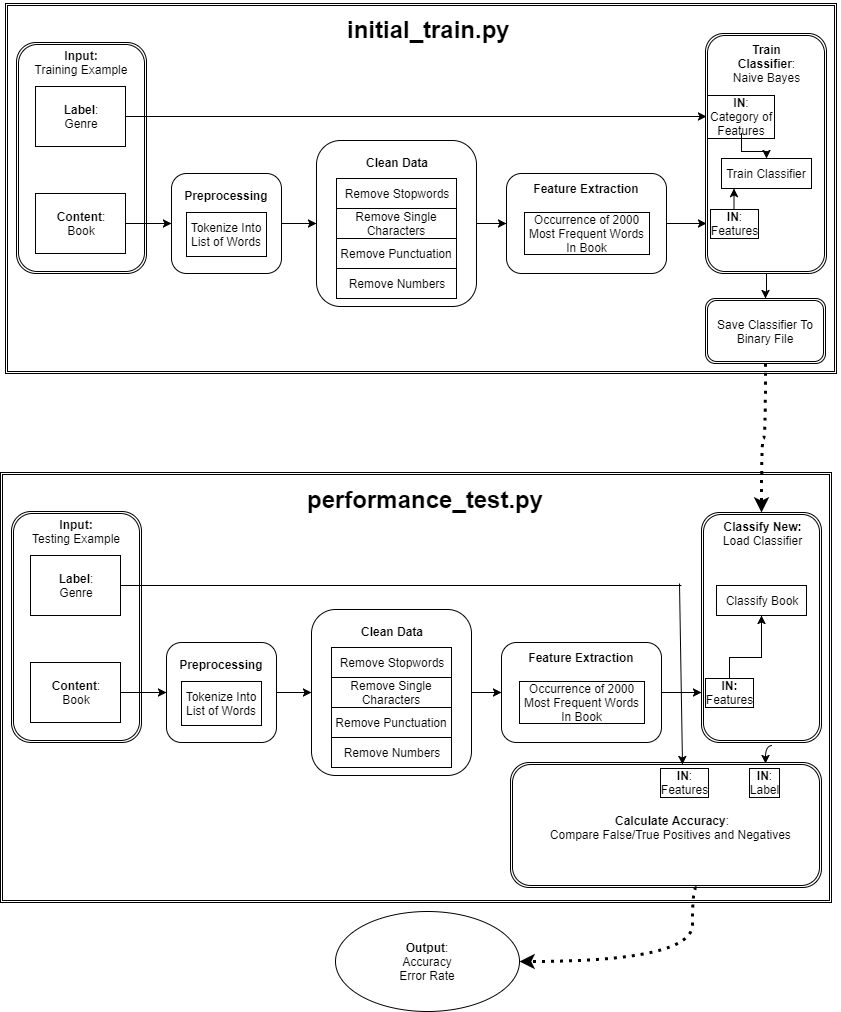




**Figure 3 – Manual Text Entry Classifications**

Note that the last sentence, being intentionally ambiguous, had an accuracy of 50%. Apart from sentence 3, It’s likely that the classifier’s genres matched that of your own guesses. If we accept that the classifier can accurately classify a subset of a book, then given the nature of Bayesian classification, the classification of an entire book follows. Thus, using heuristic methods, we prove our conjecture that our classifier is incredibly accurate in its given domain.

**Appendix I: Systems Diagram**



**Appendix II: Directory Tree**

|-- books

| |-- fantasy

| | |-- txt

| | | |-- fantasy\_book\_0.txt

| | | |-- fantasy\_book\_1.txt

| | | |-- fantasy\_book\_...

| | | |-- fantasy\_book\_15.txt

| |-- scifi

| | |-- txt

| | | |-- scifi\_book\_0.txt

| | | |-- scifi\_book\_1.txt

| | | |-- scifi\_book\_...

| | | |-- scifi\_book\_15.txt

| |-- rename\_and\_shuffle\_boooks.py

|-- book\_excerpts

| |-- fantasy

| | |-- fantasy\_text\_0.txt

| | |-- fantasy\_text\_1.txt

| | |-- fantasy\_text\_...

| | |-- fantasy\_text\_15.txt

| |-- scifi

| | |-- scifi\_text\_0.txt

| | |-- scifi\_text\_1.txt

| | |-- scifi\_text\_...

| | |-- scifi\_text\_15.txt

|-- saved

| |-- classifiers

| | |-- naivebayes\_scifi-fantasy\_classifier.pickle

| |-- datasets

| | |-- scifi-fantasy\_data.pickle

|-- initial\_train.py

|-- performance\_text.py

|-- classify\_new.py

**Appendix III: Source Code**

|-- rename\_and\_shuffle\_books.py

**import** **os**

**import** **random**

categories = list()

*# Get each folder that is named by its book category*

**for** contents **in** os.listdir("./"):

**if** str(contents) != "rename\_and\_shuffle\_books.py":

**if** str(contents) != "snippets":

categories.append(str(contents))

*# Reorder books and rename them in the form: category\_book\_booknumber*

**for** cat **in** categories:

book\_num = 0

current\_dir = "./" + str(cat) + "/txt/"

books\_in\_dir = os.listdir(current\_dir)

random.shuffle(books\_in\_dir)

**for** book **in** books\_in\_dir:

os.rename(current\_dir + book,

current\_dir + str(cat) + "\_book\_" + str(book\_num) +

".txt")

book\_num += 1

|-- Initial\_train.py

**import** **pickle** *# For saving a trained classifier and datasets*

**import** **random** *# To randomize order of training examples*

**import** **time** *# For timing execution of program*

**import** **nltk**

**import** **os**

*# Time program execution*

time\_taken = time.time()

*# Declare variables*

categories = list() *# Categories (genres) to be classified*

books\_loaded = 100 *# Number of training examples in each category*

books = list() *# Tuples of book word lists and genre (book\_words, genre)*

all\_word\_freqs = list() *# Combined genre words and their frequencies*

*# Save functions for classifiers and datasets to save computation time*

**def** save\_classifier(classifier):

**with** open('./saved/classifiers/naivebayes\_scifi-fantasy\_classifier.pickle',

'wb') **as** f:

pickle.dump(classifier, f)

f.close()

**def** save\_data(dataset):

**with** open('./saved/datasets/scifi-fantasy\_data.pickle', 'wb') **as** f:

pickle.dump(dataset, f)

f.close()

*# Define extraneous words with NLTK's default and custom book stop words*

default\_stopwords = set(nltk.corpus.stopwords.words('english'))

custom\_stopwords = set(('sci-fi', 'fantasy', 'page', 'chapter', 'said'))

all\_stopwords = default\_stopwords | custom\_stopwords

*# Populate cat list with categories and cat dict with their respective lengths*

**for** contents **in** os.listdir("./books/"):

**if** str(contents) != "rename\_and\_shuffle\_books.py":

**if** str(contents) != "snippets":

genre = str(contents)

categories.append(genre)

*# Create list of all lists containing a book's words and corresponding label*

**for** genre **in** categories:

directory = "./books/" + genre + "/txt/"

books.extend(

(list(nltk.word\_tokenize(

open(directory + genre +

"\_book\_{}.txt".format(book\_num),

'r', encoding='utf-8').read()

)

),

genre) **for** book\_num **in** range(books\_loaded))

*# Randomize order of training examples and their genres*

random.shuffle(books)

*# Extract features as top 2000 most frequently occurring words in books*

**def** book\_features(book):

*# Find 2000 most frequent words in book*

most\_frequent = nltk.FreqDist(word.lower() **for** word **in** book

**if** **not** word.isnumeric() **and** *# Remove numbers*

word.isalnum() **and** *# Remove punctuation*

len(word) > 1 **and** *# Remove single characters*

word **not** **in** all\_stopwords *# Remove stopwords*

)

most\_frequent = list(most\_frequent)[:2000] *# Only top 2000*

*# Define unique word occurrences in book*

book\_words = set(book)

features = {}

*# For each unique word in the book found in most frequent words in the book*

*# add the 'word' to a dictionary with corresponding key, contains('word')*

**for** word **in** most\_frequent:

features['contains({})'.format(word)] = (word **in** book\_words)

**return** features

*# Separate data into training and test sets and save data for later*

train\_set = [(book\_features(book), genre) **for** (book, genre) **in** books]

save\_data(train\_set)

*# Classify data using a Naive Bayes Classifier and save classifier for later*

classifier = nltk.NaiveBayesClassifier.train(train\_set)

save\_classifier(classifier)

*# Print time taken for program execution*

**print**("Time taken for " + str(books\_loaded) + " books from each genre to train **\**

the Bayesian classifier: **\n**" + str(time.time() - time\_taken) + " seconds")

|-- performance\_test.py

**import** **pickle** *# For loading a trained classifier*

**import** **nltk**

**import** **os**

*# Load functions for classifiers and datasets to save computation time*

**def** load\_classifier():

**with** open('./saved/classifiers/naivebayes\_scifi-fantasy\_classifier.pickle',

'rb') **as** f:

classifier = pickle.load(f)

f.closed

**return** classifier

*# Declare variables*

categories = list() *# Categories (genres) to be classified*

books\_loaded = 10 *# Number of training examples in each category*

books = list() *# Tuples of book word lists and genre (book\_words, genre)*

excerpts = list() *# Tuples of word lists from book excerpts and their genre*

all\_word\_freqs = list() *# Combined genre words and their frequencies*

*# Define extraneous words with NLTK's default and custom book stop words*

default\_stopwords = set(nltk.corpus.stopwords.words('english'))

custom\_stopwords = set(('sci-fi', 'fantasy', 'page', 'chapter', 'said'))

all\_stopwords = default\_stopwords | custom\_stopwords

*# Populate category list with the genres*

**for** contents **in** os.listdir("./books/"):

**if** str(contents) != "rename\_and\_shuffle\_books.py":

**if** str(contents) != "snippets":

genre = str(contents)

categories.append(genre)

*# Create list of tuples of each book's tokenized words in a list and their*

*# corresponding genre. In the form [ ([book\_words], genre) ] where [] = list*

*# and () = tuple*

**for** genre **in** categories:

directory = "./books/" + genre + "/txt/"

books.extend(

*# Start at book #99 as previous 99 were used to train*

(list(nltk.word\_tokenize(

open(directory + genre +

"\_book\_{}.txt".format(

book\_num + 100),

'r', encoding='utf-8').read()

)

),

genre) **for** book\_num **in** range(books\_loaded))

*# Create tuples for book excerpts and the genre they were taken from*

**for** genre **in** categories:

directory = "./book\_excerpts/" + genre + "/"

excerpts.extend(

*# Extract 16 excerpts from each category for 32 total*

(list(nltk.word\_tokenize(

open(directory + genre +

"\_text\_{}.txt".format(

excerpt\_num),

'r').read()

)

),

genre) **for** excerpt\_num **in** range(16))

*# Extract features as top 2000 most frequently occurring words in books*

**def** book\_features(book):

*# Find 2000 most frequent words in book*

most\_frequent = nltk.FreqDist(word.lower() **for** word **in** book

**if** **not** word.isnumeric() **and** *# Remove numbers*

word.isalnum() **and** *# Remove punctuation*

len(word) > 1 **and** *# Remove single characters*

word **not** **in** all\_stopwords *# Remove stopwords*

)

most\_frequent = list(most\_frequent)[:2000] *# Only top 2000*

*# Define unique word occurances in book*

book\_words = set(book)

features = {}

*# For each unique word in the book found in most frequent words in the book*

*# add the 'word' to a dictionary with corresponding key, contains('word')*

**for** word **in** most\_frequent:

features['contains({})'.format(word)] = (word **in** book\_words)

**return** features

*# Separate test data into a list of tuples for classifier*

test\_set = [(book\_features(book), genre) **for** (book, genre) **in** books]

*# Load NaiveBayes Classifier*

classifier = load\_classifier()

*# Performance Metric Measurements*

positive = "fantasy" *# Assume fantasy to be positive*

negative = "scifi" *# Assume Sci-fi to be negative*

TP = 0 *# True Positive*

FP = 0 *# False Positive*

FN = 0 *# False Negative*

TN = 0 *# True Negative*

*# Classify the test set and count the resulting true/false positives/negatives*

**for** test\_book **in** test\_set:

classified\_genre = classifier.classify(test\_book[0])

correct\_genre = test\_book[1]

*# if true and a positive example increment true positive count*

**if** classified\_genre == correct\_genre **and** classified\_genre == positive:

TP += 1

*# if false and a positive example increment false positive count*

**elif** classified\_genre != correct\_genre **and** classified\_genre == positive:

FP += 1

*# if true and a negative example increment true negative count*

**elif** classified\_genre == correct\_genre **and** classified\_genre == negative:

TN += 1

*# if false and a negative example increment false negative count*

**elif** classified\_genre != correct\_genre **and** classified\_genre == negative:

FN += 1

sensitivity = round((TP / (TP + FN)) \* 100, 2)

specificity = round((TN / (TN + FP)) \* 100, 2)

accuracy = round(((TN + TP) / (TN + TP + FN + FP)) \* 100, 2)

**print**("Classifier's performance metrics using the book test dataset: **\n**" +

"Sensitivity: {:0.2f}% **\n**Specificity: {:0.2f}% **\n**Accuracy: {:0.2f}%"

.format(sensitivity, specificity, accuracy))

*# Repeat process for excerpt dataset*

excerpt\_set = [(book\_features(xcerpt), genre) **for** (xcerpt, genre) **in** excerpts]

TP = 0 *# True Positive*

FP = 0 *# False Positive*

FN = 0 *# False Negative*

TN = 0 *# True Negative*

*# Classify the test set and count the resulting true/false positives/negatives*

**for** test\_excerpt **in** excerpt\_set:

classified\_genre = classifier.classify(test\_excerpt[0])

correct\_genre = test\_excerpt[1]

*# if true and a positive example increment true positive count*

**if** classified\_genre == correct\_genre **and** classified\_genre == positive:

TP += 1

*# if false and a positive example increment false positive count*

**elif** classified\_genre != correct\_genre **and** classified\_genre == positive:

FP += 1

*# if true and a negative example increment true negative count*

**elif** classified\_genre == correct\_genre **and** classified\_genre == negative:

TN += 1

*# if false and a negative example increment false negative count*

**elif** classified\_genre != correct\_genre **and** classified\_genre == negative:

FN += 1

sensitivity = round((TP / (TP + FN)) \* 100, 2)

specificity = round((TN / (TN + FP)) \* 100, 2)

accuracy = round(((TN + TP) / (TN + TP + FN + FP)) \* 100, 2)

**print**("Classifier's performance using the book excerpt test dataset:**\n**" +

"Sensitivity: {:0.2f}% **\n**Specificity: {:0.2f}% **\n**Accuracy: {:0.2f}%"

.format(sensitivity, specificity, accuracy))

num\_of\_features = len(classifier.\_feature\_probdist.items())

**print**("The classifiers 10 most informative features (total of {} features): "

.format(num\_of\_features))

classifier.show\_most\_informative\_features(25)

|-- classify\_new.py

**import** **pickle** *# For saving a trained classifier and datasets*

**import** **tkinter** **as** **tk**

**from** **tkinter** **import** filedialog **as** fd *# For prompting user for book location*

**import** **nltk**

**print**("Classify a new:**\n**[1].txt utf-8 book file **\n**[2] Manual text entry")

**while** True:

option = input('Please enter 1 or 2 **\n**')

**try**:

option = int(option)

**except** **ValueError**:

**print**('Valid number, please')

**continue**

**if** option == 1 **or** option == 2:

**break**

**else**:

**print**('Invalid Option')

**if** (option == 1):

root = tk.Tk()

root.withdraw()

file\_path = fd.askopenfilename()

**with** open(file\_path, 'r', encoding='utf-8') **as** f:

book = nltk.word\_tokenize(f.read())

**else**:

book = input("Enter text to be classified: **\n**")

book = nltk.word\_tokenize(book)

*# Load classifier*

**def** load\_classifier():

**with** open('./saved/classifiers/naivebayes\_scifi-fantasy\_classifier.pickle',

'rb') **as** f:

classifier = pickle.load(f)

f.closed

**return** classifier

*# Define extraneous words with NLTK's default and custom book stop words*

default\_stopwords = set(nltk.corpus.stopwords.words('english'))

custom\_stopwords = set(('sci-fi', 'fantasy', 'page', 'chapter', 'said'))

all\_stopwords = default\_stopwords | custom\_stopwords

*# Extract features as top 2000 most occurring words in books*

**def** book\_features(book):

*# Find 2000 most frequent words in book*

most\_frequent = nltk.FreqDist(word.lower() **for** word **in** book

**if** **not** word.isnumeric() **and** *# Remove numbers*

word.isalnum() **and** *# Remove punctuation*

len(word) > 1 **and** *# Remove single characters*

word **not** **in** all\_stopwords *# Remove stopwords*

)

most\_frequent = list(most\_frequent)[:2000] *# Only top 2000*

*# Define unique word occurrences in book*

book\_words = set(book)

features = {}

*# For each unique word in the book found in most frequent words in the book*

*# add the 'word' to a dictionary with corresponding key, contains('word')*

**for** word **in** most\_frequent:

features['contains({})'.format(word)] = (word **in** book\_words)

**return** features

*# Load Naive Bayes Classifier*

classifier = load\_classifier()

extracted\_features = book\_features(book)

genre = classifier.classify(extracted\_features)

probability = classifier.prob\_classify(extracted\_features).prob(genre)

*# Print accuracy of classifier with respect to the test set*

**print**("**\n**The classifier claims that your book genre is: " + genre)

**print**("{:.2f}**% li**kely to be ".format(probability \* 100) + genre)