Lab 08: Supervised Text Classification (Part 1)

**References:**

1. Regular Expressions: The Complete Tutorial, by Jan Goyvaerts, 2007.
2. Speech and Language Processing, by Dan Jurafsky and James H. Martin. Prentice Hall Series in Artificial Intelligence, 2008.
3. Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper, 2014.

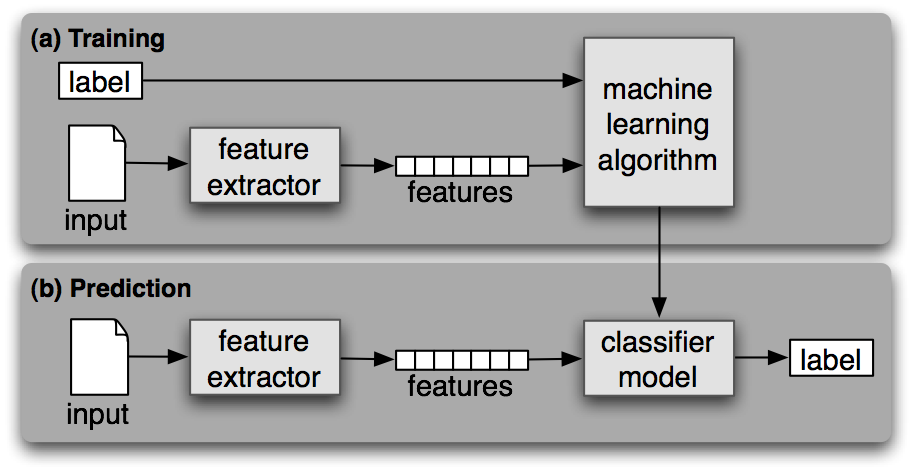
Quick Review

Classification is the task of choosing the correct class label for a given input. In basic classification tasks, each input is considered in isolation from all other inputs, and the set of labels is defined in advance. Some examples of classification tasks are:

* Deciding whether an email is spam or not.
* Deciding what the topic of a news article is, from a fixed list of topic areas such as "sports," "technology," and "politics."
* Deciding whether a given occurrence of the word bank is used to refer to a river bank, a financial institution, the act of tilting to the side, or the act of depositing something in a financial institution.

The basic classification task has a number of interesting variants. For example, in multi-class classification, each instance may be assigned multiple labels; in open-class classification, the set of labels is not defined in advance; and in sequence classification, a list of inputs are jointly classified.

A classifier is called **supervised** if it is built based on training corpora containing the correct label for each input. The framework used by supervised classification is shown below:



1. During training, a feature extractor is used to convert each input value to a feature set. These feature sets, which capture the basic information about each input that should be used to classify it, are discussed in the next section. Pairs of feature sets and labels are fed into the machine learning algorithm to generate a model.
2. During prediction, the same feature extractor is used to convert unseen inputs to feature sets. These feature sets are then fed into the model, which generates predicted labels.

Spelling Correction

***# using Autocorrect (pip install autocorrect)***

from autocorrect import Speller

spell = Speller(lang='en')

spell('caaaar')

spell('mussage')

spell('survice')

spell('hte')

***# using Textblob***

from textblob import TextBlob

b1 = TextBlob("I havv goood speling!")

print(b1.correct())

b2 = TextBlob("caaaar")

print(b2.correct())

***# using pip install pyspellchecker***

from spellchecker import SpellChecker

spell = SpellChecker()

# find those words that may be misspelled

misspelled = spell.unknown(['something', 'is', 'hapenning', 'here'])

*# Get the one `most likely` answer*

for word in misspelled:

print(spell.correction(word))

*# Get a list of `likely` options*

print(spell.candidates(word))

Grammar and Spell Check using Python

# pip install language-check

# https://pypi.org/project/language-check/

import language\_check

tool = language\_check.LanguageTool('en-US')

text = "he are a human"

match = tool.check(text)

print(match)

print(len(match))

language\_check.correct(text, match)

# https://pypi.org/project/language-tool-python/

# pip install language-tool-python

import language\_tool\_python

tool = language\_tool\_python.LanguageTool('en-US')

text = "he are a human"

matches = tool.check(text)

print(len(matches))

tool.correct(text)

Practices

Text/Document Classification (Gender Identification)

In “Natural Language Processing with Python” (the third reference above), in the Section 2.4, we can see that male and female names have some distinctive characteristics. Names ending in *a*, *e* and *i* are likely to be female, while names ending in *k*, *o*, *r*, *s* and *t* are likely to be male. Let's build a classifier to model these differences more precisely.

The first step in creating a classifier is deciding what **features** of the input are relevant, and how to **encode** those features. For this example, we'll start by just looking at the final letter of a given name. The following **feature extractor** function builds a dictionary containing relevant information about a given name:

def gender\_features(word):

return {'last\_letter': word[-1]}

gender\_features('Shrek')

The returned dictionary, known as a **feature set**, maps from feature names to their values. Feature names are case-sensitive strings that typically provide a short human-readable description of the feature, as in the example 'last letter'. Feature values are values with simple types, such as Booleans, numbers, and strings.

Now that we've defined a feature extractor, we need to prepare a list of examples and corresponding class labels.

from nltk.corpus import names

labeled\_names = ([(name, 'male') for name in names.words('male.txt')] +

[(name, 'female') for name in names.words('female.txt')])

import random

random.shuffle(labeled\_names)

Next, we use the feature extractor to process the names data, and divide the resulting list of feature sets into a **training set** and a **test set**. The training set is used to train a new "naive Bayes" classifier.

import nltk

featuresets = [(gender\_features(n), gender) for (n, gender) in labeled\_names]

train\_set, test\_set = featuresets[500:], featuresets[:500]

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

You can do more about the naive Bayes classifier later. For now, let's just test it out on some names that did not appear in its training data:

print("Neo is a", classifier.classify(gender\_features('Neo')))

print("Trinity is a", classifier.classify(gender\_features('Trinity')))

Observe that these character names from *The Matrix* are correctly classified. Although this science fiction movie is set in 2199, it still conforms to our expectations about names and genders. We can systematically evaluate the classifier on a much larger quantity of unseen data:

print("\nThe accuracy is equal to: ", nltk.classify.accuracy(classifier, test\_set))

Finally, we can examine the classifier to determine which features it found most effective for distinguishing the names' genders:

classifier.show\_most\_informative\_features(5)

This listing shows that the names in the training set that end in "a" are female 33 times more often than they are male, but names that end in "k" are male 32 times more often than they are female. These ratios are known as **likelihood ratios**, and can be useful for comparing different feature-outcome relationships.

When working with large corpora, constructing a single list that contains the features of every instance can use up a large amount of memory. In these cases, use the function nltk.classify.apply\_features, which returns an object that acts like a list but does not store all the feature sets in memory:

from nltk.classify import apply\_features

train\_set = apply\_features(gender\_features, labeled\_names[500:])

test\_set = apply\_features(gender\_features, labeled\_names[:500])

Text/Document Classification (Sentiment Analysis - Movie Review)

There are several examples of corpora where documents have been labelled with categories. Using these corpora, we can build classifiers that will automatically tag new documents with appropriate category labels. First, we construct a list of documents, labelled with the appropriate categories. For this example, we've chosen the Movie Reviews Corpus, which categorizes each review as positive or negative.

import nltk

import random

from nltk.corpus import movie\_reviews

documents = [(list(movie\_reviews.words(fileid)), category)

for category in movie\_reviews.categories()

for fileid in movie\_reviews.fileids(category)]

len(documents)

for row in range(0, len(documents)-1):

print(documents[row][1])

random.shuffle(documents)

print(documents[1])

Next, we define a feature extractor for documents, so the classifier will know which aspects of the data it should pay attention to.

For document topic identification, we can define a feature for each word, indicating whether the document contains that word. Remove print(documents[1]) and add the following lines to the code:

all\_words = []

for w in movie\_reviews.words():

wl = w.lower()

all\_words.append(wl)

all\_words = nltk.FreqDist(all\_words)

print(all\_words.most\_common(15))

Remove print(all\_words.most\_common(15)) and add the following lines to the code:

print(all\_words["good"])

print(all\_words["excellent"])

To limit the number of features that the classifier needs to process, we begin by constructing a list of the 2000 most frequent words in the overall corpus. We can then define a feature extractor that simply checks whether each of these words is present in a given document.

word\_features = list(all\_words)[:2000]

We can then define a feature extractor that simply checks whether each of these words is present in a given document.

def find\_features(document):

words = set(document)

features = {}

for w in word\_features:

features[w] = (w in words)

return features

You can test the feature extractor, by:

print((find\_features(movie\_reviews.words('neg/cv000\_29416.txt'))))

Now that we've defined our feature extractor, we can use it to train a classifier to label new movie reviews.

featureset = [(find\_features(rev), category) for (rev, category) in documents]

train\_set, test\_set = featuresets[100:], featuresets[:100]

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

classifier.show\_most\_informative\_features(5)

To check how reliable the resulting classifier is, we compute its accuracy on the test set. And once again, we can use show\_most\_informative\_features() to find out which features the classifier found to be most informative.

print(nltk.classify.accuracy(classifier, test\_set))

classifier.show\_most\_informative\_features(5)

In both the situations, we can implement algorithms such as Decision Tree, Sklearn Classifier, Support Vector Machine (SVC) and Maximum Entropy (MaxEnt) under the NLTK package.

For this kind of classification problems, we can also use one of the following classifiers from the **Sklearn** package.

Dataset and Pre-processing is crucial to make the data shape suitable for the algorithms chosen.

* [KNeighborsClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier)(3)
* [SVC](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC)(kernel="linear", C=0.025)
* [SVC](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC)(gamma=2, C=1)
* [GaussianProcessClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessClassifier.html#sklearn.gaussian_process.GaussianProcessClassifier)(1.0 \* [RBF](https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RBF.html#sklearn.gaussian_process.kernels.RBF)(1.0))
* [DecisionTreeClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier)(max\_depth=5)
* [RandomForestClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier)(max\_depth=5, n\_estimators=10, max\_features=1)
* [MLPClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier)(alpha=1, max\_iter=1000)
* [AdaBoostClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier)()
* [GaussianNB](https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB)()
* LogisticRegression()
* SGDClassifier()

***Refer:***

<https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html#sphx-glr-download-auto-examples-classification-plot-classifier-comparison-py>