Machine Learning

L715/B659

Dept. of Linguistics, Indiana University
Fall 2016

Where we're going

From Data to Classification

We want to take raw, messy text data & classify it, generally within a supervised learning framework

- We're going to focus on Python & Python-based tools
- We'll work from this tutorial: https://www.kaggle.com/c/word2vec-nlp-tutorial/details/part-1-for-beginners-bag-of-words

1. From raw data to usable raw data
   - pandas
   - Beautiful Soup
2. From usable data to meaningful units
   - NLTK
3. From meaningful units to features
   - scikit-learn (or just Python)
4. From features to classification
   - scikit-learn

Python

I'm going to assume some basic familiarity with Python (http://python.org)

- You'll want to know some basics of text processing
- The NLTK references later can help ...

```
>>> s='All I can say is, "My life is pretty plain."'
>>> s.lower()
'all i can say is, "my life is pretty plain."'
>>> s.split()
['All', 'I', 'can', 'say', 'is,', '"My', 'life',
 'is', 'pretty', 'plain."']
>>> "#".join(s.split())
'All#I#can#say#is,#"My#life#is#pretty#plain."'
>>> set(s.split())
{'is', 'plain."', 'is,', 'say', 'pretty', 'I',
 'life', 'All', 'can', '"My'}
```

pandas: Python Data Analysis Library

`pandas` provides utilities for data file storage & manipulation (http://pandas.pydata.org)

1. Install: e.g., `sudo pip install pandas`
2. Import: e.g., `import pandas as pd`
3. Use, e.g.,:

```
>>> train=pd.read_csv("labeledTrainData.tsv", \
...           header=0, delimiter="\t", quoting=3)

>>> train
        id sent  review
0 "5814_8"    1 "With all this stuff going down at the
1 "2381_9"    1 "\"The Classic War of the Worlds\" by T
2 "7759_3"    0 "The film starts with a manager (Nichol
3 "3630_4"    0 "It must be assumed that those who prai
4 "9495_8"    1 "Superbly trashy and wondrously unprete
5 "8196_8"    1 "I dont know why people think this is s
```

pandas

> *pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive.*
> *(http://pandas.pydata.org/pandas-docs/stable/, retrieved 7/26/16)*

`pandas` allows one to work with data frames (cf. R) and to easily examine the data

```
>>> train.shape
>>> train.columns.values
```

We won't deal too much with `pandas`

Cleaning Data: Beautiful Soup

`BeautifulSoup` is for cleaning up data, e.g., webpages (https://www.crummy.com/software/BeautifulSoup/)

1. Install: `pip install beautifulsoup4`
2. Import: `from bs4 import BeautifulSoup`
3. Create a BeautifulSoup object with the text in question, e.g., `soup=BeautifulSoup(html_doc,'html.parser')`
4. Do any number of things with this text:
   - Better view the XML/HTML structure: `.prettify()`
   - View some of the structured HTML contents: `.title.string`, `.find_all(a)`
   - **Get the raw text**: `.get_text()`

Example

From the documentation (https://www.crummy.com/software/BeautifulSoup/bs4/doc/):

```
>>> html_doc = """
... <html><head><title>The Dormouse's story</title></head>
... <body>
... <p class="title"><b>The Dormouse's story</b></p>
...
... <p class="story">Once upon a time there were three litt
... <a href="http://example.com/elsie" class="sister" id="l
... <a href="http://example.com/lacie" class="sister" id="l
... <a href="http://example.com/tillie" class="sister" id="
... and they lived at the bottom of a well.</p>
...
... <p class="story">...</p>
... """
```

Creating an object

```
>>> from bs4 import BeautifulSoup

>>> soup = BeautifulSoup(html_doc, 'html.parser')
```

Accessing HTML information


```
>>> soup.title
<title>The Dormouse's story</title>

>>> soup.title.string
"The Dormouse's story"

>>> soup.find_all('a')
[<a class="sister" href="http://example.com/elsie" id="link
<a class="sister" href="http://example.com/lacie" id="link
<a class="sister" href="http://example.com/tillie" id="link
```

Getting text

```
>>> print(soup.get_text())

The Dormouse's story

The Dormouse's story
Once upon a time there were three little sisters;
and their names were
Elsie,
Lacie and
Tillie;
and they lived at the bottom of a well.
...
```

Extracting Meaningful Units: NLTK

Natural Language Toolkit (NLTK) is:

> *... a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, and an active discussion forum.*

http://www.nltk.org/

Installing NLTK is mostly straightforward:

- http://nltk.org/install.html

Getting started

Download the materials from the NLTK book:

```
>>> import nltk
>>> nltk.download()
...
Downloader> d book
...
```

This command gives us various texts to work with, which we need to load:

```
>>> from nltk.book import *
```

NLTK useful utilities

You can use NLTK for many NLP & text processing tasks.

- We'll focus on two basic ones, so you won't have to redo them:
  - `word_tokenize`: tokenize into meaningful linguistic units (i.e., tokens)

    ```
    >>> nltk.word_tokenize(s)
    ['All', 'I', 'can', 'say', 'is', ',', '``', 'My',
     'life', 'is', 'pretty', 'plain', '.', "''"]
    ```

  - Stop words

    ```
    from nltk.corpus import stopwords
    print(stopwords.words("english"))
    ```

Stop word removal

```
>>> words = [w for w in words
               if not w in stopwords.words("english")]
>>> words
['All', 'I', 'say', ',', '``', 'My', 'life',
 'pretty', 'plain', '.', "''"]
```

Note that, for our purposes, it may be the stop words that we are interested in ...

Regular expressions
(how to handle punctuation)

```
>>> letters_only = re.sub("[^a-zA-Z]",
...                       " ",
...                       s.lower())
>>> letters_only
'all i can say is   my life is pretty plain  '

>>> words = letters_only.split()
>>> words
['all', 'i', 'can', 'say', 'is', 'my', 'life',
'is', 'pretty', 'plain']

>>> words = [w for w in words
             if not w in stopwords.words("english")]
>>> words
['say', 'life', 'pretty', 'plain']
```

Extracting features & classifying: scikit-learn

scikit-learn (http://scikit-learn.org/) is a machine learning package in Python

Install (http://scikit-learn.org/stable/install.html):

- `pip install -U scikit-learn`
    - You should already have `numpy` & `scipy` installed

We'll use this tutorial:
http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

- This blog seemed helpful, too: http://billchambers.me/tutorials/2015/01/14/python-nlp-cheatsheet-nltk-scikit-learn.html

Data format

Abstractly:

- Start with a list of strings, one for each item to be classified (e.g., document)
- Finish with an $n \times m$ matrix of n documents & m features

Tutorial data

If you can't find scikit-learn's tutorial data, download it from:

- https://github.com/scikit-learn/scikit-learn

To run `fetch_data.py` requires `lxml` ... which itself requires `libxml2` & `libxslt`, e.g.,

1. `sudo port install libxml libxslt`
2. `sudo pip install lxml`

Tutorial data (cont.)

Walking through the **Loading the 20 newsgroups dataset** part of the tutorial ...

- Note that `twenty_train` is a dictionary

```
>>> twenty_train.keys()
dict_keys(['target_names', 'filenames', 'target',
           'description', 'data', 'DESCR'])
```

  (For convenience, they've also created objects, e.g., `twenty_train.target_names`)

- `twenty_train['data']` is a list of documents
  - Note how all `data` (documents) & `target` (class ID) correspond, as does `filenames`

```
>>> print(twenty_train['data'][0])
From: sd345@city.ac.uk (Michael Collier)
Subject: Converting images to HP LaserJet III?
Nntp-Posting-Host: hampton
...
```

Feature extraction

Bag of words: `CountVectorizer`

`CountVectorizer` is a tool to calculate bags of words

- http://scikit-learn.org/stable/modules/feature_extraction.html

Example:

```
from sklearn.feature_extraction.text import CountVectorizer

count_vect=CountVectorizer()
X_train_counts=count_vect.fit_transform(twenty_train.data)

print(X_train_counts.shape)  # (2257, 35788)
```

(`X_train_counts` is a matrix of 2257 documents × 35,788 features (words))

Feature extraction

Bag of words: `CountVectorizer`

`count_vect.vocabulary_` allows you to see the ID associated with each word

```
print(count_vect.vocabulary_)
print(count_vect.vocabulary_.get('algorithm'))
```

We can then look up counts in specific documents:

```
>>> X_train_counts[0,4690]
0
>>> X_train_counts[2207,4690]
2
>>> X_train_counts[2241,4690]
1
```

Feature extraction

When you've done your own preprocessing

Note the difference in the Kaggle tutorial:

```
vectorizer = CountVectorizer(analyzer = "word",   \
                             tokenizer = None,    \
                             preprocessor = None, \
                             stop_words = None,   \
                             max_features = 5000)

# fit_transform() does two functions:
# First, it fits the model and learns the vocabulary;
# second, it transforms our training data
# into feature vectors. ...
train_data_features=vectorizer.fit_transform(clean_train_re
```

Feature extraction

When you've done your own preprocessing (2)

And note the additional step:

```
# Numpy arrays are easy to work with,
# so convert the result to an array
train_data_features = train_data_features.toarray()
```

See the Kaggle tutorial also for a nice way to sum up the counts of each word

Feature extraction

tf-idf

Another option is to use tf-idf (term frequency - inverse document frequency)

```
from sklearn.feature_extraction.text import TfidfTransforme

# fit estimator to data:
tf_transformer=TfidfTransformer(use_idf=False).fit(X_train_
# transform counts to tf-idf
X_train_tf=tf_transformer.transform(X_train_counts)

print(X_train_tf.shape)      # (2257, 35788)
print(X_train_tf[0,4690])    # 0.0
print(X_train_tf[2241,4690]) # 0.07352146220938072
print(X_train_tf[2207,4690]) # 0.06401843996644798 8
```

Feature extraction

tf-idf (2)

All in one go (and `use_idf` is no longer `False`):

```
tfidf_transformer = TfidfTransformer()
X_train_tfidf=tfidf_transformer.fit_transform(X_train_count
print(X_train_tfidf.shape)
```

Training a classifier

With the data properly in place, training a classifier is straightforward:

```
from sklearn.naive_bayes import MultinomialNB

clf = MultinomialNB().fit(X_train_tfidf,
                          twenty_train.target)
```

Classifying new documents

Prediction on a couple of short documents

- Note the use of `transform` instead of `fit_transform` (already fit to training data)

```
>>> docs_new = ['God is love',
                'OpenGL on the GPU is fast']
>>> X_new_counts=count_vect.transform(docs_new)
>>> X_new_tfidf=tfidf_transformer.transform(X_new_counts)
>>> predicted=clf.predict(X_new_tfidf)
>>> predicted
array([3, 1])
>>> for doc, category in zip(docs_new, predicted):
...   print('%r => %s' % (doc,twenty_train.target_names[cat
...
'God is love' => soc.religion.christian
'OpenGL on the GPU is fast' => comp.graphics
```

Building a pipeline

A `Pipeline` class allows for combining steps:

```
>>> from sklearn.pipeline import Pipeline
>>> text_clf=Pipeline([('vect',CountVectorizer()),
...                    ('tfidf',TfidfTransformer()),
...                    ('clf',MultinomialNB()),
... ])
```

Training is then straightforward:

```
>>> text_clf=text_clf.fit(twenty_train.data,
...                        twenty_train.target)
```

Evaluation

```
# create a test set & get the raw data of it
twenty_test = fetch_20newsgroups(subset='test',
              categories=categories, shuffle=True,
              random_state=42)
docs_test = twenty_test.data

# predict on the test data
predicted = text_clf.predict(docs_test)

# print(len(twenty_test.target)) # 1502
# print(twenty_test.target)      # [2 2 2 ..., 2 2 1]
# print(len(predicted))          # 1502
# print(predicted)               # [2 2 3 ..., 2 2 1]

# get the accuracy: 0.834886817577
print(np.mean(predicted == twenty_test.target))
```

Evaluation

Finer-grained evaluation


```
>>> from sklearn import metrics
>>> print(metrics.classification_report(twenty_test.target,
       predicted, target_names=twenty_test.target_names))
             precision    recall  f1-score   support

alt.atheism       0.97      0.60      0.74       319
...graphics       0.96      0.89      0.92       389
    sci.med       0.97      0.81      0.88       396
..christian       0.65      0.99      0.78       398

avg / total       0.88      0.83      0.84      1502
```

Check out the documentation for confusion matrices & more ...

Parameter tuning

`scikit-learn` offers utilities for finding the best (hyper)parameters for a model

- See the examples using `GridSearchCV`
- Watch out for expensive computation!

Practice problem

Set up a simple genre classifier using scikit-learn

1. Download the SUSANNE corpus (http://www.grsampson.net/Resources.html)
   - This will unpack into an fc2/ directory
2. Use the first 8 files of each genre (A, G, J, N) as training, next 4 as development, final 4 as testing
   - A = press reportage
   - G = belles lettres, biography, memoirs
   - J = learned (mainly scientific and technical) writing
   - N = adventure and Western fiction
3. Extract what seem to be relevant features
   - Columns: 3 = POS tag (class); 4 = word; 5 = lemma, 6 = syntactic functional information
   - Hand-examine some files first ...
4. Classify, tweaking parameters & options

SUSANNE

SUSANNE is a **corpus**: a collected body of text

- Each line corresponds to a word, with many other properties associated with it
- i.e., if you read it vertically, you can see what the text is (try `cut -f4 FILENAME` to get just the plain text)

This is a linguistically **annotated** corpus

- Someone has gone through and added part-of-speech & syntactic information (by hand)
- Most of our data will not be so nicely hand-annotated
  - But: we'll have automatic tools to give us much of this functionality

Input: SUSANNE file

```
A01:0010.03  -  YB      <minbrk>  -        [Oh.Oh]
A01:0010.06  -  AT      The      the       [O[S[Nns:s.
A01:0010.09  -  NP1s    Fulton   Fulton    [Nns.
A01:0010.12  -  NNL1cb  County   county    .Nns]
A01:0010.15  -  JJ      Grand    grand     .
A01:0010.18  -  NN1c    Jury     jury      .Nns:s]
A01:0010.21  -  VVDv    said     say       [Vd.Vd]
A01:0010.24  -  NPD1    Friday   Friday    [Nns:t.Nns:t]
A01:0010.27  -  AT1     an       an        [Fn:o[Ns:s.
A01:0010.30  -  NN1n    investigation      investigation    .
A01:0020.03  -  IO      of       of        [Po.
A01:0020.06  -  NP1t    Atlanta  Atlanta   [Ns[G[Nns.Nns]
A01:0020.09  -  GG      +<apos>s           -        .G]
A01:0020.12  -  JJ      recent   recent    .
A01:0020.15  -  JJ      primary  primary   .
A01:0020.18  -  NN1n    election           election         .Ns]Po]
A01:0020.21  -  VVDv    produced           produce  [Vd.Vd]
A01:0020.24  -  YIL     <ldquo>  -         .
A01:0020.27  -  ATn     +no      no        [Ns:o.
A01:0020.30  -  NN1u    evidence           evidence         .
```

What is the task?

Task: determine which of 4 categories a new document falls into

- class $\in$ {A, G, J, N}
  - Check what the classifier assumes
- Analysis is on a **per document** level
  - i.e., each feature **vector** refers to a whole document

Feature exploration?

What types of features could be relevant for genre classification?

- (normalized) counts of pronouns?
- average sentence length? word length?
- (normalized) punctutation counts?
- (normalized) counts of all content words?
- measure of lexical diversity (e.g., type-token ratio)?
- ...

Obtaining features

Question: How do we go from SUSANNE files to output representation?

- **Answer:** Use your favorite programming language!
  - Feel free to share corpus-reading code with each other
- Also: Unix tricks can help
  - e.g., `cut -f4 G01 | grep -ci '^he$'` gives 6 as the count of *he* in file G01
  - See Kenneth Church's *Unix for Poets* (http://www.cs.upc.edu/~padro/Unixforpoets.pdf)

Obtaining features (2)

For some systems, you do your own indexing

Consider if you wanted to use the count of every known word as a feature

- Every feature is assigned a number:
  - In training, assign a number to each word
  - In testing: read the same mapping, to assign features

More complicated features

Discuss: How would we obtain/encode these features?

- Counts of bigrams of tags - i.e., two-tag sequences (e.g., AT NP1s, NP1s NNL1cb, etc.)
- The most frequent tag in the document (e.g., NN1n)
- The 10 most frequent words in the document
- Type-token ratio
  - type = abstract idea of a word
  - token = actual instance (e.g., 7 word tokens of the word type *he* in A01)

Feature selection

How do you know which features are helping or hurting?

- Some systems provide output indicating which features are treated as more important
  - e.g., using information gain to calculate
- **Ablation** experiments on development data
  - Remove a feature or set of features & observe new classification accuracy
  - And/or build feature sets from the ground up