Applying Machine Learning to Sentiment Analysis

Hsi-Pin Ma 馬席彬

http://lms.nthu.edu.tw/course/40724

Department of Electrical Engineering

National Tsing Hua University

Outline

- Preparing the IMDb Movie Review Data for Text Processing
- Introducing the Bag-of-words Model
- Training a Logistic Regression Model for Document Classification
- Topic Modeling

Natural Language Processing (NLP)

- Computer are great at working with ***standardized and structure data***, while humans communicate using words, a form of ***unstructured data***
- NLP is a subfield of AI focusing on enabling computers to understand and process human languages

Sentiment Analysis

- A subfield of natural language processing (NLP)
- Also called opinion mining
- Concerned with analyzing the polarity of documents
- Using machine learning algorithms to classify documents based on the expressed opinions or emotions of the authors regard to a particular topic

Internet Movie Database (IMDb)

- 50,000 movie reviews labeled as positive/ negative collected by Mass et. al. in 2011
  - Positive: more than six stars on IMDb
  - Negative: fewer than five stars on IMDb
- Link
  - http://ai.stanford.edu/~amaas/data/sentiment/
- Download and preprocessing
  - movie_data.csv

After Preprocessing

```
import pandas as pd

df = pd.read_csv('movie_data.csv', encoding='utf-8')
df.head(3)
```

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Introducing the Bag-of-words Model

Bag-of-Words Model

- To represent text as numerical feature vectors
  - Create a vocabulary (alphabet) of unique tokens from the entire set of documents (e.g. words)
  - Assign an integer index to each token
  - Construct a feature vector from each document that contains the counts of how often each word occurs in the particular document
- The feature vectors are sparse

Transforming Documents into Feature Vectors

- In scikit-learn, the bag-of-words model is implemented as the **CountVectorizer** class in the **feature_extraction** text module
- By calling fit_transform method on **CountVectorizer**, we just constructed the vocabulary of the bag-of-words model and transformed the following three sentences into sparse vectors
  - The sun is shinning
  - The weather is sweet
  - The sun is shinning, the weather is sweet, and one and one is two.

Transforming Documents into Feature Vectors

Construct the vocabulary (bag: 3x9 sparse matrix)

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer()
docs = np.array([
        'The sun is shining',
        'The weather is sweet',
        'The sun is shining, the weather is sweet, and one and one is two'])
bag = count.fit_transform(docs)
```

Index of the vocabulary

```
print(count.vocabulary_)
```

```
{'the': 6, 'sun': 4, 'is': 1, 'shining': 3, 'weather': 8, 'sweet': 5, 'and': 0, 'one': 2
, 'two': 7}
```

Feature vectors

```
print(bag.toarray())
```

```
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Raw term frequencies: tf(t,d): (values in the feature vectors)

The number of times a term t occurs in a document d

Transforming Documents into Feature Vectors

- n-grams model
  - Each item or token in the vocabulary represents n words
  - Choice of n depends on the particular applications
- For example, for "the sun is shinning"
  - 1-gram: "the", "sun", "is", "shinning"
  - 2-gram: "the sun", "sun is", "is shinning"
- In **CountVectorizer**, set **ngram_range** parameter
  - 1-gram is by default
  - For 2-gram, **ngram_range**=(2,2)

Assessing Word Relevancy via Term Frequency-Inverse Document Frequency

- There are words that frequently occur across multiple documents from both classes, which typically don't contain useful or discriminatory information
- Term frequency-inverse document frequency (tf-idf) can be used to downweight those frequently occurring words in the feature vectors

$$\text{tf-idf}(t,d) = tf(t,d) \times \text{idf}(t,d)$$

$$\text{idf}(t,d) = log \frac{n_d}{1+\text{df}(d,t)}$$

total number of documents

number of documents ***d*** that contain the term ***t***

Assessing Word Relevancy via Term Frequency-Inverse Document Frequency

- Scikit-learn implements **TfidfTransformer** that makes the raw term frequencies from **CountVectorizer** as input and transforms them into tf-idfs

```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf = TfidfTransformer(use_idf=True,
                         norm='l2',
                         smooth_idf=True)
print(tfidf.fit_transform(count.fit_transform(docs))
      .toarray())
```

```
[[ 0.    0.43  0.    0.56  0.56  0.    0.43  0.    0.  ]
 [ 0.    0.43  0.    0.    0.    0.56  0.43  0.    0.56]
 [ 0.5   0.45  0.5   0.19  0.19  0.19  0.3   0.25  0.19]]
```

Assessing Word Relevancy via Term Frequency-Inverse Document Frequency

- **TfidfTransformer** has an option to normalize tf-idfs directly
  - By default, norm=l2

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}$$

  - If norm=None, will not normalize the tf-idfs

```
tfidf = TfidfTransformer(use_idf=True, norm=None, smooth_idf=True)
raw_tfidf = tfidf.fit_transform(count.fit_transform(docs)).toarray()[-1]
raw_tfidf
```

```
array([ 3.39,  3.  ,  3.39,  1.29,  1.29,  1.29,  2.  ,  1.69,  1.29])
```

```
l2_tfidf = raw_tfidf / np.sqrt(np.sum(raw_tfidf**2))
l2_tfidf
```

Manually normalize

```
array([ 0.5 ,  0.45,  0.5 ,  0.19,  0.19,  0.19,  0.3 ,  0.25,  0.19])
```

Cleaning Text Data

- The first important step before build the bag-of-words model is to clean the text data by stripping it of all unwanted characters
- Can use ***regular expressions*** to search for unwanted patterns of characters
  - A regular expression (regex or regexp) is a sequence of characters that define a search pattern
  - Each character in a regex is either a metacharacter, having a special meaning, or a regular character that has a literal meaning
  - https://docs.python.org/3/library/re.html

Metacharacters in Regular Expression

- . (dot): match any character except a newline
- *: match 0 or more repetitions of the preceding RE
- +: match 1 or more repetitions of the preceding RE
- ?: match 0 or 1 repetitions of the preceding RE
- {m}: match exactly m copies of the previous RE
- {n,m}: match from n to m copies of the previous RE
- [abc]: match to one of the three characters a, b, c
- [^a]: indicate any character which is not a
- A | B: either A or B
- \ : either escape special characters, or signals a special sequence
  - \w: Match unicode word characters
  - \W: Match any character which is not a word character

Cleaning Text Data

Display last 50 characters from the 1st document

```
df.loc[0, 'review'][-50:]
```

```
' would be money well spent.<br /><br />8 out of 10'
```

Use Python's regular expression (re) library to remove HTML markup except emotion characters

```
import re
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text)
    text = (re.sub('[\W]+', ' ', text.lower()) +
            ' '.join(emoticons).replace('-', ''))
    return text
```

```
preprocessor(df.loc[0, 'review'][-50:])
```

```
' would be money well spent 8 out of 10'
```

```
preprocessor("</a>This :) is :( a test :-)!")
```

```
'this is a test :) :( :)'
```

Apply preprocessor function to all the movie reviews

```
df['review'] = df['review'].apply(preprocessor)
```

Processing Documents into Tokens

- Split the text corpora into individual elements

```
tokenizer('runners like running and thus they run')
```

```
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

- Word stemming: Transform a word into its root form
  - The Poter stemming algorithm developed by Martin F. Poter in 1979
  - The Natural Language Toolkit (NLTK) for Python

```
from nltk.stem.porter import PorterStemmer

porter = PorterStemmer()

def tokenizer(text):
    return text.split()


def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]
```

```
tokenizer_porter('runners like running and thus they run')
```

```
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Processing Documents into Tokens

- Stop-word removal
  - Stop-words convey very little information and can be removed
    - is, and, has, like
    - 127 English stop-words in NLTK library

```
import nltk

nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/sebastian/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
True
```

```
from nltk.corpus import stopwords

stop = stopwords.words('english')
[w for w in tokenizer_porter('a runner likes running and runs a lot')[-10:]
if w not in stop]
```

```
['runner', 'like', 'run', 'run', 'lot']
```

Training a Logistic Regression Model for Document Classification

Train a Logistic Regression Model

- Classify the movie reviews into positive and negative reviews
- Strip HTML and punctuation to speed up the GridSearch later
- 25000 for training and 25000 for testing

```
X_train = df.loc[:25000, 'review'].values
y_train = df.loc[:25000, 'sentiment'].values
X_test = df.loc[25000:, 'review'].values
y_test = df.loc[25000:, 'sentiment'].values
```

Use GridSearchCV to Find Optimal Parameters

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import GridSearchCV

tfidf = TfidfVectorizer(strip_accents=None,
                        lowercase=False,
                        preprocessor=None)
param_grid = [{'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'clf__penalty': ['l1', 'l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              {'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'vect__use_idf':[False],
               'vect__norm':[None],
               'clf__penalty': ['l1', 'l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              ]

lr_tfidf = Pipeline([('vect', tfidf),
                     ('clf', LogisticRegression(random_state=0))])

gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
                           scoring='accuracy',
                           cv=5,
                           verbose=1,
                           n_jobs=-1)
```

Note about the Running Time

- Execute the code is time consuming (~30-60mins)
  - With the parameter grid, 22*235+222*35=240 models to fit
- Possible approaches to speedup
  - Reduce the size of dataset by decreasing the number of training samples. However, may result in poorly performing model

```
X_train = df.loc[:2500, 'review'].values
y_train = df.loc[:2500, 'sentiment'].values
```

  - Delete the parameters from the grid to reduce the models to fit

```
param_grid = [{'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer],
               'clf__penalty': ['l1', 'l2'],
               'clf__C': [1.0, 10.0]},
              ]
```

Use GridSearchCV to Find Optimal Parameters

```
print('Best parameter set: %s ' % gs_lr_tfidf.best_params_)
print('CV Accuracy: %.3f' % gs_lr_tfidf.best_score_)
```

```
Best parameter set: {'clf__C': 10.0, 'clf__penalty': 'l2', 'vect__ngram_range': (1, 1),
'vect__stop_words': None, 'vect__tokenizer': <function tokenizer at 0x7f781f0bd0d0>}
CV Accuracy: 0.892
```

```
clf = gs_lr_tfidf.best_estimator_
print('Test Accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Test Accuracy: 0.899
```

Online Algorithms and Out-of-core Learning

- In many real world applications, it is common to work with even larger datasets that can exceed the computer's memory
  - Out-of-core learning: Fitting the classifier incrementally on smaller branches of the dataset
- Use **partial_fit** function of **SGDClassifier** to stream the documents directly from local drive, and train a logistic regression model using small mini-branches of documents

Tokenizer and Stream Documents

```
import numpy as np
import re
from nltk.corpus import stopwords

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) +\
        ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized


def stream_docs(path):
    with open(path, 'r', encoding='utf-8') as csv:
        next(csv)  # skip header
        for line in csv:
            text, label = line[:-3], int(line[-2])
            yield text, label
```

Verification of Document Stream

```
next(stream_docs(path='movie_data.csv'))
```

```
('"In 1974, the teenager Martha Moxley (Maggie Grace) moves to the high-class area of Be
lle Haven, Greenwich, Connecticut. On the Mischief Night, eve of Halloween, she was murd
ered in the backyard of her house and her murder remained unsolved. Twenty-two years lat
er, the writer Mark Fuhrman (Christopher Meloni), who is a former LA detective that has
fallen in disgrace for perjury in O.J. Simpson trial and moved to Idaho, decides to inve
stigate the case with his partner Stephen Weeks (Andrew Mitchell) with the purpose of wr
iting a book. The locals squirm and do not welcome them, but with the support of the ret
ired detective Steve Carroll (Robert Forster) that was in charge of the investigation in
the 70\'s, they discover the criminal and a net of power and money to cover the murder.<
br /><br />""Murder in Greenwich"" is a good TV movie, with the true story of a murder o
f a fifteen years old girl that was committed by a wealthy teenager whose mother was a K
ennedy. The powerful and rich family used their influence to cover the murder for more t
han twenty years. However, a snoopy detective and convicted perjurer in disgrace was abl
e to disclose how the hideous crime was committed. The screenplay shows the investigatio
n of Mark and the last days of Martha in parallel, but there is a lack of the emotion in
the dramatization. My vote is seven.<br /><br />Title (Brazil): Not Available"',
 1)
```

Get Mini batch of Documents

```
def get_minibatch(doc_stream, size):
    docs, y = [], []
    try:
        for _ in range(size):
            text, label = next(doc_stream)
            docs.append(text)
            y.append(label)
    except StopIteration:
        return None, None
    return docs, y
```

Classifier Definition

```
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier

vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)

clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
doc_stream = stream_docs(path='movie_data.csv')
```

Model Training and Testing

```
import pyprind
pbar = pyprind.ProgBar(45)

classes = np.array([0, 1])
for _ in range(45):
    X_train, y_train = get_minibatch(doc_stream, size=1000)
    if not X_train:
        break
    X_train = vect.transform(X_train)
    clf.partial_fit(X_train, y_train, classes=classes)
    pbar.update()
```

```
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:00:31
```

```
X_test, y_test = get_minibatch(doc_stream, size=5000)
X_test = vect.transform(X_test)
print('Accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Accuracy: 0.867
```

```
clf = clf.partial_fit(X_test, y_test)
```

Topic Modeling

Topic Modeling

- Assign topics to unlabeled text documents in a corpus of news articles, scientific articles, emails, web pages, blog posts, and so on.
- Considered as a clustering task, a subcategory of unsupervised learning
- Latent Dirichlet Allocation (LDA) is a popular technique for this
  - It builds a topic per document model and words per topic model, modeled as Dirichlet distributions
  - Not to be confused with Linear Discriminant Analysis

Latent Dirichlet Allocation (LDA)

- A generative probabilistic model that tries to find groups of words that appear frequently together across different documents
  - These frequently appearing words represent the topics, assuming each document is a mixture of different words
- Given a bag-of-words matrix as input LDA decompose it into two new matrices
  - A document to topic matrix
  - A topic to word matrix
- If we multiply those two matrices together, we would be able to reproduce the input with the lowest possible error.
- However, we must define the number of topics (a hyperparameter of LDA to be specified manually) beforehand

Preprocessing

```
import pandas as pd

df = pd.read_csv('movie_data.csv', encoding='utf-8')
df.head(3)
```

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

```
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer(stop_words='english',
                        max_df=.1,
                        max_features=5000)
X = count.fit_transform(df['review'].values)
```

LDA

```
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_topics=10,
                                random_state=123,
                                learning_method='batch')
X_topics = lda.fit_transform(X)
```

```
lda.components_.shape
```

```
(10, 5000)
```

LDA Results

```
n_top_words = 5
feature_names = count.get_feature_names()

for topic_idx, topic in enumerate(lda.components_):
    print("Topic %d:" % (topic_idx + 1))
    print(" ".join([feature_names[i]
                    for i in topic.argsort()\
                        [:-n_top_words - 1:-1]]))
```

```
Topic 1:
worst minutes awful script stupid
Topic 2:
family mother father children girl
Topic 3:
american war dvd music tv
Topic 4:
human audience cinema art sense
Topic 5:
police guy car dead murder
Topic 6:
horror house sex girl woman
Topic 7:
role performance comedy actor performances
Topic 8:
series episode war episodes tv
Topic 9:
book version original read novel
Topic 10:
action fight guy guys cool
```

Topic Guess

```
Topic 1:
worst minutes awful script stupid
Topic 2:
family mother father children girl
Topic 3:
american war dvd music tv
Topic 4:
human audience cinema art sense
Topic 5:
police guy car dead murder
Topic 6:
horror house sex girl woman
Topic 7:
role performance comedy actor performances
Topic 8:
series episode war episodes tv
Topic 9:
book version original read novel
Topic 10:
action fight guy guys cool
```

1. Generally bad movies (not really a topic category)
2. Movies about families
3. War movies
4. Art movies
5. Crime movies
6. Horror movies
7. Comedies
8. Movies somehow related to TV shows
9. Movies based on books
10. Action movies

Verification of Horror Movies

```
horror = X_topics[:, 5].argsort()[::-1]

for iter_idx, movie_idx in enumerate(horror[:3]):
    print('\nHorror movie #%d:' % (iter_idx + 1))
    print(df['review'][movie_idx][:300], '...')
```

```
Horror movie #1:
House of Dracula works from the same basic premise as House of Frankenstein from the yea
r before; namely that Universal's three most famous monsters; Dracula, Frankenstein's Mo
nster and The Wolf Man are appearing in the movie together. Naturally, the film is rathe
r messy therefore, but the fact that ...

Horror movie #2:
Okay, what the hell kind of TRASH have I been watching now? "The Witches' Mountain" has
got to be one of the most incoherent and insane Spanish exploitation flicks ever and yet
, at the same time, it's also strangely compelling. There's absolutely nothing that make
s sense here and I even doubt there  ...

Horror movie #3:
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a total freakfest from
```