SentimentAnalysis-checkpoint

January 1, 2025

- Note - Don't run the cells as a live demo - some tasks can take 10 minutes or longer…

1 Text Classification

- applying machine learning to classifiy natural language for various tasks
- a comprehensive article on Text Classification: https://arxiv.org/pdf/2004.03705.pdf
- some common text classification tasks:
  1. sentiment analysis
  2. news categorization
  3. topic analysis
  4. question answering (QA)
  5. natural language inference (NLI)

1.1 Sentiment Analysis

- subfield of **natural language processing (NLP)**
- also called **opinion mining**
- apply ML algorithms to classify documents based on their polarity:
  - the attitude of the writer

1.2 General steps

1. clean and prepare text data
2. build feature vectors from text documents
3. train a machine learning model to classify positive and negative movie reviews
4. test and evaluate the model

1.3 IMDb dataset

- contains 50,000 labeled moview reviews from Internet Moview Database (IMDb)
- task is to classify reviews as **positive** or **negative**
- compressed archive can be downloaded from: http://ai.stanford.edu/~amaas/data/sentiment

1.3.1 Download and untar IMDb dataset

- on Linux and Mac use the following cells
- on Windows, manually download the archive and untar using 7Zip or other application
- or use the provided Python code

[6]:
```
%%bash
# let's download the file
# FYI - file is ~ 84 MB; may take a while depending on Internet speed...
# extracting files from tar file may take even longer...
dirPath=data
fileName=aclImdb_v1.tar.gz
url=http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
if [ -f "$dirPath/$fileName" ]; then
    echo "File $dirPath/$fileName exists."
else
    echo "File $dirPath/$fileName does not exist. Downloading from $url..."
    mkdir -p "$dirPath"
    curl -o "$dirPath/$fileName" "$url"
    cd $dirPath
    tar -xf "$fileName"
fi
```

```
File data/aclImdb_v1.tar.gz exists.
```

[7]:
```
# let's see the contents of the data folder
! ls data
```

```
aclImdb            aclImdb_v1.tar.gz
```

[2]:
```
# let's untar the compressed aclImdb_v1.tar.gz file
! tar -zxf data/aclImdb_v1.tar.gz --directory data
```

1.3.2 Python code to download and extract tar file

- this can take a while depending on the Internet speed...

[5]:
```
import os
import sys
import tarfile
import time
import urllib.request


source = 'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
target = 'data/aclImdb_v1.tar.gz'

def reporthook(count, block_size, total_size):
    global start_time
    if count == 0:
        start_time = time.time()
        return
    duration = time.time() - start_time
    progress_size = int(count * block_size)
    speed = progress_size / (1024**2 * duration)
```

```
    percent = count * block_size * 100 / total_size

    sys.stdout.write("\r%d%% | %d MB | %.2f MB/s | %d sec elapsed" %
                    (percent, progress_size / (1024**2), speed, duration))
    sys.stdout.flush()


if not os.path.isdir('data/aclImdb') and not os.path.isfile(target):
    urllib.request.urlretrieve(source, target, reporthook)
```

[6]:
```
# untar the file
if not os.path.isdir('data/aclImdb'): # if the directory doesn't exist untar␣
↪the target to path
    with tarfile.open(target, 'r:gz') as tar:
        tar.extractall(path="./data")
```

1.3.3 Preprocess the movie dataset into a more convenient format

- extract and load the movie dataset into Pandas DataFrame
- NOTE: can take up to **10 minutes** on a PC
- use Pthon Progress Indicator (PyPrind) package to show progress bar from Python code

[3]:
```
! pip install pyprind
```

```
Collecting pyprind
  Using cached PyPrind-2.11.2-py3-none-any.whl (8.6 kB)
Installing collected packages: pyprind
Successfully installed pyprind-2.11.2
```

[3]:
```
import pyprind
import pandas as pd
import os

# change the `basepath` to the directory of the
# unzipped movie dataset

basepath = 'data/aclImdb'

labels = {'pos': 1, 'neg': 0}
pbar = pyprind.ProgBar(50000)
df = pd.DataFrame()
for s in ('test', 'train'):
    for l in ('pos', 'neg'):
        path = os.path.join(basepath, s, l)
        for file in sorted(os.listdir(path)):
            with open(os.path.join(path, file),
                      'r', encoding='utf-8') as infile:
                txt = infile.read()
```

```
            df = df.append([[txt, labels[l]]],
                           ignore_index=True)
            pbar.update()
df.columns = ['review', 'sentiment']
```

```
0% [##############################] 100% | ETA: 00:00:00
Total time elapsed: 00:01:34
```

1.3.4 Shuffle and save the assembled data as CSV file

- pickle the DataFrame as a binary file for faster load

[4]:
```
import pandas as pd
import numpy as np
import pickle
```

[5]:
```
np.random.seed(0)
df = df.reindex(np.random.permutation(df.index)) # randomize files
```

[6]:
```
df
```

[6]:
```
                                                  review  sentiment
11841  In 1974, the teenager Martha Moxley (Maggie Gr…          1
19602  OK… so… I really like Kris Kristofferson a…          0
45519  ***SPOILER*** Do not read this, if you think a…          0
25747  hi for all the people who have seen this wonde…          1
42642  I recently bought the DVD, forgetting just how…          0
…                                                    …        …
21243  OK, lets start with the best. the building. al…          0
45891  The British 'heritage film' industry is out of…          0
42613  I don't even know where to begin on this one. …          0
43567  Richard Tyler is a little boy who is scared of…          0
2732   I waited long to watch this movie. Also becaus…          1

[50000 rows x 2 columns]
```

[7]:
```
# save csv format
df.to_csv('data/movie_data.csv', index=False, encoding='utf-8')
```

[8]:
```
# save DataFrame as a pickle dump
pickle.dump(df, open('data/movie_data.pd', 'wb'))
```

[9]:
```
# directly load the pickled file as DataFrame
df = pickle.load(open('data/movie_data.pd', 'rb'))
```

[10]:
```
df
```

[10]:
```
                                                  review  sentiment
11841  In 1974, the teenager Martha Moxley (Maggie Gr…          1
19602  OK… so… I really like Kris Kristofferson a…          0
45519  ***SPOILER*** Do not read this, if you think a…          0
25747  hi for all the people who have seen this wonde…          1
42642  I recently bought the DVD, forgetting just how…          0
…                                                    …        …
21243  OK, lets start with the best. the building. al…          0
45891  The British 'heritage film' industry is out of…          0
42613  I don't even know where to begin on this one. …          0
43567  Richard Tyler is a little boy who is scared of…          0
2732   I waited long to watch this movie. Also becaus…          1

[50000 rows x 2 columns]
```

1.3.5 bag-of-words model

- ML algorithms only work on numerical values
- need to encode/transform text data into numerical values using **bag-of-words** model
- **bag-of-words** technique allow us to represent text as numerical feature vectors:
  1. extract all the unique tokens – e.g., words – from entire document
  2. construct feature vector that contains the word frequency in the particular document
  3. order of the words in the document doesn't matter - hence bag-of-words
- since the unique words in each document represent only a small subset of all the words in the bag-of-words vocabulary, the feature vector will be **sparse** mostly consisting of zeros

1.3.6 transform words into feature vectors

- use `CountVectorizer` class implemented in scikit-learn
- https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
- `CountVectorizer` takes an array of text data and returns a bag-of-words vectors

[24]:
```
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)
```

[25]:
```
# let's look at the vocabulary_ contents of count object
count.vocabulary_
```

[25]:
```
{'the': 6,
 'sun': 4,
 'is': 1,
```

```
'shining': 3,
'weather': 8,
'sweet': 5,
'and': 0,
'one': 2,
'two': 7}
```

[26]:
```
bag
```

[26]:
```
<3x9 sparse matrix of type '<class 'numpy.int64'>'
        with 17 stored elements in Compressed Sparse Row format>
```

[27]:
```
bag.toarray()
```

[27]:
```
array([[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]])
```

1.3.7 bag-of-words feature vector

- the values in the feature vectors are also called the **raw term frequencies**
  - $x^i = tf(t^i, d)$
  - the number of times a term, t appears in a document, d
- indices of terms are usually assigned alphabetically

1.3.8 N-gram models

- the above model is **1-gram** or **unigram** model
  - each item or token in the vocabulary represents a single word
- if the sentence is: "The sun is shining"
  - **1-gram**: "the", "sun", "is", "shining"
  - **2-gram**: "the sun", "sun is", "is shining"
- `CountVectorizer` class allows us to use different n-gram models via its `ngram_range` parameter
- e.g. ngram_range(2, 2) will use 2-gram model

1.4 Assess word relevency via term frequency-inverse document frequency

- words often occur across multiple documents from all the classes (positive and negative in IMDb)
- frequently occuring words across classes don't contain discriminatory information
- **tf-idf** model can be used to downweight these frequently occuring words in the feature vectors

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

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

  - n_d - total number of documents

  - $\text{df}(d, t)$ - number of documents, d that contain the term t
  - *log* ensures that low document frequencies are not given too much weight

- scikit-learn implements `TfidfTransformer` class which takes the raw term frequencies from the `CountVectorizer` class as input and returns tf-idfs feature vectors

[28]:
```
from sklearn.feature_extraction.text import TfidfTransformer
```

[34]:
```
tfidf = TfidfTransformer(use_idf=True, norm='l2', smooth_idf=True)
np.set_printoptions(precision=2)
tfidf.fit_transform(bag).toarray()
```

[34]:
```
array([[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]])
```

1.4.1 Note

- 'is' (index = 1) had the largest **TF** of 3 in the third document
- after transforming, **is** now has relatively small tf-idf (0.45) in the 3^{rd} document
- `TfidfTransformer` calculates `idf` and `tf-idf` slight differently (adds 1)

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

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times (\text{idf}(t, d) + 1)$$

- by default, `TfidfTransformer` applies the L2-normalization (`norm='l2'`), which returns a vector of length 1 by dividing an un-normalized feature vector v by its L2-norm

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

- let's see an example of how `tf-idf` is calculated

[35]:
```
# unnormalized tf-idf of 'is' in document 3 can be calculated as following
tf_is = 3
n_docs = 3
idf_is = np.log((n_docs+1) / (3+1))
tfidf_is = tf_is * (idf_is + 1)
print('tf-idf of term "is" = %.2f' % tfidf_is)
```

```
tf-idf of term "is" = 3.00
```

- repeat the calculations for every term in 3^{rd} document
  - we'll get a tf-idf vectors: [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0 , 1.69, 1.29]
- let's apply L2-normalization:

$$\text{tf-idf}_{norm} = \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{[3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2]}}$$

$$= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19]$$

$$\Rightarrow \text{tfi-df}_{norm}("is", d3) = 0.45$$

[42]:
```
# calculate tfidf without normalization
tfidf = TfidfTransformer(use_idf=True, norm=None, smooth_idf=True)
raw_tfidf = tfidf.fit_transform(count.fit_transform(docs)).toarray()[-1]
raw_tfidf
```

[42]:
```
array([3.39, 3.  , 3.39, 1.29, 1.29, 1.29, 2.  , 1.69, 1.29])
```

[43]:
```
l2_tfidf = raw_tfidf / np.sqrt(np.sum(raw_tfidf**2))
l2_tfidf
# same result as TfidfTransformer with L2-regularization
```

[43]:
```
array([0.5 , 0.45, 0.5 , 0.19, 0.19, 0.19, 0.3 , 0.25, 0.19])
```

1.5 Cleaning text data

- text may have unwanted characters such as HTML/XML tags and punctuations
- convert all text into lowercase
  - we may lose characteristics of proper nouns, but they're not relevant in sentiment analysis
- remove all unwanted characters but keep emoticons such as: :) (smiley face, sad face, etc.)
  - emoticons have sentiment values
  - however, remove *nose* character (- in :-)) from the emoticons for consistency
- for simplicity, we use regular expressions though sophisticated libriaries such BeautifulSoup and Python html.parser exist for parsing html/xml documents
  - regular expressions are sufficient for this application to cleanup the unwanted characters
- let's display the last 50 chacaters from the first document in reshuffled movie review dataset

[44]:
```
df.loc[0, 'review'][-50:]
```

[44]:
```
'is seven.<br /><br />Title (Brazil): Not Available'
```

[45]:
```
import re
# create a function to do the preprocessing
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text) # remove html
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text)
    text = (re.sub('[\W]+', ' ', text.lower()) +
            ' '.join(emoticons).replace('-', '')) # convert upper to lowercase;␣
↪remove - from :-)
    return text
```

[46]:
```python
# let's preprocess the above text
preprocessor(df.loc[0, 'review'][-50:])
```

[46]: 'is seven title brazil not available'

[49]:
```python
# quick test for emoticons
preprocessor("</a>This :) is :( a test :-)! more test :-( <img />")
```

[49]: 'this is a test more test :) :(:) :('

[50]:
```python
# let's preprocess review column in DataFrame
df['review'] = df['review'].apply(preprocessor)
```

[51]:
```python
# quick test
df.loc[0, 'review'][-50:]
```

[51]: 'zation my vote is seven title brazil not available'

1.6 Processing documents into tokens

- easy way to *tokenize* documents is to split them into individual words splitting the cleaned documents at their whitespace characters

[52]:
```python
def tokenizer(text):
    return text.split()
```

[53]:
```python
tokenizer('runners like running and thus they run')
```

[53]: ['runners', 'like', 'running', 'and', 'thus', 'they', 'run']

1.6.1 Word stemming

- transforming a word into its root form
- allows to map related words typically with the same meaning to the same stem
- **Porter stemmer** is one of the oldest and simplest algorithms used to find the words' stem
- **Porter stemmer** is implemented in the **Natural Language Toolkit (NLTK)**
  - http://www.nltk.org/
- others algorithms found in NLTK are:
  - **Snowball stemmer (Porter2 or English stemmer)**
  - **Lancaster stemmer**
- must install `nltk` framework to use

[55]:
```python
! pip install nltk
```

```
Collecting nltk
  Downloading nltk-3.5.zip (1.4 MB)
     |                  | 1.4 MB 1.3 MB/s eta 0:00:01
Requirement already satisfied: click in
/Users/rbasnet/miniconda3/envs/ml/lib/python3.7/site-packages (from nltk)
```

```
(7.1.2)
Requirement already satisfied: joblib in
/Users/rbasnet/miniconda3/envs/ml/lib/python3.7/site-packages (from nltk)
(0.17.0)
Collecting regex
  Downloading regex-2020.11.13-cp37-cp37m-macosx_10_9_x86_64.whl (284 kB)
     |                 | 284 kB 9.1 MB/s eta 0:00:01
Requirement already satisfied: tqdm in
/Users/rbasnet/miniconda3/envs/ml/lib/python3.7/site-packages (from nltk)
(4.51.0)
Building wheels for collected packages: nltk
  Building wheel for nltk (setup.py) … done
  Created wheel for nltk: filename=nltk-3.5-py3-none-any.whl size=1434672
sha256=47ffb880fa3626b5ec3d08fec82109016c20e0f5449caaeaca1ebafc36e4bd02
  Stored in directory: /Users/rbasnet/Library/Caches/pip/wheels/45/6c/46/a1865e7
ba706b3817f5d1b2ff7ce8996aabdd0d03d47ba0266
Successfully built nltk
Installing collected packages: regex, nltk
Successfully installed nltk-3.5 regex-2020.11.13
```

[56]:
```
from nltk.stem.porter import PorterStemmer
```

[57]:
```
porter = PorterStemmer()
```

[58]:
```
def porter_stemmer(text):
    # use tokenizer function defined above
    return [porter.stem(word) for word in tokenizer(text)]
```

[59]:
```
porter_stemmer('runners like running and thus they run')
```

[59]:
```
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

1.6.2 Stop-words removal

- words that are extremely common in all sorts of texts and probably bear no (or only a little) useful information
- can't help in distinguishing between differenct classes of documents
  - e.g.: *is, has, and, like, are, am, etc.*
- removing stopwords can reduce the feature vector size without losing important information
- NLTK library has a set of 127 stop-words which can be downloaded using `nltk.download` function

[60]:
```
import nltk
```

[61]:
```
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/rbasnet/nltk_data…
[nltk_data]   Unzipping corpora/stopwords.zip.
```

[61]: True

[63]:
```
from nltk.corpus import stopwords
stop = stopwords.words('english')
```

[64]:
```
sentense = 'a runner likes running a lot'
[w for w in porter_stemmer(sentense) if w not in stop]
```

[64]: ['runner', 'like', 'run', 'lot']

1.7 Training a logistic regression model for document classification

- our DataFrame is alreay randomized; let's just split
- use `Pipeline` class implemented in scikit-learn - https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
- Pipeline let's us sequentially apply a list of transforms and a final estimator
- intermediate steps of the pipeline must be `transforms`,
  - that is, they must implement fit and transform methods
- the final estimator only needs to implement fit
- we'll also use `GridSearchCV` object to find the optimal set of parameters for our logistic regression model

[92]:
```
# improve our tokenizer function
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
```

[65]:
```
# split dataset into 50/50 (just following text)
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
```

[67]:
```
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],
                 'clf__penalty': ['l1', 'l2'],
                 'clf__C': [1.0, 10.0]},
                {'vect__ngram_range': [(1, 1)],
                 #'vect__stop_words': [stop, None],
                 'vect__tokenizer': [tokenizer],
                 'vect__use_idf':[False],
                 'vect__norm':[None],
                 'clf__penalty': ['l1', 'l2'],
                 'clf__C': [1.0, 10.0]},
                ]

lr_tfidf = Pipeline([('vect', tfidf),
                     ('clf', LogisticRegression(random_state=0,␣
 ↪solver='liblinear'))])

gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
                           scoring='accuracy',
                           cv=5,
                           verbose=2,
                           n_jobs=-1)
```

[69]:
```
gs_lr_tfidf.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:   28.8s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:   59.5s finished
```

[69]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('vect',
                                        TfidfVectorizer(lowercase=False)),
                                       ('clf',
                                        LogisticRegression(random_state=0,
solver='liblinear'))]),
             n_jobs=-1,
             param_grid=[{'clf__C': [1.0, 10.0], 'clf__penalty': ['l1', 'l2'],
                          'vect__ngram_range': [(1, 1)],
                          'vect__tokenizer': [<function tokenizer at
0x7fecff1764d0>]},
                         {'clf__C': [1.0, 10.0], 'clf__penalty': ['l1', 'l2'],
                          'vect__ngram_range': [(1, 1)], 'vect__norm': [None],
                          'vect__tokenizer': [<function tokenizer at
0x7fecff1764d0>],
                          'vect__use_idf': [False]}],
             scoring='accuracy', verbose=2)
```

[70]:
```python
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__tokenizer': <function tokenizer at 0x7fecff1764d0>}
CV Accuracy: 0.897
```

[71]:
```python
clf = gs_lr_tfidf.best_estimator_
print('Test Accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Test Accuracy: 0.899
```

1.8 Topic modeling with Latent Dirichlet Allocation (LDA)

- topic modeling describes the broad task of assigning topics to unlabeled text documents
- e.g., automatic categorization of documents in a large text corpus of newspaper articles into topcis:
  - sports, finance, world news, politics, local news, etc.
- topic modeling is a type of clustering task (a subcategory of unsupervised learning)
- let's use `LatentDirichletAllocation` class implemented in scikit-learn to learn different topics from the IMDb movie dataset

[72]:
```python
import pandas as pd
import pickle
```

[90]:
```python
# load the pickle dump
df = pickle.load(open('./data/movie_data.pd', 'rb'))
```

[91]:
```python
df
```

[91]:
```
                                                  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
3      hi for all the people who have seen this wonde…          1
4      I recently bought the DVD, forgetting just how…          0
…                                                    …        …
49995  OK, lets start with the best. the building. al…          0
49996  The British 'heritage film' industry is out of…          0
49997  I don't even know where to begin on this one. …          0
49998  Richard Tyler is a little boy who is scared of…          0
49999  I waited long to watch this movie. Also becaus…          1

[50000 rows x 2 columns]
```

[81]:
```python
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)
# hyperparameters: max_df = 10% - to exclude words that occur too frequently␣
 ↪across documents
# limit the max features to 5000; limit diminsionlity of the dataset
```

[82]:
```
# Note this may take a while... about 5 mins
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_components=10, # topics
                                random_state=123,
                                learning_method='batch')
# batch learning method is slower comapare to 'online' but may lead to more␣
 ↪accuracy
X_topics = lda.fit_transform(X)
```

[83]:
```
lda.components_.shape
```

[83]: (10, 5000)

[84]:
```
# let's print the the 5 most import words for each of the 10 topics
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
```

- based on reading the 5 most important words for each topic, we can guess that the LDA identified the following topics:

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

- let's confirm with the acutal contents of the reviews
- print 5 movies from the horror category (category 6 at index 5)

[86]:
```python
horror = X_topics[:, 5].argsort()[::-1]

for iter_idx, movie_idx in enumerate(horror[:5]):
    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 year before; namely that Universal's three most famous monsters; Dracula,
Frankenstein's Monster and The Wolf Man are appearing in the movie together.
Naturally, the film is rather 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 makes sense here and I even doubt
there  …

Horror movie #3:
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a total
freakfest from start to finish. A fun freakfest at that, but at times it was a
tad too reliant on kitsch rather than the horror. The story is difficult to
summarize succinctly: a carefree, normal teenage girl starts coming fac …

Horror movie #4:
Before I talk about the ending of this film I will talk about the plot. Some
dude named Gerald breaks his engagement to Kitty and runs off to Craven Castle
in Scotland. After several months Kitty and her aunt venture off to Scottland.
```

```
Arriving at Craven Castle Kitty finds that Gerald has aged and he …

Horror movie #5:
This film marked the end of the "serious" Universal Monsters era (Abbott and
Costello meet up with the monsters later in "Abbott and Costello Meet
Frankentstein"). It was a somewhat desparate, yet fun attempt to revive the
classic monsters of the Wolf Man, Frankenstein's monster, and Dracula one "la
…
```

[ ]:

```

```