8. Sentiment Analysis

COMP3314
Machine Learning

Outline

- In the age of social media, people's opinions, reviews and recommendations have become a valuable resource for political sciences and businesses
- In this chapter, we will analyze such data using a subfield of Natural Language Processing (NLP) called Sentiment Analysis
  - Analysing the polarity of documents
    - Sometimes called Opinion Mining
- We will learn how to do the following
  - Cleaning and preparing text data
  - Building feature vectors from text documents
  - Training a machine learning model to classify positive and negative reviews
  - Working with large text datasets using out-of-core learning
  - Inferring topics from document collections for categorization

Internet Movie Database (IMDb)

- We are going to work with a dataset of 50,000 movie reviews from the Internet Movie Database (IMDb) and build a predictor that can distinguish between positive and negative reviews

The IMDb Dataset

- The dataset has been prepared by Andrew Maas and was first published in the ACL conference in 2011
  - The paper is available is available here and the dataset is here (80MB)
- The dataset consists of 50,000 polar reviews that are labeled as either positive or negative
  - Positive: >5.0 stars on average
  - Negative: <=5.0 stars on average

Code - SentimentAnalysis.ipynb

- Available here on CoLab

Obtaining the IMDb Dataset

- Download the dataset with the link from the previous slide and decompress it
  - Note that in jupyter notebooks you can use ! followed by a terminal command that you want to execute

```
!wget http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar xfz aclImdb_v1.tar.gz
```

```
--2022-10-28 07:05:04--  http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
Resolving ai.stanford.edu (ai.stanford.edu)... 171.64.68.10
Connecting to ai.stanford.edu (ai.stanford.edu)|171.64.68.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 84125825 (80M) [application/x-gzip]
Saving to: ‘aclImdb_v1.tar.gz’

aclImdb_v1.tar.gz   100%[===================>]  80.23M  53.5MB/s    in 1.5s

2022-10-28 07:05:05 (53.5 MB/s) - ‘aclImdb_v1.tar.gz’ saved [84125825/84125825]
```

Preprocessing the IMDb Dataset

- We will now assemble the individual text documents from the compressed archive into a single CSV file using pandas DataFrame object
- As this may take some time, we are first going to install a new package
  - The Python Progress Indicator PyPrind

```
1 !pip install pyprind
```

```
Collecting pyprind
  Downloading https://files.pythonhosted.org/packages/1e/30/e76fb0c45da8aef49ea8d2a90d4e7a6877b45894c25f12fb96
Installing collected packages: pyprind
Successfully installed pyprind-2.11.2
```

```python
import pyprind
import pandas as pd
import os

basepath = '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: 00:01:29
```

```
1 df.shape
```

```
(50000, 2)
```

```
1 df
```

	review	sentiment
0	I went and saw this movie last night after bei...	1
1	Actor turned director Bill Paxton follows up h...	1
2	As a recreational golfer with some knowledge o...	1
3	I saw this film in a sneak preview, and it is ...	1
4	Bill Paxton has taken the true story of the 19...	1
...	...	...
49995	Towards the end of the movie, I felt it was to...	0
49996	This is the kind of movie that my enemies cont...	0
49997	I saw 'Descent' last night at the Stockholm Fi...	0
49998	Some films that you pick up for a pound turn o...	0
49999	This is one of the dumbest films, I've ever se...	0

50000 rows × 2 columns

```
import numpy as np
np.random.seed(0)
df = df.reindex(np.random.permutation(df.index))
df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

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

	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 × 2 columns

Bag-of-Words Model

- Recall that we have to convert categorical data, such as text, into a numerical form before we can pass it into an ML algorithm
- The bag-of-words model is a very simple model that allows us to represent text as numerical features
- The idea is as follows
  - Create a vocabulary of unique tokens, e.g., words, from the entire set of documents
  - Construct a feature vector from each document that contains the counts of how often each word occurs in the particular document
- Note that unique words in each document represent only a small subset of all the words in the bag-of-words vocabulary
  - Therefore, the feature vectors will mostly consist of zeros
    - I.e., we are dealing with sparse feature vectors

Transforming Words into Feature Vectors

- Let's use the CountVectorizer class to construct a bag-of-words model based on the word counts in the respective documents
- CountVectorizer takes an array of text data and constructs the bag-of-words model for us

```
1 import numpy as np
2 from sklearn.feature_extraction.text import CountVectorizer
3 count = CountVectorizer()
4 docs = np.array([
5         'The sun is shining',
6         'The weather is sweet',
7         'The sun is shining, the weather is sweet, and one and one is two'])
8 bag = count.fit_transform(docs)
9 print(count.vocabulary_)
```

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

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

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

```python
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]]
```

Term Frequencies (tf)

- These values in the feature vectors are called the raw term frequencies
  - tf(t, d)—the number of times a term, t, occurs in a document, d
- Note that in the bag-of-words model, the word or term order in a sentence or document does not matter
- The order in which the term frequencies appear in the feature vector is derived from the vocabulary indices, which are usually assigned alphabetically

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

```
1 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]]
```

N-gram Models

- The sequence of items in the bag-of-words model that we just created is also called the 1-gram or unigram model
  - Each item or token in the vocabulary represents a single word
- More generally, the contiguous sequences of items are also called n-grams
  - Items can be words or letters
- Example: The 1-gram and 2-gram representations of our first document "the sun is shining" would be constructed as follows
    - 1-gram: "the", "sun", "is", "shining"
    - 2-gram: "the sun", "sun is", "is shining"
- CountVectorizer allows us to use different n-gram models
  - While a 1-gram representation is used by default, we could switch to a 2-gram representation by initializing a new CountVectorizer instance with ngram_range=(2,2)

Word Relevancy: tf-idf

- When we are analyzing text data, we often encounter words that occur across multiple documents from both classes
  - These frequently occurring words typically don't contain useful or discriminatory information
- A useful technique called the term frequency-inverse document frequency (tf-idf) can be used to downweight these frequently occurring words
- The tf-idf is defined as the product of the term frequency (tf) and the inverse document frequency (idf)

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

- The idf(t, d) is defined as follows

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

documents

Number of documents d that contain t

Tf-idf Transformer

- The TfidfTransformer class takes the raw term frequencies from the CountVectorizer class as input and transforms them into tf-idfs

```
1 from sklearn.feature_extraction.text import TfidfTransformer
2
3 tfidf = TfidfTransformer(use_idf=True, norm='l2', smooth_idf=True)
4 np.set_printoptions(precision=4)
5 print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
```

```
[[0.     0.4337 0.     0.5585 0.5585 0.     0.4337 0.     0.    ]
 [0.     0.4337 0.     0.     0.     0.5585 0.4337 0.     0.5585]
 [0.5024 0.4451 0.5024 0.191  0.191  0.191  0.2967 0.2512 0.191 ]]
```

Tf-idf Transformer - Calculations

```
3 tfidf = TfidfTransformer(use_idf=True, norm='l2', smooth_idf=True)
```

- TfidfTransformer calculates the tf-idfs slightly differently compared to the standard textbook equations that we defined previously
- The equations for the idf and tf-idf are implemented in sk-learn as follows

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

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

Tf-idf Example Calculation

```
[[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]]
```

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

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

- Let's walk through an example and calculate the tf-idf of the word 'is' in the third document
- The word 'is' has a term frequency of 3 (tf = 3) in the third document
- The document frequency of this term is 3 since the term 'is' occurs in all three documents (df = 3)
- Thus, we can calculate the inverse document frequency as follows

$$idf(\text{"is"}, d_3) = \log\frac{1+3}{1+3} = 0$$

- We simply need to add 1 to the inverse document frequency and multiply it by the term frequency

$$tf\text{-}idf(\text{"is"}, d_3) = 3 \times (0 + 1) = 3$$

Tf-idf Example Calculation

- If we repeated this calculation for all terms in the third document, we'd obtain the following tf-idf vector
  - [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]
- The final step that we are missing in this tf-idf calculation is the L2-normalization, which can be applied as follows

$$tf\text{-}idf(d_3)_{norm} = \frac{[3.39,\ 3.0,\ 3.39,\ 1.29,\ 1.29,\ 1.29,\ 2.0,\ 1.69,\ 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}}$$

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

$$tf\text{-}idf(\text{"is"}, d_3) = 0.45$$

Cleaning Text Data

- Before we build our bag-of-words model it is important to clean the text data by stripping it of all unwanted characters
- To illustrate why this is important, let's display the last 50 characters from the first document in the reshuffled movie review dataset

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

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

- The text contains HTML as well as punctuation and other non-letter characters
- HTML does not really contain useful semantics
- Punctuation may represent useful, additional information in certain NLP contexts
- However, for simplicity, we will now remove all punctuation marks except for emoticon characters, such as :)

Regular Expressions

- We will use regular expressions (RE) to accomplish the text cleaning
  - RE offer an efficient and convenient approach to searching for characters in a string
  - They also come with a steep learning curve and we will not go into the details here
- There is a great tutorial on the Google Developers portal or you can check out the official documentation of Python's re module here

Word Capitalization

- We assume that the capitalization of a word—for example, whether it appears at the beginning of a sentence—does not contain semantically relevant information
  - In the context of this analysis, it is a simplifying assumption that the letter case does not contain information that is relevant for sentiment analysis

Cleaning Text Data

```python
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
```

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

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

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

```
'is seven title brazil not available'
```

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

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

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

Tokenization

- How to split the text into individual elements (aka token)?
  - One way to tokenize documents is to split them into individual words by splitting the cleaned documents at their whitespace characters

```
1 def tokenizer(text):
2     return text.split()
```

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

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

Porter Stemmer Algorithm

- In the context of tokenization, a useful technique is word stemming, which is the process of transforming a word into its root form. It allows us to map related words to the same stem
- The original stemming algorithm was developed by Martin F. Porter in 1979 and is hence known as the Porter stemmer algorithm
- The Natural Language Toolkit (NLTK) for Python implements the Porter stemming algorithm, which we will use in the following code section

```
1 from nltk.stem.porter import PorterStemmer
2 porter = PorterStemmer()
3 def tokenizer_porter(text):
4     return [porter.stem(word) for word in text.split()]
```

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

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

NLTK Book

- NLTK is not the focus of this chapter
  - If you are interested in NLP, we recommend that you visit the NLTK website as well as read the official NLTK book, which is freely available here

Stemming Algorithms

- The Porter stemming algorithm is probably the oldest and simplest stemming algorithm
- Other popular stemming algorithms include the newer Snowball stemmer (Porter2 or English stemmer) and the Lancaster stemmer (Paice/Husk stemmer)
- While both the Snowball and Lancaster stemmers are faster than the original Porter stemmer, the Lancaster stemmer is also notorious for being more aggressive than the Porter stemmer
- These alternative stemming algorithms are also available through this NLTK package

Stop-Word Removal

- Stop-words are those words that are extremely common in all sorts of texts and probably bear no (or only a little) useful information
  - Examples of stop-words are is, and, has, and like
- Removing stop-words can be useful if we are working with raw or normalized term frequencies rather than tf-idfs, which are already downweighting frequently occurring words

Stop-Word Removal - NTLK

- In order to remove stop-words from the movie reviews, we will use the set of 127 English stop-words that is available from the NLTK library, which can be obtained by calling the nltk.download function

```
1 import nltk
2 nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```

```
1 from nltk.corpus import stopwords
2 stop = stopwords.words('english')
3 print(stop)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', '
```

```
1 [w for w in tokenizer_porter('a runner likes running and runs a lot') if w not in stop]
```

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

Logistic Regression

- Let's train a logistic regression model to classify the movie reviews into positive and negative reviews based on the bag-of-words model
- First, we will divide the DataFrame of cleaned text documents into 25,000 documents for training and 25,000 documents for testing

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

Logistic Regression

- Next, we will use a GridSearchCV object to find the optimal set of parameters for our logistic regression model using 5-fold stratified cross-validation

```
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, solver='liblinear'))])
gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid, scoring='accuracy', cv=5, verbose=2, n_jobs=-1)
gs_lr_tfidf.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed: 27.0min
[Parallel(n_jobs=-1)]: Done 158 tasks      | elapsed: 122.2min
[Parallel(n_jobs=-1)]: Done 240 out of 240 | elapsed: 189.4min finished
```

```
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, 've
CV Accuracy: 0.897
```

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

```
Test Accuracy: 0.899
```

The Naïve Bayes Classifier

- A still very popular classifier for text classification is the naïve Bayes classifier, which gained popularity in applications of email spam filtering
- Naïve Bayes classifiers are easy to implement, computationally efficient, and tend to perform particularly well on relatively small datasets compared to other algorithms
- Although we don't discuss naïve Bayes classifiers in COMP3314, the interested reader can find an article about naïve Bayes text classification that is freely available here

Out-of-Core Learning

- Notice that it could be computationally quite expensive to construct the feature vectors for the 50,000-movie review dataset during grid search
  - The grid search fit took more than 3 hours on colab
- We will now apply a technique called out-of-core learning, which allows us to work with such large datasets by fitting the classifier incrementally on smaller batches
- We will make use of the partial_fit function of the SGDClassifier to stream the documents and train a logistic regression model using small mini-batches of documents

Out-of-Core Learning

- Let's define a generator function, stream_docs, that reads in and returns one document at a time

```
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
```

- To verify that our stream_docs function works correctly, let's read in the first document from the movie_data.csv file

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

```
('"In 1974, the teenager Martha Moxley (Maggie Grace) move
 1)
```

```python
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
```

```python
x = stream_docs(path='movie_data.csv')
print(next(x))
print(next(x))
print(next(x))
```

```
('"In 1974, the teenager Martha Moxley (Maggie Grace) moves to the high-class area of E
('"OK... so... I really like Kris Kristofferson and his usual easy going delivery of li
('"***SPOILER*** Do not read this, if you think about watching that movie, although it
```

Out-of-Core Learning

- We will now define a function, get_minibatch, that will take a document stream from the stream_docs function and return a particular number of documents specified by the size parameter

```
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
```

Out-of-Core Learning

- Unfortunately, we can't use CountVectorizer for out-of-core learning since it requires holding the complete vocabulary in memory
- Also, TfidfVectorizer needs to keep all the feature vectors of the training dataset in memory to calculate the inverse document frequencies
- However, another useful vectorizer for text processing implemented in scikit-learn is HashingVectorizer

```
1 from sklearn.feature_extraction.text import HashingVectorizer
2 vect = HashingVectorizer(decode_error='ignore', n_features=2**21,
3                          preprocessor=None, tokenizer=tokenizer)
```

```
1 from sklearn.linear_model import SGDClassifier
2 clf = SGDClassifier(loss='log', random_state=1)
3 doc_stream = stream_docs(path='movie_data.csv')
```

Out-of-Core Learning

- Having set up all the complementary functions, we can start the out-of-core learning using the following code

```
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: 00:00:07
```

```
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: 0.807
```

The word2vec Model

- A more modern alternative to the bag-of-words model is word2vec, an algorithm that Google released in 2013
- The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words
- The idea behind word2vec is to put words that have similar meanings into similar clusters, and via clever vector-spacing, the model can reproduce certain words using simple vector math, for example, king – man + woman = queen
- The original C-implementation with useful links to the relevant papers and alternative implementations can be found here

Topic Modeling

- Topic modeling describes the broad task of assigning topics to unlabeled text documents
  - For example, a typical application would be the categorization of documents in a large text corpus of newspaper articles
  - In applications of topic modeling, we then aim to assign category labels to those articles, for example, sports, finance, world news, politics, local news, and so forth
- We can consider topic modeling as a clustering task, a subcategory of unsupervised learning
- In this section, we will discuss a popular technique for topic modeling called Latent Dirichlet Allocation (LDA)

LDA

- Since the mathematics behind LDA is quite involved and requires knowledge about Bayesian inference, we will approach this topic from a practitioner's perspective and interpret LDA using layman's terms
- However, the interested reader can read more about LDA in this research paper

LDA

- LDA is a generative probabilistic model that tries to find groups of words that appear frequently together across different documents
- These frequently appearing words represent our topics, assuming that each document is a mixture of different words
- The input to an LDA is the bag-of-words model that we discussed earlier in this chapter
- Given a bag-of-words matrix as input, LDA decomposes it into two new matrices:
  - A document-to-topic matrix
  - A word-to-topic matrix

LDA

- LDA decomposes the bag-of-words matrix in such a way that if we multiply those two matrices together, we will be able to reproduce the input, the bag-of-words matrix, with the lowest possible error. In practice, we are interested in those topics that LDA found in the bag-of-words matrix
- The only downside may be that we must define the number of topics beforehand
  - The number of topics is a hyperparameter of LDA that has to be specified manually

LDA with scikit-learn

- Let's use the LatentDirichletAllocation class implemented in scikit-learn to decompose the movie review dataset and categorize it into different topics
- In the following example, we will restrict the analysis to 10 different topics
  - Your are encouraged to experiment with the hyperparameters of the algorithm to further explore the topics that can be found in this dataset

```
1 import pandas as pd
2 df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

	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

```
1 from sklearn.feature_extraction.text import CountVectorizer
2 count = CountVectorizer(stop_words='english', max_df=.1, max_features=5000)
3 X = count.fit_transform(df['review'].values)
```

LDA with scikit-learn

- The following code example demonstrates how to fit a LatentDirichletAllocation estimator to the bag-of-words matrix and infer the 10 different topics from the documents

```
1 from sklearn.decomposition import LatentDirichletAllocation
2 lda = LatentDirichletAllocation(n_components=10, random_state=123, learning_method='batch')
3 X_topics = lda.fit_transform(X)
```

- After fitting the LDA, we now have access to the components_ attribute of the lda instance, which stores a matrix containing the word importance for each of the 10 topics in increasing order

```
1 lda.components_.shape
```

```
(10, 5000)
```

LDA with scikit-learn

- To analyze the results, let's print the five most important words for each of the 10 topics
- Note that the word importance values are ranked in increasing order
- Thus, to print the top five words, we need to sort the topic array in reverse order

```
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
```

LDA with scikit-learn

- Based on reading the five most important words for each topic, you may 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. Comedy movies reviews
  - 8. Movies somehow related to TV shows
  - 9. Movies based on books
  - 10. Action movies

LDA with scikit-learn

- To confirm that the categories make sense based on the reviews, let's plot three movies from the horror movie category (horror movies belong to category 6 at index position 5)

```
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 year before; namely t

Horror movie #2:
Okay, what the hell kind of TRASH have I been watching now? "The Witches' Mountain" has got to be one of t

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

Conclusion

- In this chapter, we learned how to use machine learning algorithms to classify text documents based on their polarity
  - We learned how to encode a document as a feature vector using the bag-of-words model
  - We also learned how to weight the term frequency by relevance using tf-idf
  - Working with text data can be computationally quite expensive due to the large feature vectors that are created during this process; in the last section, we covered how to utilize out-of-core or incremental learning to train a machine learning algorithm without loading the whole dataset into a computer's memory
  - Lastly, we were introduced to the concept of topic modeling using LDA to categorize the movie reviews into different categories in an unsupervised fashion

References

- Most materials in this chapter are based on
  - Book
  - Code