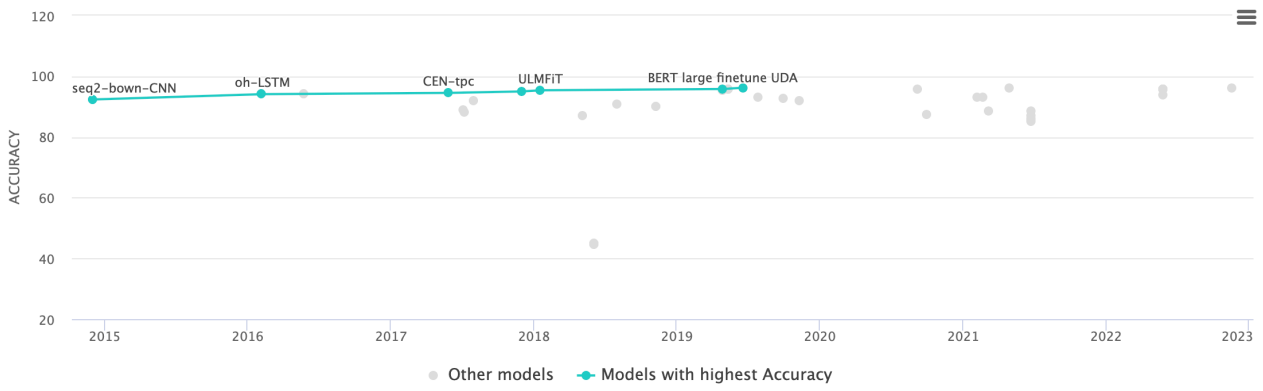
Applied Artificial Intelligence

Lab 7 — Natural Language Processing

Getting started— We will work with the **IMDB movie dataset** in this lab. You can download it from Kaggle here, or from Canvas. The data will unzip into a .csv file, containing two columns — “review” and “sentiment”.

IMDB is an interesting dataset - it’s tricky as it contains long text segments, but researchers have made good progress on it over the years, see Figure 1, from here. We’ll see how far we get this week...



Our overall goal this week is to learn how to pre-process text so that we can use it with machine learning and data analysis tools and libraries as we have done with numerical data in the past. Often these pre-processing steps will also help us shed light on the basic properties of our texts, e.g. frequent words, topics, *n*-grams, etc. and can therefore guide our subsequent analysis steps. Once all pre-processing is complete, we will build a simple Naive Bayes sentiment classifier using `sk_learn` to make automatic predictions on whether a review (text) expresses a positive or a negative sentiment. Finally, we will try to see if there are any common themes that we can identify in positive vs negative reviews.

Here is an example of a **positive** review. Note that this is a slightly “interesting” case as many of the actual words in the text are rather negative.

*“One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. They are right, as this is exactly what happened with me.

The first thing that struck me about Oz was its brutality and unflinching scenes of violence, which set in right from the word GO.*

*Trust me, this is not a show for the faint hearted or timid. This show pulls no punches with regards to drugs, sex or violence. Its is hardcore, in the classic use of the word.

It is called OZ as that is the nickname given to the Oswald Maximum Security State Penitentary. It focuses mainly on Emerald City, an experimental section of the prison where all the cells have glass fronts and face inwards, so privacy is not high on the agenda. Em City is home to many..Aryans, Muslims, gangstas, Latinos, Christians, Italians, Irish and more....so scuffles, death stares, dodgy dealings and shady agreements are never far away.

I would say the main appeal of the show is due to the fact that it goes where other shows wouldn't dare. Forget pretty pictures painted for mainstream audiences, forget charm, forget romance...OZ doesn't mess around. The first episode I ever saw struck me as so nasty it was surreal, I couldn't say I was ready for it, but as I watched more, I developed a taste for Oz, and got accustomed to the high levels of graphic violence. Not just violence, but injustice (crooked guards who'll be sold out for a nickel, inmates who'll kill on order and get away with it, well mannered, middle class inmates being turned into prison bitches due to their lack of street skills or prison experience) Watching Oz, you may become comfortable with what is uncomfortable viewing....thats if you can get in touch with your darker side."*

For comparison, here is an example of a **negative** review:

*"Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the time.

This movie is slower than a soap opera... and suddenly, Jake decides to become Rambo and kill the zombie.

OK, first of all when you're going to make a film you must Decide if its a thriller or a drama! As a drama the movie is watchable. Parents are divorcing & arguing like in real life. And then we have Jake with his closet which totally ruins all the film! I expected to see a BOOGEYMAN similar movie, and instead i watched a drama with some meaningless thriller spots.

3 out of 10 just for the well playing parents & descent dialogs. As for the shots with Jake: just ignore them"*

Overall we have 50,000 reviews in this dataset.

To pre-process your dataset, you will need to:

- Read it into your program (probably using pandas is a good idea).
- Remove any stop words and special characters.
- Make sure all words are in lower case.
- Lemmatise the data (use Spacey e.g.). Note: *lemmatisation* is the process of representing inflected words as a single "source" word: e.g. things like *walking*, *walked*, *walks*, would all be represented as *walk*. Similarly, plural women would get represented as woman, etc.

- Get a basic frequency distribution of any words used in the text — this is an optional step and can be useful for data exploration and give you ideas on what else to explore.
- Compile a vocabulary.
- Represent the data an integer-based format, e.g. one-hot encoding.

These steps don't necessarily have to be carried out in this order. Finally, split the data into a train and a test set, e.g. using a 70%-30% split or a 80%-20% split.

I will talk through the pre-processing steps (and code) in the remainder of this work sheet — try to do as much as you can by yourself, but return to the code examples whenever you need to.

I am using the following libraries today and have tested my code on Google Colab.

```
import pandas as pd
import os
import sys

# Load the library with the CountVectorizer method
from sklearn.feature_extraction.text import CountVectorizer
import numpy as np

#import matplotlib.pyplot as plt and seaborn for visualisation
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('whitegrid')
```

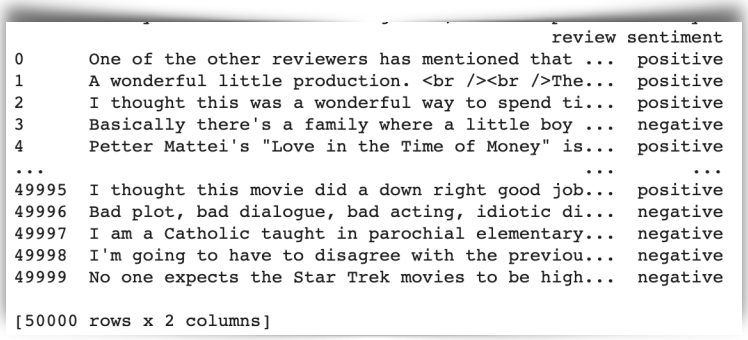
In addition I am using a few specialised libraries that deal with text processing.

```
# regular expressions for removing special characters, such as punctuation
import re

# gensim is a library for text processing including n-gram models
import gensim
from gensim.utils import simple_preprocess

# spacy is also for text processing, here we are using the lemmatiser
import spacy

# This package draws word clouds, as a form of frequency analysis
!pip install wordcloud
```

The `!pip` command is used to install a package that is not already present in Colab. Here I am installed a package for producing word clouds based on the frequency of terms in my text.

Next, mount Google Drive and (if using Colab) and read in the data using pandas.

I am getting the following output from this - 2 columns, one with review text and the other with sentiment, either positive or negative. My table has 50,000 rows.



	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive
...	...	...
49995	I thought this movie did a down right good job...	positive
49996	Bad plot, bad dialogue, bad acting, idiotic di...	negative
49997	I am a Catholic taught in parochial elementary...	negative
49998	I'm going to have to disagree with the previou...	negative
49999	No one expects the Star Trek movies to be high...	negative

[50000 rows x 2 columns]

Removing special characters and converting to lower case — With my data read in, I can see several issues that may impact classification and text analysis. This includes upper case and lower case letters (they may get mistaken for separate tokens and therefore make my language model even more sparse), special characters such as the HTML tags
 and </br> and various other punctuation marks that are likely not going to contribute to sentiment classification.

The following code will help me address this. We use the `re` package imported earlier to use regular expressions to get rid of a set of punctuation marks and to convert all text to lower case. We then print the head (first bit) of our table to see the result.

```
# Clean data ...

# Remove punctuation
imdb_df['review'] = imdb_df['review'].apply(str).map(lambda x: re.sub('[,\.!?]',
'', x))

# Convert the titles to lowercase
imdb_df['review'] = imdb_df['review'].apply(str).map(lambda x: x.lower())

# Print out the first rows of reviews
print(imdb_df['review'].apply(str).head())
```

The `map` and `lambda` function in the code is used to apply the same operation to each element in a sequential input, here our `imdb_df['review']` texts, where each individual element (review) is referred to as `x` within the bounds of the function. See here for other examples.

Try to change the regular expressions, e.g. add additional special characters or remove some, to see differently formatted text.

From this code I get the following output.

```
[50000 rows x 2 columns]
0    one of the other reviewers has mentioned that ...
1    a wonderful little production <br /><br />the ...
2    i thought this was a wonderful way to spend ti...
3    basically there's a family where a little boy ...
4    petter mattei's "love in the time of money" is...
```

If you compare this with the previous output, you will see the text changes applied.

Tokenisation— As a next step, I want to convert my text from lists of sentences to lists of lists of words. In other words, I want to represent my inputs not as complete sentences but as collections of words. This will help analysis as I can then start counting words, identifying classes of words, lemmatising them (i.e. representing them in their base rather than inflected form — dog instead of dogs, good instead of better, walk instead of walking etc.). Lemmatisation has a similar advantage as converting to lower case - it reduces the overall amounts of tokens and therefore reduces the sparsity of my dataset.

This code will initially perform the tokenisation of my data. NB if you put `deacc=True` it will remove punctuation too - but we've already done this, so I left it as False.

```
def sent_to_words(sentences):
    for sentence in sentences:
        yield(gensim.utils.simple_preprocess(str(sentence), deacc=False))  #
deacc=True removes punctuations

#data = imdb_df['review'].tolist()
data = imdb_df['review'][0:500].tolist()

data_words = list(sent_to_words(data))

print('data_words[:1]', data_words[:1])
```

Running this gives me the following output (just the first bit is shown, it's a long list).

```
data_words[:1] [['one', 'of', 'the', 'other', 'reviewers', 'has', 'mentioned', 'that', 'after', 'watching',
```

Note that this bit of the code (and the one we are going to write next) can take while to come through - so if you just want to test, you could clip the dataset here (see the commented bit above). This will use only 500 data rows from here onwards. It's not ideal for the full analysis, but sometimes good to just test and debug code.

Unigrams, bigrams and trigrams — Next we can train some simple n-gram-based language models. Strictly speaking the tokenisation process has already given us a set of unigrams. The gensim library we imported earlier can help us extract bigrams and trigrams from our text.

```
# Build the bigram and trigram models
bigram = gensim.models.Phrases(data_words, min_count=5, threshold=100) # higher
threshold fewer phrases.
print('bigram',bigram)
print('bigram data',bigram[data_words])
trigram = gensim.models.Phrases(bigram[data_words], threshold=100)
print('trigram',trigram)

# Faster way to get a sentence clubbed as a trigram/bigram
bigram_mod = gensim.models.phrases.Phraser(bigram)
trigram_mod = gensim.models.phrases.Phraser(trigram)

bigram_mod.save("bigram_mod")
trigram_mod.save("trigram_mod")

bigram_mod = gensim.models.Phrases.load("bigram_mod")
trigram_mod = gensim.models.Phrases.load("trigram_mod")
```

This bit of code trains a first set of bigrams from our tokenised (and otherwise pre-processed) data. The `min_count` threshold sets a lower threshold for how many instances of a bigram phrase we need to find before including it into our model. Increasing this parameter will lead to fewer bigrams. Decreasing it will find more, but also increase the sparsity in our dataset. The threshold parameter is dependent on the scoring function, see here for details. In simple terms, a higher threshold means fewer phrases will get found - presumably in such as way that they are more "meaningful" to the text / task at hand.

The initial set of bigrams is then used to find trigrams from the dataset, using similar parameters. The next few lines are part of the process of building the n-gram models, save them to file, and re-load them again. The saving and loading done here is optional - and can waste some time in the way it is done here. Having said that, the n-gram models take a while to build, so once you have them, it may be worthwhile saving them for latter if you want to return to the same task and not wait through them being built each time again. In this case, just get rid of the building bit for next time.

Printing off the bigrams and trigrams using this code,

```
for bigram in bigram_mod.phrasegrams.keys():
    print(bigram)

for trigram in trigram_mod.phrasegrams.keys():
    print(trigram)
```

I get the following output **for bigrams** (just the first bits copied over). As you can see it contains a number of interesting phrases, including “serial killer” which seems to make sense for a movies data, place names such as *New York* and *Hong Kong*, frequently co-occurring words such as “reminds me”, “fast forward” and other things. Looking at this list, it seems to make sense to represent these items together as “meaning units”.

```
(b'serial', b'killer')
(b'new', b'york')
(b've', b'seen')
(b'at', b'least')
(b'low', b'budget')
(b'year', b'old')
(b'reminds', b'me')
(b'ever', b'seen')
(b'high', b'school')
(b'special', b'effects')
(b'character', b'development')
(b'subject', b'matter')
(b'civil', b'war')
(b'cold', b'mountain')
(b'halfway', b'through')
(b'second', b'half')
(b'fast', b'forward')
(b'hong', b'kong')
```

For trigrams I get the following - which looks like a mixture of bigrams and trigrams — i.e. bigrams identified earlier with another word added to them, if it occurred frequently enough. This again makes sense seen that we initialised the trigram model above from the existing bigrams.

```
(b'serial', b'killer')
(b'looking', b'forward')
(b've', b'seen')
(b'at', b'least')
(b'low', b'budget')
(b'year', b'old')
(b'reminds', b'me')
(b'reminds', b'me_of')
(b'ever', b'seen')
(b've', b'ever_seen')
(b've_ever', b'seen')
(b'high', b'school')
(b'special', b'effects')
(b'character', b'development')
(b'subject', b'matter')
(b'civil', b'war')
(b'cold', b'mountain')
(b'second', b'half')
(b'the_second', b'half')
(b'second', b'half_of')
```

In the result we get a few interesting things like "'reminds me of", "the second half" or "second half of". Overall these seem less interesting perhaps than the bigrams, though can be powerful when measuring grammaticality of automatically generated language.

With our n-grams in hand, let's move on to lemmatise and remove stop words.

Lemmatisation and stop words — In this section I will remove stop words from my text using the following piece of code. You will see that gensim again has functionality for this.

```
from gensim.parsing.preprocessing import STOPWORDS
my_stop_words = STOPWORDS.union(set(['br']))

# Define functions for stopwords, bigrams, trigrams and lemmatisation
def remove_stopwords(texts):
    return [[word for word in simple_preprocess(str(doc)) if word not in
my_stop_words] for doc in texts]
```

One interesting thing one can do here is to modify the existing list of stop words with additional tokens that may be relevant to the domain. In this case, I am adding "br" and "film". I didn't test here if these are a good idea to remove but my intuition is that "film" (and probably "movie") will occur frequently without adding much distinguishing content for a sentiment analyser on film reviews. The other bit "br" is almost certainly irrelevant as it just denotes the HTML new line tag.

I then run the following - removing all tokens from the my updated stop words list from my data.

```
# Remove Stop Words
data_words_nostops = remove_stopwords(data_words)

# Form Bigrams
data_words_bigrams = make_bigrams(data_words_nostops)
```

You will notice that I am also obtaining a new set of bigrams. This relies on the following function, which uses my earlier bigram model and applies it to the new data with stop words removed.

```
def make_bigrams(texts):
    return [bigram_mod[doc] for doc in texts]

def make_trigrams(texts):
    return [trigram_mod[bigram_mod[doc]] for doc in texts]
```

Note that I could have trained trigrams here but decided to go for bigrams based on my earlier inspection of the two models.

Next, I use the Spacy library to train a lemmatiser from my bigram data. As discussed earlier, a lemmatiser converts words to their base form. I am allowing only *nouns*, *adjectives*, *verbs* and *adverbs* to be considered here as they tend to be the main meaning-bearing words in English. It's unlikely that a preposition or conjunction will help be predict the sentiment of a review.

```
def lemmatization(texts, allowed_postags=['NOUN', 'ADJ', 'VERB', 'ADV']):
    """https://spacy.io/api/annotation"""
    texts_out = []
    for sent in texts:
        doc = nlp(" ".join(sent))
        texts_out.append([token.lemma_ for token in doc if token.pos_ in
allowed_postags])
    return texts_out
```

I can apply this function as follows:

```
# Initialize spacy 'en' model, keeping only tagger component (for efficiency)
nlp = spacy.load("en_core_web_sm", disable=['parser', 'ner'])

# Do lemmatization keeping only noun, adj, vb, adv
data_lemmatized = lemmatization(data_words_bigrams, allowed_postags=['NOUN',
'ADJ', 'VERB', 'ADV'])
print('data_lemmatized[:1]', data_lemmatized[:1])
```

And will the the following output when printed to the console (just the first bit of a long list):

```
data_lemmatized[:1] [['reviewer', 'mention', 'watch', 'will', 'hook', 'right', 'exactly', 'happen', 'thing',
```

You should see the difference clearly between the earlier inflected word forms and the lemmatised output now.

This kind of representation should make it easier to apply machine learning techniques as they pre-processed our language data in such as way to make abstractions as possible as can be.

Frequency and Visualisation — As final step in the guided bit of this tutorial, I will generate a word cloud from our lemmatised data to visualise the most frequent terms. Word clouds get initialised in different ways when you run them, so I'll generate two for comparison.

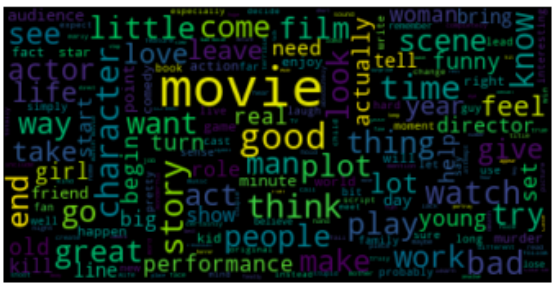
I'll use this code which I got from the [WordCloud library GitHub page](#).

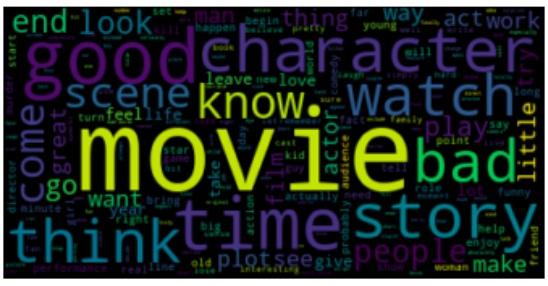
```
def getWordCloud(hashtags):

  # Read the whole text.
  text = ' '.join(hashtags)

  # Generate a word cloud image
  wordcloud = WordCloud().generate(text)

  # Display the generated image:
  # the matplotlib way:
  plt.imshow(wordcloud, interpolation='bilinear')
  plt.axis("off")

  # lower max_font_size
  wordcloud = WordCloud(max_font_size=40).generate(text)
  plt.figure()
  plt.imshow(wordcloud, interpolation="bilinear")
  plt.axis("off")
  plt.savefig('wordcloud_all.pdf', dpi=500)
  plt.show()
```

It expects strings as input, so I'll need to briefly loop over my data again to convert it to a format it will accept. Afterwards, I'll get two word clouds as printed below.

```
s = []
for l in data_lemmatized:
  t = ' '.join(l)
  s.append(t)

getWordCloud(s)
```

Looking at these, I want to go back and add "movie" to my stop list I think, but at the same time it is a good indicator that it picks up some of the contents in the data.





Part 2 - Sentiment Analysis.

This part of the practical work will be less guided. Using the new text, `data_lemmatized` and the sentiment labels in the original data_frame, train a sentiment classifier using e.g. `sk_learn's` MultiNominal Naive Bayes classifier:

https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

To evaluate your classifier, you may need to use sk_learn's train_test_split function to split your dataset into a train and a test set. Doing this with 0.33 allocated for testing, I get an accuracy 0.60 with my smaller dataset of 500 reviews. Using the full 50,000 reviews, I get 0.86. This would be a good baseline for other classifiers to build on top of.

Part 3 - Feature analysis.

Data pre-processing and explorative analysis can also help us learn something about our data and domain. Try to associate specific features, themes, words, with positive and negative reviews. A simple way to do this is to use Pandas to split the `imdb_df` dataset into 2, one of the positive and one for negative review.

Running the rest of the code over the two datasets separately, can you identify any specific words, n-grams or other that may be indicative features of positive or negative sentiment?

That's it for today - next week we'll learn about more recent & complex models to model language!