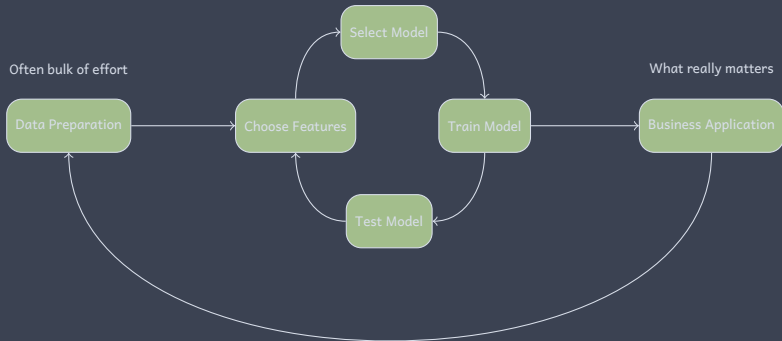


## » Feature Engineering For Text



- \* Features: need to map actual input into a vector of input features  $x$  i.e. a vector of real-values
- \* A good choice of features can make a huge difference to a machine learning system, often the difference between success and failure
- \* We'll focus first on choosing features for text input e.g. reviews, news articles. English language only. Already briefly looked at bag of words model.

## » Text Pre-processing

Text is a sequence of characters, words, sentences, paragraphs

→ Focus on text as sequence of words here.

- \* Finding word boundaries is pretty easy in English – generally there are spaces or punctuation between words, just look at this slide, but still some issues.
  - \* Not always so easy e.g. German has compound words, Chinese has no spaces.
  - \* Also in English have run-together words like *hasn't* and *we're*.
- \* First step is *tokenization*: split input character sequence into tokens.
  - \* E.g. *Here's example text, isn't it?*
  - \* Splitting on whitespace gives *Here's, example, text,, isn't, it?*
  - \* Should we split *Here's* into *Here* and *'s*? Should we treat *it?* and *it* as different tokens, or merge? Should we split *isn't* into *is* and *n't*?
    - \* Probably want to split *isn't* into *is* and *n't* if doing sentiment analysis as *n't* (“not”) is important e.g to distinguish *is good* and *isn't good*. But probably ok to truncate *Here's* to *Here* since *'s* (“is”) is a common, uninformative word
  - \* Not enough to just split on punctuation, need to use some language knowledge

## » Text Pre-processing

Typically end up with different tokens for different forms of the same word e.g. *liked*, *liking*, *likes*. Can choose to merge these together into a single token e.g. *like*. But again not so easy ...

- \* *Stemming*: merge by simply chopping off endings from tokens e.g. *liked*, *likes* → *like*. But then what about *liking*?
- \* *Lemmatisation* A bit smarter, use knowledge of language to merge properly.
- \* Need to be careful re capital letters e.g. *Us* and *US* could be pronoun and country. But if text is all caps then maybe *US=us* so need heuristics e.g. at start of sentence replace *Us* by *us*.
- \* Acronyms e.g. *eta*, *e.t.a.*, *E.T.A.* → *E.T.A.*
- \* Python
  - \* *sklearn*: tokenising splits on whitespace and chops *'s*, *'t* etc, no stemming. E.g. *likes liking liked* becomes *likes*, *liking*, *liked*
  - \* *ntlk* has more choices of tokeniser and stemmer.
    - \* E.g. applying `word_tokenizer` followed by `PorterStemmer` to *likes liking liked* gives *like*, *like*, *like*
    - \* E.g. *Here's example text, isn't it?* gives *here*, *'s*, *exampl*, *' text*, *,*, *is*, *n't*, *it*, *?*

## » Python Code

```
import nltk
nltk.download('punkt')

from sklearn.feature_extraction.text import CountVectorizer
tokenizer = CountVectorizer().build_tokenizer()
print(tokenizer("Here's example text, isn't it?"))
from nltk.tokenize import WhitespaceTokenizer
from nltk.tokenize import word_tokenize
print(WhitespaceTokenizer().tokenize("Here's example text, isn't it?"))
print(word_tokenize("Here's example text, isn't it?"))

print(tokenizer("likes liking liked"))
print(WhitespaceTokenizer().tokenize("likes liking liked"))
print(word_tokenize("likes liking liked"))

from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
tokens = word_tokenize("Here's example text, isn't it?")
stems = [stemmer.stem(token) for token in tokens]
print(stems)

tokens = word_tokenize("likes liking liked")
stems = [stemmer.stem(token) for token in tokens]
print(stems)
```

## » One-hot Encoding

We've now converted our text to tokens. How can we convert a single token to a feature vector?

- \* Use *one-hot encoding*
- \* Length of feature vector = #distinct tokens (so can be pretty large e.g. 10-100K)
- \* Map token  $i$  to a feature vector  $x$  with entry  $i$  equal to 1 and all other entries 0

E.g. *movie* might map to feature vector  $[0, 1, 0, 0, 0, 0]$

| <i>good</i> | <i>movie</i> | <i>not</i> | <i>a</i> | <i>did</i> | <i>like</i> |
|-------------|--------------|------------|----------|------------|-------------|
| 0           | 1            | 0          | 0        | 0          | 0           |

- \* Why not just map tokens to integers? E.g. map  $i$ 'th token to integer  $i$ .
  - \* Distances between feature vectors matter!
  - \* With one-hot encoding all tokens are the *same* distance from one another e.g. Euclidean distance between any two tokens is  $\sqrt{2}$
  - \* .. but if used integers then distance between tokens  $i$  and  $j$  would be  $\sqrt{(i-j)^2} = |i-j|$ , so some tokens would be much closer together than others → liable to mess up linear models unless distances reflect actual relationships
- \* Note can use one-hot encoding with any discrete quantities, not just tokens/words

## » One-hot Encoding → Bag of Words

How can we map a *sequence* of tokens (sentence, paragraph, document) to a feature vector.

- \* Use *one-hot encoding* for each word in text, then add ...
- \* E.g. *a good good movie*

| <i>good</i> | <i>movie</i> | <i>not</i> | <i>a</i> | <i>did</i> | <i>like</i> |
|-------------|--------------|------------|----------|------------|-------------|
| 0           | 0            | 0          | 1        | 0          | 0           |
| 1           | 0            | 0          | 0        | 0          | 0           |
| 1           | 0            | 0          | 0        | 0          | 0           |
| 0           | 1            | 0          | 0        | 0          | 0           |

add columns to get:

| <i>good</i> | <i>movie</i> | <i>not</i> | <i>a</i> | <i>did</i> | <i>like</i> |
|-------------|--------------|------------|----------|------------|-------------|
| 2           | 1            | 0          | 1        | 0          | 0           |

## » Bag of Words Model

How can we map a *sequence* of tokens (sentence, paragraph, document) to a feature vector.

- \* Count number of occurrences of each token in our text → later we'll then try to identify marker words like *excellent* or *disappointed*
- \* Length of feature vector = #distinct tokens
- \* Entry  $i$  of vector is set equal to number of times token  $i$  appears in text. E.g.

|                          | <i>good</i> | <i>movie</i> | <i>not</i> | <i>a</i> | <i>did</i> | <i>like</i> |
|--------------------------|-------------|--------------|------------|----------|------------|-------------|
| <i>a good good movie</i> | 2           | 1            | 0          | 1        | 0          | 0           |
| <i>not a good movie</i>  | 1           | 1            | 1          | 1        | 0          | 0           |
| <i>did not like</i>      | 0           | 0            | 1          | 0        | 1          | 1           |

- \* Note: we've thrown away information about the word order (and so any grammar) → that's why its called a “bag of words”.

## » Preserving Some Word Ordering

We can count pairs of tokens, triples etc

- \* *n*-gram: 1-gram = single token, 2-gram = pairs of tokens etc

|                          | <i>good movie</i> | <i>movie</i> | <i>did not</i> | <i>a</i> | ... |
|--------------------------|-------------------|--------------|----------------|----------|-----|
| <i>a good good movie</i> | 1                 | 1            | 0              | 1        | ... |
| <i>not a good movie</i>  | 1                 | 1            | 0              | 1        | ... |
| <i>did not like</i>      | 0                 | 0            | 1              | 0        | ... |

- \* But can quickly end up with a huge feature vector as number of *n*-grams grows quickly e.g. if have  $q$  tokens then there may be  $q^2$  pairs of tokens.



## » Python Bag of Words Example

```
docs = [  
    'This is the first document.',  
    'This is the second second document.',  
    'And the third one.',  
    'Is this the first document?']  
from sklearn.feature_extraction.text import CountVectorizer  
vectorizer = CountVectorizer()  
X = vectorizer.fit_transform(docs)  
print(vectorizer.get_feature_names())  
print(X.toarray())
```

### Output is:

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']  
[[0 1 1 1 0 0 1 0 1]  
 [0 1 0 1 0 2 1 0 1]  
 [1 0 0 0 1 0 1 1 0]  
 [0 1 1 1 0 0 1 0 1]]
```

```
vectorizer = CountVectorizer(ngram_range=(2, 2))  
X = vectorizer.fit_transform(docs)  
print(vectorizer.get_feature_names())  
print(X.toarray())
```

### Output is:

```
['and the', 'first document', 'is the', 'is this', 'second document', 'second second', 'the first', 'the second', 'the third', 'third one', 'this is', 'this the']  
[[0 1 1 0 0 0 1 0 0 0 1 0]  
 [0 0 1 0 1 1 0 1 0 0 1 0]  
 [1 0 0 0 0 0 0 0 1 1 0 0]  
 [0 1 0 1 0 0 1 0 0 0 0 1]]
```

## » Pruning: Remove Stop Words & Rare Words

Not all tokens or n-grams are interesting, so we can try to remove the boring ones so as to reduce the size of the feature vector.

### \* High-frequency tokens/n-grams

- \* Words such as *the*, *a*, *is* are v common and don't help much with discriminating between text. Referred to as *stop words* → remove

- \* *sklearn*:

```
import nltk
nltk.download('stopwords')
vectorizer = CountVectorizer(stop_words=nltk.corpus.stopwords.words('english'))
```

- \* *sklearn* vectorizer also has *df\_max* parameter to exclude words that occur in too many documents. Use cross-validation to choose a good value.

### \* Very low-frequency tokens/n-grams

- \* Typos, tokens/n-grams that appear in only one document
- \* Can encourage over-fitting e.g. classifier may learn to use a typo to identify a document → remove
- \* *sklearn* vectorizer has *df\_min* parameter to exclude words that occur in too few documents. Use cross-validation to choose a good value.

### \* How to distinguish between remaining medium-frequency tokens/n-grams?

## » Word Importance: TF-IDF

Suppose have a set  $D$  of documents e.g. a set of movie reviews.

- \* *Term frequency (TF)*  $tf(t, d)$  of token  $t$  in document  $d$  can be:
  - \* Raw count: number of times token  $t$  appears in a document  $d \rightarrow$  usual choice
  - \* Boolean: 1 if  $t$  in document  $d$ , else 0
  - \* Normalised  $\frac{\text{Number of times token } t \text{ appears in document } d}{\sum_{t' \in T} \text{Number of times token } t' \text{ appears in document } d}$ ,  $T$  the set of all tokens
  - \*  $\log(1 + \text{Number of times token } t \text{ appears in document } d)$
- \* E.g.  $tf(t, d)$  of token *great* in *a great movie and a great day out* is 2 when using raw count while  $tf(t, d)$  of token *terrible* is 0
- \* *Term frequency  $tf(t, d)$  is larger when token  $t$  occurs a lot in document  $d$*

## » Word Importance: TF-IDF

- \* *Document frequency (DF)*  $df(t)$  of token  $t$  is number of documents in  $D$  that contain token  $t$
- \* *Inverse document frequency (IDF)*  $idf(t)$  of token  $t$  in set of documents  $D$  is:
  - \*  $idf(t) = \log \frac{\text{Number of documents in } D}{1 + df(t)}$  → add 1 to denominator to prevent divide-by-zero when  $df(t) = 0$ 
    - \* sklearn a little different:  $idf(t) = 1 + \log \frac{1 + \text{Number of documents in } D}{1 + df(t)}$
  - \*  *$idf(t)$  is large when  $df(t)$  is small i.e. when token  $t$  occurs rarely in overall collection of documents  $D$*
- \* *TF-IDF*  $tfidf(t, d)$  of token  $t$  in document  $d$  is:
  - \*  $tfidf(t, d) = tf(t, d) \times idf(t)$
  - \*  $tfidf(t, d)$  increases with  $tf(t, d)$  i.e. when token  $t$  occurs more often in document  $d$
  - \*  $tfidf(t, d)$  increases with  $idf(t)$  i.e. when token  $t$  occurs more rarely in overall collection of documents  $D$
  - \*  *$tfidf(t, d)$  is large for a token that occurs a lot in document  $d$  but only rarely in overall collection of documents*  
→ a reasonable heuristic for identifying informative tokens, and widely used

## » Python TF-IDF Example

```
docs = [
    'This is the first document.',
    'This is the second second document.',
    'And the third one.',
    'Is this the first document?']
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)
print(vectorizer.get_feature_names())
print(X.toarray())

['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
[[0 1 1 1 0 0 1 0 1]
 [0 1 0 1 0 2 1 0 1]
 [1 0 0 0 1 0 1 1 0]
 [0 1 1 1 0 0 1 0 1]]

from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(norm=None)
X = vectorizer.fit_transform(docs)
print(vectorizer.get_feature_names())
print(X.toarray())

['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
[[0. 1.22 1.51 1.22 0. 0. 1. 0. 1.22]
 [0. 1.22 0. 1.22 0. 3.83 1. 0. 1.22]
 [1.92 0. 0. 0. 1.92 0. 1. 1.92 0. ]
 [0. 1.22 1.51 1.22 0. 0. 1. 0. 1.22]]
```

- \* Term frequency of *and* in document 3 is  $tf(1, 3) = 1$ .
- \* Document frequency of *and* is  $df(1) = 1$ ,  
 $idf(1) = 1 + \log((1 + 4)/(1 + 1)) = 1.92$
- \* TF-IDF of *and* in document 3 is  $tfidf(1, 3) = 1 \times 1.92 = 1.92$
- \* TF-IDF of *document* in document 1 is  
 $tfidf(2, 1) = 1 \times (1 + \log((1 + 4)/(1 + 3))) = 1.22$

## » Using TF-IDF In Bag of Words Model

- \* Recall previously set entry  $i$  of vector equal to number of times token  $i$  appears in document  $d$ . E.g.

|                          |               |             |              |            |          |            |             |
|--------------------------|---------------|-------------|--------------|------------|----------|------------|-------------|
| <i>a good good movie</i> | $\Rightarrow$ | <i>good</i> | <i>movie</i> | <i>not</i> | <i>a</i> | <i>did</i> | <i>like</i> |
| <i>not a good movie</i>  |               | 2           | 1            | 0          | 1        | 0          | 0           |
| <i>did not like</i>      |               | 1           | 1            | 1          | 1        | 0          | 0           |
|                          |               | 0           | 0            | 1          | 0        | 1          | 1           |

- \* Alternative: set entry  $i$  of vector equal to TF-IDF  $tfidf(i, d)$  of token  $i$  for document  $d$ 
  - \* Entries are now real-valued, e.g.

|                          |               |             |              |            |          |            |             |
|--------------------------|---------------|-------------|--------------|------------|----------|------------|-------------|
| <i>a good good movie</i> | $\Rightarrow$ | <i>good</i> | <i>movie</i> | <i>not</i> | <i>a</i> | <i>did</i> | <i>like</i> |
| <i>not a good movie</i>  |               | 2.57        | 1.28         | 0          | 1.28     | 0          | 0           |
| <i>did not like</i>      |               | 1.28        | 1.28         | 1.28       | 1.28     | 0          | 0           |
|                          |               | 0           | 0            | 1.28       | 0        | 1.69       | 1.69        |

- \* Often also normalise feature vector to improve numerics e.g. divide all entries in feature vector  $x$  by  $\sqrt{\sum_{j=1}^n x_j^2}$  or  $\sum_{j=1}^n |x_j|$  where  $n$  is length of vector  $x$

## » Using TF-IDF For Recommending News Articles

Example: Given a new news article find similar news articles

1. Have existing set of  $m$  articles<sup>1</sup>. Map each article to a feature vector using bag of words and TF-IDF, gives set of feature vectors  $x^{(i)}$ ,  $i = 1, 2, \dots, m$
2. New article. Also map to feature vector  $x$  using BOW/TF-IDF
3. Find  $k$  nearest neighbours of  $x$  amongst the  $x^{(i)}$ ,  $i = 1, 2, \dots, m$ 
  - \* What distance metric to use?
  - \* Could use Euclidean distance  $\sqrt{\sum_{j=1}^m (x^{(i)} - x_j)^2}$
  - \* For text also common to use *cosine similarity*  $\sum_{j=1}^m x_j^{(i)} \times x_j$ 
    - \*  $x_j^{(i)} \times x_j$  is large when both  $x_j^{(i)}$  and  $x_j$  are large, and small with one or both are small.
    - \* So cosine similarity is large when articles have many important (as ranked by TF-IDF) words in common

---

<sup>1</sup>We'll use data at <https://www.kaggle.com/snapcrack/all-the-news>

## » Python Code For Recommending News Articles

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np

# first 1000 articles from news dataset at https://www.kaggle.com/snapcrack/all-the-news
text = pd.read_csv('articles1_1000.csv')
text.head()

x = text['content']
vectorizer = TfidfVectorizer(stop_words = 'english', max_df=0.2)
X = vectorizer.fit_transform(x)
indices = np.arange(x.size)

from sklearn.model_selection import train_test_split
train, test = train_test_split(indices, test_size=0.2)

from sklearn.metrics.pairwise import cosine_distances
from sklearn.neighbors import NearestNeighbors
nbrs = NearestNeighbors(n_neighbors=3, metric=cosine_distances).fit(X[train])

test=[test[0]]
found = nbrs.kneighbors(X[test], return_distance=False)

test_i=0
print('text:\n%.300s'%x[test[test_i]])
for i in found[0]:
    print('match %d:\n%.300s'%(i,x[train[i]]))
```



## » Example Output

text:

PARIS – Former President Nicolas Sarkozy has been ordered by a judge to stand trial on charges of illegally financing his failed 2012 campaign, the Paris prosecutors office said on Tuesday, the latest impediment for a politician who not long ago was hoping for a comeback. Mr. Sarkozy, 62, a me  
match 151:

PARIS – New embezzlement allegations emerged on Wednesday against the French presidential candidate François Fillon, adding uncertainty to an already tightly contested election. Mr. Fillons campaign was thrown into turmoil last week after Le Canard Enchaîné, a weekly newspaper that mixes satir  
match 562:

PARIS – France chose an idealistic, traditional candidate in Sundays primary to represent the Socialist and parties in the presidential election this spring. The candidate, Benoît Hamon, 49, who ran on the slogan that he would make Frances heart beat  
match 534:

PARIS – Michel Onfray, a French pop philosopher, was sounding pretty upbeat on the phone, even though the title of his latest book is Decadence: The Life and Death of the Tradition. His book had just come out, with an impressive press run of 120, 000 copies, and was selling briskly in spit

... not too bad! Extend to include the popularity of each news article and this would be a simple recommender system.

## » Word2Vec and GloVe

TF-IDF is v popular: its simple, fast, often works well. *Word2Vec* and *GloVe*<sup>2</sup> differ from TF-IDF:

- \* Map individual token to **short, dense** vectors e.g. vectors with only 300 elements and all elements are non-zero.
  - \* Contrast with one-hot encoding which maps tokens to long, sparse vectors (perhaps 10–50K long and with only one non-zero element)
- \* Map tokens which tend to occur in the same context (i.e. the surrounding text tends to be the same) to vectors which are close to one another → **the mapping reflects word similarity**
  - \* E.g. *shocked* might have nearby words *horrified*, *amazed*, *astonished*, *dismayed*, *stunned*, *appalled*.
- \* Learns mapping from tokens to features vectors from data.
  - \* Word2Vec is **a neural network with one hidden layer** that is trained using a cost function that encourages assigning similar feature vectors to words that occur in similar contexts.
  - \* Needs to be trained on a fairly large dataset to get good results, and text in dataset should reflect the text of interest e.g. Wikipedia text will differ from twitter text.

---

<sup>2</sup>See <https://en.wikipedia.org/wiki/Word2vec> and [https://en.wikipedia.org/wiki/GloVe\\_\(machine\\_learning\)](https://en.wikipedia.org/wiki/GloVe_(machine_learning)). Word2Vec is by Google, GloVe is from Stanford.

## » Word2Vec and GloVe

- \* As with TF-IDF can combine Word2Vec and GloVe with a bag of words model to map from a sequence of words to a feature vector.
  - \* Map each word to a feature vector
  - \* Then sum these vectors element-wise to get a single vector for the sequence of words
  - \* E.g. if two individual word vectors are  $[0.1, 0.7, 0.3]$  and  $[0.6, 0.2, 0.1] \rightarrow$  combined vector is  $[0.7, 0.9, 0.4]$ .
- \* Extra work involved with using Word2Vec and GloVe but can improve generalisation ability of a classifier.

- e.g., NN with one hidden layer. Input target word. Output context words  
- Interesting property of word2vec embeddings: King - man + woman = Queen

<https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>  
<https://towardsdatascience.com/word2vec-explained-49c52b4ccb71>

## » Summary

- \* When working with text first port of call should be TF-IDF plus bag of words. Only look further if that doesn't give good enough performance.
  - \* Provides a baseline against which to compare the performance of other methods
- \* Word2Vec and GloVe can improve performance, but require more work to train (even with a decent GPU) unless use a pre-trained network.
- \* Text is really a *sequence* of words and this is ignored by bag of words models. Taking better account of this takes us into the realm of natural language processing e.g. see BERT<sup>3</sup> → beyond scope of this course.

---

<sup>3</sup>[https://en.wikipedia.org/wiki/BERT\\_\(language\\_model\)](https://en.wikipedia.org/wiki/BERT_(language_model))