CS4618: Artificial Intelligence I

Non-Numeric Features

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:
```
from sklearn.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Class, for use in pipelines, to select certain columns from a DataFram
e and convert to a numpy array
# From A. Geron: Hands-On Machine Learning with Scikit-Learn & TensorFlo
w, O'Reilly, 2017
# Modified by Derek Bridge to allow for casting in the same ways as pand
as.DataFrame.astype
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names, dtype=None):
        self.attribute_names = attribute_names
        self.dtype = dtype
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        X_selected = X[self.attribute_names]
        if self.dtype:
            return X_selected.astype(self.dtype).values
        return X_selected.values

# Class, for use in pipelines, to binarize nominal-valued features (whil
e avoiding the dummy variabe trap)
# By Derek Bridge, 2017
class FeatureBinarizer(BaseEstimator, TransformerMixin):
    def __init__(self, features_values):
        self.features_values = features_values
        self.num_features = len(features_values)
        self.labelencodings = [LabelEncoder().fit(feature_values) for fe
ature_values in features_values]
        self.onehotencoder = OneHotEncoder(sparse=False,
            n_values=[len(feature_values) for feature_values in features
_values])
        self.last_indexes = np.cumsum([len(feature_values) - 1 for featu
re_values in self.features_values])
    def fit(self, X, y=None):
        for i in range(0, self.num_features):
            X[:, i] = self.labelencodings[i].transform(X[:, i])
        return self.onehotencoder.fit(X)
    def transform(self, X, y=None):
        for i in range(0, self.num_features):
            X[:, i] = self.labelencodings[i].transform(X[:, i])
        onehotencoded = self.onehotencoder.transform(X)
        return np.delete(onehotencoded, self.last_indexes, axis=1)
    def fit_transform(self, X, y=None):
        onehotencoded = self.fit(X).transform(X)
        return np.delete(onehotencoded, self.last_indexes, axis=1)
    def get_params(self, deep=True):
        return {"features_values" : self.features_values}
    def set_params(self, **parameters):
        for parameter, value in parameters.items():
            self.setattr(parameter, value)
        return self
```

Data types

- Structured data:
  - **Numeric-valued**: either real- or integer-valued, such as floor area or number of bedrooms
  - **Nominal-valued**: where there is a finite set of possible values. Often these values are strings
    - For example, dwelling type ($type$) is a nominal-valued feature whose values are "Apartment", "Detached", "Semi-detached" or "Terraced".
    - The special case here is, of course, a binary-valued feature, where there are just two values. For example, the type of development ($devment$) is a nominal-valued feature whose values are "New" or "SecondHand"
    - Another special case is where there is a finite set of possible values but there is some ordering on the values, e.g. the spiciness of a curry can be "Mild", "Medium", "Hot", "Very Hot" and "Suicidal"
  - **Set-valued**: where the value of a feature is a set, but the members of the set are constrained to a finite set of nominals. For example, the genre of a movie might be a set-valued feature, e.g. the value of the genre feature for *The Blues Brothers* is $\{musical, comedy, action\}$.
  - …
- Unstructured:
  - free-form text
  - media such as images, audio and video

Data Types in the Cork Propery Dataset

$flarea$	numeric	the floor area in square metres
$type$	nominal	dwelling type: Apartment, Detached, Semi-detached, Terraced
$bdrms$	numeric	the number of bedrooms
$bthrms$	numeric	the number of bathrooms
$floors$	numeric	the number of floors
$devment$	nominal	the type of development: New or SecondHand
ber	nominal	building energy rating: A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, E1, E2, F, G
$location$	nominal	the area of Cork, e.g. Douglas, Glanmire, Wilton,...

In [4]:
```
# Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")
```

In [5]:
```
# The datatypes
df.dtypes
```

Out[5]:
```
flarea      float64
type         object
bdrms         int64
bthrms        int64
floors        int64
devment      object
ber          object
location     object
price         int64
dtype: object
```

In [6]:
```
# Summary statistics
df.describe(include="all")
```

Out[6]:

	flarea	type	bdrms	bthrms	floors	devment	ber	location
count	207.000000	207	207.000000	207.000000	207.000000	207	207	207
unique	NaN	4	NaN	NaN	NaN	2	12	36
top	NaN	Semi-detached	NaN	NaN	NaN	SecondHand	G	CityCentre
freq	NaN	65	NaN	NaN	NaN	204	25	40
mean	128.094686	NaN	3.434783	2.106280	1.826087	NaN	NaN	NaN
std	73.970582	NaN	1.232390	1.185802	0.379954	NaN	NaN	NaN
min	41.800000	NaN	1.000000	1.000000	1.000000	NaN	NaN	NaN
25%	82.650000	NaN	3.000000	1.000000	2.000000	NaN	NaN	NaN
50%	106.000000	NaN	3.000000	2.000000	2.000000	NaN	NaN	NaN
75%	153.650000	NaN	4.000000	3.000000	2.000000	NaN	NaN	NaN
max	497.000000	NaN	10.000000	10.000000	2.000000	NaN	NaN	NaN

In [7]:
```
# A few of the examples
df.head(3)
```

Out[7]:

	flarea	type	bdrms	bthrms	floors	devment	ber	location	price
0	497.0	Detached	4	5	2	SecondHand	B2	Carrigrohane	975
1	83.6	Detached	3	1	1	SecondHand	D2	Glanmire	195
2	97.5	Semi-detached	3	2	2	SecondHand	D1	Glanmire	225

Handling Nominal-Valued Features

- Most AI algorithms work only with numeric-valued features (There are exceptions)
- So, we will look at how to convert nominal-valued features to numeric-valued ones

Binary-valued features

- The simplest case, obviously, is a binary-valued feature
- We encode one value as 0 and the other as 1, e.g. "SecondHand" is 0 and "New" is 1

Unordered nominal values

- Suppose there are more than two values, e.g. Apartment, Detached, Semi-detached or Terraced.
- The obvious thing to do is to assign integers to each nominal value, e.g. 0 = Apartment, 1 = Detached, etc.
- But often this is not the best encoding
  - Algorithms may assume that the values themselves are meaningful, when they're actually arbitrary
    - E.g. an algorithm might assume that Apartments (0) are more similar to Detached houses (1) than they are to Terraced houses (3)
- Instead, we use **one-hot encoding**

One-Hot Encoding

- If the original nominal-valued feature has p values, then we use p binary-valued features:
  - In each example, exactly one of them is set to 1 and the rest are zero
- For example, there are four types of dwelling, so we have four binary-valued features:
  - The first is set to 1 if and only if the type of dwelling is Apartment
  - The second is set to 1 if and only if the house is Detached
  - And so on

  So a detached house will have $[0, 1, 0, 0]$ as their values
- Some questions:
  - One-hot encoding replaces one nominal-valued feature that has p values by p binary-valued ones — in general, one feature per nominal value. (E.g. $type$ has four values, so we get four binary features.) What is the minimum number of binary-valued features we could use?
  - Why don't we use the minimum?
  - Although we get p binary features, we only need $p - 1$. How come? (Advanced note: Look up the *dummy variable trap* to see why this might even be preferable)
  - How might one encode a set-valued feature (such as the movie genre example above)?
- In practice, it is not uncommon to be given a dataset where a nominal-valued feature has already been encoded numerically, one integer per value. You might be fooled into thinking that the feature is numeric-valued and overlook the need to use one-hot encoding on it. Watch out for this!

Ordered nominal values

- Consider the case now of a feature whose values are nominal but where there *is* an ordering
  - E.g. the ber feature in the housing dataset is like this
  - In this case, G < F < E2 < E1 < D1 ... < A1
- Some people would use the phrase 'ordinal-valued' to refer to nominal values that have an ordering
- You might be tempted to use a straightforward numeric encoding
  - E.g. 0 = G, 1 = F, 2 = E2, 3 = E1, and so on
  - This encoding preserves the ordering, e.g. that E2 < E1 because 2 < 3
  - But again this is probably not the best encoding
    - The original feature had an ordering on its values but no notion of distance
    - E.g. G < F but you cannot say by *how much* G is less than F
    - In the new feature, we have introduced a notion of distance: G is worse than F by 1, and it is 2 worse than E2
    - So this encoding has *added* 'information' that was not present in the original
- So what should we do?
  - We could use one-hot encoding: fifteen binary-valued features. But what are the weaknesses of this?
  - Another option is to use binary-valued features that represent inequalities
    - E.g. one feature is set to 1 if you have a BER of at least G
    - Another is additionally set to 1 if you have attained at least F
    - And so on

    — still fifteen binary-valued features, but no longer mutually exclusive
    - E.g. a BER of E2 is converted to the following fifteen binary-valued features: $[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
    - E.g. a BER of E1 is converted to $[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

  But, since scikit-learn doesn't offer this somewhat sophisticated encoding, and assuming we don't write our own, we will have to use one-hot encoding
- Again watch out for cases where some well-intentioned person has already encoded this kind of feature but using a naive numeric encoding

The curse of dimensionality, again

- One-hot encoding increases the number of features, sometimes quite a lot
- We may need to use dimensionality reduction (although most people don't bother!)
  - Don't use PCA, which is for numeric-valued features
  - Use, e.g., Canonical Correspondence Analysis (CCA)

Handling Nominal Values in scikit-learn

- We will add extra steps into our pipeline to convert nominal-values features into numeric ones
  - scikit-learn has some classes for doing this but they do not play nicely with pipelines, so we wll use my `FeatureBinarizer` (given earlier) instead
  - (Advanced: `FeatureBinarizer` avoids the dummy variable trap and uses just $p-1$ binary features)
- But, we now need two pipelines:
  - One takes all the numeric-valued features and, e.g., scales them
  - The other takes the numeric-valued features and their legal values and binarizes them

  You then join the pipelines using `FeatureUnion`

In [8]:
```
# The features we want to select
numeric_features = ["flarea", "bdrms", "bthrms", "floors"]
nominal_features = ["type", "devment", "ber", "location"]

# Create the pipelines
numeric_pipeline = Pipeline([
        ("selector", DataFrameSelector(numeric_features)),
        ("scaler", StandardScaler())
    ])

nominal_pipeline = Pipeline([
        ("selector", DataFrameSelector(nominal_features)),
        ("binarizer", FeatureBinarizer([df[feature].unique() for feature
in nominal_features]))])

pipeline = Pipeline([("union", FeatureUnion([("numeric_pipeline", numeri
c_pipeline),
                                             ("nominal_pipeline", nomina
l_pipeline)]))])
```

In [9]:
```
# Run the pipeline
pipeline.fit(df)
X = pipeline.transform(df)
```

```
In [10]: # Let's take a look at a few rows in X - to show you that we now have a
         2D numpy array
         print(X[:3])
```

```
[[ 4.99927973  0.45974713  2.4462228   0.45883147  0.          1.
0.
   1.          0.          1.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
1.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.        ]
 [-0.6029769  -0.35365164 -0.93520037 -2.17944947  0.          1.
0.
   1.          0.          0.          0.          0.          0.
0.
   0.          1.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   1.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.        ]
 [-0.41460881 -0.35365164 -0.08984458  0.45883147  0.          0.
1.
   1.          0.          0.          0.          0.          0.
0.
   1.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   1.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.          0.
0.
   0.          0.          0.          0.          0.        ]]
```

In [11]:
```
# So which house is most similar to yours, now that we are using all the
features?

def euc(x, xprime):
    return np.sqrt(np.sum((x - xprime)**2))

# Don't try to understand or copy this code - it's a hack that you won't
need
your_house_df = pd.DataFrame([{"flarea":114.0, "type":"Semi-detached", "
bdrms":3, "bthrms":2, "floors":2,
                               "devment":"SecondHand", "ber":"B2", "loca
tion":"Glasheen"}])
your_house_scaled = pipeline.transform(your_house_df)[0]

df.ix[np.argmin([euc(your_house_scaled, x) for x in X])]
```

Out[11]:
```
flarea              134.7
type        Semi-detached
bdrms                   3
bthrms                  2
floors                  2
devment        SecondHand
ber                    D1
location         Glasheen
price                 245
Name: 127, dtype: object
```

- Actually, there is a question of whether Euclidean distance is the best distance measure to use on nominal-valued features and on mixtures of numeric-valued features and nominal-valued features
- But, in this introductory module, we will use it!

Free-Form Text

- Suppose the objects in your dataset are **documents**, rather than houses
  - E.g. web pages, tweets, blog posts, emails, posts to Internet forums and chatrooms, …
  - They might have a little structure to them (headings and so on), but they are primarily **free-form text**
- Many AI algorithms can only handle vectors of numbers. So one way to apply AI techniques to a dataset of documents is to convert the raw text in the documents into vectors of numbers
- Our treatment of this will be brief and high-level, since many of you are studying *CS4611 Information Retrieval*, where this is covered in depth
- Furthermore, we'll use scikit-learn although its facilities for handling text are quite limited. If you really want to do AI with text, consider a more powerful library such as *NLTK* (http://www.nltk.org/ (http://www.nltk.org/)) or the *Stanford Natural Language Processing Toolkit* (https://nlp.stanford.edu/software/ (https://nlp.stanford.edu/software/))

Running Example

Suppose our dataset contains just these three documents:

Tweet 0	Tweet 1	Tweet 2
No one is born hating another person because of the color of his skin or his background or his religion.	People must learn to hate, and if they can learn to hate, they can be taught to love.	For love comes more naturally to the human heart than its opposite.

Three tweets from Barack Obama, quoting Nelson Mandela

Bag-of-words representation

- **Tokenize** each document
  - In our simple treatment, the tokens are just the words, ignoring punctuation and making everything lowercase
  - In reality, this is surprisingly complicated, e.g. is "don't" one token or two, e.g. maybe pairs of consecutive words (so-called 'bigrams') could also be tokens ("no one", "one is", "is born"); and so on
- Optionally, discard **stop-words**: common words such as "a", "the", "in", "on", "is, "are",…
  - Sometimes discarding them helps, or does no harm, e.g. spam detection
  - Other times, you lose too much, e.g. web search engines ("To be, or not to be")

```
In [12]: from sklearn.feature_extraction import stop_words

         print(stop_words.ENGLISH_STOP_WORDS)
```

```
frozenset({'yet', 'though', 'on', 'until', 'somewhere', 'out', 'whereby',
'forty', 'hasnt', 'hereupon', 'latter', 'yours', 'is', 'whither', 'might'
, 'serious', 'nine', 'whether', 'are', 'five', 'such', 'what', 'am', 'the
rein', 'thereupon', 'while', 'somehow', 'couldnt', 'where', 'thereby', 'n
ever', 'ours', 'you', 'further', 'within', 'twenty', 'last', 'some', 'emp
ty', 'everywhere', 'something', 'thick', 'onto', 'those', 'about', 'nor',
'wherein', 'there', 'whatever', 'be', 'become', 'which', 'himself', 'nowh
ere', 'yourself', 'via', 'amoungst', 'ever', 'often', 'his', 'against', '
always', 'will', 'everyone', 'namely', 'three', 'again', 'anyhow', 'whene
ver', 'from', 'per', 'rather', 'co', 'many', 'had', 'third', 'since', 'an
yway', 'this', 'whereafter', 'either', 'and', 'without', 'by', 'even', 'e
leven', 'it', 'mine', 'up', 'moreover', 'noone', 'perhaps', 'became', 'fi
fty', 'any', 'please', 'too', 'could', 'detail', 'thin', 'fire', 'un', 'a
ll', 'show', 'hereby', 'indeed', 'can', 'find', 'not', 'whom', 'meanwhile
', 'over', 'hundred', 'four', 'see', 'now', 'sometime', 'enough', 'althou
gh', 'side', 'sixty', 'back', 'being', 'along', 'under', 'he', 'would', '
anyone', 'part', 'made', 'seemed', 'the', 'only', 'across', 'below', 'of'
, 'sometimes', 'whole', 'bill', 'someone', 'hereafter', 'same', 'if', 'cr
y', 'therefore', 'ten', 'elsewhere', 'toward', 'seeming', 'otherwise', 'w
ith', 'few', 'another', 'cannot', 'in', 'full', 'one', 'top', 'whence', '
ie', 'describe', 'move', 'my', 'cant', 'herein', 'ltd', 'so', 'neverthele
ss', 'behind', 'themselves', 'do', 'several', 'thus', 'alone', 'call', 'd
own', 'i', 'our', 'anywhere', 'less', 'than', 'towards', 'may', 'whereupo
n', 'your', 'fifteen', 'or', 'still', 'was', 'well', 'inc', 'becoming', '
at', 'me', 'becomes', 'for', 'mostly', 'should', 'nothing', 'off', 'eg',
'during', 'that', 'we', 'six', 'these', 'but', 'seems', 'were', 'already'
, 'thereafter', 'hence', 'once', 'because', 'give', 'interest', 'him', 'n
one', 'everything', 'mill', 'put', 'get', 'them', 'whose', 're', 'anythin
g', 'front', 'others', 'into', 'she', 'name', 'very', 'found', 'beside',
'con', 'after', 'bottom', 'both', 'here', 'among', 'how', 'most', 'hers',
'go', 'above', 'through', 'system', 'thence', 'afterwards', 'between', 'f
ormer', 'else', 'next', 'they', 'together', 'her', 'twelve', 'as', 'has',
'amongst', 'due', 'sincere', 'ourselves', 'beforehand', 'first', 'also',
'before', 'more', 'wherever', 'who', 'yourselves', 'its', 'almost', 'wher
eas', 'upon', 'herself', 'however', 'why', 'no', 'must', 'a', 'thru', 'et
c', 'de', 'itself', 'formerly', 'keep', 'each', 'an', 'other', 'much', 'm
yself', 'beyond', 'been', 'seem', 'own', 'throughout', 'besides', 'done',
'neither', 'when', 'their', 'every', 'around', 'nobody', 'to', 'then', 'e
xcept', 'least', 'us', 'amount', 'eight', 'fill', 'latterly', 'whoever',
'have', 'take', 'two'})
```

- Optionally, apply **stemming** or **lemmatization** to the words
  - E.g. "hating" is replaced by "hate", "comes" is replaced by "come"
  - scikit-learn doesn't have a stemmer, but does make it easy to call one, if you get one from another library, e.g. NLTK
- **Count Vectorize**: each document becomes a vector, each token becomes a feature, feature-values are *frequencies* (how many times that token appears in that document)
  (In *CS4611*, features are probably referred to as 'terms')
- Optionally, **TD-IDF Vectorize**: replace the frequencies by **tf-idf** scores
  - tf-idf scores penalise words that recur across multiple documents
  - E.g. in emails, word such as "hi", "best", "regards", …
  - For the formulae, see *CS4611*
    - variants might: scale frequencies to avoid biases towards long documents (not scikit-learn); logarithmically scale frequencies (not default in scikit-learn); add 1 to part of the formula to avoid division-by-zero (default in scikit-learn); normalize the results (e.g. by default, scikit-learn divides by the L2-norm)

Running Example

- After discarding stop-words:

Tweet 0	Tweet 1	Tweet 2
born hating person color skin background religion	people learn hate learn hate taught love	love comes naturally human heart opposite

- After count vectorization:

	background	born	color	comes	hate	hating	heart	human	learn	love	naturally	opposite	people
Tweet 0:	1	1	1	0	0	1	0	0	0	0	0	0	0
Tweet 1:	0	0	0	0	2	0	0	0	2	1	0	0	1
Tweet 2:	0	0	0	1	0	0	1	1	0	1	1	1	0

- After tf-idf vectorization:

	background	born	color	comes	hate	hating	heart	human	learn	love	naturally	opposite	people
Tweet 0:	0.38	0.38	0.38	0	0	0.38	0	0	0	0	0	0	0
Tweet 1:	0	0	0	0	0.61	0	0	0	0.61	0.23	0	0	0.31
Tweet 2:	0	0	0	0.42	0	0	0.42	0.42	0	0.32	0.42	0.42	0

The dimension of these vectors

- Sparsity:
  - Here we had $n = 17$ features (columns). How many will there be in general?
  - Most of the feature-values are zero. Why?
  - We say that the matrix is **sparse**
  - It would be wasteful to store it using very long arrays. We need a data structure that only stores the non-zero elements: **sparse matrices**
    (Don't worry: scikit-learn takes care of this 'behind the scenes')
- The curse of dimensionality, yet again:
  - Reduce the number of features by
    - discarding tokens that appear in too few documents (`min_df` in scikit-learn)
    - discarding tokens that appear in too many documents (`max_df`)
    - keeping only the most frequent tokens (`max_features`)
  - Use dimensionality reduction:
    - E.g. singular value decomposition (SVD) is suitable for bag-of-words, rather than PCA

Observation about bag-of-words representations

- This representation is good for many applications in AI but it does have drawbacks too:
  - It loses all the information that English conveys through the order of words in sentences
    - E.g. "People learn to hate" and "People hate to learn" have very different meanings but end up with the same bag-of-words representation
  - It loses the information that English conveys using its stop-words, most notably negation
    - E.g. "They hate religion" and "I do not hate religion" will have the same bag-of-words representation
- This may not matter for some applications (e.g. spam detection) but will matter for others (e.g. machine translation), for which you need a different representation
- What other weaknesses does it have?

Bag-of-words representation in scikit-learn

In [13]:
```
tweets = [
    "No one is born hating another person because of the color of his sk
in or his background or his religion.",
    "People must learn to hate, and if they can learn to hate, they can
be taught to love.",
    "For love comes more naturally to the human heart than its opposite.
"
]
```

- In the example below, we put a `CountVectorizer` into a pipeline
- It does tokenization
  - By default, it converts to lowercase, it treats punctuation as spaces, and it treats two or more consecutive characters as a word. Each word becomes a token (feature)
- The example below discards stop-words using the list we saw earlier
- It also, by default, discards any word that appears in every document
- It does not do stemming or lemmatization but there are ways of incorporating a stemmer from, e.g., NLTK
- Finally, it vectorizes, producing sparse matrices of word fequencies. (There is an option to produce a binary representation, instead of frequencies)

In [14]:
```
# Create the pipeline
text_pipeline = Pipeline([
        ("vectorizer", CountVectorizer(stop_words='english'))
    ])

# Run the pipeline
text_pipeline.fit(tweets)
X = text_pipeline.transform(tweets)
```

```
In [15]: # Let's see the features
         text_pipeline.named_steps["vectorizer"].get_feature_names()

Out[15]: ['background',
          'born',
          'color',
          'comes',
          'hate',
          'hating',
          'heart',
          'human',
          'learn',
          'love',
          'naturally',
          'opposite',
          'people',
          'person',
          'religion',
          'skin',
          'taught']
```

```
In [16]: # We can look at the sparse array. The first number identifies the tweet
         (0, 1 or 2), the second is which feature, and
         # the last is the frequency
         print(X)

           (0, 0)        1
           (0, 1)        1
           (0, 2)        1
           (0, 5)        1
           (0, 13)       1
           (0, 14)       1
           (0, 15)       1
           (1, 4)        2
           (1, 8)        2
           (1, 9)        1
           (1, 12)       1
           (1, 16)       1
           (2, 3)        1
           (2, 6)        1
           (2, 7)        1
           (2, 9)        1
           (2, 10)       1
           (2, 11)       1
```

```
In [17]: # Vectorize a new document
         new_document = "Unsurprisingly, people hate to learn that their religion
         loves to hate."

         new_document_as_vector = text_pipeline.transform([new_document])
```

```
In [18]: # Notice how it ignores words that weren't in the original tweets, such
         as "unsurprisingly" and "loves"

         print(new_document_as_vector)

           (0, 4)        2
           (0, 8)        1
           (0, 12)       1
           (0, 14)       1
```

- In the example below, we put a `TfidfVectorizer` into a pipeline instead
- By default, it normalizes the values using the L2 norm (see CS46111)

In [19]:
```
# Create the pipeline
text_pipeline = Pipeline([
        ("vectorizer", TfidfVectorizer(stop_words='english'))
    ])

# Run the pipeline
text_pipeline.fit(tweets)
X = text_pipeline.transform(tweets)
```

In [20]:
```
print(X)
```

```
  (0, 15)	0.377964473009
  (0, 14)	0.377964473009
  (0, 13)	0.377964473009
  (0, 5)	0.377964473009
  (0, 2)	0.377964473009
  (0, 1)	0.377964473009
  (0, 0)	0.377964473009
  (1, 16)	0.307460988215
  (1, 12)	0.307460988215
  (1, 9)	0.233832006484
  (1, 8)	0.614921976431
  (1, 4)	0.614921976431
  (2, 11)	0.423394483412
  (2, 10)	0.423394483412
  (2, 9)	0.322002417819
  (2, 7)	0.423394483412
  (2, 6)	0.423394483412
  (2, 3)	0.423394483412
```

In [21]:
```
# Vectorize a new document
new_document = "Unsurprisingly, people hate to learn that their religion loves to hate."

new_document_as_vector = text_pipeline.transform([new_document])
```

In [22]:
```
# Notice how it ignores words that weren't in the original tweets, such as "unsurprisingly" and "loves"

print(new_document_as_vector)
```

```
  (0, 14)	0.377964473009
  (0, 12)	0.377964473009
  (0, 8)	0.377964473009
  (0, 4)	0.755928946018
```

Similarity & distance for bag-of-words representation

- For details and formulae, see CS4611
- Euclidean distance is not suitable
- Very common is **cosine similarity**, which gives values in $[0, 1]$, where 1 means 'identical'
- To get **cosine distance**, we can subtract from 1, so now 1 means 'completely different'
- The exact formulae differ depending on what is assumed about normalization
  - If we assume the vectors have been normalized, then simpler formula
  - If not, then the formula is more complicated

Similarity & distance for bag-of-words representation in scikit-learn

- The code below assumes that the vectors have already been normalized, e.g. produced by `TfidfVectorizer`

```
In [23]: def cosine(x, xprime):
             # Assumes x and  xprime are already normalized
             # Converts from sparse matrices because np.dot does not work on them
             return 1 - x.toarray().dot(xprime.toarray().T)
```

```
In [24]: # So which of Barack Obama's tweets is most similar to our new document?
         tweets[np.argmin([cosine(new_document_as_vector, x) for x in X])]
```

```
Out[24]: 'People must learn to hate, and if they can learn to hate, they can be ta
         ught to love.'
```

```
In [ ]:
```