Introduction to Text classification

approaches, workflows, and demos

NORC Data Science Brown Bag

timothy leffel // http://lefft.xyz // may29/2019

logistics

- you can find all the materials here today:
  - `https://github.com/lefft/text_classification`
  - (and at an internal location after today)
- if you want to experiment with the data and code on your own machine afterwards, you'll need:
  - Python 3.6+, with `sklearn`, `pandas`, and `numpy` installed

outline

1. a motivating example (5min)
2. what is text classification? (5min)
3. typical workflows (10min)
4. NORC use cases (5min)
5. demos (20min):
    - 4.1 topic detection (tweets)
    - 4.2 utterance type classification (movie scripts)

goals for today

- informal overview of what text classification is and why it's important for social science
- introduce some of the fundamental concepts and techniques used in text clf
- illustrate what modern tooling for text clf looks like

1. a motivating example

wading through a swamp of tweets

Electronic nicotine delivery systems – aka vaping devices – have exploded in popularity in recent years.

Suppose we want to conduct some exploratory text analysis on social media posts mentioning the most popular vaping product around – **Juul**.

wading through a swamp of tweets (con't)

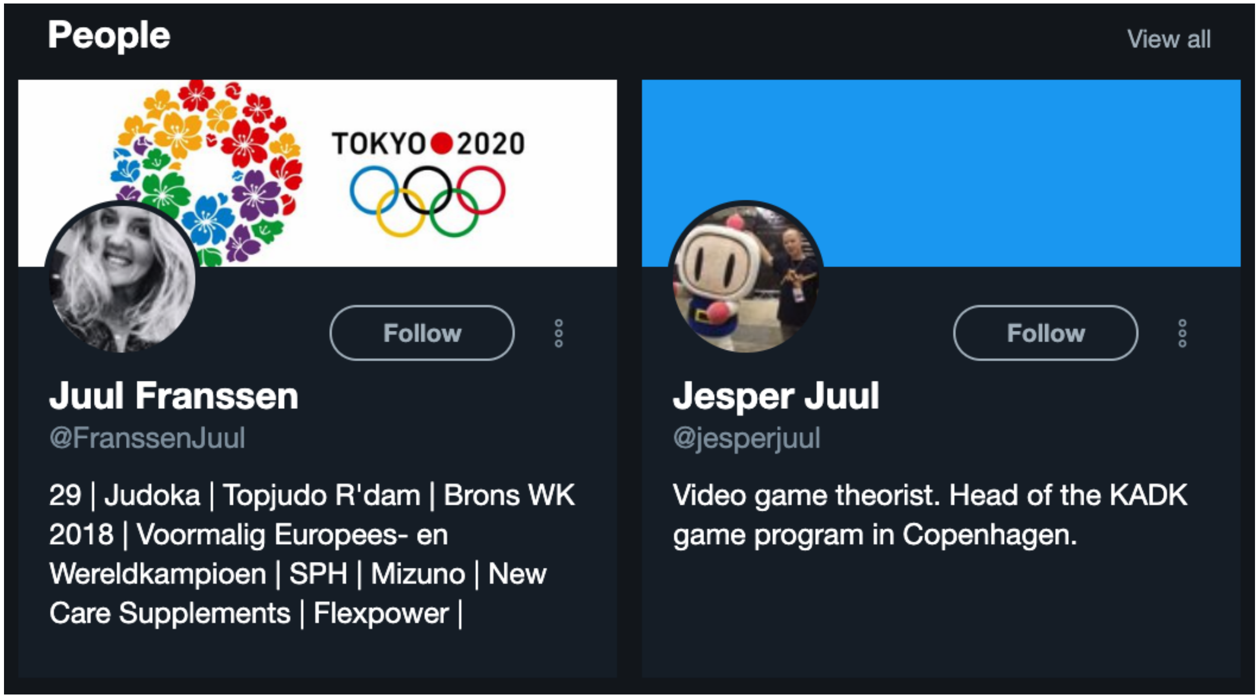
A natural approach:

1. Use general keywords like `"juul"` and `"virginia tobacco"` to collect tweets from the last (say) 12 months.
2. Extract and count frequent hashtags and words within the dataset.
3. Qualitatively inspect those lists and derive insights from them.

wading through a swamp of tweets (con't)

Problem:

- Juul doesn't have a monopoly on the string `'juul'` – and neither does English!
- Lots of content on Twitter contains `'juul'` but isn't relevant for us



wading through a swamp of tweets (con't)

Solution:

- Develop a **Juul-relevance tweet classifier**
- Remove tweets that the model predicts to be irrelevant before analysis.
- Then proceed with analysis as originally planned.

==> improvement in data quality, and therefore also in **any conclusions** drawn from the data.

==> without an intermediate classification step, analytic dataset would have unknown, undesirable, and/or unanticipated properties.

2. what is text classification?

definitions and concepts

- **Text classification** (“text clf”) is any activity that involves systematically assigning discrete categories to blobs of text – usually involves a statistical or symbolic model implemented using computers.
- Examples of specific text clf tasks:
  - detection of topics/themes in social media post text
  - inference of user intent in digital assistants (like Siri or Alexa)
  - detection of positive/negative sentiment in product reviews
  - spam filtering (and email filters more generally)
  - language detection for machine translation
  - Minority Report-style prediction of criminal intents from text messages
  - many, many more…

Today we’ll be talking about *document* classification (and not e.g. classification of words or phrases within sentences – another kind of text classification).

definitions and concepts

Here's a basic setup for developing a *supervised document classification model*:

- there is a collection of **documents** – blobs of text like social media posts or news stories or entire books or transcribed speech, etc.
- each document is associated with a **label** – a discrete category that's (usually) been assigned by a human annotator
- the name of the game is to find a function f that accurately maps text documents to categories – ultimately f will be used to predict labels for documents that haven't been annotated by humans.

definitions and concepts

A **document classifier** is nothing but a function f that takes text documents as inputs and returns values from some pre-determined set of categories.

A few more terms we'll be using:

- each label represents a **class** – think of classes as levels of a factor, or values a categorical variable can take on, or just as the set of items sharing a label.
- each word, character, hashtag, n-gram, etc. in a document represents a potential **feature** – think of features just like "predictors" in a regression modeling context (since documents are ultimately transformed into rows of a large matrix).

approaches to document classification

There are many approaches one can take to classifying text documents.

Here's a few important ones:

- symbolic/rule-based classifiers;
- traditional statistical/data-driven classifiers;
- new-wave black-box models (also data-driven); and
- the ultimate, universal text classifier…

Rule-based classifiers (symbolic)

Uses word-lists, regular expressions, and heuristics to “manually” define a classification function.

Here’s a simple (but low quality) Juul-relevance classifier:

- if document contains `'juul'` surrounded by word-boundaries, assign *relevant*
- otherwise, assign *irrelevant*

Rule-based classifiers don’t require any annotated data (beyond what’s needed to assess performance), and so can be very useful if resources are limited.

Advantage: you know *exactly* how model predictions come to be.

Disadvantage: you probably won’t think of every weird exception, and the subtleties of language will crush your dreams </3

Traditional statistical models (statistical, data-driven)

Transform the collection of documents into a **document-term matrix**, then can use familiar regression modeling techniques for categorical outcome variables. Very common is the **“bag of words”** encoding shown below.

	it	is	puppy	cat	pen	a	this
it is a puppy	1	1	1	0	0	1	0
it is a kitten	1	1	0	0	0	1	0
it is a cat	1	1	0	1	0	1	0
that is a dog and this is a pen	0	2	0	0	1	2	1
it is a matrix	1	1	0	0	0	1	0

Advantage: Nice balance of interpretability and ability to learn non-obvious patterns in data; also allows the analyst to let domain knowledge guide the process of manually engineering features.

Disadvantage: Requires decent sized human-annotated dataset.

(brief interlude: n-gram tokenization)

Sometimes we need different features – an n-**gram** is a contiguous sequence of n tokens (e.g. words or characters), a fundamental data type in NLP.

Here's an example of word bigram tokenization:

```
The dog chased the cat ==> [The dog, dog chased, chased the, the cat]
```

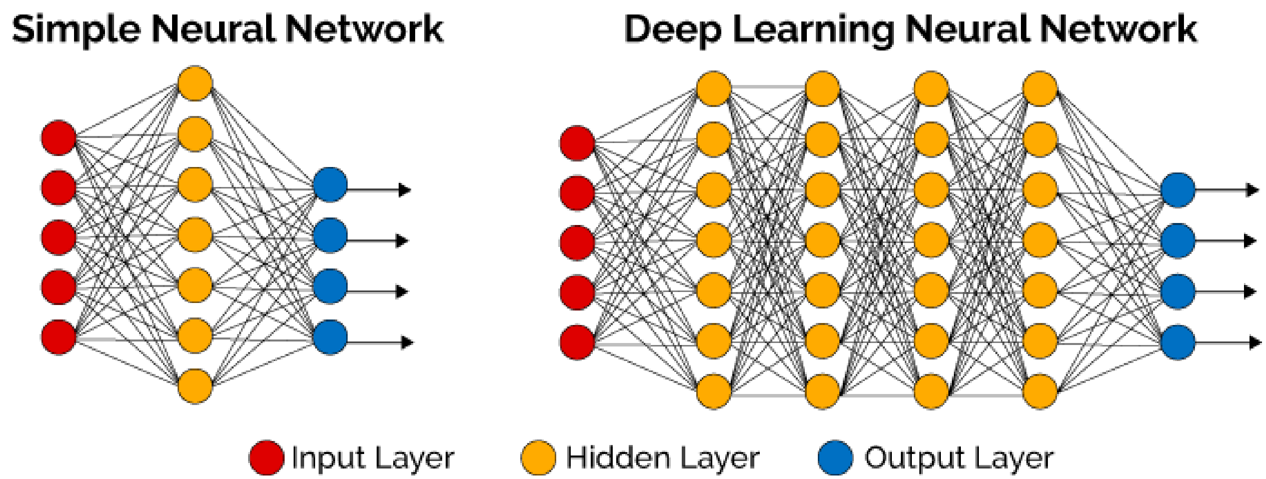
Here's an example of word {1,2}-gram tokenization:

```
The dog chased the cat ==>
  [The, The dog, dog, dog chased, chased, chased the, the, the cat, cat]
```

Surprisingly powerful – character bigram tokenization:

```
The dog chased the cat ==>
  [Th, he, e ,  d, do, og, g ,  c, ch, ha, as, se, ed, d , ...]
```

New-wave black-box models (statistical, data-driven)



Advantages: Extremely powerful for capturing highly complex patterns in data; enables flexible representations of documents via transfer learning; basically no manual feature engineering required.

Disadvantages: Extremely data hungry; predictions are usually completely opaque (w/o hours of follow-up analysis); very easy to overfit and essentially memorize a dataset (poor generalization).

The ultimate, universal text classifier (*no idea* how it works…)

- The true gold standard is almost always human judgment.
- But human judgments are expensive to collect at a large scale…

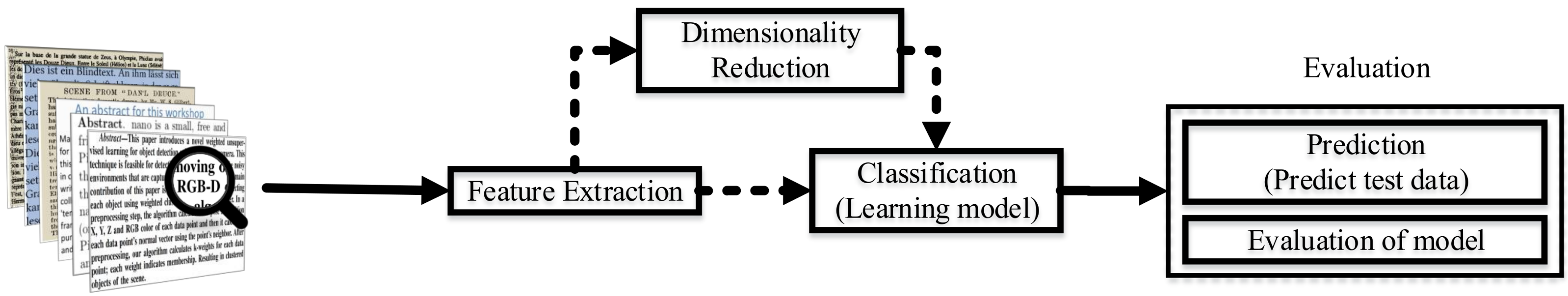
- *Advantage*: Yields the highest quality “predictions” available.
- *Disadvantage*: Compared to computers, human brains are very slow at reading and analyzing texts. Plus you never stop having to pay them.

3. a document classification workflow

text classification pipelines

Structurally, document classification is very similar to other kinds of statistical modeling in which the outcome/dependent variable is categorical.

The main difference is that document clf has the extra step of *feature extraction*: How do we turn a bunch of text documents into a numerical matrix we can use in e.g. a regression model?



c/o: `https://res.mdpi.com`

text classification pipelines (common structure)

1. text preprocessing
    - regularize text – e.g. convert to lowercase, remove irrelevant punctuation, collapse morphological variants to a common form (lemmatization), etc.
2. feature engineering
    - choose how to **tokenize** the text, and how to transform each list of tokens into a vector of values (e.g. they could be word counts, raw or weighted)
3. model selection and tuning, then training
    - what algorithms to try? what parameter spaces are relevant for each?
4. evaluation of model performance
    - what's a good performance metric for the model? (task-dependent!)
5. out-of-sample prediction (releasing the hounds)
    - the moment of truth – the model is on its own from here.

4. NORC text classification use cases

text classification at NORC

- social media and other web data
  - targeted topic detection (you know in advance what you're looking for)
  - sentiment analysis (with pre-defined categories)
- survey data analysis and administration
  - coding of responses to free-form survey items
  - coding of communications from/with respondents (e.g. transcribed voicemails?)
- collection and analysis of `.html` pages scraped from the web?
- other project work that some of you are probably involved in!

demo 1: topic detection in tweet text (binary)

the problem

- you want to analyze every tweet that talks about **Juul** from the last year;
- so you used some broad keywords like “juul” and “classic tobacco” to collect tweets;
- but these keywords (inevitably) captured tens of thousands of irrelevant posts in addition to the hundreds of thousands that are indeed relevant to Juul.
- **it’s not feasible to manually go through every tweet to weed out the junk.**

So what do you do?

- Hand-label a couple thousand sample tweets for relevance to Juul, and then
- develop a Juul-relevance tweet classifier!

a first pass at the problem

Sample dataset of 600 labeled tweets:

- `../data/tweet_samples-600.txt`

Summary file with model config and performance for a few fits:

- `../output/tweet_samples-model_info.txt`

Python script illustrating text classification workflow with `scikit-learn`:

- `../code/classify_tweet_relevance.py`

We'll go thru the highlights on the next few slides, but dig into the materials to learn more.

load and split data into train/test subsets (step 0)

```
import pandas as pd

tweets_infile = '../data/tweet_samples-600.txt'
dat = pd.read_csv(tweets_infile, sep='|', encoding='utf-8')

docs, labs = dat['text'].tolist(), dat['label'].tolist()

# some sample data points
dat[['text','label']].head(4)
```

```
##                                                text  label
## 0  RT @Juul16261: @Alfred_ot2017 @eurovision_tve ...      0
## 1  RT @sofie_druckrey: what if the green light th...      1
## 2  RT @alxsatrum: Not being addicted to nicotine😍...      1
## 3  @mishtiicy is juuling another version of vapin...      1
```

load and split data into train/test subsets (step 0)

```
from sklearn.model_selection import train_test_split

docs_train, docs_test, labs_train, labs_test = train_test_split(
  docs, labs, test_size=.33, random_state=6933)
```

text prep and feature engineering (pipeline steps 1-2)

`scikit-learn` combines these two steps in seamless fashion:

```
import re

def prep_text(doc, toss_re=r'[?.,!$-]'):
  '''simple text preprocessing routine (remove some punctuation)'''
  return re.sub(toss_re, '', doc)
```

text prep and feature engineering (steps 1-2, con't)

```
from sklearn.feature_extraction.text import CountVectorizer

# text preprocessing and feature extraction choices encoded here
kwargs = {'analyzer': 'word', 'ngram_range': (1, 2),
          'lowercase': True, 'preprocessor': prep_text}

vectorizer = CountVectorizer(token_pattern=r'\b[a-zA-Z0-9_<>]{1,}\b', **kwargs)

# use the training set vocabulary for feature construction
vectorizer.fit(docs_train)
```

```
# create separate doc-term matrices for train and test subsets
X_train = vectorizer.transform(docs_train)
X_test = vectorizer.transform(docs_test)
```

model training (step 3)

(leaving out parameter tuning!)

```
from sklearn.linear_model import LogisticRegression

# fit/train classifier using train features and labels
classifier = LogisticRegression(solver='liblinear')
classifier.fit(X_train, labs_train)
```

```
# generate test set model predictions from test matrix
preds_test = classifier.predict(X_test)
```

evaluation of model performance (step 4)

```
from sklearn.metrics import accuracy_score, f1_score

metrics = {'accuracy': round(accuracy_score(labs_test, preds_test), 3),
           'f1_score': round(f1_score(labs_test, preds_test), 3)}

print(f'performance metrics on the test set:\n  >> {metrics}')
```

```
## performance metrics on the test set:
##   >> {'accuracy': 0.838, 'f1_score': 0.889}
```

```
from sklearn.metrics import confusion_matrix

confusion_matrix(labs_test, preds_test)
```

```
## array([[ 38,  28],
##        [  4, 128]])
```

out-of-sample prediction (step 5)

(here you'd begin generating predictions on data points for which you don't have human labels.)

demo 2: classifying movie script lines by "utterance type" (multi-class)

the problem

You have a dataset of (transcriptions of) lines from popular movies, and you want to train a model to detect their "utterance types" – i.e. for each line, you want to classify it as one of the following:

- `D` – declarative sentence (i.e. a statement)
- `Q` – interrogative sentence (i.e. a question)
- `C` – imperative sentence (i.e. a command)
- `O` – anything else (e.g. an interjection, a greeting)

```
readLines("../data/cmdc_lines-annotated-551.txt", n=8)
```

```
## [1] "label|text"                        "o|Uh-huh."
## [3] "q|Who's this priest I'm thanking?" "d|That's right."
## [5] "o|Nice to meet you, Ma'am."        "d|Read a lot about you, Nick."
## [7] "o|I... submit to Our Lord."        "d|It's not a Nelwyn baby."
```

associated demo files

Sample dataset of 551 annotated movie lines:

- `../data/cmdc_lines-annotated-551.txt`

Classification workflow (very similar to tweets example):

- `../code/classify_utterance_type.py`

Summary file with model config and performance for a few fits:

- `../output/tweet_samples-model_info.txt`

load and split data into train/test subsets (step 0)

```
import pandas as pd

lines_infile = '../data/cmdc_lines-annotated-551.txt'
dat = pd.read_csv(lines_infile, sep='|', encoding='utf-8')

docs, labs = dat['text'].tolist(), dat['label'].tolist()

# some sample data points
dat[['text','label']].head(4)
```

```
##                                text label
## 0                           Uh-huh.     o
## 1  Who's this priest I'm thanking?     q
## 2                      That's right.     d
## 3          Nice to meet you, Ma'am.     o
```

load and split data into train/test subsets (step 0)

```
from sklearn.model_selection import train_test_split

docs_train, docs_test, labs_train, labs_test = train_test_split(
  docs, labs, test_size=.33, random_state=6933)
```

text prep and feature engineering (pipeline steps 1-2)

`scikit-learn` combines these two steps in seamless fashion:

```
import re

def prep_text(doc, toss_re=r'[?.,!$-]'):
  '''simple text preprocessing routine (remove some punctuation)'''
  return re.sub(toss_re, '', doc)
```

text prep and feature engineering (steps 1-2, con't)

```
from sklearn.feature_extraction.text import CountVectorizer

# text preprocessing and feature extraction choices encoded here
kwargs = {'analyzer': 'char', 'ngram_range': (1, 3),
          'lowercase': False, 'preprocessor': None}

vectorizer = CountVectorizer(token_pattern=r'\b[a-zA-Z0-9_<>]{1,}\b', **kwargs)

# use the training set vocabulary for feature construction
vectorizer.fit(docs_train)
```

```
# create separate doc-term matrices for train and test subsets
X_train = vectorizer.transform(docs_train)
X_test = vectorizer.transform(docs_test)
```

model training (step 3)

(leaving out parameter tuning!)

```
from sklearn.naive_bayes import MultinomialNB

# fit/train classifier using train features and labels
classifier = MultinomialNB()
classifier.fit(X_train, labs_train)
```

```
# generate test set model predictions from test matrix
preds_test = classifier.predict(X_test)
```

evaluation of model performance (step 4)

```
from sklearn.metrics import accuracy_score, f1_score

metrics = {'accuracy': round(accuracy_score(labs_test, preds_test), 3)}

print(f'performance metrics on the test set:\n  >> {metrics}')
```

```
## performance metrics on the test set:
##   >> {'accuracy': 0.716}
```

```
from sklearn.metrics import confusion_matrix

confusion_matrix(labs_test, preds_test)
```

```
## array([[ 3, 15,  1,  1],
##        [ 1, 70,  0,  5],
##        [ 2, 16,  4,  4],
##        [ 0,  7,  0, 54]])
```

questions? comments?? etc.???

thanks!

drop me a line if you ever wanna chat about NLP, linguistics, computing on text, etc. <3

`leffel-timothy@norc.org`