W4995 Applied Machine Learning

Working with Text Data

04/03/17

Andreas Müller

More kinds of data

- So far:
  - Fixed number of features
  - Continuous
  - Categorical
- Next up:
  - No pre-defined features
  - Free text
  - Images
  - (Audio: not this class)
  - Need to create fixed-length description

Typical Text data

May Contain Spoilers

A dude in a dopey-looking Kong suit (the same one used in KING KONG VS. GODZILLA in 1962) provides much of the laffs in this much-mocked monster flick. Kong is resurrected on Mondo Island and helps out the lunkhead hero and other good guys this time around. The vampire-like villain is named Dr. Who--funny, he doesn't look like Peter Cushing! Kong finally dukes it out with Who's pride and joy, a giant robot ape that looks like a bad metal sculpture of Magilla Gorilla. Like many of Honda's flicks this may have had some merit before American audiences diddled around with it and added new footage. The Rankin/Bass animation company had a hand in this mess. They should have stuck to superior children's programs like The Little Drummer Boy.

... than this ;-) What would happen if Terry Gilliam and Douglas Adams would have worked together on one movie? This movie starts with a touch of Brazil... when, at a certain point, the story moves straight into the twilight zone... bringing up nothing new, but just nothing... and nothing is great fun! When Dave and Andrew starts to explore their new environment the movie gets really enjoyable... bouncing heads? well... yes ;-)

anyway... this movie was, imho, the biggest surprise at this year's FantasyFilmFest...

Just like in Cube and Cypher Natali gave this one a minimalistic, weird but very special design, which makes it hard to locate the place of the story or its time... timeless somehow...

Other Types of text data

Free string – but not "words"

	country	fullName	Id	nationalPoliticalGroup	politicalGroup
0	Sweden	Lars ADAKTUSSON	124990	Kristdemokraterna	Group of the European People's Party (Christia...
1	Italy	Isabella ADINOLFI	124831	Movimento 5 Stelle	Europe of Freedom and Direct Democracy Group
2	Italy	Marco AFFRONTE	124797	Movimento 5 Stelle	Group of the Greens/European Free Alliance
3	Italy	Laura AGEA	124811	Movimento 5 Stelle	Europe of Freedom and Direct Democracy Group
4	United Kingdom	John Stuart AGNEW	96897	United Kingdom Independence Party	Europe of Freedom and Direct Democracy Group

Named Entities

Categorical (If you're lucky)

Features from Text: Bag of Words

“This is how you get ants.”

↓ tokenizer

['this','is','how','you','get', 'ants']

↓ Build a vocabulary over all document

['aardvak','amsterdam','ants', ...'you','your', 'zyxst']

↓ Sparse matrix encoding

aardvak ants get you zyxst

[0, …, 0, 1, 0, … , 0, 1 , 0, …, 0, 1, 0, …., 0]

Toy Example

```
mallory = ["Do you want ants?",
           "Because that’s how you get ants."]
```

Two documents in datasets

```
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(mallory)
print(vect.get_feature_names())
```

```
['ants', 'because', 'do', 'get', 'how', 'that', 'want', 'you']
```

```
X = vect.transform(mallory)
X
```

```
<2x8 sparse matrix of type '<class 'numpy.int64'>'
	with 10 stored elements in Compressed Sparse Row format>
```

```
X.toarray()
```

```
array([[1, 0, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 1, 1, 1, 0, 1]])
```

“bag”

```
print(mallory)
print(vect.inverse_transform(X)[0])
print(vect.inverse_transform(X)[1])
```

```
['Do you want ants?', 'Because that’s how you get ants.']
['ants' 'do' 'want' 'you']
['ants' 'because' 'get' 'how' 'that' 'you']
```

Text classification example: IMDB Movie Reviews

Data loading

```
from sklearn.datasets import load_files
reviews_train = load_files("../data/aclImdb/train/")

text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
print("class balance: {}".format(np.bincount(y_train)))
print("text_train[1]:\n{}".format(text_train[1]))
```

```
type of text_train: <class 'list'>
length of text_train: 25000
class balance: [12500 12500]
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing only. You have too see it for your
self to get at grip of how horrible a movie really can be. Not that I recommend you to do that. There are so man
y clich\xc3\xa9s, mistakes (and all other negative things you can imagine) here that will just make you cry. To
start with the technical first, there are a LOT of mistakes regarding the airplane. I won\'t list them here, but
just mention the coloring of the plane. They didn\'t even manage to show an airliner in the colors of a fictiona
l airline, but instead used a 747 painted in the original Boeing livery. Very bad. The plot is stupid and has be
en done many times before, only much, much better. There are so many ridiculous moments here that i lost count o
f it really early. Also, I was on the bad guys\' side all the time in the movie, because the good guys were so s
tupid. "Executive Decision" should without a doubt be you\'re choice over this one, even the "Turbulence"-movies
are better. In fact, every other movie in the world is better than this one.'
```

Vectorization

```
text_train_sub, text_val, y_train_sub, y_val = train_test_split(
    text_train, y_train, stratify=y_train, random_state=0)
vect = CountVectorizer()
X_train = vect.fit_transform(text_train_sub)
X_val = vect.transform(text_val)
```

```
X_train
```

```
<18750x66651 sparse matrix of type '<class 'numpy.int64'>'
	with 2580448 stored elements in Compressed Sparse Row format>
```

```
feature_names = vect.get_feature_names()
print(feature_names[:10])
print(feature_names[20000:20020])
print(feature_names[::2000])
```

```
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '0038
30', '006', '007']
['eschews', 'escort', 'escorted', 'escorting', 'escorts', 'escpeciall
y', 'escreve', 'escrow', 'esculator', 'ese', 'eser', 'esha', 'eshaan'
, 'eshley', 'esk', 'eskimo', 'eskimos', 'esmerelda', 'esmond', 'esoph
agus']
['00', 'ahoy', 'aspects', 'belting', 'bridegroom', 'cements', 'commas
', 'crowds', 'detlef', 'druids', 'eschews', 'finishing', 'gathering',
'gunrunner', 'homesickness', 'inhumanities', 'kabbalism', 'leech', 'm
akes', 'miki', 'nas', 'organ', 'pesci', 'principally', 'rebours', 'ro
botnik', 'sculptural', 'skinkons', 'stardom', 'syncer', 'tools', 'unf
lagging', 'waaaay', 'yanks']
```

Once we have X, business as usual

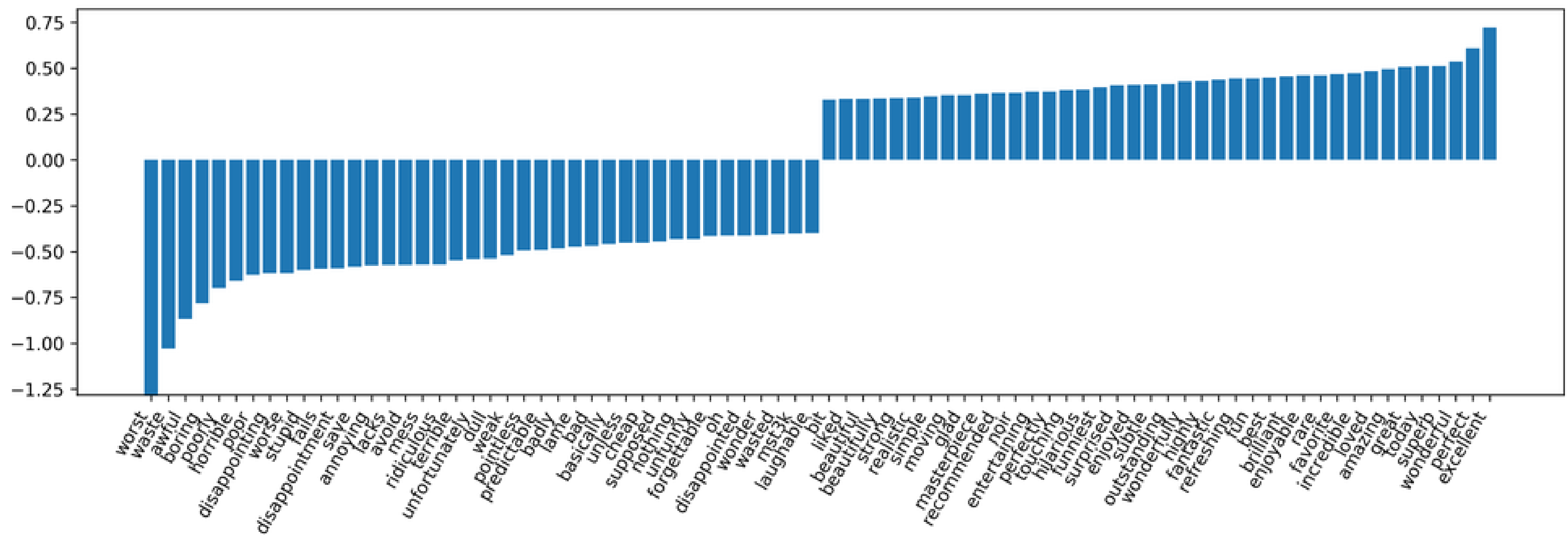
```
from sklearn.linear_model import LogisticRegressionCV
lr = LogisticRegressionCV().fit(X_train, y_train_sub)
```

```
lr.C_
```

```
array([ 0.046])
```

```
lr.score(X_val, y_val)
```

```
0.88192000000000004
```



0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
worst
waste
awful
boring
poorly
horrible
poor
disappointing
worse
stupid
fails
disappointment
save
annoying
lacks
avoid
mess
ridiculous
terrible
unfortunately
dull
weak
pointless
predictable
badly
lame
bad
basically
unless
cheap
supposed
nothing
unfunny
forgettable
oh
disappointed
wonder
wasted
mst3k
laughable
bit
liked
beautiful
beautifully
strong
realistic
simple
moving
glad
masterpiece
recommended
noir
entertaining
perfectly
touching
hilarious
funniest
surprised
enjoyed
subtle
outstanding
wonderfully
highly
fantastic
refreshing
fun
best
brilliant
enjoyable
rare
favorite
incredible
loved
amazing
great
today
superb
wonderful
perfect
excellent

Soo many options!

- How to tokenize?
- How to normalize words?
- What to include in vocabulary?

Tokenization

- Scikit-learn (very simplistic):
  - re.findall(r"\b\w\w+\b")
  - Includes numbers
  - doesn't include single-letter words
  - doesn't include – or '
- Can change regular expression "token pattern":

```
vect = CountVectorizer(token_pattern=r"\b\w+\b")
vect.fit(mallory)
print(vect.get_feature_names())
```

```
['ants', 'because', 'do', 'get', 'how', 's', 'that', 'want', 'you']
```

```
vect = CountVectorizer(token_pattern=r"\b\w[\w’]+\b")
# not actually an apostroph but some unicode pattern
# because I copy & pasted the quote
vect.fit(mallory)
print(vect.get_feature_names())
```

```
['ants', 'because', 'do', 'get', 'how', 'that’s', 'want', 'you']
```

Normalization

- Correct spelling?
- Stemming: reduce to word stem

- Lemmatization: reduce words to stem using curated dictionary and context

- Scikit-learn:

  Lower-case it.

  Configurable, use nltk or spacy

Restricting the Vocabulary

Stop Words

```
vect = CountVectorizer(stop_words='english')
vect.fit(mallory)
print(vect.get_feature_names())
```

```
['ants', 'want']
```

Also: max_df

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print(list(ENGLISH_STOP_WORDS))
```

```
['there', 'else', 'two', 'perhaps', 'get', 'inc', 'find', 'interest', 'between', 'give', 'amongst', 'however', 'forme
r', 'nine', 'please', 'us', 'about', 'almost', 'but', 'thereupon', 'call', 'ie', 'third', 'whereby', 'whole', 'whose'
, 'one', 'afterwards', 'only', 'somehow', 'is', 'eight', 'nothing', 'an', 'with', 'describe', 'than', 'itself', 'do',
'thin', 'cry', 'hundred', 'its', 'latterly', 'formerly', 'name', 'no', 'via', 'hereupon', 'well', 'system', 'so', 'un
', 'mill', 'neither', 'she', 'seems', 'or', 'though', 'against', 'wherever', 'very', 'within', 'con', 'during', 'whom
', 'per', 'front', 'much', 'sometimes', 'ten', 'next', 'those', 'anyhow', 'fill', 'became', 'along', 'never', 'this',
'that', 'our', 'all', 'be', 'may', 'made', 'should', 'for', 'keep', 'onto', 'below', 'here', 'been', 'of', 'once', 't
hemselves', 'whereas', 'three', 'hereby', 'several', 'how', 'even', 'whither', 'her', 'herself', 'other', 'will', 'ar
ound', 'a', 'seem', 'because', 'it', 'across', 'take', 'enough', 'to', 'under', 'what', 'again', 'less', 'through', '
amoungst', 'the', 'more', 'my', 'either', 'see', 'sometime', 'detail', 'thereafter', 'anyone', 'except', 'co', 'from'
, 'now', 'own', 'de', 'them', 'anywhere', 'hasnt', 'nobody', 'few', 'and', 'hence', 'alone', 'when', 'each', 'another
', 'always', 'anything', 'yet', 'four', 'therefore', 'thick', 'cant', 'since', 'can', 'twelve', 'forty', 'among', 'ov
er', 'where', 'your', 'cannot', 'on', 'becomes', 'sixty', 'whether', 'become', 'such', 'we', 'some', 'in', 'thus', 'u
pon', 'their', 'fifteen', 'they', 'could', 'mostly', 'was', 'although', 'serious', 'first', 'not', 'thence', 'thru',
'whenever', 'rather', 'before', 'moreover', 'noone', 'put', 'up', 'who', 'were', 'anyway', 'namely', 'beyond', 'latte
r', 'everyone', 'toward', 'seeming', 'whereupon', 'yourselves', 'move', 'why', 'part', 'same', 'without', 'every', 'b
ill', 'might', 'most', 'fifty', 'hereafter', 'show', 'have', 'herein', 'beforehand', 'off', 'which', 'indeed', 'many'
, 'whereafter', 'none', 'after', 'ours', 'down', 'couldnt', 'bottom', 'go', 'due', 'sincere', 'himself', 'done', 'bac
k', 'am', 'hers', 'etc', 'last', 'out', 'six', 'nor', 'until', 'meanwhile', 'nowhere', 'twenty', 'whoever', 'must', '
full', 'whatever', 'you', 'both', 'top', 'also', 'being', 'into', 'these', 'by', 'nevertheless', 'least', 'besides',
'as', 'would', 'still', 'amount', 'behind', 'side', 'has', 'any', 'ever', 'ltd', 'together', 'ourselves', 'mine', 'wh
erein', 'above', 'somewhere', 'beside', 'thereby', 'had', 'i', 'myself', 'otherwise', 'whence', 'at', 'elsewhere', 'i
f', 'further', 'he', 'his', 'eleven', 'him', 'are', 'then', 'while', 'eg', 'often', 'already', 'too', 'yourself', 'so
meone', 'fire', 'found', 'others', 'me', 're', 'everything', 'something', 'therein', 'throughout', 'becoming', 'every
where', 'yours', 'towards', 'empty', 'seemed', 'five']
```

For supervised learning often little effect on large corpuses (on small corpuses and for unsupervised learning it can help)

Infrequent Words

- Words that appear only once or twice might not be helpful:

```
vect = CountVectorizer(min_df=2)
vect.fit(mallory)
print(vect.get_feature_names())
```

```
['ants', 'you']
```

- Restrict vocabulary size to only most frequent words (for less features):

```
vect = CountVectorizer(max_features=4)
vect.fit(mallory)
print(vect.get_feature_names())
```

```
['ants', 'because', 'do', 'you']
```

```
vect = CountVectorizer(min_df=2)
X_train_df2 = vect.fit_transform(text_train_sub)
X_val_df2 = vect.transform(text_val)
print(X_train.shape)
print(X_train_df2.shape)
```

```
(18750, 66651)
(18750, 39825)
```

Removed nearly 1/3 of features!

```
vect = CountVectorizer(min_df=4)
X_train_df4 = vect.fit_transform(text_train_sub)
X_val_df4 = vect.transform(text_val)
print(X_train.shape)
print(X_train_df2.shape)
print(X_train_df4.shape)
```

```
(18750, 66651)
(18750, 39825)
(18750, 26928)
```

```
lr = LogisticRegressionCV().fit(X_train_df4, y_train_sub)
```

```
lr.C_
```

```
array([ 0.046])
```

```
lr.score(X_val_df4, y_val)
```

```
0.88095999999999997
```

As good as before

Beyond single words

- Bag of words completely removes word order.
- "didn't love" and "love" are very different!
- N-grams: tuples of consecutive words

```
"This is how you get ants."
```

Unigram tokenizer

```
['this', 'is', 'how', 'you', 'get', 'ants']
```

```
"This is how you get ants."
```

Bigram tokenizer

```
['this is', 'is how', 'how you', 'you get', 'get ants']
```

Bigrams toy example

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(mallory)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

```
Vocabulary size: 8
Vocabulary:
['ants', 'because', 'do', 'get', 'how', 'that', 'want', 'you']
```

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(mallory)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

```
Vocabulary size: 8
Vocabulary:
['because that', 'do you', 'get ants', 'how you', 'that how', 'want ants', 'you get', 'you want']
```

```
cv = CountVectorizer(ngram_range=(1, 2)).fit(mallory)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

```
Vocabulary size: 16
Vocabulary:
['ants', 'because', 'because that', 'do', 'do you', 'get', 'get ants', 'how', 'how you', 'that',
'that how', 'want', 'want ants', 'you', 'you get', 'you want']
```

Typically: higher n-grams lead to blow up of feature space!

N-grams on IMDB data

```
Vocabulary size 1-gram (min_df=4): 26928
Vocabulary size 2-gram (min_df=4): 128426
Vocabulary size 1gram, 2gram (min_df=4): 155354
Vocabulary size 1-3gram (min_df=4): 254274
Vocabulary size 1-4gram, 2gram (min_df=4): 289443
```

```
cv = CountVectorizer(ngram_range=(1, 4)).fit(text_train_sub)
print("Vocabulary size 1-4gram: {}".format(len(cv.vocabulary_)))
```

```
Vocabulary size 1-4gram: 7815528
```

More than 20x more 4-grams!

Stop-word impact on bi-grams

```
cv = CountVectorizer(ngram_range=(1, 2), min_df=4).fit(text_train_sub)
print("Vocabulary size (1, 2), min_df=4: {}".format(len(cv.vocabulary_)))
cv = CountVectorizer(ngram_range=(1, 2), min_df=4, stop_words="english").fit(text_train_sub)
print("Vocabulary size (1, 2), stopwords, min_df=4: {}".format(len(cv.vocabulary_)))
```

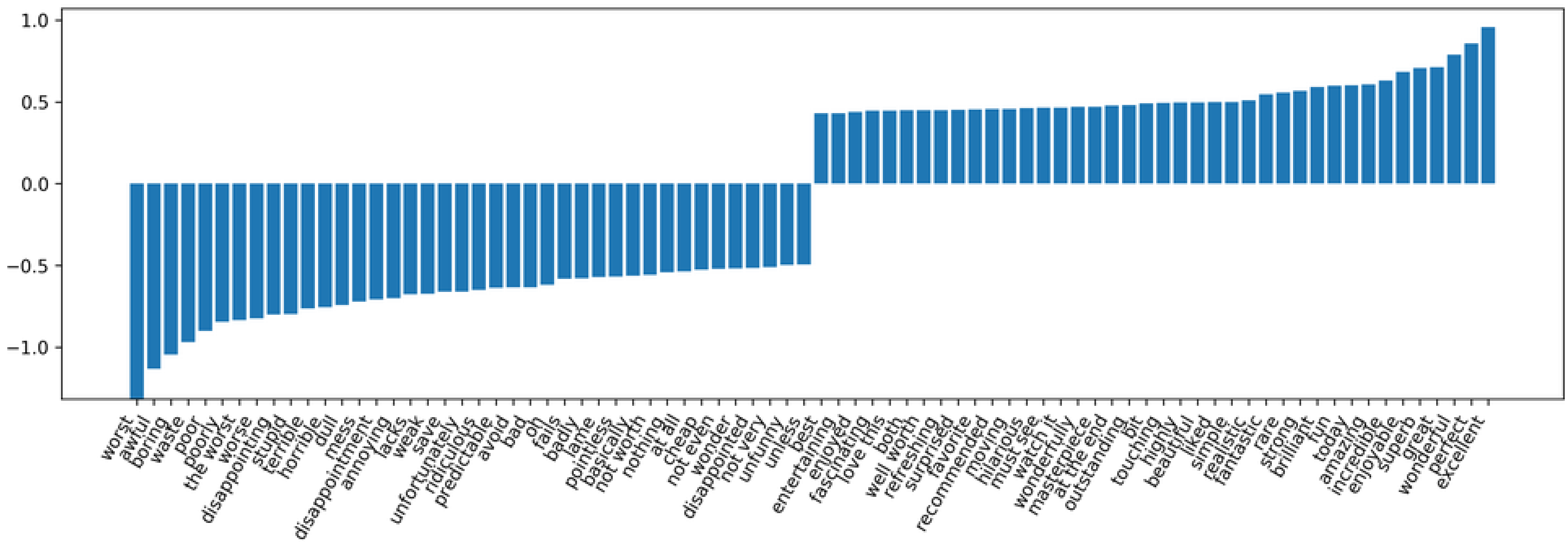
```
Vocabulary size (1, 2), min_df=4: 155354
Vocabulary size (1, 2), stopwords, min_df=4: 81085
```

Stop words impact 4-gram

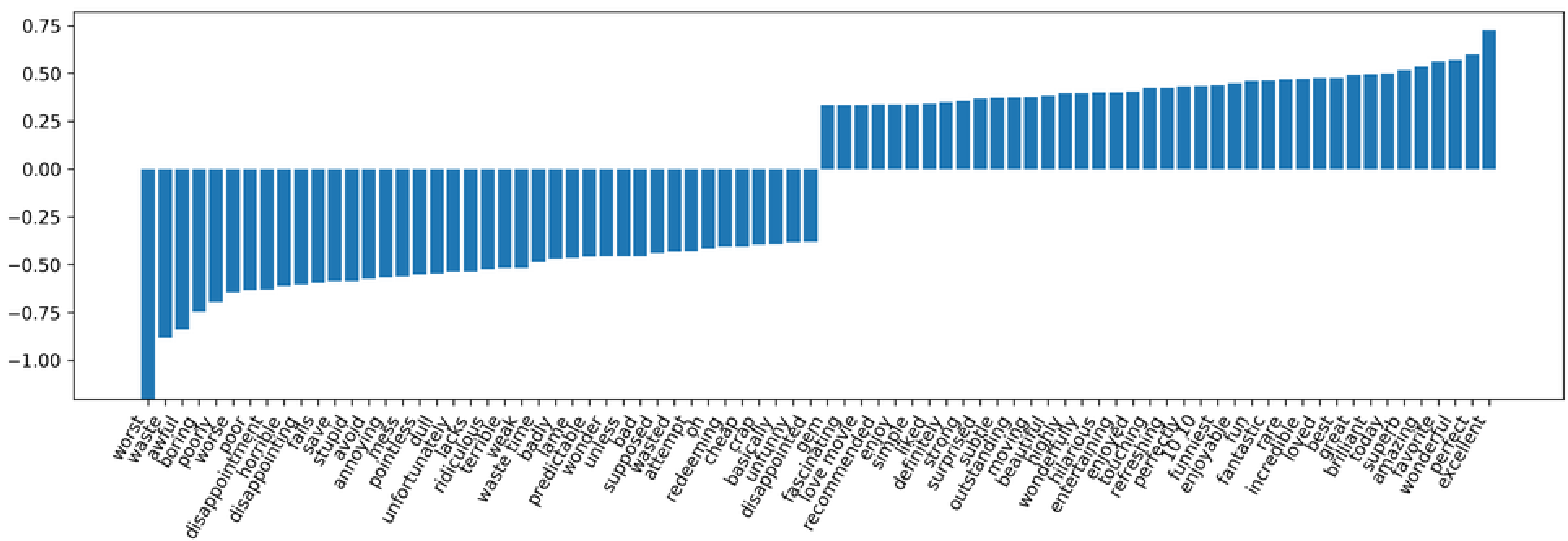
```
cv4 = CountVectorizer(ngram_range=(4, 4), min_df=4).fit(text_train_sub)
cv4sw = CountVectorizer(ngram_range=(4, 4), min_df=4, stop_words="english").fit(text_train_sub)
print(len(cv4.get_feature_names()))
print(len(cv4sw.get_feature_names()))
```

```
31585
369
```

Stopwords included



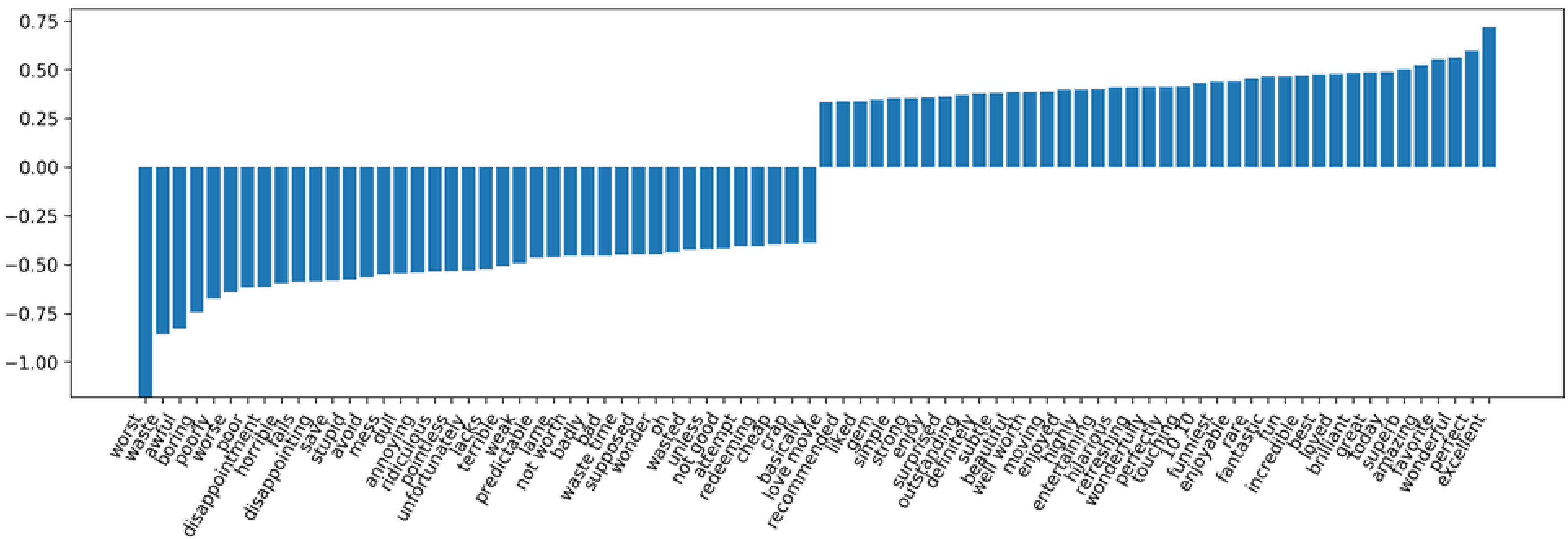
Stopwords removed (slightly worse)



```python
my_stopwords = set(ENGLISH_STOP_WORDS)
my_stopwords.remove("well")
my_stopwords.remove("not")
my_stopwords.add("ve")
```

```python
vect3msw = CountVectorizer(ngram_range=(1, 3), min_df=4, stop_words=my_stopwords)
X_train3msw = vect3msw.fit_transform(text_train_sub)
lr3msw = LogisticRegressionCV().fit(X_train3msw, y_train_sub)
X_val3msw = vect3msw.transform(text_val)
lr3msw.score(X_val3msw, y_val)
```

```
0.8831999999999999
```



Tf-idf rescaling

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

Total number of documents

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

Number of documents containing term t

Emphasizes “rare” words - “soft stop word removal”

Slightly non-standard smoothing (many +1s)

By default also L2 normalization!

TfidfVectorizer, TfidfTransformer

```
from sklearn.feature_extraction.text import TfidfVectorizer, TfidfTransformer
```

```
malory_tfidf = TfidfVectorizer().fit_transform(malory)
malory_tfidf.toarray()
```

```
array([[ 0.41 ,  0.   ,  0.576,  0.   ,  0.   ,  0.   ,  0.576,  0.41 ],
       [ 0.318,  0.447,  0.   ,  0.447,  0.447,  0.447,  0.   ,  0.318]])
```

```
malory_tfidf = make_pipeline(CountVectorizer(), TfidfTransformer()).fit_transform(malory)
malory_tfidf.toarray()
```

```
array([[ 0.41 ,  0.   ,  0.576,  0.   ,  0.   ,  0.   ,  0.576,  0.41 ],
       [ 0.318,  0.447,  0.   ,  0.447,  0.447,  0.447,  0.   ,  0.318]])
```

Character n-grams

Principle

Do␣you␣want␣ants?

Applications

- Be robust to misspelling / obfuscation
- Language detection
- Learn from Names / made-up words

Toy Example

“Naive”

```
cv = CountVectorizer(ngram_range=(2, 3), analyzer="char").fit(malory)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

```
Vocabulary size: 73
Vocabulary:
[' a', ' an', ' g', ' ge', ' h', ' ho', ' t', ' th', ' w', ' wa', ' y', ' yo', 'a
n', 'ant', 'at', 'at’', 'au', 'aus', 'be', 'bec', 'ca', 'cau', 'do', 'do ', 'e ',
'e t', 'ec', 'eca', 'et', 'et ', 'ge', 'get', 'ha', 'hat', 'ho', 'how', 'nt', 'nt
', 'nts', 'o ', 'o y', 'ou', 'ou ', 'ow', 'ow ', 's ', 's h', 's.', 's?', 'se', '
se ', 't ', 't a', 'th', 'tha', 'ts', 'ts.', 'ts?', 't’', 't’s', 'u ', 'u g', 'u
w', 'us', 'use', 'w ', 'w y', 'wa', 'wan', 'yo', 'you', '’s', '’s ']
```

Respect word boundaries:

```
cv = CountVectorizer(ngram_range=(2, 3), analyzer="char_wb").fit(malory)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

```
Vocabulary size: 74
Vocabulary:
[' a', ' an', ' b', ' be', ' d', ' do', ' g', ' ge', ' h', ' ho', ' t', ' th', '
w', ' wa', ' y', ' yo', '. ', '? ', 'an', 'ant', 'at', 'at’', 'au', 'aus', 'be',
'bec', 'ca', 'cau', 'do', 'do ', 'e ', 'ec', 'eca', 'et', 'et ', 'ge', 'get', 'ha
', 'hat', 'ho', 'how', 'nt', 'nt ', 'nts', 'o ', 'ou', 'ou ', 'ow', 'ow ', 's ',
's.', 's. ', 's?', 's? ', 'se', 'se ', 't ', 'th', 'tha', 'ts', 'ts.', 'ts?', 't’
', 't’s', 'u ', 'us', 'use', 'w ', 'wa', 'wan', 'yo', 'you', '’s', '’s ']
```

IMDB Data

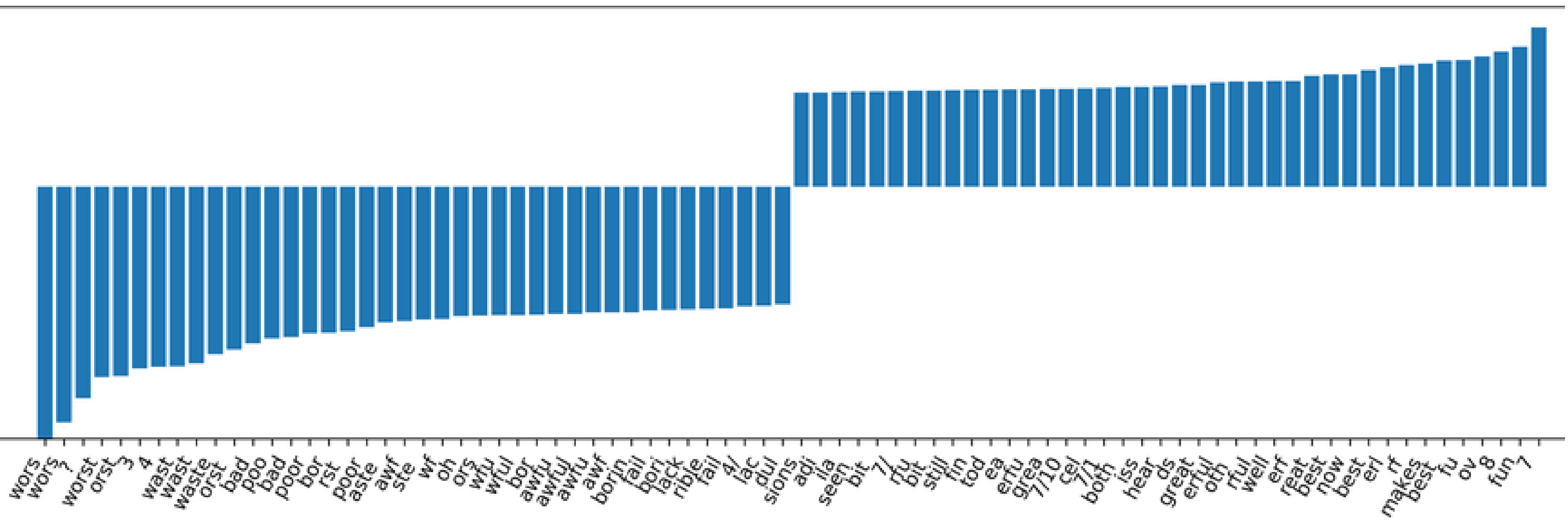
```
char_vect = CountVectorizer(ngram_range=(2, 5), min_df=4, analyzer="char_wb")
X_train_char = char_vect.fit_transform(text_train_sub)
```

```
len(char_vect.vocabulary_)
```

```
164632
```

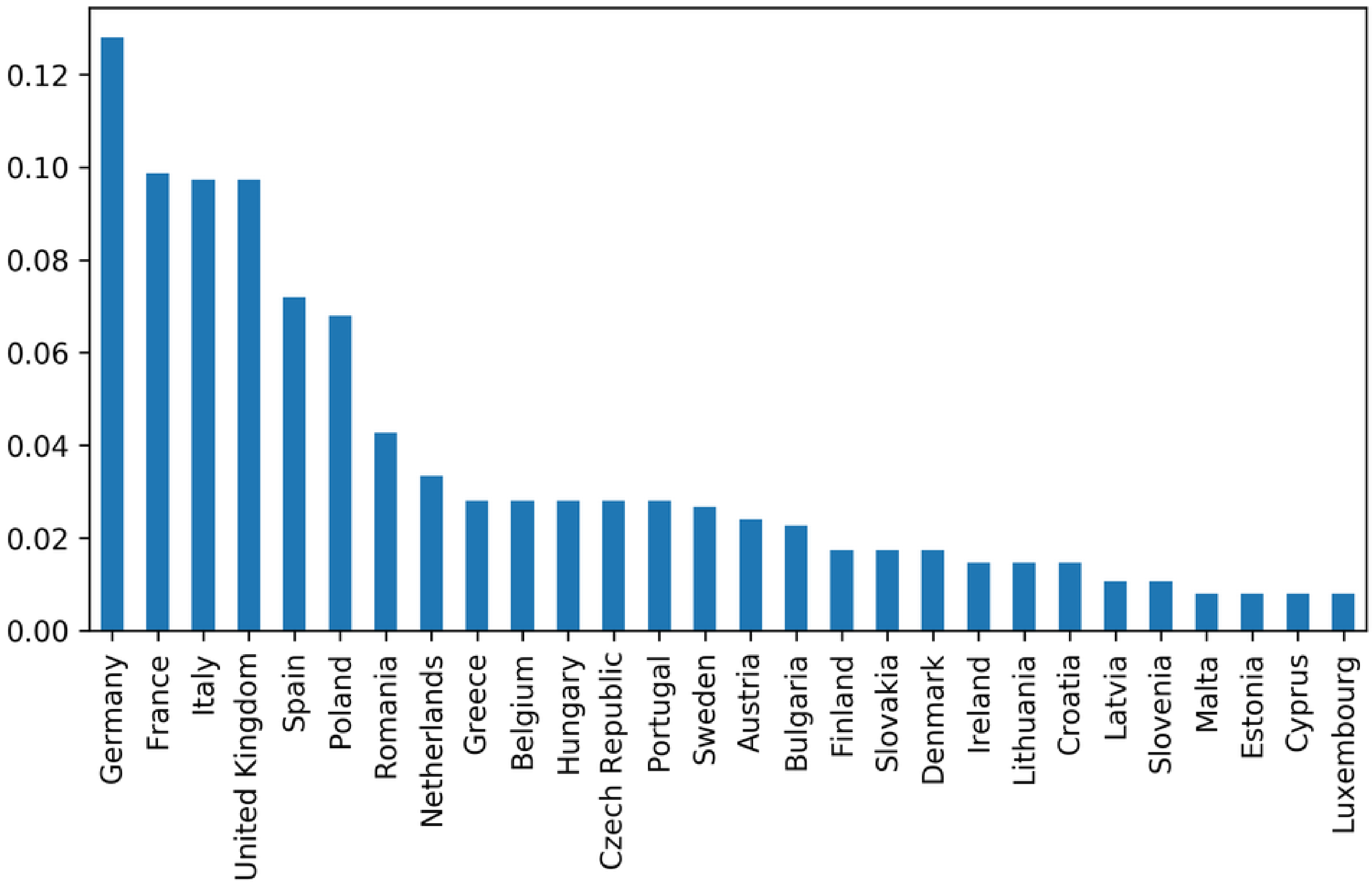
```
lr_char = LogisticRegressionCV().fit(X_train_char, y_train_sub)
X_val_char = char_vect.transform(text_val)
lr_char.score(X_val_char, y_val)
```

```
0.88112000000000001
```


wors
wors
?
worst
orst
3
4
wast
wast
waste
orst
bad
poo
bad
poor
bor
rst
poor
aste
awf
ste
wf
oh
ors
wfu
wful
bor
awfu
awful
awfu
awf
borin
fail
bori
lack
rible
fail
4/
lac
dul
sions
adi
ila
seen
bit
7/
rfu
bit
still
fin
tod
ea
erfu
grea
7/10
cel
7/1
both
iss
hear
ds
great
erful
oth
rful
well
erf
reat
best
now
best
eri
rf
makes
best
fu
ov
8
fun
7

Predicting Nationality from Name

	country	fullName	id	nationalPoliticalGroup	politicalGroup
0	Sweden	Lars ADAKTUSSON	124990	Kristdemokraterna	Group of the European People's Party (Christia...
1	Italy	Isabella ADINOLFI	124831	Movimento 5 Stelle	Europe of Freedom and Direct Democracy Group
2	Italy	Marco AFFRONTE	124797	Movimento 5 Stelle	Group of the Greens/European Free Alliance
3	Italy	Laura AGEA	124811	Movimento 5 Stelle	Europe of Freedom and Direct Democracy Group
4	United Kingdom	John Stuart AGNEW	96897	United Kingdom Independence Party	Europe of Freedom and Direct Democracy Group



Comparing words vs chars

```
bow_pipe = make_pipeline(CountVectorizer(), LogisticRegressionCV())
cross_val_score(bow_pipe, text_mem_train, y_mem_train, cv=5, scoring='f1_macro')
```

```
array([ 0.231,  0.241,  0.236,  0.28 ,  0.254])
```

```
char_pipe = make_pipeline(CountVectorizer(analyzer="char_wb"), LogisticRegressionCV())
cross_val_score(char_pipe, text_mem_train, y_mem_train, cv=5, scoring='f1_macro')
```

```
array([ 0.452,  0.459,  0.341,  0.469,  0.418])
```

Grid-search parameters

Small dataset, makes grid-search faster! (less reliable)

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import Normalizer

param_grid = {"logisticregression__C": [100, 10, 1, 0.1, 0.001],
              "countvectorizer__ngram_range": [(1, 1), (1, 2), (1, 5), (1, 7),
                                               (2, 3), (2, 5), (3, 8), (5, 5)],
              "countvectorizer__min_df": [1, 2, 3],
              "normalizer": [None, Normalizer()]
             }
grid = GridSearchCV(make_pipeline(CountVectorizer(analyzer="char"), Normalizer(), LogisticRegression()),
                    param_grid=param_grid, cv=10, scoring="f1_macro"
                   )
```

```
grid.fit(text_mem_train, y_mem_train)
```

```
grid.best_score_
```

```
0.58255198397046815
```

```
grid.best_params_
```

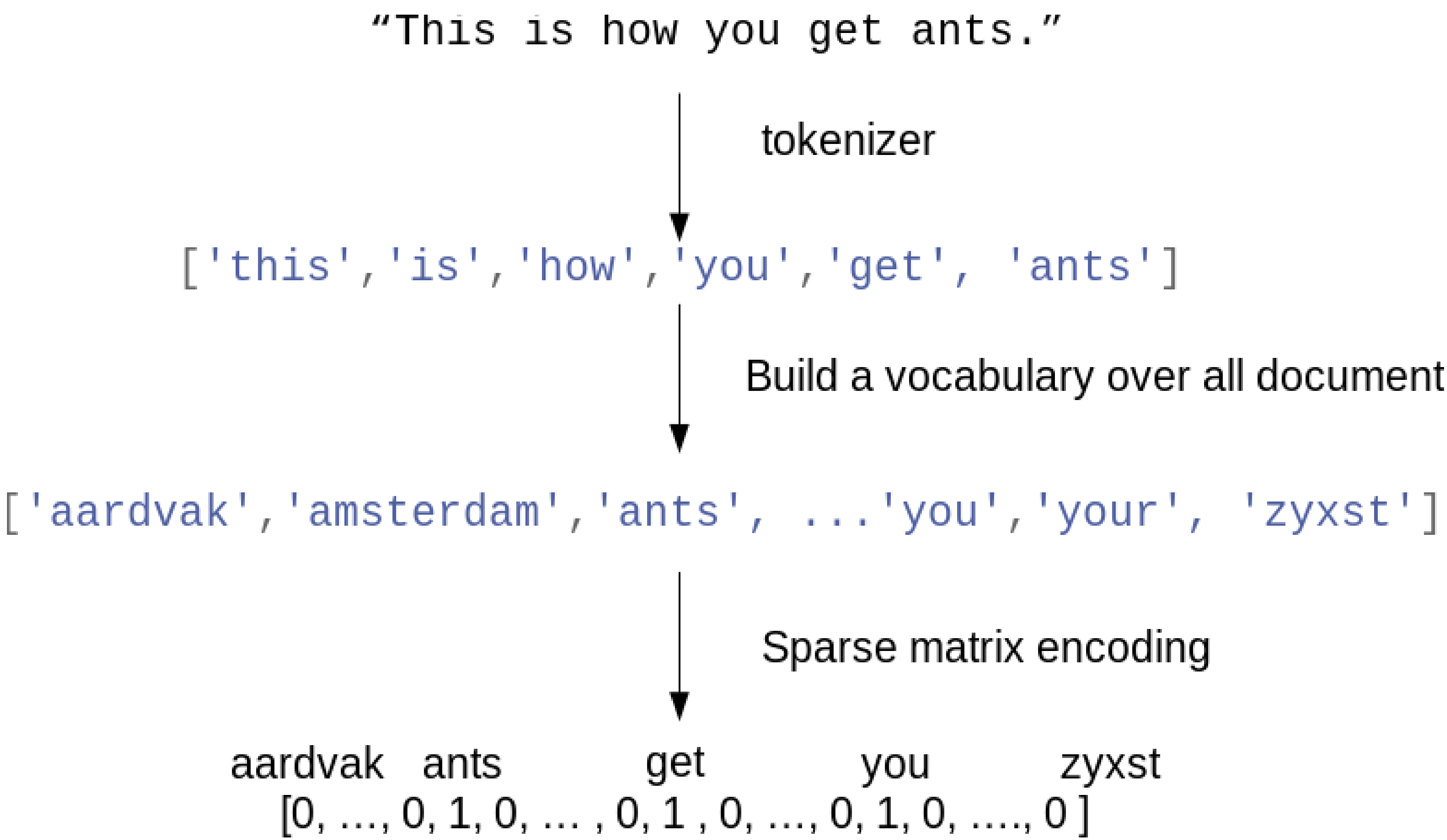
```
{'countvectorizer__min_df': 2,
 'countvectorizer__ngram_range': (1, 5),
 'logisticregression__C': 10}
```

	param_countvectorizer__ngram_range	(1, 1)	(1, 2)	(1, 5)	(1, 7)	(2, 3)	(2, 5)	(3, 8)	(5, 5)
min_df	**C**								
1	**0.001**	0.141167	0.216887	0.266306	0.267246	0.325938	0.349475	0.281929	0.0653852
	0.1	0.44827	0.520445	0.551024	0.544649	0.475185	0.507482	0.428376	0.249614
	1.0	0.480549	0.545256	0.565362	0.554272	0.515928	0.517898	0.434622	0.333195
	10.0	0.499625	0.529781	0.575243	0.548367	0.495087	0.511727	0.432281	0.360981
	100.0	0.481605	0.515618	0.569864	0.547449	0.497854	0.505122	0.440256	0.383315
2	**0.001**	0.141167	0.211798	0.251195	0.253522	0.310884	0.341462	0.242935	0.0576071
	0.1	0.441997	0.523296	0.560686	0.552423	0.487937	0.500663	0.440686	0.184905
	1.0	0.482002	0.531615	0.573458	0.570961	0.50686	0.523805	0.455477	0.293757
	10.0	0.498945	0.534128	0.582552	0.574385	0.494141	0.522637	0.409354	0.279838
	100.0	0.469252	0.52665	0.581839	0.577626	0.488827	0.517176	0.427836	0.267407
3	**0.001**	0.141167	0.212785	0.24461	0.248863	0.309673	0.336444	0.214413	0.0576071
	0.1	0.437624	0.520559	0.564124	0.556022	0.497634	0.507008	0.430714	0.167934
	1.0	0.483502	0.534692	0.564548	0.559782	0.508479	0.526124	0.441994	0.232509
	10.0	0.499686	0.525823	0.577809	0.579871	0.497012	0.510214	0.432801	0.224545
	100.0	0.481043	0.512089	0.572186	0.574859	0.490168	0.491294	0.417196	0.224545

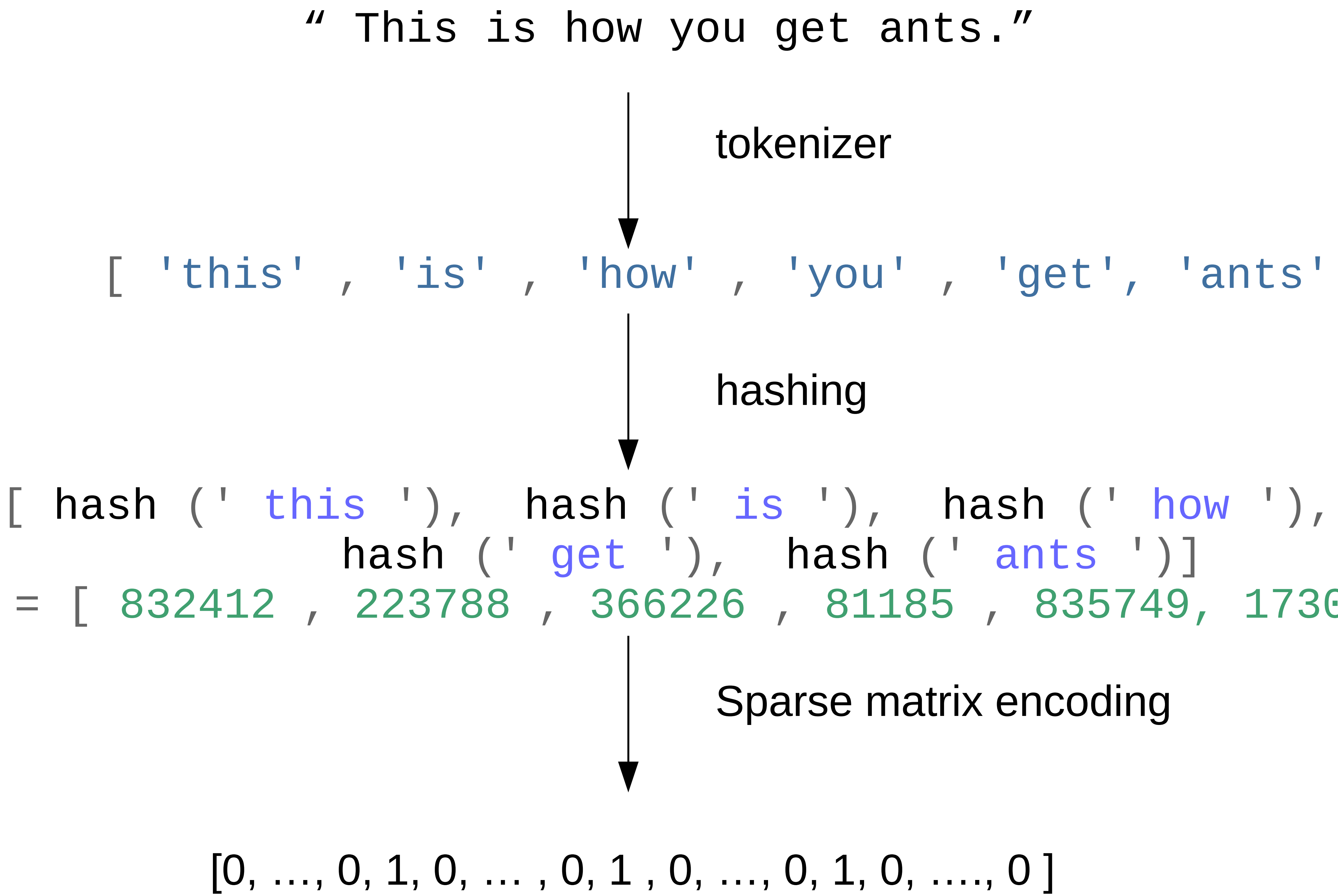
Other features

- Length of text
- Number of out-of-vocabularly words
- Presence / frequency of ALL CAPS
- Punctuation....!? (somewhat captured by char ngrams)
- Sentiment words (good vs bad)
- Whatever makes sense for the task!

Large Scale Text Vectorization



“This is how you get ants.”
tokenizer
['this','is','how','you','get', 'ants']
Build a vocabulary over all document
['aardvak','amsterdam','ants', ...'you','your', 'zyxst']
Sparse matrix encoding
aardvak ants get you zyxst
[0, …, 0, 1, 0, … , 0, 1 , 0, …, 0, 1, 0, …., 0]


“ This is how you get ants.”
tokenizer
['this' , 'is' , 'how' , 'you' , 'get', 'ants'
hashing
[hash (' this '), hash (' is '), hash (' how '),
hash (' get '), hash (' ants ')]
= [832412 , 223788 , 366226 , 81185 , 835749, 1730
Sparse matrix encoding
[0, …, 0, 1, 0, … , 0, 1 , 0, …, 0, 1, 0, …., 0]

Near drop-in replacement

- Careful: Uses l2 normalization by default!

```
from sklearn.feature_extraction.text import HashingVectorizer
hv = HashingVectorizer()
X_train = hv.transform(text_train_sub)
X_val = hv.transform(text_val)
lr = LogisticRegressionCV().fit(X_train, y_train_sub)
```

```
lr.score(X_val, y_val)
```

```
from sklearn.feature_extraction.text import HashingVectorizer
hv = HashingVectorizer()
X_train = hv.transform(text_train_sub)
X_val = hv.transform(text_val)
```

```
X_train.shape
```

```
(18750, 1048576)
```

```
lr = LogisticRegressionCV().fit(X_train, y_train_sub)
lr.score(X_val, y_val)
```

Trade-offs

Pro:

- Fast
- Works for streaming data
- Low memory footprint

Con:

- Can't interpret results
- Hard to debug
- (collisions are not a problem for performance)