From plain text to classification

Wednesday, June 13, 2018 11:13 PM

text preprocessing

We'll focus on text classification

Example: sentiment analysis

- Input: text of review
- Output: class of sentiment
  - e.g. 2 classes: positive vs negative
- Positive example:
  - The hotel is really beautiful. Very nice and helpful service at the front desk.
- Negative example:
  - We had problems to get the Wi-Fi working. The pool area was occupied with young party animals. So the area wasn't fun for us.

Text preprocessing

What is text?

You can think of text as a sequence of

- Characters
- **Words**
- Phrases and named entities
- Sentences
- Paragraphs
- …

What is a word?

It seems natural to think of a text as a sequence of words

- A word is a meaningful sequence of characters

How to find the boundaries of words?

- In English we can split a sentence by spaces or punctuation

Input: Friends, Romans, Countrymen, lend me your ears;

Output: Friends | Romans | Countrymen | lend | me | your | ears

- In German there are compound words which are written without spaces
  - "Rechtsschutzversicherungsgesellschaften" stands for "insurance companies which provide legal protection"

德语的奇葩词

- In Japanese there are no spaces at all!
  - Butyoucanstillreaditright?

text → meaningful chunk.

Tokenization

Tokenization is a process that splits an input sequence into so-called tokens

- You can think of a token as a useful unit for semantic processing
- Can be a word, sentence, paragraph, etc.

An example of simple whitespace tokenizer

- nltk.tokenize.WhitespaceTokenizer

This	is	Andrew's	text,	isn't	it?

- Problem: "it" and "it?" are different tokens with same meaning

Tokenization

Let's try to also split by punctuation

- nltk.tokenize.WordPunctTokenizer

This	is	Andrew	'	s	text	,	isn	'	t	it	?

- Problem: "s", "isn", "t" are not very meaningful

Rules

We can come up with a set of rules

- nltk.tokenize.TreebankWordTokenizer

This	is	Andrew	's	text	,	is	n't	it	?

- "'s" and "n't" are more meaningful for processing

NLTK tokenizer.

Python tokenization example

```
import nltk
text = "This is Andrew's text, isn't it?"
```

```
tokenizer = nltk.tokenize.WhitespaceTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', "Andrew's", 'text,', "isn't", 'it?']
```

```
tokenizer = nltk.tokenize.TreebankWordTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', 'Andrew', "'s", 'text', ',', 'is', "n't",
'it', '?']
```

```
tokenizer = nltk.tokenize.WordPunctTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', 'Andrew', "'", 's', 'text', ',', 'isn',
 "'", 't', 'it', '?']
```

http://text-processing.com/demo/tokenize/

ROOT stem.

Stemming example

Porter's stemmer → the oldest stemmer for English

- 5 heuristic phases of word reductions, applied sequentially
- Example of phase 1 rules:

Rule	Example
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress

Further normalization

Normalizing capital letters

can be tricky

- Us, us → us (if both are pronoun)
- us, US (could be pronoun and country)
- We can use heuristics:
  - lowercasing the beginning of the sentence
  - lowercasing words in titles
  - leave mid-sentence words as they are
- Or we can use machine learning to retrieve true casing → hard → than the original problem of sentiment analysis.

Acronyms

- eta, e.t.a., E.T.A. → E.T.A. estimated time of arrival. different forms. hard
- We can write a bunch of regular expressions → hard

Summary

- We can think of text as a sequence of tokens
- Tokenization is a process of extracting those tokens
- We can normalize tokens using stemming or lemmatization
- We can also normalize casing and acronyms
- In the next video we will transform extracted tokens into features for our model

feature extraction

Transforming tokens into features

Bag of words (BOW)

Let's count occurrences of a particular token in our text

- Motivation: we're looking for marker words like "excellent" or "disappointed"
- For each token we will have a feature column, this is called **text vectorization**.

	good	movie	not	a	did	like
good movie	**1**	**1**	0	0	0	0
not a good movie	**1**	**1**	**1**	**1**	0	0
did not like	0	0	**1**	0	**1**	**1**

- Problems:
  - we loose word order, hence the name "bag of words"
  - counters are not normalized

Let’s preserve some ordering

We can count token pairs, triplets, etc.

- Also known as n-grams
  - 1-grams for tokens
  - 2-grams for token pairs
  - …

good movie
not a good movie
did not like

good movie	**movie**	**did not**	**a**	**…**
1	**1**	0	0	…
1	**1**	0	**1**	…
0	0	**1**	0	…

- Problems:
  - too many features

Remove some n-grams

Let's remove some n-grams from features based on their occurrence frequency in documents of our corpus

Remove some n-grams

→ high & low freq. remove

Let's remove some n-grams from features based on their occurrence frequency in documents of our corpus

- **High frequency n-grams:**
  - Articles, prepositions, etc. (example: and, a, the) ← grammatical structures
  - They are called **stop-words**, they won't help us to discriminate texts → remove them
- **Low frequency n-grams:**
  - Typos, rare n-grams
  - We don't need them either, otherwise we will likely overfit
- **Medium frequency n-grams:**
  - Those are good n-grams

There're a lot of medium frequency n-grams

- It proved to be useful to look at n-gram frequency in our corpus for filtering out bad n-grams
- What if we use it for ranking of medium frequency n-grams?
- **Idea**: the n-gram with smaller frequency can be more discriminating because it can capture a specific issue in the review

TF-IDF

Term frequency (TF)

- $\mathrm{tf}(t, d)$ – frequency for term (or n-gram) t in document d
- Variants:

weighting scheme	TF weight
binary	$0, 1$
raw count	$f_{t,d}$
term frequency	$f_{t,d} / \sum_{t' \in d} f_{t',d}$
log normalization	$1 + \log(f_{t,d})$

https://en.wikipedia.org/wiki/Tf–idf

TF-IDF

Inverse document frequency (IDF)

- $N = |D|$ – total number of documents in corpus
- $|\{d \in D: t \in d\}|$ – number of documents where the term t appears
- $\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D: t \in d\}|}$ rare – big

at least 1. 所以不存在 0

$\geq 1 \Rightarrow$ positive

TF-IDF

乘积

- $\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$
- A high weight in TF-IDF is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents

high weight = high weight in one doc

low weight across all docs

Better BOW

- **Replace counters with TF-IDF**
- **Normalize the result row-wise (divide by L_2-norm)** => norm of 1.

good movie
not a good movie
did not like

good movie	movie	did not	…
0.17	**0.17**	0	…
0.17	**0.17**	0	…
0	0	**0.47**	…

Python TF-IDF example

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
texts = [
    "good movie", "not a good movie", "did not like",
    "i like it", "good one"
]
tfidf = TfidfVectorizer(min_df=2, max_df=0.5, ngram_range=(1, 2))
features = tfidf.fit_transform(texts)
pd.DataFrame(
    features.todense(),
    columns=tfidf.get_feature_names()
)
```

	good movie	like	movie	not
0	0.707107	0.000000	0.707107	0.000000
1	0.577350	0.000000	0.577350	0.577350
2	0.000000	0.707107	0.000000	0.707107
3	0.000000	1.000000	0.000000	0.000000
4	0.000000	0.000000	0.000000	0.000000

Summary

- We’ve made simple counter features in bag of words manner
- You can add n-grams
- You can replace counters with TF-IDF values
- In the next video we will train our first model on top of these features

linear model sentiment

First text classification model

Sentiment classification

Start as Sentiment.

IMDB movie reviews dataset

- http://ai.stanford.edu/~amaas/data/sentiment/
- Contains 25000 positive and 25000 negative reviews

- Contains at most 30 reviews per movie
- At least 7 stars out of 10 → positive (label = 1)
- At most 4 stars out of 10 → negative (label = 0)
- 50/50 train/test split
- Evaluation: accuracy

Sentiment classification

Features: bag of 1-grams with TF-IDF values

- 25000 rows, 74849 columns for training
- Extremely sparse feature matrix – 99.8% are zeros

acting	actingjob	actings	actingwise
0.000000	0.0	0.0	0.0
0.000000	0.0	0.0	0.0
0.053504	0.0	0.0	0.0
0.033293	0.0	0.0	0.0
0.000000	0.0	0.0	0.0

extremely sparse

=)

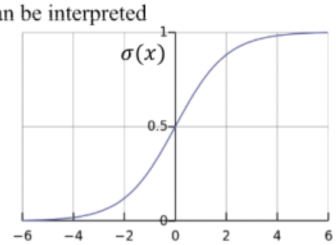
Use linear model (???

Naive Bayes

Sentiment classification

Model: Logistic regression

- $p(y=1|x) = \sigma(w^T x)$
- Linear classification model
- Can handle sparse data
- Fast to train
- Weights can be interpreted



Sentiment classification

Logistic regression over bag of 1-grams with TF-IDF

- Accuracy on test set: 88.5%
- Let's look at learnt weights:

ngram	weight
great	9.042803
excellent	8.487379
perfect	6.907277
best	6.440972
wonderful	6.237365

Top positive

VS

ngram	weight
worst	-12.748257
awful	-9.150810
bad	-8.974974
waste	-8.944854
boring	-8.340877

Top negative

Better sentiment classification

Let's try to add 2-grams

- Throw away n-grams seen less than 5 times
- 25000 rows, 156821 columns for training

and am	and amanda	and amateur	and amateurish	and amazing
0.068255	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0

Better sentiment classification

Logistic regression over bag of 1,2-grams with TF-IDF

- Accuracy on test set: 89.9% (+1.5%)
- Let's look at learnt weights:

well worth	13.788515
best	13.633200
rare	13.570259
better than	13.500025

VS

bad	-24.467648
poor	-24.319746
the worst	-23.773352
waste	-22.880340

Near top positive

Near top negative

How to make it even better

Play around with tokenization

- Special tokens like emoji, ":)" and "!!!" can help

Try to normalize tokens

- Adding stemming or lemmatization

Try different models

- SVM, Naïve Bayes, …

Throw BOW away and use Deep Learning

- https://arxiv.org/pdf/1512.08183.pdf
- Accuracy on test set in 2016: 92.14% (+2.5%)

Summary

simpler model works and has its merit.

- Bag of words and simple linear models actually work for texts
- The accuracy gain from deep learning models is not mind blowing for sentiment classification
- In the next video we'll look at spam filtering task

hashing

Spam filtering task

Mapping n-grams to feature indices

If your dataset is small you can store {n-gram → feature index} in hash map.

But if you have a huge dataset that can be a problem

- Let's say we have 1 TB of texts distributed on 10 computers
- You need to vectorize each text
- You will have to maintain {n-gram → feature index} mapping — e.g. one of the machine finds a new n-gram. Add a new item of the mapping. (new ~~feature~~ index).
  - May not fit in memory on one machine
  - Hard to synchronize
- An easier way is hashing: {n-gram → hash(n-gram) % 2^{20}}
  - Has collisions but works in practice
  - sklearn.feature_extraction.text.**HashingVectorizer**
  - Implemented in **vowpal wabbit** library

$\boxed{\text{Solution}}$ use hashing

Collision is a problem to solve for hashing.

lots of collision — not good

to higher

collision negletable.

Spam filtering is a huge task

Spam filtering proprietary dataset

- https://arxiv.org/pdf/0902.2206.pdf
- 0.4 million users
- 3.2 million letters
- 40 million unique words

Let's say we map each token to index using hash function ϕ

hash collision is unlikely.

- $\phi(x) = \text{hash}(x) \ \% \ 2^b$
- For $b = 22$ we have 4 million features ← from 40 mil
- That is a huge improvement over 40 million features
- It turns out it doesn't hurt the quality of the model

Hashing example

take a small b. 2^b ...

$\phi(S) = hash(s) \% 2^b$.

- $\phi(good) = 0$
- $\phi(movie) = 1$
- $\phi(not) = 2$
- $\boldsymbol{\phi(a) = 3}$
- $\boldsymbol{\phi(did) = 3}$
- $\phi(like) = 4$

Hash collision

$$\mathbf{hash}(\boldsymbol{s}) = \overset{0-256}{\boldsymbol{s}[\mathbf{0}]} + \overset{0-256}{\boldsymbol{s}[\mathbf{1}]}\boldsymbol{p}^{\mathbf{1}} + \cdots + \boldsymbol{s}[\boldsymbol{n}]\boldsymbol{p}^{\boldsymbol{n}}$$

s – string
p – fixed prime number → 是怎么选间隔的 s[0] s[1]…s[n] 呢?
$s[i]$ – character code

→ some string hashes into the same value — collision.

good movie
not a good movie
did not like

这表示 →

0 (good)	1 (movie)	2 (not)	3	4
1	1	0	0	0
1	1	1	1	0
0	0	1	1	1

u: user ID

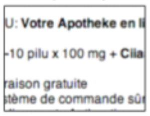
Trillion features with hashing

u_

Personalized tokens trick

- $\phi_o(token) = \text{hash}(token) \ \% \ 2^b$
- for certain user → $\phi_u(token) = \text{hash}(u + \text{"\_"} + token) \ \% \ 2^b$ new features. 就在这个词前加u_
- We obtain 16 trillion pairs (user, word) but still 2^b features

text document (email)



bag of words

NEU
Votre
Apotheke
…

bag of words (personalized)

NEU
USER123_NEU
Votre
USER123_Votre
Apotheke
USER123_Apotheke
…

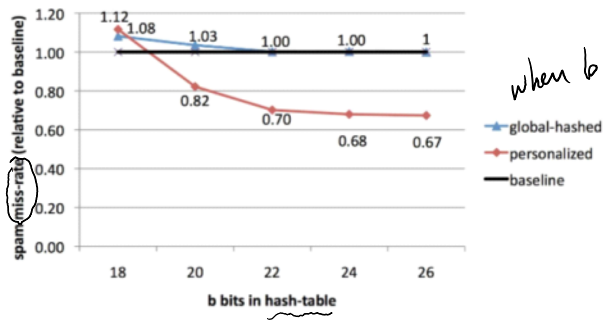
$\phi_0(x) + \phi_u(x)$

$$\begin{array}{c} 1 \\ 0 \\ -1 \\ 0 \\ 0 \\ -1 \\ 0 \\ 1 \\ 0 \\ \dots \end{array}$$

https://arxiv.org/pdf/0902.2206.pdf

Experimental results

- For $b = 22$ it performs just like a linear model on original tokens
- We observe that personalized tokens give a huge improvement in miss-rate!



这样是不管 interpretation 的?

small # of features — some features coded the same when ~~that~~ they happen to have collision?

when b is large. you do not lose.

Why the personalize hashing work?

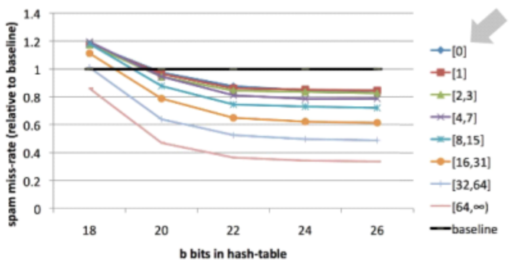
Localized!

Why personalized features work

Personalized features capture "local" user-specific preference

- Some users might consider newsletters a spam but for the majority of the people they are fine

How will it work for new users?



What?

Number of emails in training

less example for an user — hurt quality

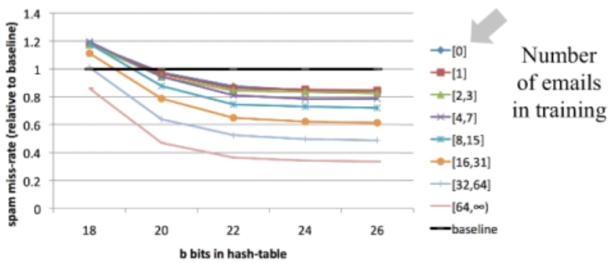
But even for an user with no training example

⇒ personalized hashing still works better

Why personalized features work

It turns out we learn better "global" preference having personalized features which learn "local" user preference

- You can think of it as a more universal definition of spam



Why the size matters

Why do we need such huge datasets?

- It turns out you can learn better models using the same simple linear classifier

Ad click prediction

可以看看是什么

- https://arxiv.org/pdf/1110.4198.pdf
- Trillions of features, billions of training examples

Sample 大还是有帮助的.

- Data sampling hurts the model

	1%	10%	100%
auROC	0.8178	0.8301	0.8344
auPRC	0.4505	0.4753	0.4856
NLL	0.2654	0.2582	0.2554

Sampling rate

Vowpal Wabbit

A well-known ML model used to train linear model

还有专做 linear model 的.

- A popular machine learning library for training $\boxed{\text{linear models}}$
- Uses feature hashing internally
- Has lots of features
- Really fast and scales well

VOWPAL WABBIT

Format: label | sparse features ...
1 | 13:3.9656971e-02 24:3.4781646e-02 ...
which corresponds to:
1 | tuesday year ...
command: time vw –sgd rcv1.train.txt -c

https://github.com/JohnLangford/vowpal_wabbit/wiki

6

Summary

- We've taken a look on applications of feature hashing
- Personalized features is a nice trick
- Linear models over bag of words scale well for production
- In the next video we'll take a look at text classification problem using deep learning

Usually . use linear model as a baseline

1 point

1. Choose true statements about text tokens.

- [ ] A model without stemming/lemmatization can be the best
- [ ] Lemmatization is always better than stemming
- [x] Stemming can be done with heuristic rules
- [x] Lemmatization needs more storage than stemming to work

---

1 point

2. Imagine you have a texts database. Here are stemming and lemmatization results for some of the **words**:

Word	Stem	Lemma
operate	oper	operate
operating	oper	operating
operates	oper	operates
operation	oper	operation
operative	oper	operative
operatives	oper	operative
operational	oper	operational

Imagine you want to find results in your texts database using the following queries:

1. **operating system** (we are looking for articles about OS like Windows or Linux)
2. **operates in winter** (we are looking for machines that can be operated in winter)

Before execution of our search we apply either stemming or lemmatization to both query and texts. Compare stemming and lemmatization for a given query and choose the correct statements.

- [ ] Stemming provides higher F1-score for **operating system** query.
- [x] Lemmatization provides higher precision for **operates in winter** query.
- [x] Stemming provides higher recall for **operates in winter** query.
- [ ] Stemming provides higher precision for **operating system** query.

1 point

3. Choose correct statements about bag-of-words (or n-grams) features.

- [x] You get the same vectorization result for any words permutation in your text. → not work for 2gram, 3gram
- [x] We prefer **sparse** storage formats for bag-of-words features.
- [ ] Classical bag-of-words **vectorizer** (object that does vectorization) needs an amount of RAM at least proportional to T, which is the number of unique tokens in the dataset.
- [ ] Hashing **vectorizer** (object that does vectorization) needs an amount of RAM proportional to vocabulary size to operate. only need to store a hashing function
- [x] For bag-of-words features you need an amount of RAM at least proportional to $N \times T$, where N is the number of documents, T is the number of unique tokens in the dataset.

4. Let's consider the following texts:

- good movie
- not a good movie
- did not like
- i like it
- good one

Let's count **Term Frequency** here as a distribution over tokens in a particular text, for example for text "good one" we have TF = 0.5 for "good" and "one" tokens.

Term frequency (TF)

- $\mathrm{tf}(t, d)$ – frequency for term (or n-gram) t in document d
- Variants:

weighting scheme	TF weight
binary	$0, 1$
raw count	$f_{t,d}$
$\boxed{\text{term frequency}}$	$\boxed{f_{t,d} / \sum_{t' \in d} f_{t',d}}$
log normalization	$1 + \log(f_{t,d})$

why not add 1?

Inverse document frequency (IDF)

- $N = |D|$ – total number of documents in corpus
- $|\{d \in D : t \in d\}|$ – number of documents where the term t appears
- $\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D:\, t \in d\}|}$

$0.5 \times \log(5/3) + 0.5 \times \log(5/2)$

What is the **sum** of TF-IDF values for 1-grams in "good movie" text? Enter a math expression as an answer. Here's an example of a valid expression: *log(1/2)*0.1*.

Preview

$-0.306852819440055 \log(2) + (-\log(2) + 1)(-\log(3) + \log(5)) + 0.306852819440055 \log(5)$