More LMs and Naïve Bayes

CS 780/880 Natural Language Processing Lecture 8

Samuel Carton, University of New Hampshire

Last lecture

Key idea: Probabilistic language modeling

Concepts

- Conditional probability
- Chain rule
- N-gram models
- Uses of language models
  - Generation
  - Evaluation
- Perplexity

Unigram model

Basic idea: model the text as the individual words occurring independently

- Parametrized by corpus token frequencies

$$P(w^{(1)} \ldots w^{(N)}) = \prod_{i=1}^{N} P(w^{(i)})$$

What's the problem with this?

Bigram model

Basic idea: model text as words being dependent on **only** the prior word

- Parameterized by token co-occurrence frequencies

$$P(w^{(1)} \dots w^{(N)}) = \prod_{i=1}^{N} P(w^{(i)} \mid w^{(i-1)})$$

A bigram model is a type of **Markov Chain**

Example: identifying the hater

Enron dataset: ~500,000 emails from Enron company, released due to scandal

- My corpus: 2000 emails each sampled from the top 5 emailers

Question: Who sent the email “You are a huge jerk and I hate you”?

NLTK Language model functionality

```python
from nltk.util import bigrams

# NLTK has functionality for generating all N-grams from a sequence of tokens
list(bigrams(enron_df['tokenized'][0]))[0:10]
```

```
[('what', 'are'),
 ('are', 'you'),
 ('you', 'talking'),
 ('talking', 'about'),
 ('about', '?'),
 ('?', 'sandra'),
 ('sandra', 'dial'),
 ('dial', '04/24/2000'),
 ('04/24/2000', '03:36'),
 ('03:36', 'pm')]
```

NLTK Language model functionality

```
# We'll take a look at the first email in our dataset
print(enron_df['message_text'][0])
```

```
What are you talking about?



Sandra Dial
04/24/2000 03:36 PM
To: Chris Germany/HOU/ECT@ECT
cc: Scott Hendrickson/HOU/ECT@ECT
Subject: Re: Feb 00 (RE: Voice Mail)

Scott-- Thanks.


Dude (CG),

I knew you had to be involved with this somehow.....  :-)

Can you help me with this, please. Thanks. Let me know if you need more info
( I really have no more info other than this e-mail thread). Thanks man.

S
x5-7213
--------------------- Forwarded by Sandra Dial/HOU/ECT on 04/24/2000 03:30
PM ---------------------------


Scott Hendrickson
04/24/2000 03:31 PM
To: Sandra Dial/HOU/ECT@ECT
cc:
Subject: Re: Feb 00 (RE: Voice Mail)

Well, just in that amount of time, I found out more about the deal....

I was inherited from CES.  CES had originally made the sale and when we
bought their book of business, we ended up with that sale.  You may want to
speak with Chris Germany about it, he was very involved with the CES
assimilation.

Scott
```

NLTK Language model functionality

```
# It also has functionality for adding "beginning-of-sequence" and "end-of-sequence" tokens to a
# token sequence

from nltk.util import pad_sequence
list(pad_sequence(enron_df['tokenized'][0],
                  pad_left=True,
                  left_pad_symbol="<s>",
                  pad_right=True,
                  right_pad_symbol="</s>",
                  n=2))[0:10]
```

```
['<s>',
 'what',
 'are',
 'you',
 'talking',
 'about',
 '?',
 'sandra',
 'dial',
 '04/24/2000']
```

NLTK Language model functionality

```python
# Both of these utilities are packaged into a convenient pipeline
# which will take in a collection of (tokenized) texts, pad each one
# with start/end tokens, and then count all n-grams up to the number you
# give it (in our case, 2)
from nltk.lm.preprocessing import padded_everygram_pipeline

# I am training this model on just the first text
train, vocab = padded_everygram_pipeline(2, enron_df['tokenized'][0:1])
```

```python
# These things are generators, so they don't do anything until called on
print(train)
print(vocab)
```

```
<generator object padded_everygram_pipeline.<locals>.<genexpr> at 0x7cf18a443060>
<itertools.chain object at 0x7cf16ce91a20>
```

NLTK Language model functionality

```python
# Both of these utilities are packaged into a convenient pipeline
# which will take in a collection of (tokenized) texts, pad each one
# with start/end tokens, and then count all n-grams up to the number you
# give it (in our case, 2)
from nltk.lm.preprocessing import padded_everygram_pipeline

# I am training this model on just the first text
train, vocab = padded_everygram_pipeline(2, enron_df['tokenized'][0:1])
```

```python
# These things are generators, so they don't do anything until called on
print(train)
print(vocab)
```

```
<generator object padded_everygram_pipeline.<locals>.<genexpr> at 0x7cf18a443060>
<itertools.chain object at 0x7cf16ce91a20>
```

NLTK Language model functionality

```
# Once we have training corpus and a vocabulary, we can create a MLE model to
# fit across it
from nltk.lm import MLE
lm = MLE(2)

#Then we fit the model
lm.fit(train, vocab)
print(lm.vocab)
print(len(lm.vocab))
```

```
<Vocabulary with cutoff=1 unk_label='<UNK>' and 110 items>
110
```

NLTK Language model functionality

```python
# We can look up a piece of text in the vocabulary
lm.vocab.lookup(enron_df['tokenized'][0])[0:10]
```

```
('what',
 'are',
 'you',
 'talking',
 'about',
 '?',
 'sandra',
 'dial',
 '04/24/2000',
 '03:36')
```

```python
# And any token it doesn't recognize it will replace with <UNK>
lm.vocab.lookup(enron_df['tokenized'][1])[0:10]
```

```
('i', '<UNK>', 'we', '<UNK>', 'about', 'this', 'the', 'other', '<UNK>', '.')
```

NLTK Language model functionality

```
# We can look up a piece of text in the vocabulary
lm.vocab.lookup(enron_df['tokenized'][0])[0:10]
```

```
('what',
 'are',
 'you',
 'talking',
 'about',
 '?',
 'sandra',
 'dial',
 '04/24/2000',
 '03:36')
```

```
# And any token it doesn't recognize it will replace with <UNK>
lm.vocab.lookup(enron_df['tokenized'][1])[0:10]
```

```
('i', '<UNK>', 'we', '<UNK>', 'about', 'this', 'the', 'other', '<UNK>', '.')
```

NLTK Language model functionality

```python
# We can see how many ngrams it found
print(lm.counts)
```

```
<NgramCounter with 2 ngram orders and 443 ngrams>
```

```python
# We can count how many instances of the 'a' unigram it found
lm.counts['the']
```

```
3
```

```python
# And how many counts of the "in that" bigram
lm.counts[['in']]['that']
```

```
1
```

NLTK Language model functionality

```
# We can look up probabilities of particular unigrams under the model!
lm.score('the')
```

```
0.013513513513513514
```

```
lm.score('a')
```

```
0.0
```

```
# And we can do the same for bigrams
lm.score('that', ['in'])
```

```
1.0
```

NLTK Language model functionality

```
# It's often useful to report log-probabilities rather than raw probabilities
# because of underflow
lm.logscore('the')
```

```
-6.209453365628949
```

```
# Although this can result in weird arithmatic when dealing with unknown tokens
lm.logscore('a')
```

```
-inf
```

NLTK Language model functionality

```
# Finally, we can generate new sequences according to the model
lm.generate(1, random_seed=3)
```

```
'00'
```

```
lm.generate(10, random_seed=7)
```

```
['</s>', '--', '--', '--', '--', '--', '--', '--', '-', ')']
```

Modeling our Enron data

```python
from nltk.lm.preprocessing import padded_everygram_pipeline

# First I train a global model on all the emails
global_train, global_vocab = padded_everygram_pipeline(2, enron_train_df['tokenized'])

# Use a so-called "Lidstone" model to do the add-one smoothing I talked about last lecture
from nltk.lm import Lidstone
global_lm = Lidstone(order=2, gamma=1)

global_lm.fit(global_train, global_vocab)
```

```python
# Here's an email generated by our global LM
display(' '.join(global_lm.generate(100, text_seed='<s>',random_seed=8)))
```

```
', vladimir gorny , a consumer welfare ( hour : dicarlo , ccampbell @ ect , 2000.= if the talks between air credits will attempt to discover the escrow
as of the talking to : 1.0 content-type : 7bit x-from : pverde_5_devers interchg_id : can the borland database ! an gsa would remain responsible for ban
kruptcy filing in abilene , john.anderson @ enron.com , '' < /o=enron/ou=na/cn=recipients/cn=pdavis1 > , sarah novosel/corp/enron @ enron.com , monika.c
ausholli @ enron employees at $ 50 books . i am on behalf of students and appointments . = and=20 electricity rates could review what .'
```

Modeling our Enron data

```
# Then I try training an individual LM for each person
personal_lms = {}

from nltk.lm import Lidstone

#I use the pandas groupby function to split up the training set by name
for group_name, group_df in enron_train_df.groupby('name'):
  group_train, group_vocab = padded_everygram_pipeline(2, group_df['tokenized'])
  group_lm = Lidstone(order=2,gamma=12)
  group_lm.fit(group_train, group_vocab) # Use the same (global) vocabulary for everyone
  personal_lms[group_name] = group_lm
```

```
personal_lms
```

```
{'chris': <nltk.lm.models.Lidstone at 0x7cf168b35f60>,
 'jeff': <nltk.lm.models.Lidstone at 0x7cf1683fd120>,
 'kay': <nltk.lm.models.Lidstone at 0x7cf167ba5ae0>,
 'pete': <nltk.lm.models.Lidstone at 0x7cf16722a2f0>,
 'vince': <nltk.lm.models.Lidstone at 0x7cf16729d000>}
```

Modeling our Enron data

```python
new_email = "you are a huge jerk and I hate you"
```

```python
import numpy as np
def score_new_email(email_text:str, lm:nltk.lm.models.MLE):
  tokens = preprocess_enron(email_text) #tokenize the new email
  tokens = list(pad_sequence(tokens, # add start/end tokens
                pad_left=True,
                left_pad_symbol="<s>",
                pad_right=True,
                right_pad_symbol="</s>",
                n=2))
  print(tokens)
  tokens = lm.vocab.lookup(tokens) # Look up the tokens in the vocab
  print(tokens)
  scores = []
  for i in range(1, len(tokens)): # Look up score of each token given previous token
    scores.append(lm.logscore(tokens[i], [tokens[i-1]]))

  return np.sum(scores), scores
```

Modeling our Enron data

```
for name, personal_lm in personal_lms.items():
  score, scores = score_new_email(new_email, personal_lm)
  print(name, np.round(score,2))
  print('\tScores',np.round(scores,2))
  print('---')
```

```
['<s>', 'you', 'are', 'a', 'huge', 'jerk', 'and', 'i', 'hate', 'you', '</s>']
('<s>', 'you', 'are', 'a', 'huge', '<UNK>', 'and', 'i', 'hate', 'you', '</s>')
chris -127.87
	Scores [-12.04 -10.31 -12.23 -13.79 -13.99 -13.99  -9.69 -13.8  -13.99 -14.01]
---
['<s>', 'you', 'are', 'a', 'huge', 'jerk', 'and', 'i', 'hate', 'you', '</s>']
('<s>', 'you', 'are', 'a', 'huge', 'jerk', 'and', 'i', 'hate', 'you', '</s>')
jeff -135.48
	Scores [-12.85 -11.56 -12.66 -13.88 -14.84 -14.84 -10.54 -14.74 -14.73 -14.85]
---
['<s>', 'you', 'are', 'a', 'huge', 'jerk', 'and', 'i', 'hate', 'you', '</s>']
('<s>', 'you', 'are', 'a', 'huge', '<UNK>', 'and', 'i', 'hate', 'you', '</s>')
kay -126.97
	Scores [-12.65  -8.78 -12.95 -13.86 -14.05 -14.05  -9.66 -13.5  -14.05 -13.41]
---
['<s>', 'you', 'are', 'a', 'huge', 'jerk', 'and', 'i', 'hate', 'you', '</s>']
('<s>', 'you', 'are', 'a', '<UNK>', '<UNK>', 'and', 'i', '<UNK>', 'you', '</s>')
pete -113.7
	Scores [-11.44 -11.36 -11.36 -11.42 -11.35 -11.35 -11.36 -11.35 -11.35 -11.36]
---
['<s>', 'you', 'are', 'a', 'huge', 'jerk', 'and', 'i', 'hate', 'you', '</s>']
('<s>', 'you', 'are', 'a', 'huge', '<UNK>', 'and', 'i', 'hate', 'you', '</s>')
vince -134.97
	Scores [-14.71  -9.92 -12.97 -14.14 -14.7   -14.7   -9.92 -14.72 -14.7   -14.5 ]
---
```

Bayes Rule

Conditional probability

When two variables may be dependent, then their joint probability is expressed as follows:

$$P(X,Y) = P(Y)P(X|Y) = P(X)P(Y|X)$$

If they happen to be independent, then $P(X|Y) = P(X)$ and $P(Y|X) = P(Y)$, so

$$P(X,Y) = P(Y)P(X) = P(X)P(Y)$$

Bayes Rule

It follows from

$$P(X,Y) \;=\; P(Y)P(X|Y) \;=\; P(X)P(Y|X)$$

that

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}$$

Examples

$$P(Lung\ cancer|Cough) = \frac{P(Cough|Lung\ cancer)P(Lung\ cancer)}{P(Cough)}$$

$$P(Conspiracy|Event) = \frac{P(Event|Conspiracy)P(Conspiracy)}{P(Event)}$$

$$P(Barista\ likes\ you|Smiles\ when\ they\ give\ you\ coffee)$$
$$= \frac{P(Smiles\ when\ they\ give\ you\ coffee|Barista\ likes\ you)P(Barista\ likes\ you)}{P(Smiles\ when\ they\ give\ you\ coffee)}$$

Relative probabilities

Often we only care about the relative probability of two possible outcomes, rather than their true probability:

$$P(Lung\ cancer|Cough) \text{ vs. } P(COVID|Cough)$$

$$\frac{P(Cough|Lung\ cancer)P(Lung\ cancer)}{P(Cough)} \text{ vs. } \frac{P(Cough|COVID)P(COVID)}{P(Cough)}$$

Because we only care about the relative value, we can ignore the denominator

$P(Cough|Lung\ cancer) \approx P(Cough|COVID) \approx 1.0$

~50 million COVID cases in 2022, ~300k new lung cancer cases in 2023

So P(COVID) = .15, and P(Lung cancer) = 0.001

So $P(COVID|Cough)$ is **150 times** higher than $P(Lung\ cancer|Cough)$

https://www.cancer.org/cancer/lung-cancer/about/key-statistics.html

https://covid.cdc.gov/covid-data-tracker/#trends_totalcases_select_00

Base rate fallacy

A lot of fallacious thinking comes from ignoring the **base rates** P(X) and P(Y) in $\frac{P(Y|X)P(X)}{P(Y)}$

$$P(Hypothesis \mid Rare\ event) = \frac{P(Rare\ event \mid Hypothesis)P(Hypothesis)}{P(Rare\ event)}$$

P(Hypothesis) is often lower than you think

P(Rare event) is often higher than you think

https://en.wikipedia.org/wiki/Base_rate_fallacy

https://en.wikipedia.org/wiki/List_of_cognitive_biases

Naïve Bayes

Application to text

Classification:

$$P(Class \mid Words)$$

$$P(Class\ 0 \mid Words) \text{ vs. } P(Class\ 1 \mid Words)$$

$$\frac{P(Words \mid Class\ 0)P(Class\ 0)}{P(Words)} \text{ vs. } \frac{P(Words \mid Class\ 1)P(Class\ 1)}{P(Words)}$$

We can ignore P(Words), but how do we calculate:

- $P(Words \mid Class\ 0)$
- $P(Class\ 0)$
- $P(Words \mid Class\ 1)$
- $P(Class\ 1)$

Application to text

$$P(Class\ 0) = \frac{\#\ Class\ 0}{\#\ Class\ 0 + \#\ Class\ 1}$$

- And likewise for class 1

$P(Words \mid Class\ 0)$

- Build an n-gram model of all texts for which class is Class 0
- Use this model to estimate $P(Words \mid Class\ 0)$
- And likewise for Class 1

Naïve Bayes

Basic idea: apply Bayes rule to find relative likelihoods of $P(Class\ 0 \mid Words)$ vs. $P(Class\ 1 \mid Words)$, using **unigram model** for $P(Words \mid Class\ C)$

So if we consider words = $\{w_0, w_1, \dots, wN\}$:

$$P(Class\ 0 \mid Words) \propto P(Class\ 0) \prod_{i=1}^{N} P(wi \mid Class\ 0)$$

$$P(Class\ 1 \mid Words) \propto P(Class\ 1) \prod_{i=1}^{N} P(wi \mid Class\ 1)$$

Read the SST-2 dataset

```
1 display(dev_df)
```

	sentence	label
0	it 's a charming and often affecting journey .	1
1	unflinchingly bleak and desperate	0
2	allows us to hope that nolan is poised to emba...	1
3	the acting , costumes , music , cinematography...	1
4	it 's slow -- very , very slow .	0
...	...	...
867	has all the depth of a wading pool .	0
868	a movie with a real anarchic flair .	1
869	a subject like this should inspire reaction in...	0
870	... is an arthritic attempt at directing by ca...	0
871	looking aristocratic , luminous yet careworn i...	1

872 rows × 2 columns

Preprocess and vectorize the data

```python
from nltk import PorterStemmer
```

```python
# for this dataset, the tokenization has already been done for us
stemmer = PorterStemmer()
def preprocess(s):
    return ' '.join([stemmer.stem(token) for token in s.strip().split(' ')])
```

```python
train_df['preprocessed'] = train_df['sentence'].apply(preprocess)
dev_df['preprocessed'] = dev_df['sentence'].apply(preprocess)
```

```python
from sklearn.feature_extraction.text import CountVectorizer
```

```python
# Why are we using a CountVectorizer here instead of TF-IDF?

vectorizer = CountVectorizer()
train_X = vectorizer.fit_transform(train_df['preprocessed'])
dev_X = vectorizer.transform(dev_df['preprocessed'])
```

Build and evaluate the model

```
from sklearn.naive_bayes import MultinomialNB
```

```
# See https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive_bayes.MultinomialNB
# for hyperparameter options

model = MultinomialNB()
```

```
model.fit(train_X, train_df['label'])
```

```
MultinomialNB()
```

```
dev_py = model.predict(dev_X)
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
def evaluate_predictions(y, py):
    print(f'Accuracy: {accuracy_score(y, py):.3f}')
    print(f'Precision: {precision_score(y, py):.3f}')
    print(f'Recall: {recall_score(y, py):.3f}')
    print(f'F1: {f1_score(y, py):.3f}')
```

Build and evaluate the model

Naïve Bayes:

```
# Evaluating on the dev set
evaluate_predictions(dev_df['label'], dev_py)
```

```
Accuracy: 0.807
Precision: 0.794
Recall: 0.840
F1: 0.816
```

```
# Evaluating on a sample of the training set
train_py = model.predict(train_X[0:1000])
evaluate_predictions(train_df['label'].iloc[0:1000],train_py)
```

```
Accuracy: 0.891
Precision: 0.894
Recall: 0.906
F1: 0.900
```

K-nearest-neighbors:

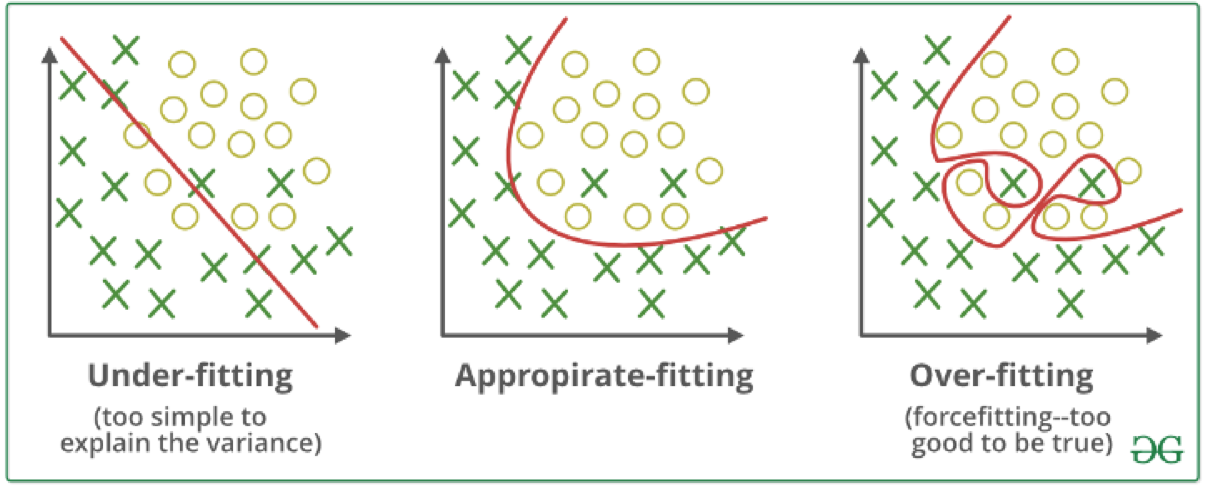
```
evaluate_model(dev_X, dev_y, classifier)
```

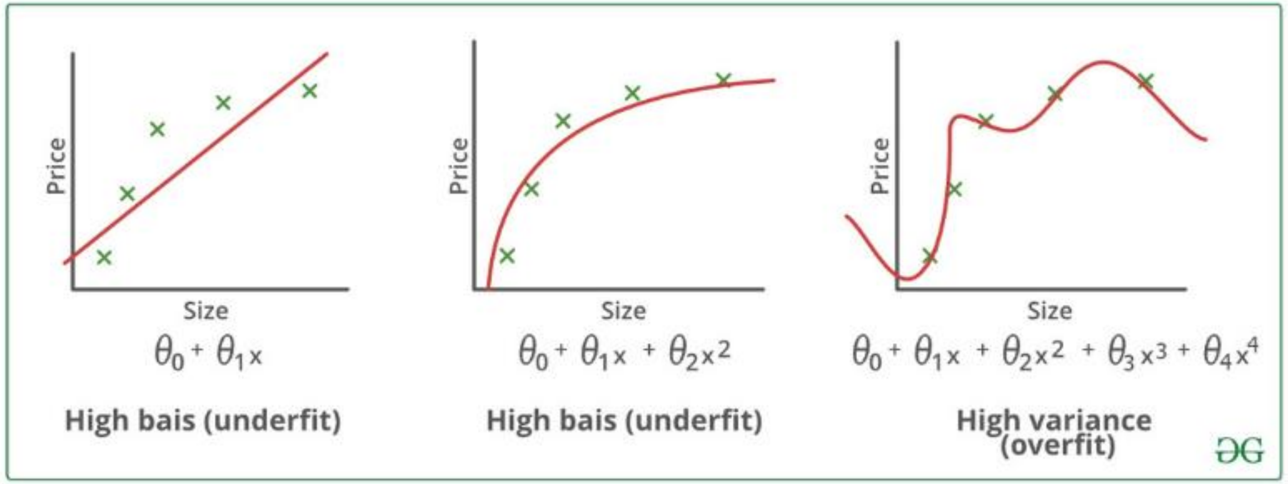
```
Accuracy: 0.742
Precision: 0.707
Recall: 0.842
F1: 0.769
```

```
evaluate_model(train_X[0:1000], train_y[0:1000], classifier)
```

```
Accuracy: 0.948
Precision: 0.945
Recall: 0.959
F1: 0.952
```

Overfitting and underfitting





Overfitting: model overly tuned to quirks of the training data—doesn't generalize

Underfitted: model not tuned enough to training data—doesn't capture data structure

Related (but not identical) to **bias-variance trade-off**

- High bias → underfitting
- High variance → overfitting

https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

Explaining the model

```python
# We can get the (log) probability of each word for each class
print('Word log-probs:')
display(model.feature_log_prob_)

# It will have one row for each class and one column for each word in the vocabulary
print('Log-probs matrix shape:')
display(model.feature_log_prob_.shape)

```

```
Word log-probs:
array([[-11.12495518,  -8.09240893, -10.20866444, ..., -12.51124954,
        -10.71949007, -10.90181162],
       [-11.03503481,  -9.50897851, -10.15956608, ..., -11.25817837,
        -12.64447273, -11.95132555]])
Log-probs matrix shape:
(2, 10106)
```

Explaining the model

```
# We can identify the words that were the biggest distinguishers by calculating
# the diff between the two rows
word_prob_diffs = model.feature_log_prob_[0] - model.feature_log_prob_[1]
word_prob_diffs
```

```
array([-0.08992036,  1.41656958, -0.04909837, ..., -1.25307117,
        1.92498266,  1.04951392])
```

```
# And then we can use numpy.argsort() and numpy.abs() to find the indices of the
# words with the biggest diff (positive or negative)
import numpy as np
sorted_diff_indices = np.argsort(np.abs(word_prob_diffs))
sorted_diff_indices
```

```
array([6103, 9942, 7833, ..., 6692, 6721, 9402])
```

```
# Numpy argsort always goes in ascending order, so to get the top K indices
# we have to grab the last K indices

# We can use -1 as the third part of our slice, to get these back in reverse order
k= 10
top_k_indices = sorted_diff_indices[:-k:-1]
```

Explaining the model

```python
# Then we can find the words and values associated with those indices
vocab = vectorizer.get_feature_names_out()
top_words = vocab[top_k_indices]
top_diffs = word_prob_diffs[top_k_indices]

print(f'Top {k} distinguishing words in our Naive Bayes classifier')
for word, diff in zip(top_words,top_diffs):
  print(f'\tWord:"{word}" - Diff: {diff:.3f}')
```

```
Top 10 distinguishing words in our Naive Bayes classifier
	Word:"unfunni" - Diff: 4.861
	Word:"poorli" - Diff: 4.698
	Word:"pointless" - Diff: 4.677
	Word:"tiresom" - Diff: 4.588
	Word:"eleg" - Diff: -4.410
	Word:"unnecessari" - Diff: 4.410
	Word:"badli" - Diff: 4.382
	Word:"embrac" - Diff: -4.355
	Word:"inept" - Diff: 4.338
```

Interpreting log-probability differences

If :

$$\log(P(w_i|\ class\ 0))\ -\ \log(P(w_i|\ class\ 1))\ =\ 4.8$$

Then:

$$\frac{P(wi|class\ 0)}{P(wi|class\ 1)} = e^{4.8} = 2.718^{4.8} = 121.51$$

Meaning that w_i (“unfunny” in this case) is **121.51** times more likely to occur in class 0 than in class 1

Explaining individual predictions

```python
sentence = 'the movie was pretty awful : not good at all .'
preprocessed_sentence = preprocess(sentence)
sentence_x = vectorizer.transform([preprocessed_sentence])
py = model.predict(sentence_x)
print(f'Model prediction for "{preprocessed_sentence}": {py}')
```

```
Model prediction for "the movi wa pretti aw : not good at all .": [0]
```

```python
# we can find the vocab indices for the tokens in the sentence
sentence_tokens = preprocessed_sentence.split(' ')
token_indices = [vectorizer.vocabulary_[token] for token in sentence_tokens \
                 if token in vectorizer.vocabulary_] #there's one or two stopwords to ignore
token_indices
```

```
[8892, 5827, 9702, 6828, 680, 6067, 3791, 615, 339]
```

Explaining individual sentences

Is this overfitting?

```
# Then we can do the same thing as we did with the top indices above
sentence_words = vocab[token_indices]
sentence_diffs = word_prob_diffs[token_indices]
print(f'Class probability differences for tokens in the sentence:')
for word, diff in zip(sentence_words,sentence_diffs):
    print(f'\tWord:"{word}" - Diff: {diff:.3f}')
```

```
Class probability differences for tokens in the sentence:
	Word:"the" - Diff: -0.032
	Word:"movi" - Diff: 0.196
	Word:"wa" - Diff: 0.836
	Word:"pretti" - Diff: -0.643
	Word:"aw" - Diff: 1.574
	Word:"not" - Diff: 0.679
	Word:"good" - Diff: -1.021
	Word:"at" - Diff: 0.080
	Word:"all" - Diff: 0.111
```

Concluding thoughts

Naïve bayes: application of Bayes Rule + unigram language modeling to classification

Huge deal in 1998

Limitations?