Basic statistical language modeling

CS 780/880 Natural Language Processing Lecture 7

Samuel Carton, University of New Hampshire

Last lecture

Key idea: Dimension reduction

Concepts

- Dimensionality of data
- Variance of data
- Principle components
- Matrix factorization
- SVD and PCA
- Application to clustering

Toolkits

- Scikit-learn for SVD

Probability review

Random variables

A random variable X can take different values depending on chance

Notation:

- p(X = x) is the probability that r.v. X takes value x
  - p(x) is shorthand for the same
- p(X) is the distribution over values X can take (a function)

Example: flipping a coin; P(X = heads) = P(X=tails) = 0.5

Discrete distributions

A discrete distribution enumerates the values a random variable can take and how likely each one is

Examples:

p(flipping a coin) = [0.5, 0.5]

p(rolling a die) = [.167, .167, .167, .167, .167, .167]

p(flipping a rigged coin) = [0.25, 0.75]

p(rolling a weighted die) = [0.1, 0 .1, 0.1,0 .1, 0.1, 0.5]

How does **entropy** relate to the values in the discrete distribution?

Joint probability and product rule

The **joint probability** of two random variables X and Y describes the total chance they take on a particular pair of values: $p(X = x, Y = y)$

If X and Y are **independent**, then $p(X = x, Y = y) = p(X = x) * p(Y = y)$

Example: two coin flips X and Y. $p(X = \text{heads}, Y = \text{heads}) = 0.5 * 0.5 = 0.25$

Conditional probability

If X and Y are **dependent**, then you have to think of the probability of X given Y:
$p(X = x \mid Y = y)$

In this case, the joint probability of X and Y is $p(Y=y) * p(X = x \mid Y = y)$

Example: Weather is 50% sunny and 50% cloudy; I am 25% likely to run when sunny and 10% likely when rainy.

P(run|sunny) = **.25**

P(run, sunny) = 0.5 * 0.25 = **0.125**

P(run) = P(run, sunny) + P(run, cloudy) = 0.5 * 0.25 + 0.5 * 0.1 = **0.175**

How does **mutual information** relate to dependence versus independence?

Chain rule

If we generalize to N joint random variables, we end up with the **chain rule**

$$
\begin{aligned}
P(X_1, X_2, \ldots, X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_2, X_1)....P(X_n|X_1, ...X_{n-1}) \\
&= P(X_1)\prod_{i=2}^{n} P(X_i|X_1 \ldots X_{i-1})
\end{aligned}
$$

Conditional probability table

With two variables X and Y, we can summarize their joint distribution with a **conditional probability table**

Each cell is P(X=column | y = row)

Example:

X

Y		Run	Don't run
	Sunny	0.25	0.75
	Cloudy	0.1	0.9

Maximum-likelihood probabilistic modeling

Whenever we build a probabilistic model of some phenomenon, we are deciding to fit it within some probabilistic form, and then finding the **most likely** parameters of our model to fit the data.

Example:

- Model: the weather is sunny with probability p, and cloudy with probability 1-p
- Data: 10 days; [sunny, cloudy, sunny, sunny, cloudy, sunny, sunny, cloudy, sunny, sunny]
- MLE estimate of p:

Maximum-likelihood probabilistic modeling

Whenever we build a probabilistic model of some phenomenon, we are deciding to fit it within some probabilistic form, and then finding the **most likely** parameters of our model to fit the data.

Example:

- Model: the weather is sunny with probability p, and cloudy with probability 1-p
- Data: 10 days; [sunny, cloudy, sunny, sunny, cloudy, sunny, sunny, cloudy, sunny, sunny]
- MLE estimate of *p*: 0.7

How did we do that?

Probabilistic NLP terminology

Statistical model (i.e. generative story): A hypothesis about how the data was generated

- E.g. “Every day is either sunny or rainy based on probability *p*, independent of previous day”
- Always an abstraction!

Parameters: the specific numbers associated with the model

- E.g. the specific *p* we choose for sunny vs cloudy
- Often denoted as θ

Observations: the real-world data we use to determine the parameters of the model

- E.g. we saw 7 sunny says and 3 cloudy days, so we think *p* is 0.7

Probabilistic NLP terminology

Likelihood: The probability of a given outcome or outcomes given a parameterized model

- E.g. if we think *p* is 0.7, then what is P(weather=cloudy)
- E.g. if we think *p* is 0.7, what is P(weather1=cloudy, weather2=sunny, weather3=sunny…)

Maximum likelihood estimation: Modeling paradigm where we choose ϴ to maximize likelihood of observed data

- So if we see a particular sequence of 7 sunny days and 3 cloudy days…
  - Likelihood if p=0.7 is: $0.7^7 * 0.3^3 = 0.0022$
  - Likelihood if p=0.5 is: $0.5^{10} = 0.0010$

MLE with conditional probability

Model:

- the weather is sunny with probability p and cloudy with probability 1-p
- I run with probability r_s when it is sunny and probability r_c when it is cloudy

Data: 10 runs

sunny	cloudy	cloudy	sunny	cloudy	sunny	cloudy	sunny	sunny	cloudy
run	no run	no run	no run	no run	no run	no run	run	no run	run

Counts:

	Run	Don't run
Sunny	2	4
Cloudy	1	3

$p =$

$r_s =$

$r_c =$

MLE with conditional probability

Model:

- the weather is sunny with probability p and cloudy with probability 1-p
- I run with probability r_s when it is sunny and probability r_c when it is cloudy

Data: 10 runs

sunny	cloudy	cloudy	sunny	cloudy	sunny	cloudy	sunny	sunny	cloudy
run	no run	no run	no run	no run	no run	no run	run	no run	run

Counts:

	Run	Don't run
Sunny	2	4
Cloudy	1	3

p = 2+4 / (2+4+1+3) = 6/10 = .6

r_s = 2 / (2+4) = 2/6 = 0.33

r_c = 1 / (1+3) = 1/4 = 0.25

Probabilistic language modeling

Unigram model

Basic idea: probability of a given word w depends **only** on its overall frequency within the corpus

- So the probability of a given text is the product of the individual word probabilities

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

Similar to bag-of-words in that it doesn't respect word order, but bag-of-words isn't explicitly probabilistic

Bigram model

Basic idea: the probability of word i depends **only** on word i-1

Example: “I am” is more likely than “I is”

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

What can we do with a language model?

Two main things:

1. Generate new text
2. Assess the likelihood of existing text

The corpus

```
review_0 = "The film was a delight--I was riveted."
review_1 = "It's the most delightful and riveting movie."
review_2 = "It was a terrible flick, the worst I have ever seen."
review_3 = "I have a feeling the film was recut poorly."

reviews = [review_0, review_1, review_2, review_3]
```

Preprocessing

```
raw_token_seqs = [word_tokenize(review.lower()) for review in reviews]
raw_token_seqs
```

```
[['the', 'film', 'was', 'a', 'delight', '--', 'i', 'was', 'riveted', '.'],
 ['it', "'s", 'the', 'most', 'delightful', 'and', 'riveting', 'movie', '.'],
 ['it',
  'was',
  'a',
  'terrible',
  'flick',
  ',',
  'the',
  'worst',
  'i',
  'have',
  'ever',
  'seen',
  '.'],
 ['i', 'have', 'a', 'feeling', 'the', 'film', 'was', 'recut', 'poorly', '.']]
```

Preprocessing

```python
# For convenience, we are going to add an imaginary [START] and [END] token at the
# beginning and end of each sequence. You'll see why in a little bit.

token_seqs = [['[START]'] + seq + ['[END]'] for seq in raw_token_seqs]
token_seqs
```

```
[['[START]',
  'the',
  'film',
  'was',
  'a',
  'delight',
  '--',
  'i',
  'was',
  'riveted',
  '.',
  '[END]'],
 ['[START]'
```

Counting

```python
token_counts = {}
for token_seq in token_seqs:
  for i in range(len(token_seq)):
    prev_token = token_seq[i]
    if prev_token not in token_counts:
      token_counts[prev_token] = {}
    if i < len(token_seq)-1: #if there is a next token, add it to the counts dict
      next_token = token_seq[i+1]

      if next_token not in token_counts[prev_token]:
        token_counts[prev_token][next_token] = 1
      else:
        token_counts[prev_token][next_token] += 1
token_counts
```

```
{'[START]': {'the': 1, 'it': 2, 'i': 1},
 'the': {'film': 2, 'most': 1, 'worst': 1},
 'film': {'was': 2},
 'was': {'a': 2, 'riveted': 1, 'recut': 1},
 'a': {'delight': 1, 'terrible': 1, 'feeling': 1},
 'delight': {'--': 1},
 '--': {'i': 1},
 'i': {'was': 1, 'have': 2},
 'riveted': {'.': 1},
 '.': {'[END]': 4},
 '[END]': {},
 'it': {"'s": 1, 'was': 1},
 "'s": {'the': 1},
 'most': {'delightful': 1},
 'delightful': {'and': 1},
 'and': {'riveting': 1},
 'riveting': {'movie': 1},
 'movie': {'.': 1},
 'terrible': {'flick': 1},
 'flick': {',': 1},
 ',': {'the': 1},
 'worst': {'i': 1},
 'have': {'ever': 1, 'a': 1},
 'ever': {'seen': 1},
 'seen': {'.': 1},
 'feeling': {'the': 1},
 'recut': {'poorly': 1},
 'poorly': {'.': 1}}
```

Counting

```
count_df = pd.DataFrame.from_records(data=counts, columns=vocabulary).fillna(0.0)
count_df.index = vocabulary
count_df
```

	[START]	the	film	was	a	delight	--	i	riveted	.	...	terrible	flick	,	worst	have	ever	seen	feeling	recut	poorly
[START]	0.0	1.0	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
the	0.0	0.0	2.0	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
film	0.0	0.0	0.0	2.0	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	0.0	0.0	0.0
was	0.0	0.0	0.0	0.0	2.0	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
a	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
delight	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
--	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
i	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	2.0	0.0	0.0	0.0	0.0	0.0
riveted	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	0.0

Counts to probabilities

	[START]	the	film	was	a	delight	--	i	riveted	.	...	terrible	flick	,	worst	have	ever	seen	feeling	recut	poorly
[START]	0.0	0.25	0.0	0.00	0.0	0.00	0.0	0.25	0.00	0.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.00	0.0
the	0.0	0.00	0.5	0.00	0.0	0.00	0.0	0.00	0.00	0.0	...	0.00	0.0	0.0	0.25	0.00	0.0	0.0	0.00	0.00	0.0
film	0.0	0.00	0.0	1.00	0.0	0.00	0.0	0.00	0.00	0.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.00	0.0
was	0.0	0.00	0.0	0.00	0.5	0.00	0.0	0.00	0.25	0.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.25	0.0
a	0.0	0.00	0.0	0.00	0.0	0.33	0.0	0.00	0.00	0.0	...	0.33	0.0	0.0	0.00	0.00	0.0	0.0	0.33	0.00	0.0
delight	0.0	0.00	0.0	0.00	0.0	0.00	1.0	0.00	0.00	0.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.00	0.0
--	0.0	0.00	0.0	0.00	0.0	0.00	0.0	1.00	0.00	0.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.00	0.0
i	0.0	0.00	0.0	0.33	0.0	0.00	0.0	0.00	0.00	0.0	...	0.00	0.0	0.0	0.00	0.67	0.0	0.0	0.00	0.00	0.0
riveted	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.00	1.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.00	0.0
.	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.00	0.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.00	0.0
[END]	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.00	0.00	0.0	...	0.00	0.0	0.0	0.00	0.00	0.0	0.0	0.00	0.00	0.0

Generating text

Once we have a MLE estimate model, we can **generate** text by just sampling from our model one word at a time

We can randomly sample, or take the most probable word at each step

We can stop after N tokens, or when we hit some stopping condition (like a [STOP] token, or a ".")

Generating text

```
]
random.seed(123)
prev_token = '[START]' # By defining this special token, we gave ourselves a
# place to start when we want to generate text
count = 0
while count < 20 and prev_token != '[END]':
  probs = cpt_df.loc[prev_token] #remember we want to get the row, not the column
  next_token = choice(vocabulary, p=probs)
  print(next_token)
  prev_token = next_token
  count += 1
```

```
i
have
a
delight
--
i
was
a
feeling
the
most
delightful
and
riveting
movie
.
[END]
```

Generating text

```
# We can also just pick the maximum-likelihood next token at every step
prev_token = '[START]'
count = 0
while count < 20 and prev_token != '[END]':
  probs = cpt_df.loc[prev_token]
  next_token = vocabulary[np.argmax(probs)]
  print(next_token)
  prev_token = next_token
  count += 1
```

```
it
was
a
delight
--
i
have
a
delight
--
i
have
a
delight
--
i
have
a
delight
--
```

Assessing text probability

Given our model, we can calculate the likelihood that a given text was produced by the model.

Example: Bigram model

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

p(“this is a sentence .”) = p(this|[START])p(is|this)p(a|is)p(sentence|a)p(.|sentence)

One token at a time

```python
review_0 = "The film was a delight--I was riveted."
review_1 = "It's the most delightful and riveting movie."
review_2 = "It was a terrible flick, the worst I have ever seen."
review_3 = "I have a feeling the film was recut poorly."

reviews = [review_0, review_1, review_2, review_3]
```

```python
# If we know the previous token, we can calculate the probability of any given next token
previous_token = previous_tokens[-1]
print(f'Previous tokens: {previous_tokens}')
for next_token in possible_next_tokens:
  print(f'\tPossible next token: "{next_token}";\tProbability: {cpt_df.loc[previous_token, next_token]}')
```

```
Previous tokens: ['it', 'was', 'not', 'the', 'most', 'riveting']
        Possible next token: "movie";   Probability: 1.0
        Possible next token: "film";    Probability: 0.0
        Possible next token: "delight"; Probability: 0.0
        Possible next token: ".";       Probability: 0.0
        Possible next token: "[END]";   Probability: 0.0
```

Over a whole sequence

```python
# By applying the chain rule, we can calculate the probability of a whole sequence

# This is one of the sequences from the corpus
sequence = ['[START]', 'it', "'s", 'the', 'most', 'delightful', 'and', 'riveting', 'movie', '.', '[END]']

def sequence_likelihood(seq:list, cpt:pd.DataFrame):
  cumulative_likelihood = 1.0
  for i in range(len(seq)-1):
    prev_token = seq[i]
    next_token = seq[i+1]
    prob = cpt.loc[prev_token, next_token]
    cumulative_likelihood = cumulative_likelihood*prob
  return cumulative_likelihood

print(f'Cumulative likelihood of entire sequence: {sequence_likelihood(sequence, cpt_df):.3f}')
```

```
Cumulative probability of entire sequence: 0.062
```

Over a whole sequence

Calculating cumulative log-likelihood is generally preferred to raw likelihood

- Why?

```
# This is one of the sequences from the corpus
sequence = ['[START]', 'it', "'s", 'the', 'most', 'delightful', 'and', 'riveting', 'movie', '.', '[END]']
```

```
def sequence_log_likelihood(seq:list, cpt:pd.DataFrame):
    cumulative_log_likelihood = 0.0
    for i in range(len(seq)-1):
        prev_token = seq[i]
        next_token = seq[i+1]
        prob = cpt.loc[prev_token, next_token]
        cumulative_log_likelihood = cumulative_log_likelihood + np.log(prob)
    return cumulative_log_likelihood

print(f'Cumulative log-likelihood of entire sequence: {sequence_log_likelihood(sequence, cpt_df):.3f}')
```

```
Cumulative log-probability of entire sequence: -2.773
```

Perplexity

One way to assess the quality of a probabilistic language model is to calculate the average likelihood of the training data under that model.

This measure, called "perplexity", captures the intuition that a good language model of a corpus is one that would have been very likely to generate that corpus

```
normalized_sequence_likelihoods = [-sequence_log_likelihood(seq, cpt_df)/len(seq) for seq in token_seqs]
print(f'Length-normalized sequence negative log-likelihoods: {normalized_sequence_likelihoods}')
perplexity = np.mean(normalized_sequence_likelihoods)
print(f'Perplexity: {perplexity:.3f}')
```

```
Perplexity: 0.408
```

Can only really be compared to itself for a given corpus

Smoothing

Any model bigger than a unigram model suffers from **sparsity** issues that make certain sequences impossible, and screws with all the math

Example: “was delightful” is an impossible bigram in our toy corpus because those two words never happen to occur together, even though they very much could.

```
review_0 = "The film was a delight--I was riveted."
review_1 = "It's the most delightful and riveting movie."
review_2 = "It was a terrible flick, the worst I have ever seen."
review_3 = "I have a feeling the film was recut poorly."

reviews = [review_0, review_1, review_2, review_3]
```

Solution: add some **smoothing** to the model which makes any bigram possible (if not likely)

Smoothing

```
# Big problem:
# We know this was an actual sequence:
real_sequence = ['[START]', 'it', "'s", 'the', 'most', 'delightful', 'and', 'riveting', 'movie', '.', '[END]']
# Our model rates this sequence as possible:
print(f'Cumulative likelihood of real sequence: {sequence_likelihood(real_sequence, cpt_df):.3f}')
```

```
Cumulative likelihood of real sequence: 0.062
```

```
# But what about this (very similar) sentence where "riveting" and "delightful" have been switched:
possible_sequence = ['[START]', 'it', "'s", 'the', 'most', 'riveting', 'and', 'delightful', 'movie', '.', '[END]']
# Our model rates this sequence as impossible!
print(f'Cumulative likelihood of possible sequence: {sequence_likelihood(possible_sequence, cpt_df):.3f}')
```

```
Cumulative likelihood of possible sequence: 0.000
```

Smoothing

```
1 smoothed_count_df = count_df+1
2 smoothed_count_df
```

	[START]	the	film	was	a	delight	--	i	riveted	.	...	terrible	flick	,	worst	have	ever	seen	feeling	recut	poorly
[START]	1.0	2.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
the	1.0	1.0	3.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0	1.0	1.0
film	1.0	1.0	1.0	3.0	1.0	1.0	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
was	1.0	1.0	1.0	1.0	3.0	1.0	1.0	1.0	2.0	1.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0
a	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0	...	2.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0
delight	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
--	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
i	1.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0	1.0	1.0	...	1.0	1.0	1.0	1.0	3.0	1.0	1.0	1.0	1.0	1.0
riveted	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	...	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Smoothing

```
# Now any sequence has at least a chance of being possible:
print(f'Smoothed likelihood of possible sequence: {sequence_likelihood(possible_sequence, smoothed_cpt_df):.20f}')
```

```
Smoothed likelihood of possible sequence: 0.00000000000041044225
```

```
# And calculating the log-likelihoods isn't problematic

print(f'Smoothed log-likelihood of possible sequence: {sequence_log_likelihood(possible_sequence, smoothed_cpt_df):.3f}')
```

```
Smoothed log-likelihood of possible sequence: -28.522
```

```
# But the real sequences are still much more likely
# (though much less so than in the unsmoothed model)
print(f'Smoothed likelihood of real sequence: {sequence_likelihood(real_sequence, smoothed_cpt_df):.20f}')
```

```
Smoothed likelihood of real sequence: 0.00000000000656707607
```

```
# So the relative value is easier to compare when we look at the log-likelihoods:

print(f'Smoothed log-likelihood of real sequence: {sequence_log_likelihood(real_sequence, smoothed_cpt_df):.3f}')
```

```
Smoothed log-likelihood of real sequence: -25.749
```

N-gram models

Unigram model: $P(w^{(1)} \dots w^{(N)}) = \prod_{i=1}^{N} P(w^{(i)})$

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

Trigram model: $P(w^{(1)} \dots w^{(N)}) = \prod_{i=1}^{N} P(w^{(i)} \mid w^{(i-1)}, w^{(i-2)})$

…and so on. But what's the problem? What stops us from conditioning on every previous token?

N-gram models

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

Bigram model: $$P(w^{(1)} \ldots w^{(N)}) = \prod_{i=1}^{N} P(w^{(i)} \,|\, w^{(i-1)})$$

Trigram model: $$P(w^{(1)} \ldots w^{(N)}) = \prod_{i=1}^{N} P(w^{(i)} \,|\, w^{(i-1)}, w^{(i-2)})$$

…and so on. But what's the problem? What stops us from conditioning on every previous token?

- data sparsity
- # of parameters

Concluding thoughts

Probabilistic modeling of text: count word occurrences and normalize to conditional probability distributions

Differing context sizes

- Unigrams: words occur independently
- Bigrams: words depend **only** on previous word
- Trigrams: words depend on previous two words
- etc.

Two important tasks

- Generate new text
- Assess text likelihood under model

Discussion question: how to do classification with these abilities?