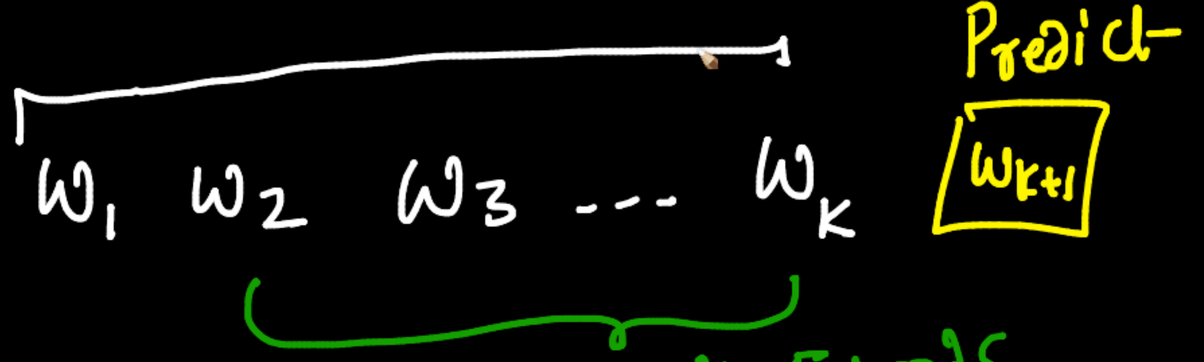
Chrome File Edit View History Bookmarks Profiles Tab Window Help 9:07

Task:



HINT: Probability

1 / 2

① $P(w_{k+1} \mid w_k\ w_{k-1}\ w_{k-2}\ w_{k-3}\ w_{k-4})$ — past 5 words

Classical NLP

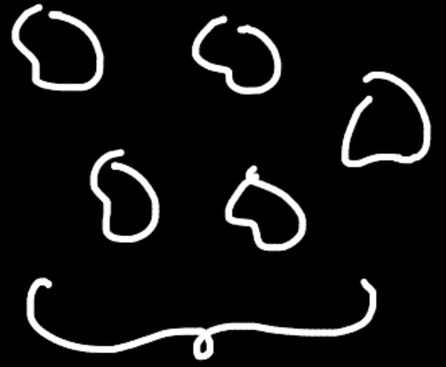
$V \cdot V \cdot V$ — fast + cheap

Voc: V

② Clustering:

↳ word|sent → vec (word2vec + TFIDF)

(?) $S_i \rightarrow V_i \longrightarrow$ clustering →

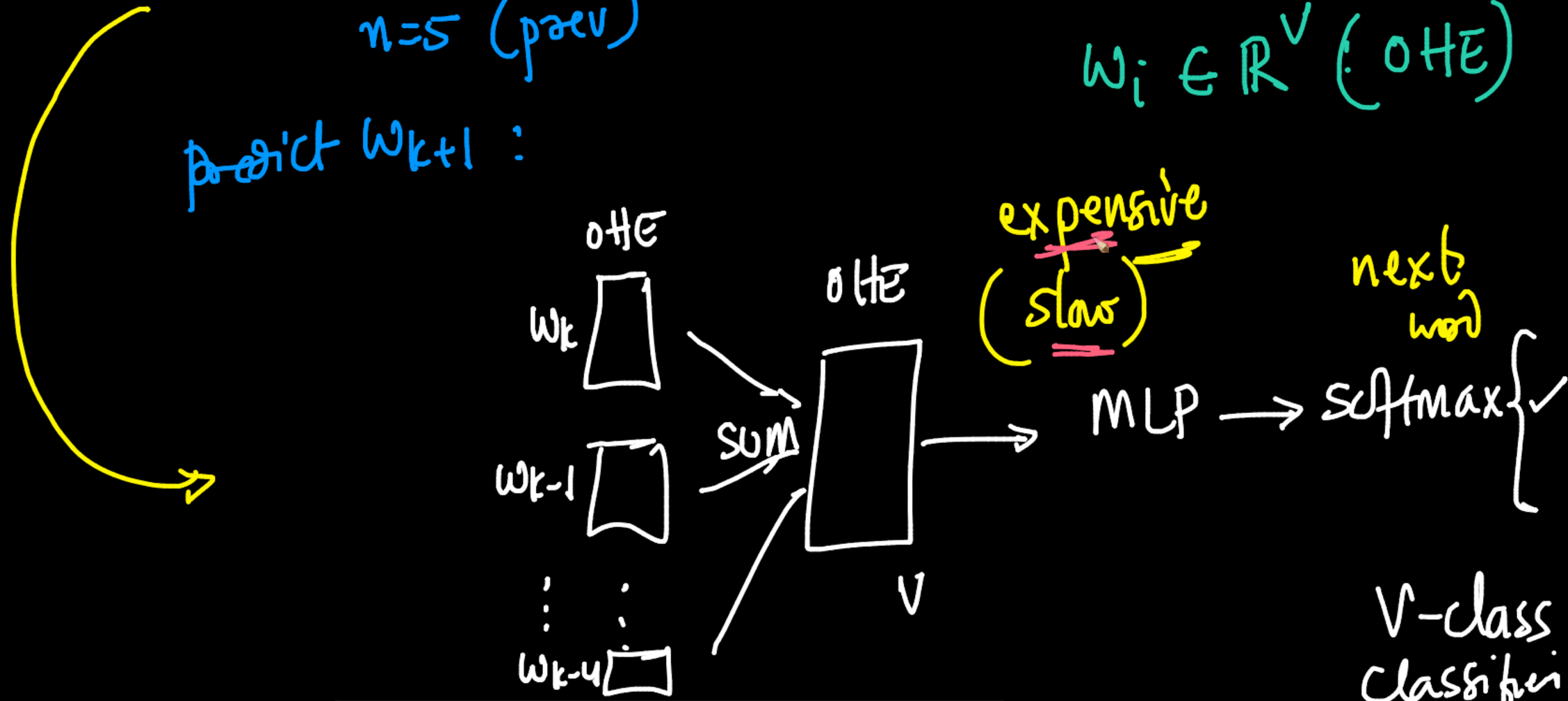


(3) multi-class classfn ✓ (MLP)

$n=5$ (prev)

predict w_{k+1} :

$w_i \in \mathbb{R}^V$ (OHE)



no Past $\rightarrow$ $P(w_{k+1}) = \frac{n_{k+1}}{\#\text{words}}$ (train data)

$n=1$ $\rightarrow$ $P(w_{k+1} \mid w_k) = \frac{P(w_{k+1}, w_k)}{P(w_k)}$

$= \frac{\#_{\text{count}}(w_{k+1}, w_k)}{\#_{\text{count}}(w_k)}$

Chrome File Edit View History Bookmarks Profiles Tab Window Help

Implementation

Key

Value

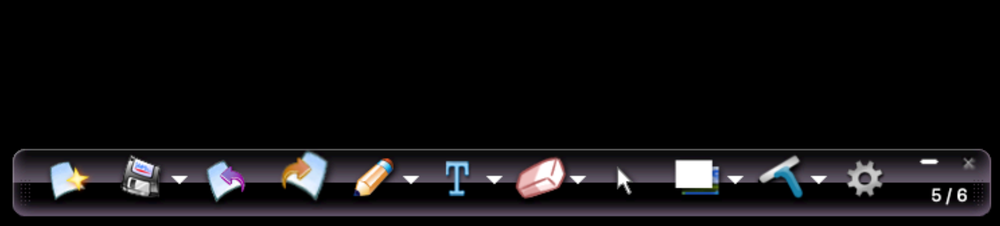
w_1

w_2

w_V

Cheap + fast

resume → w_i → attach → w_j with highest $P(w_j | w_i)$



e.g.: w_i = resume w_j = attached

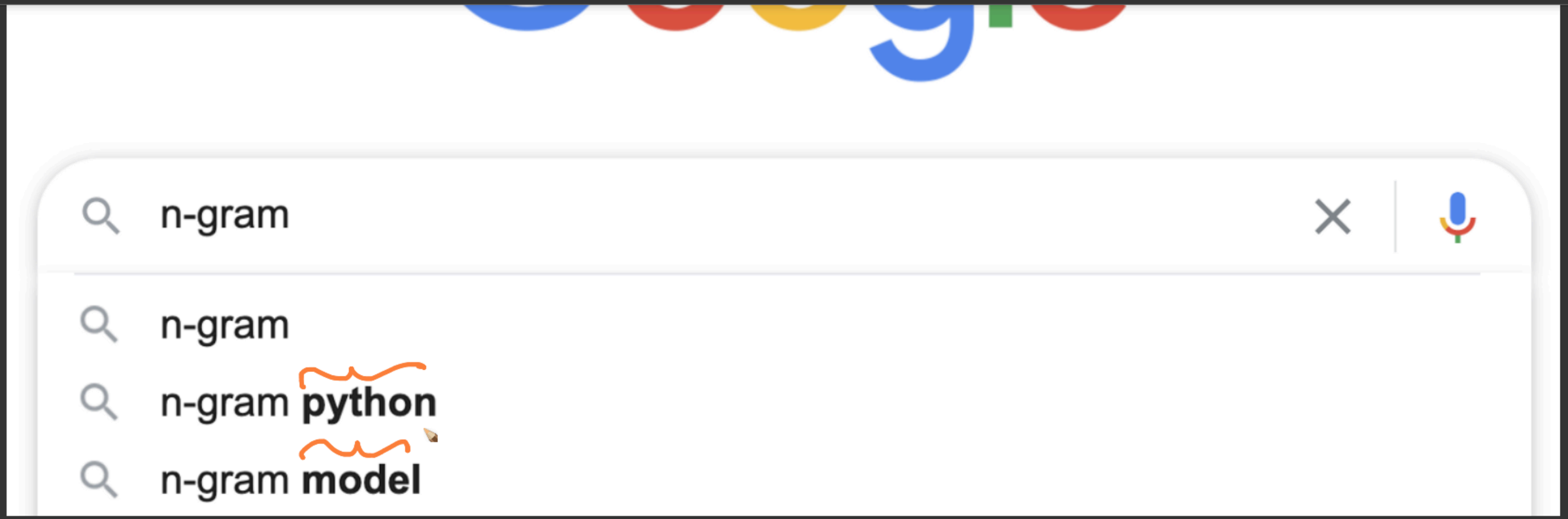
$P(w_1 | w_i)$

$P(w_2 | w_i)$

max $\leftarrow P(w_j = \text{attached} \mid w_i = \text{resume})$

$P(w_v | w_i)$

+ Code + Text Connect



How can we implement this feature ??

1.We can get an idea which word sequence came frequently in training data. for example we can see that in most cases "please find my CV" is followed by word "attached".

no Past → $P(w_{k+1})$ → Unigram

$n = 1$ → $P(w_{k+1} \mid w_k)$ → bi'gram

$n = 2$ → $P(w_{k+1} \mid w_k\ w_{k-1})$ → Tri'gram

$P(w_{k+1} \mid w_k\ w_{k-1} \cdots w_{k-n+1})$ → n-gram

$$P(w_{k+1} \mid \overbrace{w_k \; w_{k-1}}) = \frac{\#(w_{k+1}, w_k, w_{k-1})}{\#(w_k, w_{k-1})}$$

(1 over w_{k+1}, w_k, w_{k-1}; 2 over w_k, w_{k-1})

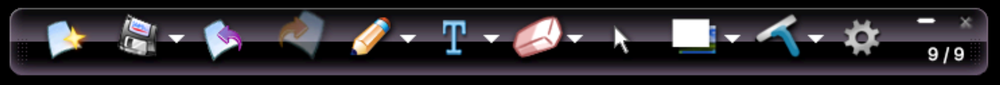
Problem?

→ Computationally expensive

→ Underflow: v. small probabilities

log Prob

→ Smoothing (Laplace)



- We can model this as a conditional probability, wherein, having seen a word we can predict the probability of next word.

Exploring probabilities of words with an example

- Let us predict the probability of the sentence - "its water is so transparent".
- Using chain rule, the probability can be calculated using:
  - P(its water is so transparent) = P(its) * P(water|its) * P(is|its water) * P(so|its water is) * P(transparent|its water is so)

Do we notice pitfalls with the above equation?

- We will have to calculate a lot of probabilities with as the size of sentence scales.

How about we simply estimate by counting and dividing the results?

- P(transparent | its water is so) = $\frac{count(itswaterissotransparent)}{count(itswaterisso)}$

Again, there are few obvious drawbacks with the above?

- There are far too many possible sentences in this method that would need to be calculated.
- This will result in very sparse data making results unreliable.
- Also, language is creative, any particular context might have not occured before.

+ Code + Text Connect

Should we see all the words typed to predict the possible next word ??

1.Find the probability of each word. Do not see any previous word, simply recommend the post probable word in the corpus. (**Unigram**)

> **P(I have a dream) = P(I) * p(have) * p(a) * p(dream)**

2. Find the probability of a set of two words (how the given two words come together). - **Bigram**

> **P(I have a dream) = P(I/) * p(have/I) * p(a/have) * p(dream/a)**

3. Find the proability of a set of three words i.e. next word will be recommended on the basis of previous two words - **Trigram**

> **P(I have a dream) = P(I/,) * p(have/, I) * p(a/i,have) * p(dream/have,a)**

4. Find the probabiliy of a set of n words i.e. next word will be recommended on the basis of last n-1 words - **n-gram**

Assume Let's assume we have a corpus of only 3 sentences. Let's try to frame the sentence: "I have a dream"

text
< start > I want a dream job </ start >
< start > I have a dog </ start >
< start > I have a dream company </ start >

Should we see all the words typed to predict the possible next word ??

1.Find the probability of each word. Do not see any previous word, simply recommend the post probable word in the corpus. (**Unigram**)

> **P(I have a dream) = P(I) * p(have) * p(a) * p(dream)**

2. Find the probability of a set of two words (how the given two words come together). - **Bigram**

> **P(I have a dream) = P(I/.) * p(have/I) * p(a/have) * p(dream/a)**

3. Find the proability of a set of three words i.e. next word will be recommended on the basis of previous two words - **Trigram**

> **P(I have a dream) = P(I/,) * p(have/, I) * p(a/i,have) * p(dream/have,a)**

4. Find the probabiliy of a set of n words i.e. next word will be recommended on the basis of last n-1 words - **n-gram**

Assume Let's assume we have a corpus of only 3 sentences. Let's try to frame the sentence: "I have a dream"

text
< start > I want a dream job </ start >
< start > I have a dog </ start >
< start > I have a dream company </ start >

Should we see all the words typed to predict the possible next word ??

1.Find the probability of each word. Do not see any previous word, simply recommend the post probable word in the corpus. (**Unigram**)

> **P(I have a dream) = P(I) * p(have) * p(a) * p(dream)**

2. Find the probability of a set of two words (how the given two words come together). - **Bigram**

> **P(I have a dream) = P(I/) * p(have/I) * p(a/have) * p(dream/a)**

3. Find the proability of a set of three words i.e. next word will be recommended on the basis of previous two words - **Trigram**

> **P(I have a dream) = P(I/,) * p(have/, I) * p(a/i,have) * p(dream/have,a)**

4. Find the probabiliy of a set of n words i.e. next word will be recommended on the basis of last n-1 words - **n-gram**

Assume Let's assume we have a corpus of only 3 sentences. Let's try to frame the sentence: "I have a dream"

text
< start > I want a dream job </ start >
< start > I have a dog </ start >
< start > I have a dream company </ start >

+ Code + Text Connect

Should we see all the words typed to predict the possible next word ??

1.Find the probability of each word. Do not see any previous word, simply recommend the post probable word in the corpus. (**Unigram**)

> **P(I have a dream) = P(I) * p(have) * p(a) * p(dream)**

2. Find the probability of a set of two words (how the given two words come together). - **Bigram**

> **P(I have a dream) = P(I/) * p(have/I) * p(a/have) * p(dream/a)**

3. Find the proability of a set of three words i.e. next word will be recommended on the basis of previous two words - **Trigram**

> **P(I have a dream) = P(I/,) * p(have/, I) * p(a/i,have) * p(dream/have,a)**

4. Find the probabiliy of a set of n words i.e. next word will be recommended on the basis of last n-1 words - **n-gram**

n-gram PLMs

Assume Let's assume we have a corpus of only 3 sentences. Let's try to frame the sentence: "I have a dream"

text
< start > I want a dream job </ start >
< start > I have a dog </ start >
< start > I have a dream company </ start >

Q

p one-time

$\log P(w_{k+1} \mid \text{last } 5 \text{ words})$ = bg-likelihood

K	V
last 5-words	w_{k+1}

Problem:

size of dict

v.v.large

✓ { disb-dict → Memcache Redis

Assume Let's assume we have a corpus of only 3 sentences. Let's try to frame the sentence: "I have a dream"

text
< start > I want a dream job </ start >
< start > I have a dog </ start >
< start > I have a dream company </ start >

<end>

Uni-gram Model

- **Unigram** - For uni-gram models we do not look at the previous words to predict the next word.
- Each word probability is calculated to predict the occurance

Count table

< start >	I	want	a	dream	job	have	dog	company	< / start >
3/20	3/20	1/20	3/20	2/20	1/20	2/20	1/20	1/20	3/20

- Calculate the probabilities of occurance of each word in the corpus.

Probability

< start >	I	want	a	dream	job	have	dog	company	< / start >
0.15	0.15	0.05	0.15	0.1	0.05	0.1	0.05	0.05	0.15

Uni-gram Model

- **Unigram** - For uni-gram models we do not look at the previous words to predict the next word.
- Each word probability is calculated to predict the occurance

Count table

< start >	I	want	a	dream	job	have	dog	company	< / start >
3/20	3/20	1/20	3/20	2/20	1/20	2/20	1/20	1/20	3/20

P(wi)

- Calculate the probabilities of occurance of each word in the corpus.

Probability

< start >	I	want	a	dream	job	have	dog	company	< / start >
0.15	0.15	0.05	0.15	0.1	0.05	0.1	0.05	0.05	0.15

> **P(I have a dream) = P(I) * p(have) * p(a) * p(dream)**

2. **Bigram** - For Bi-gram models we calculate what is the probability of occurance of a word given the previous word.

count table

spell check

↳ $P(\text{wall} \mid \text{wlal})$ (1)

$P(\text{l} \mid \text{wa})$ (2)

char-level model

We will build a N-gram model that will learn word probabilities using MLE

What is MLE?

- MLE or Maximum Likelihood Estimator is a technique to estimate the parameters of an assumed probability, given some observed data.
- We can compute a particular bi-gram probability of a word w_n, given the previous word w_{n-1}, by computing the count of bigram C($w_{n-1}\,w_n$ and normalize by the sum of all bigrams that share the first word w_{n-1}.
  - $P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(W_{n-1}w)}$
- Note - Since sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count of that word, we can simplify the above equation to:
  - $P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(W_{n-1})}$
- The above calculation can be extended to other n-grams as well.

Unigram Model

Evaluating the model

How can we find out the probability of the sentence using word probabilities??

- **naive assumption** - Each word in the sentence is independent of each other **(markov assumption)**
- each sentence probability is the product of word probabilities.

$$P_{eval}(\text{i have a dream [END]}) = P_{train}(\text{i})\, P_{train}(\text{have} \mid \text{i})\, P_{train}(\text{a} \mid \text{i have})\, P_{train}(\text{dream} \mid \text{i have a})\, P_{train}([\text{END}] \mid \text{i have a dream})$$
$$= P_{train}(\text{i})\, P_{train}(\text{have})\, P_{train}(\text{a})\, P_{train}(\text{dream})\, P_{train}([\text{END}])$$

Now How can we find out the probability of entire email text (having multiple sentences)??

- **Naive Assumption** - Each sentence is independendt of from other sentences (same as words)
- The Probability of entire text is the product of probabilities of sentences

$$\text{Evaluation text}: \quad \text{"I have a dream. We are free at last."}$$
$$P_{eval}(\text{text}) = P_{eval}(\text{i have a dream [END]})\, P_{eval}(\text{we are free at last [END]})$$
$$P_{eval}(\text{i have a dream [END]}) = P_{train}(\text{i})\, P_{train}(\text{have})\, P_{train}(\text{a})\, P_{train}(\text{dream})\, P_{train}([\text{END}])$$
$$P_{eval}(\text{we are free at last [END]}) = P_{train}(\text{we})\, P_{train}(\text{are})\, P_{train}(\text{free})\, P_{train}(\text{at})\, P_{train}(\text{last})\, P_{train}([\text{END}])$$

+ Code + Text Connect

Evaluating the model

$\sum \log P(i)$

Underflow

How can we find out the probability of the sentence using word probabilities??

- **naive assumption** - Each word in the sentence is independent of each other (**markov assumption**)
- each sentence probability is the product of word probabilities.

$$P_{eval}(\text{i have a dream [END]}) = P_{train}(\text{i})\, P_{train}(\text{have} \mid \text{i})\, P_{train}(\text{a} \mid \text{i have})\, P_{train}(\text{dream} \mid \text{i have a})\, P_{train}(\text{[END]} \mid \text{i have a dream})$$
$$= P_{train}(\text{i})\, P_{train}(\text{have})\, P_{train}(\text{a})\, P_{train}(\text{dream})\, P_{train}(\text{[END]})$$

Now How can we find out the probability of entire email text (having multiple sentences)??

- **Naive Assumption** - Each sentence is independendt of from other sentences (same as words)
- The Probability of entire text is the product of probabilities of sentences

$$\text{Evaluation text}: \quad \text{"I have a dream. We are free at last."}$$
$$P_{eval}(\text{text}) = P_{eval}(\text{i have a dream [END]})\, P_{eval}(\text{we are free at last [END]})$$
$$P_{eval}(\text{i have a dream [END]}) = P_{train}(\text{i})\, P_{train}(\text{have})\, P_{train}(\text{a})\, P_{train}(\text{dream})\, P_{train}(\text{[END]})$$
$$P_{eval}(\text{we are free at last [END]}) = P_{train}(\text{we})\, P_{train}(\text{are})\, P_{train}(\text{free})\, P_{train}(\text{at})\, P_{train}(\text{last})\, P_{train}(\text{[END]})$$

what is the issue with log Probability??

How to deal with unknown unigrams?? - For a unigram that appears in the evaluation text but not in the training text, its count in the training text — hence its probability — will be zero.

AND

the log of this zero probability is negative infinity, leading to a negative infinity average log likelihood for the entire model

$$n_{train}(unigram) = 0$$

$$\implies P_{train}(unigram) = \frac{0}{N_{train}} = 0$$

$$\implies \log(P_{train}(unigram)) = \log(0) = -\infty$$

#Comment: Words not appearing in the trainset but present in the eval set is not a problem only specific to using log. Kindly mention that because of new words in the eval set the probaility will become zero so the entire sentence probailty becomes zero. Then mention the log(0)

1. Add a pseudo-count of k to all the unigrams in our vocabulary.

> This will make the count of [UNK] k from 0 (0+K)

> The probability of word appearing in eval set but not in training set will not have 0 probability now, why ??

2. The most common value of k is 1, and this goes by the intuitive name of “add-one smoothing”.

$$P_{train}(unigram) = \frac{n_{train}(unigram) + k}{N_{train} + kV}$$

unigram vocabulary size

> The probability of word appearing in eval set but not in training set will not have 0 probability now, why ??

2. The most common value of k is 1, and this goes by the intuitive name of “add-one smoothing”.

$$P_{train}(unigram) = \frac{n_{train}(unigram) + k}{N_{train} + kV}$$

unigram vocabulary size

(number of unique unigrams in training text)

The probability of word appearing in eval set but not in training set will not have 0 probability now, why ??

2. The most common value of k is 1, and this goes by the intuitive name of “add-one smoothing”.

stethescope

$$P_{train}(unigram) = \frac{n_{train}(unigram) + k}{N_{train} + kV}$$

$n_{train}(unigram) \rightarrow 0$

$k = 1$

$kV \rightarrow 1$

unigram vocabulary size

(number of unique unigrams in training text)

Effect of Laplace smoothing

Because of the additional pseudo-count k to each unigram, each time the unigram model encounters an unknown word in the evaluation text, it will convert said unigram to the unigram [UNK].

$$P_{train}([UNK]) = \frac{k}{N_{train} + kV}$$

N-Gram

- The probability of each word depends on the n-1 words before it. For a trigram model (n = 3), for example, each word's probability depends on the 2 words immediately before it.
- This probability is estimated as the fraction of times this n-gram appears among all the previous (n-1)-grams in the training set. In other words, training the n-gram model is nothing but calculating these conditional probabilities from the training text.

Markov property: → bi'gram

HMM ✗

w_1 ✓ w_2 ✓ w_3 ✓ w_4

$P(w_4 | w_3)$

simple assumption

✓ $\{ P(w_4 | w_2 w_3 w_1) \}$

Global

P(baby | ba)

↓

low

bath soap

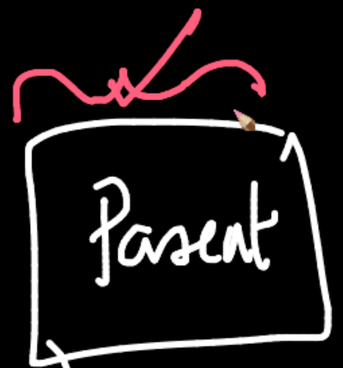
+

personalized

P(baby | ba)

↓

high ✓



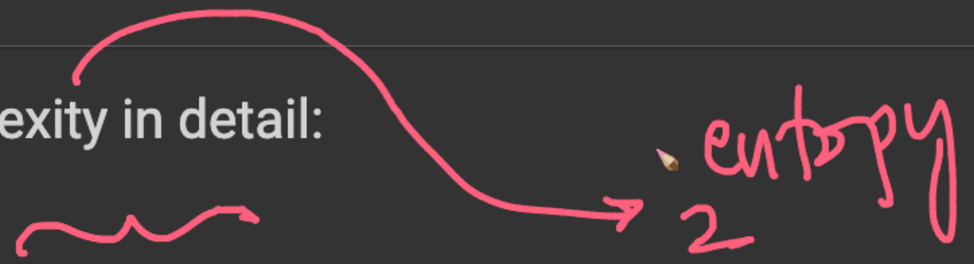
- Perplexity of a language model is the **inverse probability** of the test set, **normalized** by the number of words.

Let us understand Perplexity in detail:

For a test set W; containing words $w_1, w_2, w_3, \ldots, w_n$

- The perplexity can be derived using:
  - PP(W) = $P(w_1, w_2, w_3, \ldots, w_n)^{-1/N}$
  - $\sqrt[n]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_n)}}$
- Using chain rule of probability, we can expand:
  - PP(W) = $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots . w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.

Let us understand Perplexity in detail:



For a test set W; containing words $w_1, w_2, w_3, \ldots, w_n$

- The perplexity can be derived using:
  - PP(W) = $P(w_1, w_2, w_3, \ldots, w_n)^{-1/N}$
  - $\sqrt[n]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_n)}}$
- Using chain rule of probability, we can expand:
  - PP(W) = $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots . w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.

```
[ ]  def perplexity(n_gram, input_sentence):
         tokenized_sentence = input_sentence.split()
         pp_w = []
```

Let us understand Perplexity in detail:

For a test set W; containing words $w_1, w_2, w_3, \ldots, w_n$

- The perplexity can be derived using:
  - PP(W) = $P(w_1, w_2, w_3, \ldots, w_n)^{-1/N}$
  - $\sqrt[n]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_n)}}$
- Using chain rule of probability, we can expand:
  - PP(W) = $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots . w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.

```
[ ]  def perplexity(n_gram, input_sentence):
         tokenized_sentence = input_sentence.split()
         pp_w = []
```

Let us understand Perplexity in detail:

For a test set W; containing words $w_1, w_2, w_3, \ldots, w_n$

- The perplexity can be derived using:
  - PP(W) = $P(w_1, w_2, w_3, \ldots, w_n)^{-1/N}$
  - $\sqrt[n]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_n)}}$
- Using chain rule of probability, we can expand:
  - PP(W) = $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots . w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.

inverse

$P(w_1 w_2) = P(w_1) P(w_2)$

$P(w_1 | w_2) = P(w_1) P(w_1 | w_2)$

$P(w_1 w_2 w_3) = P(w_1)$

$P(w_1 | w_2)$
$P(w_2 | w_3)$

```
[ ]  def perplexity(n_gram, input_sentence):
         tokenized_sentence = input_sentence.split()
         pp_w = []
```

Let us understand Perplexity in detail:

For a test set W; containing words $w_1, w_2, w_3, \ldots, w_n$

- The perplexity can be derived using:
  - PP(W) = $P(w_1, w_2, w_3, \ldots, w_n)^{-1/N}$
  - $\sqrt[n]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_n)}}$
- Using chain rule of probability, we can expand:
  - PP(W) = $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots . w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.

```
[ ]  def perplexity(n_gram, input_sentence):
         tokenized_sentence = input_sentence.split()
         pp_w = []
```

Let us understand Perplexity in detail:

For a test set W; containing words $w_1, w_2, w_3, \ldots, w_n$

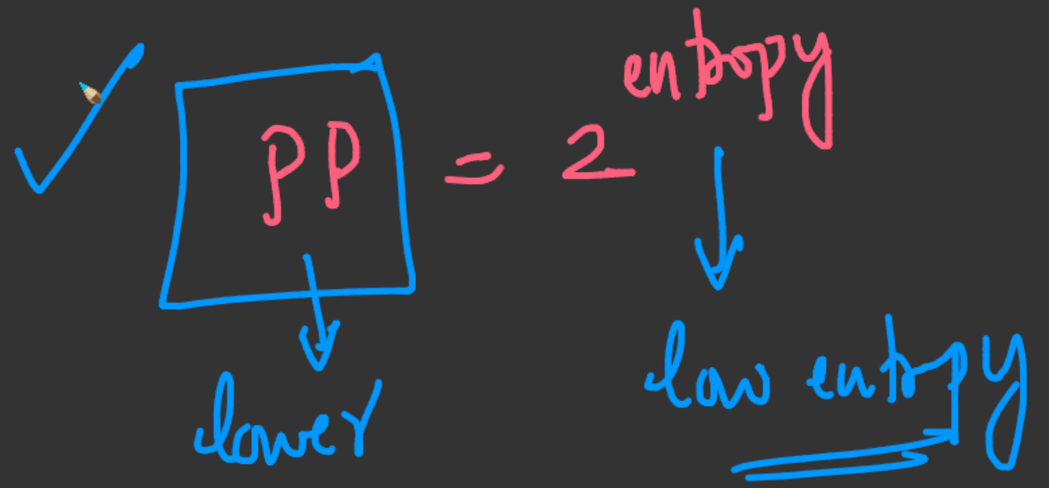
- The perplexity can be derived using:
  - PP(W) = $P(w_1, w_2, w_3, \ldots, w_n)^{-1/N}$
  - $\sqrt[n]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_n)}}$
- Using chain rule of probability, we can expand:
  - PP(W) = $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots . w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.

```
[ ]  def perplexity(n_gram, input_sentence):
         tokenized_sentence = input_sentence.split()
         pp_w = []
```

Let us understand Perplexity in detail:

For a test set W; containing words $w_1, w_2, w_3, \ldots, w_n$

- The perplexity can be derived using:
  - PP(W) = $P(w_1, w_2, w_3, \ldots, w_n)^{-1/N}$
  - $\sqrt[n]{\frac{1}{P(w_1, w_2, w_3, \ldots, w_n)}}$
- Using chain rule of probability, we can expand:
  - PP(W) = $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots . w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.




```
[ ]  def perplexity(n_gram, input_sentence):
         tokenized_sentence = input_sentence.split()
         pp_w = []
```

- $\sqrt[n]{\frac{1}{P(w_1,w_2,w_3,\ldots,w_n)}}$

- Using chain rule of probability, we can expand:
  - $PP(W) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}}$
- For a bi-gram model, the perplexity of W can be calculated using:
  - $\sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_{i-1})}}$
- **Lower** the **perplexity** score, **higher** the model's **understanding** of the language.

$PP = 2^{entropy}$

```
[ ] def perplexity(n_gram, input_sentence):
        tokenized_sentence = input_sentence.split()
        pp_w = []
        N = len(tokenized_sentence)

        if n_gram == 2:
            tokenized_sentence.insert(0, None)
            tokenized_sentence.append(None)
            for i in tokenized_sentence[:-1]:
                try:
                    pp_w.append(1 / (0.0001+bigram[i][tokenized_sentence[tokenized_sentence.index(i)+1]]))
                except:
                    continue
```

- $P^*_{Add-k}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV}$

$P(w_n|w_{n-2}w_{n-1}) = \frac{\#(w_n, w_{n-1}, w_{n-2})}{\#(w_{n-2}, w_{n-1})}$

Backoff and Interpolation

- Most applied technique for OOV and produces the best results when compared to smoothing techniques.
- Instead of smoothing and shifting the distribution, we can rely on the knowledge we have already computed.
- In this technique, if we have no examples of a particular tri-gram $P(w_n|w_{n-2}wn-1)$, we can simply estimate its probability by using the bigram probability $P(w_n|wn-1)$.
- In case we dont have counts to compute probability of the bi-gram, we can cascade and use the probability of the unigram $P(w_n)$.
- Using interpolation helps in generalization across different contexts.
- Techniques to make the interpolation effective:
  - Linear interpolation - Probabilities across each n-gram will have same weightage.
  - "λ" interpolation - Weights are assigned to the probabilities based on their computation method.
    - Tri-gram probabilites will have the highest weights, followed by bi-gram and uni-gram.

$P(w_n|w_{n-2}wn-1) = \lambda 1 P(w_n) + \lambda 2 P(w_n|w_{n-1}) + \lambda 3 P(w_n|w_{n-2}w_{n-1})$

Unigram bigram Trigrams

What are the limitations of N-gram approach to Language Modeling?

(e.g)

$$p(\text{click} \mid \underbrace{\text{Hitechcity}}_{\text{loc}}) = \frac{2}{1000}$$

$$\downarrow \text{backoff}$$

$$p(\text{clk} \mid \text{Hyd}) = \frac{200}{10000}$$

- $P^*_{Add-k}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV}$

Backoff and Interpolation

- Most applied technique for OOV and produces the best results when compared to smoothing techniques.
- Instead of smoothing and shifting the distribution, we can rely on the knowledge we have already computed.
- In this technique, if we have no examples of a particular tri-gram P($w_n|w_{n-2}wn-1$) , we can simply estimate its probability by using the bigram probability P($w_n|wn-1$).
- In case we dont have counts to compute probability of the bi-gram, we can cascade and use the probability of the unigram P(w_n).
- Using interpolation helps in generalization across different contexts.
- Techniques to make the interpolation effective:
  - Linear interpolation - Probabilities across each n-gram will have same weightage.
  - "λ" interpolation - Weights are assigned to the probabilities based on their computation method.
    - Tri-gram probabilites will have the highest weights, followed by bi-gram and uni-gram.
      P($w_n|w_{n-2}wn-1$) = $\lambda 1 P(w_n) + \lambda 2 P(w_n|w_{n-1}) + \lambda 3 P(w_n|w_{n-2}w_{n-1})$

What are the limitations of N-gram approach to Language Modeling?

- Tri gram probabilites will have the highest weights, followed by bi gram and uni gram.

$$P(w_n|w_{n-2}wn-1) = \lambda 1 P(w_n) + \lambda 2 P(w_n|w_{n-1}) + \lambda 3 P(w_n|w_{n-2}w_{n-1})$$

What are the limitations of N-gram approach to Language Modeling?

- Higher the N, the better is the model. But this leads to high computation overhead and would require large computation power, which may not be easily available.
- N-grams are a sparse representation of language. This is because we build the model based on the probability of words co-occurring. Zero probabilities are assigned to words that are not present in the training corpus.
- Even with techniques to build non-zero probability models, the results may not be accurate in case of generalization of words over multiple contexts.

We can overcome the limitations of N-gram based Lanugage Models by using Neural Language Models.

- Neural Language Models (NLM) address the n-gram issues through parameterization of words as vectors (word embeddings) and using them as inputs to a neural network.
- We could use robust architectures like RNN and Encoders to understand word contexts better, there-by being more accurate in predicting the next set of words

multiple contexts.

We can overcome the limitations of N-gram based Lanugage Models by using Neural Language Models.

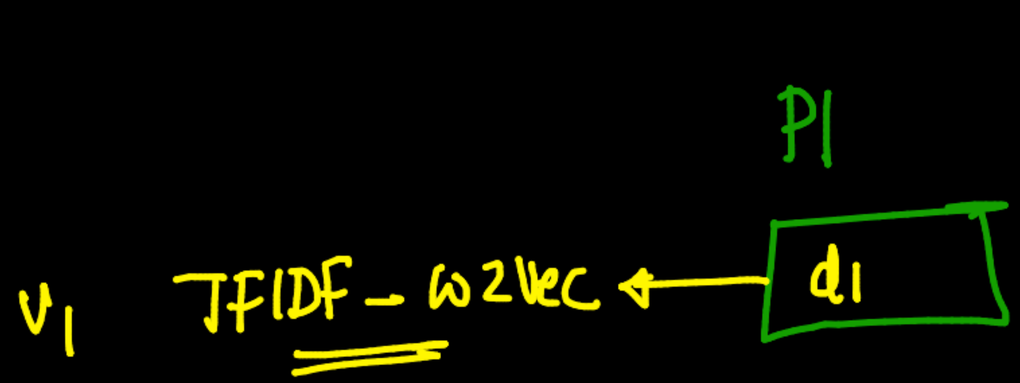
- Neural Language Models (NLM) address the n-gram issues through parameterization of words as vectors (word embeddings) and using them as inputs to a neural network.
- We could use robust architectures like RNN and Encoders to understand word contexts better, there-by being more accurate in predicting the next set of words.

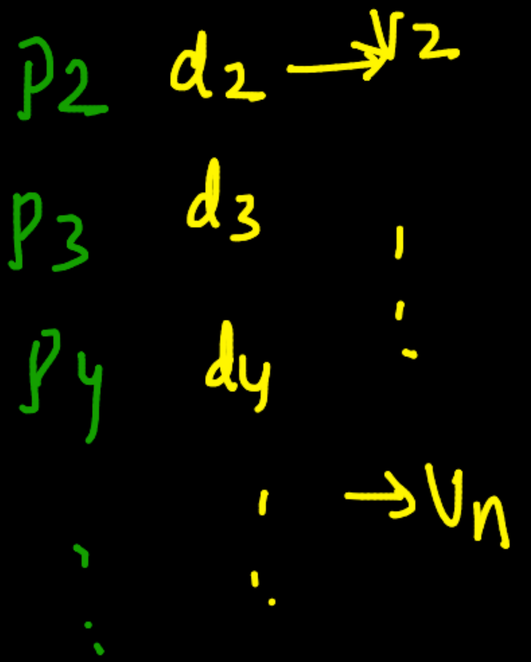
LSTM | GRU

Conclusion:

- We have modelled the problem of helping people write emails faster as "next-word" prediction problem.
- We have explored Probabilistic Language Models.
- We auto-completed the sentence by building bi-gram and tri-gram language models.
- We explored Perplexity as a technique to ascertain the LM's understanding.
- We saw how Smoothing and interpolation techniques can help over-come the dis-advantages of OOV.

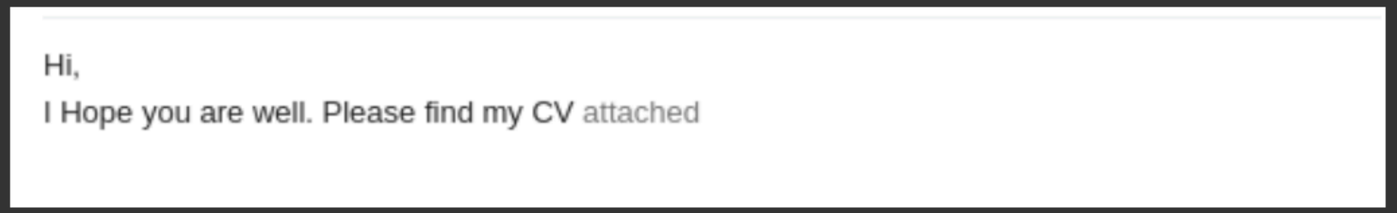
Content-based:





Email autofill

*You are working as machine learning engineer at Gmail team at Google. Your manager want you to design a solution to suggest the next word in email when they are typing it, to improve their experience and save time. Example as we can see in the below image:-



What are the other real life application of autofill? 1.typing in whatsapp (auto suggest three possible next words) 2.Autofill Google Search