ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

n-gram language models

2024–25

Ion Androutsopoulos

http://www.aueb.gr/users/ion/

These slides are partly based on material from the book *Speech and Language Processing* by D. Jurafsky και J.H. Martin, 2η edition, Pearson Education, 2009.

Contents

- *n*-gram language models.
- Estimating probabilities from corpora.
- Entropy, cross-entropy, perplexity.
- Edit distance.
- Context-aware spelling correction.
- Beam-search decoding.

Language models

- **How probable** is it to encounter (e.g., in news articles) the following sentences (word sequences)?
  - *The government announcement new austerity metrics hopping to decrease the deficit.*
  - *The government announced new austerity measures hoping to reduce the deficit.*
- In many cases, **candidate alternative sentences** are produced. We wish to keep the **most probable** ones.
  - Speech recognition, optical character recognition, machine translation, smartphone keyboards, spelling and syntax checking, text normalization of social media posts…

n-gram language models

- **Notation** for sequences of words:

$$\langle w_1, w_2, \ldots, w_k \rangle = w_1^k$$

- ***n*-gram**: sequence of ***n*** **consecutive words**.
  - **Trigrams**: “the government announced”, “government announced new”, “announced new austerity”, …
  - **Bigrams**: “the government”, “government announced”, “announced new”, “new austerity”, …
  - In other cases, sequences of *n* **consecutive characters**.
- **Chain rule**:

$$P(w_1^k) = P(w_1, \ldots, w_k) = P(w_1) \cdot P(w_2 \mid w_1) \cdot$$

$$P(w_3 \mid w_1, w_2) \cdot P(w_4 \mid w_1^3) \cdots P(w_k \mid w_1^{k-1})$$

How do we estimate the probabilities?

- Simplest approach: **maximum likelihood estimates** from a **corpus** of *C* tokens:

$$P_{MLE}(\text{the}) = \frac{c(\text{the})}{C}$$

$$P_{MLE}(\text{government} \mid \text{the}) = \frac{c(\text{the}, \text{government})}{c(\text{the})}$$

$$P_{MLE}(\text{announced} \mid \text{the}, \text{gov}) = \frac{c(\text{the}, \text{gov}, \text{announced})}{c(\text{the}, \text{gov})}$$

- Many *n*-grams (esp. 4-grams, 5-grams, …) will be **very rare** or **may not occur** even in a **large corpus**.
  - Very **poor** or **zero** probability **estimates**.
  - Leading to a **zero chain product**…

Markov assumption

- **Bigram** language model:

$$P(w_1^k) = P(w_1, \ldots, w_k) = P(w_1) \cdot P(w_2 \mid w_1) \cdot$$

$$P(w_3 \mid w_1, w_2) \cdot P(w_4 \mid w_1^3) \cdots P(w_k \mid w_1^{k-1}) \simeq$$

$$\boxed{P(w_1 \mid start) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_2) \cdots P(w_k \mid w_{k-1})}$$

- **Trigram** language model:

$$\boxed{\begin{gathered} P(w_1 \mid start_1, start_2) \cdot P(w_2 \mid start_2, w_1) \cdot P(w_3 \mid w_1, w_2) \cdot \\ P(w_4 \mid w_2, w_3) \cdots P(w_k \mid w_{k-2}, w_{k-1}) \end{gathered}}$$

- **Stationarity**: We assumed probabilities do not depend on where the n-grams are encountered. E.g., in P(announced | the, government), we do not examine if "announced" occurs as the 3rd or 4th or … word in the sentences of the corpus.
- Strictly speaking, we also need an ***end*** pseudo-token. See study exercises.

Laplace smoothing

- Even with a Markov assumption, we will still have many ***n*-grams** that **do not occur in the corpus**.
- **Laplace smoothing** for **unigrams:** if we have $|V|$ vocabulary words (distinct words),

$$P_{Laplace}(W = w) = \frac{c(w)+1}{C+|V|}$$

Add a pseudo-occurrence of each vocabulary word. More generally, of each possible value of the random variable (here W).

- Similarly, e.g., for **trigrams**:

$$P_{Laplace}(W_k = w_k \mid w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k)+1}{c(w_{k-2}, w_{k-1})+|V|}$$

Add a pseudo-occurrence of each possible trigram that starts with w_{k-2}, w_{k-1}. There are $|V|$ such trigrams in total.

- But we **over-estimate** rare bigrams, trigrams, …

Add-α smoothing

- For **unigrams:** if we have $|V|$ vocabulary words,

$$P_{Laplace}(W = w) = \frac{c(w) + \alpha}{C + \alpha \cdot |V|}$$

We tune a $(0 \leq a \leq 1)$ on held-out data (see below).

- Similarly, e.g., for **trigrams**:

$$P_{Laplace}(W_k = w_k \mid w_{k-2}, w_{k-1}) = \frac{c(w_{k-2}, w_{k-1}, w_k) + \alpha}{c(w_{k-2}, w_{k-1}) + \alpha \cdot |V|}$$

- Better, but still **poor estimates** for language models.
  - In practice, Laplace and add-α smoothing are **not used in language models** (but often work well in classification tasks).
  - See **optional reading slides for better estimates** for *n*-gram LMs (e.g., Knesser-Ney smoothing, backoff models).

Linear interpolation

- We use a **linear combination** of estimates from ***n*-gram language models** with different *n* values.

$$P_{\text{int}}(w_k \mid w_{k-2}, w_{k-1}) = \lambda_1 \cdot P(w_k \mid w_{k-2}, w_{k-1}) +$$

$$\lambda_2 \cdot P(w_k \mid w_{k-1}) + \lambda_3 \cdot P(w_k) \quad \text{with} \sum_{i=1}^{3} \lambda_i = 1$$

LMs as next word predictors

- **Sequence probability** using a bigram LM:

$$P(w_1^k) = P(w_1, \ldots, w_k) = P(w_1) \cdot P(w_2 \mid w_1) \cdot$$

$$P(w_3 \mid w_1, w_2) \cdot P(w_4 \mid w_1^3) \cdots P(w_k \mid w_1^{k-1}) \simeq$$

$$\boxed{P(w_1 \mid start) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_2) \cdots P(w_k \mid w_{k-1})}$$

- We can think of the **LM** as a system that **provides the probabilities $\boldsymbol{P(w_i|w_{i-1})}$**, which we then multiply.
  - Or the probabilities $\boldsymbol{P(w_i|w_{i-2}, w_{i-1})}$ for a **trigram LM**.
  - Or the probabilities $\boldsymbol{P(w_i|h_1^{i-1})}$ for an LM that considers all the **"history" (previous words) $\boldsymbol{h_1^{i-1}}$**, e.g., in an **RNN LM**.
- An **LM** provides a **distribution $\boldsymbol{P(w|h_1^{i-1})}$** showing how probable it is for **every word** $w \in V$ to be the next one.

Spelling correction/normalization

- The words we see:

  He pls gd ftball.

- Possible candidate corrections:

  He please god football.

  He plays god football.

  He plays good football.

  He players good football.

  …

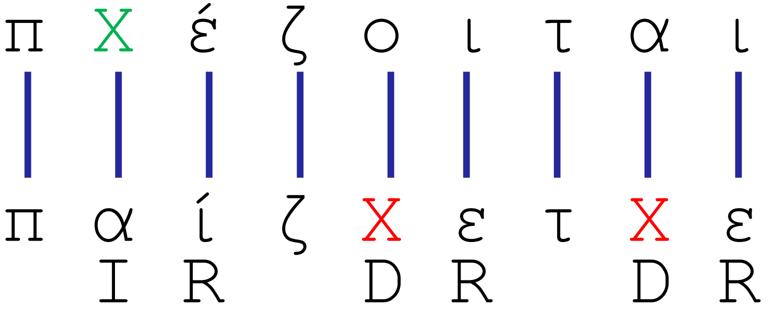
  He pleases god ball.

The **green words** are **vocabulary words** with **small distance** (e.g., Levenshtein) from the out-of-vocabulary words.

A **language model** estimates **how well** the **words** of each candidate sequence **fit together**.

Edit distance

- Input: **two strings** (e.g., words from tweet and dictionary).
- What is the **total minimum cost** to **convert one input string to the other**, using particular operators?
- **Levenshtein** distance (one possible edit distance):
  - Operators: **insert** (I, cost **1**), **delete** (D, cost **1**), **replace** (R, cost **2**). Other work may set the cost of R to 1.
- When converting from **Greeklish to Greek**, we may want to set, for example, $R(e, \varepsilon) < R(e, \alpha)$.



π X έ ζ ο ι τ α ι

π α ί ζ X ε τ X ε

I R D R D R

We also get an **alignment** of characters. Similarly, we can compute the edit distance and alignment of the **words of two sentences**, by applying I, D, R to words instead of characters.

Actually two types of errors…

- The **wrong words** may actually be **vocabulary words**!
  - 1st type: “he plays good football” → “he pls gd ftball”.
  - 2nd type: “he plays good football” → “he please god ftball”.
- Let’s **continue to focus** on the **1st type** for the moment.
  - The wrong words are all **out of vocabulary words** (e.g., words that do not occur at least 10 times in a large corpus).
- For each wrong word, get candidate corrections:
  - Simplest case: get vocabulary words at a **small Levenshtein distance** from the wrong word.
  - Alternatively use an edit distance that takes into account the **keyboard layout**, the **visual similarity** of characters etc., possibly modifying the Replace operator accordingly.

Correcting errors of the 1st type

- The words we see:

  w_1^k: He pls gd ftball.

- Possible candidate corrections:

  t_1^k: He please god football.

  t_1^k: He plays god football.

  t_1^k: He plays good football.

  t_1^k: He players good football.

  …

  t_1^k: He pleases god ball.

The **green words** are **vocabulary words** with **small distance** (e.g., Levenshtein) from the out-of-vocabulary words.

A **language model** estimates **how well** the **words** of each candidate sequence **fit together**.

More to be discussed…

- Exactly how do we **combine** the **edit distances** with a **language model** to correct **errors of the 1st type**?
- How do we correct **errors of the 2nd type**?
- How do we **evaluate** a **language model**?
  - Among different language models (e.g., using different *n* or smoothing), which one is the best?

A noisy channel model

- We assume that **all the words** were **initially correct**, but were **transmitted** through a **noisy channel**.
  - Here the channel distorts the words by occasionally **inserting**, **deleting**, or **replacing** letters.
- We try to **guess the initial** (correct) words **from the observed** ones.
  - Initial (correct) words: $t_1^k = \langle t_1, t_2, \ldots, t_k \rangle$

    channel

  - Observed words: $w_1^k = \langle w_1, w_2, \ldots, w_k \rangle$
- We seek the **most probable initial words**:

$$\hat{t}_1^k = \arg\max_{t_1^k} P(t_1^k \mid w_1^k) = \arg\max_{t_1^k} \frac{P(t_1^k) \cdot P(w_1^k \mid t_1^k)}{\cancel{P(w_1^k)}}$$

The most probable initial words

- For each observed sequence (e.g., sentence) w_1^k:

$$\hat{t}_1^k = \arg\max_{t_1^k} P(t_1^k \mid w_1^k) = \arg\max_{t_1^k} P(t_1^k) \cdot P(w_1^k \mid t_1^k)$$

In each **candidate sequence $\boldsymbol{t_1^k}$**, every **wrong word** has been replaced by a **vocabulary word** at a **small distance** from the wrong one.

Language model
(e.g., trigram model)

Simplest approach: probabilities **inversely proportional to edit distance** (with normalization). See **J&M for better ideas**.

$$P(w_1^k \mid t_1^k) = P(w_1 \mid t_1^k) \cdot P(w_2 \mid w_1, t_1^k) \cdot \ldots P(w_k \mid w_1^{k-1}, t_1^k)$$

$$\simeq P(w_1 \mid t_1) \cdot P(w_2 \mid t_2) \cdot \ldots P(w_k \mid t_k) = \prod_{i=1}^{k} P(w_i \mid t_i)$$

  - ○ We assume that the **probability** to encounter an **observed word depends only** on the corresponding **initial word**.

Correcting errors of both types

- The words we see:

  w_1^k: He pls god ftball.

- Possible candidate corrections:

  t_1^k: He please god football.

  t_1^k: He plays god football.

  t_1^k: He plays good football.

  t_1^k: Her players good football.

  …

  t_1^k: Her pleases god ball.

We now **replace every word** (even vocabulary words) by other **close vocabulary words** (or the same word).

Again, a **language model** estimates **how well** the **words** of each candidate sequence **fit together**.

Generalization for 2nd type of errors

- Now **every observed word may be wrong**.

$$\hat{t}_1^k = \underset{t_1^k}{\arg\max}\, P(t_1^k \mid w_1^k) = \underset{t_1^k}{\arg\max}\, P(t_1^k) \cdot P(w_1^k \mid t_1^k)$$

In each **candidate sequence t_1^k, every observed word** may have been **replaced** by a **vocabulary word** at a **small distance**. Many **more candidates**!

Language model
(e.g., trigram model)

$$P(w_1^k \mid t_1^k) \simeq \prod_{i=1}^{k} P(w_i \mid t_i)$$

Again, simplest approach: probabilities **inversely proportional to edit distance** (with normalization). See **J&M for better ideas**.

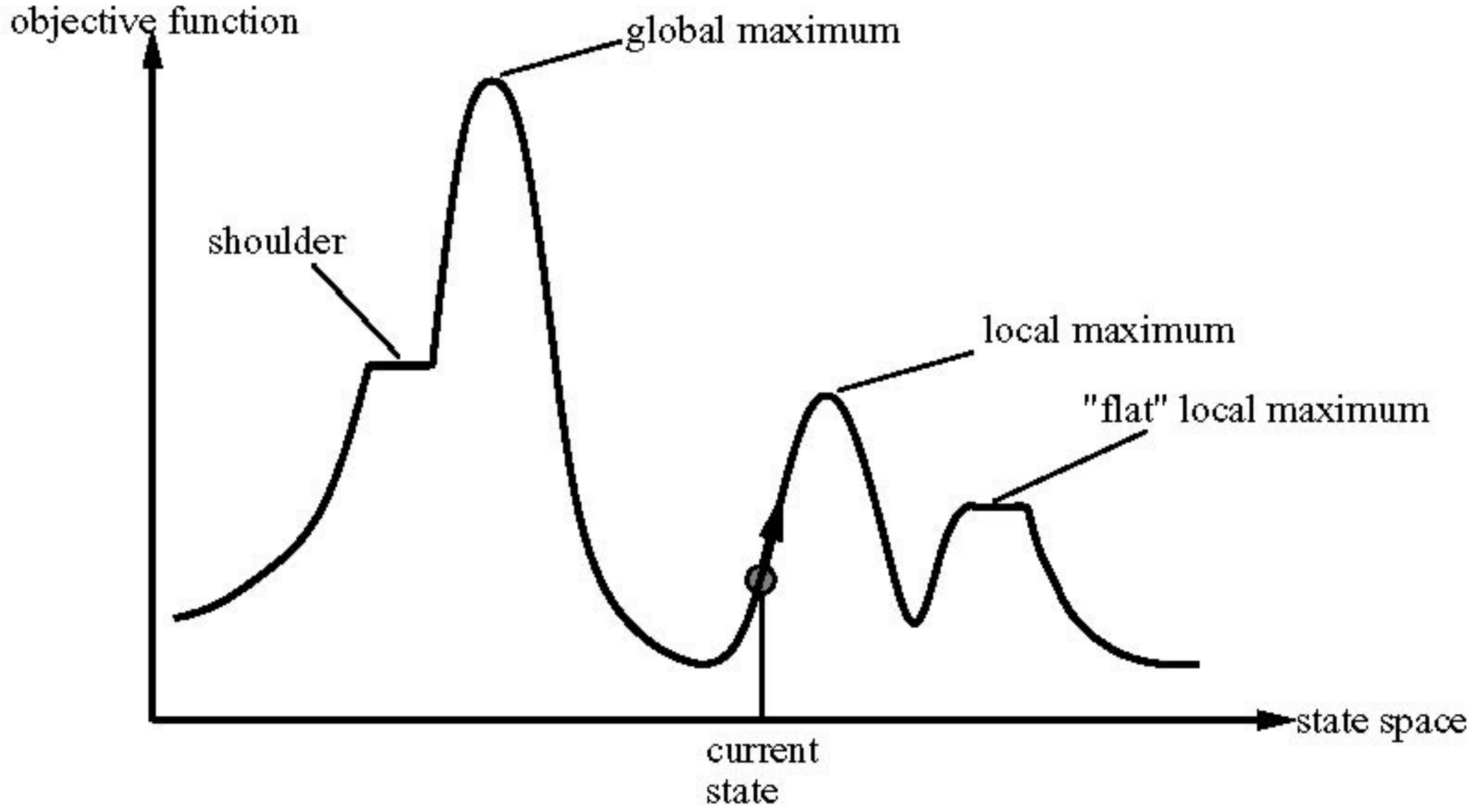
- **Finding the best candidate sequence** $t_1, \ldots, t_k$ is a "decoding" problem, which can be solved with **heuristic search** (e.g., beam search) or **dynamic programming** (e.g., Viterbi).

Hill climbing search (HC)

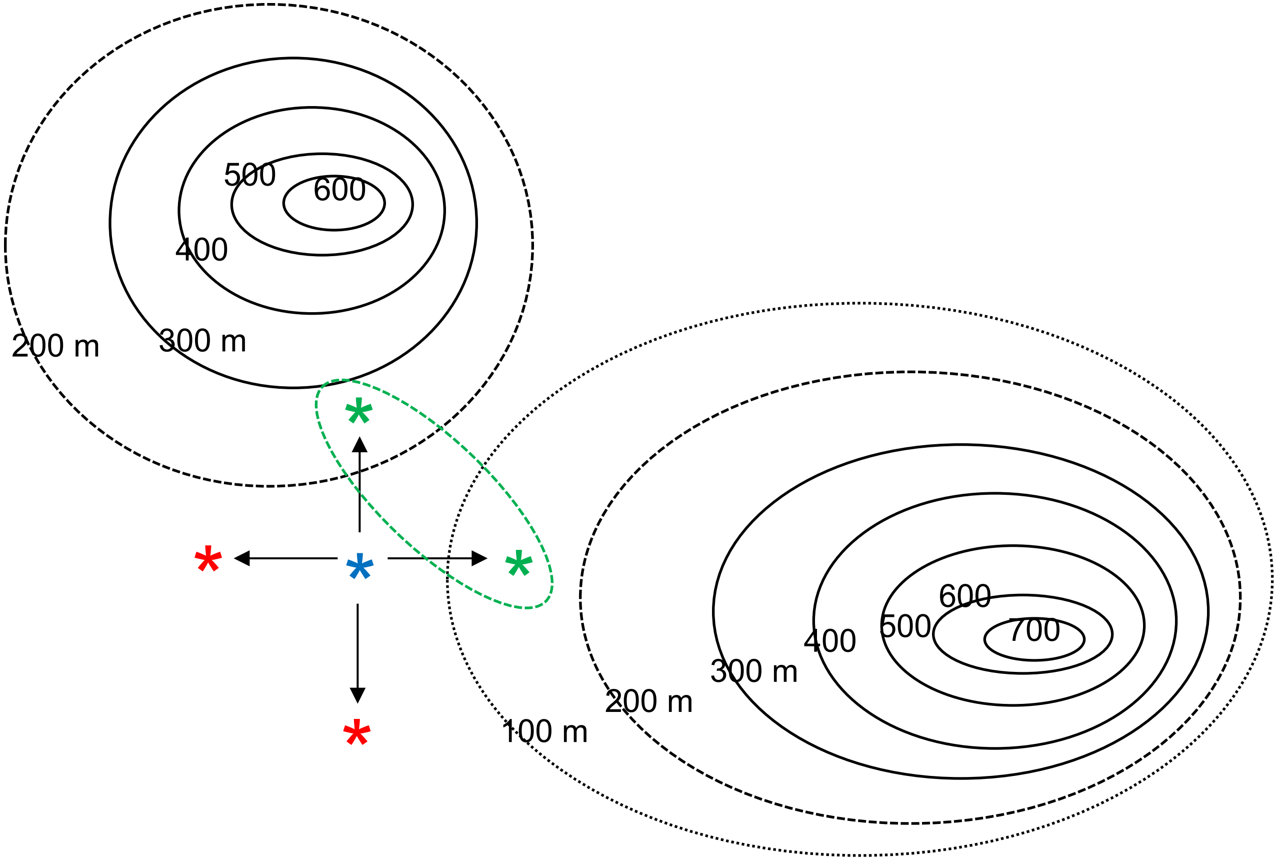
1. Make the **initial state** the **current state**.
2. **Generate** and **assess** the **children-states** of the current state.
3. If **no child-state is better** than the current state, **return the current state**.
4. Make the **best child-state** the **current state**.
5. Go to **step 2**.

Spoiler alert: Most **neural networks** are also trained using a **kind of HC** (**SGD**, stochastic gradient descent), where the **state contains the weights** of the network.

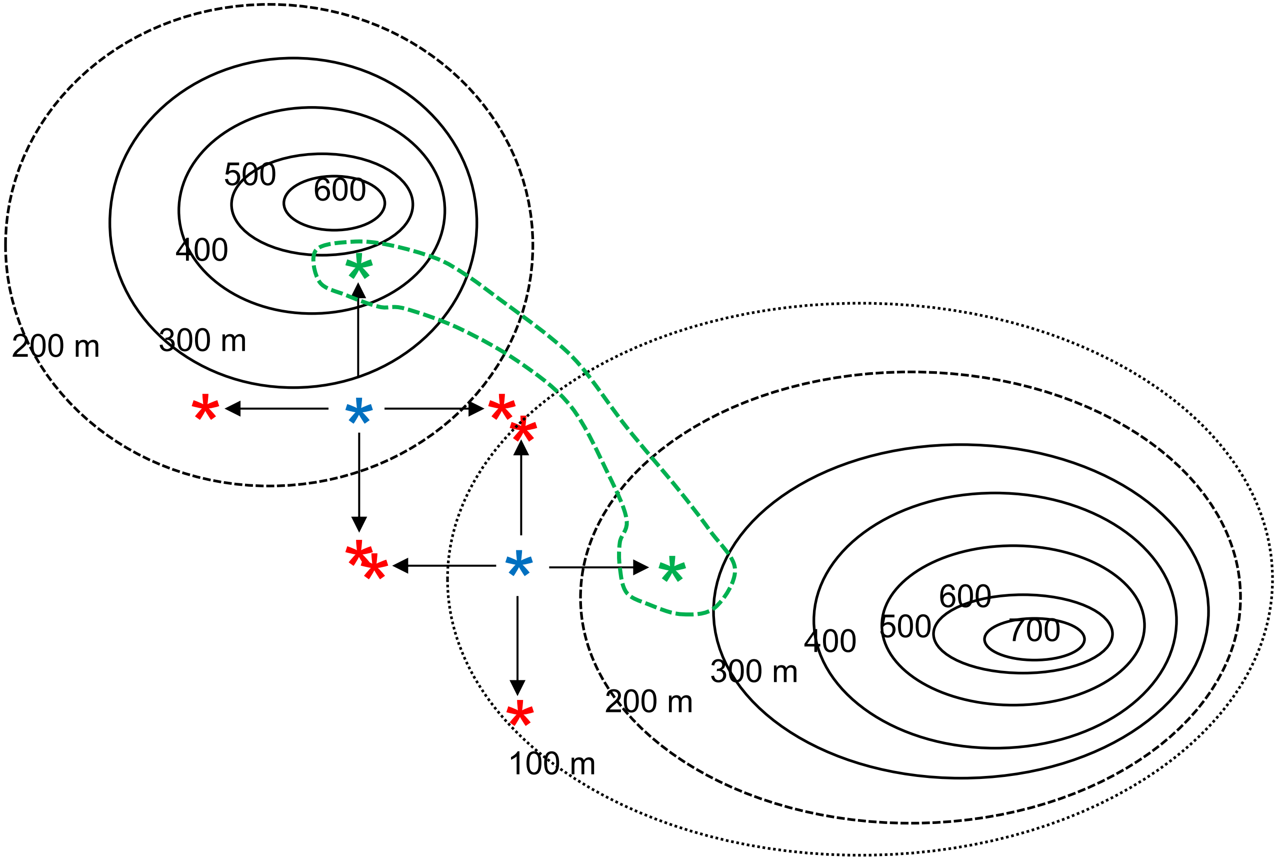
Hill climbing



(Local) Beam search



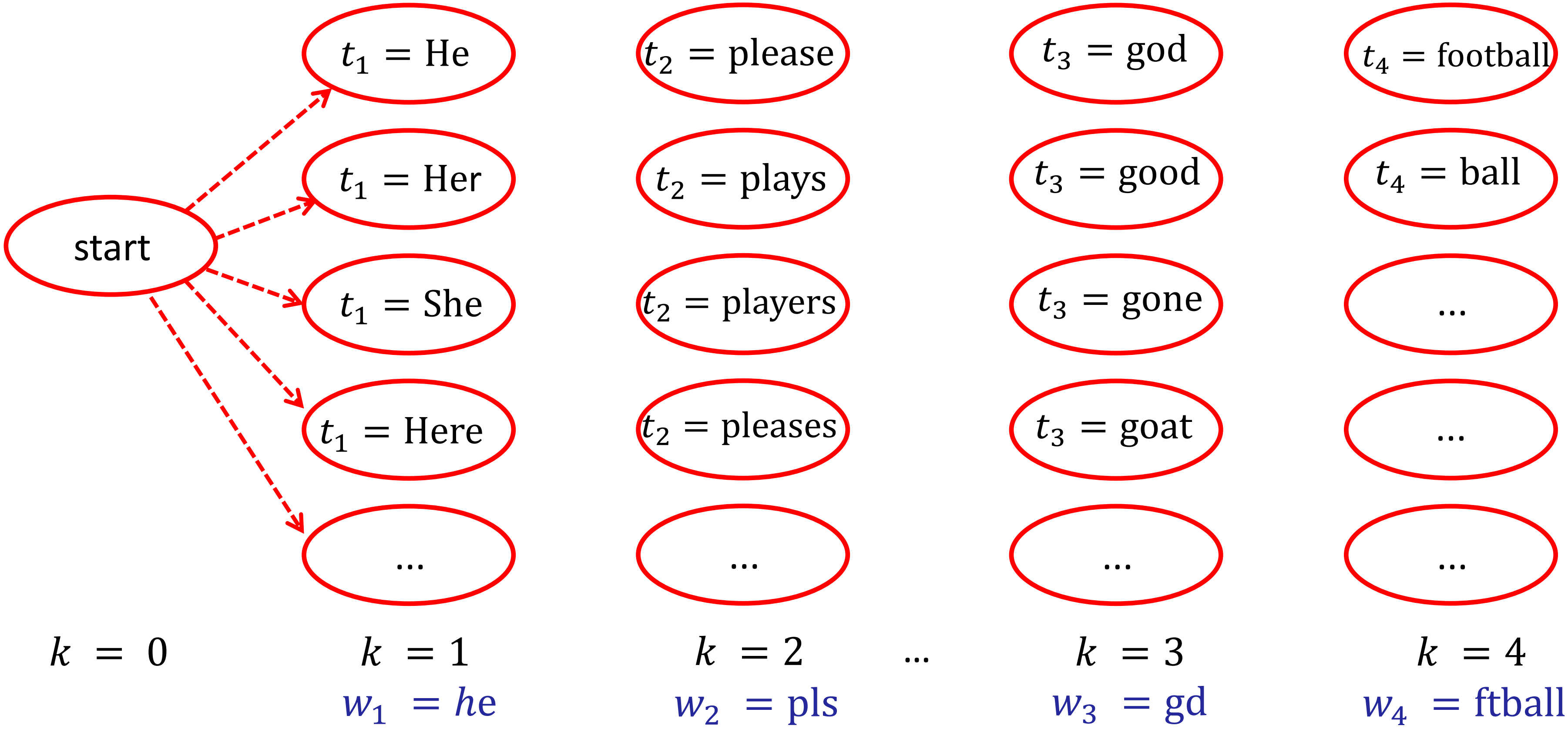
(Local) Beam search



(Local) Beam search

- Like HC, but we keep ***k*** **states** in the **search frontier**.
  - Initially *k* random states.
- At each step, **produce** and **assess** the **children-states of the *k* states** in the frontier.
  - If a final state criterion exists and we reach a final state, stop.
- **Keep** the ***k*** **best of the children-states** and repeat.
  - Until we exceed a maximum number of iterations.
- We often **repeat the search several times**, starting from **different initial *k* states**.
  - Random restarts are also useful in HC.
  - In neural nets, restarts with different random initial weights.
  - In spelling correction decoding, there is only one initial state.

Beam search decoder

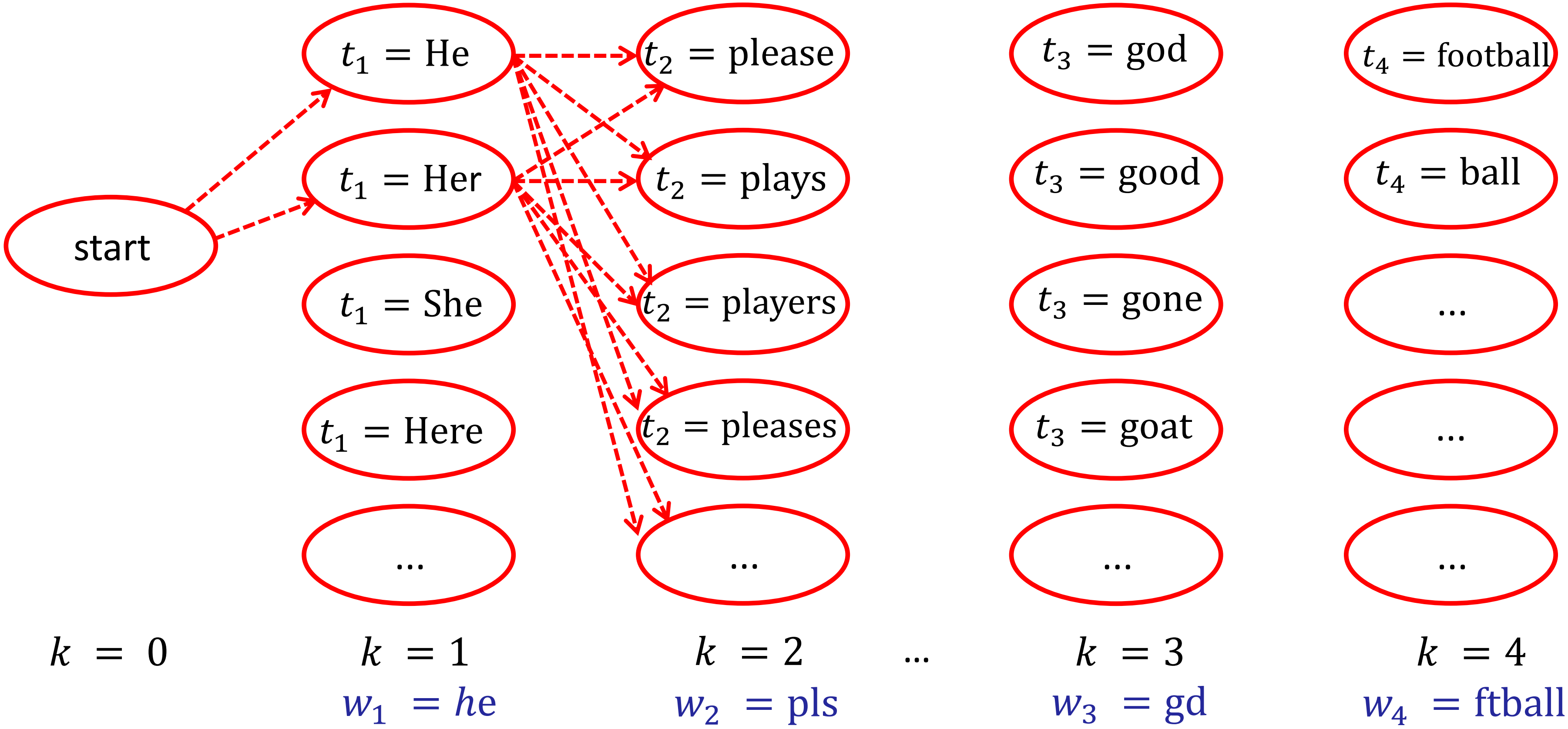


We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^{k} P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^{k} P(t_i|t_{i-1})$

Beam search decoder



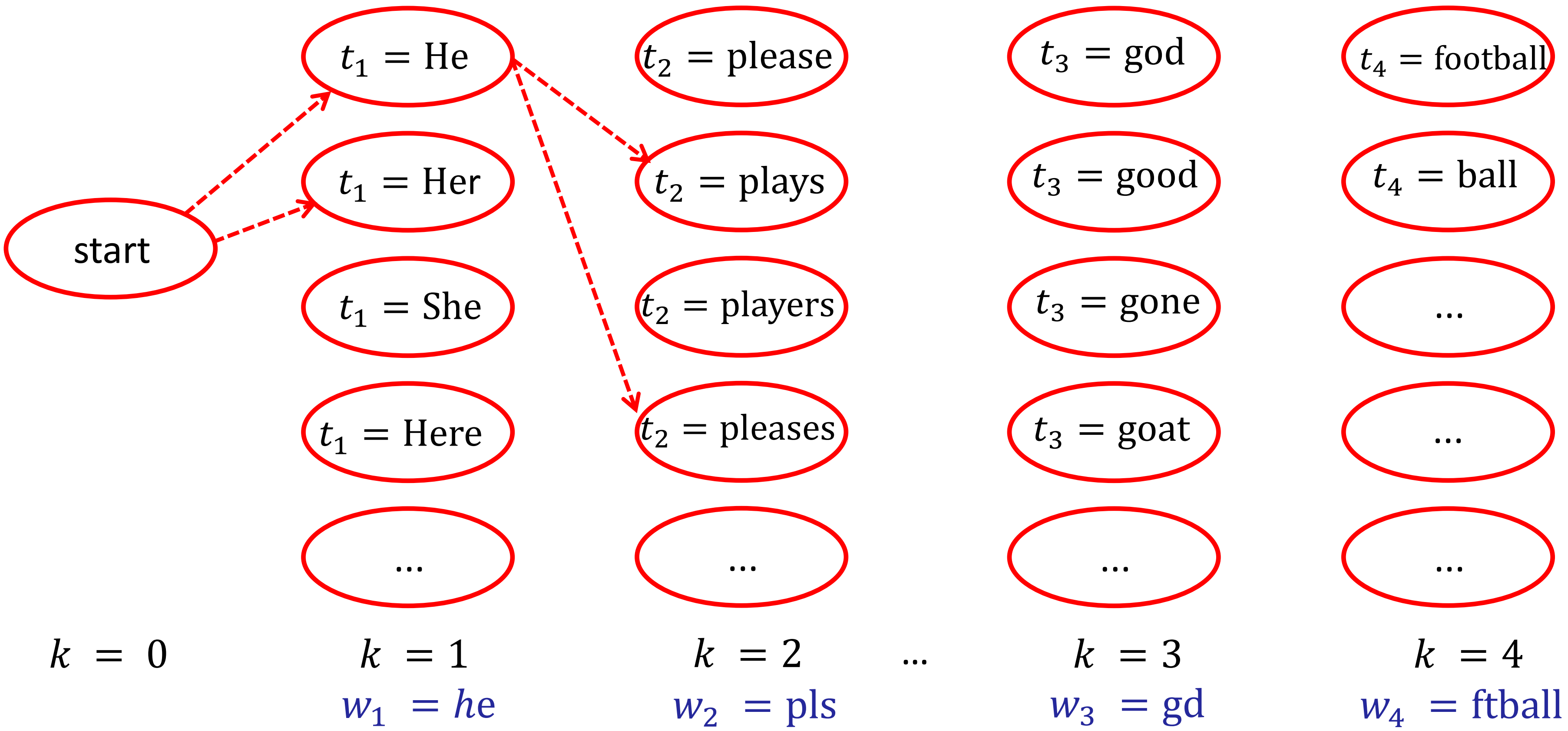
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^{k} P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^{k} P(t_i|t_{i-1})$

For each k, we keep the b (here $b = 2$) best paths only.

Beam search decoder



start

$t_1 = \text{He}$

$t_1 = \text{Her}$

$t_1 = \text{She}$

$t_1 = \text{Here}$

…

$t_2 = \text{please}$

$t_2 = \text{plays}$

$t_2 = \text{players}$

$t_2 = \text{pleases}$

…

$t_3 = \text{god}$

$t_3 = \text{good}$

$t_3 = \text{gone}$

$t_3 = \text{goat}$

…

$t_4 = \text{football}$

$t_4 = \text{ball}$

…

…

…

$k = 0$

$k = 1$
$w_1 = he$

$k = 2$
$w_2 = \text{pls}$

…

$k = 3$
$w_3 = \text{gd}$

$k = 4$
$w_4 = \text{ftball}$

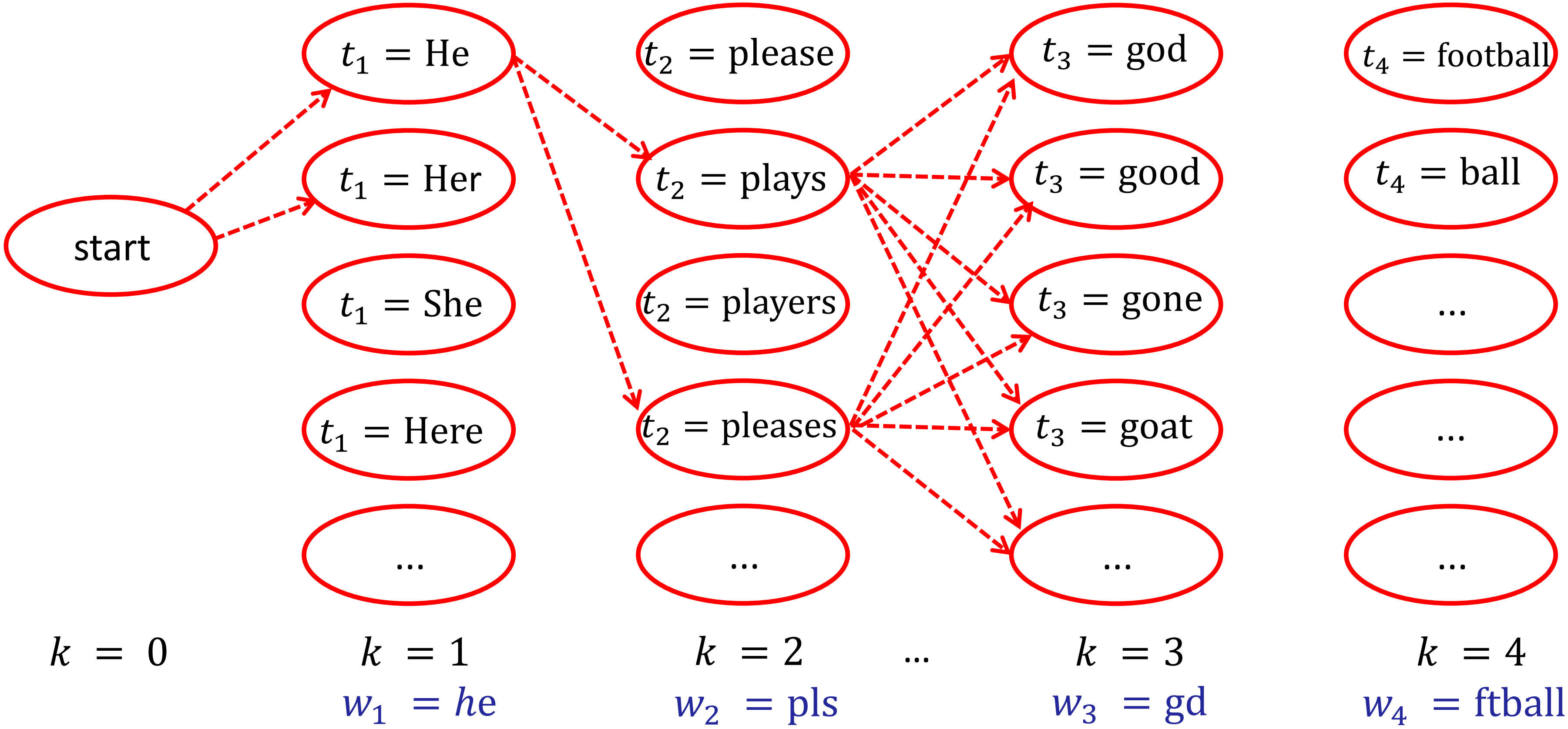
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^{k} P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^{k} P(t_i|t_{i-1})$

For each k, we keep the b (here $b = 2$) best paths only.

Beam search decoder



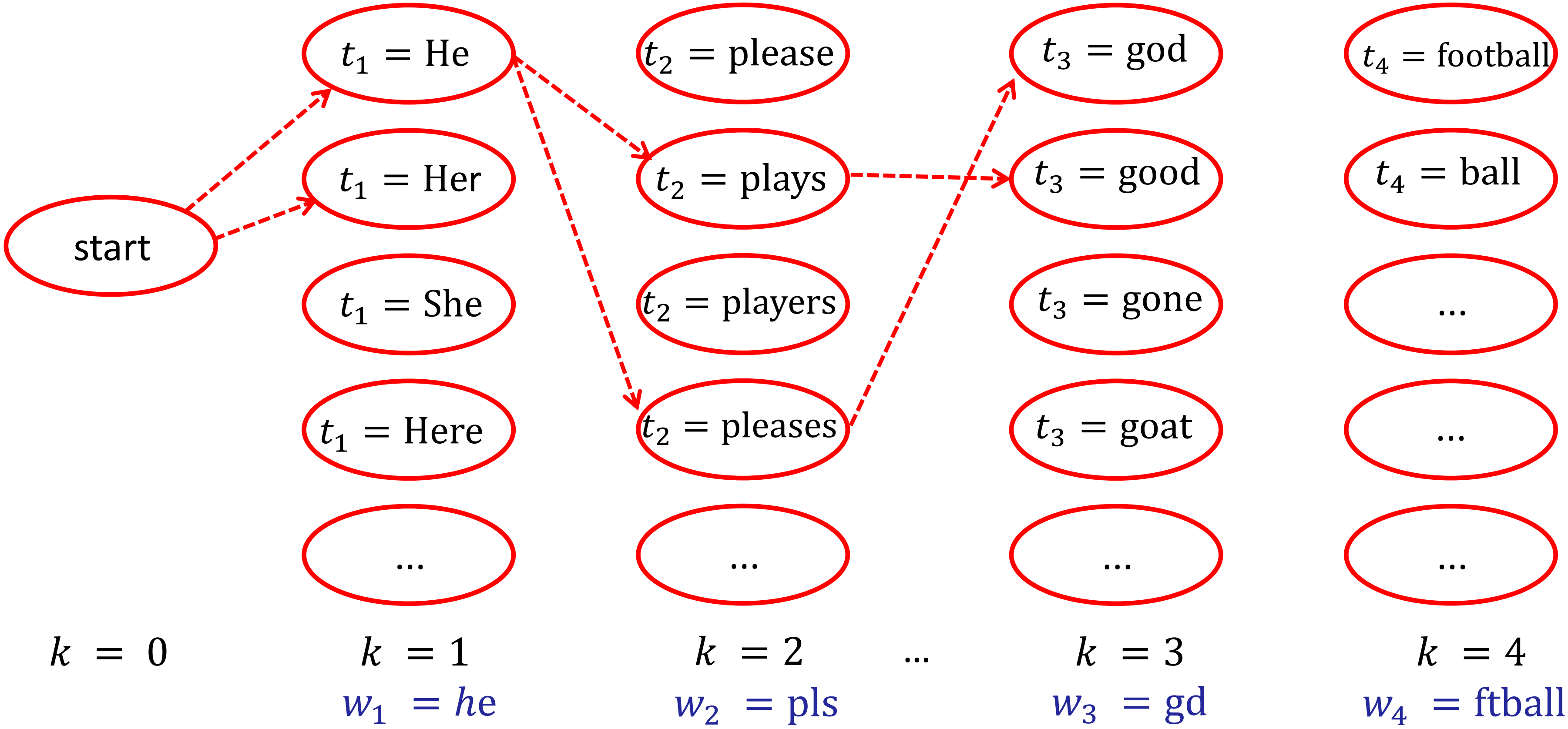
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^{k} P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^{k} P(t_i|t_{i-1})$

For each k, we keep the b (here $b = 2$) best paths only.

Beam search decoder



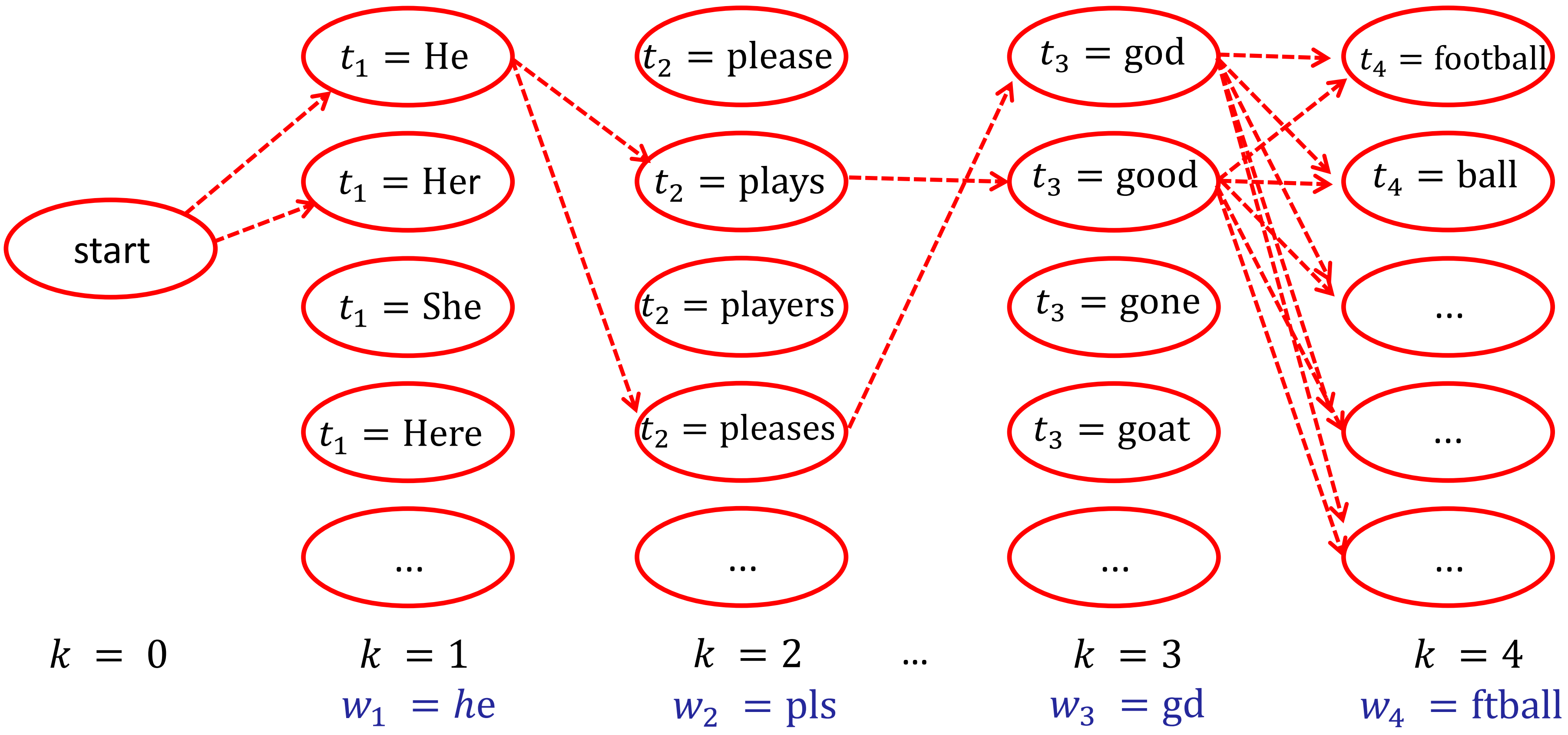
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^{k} P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^{k} P(t_i|t_{i-1})$

For each k, we keep the b (here $b = 2$) best paths only.

Beam search decoder



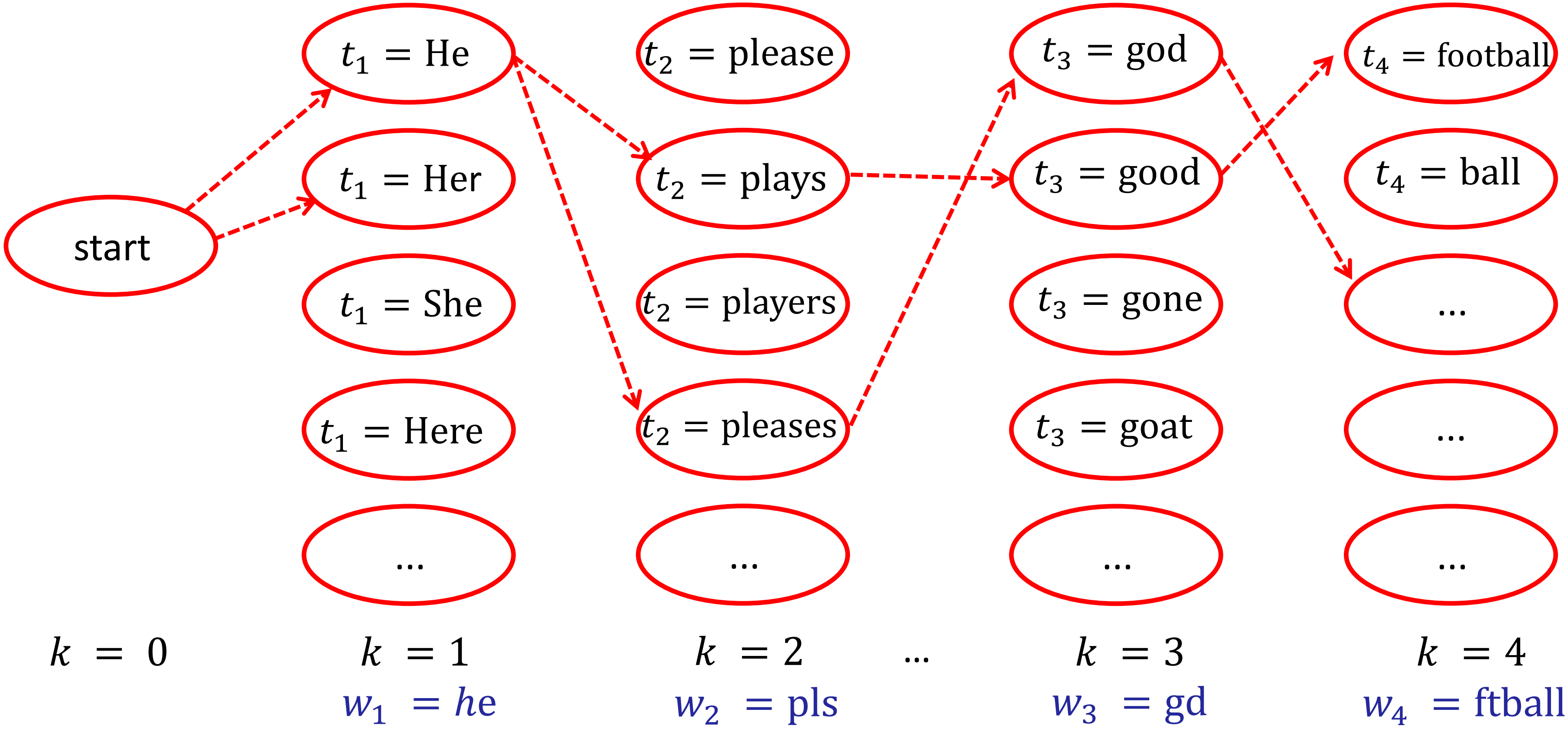
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

With our previous simplifications: $\prod_{i=1}^{k} P(w_i|t_i)$

For a bigram language model: $\prod_{i=1}^{k} P(t_i|t_{i-1})$

For each k, we keep the b (here $b = 2$) best paths only.

Beam search decoder



start

$t_1 = \text{He}$

$t_1 = \text{Her}$

$t_1 = \text{She}$

$t_1 = \text{Here}$

…

$t_2 = \text{please}$

$t_2 = \text{plays}$

$t_2 = \text{players}$

$t_2 = \text{pleases}$

…

$t_3 = \text{god}$

$t_3 = \text{good}$

$t_3 = \text{gone}$

$t_3 = \text{goat}$

…

$t_4 = \text{football}$

$t_4 = \text{ball}$

…

…

…

$k = 0$

$k = 1$ $w_1 = he$

$k = 2$ $w_2 = \text{pls}$

…

$k = 3$ $w_3 = \text{gd}$

$k = 4$ $w_4 = \text{ftball}$

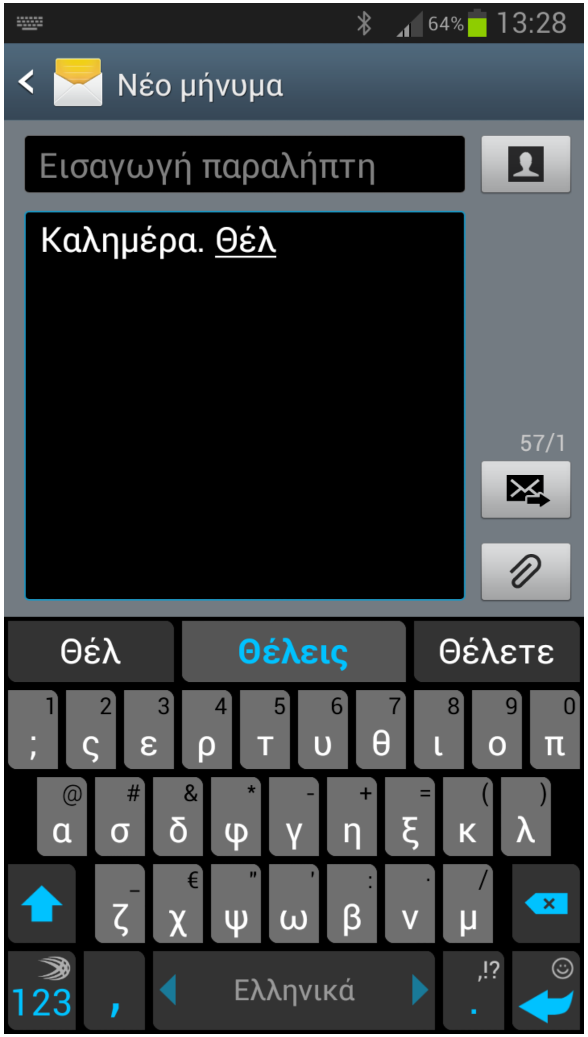
We search for a path from *start* to a state of column $k = 4$ that maximizes $P(t_1^k)P(w_1^k|t_1^k)$ or that minimizes $L_k = -\lambda_1 \log P(t_1^k) - \lambda_2 \log P(w_1^k|t_1^k)$.

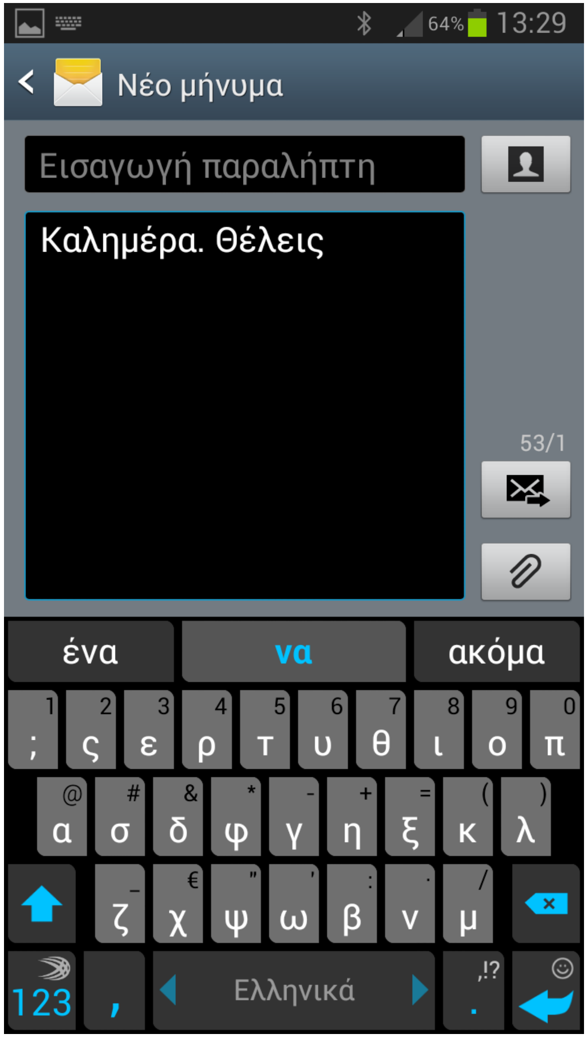
With our previous simplifications: $\prod_{i=1}^{k} P(w_i|t_i)$

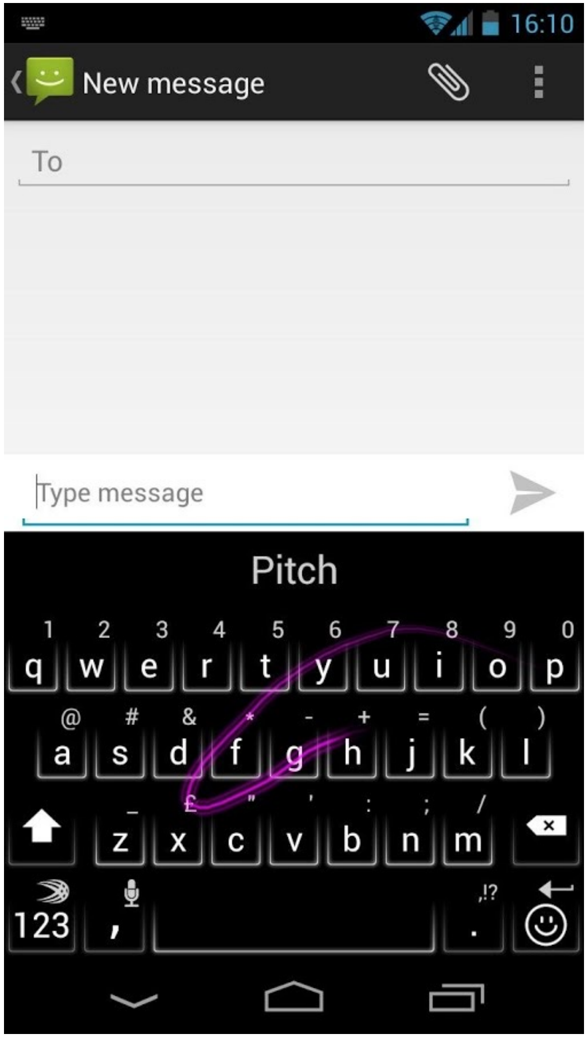
For a bigram language model: $\prod_{i=1}^{k} P(t_i|t_{i-1})$

For each k, we keep the b (here $b = 2$) best paths only.

Smart keyboards


Καλημέρα. Θέλ


Καλημέρα. Θέλεις



Images from: http://www.swiftkey.net/

More to be discussed…

- Exactly how do we **combine** the **edit distances** with a **language model** to correct **errors of the 1st type**?
- How do we correct **errors of the 2nd type**?
- How do we **evaluate** a **language model**?
    - Among different language models (e.g., using different *n* or different smoothing), which one is the best?

Encoding example and entropy

- Let the **possible values** of a random variable ***C*** be:
  - c_1 with $P(c_1) = 1/4$, c_2 with $P(c_2) = 1/4$, c_3 with $P(c_3) = 1/2$.
- A good **encoding**:
  - Use **fewer bits** for **more probable values**.
  - c_1 → 10, c_2 → 11. We use $-\log_2(1/4) = 2$ bits.
  - c_3 → 0. We use $-\log_2(1/2) = 1$ bits.
  - Exp. number of transmitted bits: $1/4 \cdot 2 + 1/4 \cdot 2 + 1/2 \cdot 1 = 1.5$
- Information theory says the **ideal encoding** (lowest expected number of transmitted bits) uses **$-\log_2 P(c_i)$ bits for value c_i**.
  - We may need to use a slightly different number of bits in practice, if the $P(c_i)$ probabilities are not powers of 2.
- **With an ideal encoding** (as above), the **expected number of transmitted bits is the Entropy** $H(C)$ of C.
  - It shows **how uncertain we are about the value** of C, i.e., **how much information (in bits) we need to transmit** to let somebody know its value.

Entropy

- **Entropy** of a random variable *C*:
  - **How uncertain** we are about the **value of *C***.
  - **How much information** (in bits, with an ideal encoding) we need to receive **to be certain** about the value of *C*.
  - What is the **expected number of bits** (with an ideal encoding) that we need to receive **to be certain** about the value of *C*.

$$H(C) = -\overset{\text{Expected value}}{\sum_{c_i}} P(C=c_i)\cdot \underbrace{\boxed{\log_2 P(C=c_i)}}_{\text{Bits used by the ideal encoding for each value.}}$$

- If *C* has only two possible values:

$$H(C) = -P(C=1)\cdot\log_2 P(C=1) - P(C=0)\cdot\log_2 P(C=0)$$

Probabilities estimated from training data.

Example

- Collection of **800 training** e-mail messages.
  - Messages received in the past, manually classified.
  - **200 spam**. **600 ham** (non-spam).
- Estimate the **entropy of *C*** using the training messages.
  - ***C* = 1** (spam) ή ***C* = 0** (ham).
  - $\log_2 3 = 1.585$
- Repeat when **all the training messages** are in **one category** (all spam, or all ham).
- Repeat when we have an **equal number** of training messages **per category** (**400 spam**, **400 ham**).

$$H(C) = -P(C=1)\cdot\log_2 P(C=1) - P(C=0)\cdot\log_2 P(C=0)$$

Cross-entropy

- The **entropy** of a random variable C shows **how uncertain** we are about its value.

$$H(C) = -\sum_{c_i} P(C = c_i) \cdot \log_2 P(C = c_i)$$

  - **How many bits** (expected value) we need to **transmit** (or receive) with an **ideal encoding** to transmit (receive) its value.

- If we use an **encoding** based on **inaccurate probability estimates** P_m instead of the correct probabilities P:

$$H_{P_m}(C) = -\sum_{c_i} P(C = c_i) \cdot \log_2 P_m(C = c_i) \geq H(C)$$

  - We need to **transmit more bits**, because we don't use an ideal encoding (which uses $-\log_2 P(c_i)$ bits per value).

Cross-entropy – continued

- If we have **two models *P*$_{m1}$(*C*), *P*$_{m2}$(*C*)** both trying to **estimate the correct probabilities *P*(*C*)**, which one is the **best**?

$$H_{P_{m1}}(C) = -\sum_{c_i} P(C=c_i)\cdot\log_2 P_{m1}(C=c_i) \geq H(C)$$

$$H_{P_{m2}}(C) = -\sum_{c_i} P(C=c_i)\cdot\log_2 P_{m2}(C=c_i) \geq H(C)$$

  - The one with the **smallest cross-entropy**.
  - It allows **transmitting** the values of *C* using **fewer bits**.
  - Its **encoding is based** on **more accurate probability estimates**.

- **Kullback–Leibler divergence** (relative entropy):

$$D_{KL}(P_m \,\|\, P) = H_{P_m}(C) - H(C) = -\sum_{c_i} P(C=c_i)\cdot\log_2 \tfrac{P_m(C=c_i)}{P(C=c_i)}$$

Cross-entropy with 1-hot $P(C = c_i)$

- **If** the **correct probability distribution** is **1-hot**:
  - i.e., for some i^*, $P(c_{i^*}) = 1$; for all other $i \neq i^*$, $P(c_i) = 0$.
  - E.g., in **classification**, often a **single correct label** per instance.

$$H_{P_m}(C) = -\sum_{c_i} P(C = c_i) \cdot \log_2 P_m(C = c_i) =$$

$$= -\log_2 P_m(C = c_{i^*})$$

  - The same as the **negative log-likelihood** of the “correct” c_{i^*}.
  - The probability assigned by the model to the “correct” c_{i^*}.

- For **two models**:

$$H_{P_{m_1}}(C) = -\log_2 P_{m_1}(C = c_{i^*})$$
$$H_{P_{m_2}}(C) = -\log_2 P_{m_2}(C = c_{i^*})$$

  - The **best model** has the **lowest cross-entropy**, or **highest log-likelihood** of the correct answer.

Evaluating LMs with cross-entropy

- The **correct probability distribution** for the **next word** W_j (given the **history** h_1^{j-1}) in a **test corpus** is **1-hot**:
  - For some $w^* \in V$, $P\left(W_j = w^* | h_1^{j-1}\right) = 1$; and for every other $w \neq w^* \in V$, $P\left(W_j = w | h_1^{j-1}\right) = 0$.

$$H_{P_m}(W_j) = -\sum_{w \in V} P\left(W_j = w | h_1^{j-1}\right) \cdot \log_2 P_m\left(W_j = w | h_1^{j-1}\right) =$$

$$= -\log_2 P_m\left(W_j = w^* | h_1^{j-1}\right)$$

- For all the ***N*** **word occurrences** of a **test corpus**:

$$H_{P_m}(W_1^N) = -\frac{1}{N}\sum_{j=1}^{N} \sum_{w \in V} P\left(W_j = w | h_1^{j-1}\right) \cdot \log_2 P_m\left(W_j = w | h_1^{j-1}\right)$$

Per-word cross-enropy

$$= -\frac{1}{N}\sum_{j=1}^{N} \log_2 P_m\left(W_i = w_i^* | h_1^{j-1}\right)$$

  - The **negative log-likelihood** the model gives to the **test corpus**.

Evaluating LMs with cross-entropy

- For **two models** evaluated on the **same test corpus**:

$$H_{P_{m_1}}(W_1^N) = -\frac{1}{N}\sum_{j=1}^{N} \log_2 P_{m_1}\left(W_i = w_i^* | h_1^{j-1}\right)$$

$$H_{P_{m_2}}(W_1^N) = -\frac{1}{N}\sum_{j=1}^{N} \log_2 P_{m_2}\left(W_i = w_i^* | h_1^{j-1}\right)$$

  - The **best model** is the one with the **lowest cross-entropy**.
  - It gives a **higher log-likelihood** to the **test corpus**.

Cross-Entropy and Perplexity

- For example, if a **bigram language model** is used:

$$-\frac{1}{N}\log_2 P_m(w_1^N) = -\frac{1}{N}\log_2[P_m(w_1|start)P_m(w_2|w_1)\ldots]$$

$$= -\frac{1}{N}[\log_2 P_m(w_1|start) + \log_2 P_m(w_2|w_1) + \ldots]$$

- Usually **perplexity** scores are published:

$$\text{Perplexity} = 2^{H_{P_m}(L) \cong} 2^{-\frac{1}{N}\log_2 P_m(w_1^n)}$$

- The **lower** the perplexity, the **better** the model.
  - Alternative interpretation: a language model with **perplexity *r*** is **as uncertain** (same entropy) **about the next word** as a model that selects **uniformly and independently** words from a **vocabulary of *r* words**.

Comparison 1–4-Gram

$-\log P(w_i|w_{i-1})$

$-\log P(w_i|w_{i-2}, w_{i-1})$

word	unigram	bigram	trigram	4-gram
i	6.684	3.197	3.197	3.197
would	8.342	2.884	2.791	2.791
like	9.129	2.026	1.031	1.290
to	5.081	0.402	0.144	0.113
commend	15.487	12.335	8.794	8.633
the	3.885	1.402	1.084	0.880
rapporteur	10.840	7.319	2.763	2.350
on	6.765	4.140	4.150	1.862
his	10.678	7.316	2.367	1.978
work	9.993	4.816	3.498	2.394
.	4.896	3.020	1.785	1.510
</s>	4.828	0.005	0.000	0.000
average	8.051	4.072	2.634	2.251
perplexity	265.136	16.817	6.206	4.758

$-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{i-2}, w_{i-1})$

$-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{i-1})$

From P. Blunsom's presentation "*From Language Modelling to Machine Translation*"
http://videolectures.net/deeplearning2015_blunsom_machine_translation/

Training, development, test data

- **Training data**:
  - Used to **estimate** (learn) the **probabilities of *n*-grams**.
  - More generally, we **train our model** on these data.
- **Development data:**
  - Used **to select models** (e.g., 2-gram or 3-gram LM), **tune hyper-parameters** (e.g., λ of interpolated LMs), select best **training epochs** (in neural networks) etc.
  - If we **make these choices by evaluating on test data,** we indirectly **train our model on the test dataset**!
- **Test data**:
  - Used for the final **evaluation of our model**, to see how well it performs on **unseen data**.

Training, development, test data

n-gram probability estimates (more generally, learning algorithm applied to this subset)

training data

perplexity (or other score, for hyper-parameter tuning etc.)

development data

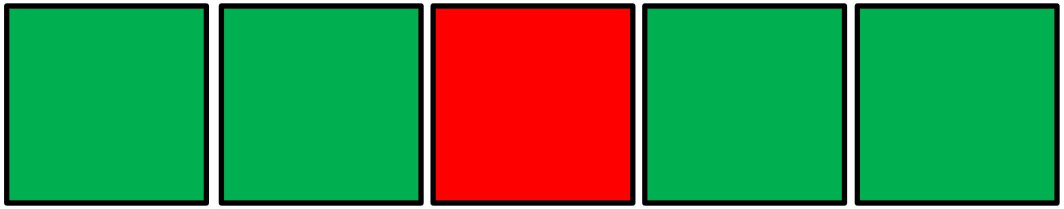
final perplexity (or other score, to check the performance on unseen data)

test data

- In competitions, the **test data may not be publicly available**.
- The **development data** may have to be **“held out” from the training data**.
  - This **reduces** the size of the **training set**.
  - And a **small development** set, **may not be representative**.

Cross-validation

- **Instead of holding out dev data** from the training data:
  - Divide the training data into ***n* parts** (e.g., 5), often **preserving class ratios** (e.g., positives/negatives) in all parts (“stratified”).
  - Perform ***n* iterations** (folds).
  - In each iteration, use **a different part** as **dev data** and the **other *n* – 1 parts** as **training data**.
  - **Average** (e.g., the *F*-measure score) **over the iterations**.
- **Even better** (but costly): **iterative cross-validation**.
  - **Repeat** the cross-validation **multiple times**, with **different random segmentations** into ***n* parts**.



Further issues with n-gram LMs

- What about similar words?
  - she *bought* a bicycle
  - she *purchased* a bicycle

✗ cannot share strength among similar words

- Long-distance dependencies?
  - for *programming* she yesterday purchased her own brand new *laptop*
  - for *running* she yesterday purchased her brand new *sportswatch*

✗ cannot handle long-distance dependencies

Adapted from Graham Neubig

Slide from the talk of Barbara Plank on Encoder-Decoder Models at AthNLP 2024 (https://athnlp.github.io/2024/presentations.html).

Additional optional study slides.

Kneser-Ney smoothing

- E.g., for bigrams w_{k-1}, w_k:
  - ✓ green, apple
  - ✓ green, paper
  - ✓ green, book
  - ✓ …
  - ✓ …
  - × green, mouse
  - × green, cyclotron
  - × green, York
  - × …

Encountered in the corpus, i.e., $c(w_{k-1}, w_k) > 0$. **Steal probability mass** from each estimate $\frac{c(w_{k-1}, w_k)}{c(w_{k-1})}$, i.e., use $\frac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}$, where D is constant.

Not encountered in the corpus, i.e., $c(w_{k-1}, w_k) = 0$. **Distribute** to them the **probability mass stolen from all the encountered** $\boldsymbol{w_{k-1}, w_k}$ that had the **same** $\boldsymbol{w_{k-1}}$ ("green"). **Distribute proportionally to** $\boldsymbol{P(w_k)}$ (e.g., "mouse" is more frequent than "cyclotron").

Kneser-Ney smoothing

- Formula for ideas of previous slide (*D* is constant):

$$P_{KN}(w_k \mid w_{k-1}) = \begin{cases} \dfrac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}, & \text{if } c(w_{k-1}, w_k) > 0 \\ a(w_{k-1}) \cdot P(w_k), & \text{else} \end{cases}$$

  - α values needed to ensure that probabilities sum up to 1.

Improved K-N smoothing

- Instead of $P(w_k)$, distribute the stolen probability mass proportionally to:

$$Prev(w_k) = \frac{prev(w_k)}{\sum_{v \in V \,:\, c(w_{k-1}, v) = 0} prev(v)}$$

The denominator ensures that the $Prev(w_k)$ scores of all the words w_k that need to receive stolen probability mass **sum to 1**.

where:

$$prev(w_k) = |\{w \in V \,:\, c(w, w_k) > 0\}|$$

How many vocabulary (distinct) words occur **immediately before w_k in the corpus**. E.g., "York" may occur almost always after "New"; hence "green York" should not be given much of the probability mass stolen from the encountered bigrams that start with "green".

Improved K-N smoothing

$$P_{KN}(w_k \mid w_{k-1}) = \begin{cases} \dfrac{c(w_{k-1}, w_k) - D}{c(w_{k-1})}, & \text{if } c(w_{k-1}, w_k) > 0 \\ a(w_{k-1}) \cdot Prev(w_k), & \text{else} \end{cases}$$

$$a(w_{k-1}) = \frac{D}{c(w_{k-1})} \cdot |\{w \in V : c(w_{k-1}, w) > 0\}|$$

Total probability mass stolen from bigrams that start with w_{k-1} (w_{k-1} =“green” in our example).

Katz backoff

- Consult an *n*-gram model with a **smaller *n*, whenever necessary**. For example, when using a trigram model:

$$P_{Katz}(w_k \mid w_{k-2}, w_{k-1}) = \begin{cases} P(w_k \mid w_{k-2}, w_{k-1}), \text{ if } c(w_{k-2}^{k}) > 0 \\ a(w_{k-2}, w_{k-1}) \cdot P_{Katz}(w_k \mid w_{k-1}), \text{ else} \end{cases}$$

$$P_{Katz}(w_k \mid w_{k-1}) = \begin{cases} P(w_k \mid w_{k-1}), \text{ if } c(w_{k-1}, w_k) > 0 \\ a(w_{k-1}) \cdot P(w_k), \text{ else} \end{cases}$$

- The *α* values are needed to ensure that:

$$\sum_{w_k \in V} P_{Katz}(w_k \mid w_{k-2}, w_{k-1}) = 1 \qquad \sum_{w_k \in V} P_{Katz}(w_k \mid w_{k-1}) = 1$$

  - Consult the book of Jurafsky & Martin for formulae to compute the *α* values and (many) other smoothing methods.

Computing Levenshtein distance

- How can we **convert**:

  πέζοι to *παίζω*

  based on shorter (by one final letter) forms of *πέζοι* and/or *παίζω*?
- 1st way: Delete the last letter of *πέζοι* and convert *πέζο* to *παίζω*.

  πέζο ~~*ι*~~ → *παίζω*

  Del(*ι*) + cost(*πέζο*, *παίζω*)
- 2nd way: Convert *πέζοι* to *παίζ* and add *ω* to the end of *παίζ*.

  πέζοι → *παίζω*

  cost(*πέζοι*, *παίζ*) + Ins(*ω*)

Computing Levenshtein distance (II)

- 3rd way: Convert *πέζο* to *παίζ* and replace *ι* by *ω*.

πέζο$\boxed{\iota}$ → *παίζ*$\boxed{\omega}$

cost(*πέζο*, *παίζ*) + Rep(*ι*, *ω*)

- **Which way is the best?**
  - The one with the **smallest cost**.
  - **At each step**, we consider all three ways and we select **the cheapest one (Ins, Del, or Rep)**.

Computing Levenshtein distance

	#	π	α	ί	ζ	ε	τ	ε
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7
π	1 ↑							
έ	2 ↑							
ζ	3 ↑							
ο	4 ↑							
ι	5 ↑							
τ	6 ↑							
α	7 ↑							
ι	8 ↑							

↑ Del (+1)

← Ins (+1)

What is the (min) cost to **convert** "#" to "**#παί**"?

Cost to **convert** "#" to "**#πα**" and **insert** "**ί**".

What is the (min) cost to **convert** "**#πέζ**" to "#"?

Cost to **delete** «ζ» and **convert** "**#πέ**" to "#".

Computing Levenshtein distance

	#	π	α	ί	ζ	ε	τ	ε
#	0	1 1+1=2	2	3	4	5	6	7
π	1	0+0=0 0 1+1=2						
έ	2							
ζ	3							
ο	4							
ι	5							
τ	6							
α	7							
ι	8							

↑ Del (+1)

← Ins (+1)

↖ Rep (+2, or 0 for same letter)

What is the (min) cost to **convert** “#**π**” to “#**π**”?

1st way: **Delete** the “**π**” of the first (vertical) string and **convert** the remaining string “#” to “#**π**”.

2nd way: **Convert** the “#**π**” of the first string to “#” and **add** “**π**” to the resulting string.

3rd way: **Convert** the “#” of the first string to “#” and replace the “**π**” of the first string by “**π**” (same letter).

Computing Levenshtein distance

	#	**π**	**α**	**ί**	**ζ**	**ε**	**τ**	**ε**
#	0	1	2	3	4	5	6	7
π	1	0	1					
έ	2							
ζ	3							
ο	4							
ι	5							
τ	6							
α	7							
ι	8							

2+1=3

1+2=3

0+1=1

↑ Del (+1)

← Ins (+1)

↖ Rep (+2, or 0 for same letter)

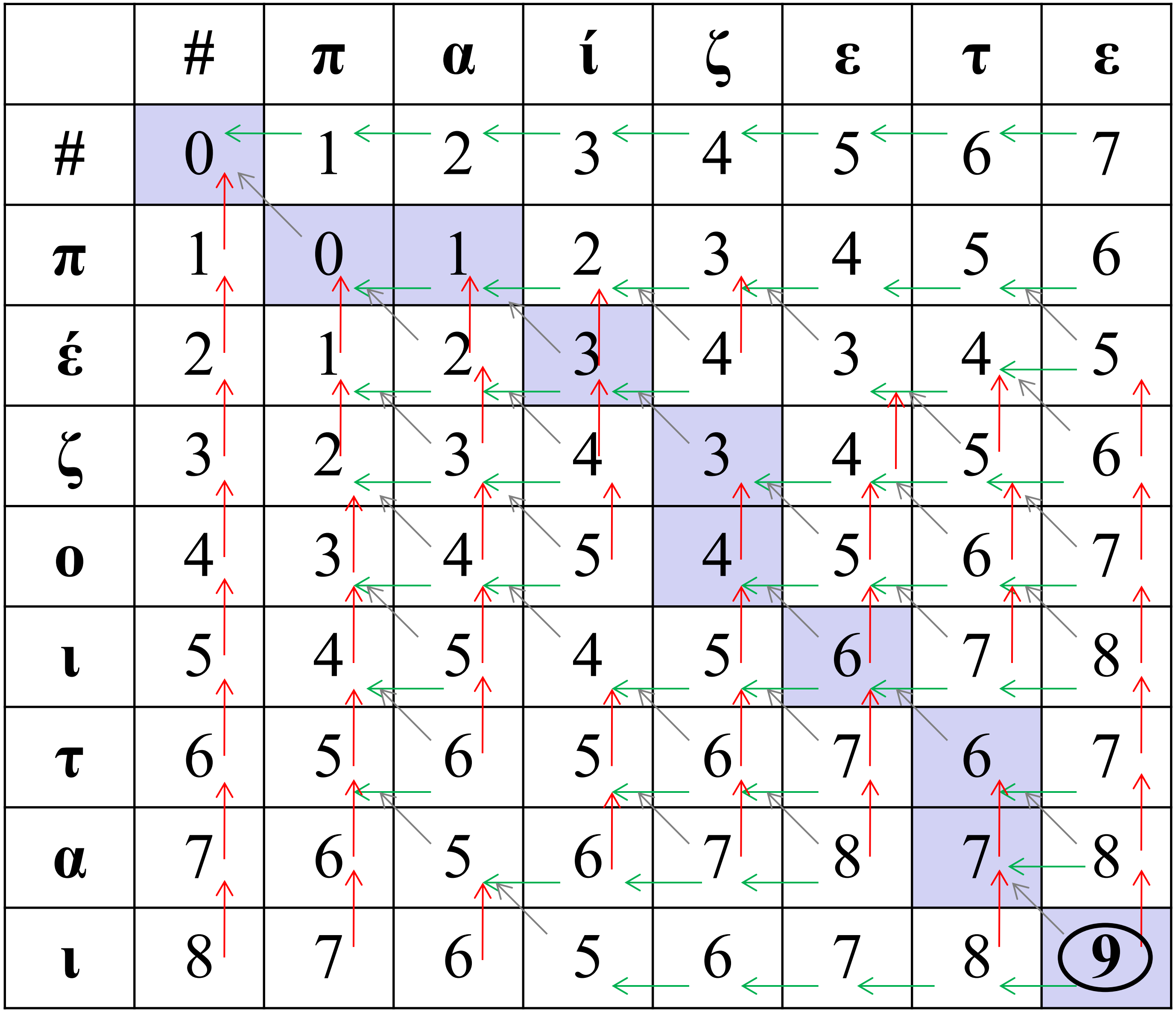
What is the (min) cost to **convert "#π"** to **"#πα"**;

1st way: **Delete** the **"π"** of the first (vertical) string and **convert** the remaining string "#" to "**#πα**".

2nd way: **Convert** the "**#π**" of the first string to "**#π**" (no change) and **add "α"** to the resulting string.

3rd way: **Convert** the "#" of the first string to "**#π**" and **replace** the "**π**" of the first string by "**α**".

Computing Levenshtein distance



	#	π	α	ί	ζ	ε	τ	ε
#	0	1	2	3	4	5	6	7
π	1	0	1	2	3	4	5	6
έ	2	1	2	3	4	3	4	5
ζ	3	2	3	4	3	4	5	6
ο	4	3	4	5	4	5	6	7
ι	5	4	5	4	5	6	7	8
τ	6	5	6	5	6	7	6	7
α	7	6	5	6	7	8	7	8
ι	8	7	6	5	6	7	8	9

↑ Del (+1)

← Ins (+1)

↖ Rep (+2, or 0 for same letter)

Shaded cells show one of the possible alignments.

For each cell, the outgoing arrows point to the neighbor(s) the cell's (best) value is based on.

Language entropy

- We are given ***n*-tuples of words** of a **language *L***. How **uncertain** are we about the ***n*-tuple** we will be given?

$$H(W_1^n \mid L) = H(W_1, \ldots, W_n \mid L) = -\sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n)$$

- **Per-word entropy** (entropy rate):

$$\frac{1}{n} \cdot H(W_1^n \mid L) = -\frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n)$$

- **Entropy of language** L:

$$H(L) = \lim_{n \to +\infty} \frac{1}{n} H(W_1^n \mid L) =$$

$$-\lim_{n \to +\infty} \frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n)$$

Language entropy – continued

- **Entropy of language** L:

$$H(L) = \lim_{n \to +\infty} \frac{1}{n} H(W_1^n \mid L) =$$

$$- \lim_{n \to +\infty} \frac{1}{n} \sum_{w_1^n \in V(L)^n} P(W_1^n = w_1^n) \cdot \log_2 P(W_1^n = w_1^n)$$

- **It can be proven** (Shannon-McMillan-Breiman) that:

$$H(L) = - \lim_{n \to +\infty} \frac{1}{n} \log_2 P(W_1^n = w_1^n) \simeq \boxed{- \frac{1}{N} \log_2 P(w_1^N)}$$

  - In other words, we can use **only one, very long sequence of** N **words** (a corpus) of the language L.
  - More precisely, assuming that the language is **stationary** and **ergodic**.

Language cross-entropy

- If we have **two language models *P*$_{m1}$, *P*$_{m2}$** for a language *L*, which one is the **best**?

$$H_{P_{m1}}(L) = -\lim_{n\to+\infty}\frac{1}{n}\sum_{w_1^n\in V(L)^n} P(W_1^n = w_1^n)\cdot\log_2 P_{m1}(W_1^n = w_1^n)$$

$$= -\lim_{n\to+\infty}\frac{1}{n}\log_2 P_{m1}(w_1^n) \simeq \boxed{-\frac{1}{N}\log_2 P_{m1}(w_1^N)}$$

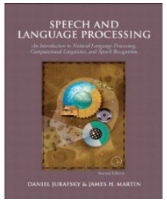
$$H_{P_{m2}}(L) = -\lim_{n\to+\infty}\frac{1}{n}\sum_{w_1^n\in V(L)^n} P(W_1^n = w_1^n)\cdot\log_2 P_{m2}(W_1^n = w_1^n)$$

$$= -\lim_{n\to+\infty}\frac{1}{n}\log_2 P_{m2}(w_1^n) \simeq \boxed{-\frac{1}{N}\log_2 P_{m2}(w_1^N)}$$

  - ○ The model with the **smallest language cross-entropy**. In effect, the model that assigns the **largest probability** to the **test corpus**.
  - ○ Always true: $H_{P_m}(L) \geq H(L)$

Recommended reading

- Jurafsky & Martin (2nd ed.): chapter 4 (not sections 4.5.2, 4.5.3, 4.7.1, 4.9.2), sections 3.10, 3.11, 5.9.
  - Available at AUEB's library.
  - See also the free draft of the 3rd edition: http://web.stanford.edu/~jurafsky/slp3/


SPEECH AND LANGUAGE PROCESSING
DANIEL JURAFSKY & JAMES H. MARTIN

- For more information, consult chapters 2 and 6 of Manning & Schütze's book *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.
  - Available at AUEB's library.
  - Chapter 2 introduces basic concepts of probability theory, entropy, the noisy channel etc.
  - Chapter 6 covers *n*-gram language models.