Tokenization

CSCI 1460: Computational Linguistics
Lecture 4
Announcements

• Assignment 1 is out!
• I’ll be traveling next week. I’ll upload a pre-recorded lecture on Tuesday. Charlie will give a lecture on Thursday.
Quiz 3 Recap

In the below piece of text, which of the following preprocessing steps appear to have been run? Check all that apply.

65 / 92 correct responses

- Tokenization: 24 (26.1%)
- Lemmatization: 92 (100%)
- Stop Word Removal: 90 (97.8%)
- Vocabulary Thresholding/Removing Rare Words: 2 (2.2%)
- Tagging: 0 (0%)
- Lowercasing: 92 (100%)
Can you please explain in lecture the correct answer for these two from question #3:

Spelling out contractions (i.e., "I'm" -> "I am")
Spelling out abbreviations (i.e., "RI" -> "Rhode Island")

I selected Fewer for both of them because in the question it states "You can assume you are working with a very large corpus in which all words occur at least once." So I take that to mean that "I'm", "I", "am", "RI", "Rhode", "Island" and "Rhode Island" are all already in the corpus, so when you remove the contractions and abbreviations from the corpus that will result in less features in the matrix.

Is this the correct way to think about this question? Can you please elaborate on this 2 cases in the next lecture? Thank you!
Topics

• (English) Morphology 101
• Finite State Machines
• Subword Tokenization
• [Saving for Later] Supporting all languages!
Tokenization vs. Morphological Analysis

- Tokenization — splitting a string (sentence) into “words”
  - Unoaked chardonnay. -> [“unoaked”, “chardonnay”, “.”]
- Morphological parsing — splitting a “word” into morphemes
  - unoaked -> [“un”, “oak”, “ed”]
- These processes are not really different! Rather, points along a continuum
Tokenization

• Input: String of characters
• Output: list of (hopefully meaningful) units
• The Rhode Island-based baby food producer didn’t comment.
• “The” “Rhode Island” “based” “baby” “food” “produc” “er” “did” “n’t” “comment”
Topics

• (English) Morphology 101
• Finite State Machines
• Subword Tokenization
• [Saving for Later] Supporting all languages!
Can’t we just split on white space?
Can’t we just split on white space?

Problem 1: Compounding

- We can form new semantically meaningful units by combining existing words
- These compounds may or may not contain whitespace and punctuation
- E.g.,
  - “ice cream”
  - “website”, “web site”
  - “Rhode Island-based”
Can’t we just split on white space?

Problem 1: Compounding

- We can form new semantically meaningful units by combining existing words.
- These compounds may or may not contain whitespace and punctuation.
- E.g.,
  - “ice cream”
  - “website”, “web site”
  - “Rhode Island-based”
Can’t we just split on white space?

Problem 2: Many writing systems don’t use whitespace

• Chinese:
  • 我开始写小说 =
  • 我 (I) 开始 (start(ed)) 写 (writing) 小说 (novel(s))

• Turkish:
  • uygurlaštıramadıklarımızdanmişsınızcasına
  • uygur (civilized) laş (become) tır (cause to X) ama (not able) dık (past participle) lar (plural) ımız (first person plural) dan (from/among) mış (past tense) sınız (second person plural) casına (adverb form)
  • “behaving as if you are among those we could not civilize”
Can’t we just split on white space?

Problem 3: Clitics

• Clitic: like a word, but only every occurs with other words; needs a “host”

• English:
  • “doesn’t”, “I’m”

• Italian:
  • “L’ho veduta ieri” (I saw her yesterday)
Can’t we just split on white space?
Problem 4: Word formation is productive!

• Can combine things in infinitely many ways
• New word enters the language: Zoom
  • All inflections are instantly valid: zoomed, zooming, will have zoomed
  • Particles: zoom in, zoom it, zoomed out
  • Compounds: zoom meeting, zoom party, zoom fatigue
• …
Topics

• (English) Morphology 101
• **Finite State Machines**
• Subword Tokenization
• [Saving for Later] Supporting all languages!
Finite State Machines

- Key idea from Theory of Computation! (Take 1010 for more!)
- Main computational tool for morphological parsing in NLP
- Two related tools:
  - Finite State Automata (FSA): Used to “accept” a string—i.e., tells me whether or not a string is “in the language” defined by the FSA
  - Finite State Transducer (FST): Used to “translate” one string into another, i.e., output a new string for each input
Finite State Automata

• Model/rules which describes a language
• Can say whether or not a given string is “in” the language or not
• E.g., computational procedure for telling us:
  • dog, dogs, goose, geese, mouse, mice —> Good!
  • gooose, mouses, deg, meuse, dig, gice —> Bad!
Finite State Automata
Simple Example

“sheep” language: contains “b” followed by at least two “a’s followed by “!”

baa!       ba!
baaa!      baba!
baaaaaaa!  abaaa!
Finite State Automata

Simple Example

“sheep” language: contains “b” followed by at least two “a”s followed by “!”

<table>
<thead>
<tr>
<th>baa!</th>
<th>ba!</th>
</tr>
</thead>
<tbody>
<tr>
<td>baaa!</td>
<td>baba!</td>
</tr>
<tr>
<td>baaaaaaaa!</td>
<td>abaaa!</td>
</tr>
</tbody>
</table>

Note: NOT machine learning!* We are going to write down a rule-based procedure for identifying strings in the language. No training. No statistics.
Finite State Automata

Simple Example

“sheep” language: contains “b” followed by at least two “a”s followed by “!”

- baa!
- baaa!
- baaaaaaa!
- baaaaaaa!
- ba!
- baba!
- abaaa!

Note: NOT machine learning!* We are going to write down a rule-based procedure for identifying strings in the language. No training. No statistics.

*Though, today, most morphological analysis would use machine learning
Finite State Automata

Simple Example

start state

\[
\begin{array}{cccc}
 b & a & a & a \\
 \ \ & a & ! & \\
\end{array}
\]
Finite State Automata
Simple Example

start of tape

q0 b a a a !

q0 b → q1 a → q2 a → q3 ! → q4

a

!
Finite State Automata
Simple Example

q0  b  a  a  a  !

q0 → b → q1 → a → q2 → a → q3 → ! → q4
Finite State Automata

Simple Example

q0

\[ b \quad a \quad a \quad a \quad ! \]

q4

q1 \quad a \quad q2 \quad a \quad q3

b

a

!
Finite State Automata

Simple Example
Finite State Automata

Simple Example
Finite State Automata

Simple Example

q0 \xrightarrow{b} q1 \xrightarrow{a} q2 \xrightarrow{a} q3 \xrightarrow{!} q4

b a a a a !
Finite State Automata

Simple Example

q0 \xrightarrow{b} q1 \xrightarrow{a} q2 \xrightarrow{a} q3 \xrightarrow{a} q4

b a a a !
Finite State Automata

Simple Example

\[
\begin{array}{cccc}
  b & a & a & a \\
\end{array}
\]

\[
q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \xrightarrow{a} q_4
\]
Finite State Automata

Simple Example

\[
\begin{array}{cccc}
  b & a & a & a \\
\end{array}
\]

\[
\begin{array}{cccc}
  q_0 & q_1 & q_2 & q_3 & q_4 \\
  \rightarrow b & \rightarrow a & \rightarrow a & \rightarrow ! & \rightarrow \\
\end{array}
\]
Finite State Automata
Simple Example

\[
\begin{array}{cccccc}
b & a & a & a & !
\end{array}
\]

q0 \rightarrow b \rightarrow q1 \rightarrow a \rightarrow q2 \rightarrow a \rightarrow q3 \rightarrow ! \rightarrow q4
Finite State Automata
Simple Example

```
b  a  a  a  !
```

q0  b  q1  a  q2  a  q3  !  q4

end of tape
final/accept state
Finite State Automata
Simple Example

```
q0 -> q1 -> q2 -> q3 -> q4
b  a  a  a  !
```

- **Initial State:** q0
- **Final/accept state:** q4
- **Input Symbols:** b, a
- **Transition Label:**
  - q0 -> q1: b
  - q1 -> q2: a
  - q2 -> q3: a
  - q3 -> q4: !
- **End of Tape** indicator
Finite State Automata

Simple Example

```
  b a b a !
```

```
q0 b q1 a q2 a q3 ! q4
```

- q0 transitions to q1 on b.
- q1 transitions to q2 on a.
- q2 transitions to itself on a.
- q3 transitions to q4 on !.
Finite State Automata
Simple Example

q0 \rightarrow b \rightarrow q1 \rightarrow a \rightarrow q2 \rightarrow a \rightarrow q3 \rightarrow ! \rightarrow q4
Finite State Automata
Formal Definition

- \( Q = \{q_0, q_1, \ldots q_{N-1}\} \) = a finite set of states
- \( \Sigma = \{i_0, i_1, \ldots i_{M-1}\} \) = an alphabet, i.e., a finite set of symbols
- \( q_0 \) = a designated start state
- \( F \subseteq Q \) = a designated set of final states
- \( \delta(q,i) \) = a transition function that, given a state and a symbol from the alphabet, returns the new state
Finite State Automata

Implementation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- q0 -> q1 with input b
- q1 -> q2 with input a
- q2 -> q3 with input a
- q3 -> q4 with input !
Finite State Automata
Implementation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

from state {row}, on seeing symbol {column}, go do state {cell}
Finite State Automata
Implementation

from state {row},
on seeing symbol {column},
go do state {cell}
Finite State Automata
Implementation

designate subset of states as “accept states”

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>–</td>
<td>4</td>
</tr>
<tr>
<td>4*</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Diagram:
- States: q0, q1, q2, q3, q4
- Transitions:
  - q0 → q1 on b
  - q1 → q2 on a
  - q2 → q3 on a
  - q3 → q4 on !
Finite State Automata

Implementation

recognize(tape, machine) —> accept or reject

index ← beginning of tape
cur_state ← initial state of machine

loop
  if end of input reached then
    if cur_state in FinalStates then
      return Accept
    else
      return Reject
  else
    if Transition[cur_state, tape[index]] is empty then
      return Reject
    else
      cur_state <- Transition[cur_state, tape[index]]
      index += 1
  end
Finite State Automata

Implementation

recognize(tape, machine) -> accept or reject
index <- beginning of tape
cur_state <- initial state of machine
loop
  if end of input reached then
    if cur_state in FinalStates then
      return Accept
    else
      return Reject
  elif Transition[cur_state, tape[index]] is empty then
    return Reject
  else
    cur_state <- Transition[cur_state, tape[index]]
    index += 1
  end
Finite State Automata

Implementation

recognize(tape, machine) -> accept or reject
index <- beginning of tape
cur_state <- initial state of machine

loop
    if end of input reached then
        if cur_state in FinalStates then
            return Accept
        else
            return Reject
    elif Transition[cur_state, tape[index]] is empty then
        return Reject
    else
        cur_state <- Transition[cur_state, tape[index]]
        index += 1
end
Finite State Automata

Implementation

\[ \text{recognize(tape, machine)} \rightarrow \text{accept or reject} \]

\[ \text{index} \leftarrow \text{beginning of tape} \]

\[ \text{cur_state} \leftarrow \text{initial state of machine} \]

\[ \text{loop} \]

if end of input reached then
  if \text{cur_state} in \text{FinalStates} then
    return Accept
  else
    return Reject
else
  \text{cur_state} <- \text{Transition[cur_state, tape[index]]}
  \text{index} += 1
end
Finite State Automata
Non-Determinism
Finite State Automata
Non-Determinism

• Two ways of representing:
Finite State Automata
Non-Determinism

- Two ways of representing:
  - Multiple links out of the same node

![Diagram of a non-deterministic finite state automaton]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2,3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finite State Automata

Non-Determinism

- Two ways of representing:
  - Multiple links out of the same node
  - Epsilon transitions

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
<th>b</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>2,3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>4*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Finite State Automata
Non-Determinism

- Two ways of representing:
  - Multiple links out of the same node
  - Epsilon transitions
- Solvable using e.g., parallelism

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
<th>b</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>2,3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>4*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Diagram:
- States: q0, q1, q2, q3, q4
- Transitions:
  - q0 to q1 on 'b'
  - q1 to q2 on 'a'
  - q2 to q3 on 'a'
  - q3 to q4 on '!'
Goal: Accept valid English nouns (singular or plural)
Finite State Automata

Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

doesn't accept singular nouns!
(cat, dog, rat)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)
Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

noun-lemma: \{cat, dog, rat\}
plural-marker: \{s\}
Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

noun-lemma: {cat, dog, rat, mouse}
plural-marker: {s}

problems?
Finite State Automata

English plurals

noun-lemma: \{cat, dog, rat, mouse\}
plural-marker: \{s\}

accepts "mouses"
doesn't accept "mice"

Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

noun-lemma: \{cat, dog, rat, mouse\}
plural-marker: \{s\}

accepts “mouses”
doesn’t accept “mice”
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

- **reg-noun**: \{cat, dog, rat\}
- **irreg-noun**: \{mouse\}
- **plural-marker**: \{s\}

- **accepts** "mouses"
- **doesn't accept** "mice"
Goal: Accept valid English nouns (singular or plural)
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

reg-noun: {cat, dog, rat}
irreg-noun: {mouse, mice}
plural-marker: {s}

accepts "mouses"
doesn't accept "mice"
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

- **reg-noun:** {cat, dog, rat}
- **irreg-noun:** {mouse, mice}
- **plural-marker:** {s}

Diagram:

- **q0**
  - Transition on **reg-noun** to **q1**
- **q1**
  - Transition on **plural-marker** to **q2**
  - Transition on **ε** to **q2**
- **q2**
  - Transition on **irreg-noun** to **q0**

Problems?
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

- **reg-noun**: {cat, dog, rat}
- **irreg-noun**: {mouse, mice}
- **plural-marker**: {s}
- **obsession-marker**: {athon, mania, aholic}

- **q0**
  - Transition on **reg-noun** to **q1**

- **q1**
  - Transition on **plural-marker** to **q2**
  - Transition on **ε** from **q1** to **q2**

- **q2**
  - Transition on **irreg-noun**

Examples:
- dogmania, mouseathon, cataholic...
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

differentiate singular and plural...

![Diagram of a finite state automaton with states q0, q1, and q2. The transitions are labeled with reg-noun, plural-marker, and irreg-noun. The initial state is q0, and the accepting state is q2. The goal is to accept valid English nouns (singular or plural).]
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

- reg-noun: \{cat, dog, rat\}
- irreg-sing: \{mouse\}
- irreg-plur: \{mice\}
- plural-marker: \{s\}
- obsession-marker: \{athon, mania, aholic\}

\[
\begin{array}{c}
q_0 \xrightarrow{\text{reg-noun}} q_1 \xrightarrow{\text{plural-marker}} q_2 \\
q_2 \xleftarrow{\epsilon} q_1 \xleftarrow{\text{irreg-sing}} q_0 \\
q_2 \xleftarrow{\text{irreg-plur}} q_1
\end{array}
\]

anyone can go straight to accept
Finite State Automata

English plurals

- **Goal:** Accept valid English nouns (singular or plural)

- **Regular nouns** (reg-noun): {cat, dog, rat}
- **Irregular singular** (irreg-sing): {mouse}
- **Irregular plural** (irreg-plur): {mice}
- **Plural marker** (plural-marker): {s}
- **Obsession marker** (obsession-marker): {athlon, mania, aholic}

Diagram:

- Transition from state q0 to q1 on *reg-noun*.
- Transition from q1 to q2 on *plural-marker*.
- Transition from q2 on *irreg-plur*.
- Transition from q3 on *irreg-sing*.

- Minimal **epsilon transitions** (ε) to reach states q1 and q2.

The diagram represents a finite state automaton designed to recognize valid English nouns, including their singular and plural forms, with optional obsession markers.
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

Singular nouns can optionally add "obsession markers"

Diagram:

$q0 \xrightarrow{\text{reg-noun}} q1 \xrightarrow{\varepsilon} q2$
$q1 \xrightarrow{\varepsilon} q3 \xrightarrow{\varepsilon} q2$
$q3 \xrightarrow{\text{irreg-sing}} q1 \xrightarrow{\varepsilon} q2$
$q3 \xrightarrow{\text{irreg-plur}} q2$

Example: cat
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

reg-noun: \{cat, dog, rat\}
irreg-sing: \{mouse\}
irreg-plur: \{mice\}
plural-marker: \{s\}
obsession-marker: \{athon, mania, aholic\}

Singular nouns can optionally add "obsession markers"
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

- **reg-noun**: \{cat, dog, rat\}
- **irreg-sing**: \{mouse\}
- **irreg-plur**: \{mice\}
- **plural-marker**: \{s\}
- **obsession-marker**: \{athon, mania, aholic\}

Singular nouns can optionally add "obsession markers".

- **q0**: reg-noun
- **q1**: plural-marker
- **q2**:
- **q3**: irreg-sing, irreg-plur

Example: "ratmania"
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

reg-noun: \{cat, dog, rat\}
irreg-sing: \{mouse\}
irreg-plur: \{mice\}
plural-marker: \{s\}
obsession-marker: \{athon, mania, aholic\}

Singular nouns can optionally add "obsession markers"

ratsmania?
Finite State Automata

English plurals

Goal: Accept valid English nouns (singular or plural)

reg-noun: {cat, dog, rat}
irreg-sing: {mouse}
irreg-plur: {mice}
plural-marker: {s}
obsession-marker: {athon, mania, aholic}

Singular nouns can optionally add "obsession markers"

rats mania?
Finite State Transducers

- FSAs can tell you if a string follows the rules or not
- FSAs don’t give you the **parse**: i.e., record of the rules that a particular input uses
- FSTs: simple extension of FSAs which **produces a parse as output**
Finite State Transducers

- Two tapes: an input tape and an output tape
- Used to "translate" one language to another
- E.g., "translate" a word (cats) into a morphological parse (cat +N +pl)
Finite State Transducers

If you read this symbol on the tape...

q0 -> q1 (b:a)
q1 -> q2 (a:b)
q2 -> q3 (a:b)
q3 -> q4 (a:b)
q3 -> q4 (::)
Finite State Transducers

...output this symbol on the other tape...
Finite State Transducers

...and then update to this state.
Finite State Transducer

If input string is “accepted”, the FST will end with a valid translation on the output. Otherwise, it will reject (and output tape is ignored),

<table>
<thead>
<tr>
<th>b</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>.</td>
</tr>
</tbody>
</table>
Finite State Transducer

Input tape: cats #
Output tape: 

States:
- q0: reg-noun
- q1: ε:+N
- q2: irreg-sing
- q3: ε:+N
- q4: ε:+N
- q5: ε:+N
- q6: ε:+N
- q7: #:+Sg

Transitions:
- q0 to q1: reg-noun
- q1 to q4: ε:+N
- q2 to q5: ε:+N
- q3 to q6: ε:+N
- q4 to q7: #:+Sg
- q5 to q7: #:+Sg
- q6 to q7: #:+Pl
- q7 to q7: # = end of token
Finite State Transducer

reg-noun:reg-noun

irreg-plur:irreg-plur

irreg-sing:irreg-sing

input tape: cats #

output tape:
Finite State Transducer

input tape: cats #

output tape:
Finite State Transducer

input tape: cats #  
output tape: cat
Finite State Transducer

input tape: cats #  output tape: cat

- reg-noun: reg-noun
- irreg-sing: irreg-sing
- irreg-plur: irreg-plur

q0 -> q1 -> q4
q2 -> q5
q3 -> q6
q4 -> q7

# = end of token
s#: +Pl
#: +Sg
#: +Pl
#: +Sg
ε: +N
Finite State Transducer

- **q0**: reg-noun:reg-noun
- **q1**: ε:+N
- **q2**: irreg-sing:irreg-sing
- **q3**: irreg-plur:irreg-plur
- **q4**: ε:+N
- **q5**: ε:+N
- **q6**: ε:+N
- **q7**: # = end of token

Input tape: **cats #**
Output tape: **cat**
Finite State Transducer

input tape:  c a t s #  output tape:  c a t +N
Finite State Transducer

reg-noun:reg-noun

irreg-sing:irreg-sing

irreg-plur:irreg-plur

# = end of token

input tape: cats #

output tape: cat +N
Finite State Transducer

Input tape: cats #

Output tape: cat +N

States:
- q0
- q1
- q2
- q3
- q4
- q5
- q6
- q7

Transitions:
- reg-noun:reg-noun
- irreg-sing:irreg-sing
- irreg-plur:irreg-plur

Symbols:
- # = end of token
- s#:+Pl
- #:+Sg
- #:+Pl
- $t$:+N
Finite State Transducer

- **State q0**: Transition on reg-noun:reg-noun
- **State q1**: Transition on $\epsilon$:+$N$
- **State q2**: Transition on irreg-sing:irreg-sing
- **State q3**: Transition on irreg-plur:irreg-plur
- **State q4**: Transition on $\epsilon$:+$N$
- **State q5**: Transition on $\epsilon$:+$N$
- **State q6**: Transition on $\epsilon$:+$N$
- **State q7**: Transition on $\epsilon$:+$N$ and #:+Pl

Input tape: **cats #**

Output tape: **cat t +N +Pl**

# = end of token

s#:+Pl
Finite State Transducer

reg-noun:reg-noun

q0 -> q1: $\epsilon$:+$N$

irreg-sing:irreg-sing

q2: $\epsilon$:+$N$

irreg-plur:irreg-plur

q3: $\epsilon$:+$N$

q4 -> q7: #:+Sg

q5: #:+Sg

q6: #:+Pl

# = end of token

s#:+Pl

input tape: cats#

output tape: cat +N +Pl
Topics

- (English) Morphology 101
- Finite State Machines
- **Subword Tokenization**
- [Saving for Later] Supporting all languages!
Subword Tokenization

• Problem:
  • Tokenizers result in lots of unknown words (<OOV>s)
  • Morphological analysis can help break, but:
    • Running morphological analyzers is expensive and language specific
    • Rare words might be “guessable” from cues other than standard linguistic analysis
  • “The food was disvotable”
Subword Tokenization

• Solution:
  • “Subwords”
    • aka: “WordPieces”, “SentencePieces”, “BytePairEncoding”
  • Algorithmically identify common sub sequences of characters and merge them into atomic units
  • *Not* linguistically informed, based on data-compression algorithms
  • But works quite well in practice!
Subword Tokenization
Byte Pair Encoding

Neural Machine Translation of Rare Words with Subword Units

**Rico Sennrich** and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk,bhaddow@inf.ed.ac.uk
Subword Tokenization

Byte Pair Encoding

**Algorithm 1 Learn BPE operations**

```python
import re, collections

def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i], symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'{}'.format(bigram) + bigram + r'{}/'.format(bigram))
    for word in v_in:
        w_out = p.sub(' '.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out
vocab = ['1 o w </w>' : 5, '1 o w e r </w>' : 2, 'n o w e s t </w>' : 6, 'w i d e s t </w>' : 3]

num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
print(best)
```

---

**Neural Machine Translation of Rare Words with Subword Units**

Rico Sennrich and Barry Haddow and Alexandra Birch
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Break our words into characters.

Define a number of iterations to run (“hyperparameter”)
Subword Tokenization

Byte Pair Encoding

### Algorithm 1 Learn BPE operations

```python
import re, collections

def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i]+symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'{}'.format(bigram) + bigram + r'{}\S'.format(bigram))
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out

vocab = {'i o w /<w/>': 5, 'i o w e r /<w/>': 2, 'n o w o s t /<w/>': 6, 'w i d o s t /<w/>': 3}
num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
print(best)
```

---

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and Barry Haddow and Alexandra Birch
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Count the frequency of all the pairs of symbols
Subword Tokenization

Byte Pair Encoding

Algorithm 1 Learn BPE operations

import re, collections

def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i]+symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, vocab):
    v_out = {}
    bigram = re.escape(''.join(pair))
    p = re.compile(r'(?<!\%s)' % bigram + r'(?!!\S)')
    for word in vocab:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = vocab[word]
    return v_out

vocab = {'i o w <:/w>' : 5, 'l o w e r <:/w>' : 2,
         'n o w e s t <:/w>' : 6, 'w i d e s t <:/w>' : 3}

num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    print(best)

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and Barry Haddow and Alexandra Birch
School of Informatics, University of Edinburgh
{rico.sennrich, a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Merge the most frequent pair of characters, treating them as a new token.
Subword Tokenization

Byte Pair Encoding

Algorithm 1 Learn BPE operations

```python
import re, collections

def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i], symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'\b{}\b'.format(bigram))
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out

vocabs = ['i o w</w>', 'l o w e r</w>', 'n e w s t</w>', 'w i d e s t</w>']
num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
print(best)
```

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and Barry Haddow and Alexandra Birch
School of Informatics, University of Edinburgh
{rico.sennrich, a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

Repeat
Subword Tokenization
Byte Pair Encoding

1460 is fun
she is funny
they’re 14
$60 refund
Subword Tokenization
Byte Pair Encoding

1 4 6 0 # i s # f u n <eos>
s h e # i s # f u n n y <eos>
t h e y ’ r e # 1 4 <eos>
$ 6 0 # r e f u n d <eos>

Break our words into characters.
Subword Tokenization

Byte Pair Encoding

Build a vocabulary

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t}
Subword Tokenization
Byte Pair Encoding

Build a character count dictionary

<table>
<thead>
<tr>
<th>Character</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>un</td>
<td>3</td>
</tr>
<tr>
<td>fu</td>
<td>3</td>
</tr>
<tr>
<td># f</td>
<td>2</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
</tr>
<tr>
<td># i</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>2</td>
</tr>
<tr>
<td>re</td>
<td>2</td>
</tr>
<tr>
<td>0 #</td>
<td>2</td>
</tr>
<tr>
<td>e #</td>
<td>2</td>
</tr>
<tr>
<td>he</td>
<td>2</td>
</tr>
<tr>
<td>s #</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>{all the rest}</td>
<td>1</td>
</tr>
</tbody>
</table>

vocab = \{# e n f s u i h 1 0 r 4 6 y ‘ $ d t\}
Subword Tokenization

Byte Pair Encoding

merge most frequent pair

1 4 6 0 # i s # fu n <eos>

s h e # i s # fu n n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f u n d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>un</td>
<td>3</td>
</tr>
<tr>
<td>fu</td>
<td>3</td>
</tr>
<tr>
<td># f</td>
<td>2</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
</tr>
<tr>
<td># i</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>2</td>
</tr>
<tr>
<td>re</td>
<td>2</td>
</tr>
<tr>
<td>0 #</td>
<td>2</td>
</tr>
<tr>
<td>e #</td>
<td>2</td>
</tr>
<tr>
<td>he</td>
<td>2</td>
</tr>
<tr>
<td>s #</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>{all the rest}</td>
<td>1</td>
</tr>
</tbody>
</table>
Subword Tokenization
Byte Pair Encoding

merge most frequent pair

```
1 4 6 0 # i s # f un <eos>
s h e # i s # f un n y <eos>
t h e y ’ r e # 1 4 <eos>
$ 6 0 # r e f un d <eos>
```

vocab = \{# e n f s u i h 1 0 r 4 6 y ‘ $ d t un\}

merges = \{(u,n)\}

<table>
<thead>
<tr>
<th>un</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>fu</td>
<td>3</td>
</tr>
<tr>
<td># f</td>
<td>2</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
</tr>
<tr>
<td># i</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>2</td>
</tr>
<tr>
<td>re</td>
<td>2</td>
</tr>
<tr>
<td>0 #</td>
<td>2</td>
</tr>
<tr>
<td>e #</td>
<td>2</td>
</tr>
<tr>
<td>he</td>
<td>2</td>
</tr>
<tr>
<td>s #</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>{all the rest}</td>
<td>1</td>
</tr>
</tbody>
</table>
**Subword Tokenization**

**Byte Pair Encoding**

<table>
<thead>
<tr>
<th>fun</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>2</td>
</tr>
<tr>
<td>re</td>
<td>2</td>
</tr>
<tr>
<td>e #</td>
<td>2</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
</tr>
<tr>
<td># f</td>
<td>2</td>
</tr>
<tr>
<td># i</td>
<td>2</td>
</tr>
<tr>
<td>1 4</td>
<td>2</td>
</tr>
<tr>
<td>6 0</td>
<td>2</td>
</tr>
<tr>
<td>s #</td>
<td>2</td>
</tr>
<tr>
<td>0 #</td>
<td>2</td>
</tr>
<tr>
<td>{all the rest}</td>
<td>1</td>
</tr>
</tbody>
</table>

vocabulary

m = \{# e n f s u i h 1 0 r 4 6 y ' $ d t u n} 

merges = \{(u,n)\}
Subword Tokenization

Byte Pair Encoding

merge most frequent pair

<table>
<thead>
<tr>
<th>fun</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>2</td>
</tr>
<tr>
<td>re</td>
<td>2</td>
</tr>
<tr>
<td>e #</td>
<td>2</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
</tr>
<tr>
<td># f</td>
<td>2</td>
</tr>
<tr>
<td># i</td>
<td>2</td>
</tr>
<tr>
<td>1 4</td>
<td>2</td>
</tr>
<tr>
<td>6 0</td>
<td>2</td>
</tr>
<tr>
<td>s #</td>
<td>2</td>
</tr>
<tr>
<td>0 #</td>
<td>2</td>
</tr>
<tr>
<td>{all the rest}</td>
<td>1</td>
</tr>
</tbody>
</table>

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un}
merges = {(u,n)}

1 4 6 0 # is # fun <eos>

s h e # is # fun n y <eos>

t h e y ’ r e # 1 4 <eos>

$ 6 0 # r e f un d <eos>
### Byte Pair Encoding

#### Subword Tokenization

**merge most frequent pair**

- `1 4 6 0 # i s # fun <eos>`
- `s h e # i s # fun n y <eos>`
- `t h e y ' r e # 1 4 <eos>`
- `$ 6 0 # r e fun d <eos>`

**vocab = {# e n f s u i h 1 0 r 4 6 y ' $ d t un fun}**

**merges = {(u,n), (f, un)}**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fun</strong></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>he</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>re</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>e #</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>is</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong># f</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong># i</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>1 4</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>6 0</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>s #</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>0 #</strong></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>{all the rest}</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Subword Tokenization

Byte Pair Encoding

recompute pair frequencies

1460 # is # fun <eos>

she # is # funny <eos>

they're # 14 <eos>

$60 # re fun d <eos>

vocab = {# e n f s u i h 1 0 r 4 6 y ' $ d t un fun}
merges = {(u,n), (f, un)}
Subword Tokenization
Byte Pair Encoding

merge most frequent pair

vocabulary = {# e n f s u i h 1 0 r 4 6 y ‘$ d t un fun}  
merges = {(u,n), (f, un)}
**Subword Tokenization**

*Byte Pair Encoding*

merge most frequent pair

```
1 4 6 0 # i s # fun <eos>
```

```
s h e # i s # fun n y <eos>
```

```
t h e y ’ re # 1 4 <eos>
```

```
$ 6 0 # re fun d <eos>
```

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun re}

merges = {(u,n), (f, un), (r, e)}
Subword Tokenization

Byte Pair Encoding

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#)}

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t u n f u n r e 14 i s i s# h e 0# 60#}

six more iterations

14 60# is# fun <eos>

s he # is# fun n y <eos>

t he y ’ re # 14 <eos>

$ 60# re fun d <eos>
Subword Tokenization

Byte Pair Encoding

vocabulary

\[ \text{vocabulary} = \{ #, e, n, f, s, u, i, h, 1, 0, r, 4, 6, y, ' \} \]

merges

\[ \text{merges} = \{(u, n), (f, un), (r, e), (1, 4), (i, s), (is, #), (h, e), (0, #), (6, 0#), (is#, \text{fun})\} \]

seven more iterations
Subword Tokenization
Byte Pair Encoding

To parse new inputs, just apply merges in order

vocabulary = \{# e n f s u i h 1 0 r 4 6 y ' $ d t u n f u n r e 14 i s i s# h e 0# 60# is#f u n\}

merges = \{(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)\}

140 red fungi
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

vocabulary = {# e n f s u i h 1 0 r 4 6 y ‘
$ d t u n f u n r e 1 4 i s i s # h e 0 # 6 0 #
is#fun}

merges = {u,n, (f, un), (r, e), (1,4) (i, s) (is, #)
(h, e) (0, #) (6, 0#) (is#, fun)}
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

\[
\text{vocab} = \{\#, e, n, f, s, u, i, h, 1, 0, r, 4, 6, y, \ ' \$
\text{d, t, un, fun, re, 14, is, is#, he, 0#, 60#, is#, fun}\}
\]

\[
\text{merges} = \{(u,n), (f, un), (r, e), (1,4) (i, s) (is, #)
\}
\]

\[
\text{h, e) (0, #) (6, 0#) (is#, fun)\}
\]
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

vocabulary = {# e n f s u i h 1 0 r 4 6 y ' $ d t un fun re 14 is is# he 0# 60# is# fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)}
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

\[
\text{vocab} = \{\#, \text{e, n, f, s, u, i, h, 1, 0, r, 4, 6, y, '}, \text{d, t, un, fun, re, 14, is, is, #, he, 0, 60, is, fun}\}
\]

\[
\text{merges} = \{(u, n), (f, un), (r, e), (1, 4), (i, s), (is, #), (h, e), (0, #), (6, 0#, is, fun)\}
\]
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

vocab = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun re 14 is is# he 0# 60# is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)}
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

\[
\text{merges} = \{(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)\}
\]

\[
\text{vocab} = \{# e n f s u i h 1 0 r 4 6 y \; \$ \; d t \; u n \; \text{fun} \; \text{re} \; 14 \; \text{is} \; \text{is#} \; \text{he} \; 0\# \; 60\# \; \text{is#fun}\}
\]
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order.

\[ \text{merges} = \{(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)\} \]

\[ \text{vocab} = \{# e n f s u i h 1 0 r 4 6 y ' \}$ d t u n f u n r e 14 i s i s# h e 0# 60# i s# f u n} \]
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

vocab = {# e n f s u i h 1 0 r 4 6 y ' $ d t u n f u n r e 1 4 i s i s # h e 0 # 6 0 # i s # f u n}

merges = {(u, n), (f, un), (r, e), (1, 4) (i, s) (is, #) (h, e) (0, #) (6, 0 #) (is#, fun)}
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

vocabulary:

```
# e n f s u i h 1 0 r 4 6 y ' $ d t u n f u n r e 14 i s i s# h e 0# 60# i s# f u n
```

merges:

```
{(u, n), (f, un), (r, e), (1, 4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)}
```
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

vocab = {# e n f s u i h 1 0 r 4 6 y ' $ d t u n f u n r e 14 i s i s# h e 0# 60# i s# f u n}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)}
Subword Tokenization

Byte Pair Encoding

To parse new inputs, just apply merges in order

\[
\text{vocab} = \{#, e, n, f, s, u, i, h, 1, 0, r, 4, 6, y, ' \}
\]
\[
\text{d, t, un, fun, re, 14, is, is#, he, 0, 60#, is#fun}\}
\]

\[
\text{merges} = \{(u, n), (f, un), (r, e), (1, 4), (i, s), (is, #), (h, e), (0, #), (6, 0#), (is#, fun)\}
\]
Subword Tokenization
Byte Pair Encoding

vocabulary = {# e n f s u i h 1 0 r 4 6 y ‘ $ d t un fun re 14 is is# he 0# 60# is#fun}

merges = {(u,n), (f, un), (r, e), (1,4) (i, s) (is, #) (h, e) (0, #) (6, 0#) (is#, fun)}

14 0# re d # fun <oov> i

Note: in this toy example, “g” is actually <oov>.
In reality, character vocabs are very large so nothing is really <oov>.
Subword Tokenization vs. Morphological Analysis

she is funny <eos>

she be+1p+sing fun+adj <eos>

they’re 14 <eos>

they be+1p+sing 14 <eos>