

Chapter 1 Introduction to Natural Language Processing (NLP)

What is NLP?

การประมวลผลทางภาษาธรรมชาติ (NLP) คือสาขาย่อยใน:

- Artificial Intelligence: รวมความสามารถของมนุษย์เข้าสู่เครื่องจักร
- Linguistics: ศึกษาและทำให้เข้าใจระบบของภาษามนุษย์ โดยใช้แนวคิดทฤษฎี และวิธีการของนักภาษาศาสตร์
- Cognitive Science: ศึกษาความคิด การเรียนรู้ การตัดสินใจ การตัดสินใจของมนุษย์
- Computer Science: นำความรู้ด้านการคำนวณ การเขียนโปรแกรมเข้าด้วยกันเพื่อใช้แก้ปัญหาต่าง ๆ

NLP ช่วยให้เครื่องจักรสามารถวิเคราะห์และสร้างข้อมูลภาษาธรรมชาติได้

NLP เริ่มต้นจากข้อความที่ไม่มีโครงสร้าง (unstructured text) ทำให้ข้อมูลที่ไม่มีโครงสร้าง สามารถเข้าใจโดยมนุษย์และคอมพิวเตอร์ได้

What is Unstructured Text?

Unstructured" ในบริบทของข้อมูลหมายถึง?

- ข้อความมักถูกเรียกว่าข้อมูลที่ไม่มีโครงสร้าง (unstructured data)
- จริง ๆ แล้วข้อความมีโครงสร้างอยู่เบื้องหลัง
- จริง ๆ แล้วข้อความมีโครงสร้าง เช่น การสะกดคำ, การใช้เครื่องหมายวรรคตอน, การสร้างประโยคอย่างถูกต้อง และการพัฒนาความคิดอย่างถูกต้อง
- แต่ถึงอย่างนั้น ข้อความก็ยังไม่ถูกพิจารณาว่าเป็นโครงสร้างในสายตาของคอมพิวเตอร์
- ข้อความไม่สามารถใส่ลงในระบบจัดการฐานข้อมูลมาตรฐาน (DBMS)

ตัวอย่างของ DBMS: (MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, SQLite, MongoDB)

"Unstructured" ในบริบทของข้อมูลหมายถึง:

ข้อมูลที่ไม่ได้จัดเก็บในรูปแบบที่กำหนดหรือมาตรฐานเฉพาะ เช่น ข้อความในอีเมล โซเชียลมีเดีย หรือเอกสารที่ไม่มีโครงสร้างชัดเจน ทำให้ไม่สามารถจัดการและวิเคราะห์ได้โดยตรงด้วยระบบฐานข้อมูลแบบดั้งเดิม (DBMS) ข้อมูลเหล่านี้ยังไม่ได้จัดรูปแบบเป็นแถวหรือคอลัมน์ที่ชัดเจนเหมือนในฐานข้อมูลที่มีโครงสร้าง ทำให้การประมวลผลโดยเครื่องจักรเป็นเรื่องท้าทายมากขึ้น

What is Structured Text?

ข้อมูลที่มีโครงสร้าง

- ข้อมูลที่มีโครงสร้างคือข้อมูลที่สามารถใส่ลงในระบบจัดการฐานข้อมูลมาตรฐาน (DBMS) ได้อย่างเหมาะสม
 - คอมพิวเตอร์คาดหวังให้ข้อมูลอยู่ในรูปแบบของเรคคอร์ด (มีคีย์และคุณลักษณะอื่น ๆ)

หนึ่งในคำถามที่น่าสนใจคือ:

- จะทำอย่างไรเพื่อแปลงข้อมูลที่ไม่มีโครงสร้างให้เป็นข้อมูลที่มีโครงสร้าง?
 - การแยกข้อมูล (Data Parsing): แยกข้อมูลจากแหล่งที่ไม่มีโครงสร้าง
 - การจัดรูปแบบ (Data Formatting): จัดระเบียบข้อมูลให้อยู่ในรูปแบบตารางหรือฐานข้อมูล
 - การใช้ NLP: ใช้เทคนิคประมวลผลภาษาธรรมชาติ เช่น การแยกคำ (Tokenization) และการวิเคราะห์ความหมาย (Semantic Analysis)
 - การใช้เครื่องมือ (Tools): ใช้เครื่องมือเช่น Google Cloud NLP, IBM Watson ในการแปลงข้อมูล

NLU vs NLG

- NLU (Natural Language Understanding) คือการแปลงภาษามนุษย์ให้อยู่ในรูปแบบที่เครื่องเข้าใจได้
 - NLG (Natural Language Generation) คือการแปลงข้อมูลที่เครื่องสร้างขึ้นให้อยู่ในภาษาที่มนุษย์เข้าใจได้
-

NLP Goals

- เป้าหมายหลักของ NLP คือการออกแบบและสร้างระบบคอมพิวเตอร์ที่สามารถ
 - ประมวลผลและวิเคราะห์ภาษาธรรมชาติ เช่น ภาษาไทยหรือภาษาอังกฤษ
 - เข้าใจเนื้อหาของข้อมูลที่ป้อนเข้า (เช่น คำพูด)
 - สร้างผลลัพธ์ในรูปแบบของภาษาธรรมชาติ

Level of understanding in NLP (ระดับความเข้าใจใน NLP)

- Phonological Analysis:** การวิเคราะห์เสียงพูด (Interpreting speech sounds)
- Optical Character Recognition (OCR):**
 - OCR คือกระบวนการแปลงภาพของข้อความที่พิมพ์หรือเขียนมือให้อยู่ในรูปแบบข้อความที่เครื่องจักรเข้าใจได้ เช่น จากเอกสารที่สแกนหรือภาพถ่ายเอกสาร
- Tokenization:**
 - เป็นขั้นตอนแรกใน NLP
 - Tokenizer จะแยกข้อมูลที่ไม่มีโครงสร้างและข้อความภาษาธรรมชาติออกเป็นชิ้นส่วนเล็กๆ เช่น การแยกย่อหน้าเป็นประโยคหรือคำ

ข้อความในรูปภาพแปลได้ดังนี้:

1. Morphological Analysis (การวิเคราะห์ทางสัณฐานวิทยา)

- ศึกษาและทำความเข้าใจโครงสร้างของคำ (Structure of words)
- ระบุว่าคำถูกสร้างขึ้นมาอย่างไรโดยการใช้หน่วยคำที่เป็นหน่วยความหมายที่เล็กที่สุด (Morphemes)
- สามารถแบ่งคำออกเป็นสามส่วนย่อยได้ (คำนำหน้า, รากคำ, และคำต่อท้าย) เช่น คำว่า "unhappiness"
 - คำนำหน้า: un- (เช่น unhappy (adj) -> unhappiness (n))
 - รากคำ: happy (adj)

- คำต่อท้าย: -ness (เช่น happiness (n) -> ความสุข)

2. Lexical Analysis (การวิเคราะห์โครงสร้างของคำ)

- วิเคราะห์โครงสร้างของคำ
- มีสองเทคนิคที่ใช้ดังนี้:
 - **Stemming:** ลดรูปคำให้เหลือเพียงรากศัพท์ของคำ (dictionary root) เช่น "running" ลดรูปเป็น "run" หรือ "studies" ลดรูปเป็น "studi"
 - **Lemmatization:** ลดรูปและพิจารณาความหมายของคำในบริบท เช่น "better" ถูกลดรูปเป็น "good" แทนที่จะเป็น "bet" หรือ "studies" ถูกลดรูปเป็น "study"

3. Syntactic Analysis (Parsing) (การวิเคราะห์ในเชิงโครงสร้าง)

- วิเคราะห์ภาษาธรรมชาติตามหลักไวยากรณ์เพื่อหาความหมายในพจนานุกรมของประโยค
- เข้าใจรูปแบบของประโยค (Subject, Verb, Object, Preposition)

4. Semantic Analysis (การวิเคราะห์ในเชิงความหมาย)

- เข้าใจบริบทของข้อความและเข้าใจอารมณ์
- ใช้ในเครื่องมือ เช่น การแปลภาษาอัตโนมัติ แชทบอท เครื่องมือค้นหา และการวิเคราะห์ข้อความ

5. Discourse Analysis (การวิเคราะห์เนื้อความ)

- วิเคราะห์ลักษณะของข้อความทั้งหมดที่มีความหมายโดยการเชื่อมโยงระหว่างประโยคที่เป็นส่วนประกอบ
- เน้นไปที่ลักษณะของพฤติกรรมทางภาษาศาสตร์ เช่น
 - การศึกษารูปแบบการออกเสียง
 - โครงสร้างประโยค
 - การแทนความหมาย
 - การแก้ไขความคลุมเครือ
- ตัวอย่าง: "John go to school, he loves NLP course." (รู้ว่าคุณนี่กำลังพูดถึงอะไร)

6. Pragmatic Analysis (การวิเคราะห์ในเชิงปฏิบัติการ)

- วิเคราะห์ความหมายพื้นฐานของข้อความ
- อธิบายว่ามีความหมายเพิ่มเติมอย่างไรในข้อความ
- ต้องการความรู้เกี่ยวกับโลก (เช่น ความเข้าใจในเจตนา แผนการ และเป้าหมาย)
- ตัวอย่าง:
 - "Close the window?" (คำแนะนำหรือคำสั่ง)
 - "Do you have a watch?" (หมายถึงการถามเวลา)

1. Morphological Analysis: วิเคราะห์โครงสร้างของคำและส่วนประกอบ (เช่น คำนำหน้า, รากคำ, คำต่อท้าย)

2. Lexical Analysis: วิเคราะห์ความหมายของคำและลดรูปคำให้เหลือรากศัพท์

3. Syntactic Analysis: วิเคราะห์โครงสร้างไวยากรณ์ของประโยคตามหลักไวยากรณ์

4. Semantic Analysis: วิเคราะห์ความหมายและอารมณ์ของข้อความ

5. Discourse Analysis: วิเคราะห์การเชื่อมโยงและความหมายของข้อความทั้งหมด

6. Pragmatic Analysis: วิเคราะห์ความหมายที่แท้จริงและเจตนาของข้อความในบริบทต่าง ๆ

Example of NLP Tools: MonkeyLearn

1. Topic Labeling:

- ช่วยในการระบุหัวข้อของข้อความ เช่น การจำแนกว่าเป็นปัญหาการสั่งซื้อ (Order Issue)

2. Sentiment Analysis:

- วิเคราะห์อารมณ์ของข้อความ เช่น การประเมินว่าเป็นความคิดเห็นเชิงบวก เชิงลบ หรือเป็นกลาง

3. Intent Detection:

- ตรวจจับเจตนาของข้อความ เช่น การระบุว่าผู้ใช้สนใจในการขอเดโม (Interested in Demo)

4. Feature Extraction:

- ดึงข้อมูลคุณสมบัติสำคัญจากข้อความ เช่น ขนาดหน้าจอ, โปรเซสเซอร์, หน่วยความจำ

5. Keyword Extraction:

- ดึงคำสำคัญออกจากข้อความ เช่น การดึงชื่อผลิตภัณฑ์หรือแบรนด์

6. Entity Extraction:

- แยกหน่วยข้อมูลที่มีความหมายเฉพาะ เช่น ชื่อบุคคล สถานที่ หรือชื่อบริษัท

MonkeyLearn ช่วยให้คุณสามารถวิเคราะห์และจัดการข้อมูลข้อความได้อย่างมีประสิทธิภาพ ไม่ว่าจะเป็นการจัดหมวดหมู่ การวิเคราะห์อารมณ์ หรือการดึงข้อมูลเฉพาะจากข้อความ ทั้งหมดนี้สามารถทำได้โดยไม่จำเป็นต้องมีความรู้เชิงลึกในการเขียนโปรแกรม

NLP Libraries: NLTK

NLTK (Natural Language Toolkit) เป็นไลบรารีในภาษา Python ที่ออกแบบมาเพื่อช่วยในการประมวลผลภาษาธรรมชาติ (Natural Language Processing - NLP) มีเครื่องมือและฟีเจอร์ต่าง ๆ ที่ช่วยให้นักพัฒนาสามารถจัดการกับข้อมูลข้อความได้อย่างมีประสิทธิภาพ ฟีเจอร์หลักของ NLTK ได้แก่:

1. Tokenizing:

- ใช้ในการแบ่งข้อความออกเป็นคำ (word) หรือประโยค (sentence) เพื่อให้สามารถจัดการกับข้อมูลข้อความขนาดเล็กได้ง่ายขึ้น

2. Filtering Stop Words:

- ใช้ในการกรองคำที่ไม่สำคัญ (stop words) ออกในระหว่างการประมวลผลข้อความ เช่น คำว่า "in," "is," "an" เป็นต้น

3. Stemming:

- ใช้ในการลดรูปคำให้เหลือเพียงรากศัพท์ (root word) ซึ่งช่วยให้คอมพิวเตอร์เข้าใจความหมายของคำได้ดีขึ้น

4. Tagging Parts of Speech (POS):

- ใช้ในการระบุส่วนประกอบของคำในประโยคตามหลักไวยากรณ์ เช่น กริยา, นาม

5. Name Entity Recognition (NER):

- ใช้ในการค้นหาหน่วยข้อมูลที่มีความหมายเฉพาะในข้อความ เช่น ชื่อบุคคล สถานที่ หรือองค์กร และระบุประเภทของหน่วยข้อมูลนั้น

Example of Text tokenization & multiword

1. Basic filtering:

- ข้อความเริ่มต้น: "It's not 'cool' that ping-pong is not included in Rio 2016."
- กรองขั้นพื้นฐาน: ลบเครื่องหมายที่ไม่จำเป็นและเปลี่ยนข้อความเป็น "it not cool that ping pong is not included in rio 2016"

2. Tokenization:

- แบ่งข้อความออกเป็นคำ (tokens): "it," "not," "cool," "that," "ping," "pong," "is," "not," "included," "in," "rio," "2016"

3. Stopwords filtering:

- ลบคำที่ไม่จำเป็นหรือคำที่ไม่สำคัญออก เช่น "it," "not," "that," "is," "not," "in"
- คำที่เหลือหลังการกรอง: "cool," "ping," "pong," "included," "rio," "2016"

4. Multiwords grouping:

- จัดกลุ่มคำที่ควรอยู่ด้วยกัน เช่น "ping pong" แทนที่จะเป็น "ping" และ "pong" แยกกัน

ผลลัพธ์สุดท้าย:

- "cool ping pong included rio 2016"

กระบวนการนี้ช่วยในการจัดโครงสร้างข้อมูลข้อความเพื่อการวิเคราะห์ต่อไปได้อย่างมีประสิทธิภาพ โดยลบคำที่ไม่จำเป็นและเน้นคำสำคัญที่สามารถใช้ในการวิเคราะห์ได้

Example of POS Tagging

POS Tagging (Part-of-Speech Tagging) ซึ่งเป็นกระบวนการในการระบุและจัดหมวดหมู่คำในประโยคตามหน้าที่ทางไวยากรณ์ของคำแต่ละคำ เช่น:

- "John" ถูกแท็กเป็น **Noun** (คำนาม)
- "likes" ถูกแท็กเป็น **Verb** (คำกริยา)
- "the" ถูกแท็กเป็น **Determiner** (คำบ่งชี้)
- "blue" ถูกแท็กเป็น **Adjective** (คำคุณศัพท์)
- "house" ถูกแท็กเป็น **Noun** (คำนาม)
- "at" ถูกแท็กเป็น **Preposition** (คำบุพบท)
- "the" ถูกแท็กเป็น **Determiner** (คำบ่งชี้)
- "end" ถูกแท็กเป็น **Noun** (คำนาม)
- "of" ถูกแท็กเป็น **Preposition** (คำบุพบท)
- "the" ถูกแท็กเป็น **Determiner** (คำบ่งชี้)
- "street" ถูกแท็กเป็น **Noun** (คำนาม)

การใช้ POS Tagging ช่วยในการทำความเข้าใจโครงสร้างของประโยคและความสัมพันธ์ระหว่างคำในประโยค ซึ่งเป็นขั้นตอนสำคัญในกระบวนการประมวลผลภาษาธรรมชาติ (NLP)

Example of Named Entity Recognition

Named Entity Recognition (NER) ซึ่งเป็นกระบวนการในการระบุและจัดประเภทหน่วยข้อมูลที่มีความหมายเฉพาะในข้อความ ตัวอย่างของหน่วยข้อมูลที่ NER สามารถตรวจจับและจัดประเภทได้มีดังนี้:

- **Person (บุคคล):** Michael Jackson, Oprah Winfrey, Barack Obama, Susan Sarandon
- **Location (สถานที่):** Canada, Honolulu, Bangkok, Brazil, Cambridge
- **Organization (องค์กร):** Samsung, Disney, Yale University, Google
- **Time (เวลา):** 15:35, 12 PM
- **Quantity (ปริมาณ):** 4.7-inch, second
- **Communication (ผลิตภัณฑ์หรือแบรนด์):** iPhone, iPhone SE

ในตัวอย่างนี้ NER ได้จัดประเภทคำว่า "Apple" เป็นองค์กร (ORG), "today" เป็นวันที่ (DATE), และ "4.7-inch" เป็นจำนวนหรือปริมาณ (QUANTITY) เป็นต้น กระบวนการนี้ช่วยในการทำความเข้าใจและดึงข้อมูลสำคัญจากข้อความอย่างมีประสิทธิภาพ

Chapter 2 Regular Expressions and Word Tokenization

What are regular expressions?

Regular Expressions คืออะไร?

- **Regular expressions (Regex)** คือสตริงที่มีไวยากรณ์พิเศษ
- **Regex** ช่วยให้เราสามารถจับคู่แพทเทิร์นในสตริงอื่น ๆ ได้

การประยุกต์ใช้ Regular Expressions

- ค้นหาสิ่งทั้งหมดในเอกสาร
- แยกที่อยู่อีเมล
- ลบหรือแทนที่อักขระที่ไม่ต้องการ

ตัวอย่างการใช้งาน Regular Expressions ใน Python

```
import re
re.match('abc', 'abcdef')
```

ผลลัพธ์จะจับคู่ abc กับ abcdef ได้ (<re.Match object; span=(0, 3), match='abc'>)

```
word_regex = 'w+' -> เป็นการสร้างแพทเทิร์นสำหรับคำ
re.match(word_regex, 'hi there!')
```

จับคู่ hi ใน hi there! ได้ (<re.Match object; span=(0, 2), match='hi'>)

Common Regex Patterns

- **Pattern** คือชุดของตัวอักษรหรือสัญลักษณ์ที่สามารถจับคู่กับข้อความ คำ หรือเครื่องหมายวรรคตอนที่แท้จริงได้

ตัวอย่างแพทเทิร์นทั่วไป

Pattern	Matches	Example
\w+	Word	'Magic'
\d	Digit	9
\s	Space	' ' (Whitespace)
\S	Not space	'no_spaces'
[a-z]	Lowercase group	'abc'

รายละเอียดเพิ่มเติม

- Wildcard (.) จะจับคู่กับตัวอักษรหรือสัญลักษณ์ใด ๆ ก็ได้
- อักขระ + และ * จะทำให้แพทเทิร์นจับคู่ซ้ำ ๆ ของตัวอักษรเดียวหรือทั้งแพทเทิร์น

Pattern	Matches	Example
.	ตัวอักษรใด ๆ ยกเว้นบรรทัดใหม่	a.b จับคู่ aab, alb
^	จุดเริ่มต้นของสตริง	^Hello จับคู่ Hello World
\$	จุดสิ้นสุดของสตริง	World\$ จับคู่ Hello World
*	จำนวนซ้ำใด ๆ ของแพทเทิร์นก่อนหน้านี้ (รวมถึง 0 ครั้ง)	ab*c จับคู่ ac, abc

Pattern	Matches	Example
+	จำนวนซ้ำใด ๆ ของแพทเทิร์นก่อนหน้า (อย่างน้อย 1 ครั้ง)	ab+c จับคู่ abc, abbc
?	จำนวนซ้ำ 0 หรือ 1 ครั้งของแพทเทิร์นก่อนหน้า	colou?r จับคู่ color, colour
{n}	จำนวนซ้ำของแพทเทิร์นก่อนหน้า n ครั้ง	a{3} จับคู่ aaa
{n, }	จำนวนซ้ำของแพทเทิร์นก่อนหน้า n ครั้งขึ้นไป	a{2, } จับคู่ aa, aaa
{n, m}	จำนวนซ้ำระหว่าง n ถึง m ครั้งของแพทเทิร์นก่อนหน้า	a{2, 4} จับคู่ aa, aaa, aaaa
[]	ตัวอักษรใด ๆ ภายในวงเล็บเหลี่ยม	[aeiou] จับคู่ a, e, i, o, u
[^]	ตัวอักษรใด ๆ ที่ไม่อยู่ในวงเล็บเหลี่ยม	[^aeiou] จับคู่ b, c, d
\b	ขอบเขตของคำ	\bword\b จับคู่กับคำ word
\B	ไม่ใช่ขอบเขตของคำ	\Bword\B จับคู่ swording
\d	ตัวเลข (Digit)	\d+ จับคู่ 123, 4567
\D	ตัวอักษรที่ไม่ใช่ตัวเลข	\D+ จับคู่ abc, Hello
\w	ตัวอักษรที่เป็นคำ (Word Character: [a-zA-Z0-9_])	\w+ จับคู่ word, hello_123
\W	ตัวอักษรที่ไม่ใช่คำ (Non-Word Character)	\W+ จับคู่ !@#, (ช่องว่าง)
\s	ช่องว่าง (Whitespace)	\s+ จับคู่กับช่องว่างหรือ tab
\S	ตัวอักษรที่ไม่ใช่ช่องว่าง (Non-Whitespace)	\S+ จับคู่กับ Hello, 1234
\A	จุดเริ่มต้นของสตริง	\AHello จับคู่กับ Hello World
\Z	จุดสิ้นสุดของสตริง	World\Z จับคู่กับ Hello World
`	`	ใช้สำหรับการจับคู่ OR
()	ใช้สำหรับการจับกลุ่ม (Group) และจับคู่	(abc)+ จับคู่ abc, abcabc
(?i)	เปิดการจับคู่ที่ไม่สนใจตัวพิมพ์เล็ก-ใหญ่	(?i)hello จับคู่ hello, Hello, HELLO
(?<=...)	Positive lookbehind assertion	(?<=@)\w+ จับคู่กับคำหลังจากเครื่องหมาย @ เช่น @example จับคู่ example
(?<!...)	Negative lookbehind assertion	(?<!@)\w+ จับคู่กับคำที่ไม่ตามหลังเครื่องหมาย @
(?#...)	Comment (ใช้ใส่คอมเมนต์ในแพทเทิร์น)	a(?#this is a comment)b จับคู่กับ ab
(.*)	Greedy match (จับคู่ทุกอย่างจนกว่าจะเจอสิ่งที่ไม่ตรง)	a.*b จับคู่ acb, a123b, azyzb
(.+?)	Lazy match (จับคู่เฉพาะที่จำเป็น)	<.+?> จับคู่กับ <tag>, <a>, ไม่จับคู่ <tag><a>
\Q...\E	ลบความหมายพิเศษของตัวอักษรทั้งหมดในแพทเทิร์นระหว่าง \Q และ \E	\Q\$1.00\E จับคู่กับ \$1.00 โดยไม่ต้องเอาความหมายพิเศษของ \$ หรือ . มาใช้
\n	Line feed (newline)	จับคู่กับบรรทัดใหม่
\r	Carriage return	จับคู่กับ carriage return ในข้อความ

re module

split: แบ่งสตริงตามแพทเทิร์นของ regex

- ตัวอย่าง: `re.split('\s+', 'Split on spaces.')` จะแบ่งข้อความตามช่องว่างเป็น `['Split', 'on', 'spaces.']`

findall: ค้นหาและดึงแพทเทิร์นทั้งหมดในสตริง

- ตัวอย่าง: `print(re.findall(word_regex2, 'Split on spaces!'))` จะแสดงผลเป็น `['spaces']`

search: ค้นหาแพทเทิร์นในสตริงและส่งคืนการจับคู่แรกที่พบ

- ตัวอย่าง: `print(re.search(word_regex2, 'Split on spaces!'))` จะพบ `spaces` และแสดงตำแหน่งในสตริง

match: จับคู่ทั้งสตริงหรือสตริงย่อยที่เริ่มต้นตามแพทเทิร์นที่กำหนด

- ตัวอย่าง: `print(re.match(word_regex2, 'Split on spaces!'))` จะส่งคืน `None` เนื่องจากไม่พบการจับคู่ที่จุดเริ่มต้นของสตริง

Regular Expression สำหรับอีเมล

`/[w._%+-]+@[a-zA-Z-]+\.[a-zA-Z]{2,4}/`

- `[w._%+-]+`: จับคู่ส่วนแรกของอีเมลก่อน `@` โดยรับตัวอักษร, ตัวเลข, และสัญลักษณ์พิเศษ
- `@`: จับคู่กับสัญลักษณ์ `@`
- `[a-zA-Z-]+`: จับคู่ชื่อโดเมนที่ประกอบด้วยตัวอักษรและสัญลักษณ์ `.` หรือ `-`
- `\.`: จับคู่กับจุด `.` ที่คั่นระหว่างชื่อโดเมนกับนามสกุลโดเมน
- `[a-zA-Z]{2,4}`: จับคู่นามสกุลโดเมนที่ประกอบด้วยตัวอักษร 2 ถึง 4 ตัว เช่น `.com`, `.org`

ในภาพที่คุณอัปโหลด มีการถามคำถามเกี่ยวกับการเลือก **Regex pattern** ที่จะให้ผลลัพธ์ตามที่ระบุ โดยข้อมูลในภาพอธิบายดังนี้:

Which pattern?

- คำถาม: แพทเทิร์น Regex ได้จากตัวเลือกที่กำหนดที่จะให้ผลลัพธ์เป็นข้อความที่แบ่งเป็นคำตามที่แสดงในตัวอย่าง?

```
my_string = "Let's write RegEx!"  
re.findall(PATTERN, my_string)
```

ผลลัพธ์ที่ต้องการ: ['Let', 's', 'write', 'RegEx'] → b) PATTERN = r"\w+"

re.split() and re.findall()

การใช้งานโมดูล re:

- โมดูล re เป็นโมดูลใน Python ที่นำเข้า (import) เพื่อใช้งานฟังก์ชันต่าง ๆ ที่เกี่ยวข้องกับ Regular Expressions

การตั้งค่า my_string:

- สตริงตัวอย่างที่ใช้ในการทดสอบ:

```
my_string = "Let's write RegEx! Won't that be fun? I sure think so. Can you find 4  
sentences? Or perhaps, all 19 words?"
```

หมายเหตุเกี่ยวกับการใช้ r ก่อนแพทเทิร์นใน regex:

- การใช้ r ก่อนแพทเทิร์น เช่น r"\n" เป็นสิ่งสำคัญ เพราะช่วยให้ Python ตีความแพทเทิร์นเป็นสตริงดิบ (raw string) โดยไม่แปลง \n เป็นบรรทัดใหม่ (newline)
- ถ้าไม่ใช้ r อาจเกิดปัญหาในการตีความแพทเทิร์นหรือเกิดปัญหาจาก escape sequences ในสตริงได้

การแบ่งสตริง my_string ตามจุดสิ้นสุดของประโยค เพื่อทำสิ่งนี้:

- เขียนแพทเทิร์นชื่อ sentence_endings เพื่อจับคู่จุดสิ้นสุดของประโยค (เช่น . ? !)
- ใช้ฟังก์ชัน re.split() เพื่อแบ่งสตริง my_string ตามแพทเทิร์นที่กำหนด แล้วพิมพ์ผลลัพธ์ออกมา

re.split() การแบ่งสตริงตามจุดสิ้นสุดของประโยค:

ขั้นตอนที่ 1: กำหนดแพทเทิร์นสำหรับจับจุดสิ้นสุดของประโยค

```
sentence_endings = r"[.?!]"
```

- แพทเทิร์นนี้จะจับเครื่องหมายที่ใช้ในการสิ้นสุดประโยค เช่น จุด (.), เครื่องหมายคำถาม (?), และเครื่องหมายอัศเจรีย์ (!)

ขั้นตอนที่ 2: ใช้ฟังก์ชัน `re.split()` เพื่อแบ่งสตริงตามแพทเทิร์นที่กำหนด

```
print(re.split(sentence_endings, my_string))
```

ผลลัพธ์จะเป็นการแบ่งสตริงออกเป็นหลายส่วนตามจุดสิ้นสุดของประโยค:

```
["Let's write RegEx", " Won't that be fun", " I sure think so", " Can you find 4 sentences", " Or perhaps, all 19 words", ""]
```

`re.findall()`

- ค้นหาและพิมพ์คำที่ขึ้นต้นด้วยตัวอักษรพิมพ์ใหญ่ใน `my_string` โดยการเขียนแพทเทิร์นชื่อ `capitalized_words` และใช้ฟังก์ชัน `re.findall()`
- จำแพทเทิร์น `[a-z]` ที่ใช้จับกลุ่มตัวอักษรพิมพ์เล็กได้ใช่ไหม? ให้แก้ไขแพทเทิร์นนั้นให้เหมาะสมเพื่อจับกลุ่มตัวอักษรพิมพ์ใหญ่แทน

ขั้นตอนที่ 1: เขียนแพทเทิร์นชื่อ `capitalized_words` เพื่อจับคู่คำที่ขึ้นต้นด้วยตัวอักษรพิมพ์ใหญ่

```
capitalized_words = r"[A-Z]\w+"
```

- แพทเทิร์นนี้จะจับคู่กับคำที่เริ่มต้นด้วยตัวอักษรพิมพ์ใหญ่ (A-Z) และตามด้วยตัวอักษรอื่น ๆ

ขั้นตอนที่ 2: ใช้ฟังก์ชัน `re.findall()` เพื่อค้นหาคำที่ตรงกับแพทเทิร์นในสตริง `my_string` แล้วพิมพ์ผลลัพธ์ออกมา

```
print(re.findall(capitalized_words, my_string))
```

ผลลัพธ์จะเป็นคำที่ขึ้นต้นด้วยตัวอักษรพิมพ์ใหญ่ เช่น `['Let', 'RegEx', 'Won', 'Can', 'Or']`

เขียนแพทเทิร์นชื่อ `spaces` เพื่อจับคู่กับช่องว่างหนึ่งช่องหรือมากกว่านั้น (`"\s+"`) จากนั้นใช้ฟังก์ชัน `re.split()` เพื่อแบ่งสตริง `my_string` ตามแพทเทิร์นนี้ โดยคงเครื่องหมายวรรคตอนทั้งหมดไว้ และพิมพ์ผลลัพธ์ออกมา

```
spaces = r"\s+"
```

- แพทเทิร์น \s+ นี้จะจับคู่กับช่องว่าง (space) หนึ่งช่องหรือมากกว่านั้น

```
print(re.split(spaces, my_string))
```

ผลลัพธ์จะเป็นการแบ่งสตริงออกเป็นคำตามช่องว่าง พร้อมกับคงเครื่องหมายวรรคตอนทั้งหมดไว้:

```
["Let's", 'write', 'RegEx!', "Won't", 'that', 'be', 'fun?', 'I', 'sure', 'think', 'so.', 'Can', 'you', 'find', '4',  
'sentences?', 'Or', 'perhaps,', 'all', '19', 'words?']
```

ค้นหาตัวเลขทั้งหมดในสตริง `my_string`:

ขั้นตอนที่ 1: เขียนแพทเทิร์นชื่อ `digits` เพื่อจับคู่กับตัวเลข

```
digits = r"\d+"
```

- แพทเทิร์น \d+ นี้จะจับคู่กับตัวเลขอย่างน้อยหนึ่งตัว

ขั้นตอนที่ 2: ใช้ฟังก์ชัน `re.findall()` เพื่อค้นหาตัวเลขทั้งหมดที่ตรงกับแพทเทิร์นในสตริง `my_string` และพิมพ์ผลลัพธ์ออกมา

```
print(re.findall(digits, my_string))
```

ผลลัพธ์จะเป็นรายการของตัวเลขที่พบในสตริง `['4', '19']`

What is tokenization?

Tokenization คืออะไร?

- Tokenization คือการแปลงสตริงหรือเอกสารให้เป็น tokens หรือส่วนเล็ก ๆ
- เป็นขั้นตอนหนึ่งของการ preprocessing ข้อความใน NLP ซึ่งมักจะเป็นขั้นตอนแรกในกระบวนการประมวลผล

- มีกฎและทฤษฎีหลายแบบที่ใช้ในการแบ่ง tokens ซึ่งทุกคนสามารถสร้างกฎของตนเองได้โดยใช้ Regular Expressions (Regex)

ตัวอย่างของ Tokenization:

- การแบ่งคำหรือประโยคออกเป็นส่วนย่อย ๆ
- การแยกเครื่องหมายวรรคตอน เช่น , . !
- การแยก hashtags ทั้งหมดในทวีต

nlTK library

ตัวอย่างการใช้เมธอด word_tokenize จาก nltk.tokenize เพื่อแบ่งสตริงเป็น tokens:

```
from nltk.tokenize import word_tokenize  
word_tokenize("Hi there!")
```

ผลลัพธ์ที่ได้ ['Hi', 'there', '!']

Why tokenize?

- ทำให้การระบุส่วนของคำพูดง่ายขึ้น (Easier to map part of speech)
- ช่วยในการจับคู่คำที่ใช้ทั่วไป (Matching common words)
- ลบ tokens ที่ไม่ต้องการออก (Removing unwanted tokens)

ตัวอย่าง:

- ประโยคก่อน Tokenization:

"I don't like Sam's shoes."

- ผลลัพธ์หลัง Tokenization:

['I', 'do', 'n't', 'like', 'Sam', 's', 'shoes', '.']

- คำและเครื่องหมายต่าง ๆ ถูกแยกออกเป็น tokens

Other nltk tokenizers

1. sent_tokenize:

- ใช้ในการแบ่งเอกสารออกเป็นประโยค (tokenize a document into sentences)
- เช่น การแบ่งประโยคในข้อความ "Hello, world!" ให้แยกออกเป็น "Hello," และ "world!"

2. regexp_tokenize:

- ใช้ในการแบ่งสตริงหรือเอกสารตามแพทเทิร์นที่กำหนดไว้ล่วงหน้าด้วย Regular Expressions (tokenize a string or document based on a regular expression pattern)
- แพทเทิร์นเหล่านี้ถูกกำหนดตามที่คุณต้องการ เช่น การจับคู่คำที่ตรงกับรูปแบบเฉพาะ

3. TweetTokenizer:

- เป็นคลาสพิเศษที่ใช้เฉพาะกับการทำ Tokenization สำหรับทวีต (special class just for tweet tokenization)
- อนุญาตให้คุณแยก hashtags, mentions และเครื่องหมายอัศเจรีย์จำนวนมากออกจากกัน

Word tokenization with NLTK 1

- การใช้งาน word_tokenize และ sent_tokenize:
 - ใช้ word_tokenize และ sent_tokenize จากโมดูล nltk.tokenize เพื่อทำการ Tokenization ทั้งคำและประโยคจากสตริงใน Python
- การนำเข้า (import) ฟังก์ชัน sent_tokenize และ word_tokenize จาก nltk.tokenize:

```
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
```

การอ่านไฟล์ข้อความ (TXT file) และเก็บไว้ในตัวแปร scene_one:

- ตัวอย่างการอ่านไฟล์

```
f = open("scene_one.txt", "r")
scene_one = f.read()
```

Word tokenization with NLTK 2

ทำการ Tokenize ทุกประโยคใน scene_one โดยใช้ฟังก์ชัน sent_tokenize():

- ตัวอย่างโค้ดในการแบ่ง scene_one ออกเป็นประโยค

```
sentences = sent_tokenize(scene_one)
```

ทำการ Tokenize ประโยคที่สี่ใน sentences (ซึ่งสามารถเข้าถึงได้โดย sentences[3]) โดยใช้ฟังก์ชัน word_tokenize():

- ตัวอย่างโค้ดในการแบ่งคำในประโยคที่สี่

```
tokenized_sent = word_tokenize(sentences[3])
```

Word tokenization with NLTK 3

1. ค้นหาที่ไม่ซ้ำกันในทั้งฉาก (scene):

- ทำการ **tokenize** คำทั้งหมดใน scene_one โดยใช้ฟังก์ชัน word_tokenize() และแปลงผลลัพธ์เป็นเซต (set) เพื่อหาคำที่ไม่ซ้ำกัน

- ตัวอย่างโค้ด:

```
unique_tokens = set(word_tokenize(scene_one))
```

2. พิมพ์ผลลัพธ์คำที่ไม่ซ้ำกันที่พบ:

- ใช้คำสั่ง print เพื่อแสดงผลคำที่ไม่ซ้ำกัน

```
print(unique_tokens)
```

More regex with re.search() 1

การใช้ re.search() และ re.match() เพื่อค้นหาโทเคนเฉพาะ:

- ใช้ re.search() เพื่อค้นหาการปรากฏครั้งแรกของคำว่า "coconuts" ใน scene_one และเก็บผลลัพธ์ในตัวแปร match

```
match = re.search("coconuts", scene_one)
```

- พิมพ์ตำแหน่งเริ่มต้นและสิ้นสุดของ match โดยใช้ .start() และ .end()

```
print(match.start(), match.end())
```

More regex with re.search() 2

การเขียน Regular Expression เพื่อค้นหาอะไรก็ได้ในวงเล็บเหลี่ยม:

- เขียน Regular Expression ที่ชื่อว่า pattern1 เพื่อค้นหาข้อความใด ๆ ที่อยู่ในวงเล็บเหลี่ยม

```
pattern1 = r"\[.*_\]"
```

- ใช้ re.search() กับ pattern1 เพื่อค้นหาข้อความแรกที่อยู่ในวงเล็บเหลี่ยมใน scene_one และพิมพ์ผลลัพธ์

```
print(re.search(pattern1, scene_one))
```

More regex with re.search() 3

สร้างแพทเทิร์นเพื่อจับคู่กับสัญลักษณ์ในสคริปต์ (script notation):

- ตัวอย่างเช่น การจับคู่กับคำว่า "Character:"
- แพทเทิร์นจะถูกเก็บไว้ในตัวแปร pattern2
- แพทเทิร์นนี้จะจับคู่คำหรือช่องว่างที่นำหน้าด้วยเครื่องหมาย : โดยสามารถรวมถึงช่องว่างที่อาจอยู่ระหว่างคำและ :

```
pattern2 = r"[\w\s]+:"
```

ใช้ `re.match()` กับแพทเทิร์นใหม่เพื่อค้นหาและพิมพ์สัญลักษณ์ในสคริปต์ที่อยู่ในบรรทัดที่สี่:

- ข้อความที่ถูกแบ่งเป็นประโยคสามารถเข้าถึงได้ผ่าน `sentences` และประโยคที่สี่คือ `sentences[3]`
- ใช้ `re.match()` เพื่อตรวจสอบและพิมพ์ผลลัพธ์

```
print(re.match(pattern2, sentences[3]))
```

Regex groups using or "|"

Regular Expression (Regex) ในภาษา Python โดยเฉพาะการใช้ตัวดำเนินการ | เพื่อสร้างกลุ่มตัวเลือก (OR) และการจับกลุ่มด้วยวงเล็บ ()

1. `import re`: บรรทัดนี้ใช้เพื่อนำเข้าโมดูล `re` ซึ่งเป็นโมดูลมาตรฐานของ Python ที่ใช้สำหรับการทำงานกับ Regular Expression
2. `match_digits_and_words = '(\d+|\W+)'`:
 - `(\d+|\W+)`: เป็นรูปแบบ Regex ที่ถูกกำหนดขึ้น
 - `()` คือการสร้างกลุ่ม
 - `\d+`: หมายถึง ตัวเลขหนึ่งตัวขึ้นไป
 - `|`: หมายถึง หรือ (OR)
 - `\W+`: หมายถึง ตัวอักษรที่ไม่ใช่ตัวเลขหรืออักขระตัวใหญ่ตัวเล็กหนึ่งตัวขึ้นไป
 - ดังนั้น รูปแบบนี้จะจับคู่กับทั้งกลุ่มของตัวเลขหรือกลุ่มของตัวอักษรที่ไม่ใช่ตัวเลขหรืออักขระตัวใหญ่ตัวเล็ก
3. `re.findall(match_digits_and_words, 'He has 11 cats')`:
 - `re.findall()`: เป็นฟังก์ชันที่ใช้ค้นหาทุกส่วนที่ตรงกับรูปแบบ Regex ที่กำหนดไว้ในสตริงที่ให้มา
 - `match_digits_and_words`: คือรูปแบบ Regex ที่เราได้กำหนดไว้

- 'He has 11 cats': คือสตริงที่เราต้องการค้นหา
- ผลลัพธ์ที่ได้คือ ['He', 'has', '11', 'cats'] ซึ่งเป็น list ของทุกส่วนที่ตรงกับรูปแบบ Regex ที่พบในสตริง

ตัวอย่างและการทำงาน

ในตัวอย่างนี้ เราต้องการค้นหาทั้งคำและตัวเลขในประโยค "He has 11 cats" รูปแบบ Regex ที่เราสร้างขึ้นจะจับคู่กับ:

- "He" เพราะเป็นกลุ่มของตัวอักษรที่ไม่ใช่ตัวเลขหรืออักษรตัวใหญ่ตัวเล็ก
- "has" เพราะเป็นกลุ่มของตัวอักษรที่ไม่ใช่ตัวเลขหรืออักษรตัวใหญ่ตัวเล็ก
- "11" เพราะเป็นกลุ่มของตัวเลข
- "cats" เพราะเป็นกลุ่มของตัวอักษรที่ไม่ใช่ตัวเลขหรืออักษรตัวใหญ่ตัวเล็ก

Regex ranges and groups

Regular Expression (Regex) เพื่อกำหนดรูปแบบของข้อความ โดยใช้คอนเซ็ปต์ของ ช่วง (range) และ กลุ่ม (group)

ช่วง (Range)

- คืออะไร: การระบุชุดของอักขระที่ต้องการจับคู่
- รูปแบบ: [a-z] หมายถึง อักขระใดๆ ตั้งแต่ a ถึง z
- ตัวอย่าง:
 - [A-Za-z]+: จับคู่กับอักขระภาษาอังกฤษตัวใหญ่หรือตัวเล็กหนึ่งตัวขึ้นไป (เช่น ABCDEFghijk)
 - [0-9]: จับคู่กับตัวเลข 0 ถึง 9 (เช่น 9)
 - [A-Za-z\-\._]+: จับคู่กับอักขระภาษาอังกฤษตัวใหญ่หรือตัวเล็ก, ขีดกลาง (-) และจุด (.) หนึ่งตัวขึ้นไป (เช่น My-Website.com)

กลุ่ม (Group)

- คืออะไร: การรวมตัวเลือกต่างๆ เข้าด้วยกัน
- รูปแบบ: ใช้วงเล็บ () ครอบกลุ่มตัวเลือก
- ตัวอย่าง:
 - (a-z): จับคู่กับ a, - หรือ z (เช่น a-z)
 - (\s+|\.): จับคู่กับช่องว่างหนึ่งช่องขึ้นไป หรือ เครื่องหมายจุลภาค (เช่น , หรือ " ")

สรุป

- Regex เป็นเครื่องมือที่ทรงพลังสำหรับการค้นหาและจัดการข้อความโดยใช้รูปแบบ
- ช่วง ช่วยให้เราระบุชุดของอักขระที่ต้องการจับคู่ได้อย่างแม่นยำ
- กลุ่ม ช่วยให้เราสร้างรูปแบบที่ซับซ้อนขึ้น โดยการรวมตัวเลือกต่างๆ เข้าด้วยกัน

ตัวอย่างการใช้งานจริง

- ตรวจสอบรูปแบบอีเมล: `^[a-zA-Z0-9_+]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-]+$`
- สกัดเบอร์โทรศัพท์: `\d{10}` (สำหรับเบอร์โทรศัพท์ 10 หลัก)
- ค้นหา URL: `https?:/(?:www.?(^/)+)\.[a-z]{2,}`

ประโยชน์ของ Regex

- การตรวจสอบข้อมูล: เช่น ตรวจสอบรูปแบบของอีเมล, เบอร์โทรศัพท์, รหัสผ่าน
 - การสกัดข้อมูล: เช่น สกัดข้อมูลที่ต้องการจากข้อความขนาดใหญ่
 - การแทนที่ข้อความ: เช่น แทนที่ข้อความบางส่วนด้วยรูปแบบอื่น
-

Character range with `re.match()`

```
import re
my_str = 'match lowercase spaces nums like 12, but no commas'
match_str = ('[a-z0-9,]+') ————> ถ้าเพิ่ม , ก็เพิ่มไปจนถึงตัว ,
print(re.match(match_str, my_str))
```

```
<re.Match object; span=(0, 35), match='match lowercase spaces nums like 12'>
```

โค้ด Python นี้ใช้ไลบรารี re ซึ่งเป็นไลบรารีสำหรับการทำงานกับ Regular Expression เพื่อค้นหารูปแบบของข้อความในสตริงที่กำหนด

อธิบายโค้ดที่ละบรรทัด

1. import re: นำเข้าไลบรารี re สำหรับการทำงานกับ Regular Expression
2. my_str = 'match lowercase spaces nums like 12, but no commas': กำหนดสตริงที่ต้องการค้นหา
3. match_str = '[a-z0-9,]+': กำหนดรูปแบบ (pattern) ที่ต้องการค้นหา
 - o [a-z0-9,]+: หมายถึง การจับคู่กับอักขระใดๆ ที่เป็นตัวอักษรภาษาอังกฤษตัวเล็ก (a-z) หรือตัวเลข (0-9) อย่างน้อยหนึ่งตัวขึ้นไป
4. print(re.match(match_str, my_str)): พิมพ์ผลลัพธ์ของการค้นหา
 - o re.match(): ฟังก์ชันสำหรับค้นหารูปแบบในสตริง
 - o match_str: รูปแบบที่ต้องการค้นหา
 - o my_str: สตริงที่ต้องการค้นหา
 - o ผลลัพธ์: <re.Match object; span=(0, 35), match='match lowercase spaces nums like 12'> หมายถึง พบรูปแบบที่ตรงกันตั้งแต่ตำแหน่งที่ 0 ถึง 35 (นับรวมทั้งสิ้น 36 ตัวอักษร) และรูปแบบที่ตรงกันคือ 'match lowercase spaces nums like 12'

การทำงานของโค้ด

โค้ดนี้จะพยายามค้นหาคำที่ประกอบด้วยตัวอักษรภาษาอังกฤษตัวเล็กและตัวเลขเท่านั้นในสตริง my_str โดยเริ่มต้นจากตำแหน่งแรกของสตริง เนื่องจาก [a-z0-9]+ จะจับคู่กับลำดับของตัวอักษรและตัวเลขที่ยาวที่สุดเท่าที่จะเป็นไปได้ ดังนั้นจึงจับคู่ได้ตั้งแต่ต้นจนถึงคำว่า 12

Choosing a tokenizer

- เป้าหมาย: เลือก Pattern ของ Regular Expression ที่เหมาะสมที่สุดเพื่อแบ่งประโยคเป็น Token โดยรักษาเครื่องหมายวรรคตอนแต่ละตัวเป็น Token แยก และรักษา "#1" ไว้เป็น Token เดียว
- ตัวเลือก: มี Pattern ให้เลือก 4 ตัวเลือก
- เครื่องมือ: ใช้ฟังก์ชัน regexp_tokenize จาก NLTK เพื่อทดลอง Pattern ต่างๆ

วิเคราะห์ Pattern แต่ละตัวเลือก

- a) r"(\w+|?|!)":
 - จับคู่กับ: ตัวอักษรหนึ่งตัวขึ้นไป หรือเครื่องหมายคำถาม หรือเครื่องหมายอัศเจรีย์
 - ข้อเสีย: จะแบ่ง "#1" ออกเป็น "#", "1"
- b) r"(\w+|#\d|?|!)":
 - จับคู่กับ: ตัวอักษรหนึ่งตัวขึ้นไป หรือรูปแบบ "#ตามด้วยตัวเลข" หรือเครื่องหมายคำถาม หรือเครื่องหมายอัศเจรีย์
 - เหมาะสมที่สุด: Pattern นี้สามารถรักษา "#1" ไว้เป็น Token เดียว และยังจับคู่กับตัวอักษรและเครื่องหมายวรรคตอนได้ครบถ้วน
- c) r"(\#\d\w+?!)":
 - จับคู่กับ: รูปแบบ "#ตามด้วยตัวเลข ตามด้วยตัวอักษรหนึ่งตัวขึ้นไป ตามด้วยเครื่องหมายคำถาม และเครื่องหมายอัศเจรีย์"
 - ไม่เหมาะสม: เจาะจงเกินไป ไม่ครอบคลุมรูปแบบอื่นๆ
- d) r"\s+":

- จับคู่กับ: ช่องว่างหนึ่งช่องขึ้นไป
- ไม่เหมาะสม: จะแบ่งประโยคออกเป็นคำแต่ละคำ แต่ไม่รักษาเครื่องหมายวรรคตอน

สรุป

Pattern ที่เหมาะสมที่สุดคือ b) `r"(\w+|#\d|?|!)"`

การทดลองด้วย NLTK

- `from nltk.tokenize import regexp_tokenize`
- `my_string = "SOLDIER #1: Found them? In Mercea? The coconut's tropical!"`
- `pattern2 = r"(\w+|#\d|?|!)"`
- `tokens = regexp_tokenize(my_string, pattern2)`
- `print(tokens)`

`['SOLDIER', '#1', 'Found', 'them', '?', 'In', 'Mercea', '?', 'The', 'coconut', 's', 'tropical', '!']`

Regex with NLTK tokenization1

Twitter เป็นแหล่งข้อมูลที่ใช้บ่อยสำหรับการประมวลผลภาษาธรรมชาติ (NLP) และงานต่างๆ ในการออกกำลังกายนี้ คุณจะสร้างตัวแบ่งคำที่ซับซ้อนขึ้นสำหรับทวีตที่มีแฮชแท็กและการกล่าวถึง (mentions) โดยใช้ `nltk` และ `regex` คลาส `nltk.tokenize.TweetTokenizer` ให้วิธีการและคุณลักษณะเพิ่มเติมสำหรับการแยกทวีต

จาก `nltk.tokenize` ให้นำเข้า `regexp_tokenize` และ `TweetTokenizer`

```
from nltk.tokenize import regexp_tokenize
from nltk.tokenize import TweetTokenizer
```

Regex with NLTK tokenization2

Regex กับการแบ่งคำด้วย NLTK

- ตั้งค่า • รูปแบบ regex เพื่อกำหนดแฮชแท็กที่เรียกว่า pattern1 ได้ถูกกำหนดไว้ให้คุณแล้ว ให้เรียกใช้ `regex_tokenize()` ด้วยรูปแบบแฮชแท็กนี้บนทวีตแรกใน `tweets` และกำหนดผลลัพธ์เป็น `hashtags`
- พิมพ์ `hashtags`

```
from nltk.tokenize import regex_tokenize

tweets = ['This is the #nlp exercise! #python', '#NLP is super fun! <3 #learning', 'Thanks @fitmkmtnb :) #nlp #python']

# กำหนดรูปแบบ regex เพื่อค้นหาแฮชแท็ก: pattern1
pattern1 = r"#\w+"

# ใช้รูปแบบนี้กับทวีตแรกในรายการ tweets
hashtags = regex_tokenize(tweets[0], pattern1)
print(hashtags)
```

- `regex_tokenize(tweets[0], pattern1)` ใช้รูปแบบ regex `pattern1` บนทวีตแรกในรายการ `tweets` เพื่อดึงแฮชแท็ก
- `tweets[0]` คือทวีตแรกในรายการ
- `pattern1` คือรูปแบบ regex เพื่อค้นหาแฮชแท็ก

Regex with NLTK tokenization3

- เขียนรูปแบบใหม่ที่เรียกว่า `pattern2` เพื่อจับคู่การกล่าวถึง (mentions) และแฮชแท็ก รูปแบบการกล่าวถึงจะเป็นเช่น `@fitmkmtnb`
- จากนั้น, เรียกใช้ `regex_tokenize()` ด้วยรูปแบบแฮชแท็กใหม่ของคณบนทวีตสุดท้ายใน `tweets` และกำหนดผลลัพธ์เป็น `mentions_hashtags`
- คุณสามารถเข้าถึงองค์ประกอบสุดท้ายของรายการโดยใช้ `-1` เป็นดัชนี เช่น `tweets[-1]`
- พิมพ์ `mentions_hashtags`

```
# เขียนรูปแบบที่จับคู่้ทั้งการกล่าวถึง (@) และแฮชแท็ก
pattern2 = r"@#\w+"

# ใช้รูปแบบนี้กับทวิตสุดท้ายในรายการ tweets
mentions_hashtags = regexp_tokenize(tweets[-1], pattern2)
print(mentions_hashtags)
```

- pattern2 = r"@#\w+" คือรูปแบบ regex ที่จับคู่[้]ทั้งการกล่าวถึง (เริ่มต้นด้วย @) และแฮชแท็ก (เริ่มต้นด้วย #)
- regexp_tokenize(tweets[-1], pattern2) ใช้รูปแบบนี้กับทวิตสุดท้ายในรายการ tweets เพื่อดึงการกล่าวถึงและแฮชแท็ก
- tweets[-1] เข้าถึงทวิตสุดท้ายในรายการ

Regex with NLTK tokenization 4

- สร้างอินสแตนซ์ของ TweetTokenizer เรียกว่า tknzs และใช้มันในลิสต์คอมพรีเอนชัน (list comprehension) เพื่อแบ่งทวิตแต่ละอันเป็นรายการใหม่ที่เรียกว่า all_tokens
- เพื่อทำเช่นนี้ ให้ใช้วิธี .tokenize() ของ tknzs โดยให้ t เป็นตัวแปรการวนรอบของคุณ
- พิมพ์ all_tokens

```
# ใช้ TweetTokenizer เพื่อแบ่งทวิตทั้งหมดเป็นรายการเดียว
tknzs = TweetTokenizer()
all_tokens = [tknzs.tokenize(t) for t in tweets]
print(all_tokens)
```

- tknzs = TweetTokenizer() สร้างอินสแตนซ์ของ TweetTokenizer
 - tknzs.tokenize(t) ใช้เพื่อแบ่งทวิตแต่ละอันเป็นรายการของคำ
 - [tknzs.tokenize(t) for t in tweets] สร้างรายการใหม่ที่ประกอบด้วยรายการของคำที่แบ่งแล้วจากทวิตทั้งหมด
-

Non-ascii tokenization 1

การแบ่งคำที่ไม่ใช่ ASCII

• ฝึกการแบ่งคำขั้นสูงโดยการแบ่งข้อความที่ใช้ตัวอักษรที่ไม่ใช่ ASCII • โมดูลต่อไปนี้ได้ถูกนำเข้ามาลวงหน้าจาก nltk.tokenize:

- `regex_tokenize`
- `word_tokenize`

```
from nltk.tokenize import regex_tokenize
from nltk.tokenize import word_tokenize
```

• Set `german_text = "Wann gehen wir Pizza essen? 🍕 Und fährst du mit Über? 🚆"`

• ตั้งค่า • ช่วงยูนิโค้ดสำหรับอโมจิคือ:

- `('U0001F300'-'U0001F5FF')`
- `('U0001F600'-'U0001F64F')`
- `('U0001F680'-'U0001F6FF')`
- `('u2600'-'u26FF'-'u2700'-'u27BF')`

Non-ascii tokenization 2

แบ่งคำทั้งหมดใน `german_text` โดยใช้ `word_tokenize()` และพิมพ์ผลลัพธ์

```
# แบ่งคำและพิมพ์คำทั้งหมดใน german_text
all_words = word_tokenize(german_text)
print(all_words)
```

```
['Wann', 'gehen', 'wir', 'Pizza', 'essen', '?', '🍷', 'Und', 'fährst', 'du',  
'mit', 'Über', '?', '🚗']
```

Non-ascii tokenization 3

- แบ่งเฉพาะคำที่เป็นตัวพิมพ์ใหญ่ใน `german_text` • ก่อนอื่น เขียนรูปแบบที่เรียกว่า `capital_words` เพื่อจับคู่เฉพาะคำที่เป็นตัวพิมพ์ใหญ่ ตรวจสอบให้แน่ใจว่าได้รวมตัวอักษรเยอรมัน `Ü` ด้วย! ใช้ตัวอักษรนี้ในแบบฝึกหัดโดยการคัดลอกและวางจากคำแนะนำเหล่านี้ • จากนั้น, ใช้ `regex_tokenize()` เพื่อแบ่งคำตามรูปแบบนี้

```
# แบ่งคำและพิมพ์เฉพาะคำที่เป็นตัวพิมพ์ใหญ่  
capital_words = r'\b[A-ZÄÖÜ][A-ZÄÖÜa-zäöüß]*\b'  
print(regex_tokenize(german_text, capital_words))
```

ผลลัพธ์ที่พิมพ์ออกมาจะเป็น:

```
['Wann', 'Pizza', 'Und', 'Über']
```

Non-ascii tokenization 4

- แบ่งเฉพาะอีโมจิใน `german_text` รูปแบบที่ใช้ช่วงยูนิโค้ดสำหรับอีโมจิได้ถูกเขียนให้แล้วในข้อความการบ้านของคุณ หน้าที่ของคุณคือใช้ `regex_tokenize()` เพื่อแบ่งอีโมจิ

```
# แบ่งคำและพิมพ์เฉพาะอีโมจิ  
emoji_pattern = r'[\U0001F300-\U0001F5FF|\U0001F600-\U0001F64F|\U0001F680-  
\U0001F6FF|\u2600-\u26FF\u2700-\u27BF]'  
print(regex_tokenize(german_text, emoji_pattern))
```

- `emoji_pattern = r'[\U0001F300-\U0001F5FF|\U0001F600-\U0001F64F|\U0001F680-\U0001F6FF|\u2600-\u26FF\u2700-\u27BF]'`: รูปแบบ regex นี้ใช้ช่วงยูนิโค้ดที่ระบุเพื่อจับคู่กับอีโมจิ

- `regex_tokenize(german_text, emoji_pattern)`: ใช้รูปแบบนี้เพื่อแบ่งข้อความและจับคู่เฉพาะอีโมจิ
- `print()`: พิมพ์ผลลัพธ์ที่ได้

ผลลัพธ์ที่พิมพ์ออกมาจะเป็นรายการของอีโมจิที่พบใน `german_text`

Getting started with matplotlib

เริ่มต้นใช้งาน Matplotlib

- Matplotlib เป็นไลบรารีสำหรับการสร้างกราฟที่ใช้ในโครงการ Python แบบโอเพ่นซอร์สหลายโปรเจกต์
 - มีฟังก์ชันที่ตรงไปตรงมาแต่มีตัวเลือกมากมาย
 - ฮิสโตแกรม (Histograms): ใช้เพื่อแสดงการกระจายของชุดข้อมูล โดยการแสดงความถี่ของข้อมูลในช่วงที่กำหนด (bins)
 - แผนภูมิแท่ง (Bar Charts): เหมาะสำหรับการเปรียบเทียบปริมาณระหว่างหมวดหมู่ต่าง ๆ โดยสามารถใช้แท่งแนวนอนหรือแนวตั้งได้
 - แผนภูมิเส้น (Line Charts): ใช้เพื่อแสดงแนวโน้มตามเวลา หรือข้อมูลที่ต่อเนื่อง โดยการเชื่อมต่อจุดข้อมูลด้วยเส้น
 - แผนภูมิการกระจาย (Scatter Plots): แสดงความสัมพันธ์ระหว่างตัวแปรสองตัว แต่ละจุดแทนคู่ของค่า เพื่อให้สามารถระบุความสัมพันธ์หรือกลุ่มได้
 - นอกจากนี้ยังมีฟังก์ชันขั้นสูง เช่น:
 - กราฟ 3 มิติ (3D Graphs): Matplotlib รองรับการสร้างกราฟ 3 มิติผ่านโมดูล `mpl_toolkits.mplot3d` ซึ่งช่วยในการแสดงข้อมูลในสามมิติ
 - แอนิเมชัน (Animations): คุณสามารถสร้างการแสดงผลที่เคลื่อนไหวและแอนิเมชันเพื่อแสดงการเปลี่ยนแปลงของข้อมูลตามเวลาได้ด้วยโมดูลแอนิเมชันของ Matplotlib
-

Plotting a histogram with matplotlib

การสร้างฮิสโตแกรมด้วย Matplotlib

```
from matplotlib import pyplot as plt

# สร้างฮิสโตแกรมโดยใช้ข้อมูลที่ให้มา ฟังก์ชัน hist() ใช้ในการสร้างฮิสโตแกรมจากข้อมูลที่ให้มา ซึ่งในกรณีนี้
คือรายการ [1, 5, 5, 7, 7, 7, 9]

plt.hist([1, 5, 5, 7, 7, 7, 9])

# แสดงกราฟ ใช้เพื่อแสดงกราฟที่สร้างขึ้น
plt.show()
```

Combining NLP data extraction with plotting

```
from matplotlib import pyplot as plt
from nltk.tokenize import word_tokenize

# แยกข้อความเป็นคำ
words = word_tokenize("This is a pretty cool tool!")

# คำนวณความยาวของแต่ละคำ
word_lengths = [len(w) for w in words]

# สร้างฮิสโตแกรมของความยาวคำ
plt.hist(word_lengths)

# แสดงกราฟ
plt.show()
```

Charting practice 1

ขั้นตอนในการอ่านไฟล์และแยกบรรทัด:

1. อ่านไฟล์ txt (holy_grail.txt) และเก็บไว้ในตัวแปร holy_grail
2. แยกสคริปต์ holy_grail ออกเป็นบรรทัดโดยใช้ตัวคั่นบรรทัดใหม่ ('\n')

- # อ่านไฟล์ TXT
- f = open("holy_grail.txt", "r")
- holy_grail = f.read()
-
- # แยกสคริปต์ออกเป็นบรรทัด: lines
- lines = holy_grail.split('\n')

holy_grail.split('\n'): ใช้เมธอด split() เพื่อแยกเนื้อหาของตัวแปร holy_grail ออกเป็นบรรทัด ๆ โดยใช้ตัวคั่นบรรทัดใหม่ (\n)

Charting practice 2

- ใช้ re.sub() ภายใน list comprehension เพื่อแทนที่ข้อความ เช่น ARTHUR: และ SOLDIER #1
- ใช้ list comprehension เพื่อแยกบรรทัดด้วย regex_tokenize() โดยเก็บเฉพาะคำ

- import re
- from nltk.tokenize import regex_tokenize
-
- # แทนที่ข้อความของตัวละครในสคริปต์
- pattern = "[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:"
- lines = [re.sub(pattern, "", l) for l in lines]
-
- # แยกแต่ละบรรทัด: tokenized_lines
- tokenized_lines = [regex_tokenize(s, "\w+") for s in lines]

- `re.sub(pattern, "", l)`: ใช้ `re.sub()` เพื่อแทนที่ข้อความที่ตรงกับพาดเทิร์น (pattern) ด้วยสตริงว่าง ("") ในแต่ละบรรทัด `l`
- `regex_tokenize(s, "\w+")`: ใช้ `regex_tokenize()` เพื่อแยกบรรทัด `s` โดยเก็บเฉพาะคำที่ตรงกับพาดเทิร์น `"\w+"`

Charting practice 3

การฝึกสร้างกราฟ

ขั้นตอนในการสร้างกราฟ:

- ใช้ list comprehension เพื่อสร้างรายการของความยาวบรรทัดที่เรียกว่า `line_num_words`
- ใช้ `plt.hist()` เพื่อสร้างฮิสโตแกรมของ `line_num_words`
- ใช้ `plt.show()` เพื่อแสดงกราฟ

```
● from matplotlib import pyplot as plt
●
● # สร้างรายการของความยาวบรรทัด: line_num_words
● line_num_words = [len(t_line) for t_line in tokenized_lines]
●
● # สร้างฮิสโตแกรมของความยาวบรรทัด
● plt.hist(line_num_words)
●
● # แสดงกราฟ
● plt.show()
```

- `[len(t_line) for t_line in tokenized_lines]`: ใช้ list comprehension เพื่อคำนวณความยาวของแต่ละบรรทัดใน `tokenized_lines` โดยใช้ฟังก์ชัน `len()`
- `plt.hist(line_num_words)`: สร้างฮิสโตแกรมของความยาวบรรทัด

- plt.show(): แสดงกราฟอีสโตแกรม

บทที่ 3.1

หน้า 2: Outline

- บทนี้ประกอบไปด้วยเนื้อหาหลัก 2 ส่วน:
 1. การนับจำนวนคำด้วยวิธี Bag-of-Words
 2. การเตรียมข้อความเบื้องต้น (Simple text preprocessing)

หน้า 4: Bag-of-Words

- **Bag-of-Words** เป็นวิธีการเบื้องต้นในการค้นหาหัวข้อในข้อความ
- กระบวนการเริ่มต้นจากการสร้าง tokens โดยใช้การ tokenization แล้วนับจำนวนของแต่ละ token
- ยิ่งคำใดปรากฏบ่อยขึ้น คำนั้นอาจมีความสำคัญในข้อความมากขึ้น
- **Bag-of-Words** เป็นวิธีที่ดีในการระบุคำที่สำคัญในข้อความ

หน้า 5: Example of Bag-of-Words

- ตัวอย่างของข้อความ: "The cat is in the box. The cat likes the box. The box is over the cat."
- คำที่ถูก strip (ลบเครื่องหมายต่าง ๆ) และการนับคำหลังจากการเตรียมข้อความ

คำ	ความถี่
The	3
cat	3
box	3
is	1
in	1
likes	1
over	1

- หากมีการ preprocess เพิ่ม เช่น การทำ lowercase ทุกคำ คำที่ถูกนับจะไม่แยกออกเป็นคำเดียวกันอีก

หน้า 6: การใช้งาน Bag-of-Words ใน Python

ในหน้านี้จะสอนวิธีการสร้าง Bag-of-Words ด้วยการเขียนโค้ด Python โดยใช้ไลบรารี nltk สำหรับการ tokenization และใช้ Counter จากไลบรารี collections เพื่อคำนวณความถี่ของคำในข้อความ

```
from nltk.tokenize import word_tokenize

from collections import Counter

# ข้อความตัวอย่าง

text = """The cat is in the box. The cat likes the box. The box is over the cat."""

# การสร้าง tokens จากข้อความ

counter = Counter(word_tokenize(text))

# แสดงผลความถี่ของแต่ละคำ

print(counter)
```

ในโค้ดนี้:

- ใช้ word_tokenize() จาก nltk เพื่อทำการแบ่งข้อความเป็น tokens
- ใช้ Counter จาก collections เพื่อนับจำนวนความถี่ของแต่ละ token ที่เกิดขึ้นในข้อความ

เมื่อรันโค้ดนี้ จะแสดงผลลัพธ์เป็นความถี่ของแต่ละคำในข้อความ เช่น:

```
arduino Copy code
Counter({'The': 3, 'cat': 3, 'the': 3, 'box': 3, '.': 3, 'is': 2, 'in': 1, 'likes': 1, 'ov
```

ในตัวอย่างนี้ คำว่า "The", "cat", "the", และ "box" แต่ละคำปรากฏ 3 ครั้งในข้อความ

การนับคำที่พบมากที่สุด:

```
python Copy code
# นับคำที่พบน้อยที่สุด 2 คำ
count = counter.most_common(2)
print(count)
```

ผลลัพธ์ที่ได้:

```
css Copy code
[('The', 3), ('cat', 3)]
```

คำที่พบน้อยที่สุดคือ "The" และ "cat" โดยแต่ละคำปรากฏ 3 ครั้ง

หน้า 8: การทดสอบความเข้าใจเกี่ยวกับ Bag-of-Words

หน้านี้เป็นแบบฝึกหัดเพื่อทดสอบความเข้าใจของผู้อ่านเกี่ยวกับการใช้งาน Bag-of-Words โดยให้เลือกคำตอบที่ถูกต้องสำหรับการสร้าง Bag-of-Words จากข้อความที่กำหนด

ตัวอย่างคำถาม: "จากข้อความ The cat is in the box. The cat box. ให้เลือกคำตอบที่ถูกต้องสำหรับ Bag-of-Words"

- ตัวเลือก: a) ('the', 3), ('box.', 2), ('cat', 2), ('is', 1)
b) ('The', 3), ('box', 2), ('cat', 2), ('is', 1), ('in', 1), ('.', 1)
c) ('the', 3), ('cat box', 1), ('cat', 1), ('box', 1), ('is', 1), ('in', 1)
d) ('The', 2), ('box', 2), ('.', 2), ('cat', 2), ('is', 1), ('in', 1), ('the', 1)

การสร้าง Bag-of-Words ที่ถูกต้องจะได้:

- "the" ปรากฏ 3 ครั้ง

- "cat" ปรากฏ 2 ครั้ง
- "is" ปรากฏ 1 ครั้ง
- "in" ปรากฏ 1 ครั้ง
- "box" ปรากฏ 2 ครั้ง

ดังนั้นคำตอบที่ถูกต้องจะเป็นตัวเลือก:

d) ('The', 2), ('box', 2), ('.', 2), ('cat', 2), ('is', 1), ('in', 1), ('the', 1)

หาก Bag-of-Words ถูกสร้างอย่างถูกต้อง ควรจะมีการนับคำแบบไม่สนใจเครื่องหมายพิเศษหรือการใช้ตัวพิมพ์เล็ก-ใหญ่ ซึ่งหมายความว่า 'the' และ 'The' ควรจะถูกรวมกัน รวมทั้งควรตัดเครื่องหมายจุด . ออกไป

หากเราเลือกคำตอบที่ไม่สนใจตัวพิมพ์และเครื่องหมายพิเศษ:

a) ('the', 3), ('box', 2), ('cat', 2), ('is', 1), ('in', 1)

จะเป็นคำตอบที่ถูกต้องมากที่สุดที่สุดในบริบทนี้


หน้า 9: Building a Counter with Bag-of-Words

- การสร้าง Counter ด้วย Bag-of-Words โดยใช้บทความ Wikipedia เป็นตัวอย่าง
 - การนำเข้า Counter จาก collections
 - การอ่านบทความจากไฟล์ TXT
 - การ tokenization ของบทความ

หน้า 10-11: Building a Counter with Bag-of-Words (ต่อ)

- การแปลง tokens ทั้งหมดให้เป็น lowercase ด้วย list comprehension
- การสร้าง Bag-of-Words Counter เรียกว่า bow_simple
- การแสดง 10 คำที่พบบ่อยที่สุดในบทความ

python

 Copy code

```
tokens = word_tokenize(article)
lower_tokens = [t.lower() for t in tokens]
bow_simple = Counter(lower_tokens)
print(bow_simple.most_common(10))
```

หน้า 13: Simple Text Preprocessing

- การเตรียมข้อความเบื้องต้น (Preprocessing)
 - การเตรียมข้อความเป็นการทำให้ข้อมูลเหมาะสมสำหรับการนำไปใช้กับ Machine Learning หรือการวิเคราะห์ทางสถิติ
 - ตัวอย่าง:
 - การทำ tokenization เพื่อสร้าง Bag-of-Words
 - การทำ lowercase คำทั้งหมด
 - การทำ Lemmatization หรือ Stemming เพื่อลดคำให้เป็นรากศัพท์
 - การลบ stop words, สัญลักษณ์ที่ไม่จำเป็น หรือ tokens ที่ไม่ต้องการ
 - การทดลองใช้วิธีต่าง ๆ เพื่อดูผลลัพธ์ที่ดีที่สุด

หน้า 14: Preprocessing Example

- ตัวอย่างข้อความก่อนและหลังการ preprocess:
 - ข้อความก่อน preprocess: "Dogs, cats, and birds are pets. Fish and rabbit is also pets."
 - หลัง preprocess จะได้ tokens: "dog", "cat", "bird", "fish", "rabbit"

หน้า 15: Text Preprocessing with Python

- การใช้ Python ในการ preprocess ข้อความ:
 - การลบ stop words ด้วย nltk.corpus.stopwords
 - การใช้ word_tokenize เพื่อลบคำที่ไม่ใช่อักษร (non-alphabetic characters)

```
python Copy code

import nltk
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

text = "The cat is in the box. The cat likes the box. The box is over the cat in the house"
tokens = [w for w in word_tokenize(text.lower()) if w.isalpha()]
no_stops = [t for t in tokens if t not in stopwords.words('english')]
print(Counter(no_stops).most_common(2))
```

หน้า 17 ขั้นตอนการเตรียมข้อความ (Text Preprocessing)

การเตรียมข้อความ เป็นขั้นตอนสำคัญในการเตรียมข้อมูลข้อความดิบเพื่อการวิเคราะห์ด้วยโมเดลการเรียนรู้ของเครื่อง (machine learning) ขั้นตอนที่ใช้บ่อย ได้แก่:

1. **Stemming:** การลดรูปคำให้เหลือแค่รากของคำ
2. **Lemmatization:** การลดรูปคำให้เหลือแค่รูปแบบพื้นฐานตามพจนานุกรม
3. **Lowercasing:** การเปลี่ยนตัวอักษรทั้งหมดให้เป็นตัวพิมพ์เล็ก เพื่อความสม่ำเสมอ
4. **Spelling Correction:** การแก้ไขคำที่สะกดผิดเพื่อให้ข้อความมีมาตรฐานเดียวกัน
5. **Removing Stop Words:** การลบคำที่ใช้บ่อยในภาษา (เช่น "และ", "ของ") ที่ไม่จำเป็นในการวิเคราะห์
6. **Removing Unwanted Tokens:** การลบสัญลักษณ์พิเศษ ตัวเลข หรือองค์ประกอบอื่นๆ ที่ไม่จำเป็นในการวิเคราะห์
7. **Stripping Word Endings:** การตัดคำลงท้ายออก ซึ่งคล้ายกับการใช้ Stemming เพื่อลดรูปคำให้เป็นราก

การประเมินแต่ละตัวเลือก:

a) Stems, spelling corrections, lowercase.

- ตัวเลือกรวมขั้นตอนที่ถูกต้องในการเตรียมข้อความ: การลดรูปคำ (stemming), การแก้ไขคำสะกด (spelling corrections), และการเปลี่ยนตัวอักษรเป็นตัวพิมพ์เล็ก (lowercasing) ซึ่งเป็นขั้นตอนที่มีประโยชน์ในการเตรียมข้อความ

b) Lemmatization, lowercasing, removing unwanted tokens.

- ตัวเลือกรวมขั้นตอนที่ถูกต้องเช่นกัน Lemmatization, การเปลี่ยนตัวอักษรเป็นตัวพิมพ์เล็ก และการลบองค์ประกอบที่ไม่ต้องการ เป็นขั้นตอนที่พบได้บ่อยและมีประโยชน์ในการเตรียมข้อความ

c) Removing stop words, leaving in capital words.

- การลบ stop words เป็นขั้นตอนที่ถูกต้อง แต่ "การเหลือคำที่เป็นตัวพิมพ์ใหญ่ไว้" ไม่ใช่ขั้นตอนที่ถือว่าถูกต้อง เพราะปกติแล้วเราจะใช้ lowercasing เพื่อให้ข้อความมีความสม่ำเสมอ

d) Strip stop words, word endings, and digits.

- การลบ stop words และตัวเลขเป็นขั้นตอนที่ใช้บ่อย และการลบคำลงท้ายก็คล้ายกับการใช้ stemming ซึ่งเป็นขั้นตอนที่ถูกต้อง แต่ตัวเลือกนี้ค่อนข้างกว้างและไม่ละเอียดเท่าตัวเลือกอื่น

สรุป:

ตัวเลือก **a)** และ **b)** เป็นตัวเลือกที่ถูกต้อง เนื่องจากรวมขั้นตอนการเตรียมข้อความที่มีประสิทธิภาพและใช้งานบ่อย อย่างไรก็ตาม **b)** เป็นตัวเลือกที่ครอบคลุมและถูกต้องที่สุดเพราะมีการใช้ Lemmatization ซึ่งเป็นขั้นตอนที่นิยมในหลายๆ กรณี และรวมถึงการลบองค์ประกอบที่ไม่ต้องการ ทำให้ **b)** เป็นคำตอบที่ดีที่สุด

คำตอบที่ถูกต้อง: b) Lemmatization, lowercasing, removing unwanted tokens.

หน้า 18-21: Text Preprocessing Practice

- การฝึกปฏิบัติเพื่อทำการ preprocess ข้อความเพิ่มเติม:
 - การลบ stop words และตัวอักษรที่ไม่ใช่ตัวอักษร
 - การใช้ WordNetLemmatizer เพื่อทำการ lemmatize คำ
 - การสร้าง Bag-of-Words ใหม่และการแสดงคำที่พบมากที่สุด


```
from nltk.stem import WordNetLemmatizer
alpha_only = [t for t in lower_tokens if t.isalpha()] no_stops = [t for t in alpha_only if t not in
stopwords.words('english')]
wordnet_lemmatizer = WordNetLemmatizer() lemmatized = [wordnet_lemmatizer.lemmatize(t) for
t in no_stops] bow = Counter(lemmatized) print(bow.most_common(10))
```

WordNetLemmatizer เป็นเครื่องมือใน NLTK ที่ใช้สำหรับการทำ Lemmatization ซึ่งเป็นการแปลงคำให้เป็นรูปแบบฐาน (root form) เช่น คำว่า "running" จะถูกแปลงเป็น "run"

โค้ดบรรทัดนี้จะสร้างลิสต์ alpha_only ที่ประกอบด้วยเฉพาะคำที่เป็นตัวอักษร (ไม่มีตัวเลขหรือสัญลักษณ์) โดยใช้ isalpha() เพื่อตรวจสอบแต่ละ token ว่าเป็นตัวอักษรล้วนหรือไม่

โค้ดนี้จะลบคำที่ไม่จำเป็น (Stop Words) เช่น "is", "the", "in" ออกจากลิสต์ โดยใช้

stopwords.words('english') จาก NLTK เพื่อดึงรายการคำที่เป็น Stop Words ในภาษาอังกฤษ

โค้ดนี้จะสร้างลิสต์ lemmatized ที่ประกอบด้วยคำที่ผ่านการทำ Lemmatization แล้ว โดยใช้

wordnet_lemmatizer.lemmatize() กับแต่ละคำในลิสต์ no_stops

สุดท้าย โค้ดนี้จะสร้าง Bag-of-Words Counter โดยใช้ Counter จากลิสต์ lemmatized และแสดงคำที่พบบ่อยที่สุด 10 คำพร้อมจำนวนครั้งที่ปรากฏ

บทที่ 3.2

หน้า 2: สารบัญ

- บทนี้ประกอบด้วยหัวข้อหลักสองส่วน:
 1. การแนะนำ Gensim
 2. การใช้งาน TF-IDF ร่วมกับ Gensim

หน้า 4: การแนะนำ Gensim

- **Gensim** เป็นไลบรารีโอเพนซอร์สยอดนิยมสำหรับการประมวลผลภาษาธรรมชาติ (NLP)

- Gensim ใช้โมเดลระดับวิชาการในการทำงาน เช่น การสร้างเอกสารหรือเวกเตอร์ของคำ (Word vectors) และการเปรียบเทียบเอกสาร

Gensim เป็นไลบรารีที่ได้รับความนิยมอย่างมากในงานประมวลผลภาษาธรรมชาติ (Natural Language Processing หรือ NLP) ที่เป็นโอเพ่นซอร์ส โดย Gensim ถูกใช้ในงานด้านวิชาการชั้นนำหลายด้านสำหรับการทำงานที่ซับซ้อน เช่น:

1. **การสร้างเวกเตอร์ของเอกสารหรือคำศัพท์:** Gensim สามารถสร้างเวกเตอร์ที่แทนเอกสารหรือคำศัพท์ ซึ่งเวกเตอร์เหล่านี้มักเรียกว่า "การฝังคำ" หรือ "การเวกเตอร์โทเซชันของคำ" ซึ่งเป็นวิธีการใน NLP ที่ใช้ในการแปลงคำให้เป็นรูปแบบที่คอมพิวเตอร์สามารถเข้าใจได้ โดยการแปลงคำเป็นเวกเตอร์ตัวเลข ซึ่งมีความสำคัญต่อการทำงานต่างๆ ของ NLP เช่น การทำนายคำถัดไป การหาความเหมือนกันของคำ และการทำความเข้าใจความหมายของคำ
2. **การระบุหัวข้อ (Topic Identification):** Gensim มีความสามารถในการระบุหัวข้อหลักที่อยู่ในชุดเอกสารต่างๆ ซึ่งมีประโยชน์ในการสรุปข้อมูลจากชุดข้อความขนาดใหญ่ การจัดประเภทเอกสาร และแม้กระทั่งในการสร้างระบบแนะนำเนื้อหา
3. **การเปรียบเทียบเอกสาร:** ไลบรารีนี้ยังสนับสนุนการเปรียบเทียบเอกสารต่างๆ เพื่อกำหนดความเหมือนหรือวิเคราะห์ความสัมพันธ์ระหว่างเอกสาร ซึ่งมีประโยชน์ในการจัดกลุ่มเอกสารที่คล้ายกันเข้าด้วยกัน หรือค้นหาเอกสารที่มีเนื้อหาคล้ายคลึงกัน

การฝังคำหรือการเวกเตอร์โทเซชันของคำ เป็นกระบวนการในการแปลงคำศัพท์หรือวลีจากคำศัพท์ในภาษาธรรมชาติให้เป็นเวกเตอร์ของตัวเลขจริง โดยเวกเตอร์เหล่านี้จะถูกนำไปใช้ในงาน NLP ต่างๆ เช่น การทำนายคำ การหาความเหมือนของคำ และการทำความเข้าใจความหมายของคำ

ในภาพยังมีการแสดงวิธีการที่ใช้ในการแปลงความถี่ของคำในเอกสารให้เป็นเวกเตอร์ ซึ่งถูกนำไปใช้ในการวิเคราะห์เพิ่มเติมในงาน NLP ตารางที่แสดงในภาพคือการกระจายตัวของคำในเอกสารต่างๆ โดยแต่ละแถวแทนคำศัพท์และแต่ละคอลัมน์แทนเอกสาร ค่าตัวเลขในตารางแสดงถึงความถี่ของแต่ละคำในเอกสารนั้นๆ ซึ่งข้อมูลนี้จะถูกนำไปสร้างเวกเตอร์ของคำหรือเอกสารต่อไป เวกเตอร์เหล่านี้ช่วยในการจับความหมายของคำในบริบทต่างๆ ของเอกสาร ซึ่งมีประโยชน์ในการทำหัวข้อโมเดลและการเปรียบเทียบความเหมือนของเอกสาร.

หน้า 5: การสร้างดัชนีด้วย Gensim

- Gensim ช่วยให้สามารถสร้างคอร์ปัสและดัชนีด้วยคลาสและฟังก์ชันที่ง่าย
- คอร์ปัส (corpus) คือชุดของข้อความที่ใช้ในการทำงานด้านการประมวลผลภาษาธรรมชาติ

หน้า 7: การสร้างและการสืบค้นคอร์ปัสด้วย Gensim

- วิธีการสร้างดัชนีและคอร์ปัสด้วย Gensim:
 - ติดตั้ง Gensim โดยใช้ pip install gensim
 - สร้างรายการของ token จากเอกสาร (articles) โดยการ preprocess ข้อความ เช่น การแปลงเป็นตัวอักษรเล็ก การ tokenization การลบ stop words และการลบเครื่องหมายวรรคตอน

หน้า 8: การสร้างรายการของ document tokens

python

Copy code

```
articles = []
for i in range(10):
    # อ่านไฟล์บทความ
    f = open(f".\ch3\wiki\wiki_article_{i}.txt", "r")
    article = f.read()
    # ทำการ Tokenize บทความ
    tokens = word_tokenize(article)
    # แปลง tokens ให้เป็นตัวอักษรเล็ก
    lower_tokens = [t.lower() for t in tokens]
    # กรองเฉพาะคำที่เป็นตัวอักษร
    alpha_only = [t for t in lower_tokens if t.isalpha()]
    # ลบ stop words
    no_stops = [t for t in alpha_only if t not in stopwords.words('english')]
    # สร้าง WordNetLemmatizer
    wordnet_lemmatizer = WordNetLemmatizer()
    # ทำ Lemmatize กับทุกคำ
    lemmatized = [wordnet_lemmatizer.lemmatize(t) for t in no_stops]
    # เพิ่มรายการ lemmatized ลงใน articles
    articles.append(lemmatized)

print(articles[0])
```

อธิบายได้:

1. อ่านไฟล์บทความ: อ่านเนื้อหาจากไฟล์ .txt และบันทึกในตัวแปร article
2. **Tokenize**: ทำการแบ่งข้อความในบทความเป็น tokens (คำแต่ละคำ) ด้วย word_tokenize()
3. แปลงเป็นตัวอักษรเล็ก: ใช้ lower() เพื่อแปลงคำทุกคำให้เป็นตัวอักษรเล็ก
4. กรองคำที่เป็นตัวอักษร: ใช้ isalpha() เพื่อตรวจสอบและคงไว้เฉพาะคำที่เป็นตัวอักษรล้วน
5. ลบ **Stop Words**: ลบคำที่ไม่จำเป็นออกไปจากรายการ เช่น "the", "is", "in"
6. **Lemmatization**: ใช้ WordNetLemmatizer เพื่อแปลงคำให้เป็นรากศัพท์

7. บันทึกในรายการ: เพิ่มรายการ lemmatized ลงในลิสต์ articles

หน้า 9: การสร้างและการสืบค้นคอร์ปัสด้วย Gensim

โค้ดตัวอย่าง:

```
python Copy code  
  
# นำเข้า Dictionary จาก gensim.corpora  
from gensim.corpora import Dictionary  
  
# สร้างดิกชันนารีจาก articles  
dictionary = Dictionary(articles)
```

อธิบายโค้ด:

- โค้ดนี้จะสร้างดิกชันนารีโดยใช้ Dictionary จาก gensim.corpora ซึ่งจะนำ tokens ที่อยู่ใน articles มาสร้างเป็นดิกชันนารีที่มีการจับคู่ระหว่างคำ (tokens) กับหมายเลขประจำตัว (token IDs)

หน้า 10: การสืบค้นข้อมูลจากดิกชันนารี

```
python  
  
# หาค่า id สำหรับคำว่า "computer"  
computer_id = dictionary.token2id.get("computer")  
  
# ใช้ id เพื่อแสดงคำที่ตรงกับ id นั้น  
print(dictionary.get(computer_id))
```

อธิบายโค้ด:

- โค้ดนี้จะใช้ .token2id เพื่อหาหมายเลขประจำตัว (id) ของคำที่ต้องการ เช่น "computer" และใช้ .get() เพื่อแสดงคำที่ตรงกับ id นั้น

หน้า 11: การสร้างคอร์ปัสด้วย Gensim

โค้ดตัวอย่าง:

```
python Copy code  
  
# สร้างคอร์ปัสจากดิกชันนารี  
corpus = [dictionary.doc2bow(a) for a in articles]
```

อธิบายโค้ด:

- `doc2bow()` เป็นวิธีการแปลงเอกสาร (บทความ) ให้เป็น Bag-of-Words (BoW) โดยจะนับจำนวนครั้งที่แต่ละคำปรากฏในเอกสารนั้น ๆ และบันทึกเป็นรายการในคอร์ปัส

หน้า 12: Gensim Bag-of-Words

โค้ดตัวอย่าง:

```
python  
  
from collections import defaultdict  
  
# บันทึกเอกสารที่สอง  
doc = corpus[1]  
  
# เรียงลำดับคำในเอกสารตามความถี่  
bow_doc = sorted(doc, key=lambda w: w[1], reverse=True)
```

อธิบายโค้ด:

- โค้ดนี้ใช้ `defaultdict` จาก `collections` เพื่อสร้างโครงสร้างข้อมูลสำหรับการวิเคราะห์ต่อไป
- `sorted()` จะเรียงลำดับคำในเอกสารตามจำนวนครั้งที่คำปรากฏในเอกสารจากมากไปน้อย

หน้า 13: การแสดงคำที่พบบ่อยที่สุดในเอกสาร

python

```
# แสดงคำที่พบบ่อยที่สุด 5 คำในเอกสาร
for word_id, word_count in bow_doc[:5]:
    print(dictionary.get(word_id), word_count)
```


อธิบายโค้ด:

- โค้ดนี้จะแสดงคำที่พบบ่อยที่สุด 5 คำในเอกสาร `doc` พร้อมจำนวนครั้งที่แต่ละคำปรากฏ โดยใช้ `id` ของคำจากดิกชันนารีเพื่อนำคำที่ตรงกับ `id` นั้นมาแสดงผล

หน้า 14: การสร้างรายการคำจากคอร์ปัสทั้งหมด

โค้ดตัวอย่าง:

python

 Copy code

```
# สร้าง defaultdict สำหรับนับจำนวนคำทั้งหมด
total_word_count = defaultdict(int)
for word_id, word_count in itertools.chain.from_iterable(corpus):
    total_word_count[word_id] += word_count
```

อธิบายโค้ด:

- โค้ดนี้สร้าง `defaultdict` ที่เก็บจำนวนครั้งที่คำแต่ละคำปรากฏในคอร์ปัสทั้งหมด โดยใช้ `itertools.chain.from_iterable` เพื่อนำเอกสารทั้งหมดมารวมกันเป็นรายการเดียว

หน้า 15: การแสดงคำที่พบบ่อยที่สุดในคอร์ปัสทั้งหมด

โค้ดตัวอย่าง:

```
python Copy code  
  
# สร้างรายการคำเรียงตามความถี่จากมากไปน้อย  
sorted_word_count = sorted(total_word_count.items(), key=lambda w: w[1], reverse=True)  
  
# แสดงคำที่พบบ่อยที่สุด 5 คำในคอร์ปัสทั้งหมด  
for word_id, word_count in sorted_word_count[:5]:  
    print(dictionary.get(word_id), word_count)
```

อธิบายโค้ด:

- โค้ดนี้จะแสดงคำที่พบบ่อยที่สุด 5 คำในคอร์ปัสทั้งหมด พร้อมจำนวนครั้งที่แต่ละคำปรากฏ โดยเรียงตามลำดับจากมากไปน้อย

หน้า 17: การใช้ TF-IDF ร่วมกับ Gensim

- TF-IDF (Term Frequency-Inverse Document Frequency) เป็นเทคนิคที่ใช้ในการหาคำที่สำคัญในแต่ละเอกสารในคอร์ปัส โดยการพิจารณาความถี่ของคำในเอกสารและความถี่ของคำในคอร์ปัสทั้งหมด
- เทคนิคนี้จะช่วยให้คำที่พบในเอกสารเดียวกันมาก ๆ แต่ไม่พบในเอกสารอื่นมีน้ำหนักมากขึ้น และคำที่พบในทุก

หน้า 18: สูตรการคำนวณ TF-IDF

- **TF-IDF (Term Frequency-Inverse Document Frequency):** เป็นสูตรที่ใช้ในการคำนวณความสำคัญของคำในเอกสารหนึ่ง ๆ โดยพิจารณาจากความถี่ที่คำนั้นปรากฏในเอกสาร (Term Frequency หรือ TF) และจำนวนเอกสารทั้งหมดที่มีคำนั้นปรากฏอยู่ (Inverse Document Frequency หรือ IDF)

สูตร: $w_{i,j} = tf_{i,j} \times \log \left(\frac{N}{df_i} \right)$

ที่:

- $w_{i,j}$ คือค่าน้ำหนัก TF-IDF ของคำที่ i ในเอกสารที่ j
- $tf_{i,j}$ คือความถี่ที่คำ i ปรากฏในเอกสาร j
- df_i คือจำนวนเอกสารที่มีคำ i ปรากฏอยู่
- N คือจำนวนเอกสารทั้งหมด

หน้า 19: การใช้ TF-IDF ร่วมกับ Gensim

- Gensim สามารถใช้ในการสร้างโมเดล TF-IDF จากคอร์ปัสที่สร้างขึ้นในขั้นตอนก่อนหน้า
- สำหรับเอกสารแรกในคอร์ปัส เราจะเห็นน้ำหนักของแต่ละคำ (token) พร้อมกับ ID ของคำเหล่านั้น

โค้ดตัวอย่าง:

```
python Copy code  
  
from gensim.models.tfidfmodel import TfidfModel  
  
# สร้างโมเดล TF-IDF จากคอร์ปัส  
tfidf = TfidfModel(corpus)  
  
# แสดงน้ำหนัก TF-IDF ของเอกสารแรก  
print(tfidf[corpus[0]])
```

ผลลัพธ์ที่คาดหวัง:

```
python Copy code  
  
[(0, 0.04637388957601683),  
 (1, 0.04637388957601683),  
 (2, 0.04637388957601683),  
 (3, 0.04637388957601683),  
 (5, 0.04637388957601683),  
 ...  
 (33, 0.4637388957601683)]
```



- ค่าน้ำหนักที่แสดงจะช่วยให้เราสามารถระบุค่าที่เป็นหัวข้อหรือคีย์เวิร์ดสำคัญในเอกสารนั้น ๆ ได้

หน้า 21: แบบฝึกหัดการคำนวณ TF-IDF

- ในหน้านี้มีการฝึกคำนวณค่าน้ำหนัก TF-IDF สำหรับคำว่า "computer" ซึ่งปรากฏในเอกสารหนึ่ง 5 ครั้ง โดยที่เอกสารนั้นมีคำทั้งหมด 100 คำ และมีเอกสารทั้งหมดในคอร์ปัส 200 เอกสาร โดยที่ 20 เอกสารมีคำว่า "computer" ปรากฏอยู่

What is TF-IDF?

- You want to calculate the TF-IDF weight for the word "computer", which appears 5 times in a document containing 100 words. Given a corpus containing 200 documents, with 20 documents mentioning the word "computer", TF-IDF can be calculated by multiplying term frequency with inverse document frequency.
- Which of the below options is correct?
 - a) $(5 / 100) * \log(200 / 20)$
 - b) $(5 * 100) / \log(200 * 20)$
 - c) $(20 / 5) * \log(200 / 20)$
 - d) $(200 * 5) * \log(400 / 5)$

TF-IDF (Term Frequency-Inverse Document Frequency) คือวิธีการหนึ่งใน NLP ที่ใช้ในการประเมินความสำคัญของคำในเอกสารหนึ่ง ๆ ภายในชุดเอกสาร (corpus) โดยคำนวณจากสองส่วนหลักคือ:

1. **Term Frequency (TF):** ความถี่ของคำในเอกสารนั้น โดยทั่วไปคำนวณจากจำนวนครั้งที่คำปรากฏในเอกสารหารด้วยจำนวนคำทั้งหมดในเอกสาร เช่น ในกรณีนี้คำว่า "computer" ปรากฏ 5 ครั้งในเอกสารที่มี 100 คำ ดังนั้น $TF = 5/100$
2. **Inverse Document Frequency (IDF):** ความถี่ผกผันของเอกสารที่มีคำนี้อยู่ คำนวณจากจำนวนเอกสารทั้งหมดใน corpus หารด้วยจำนวนเอกสารที่มีคำนี้ปรากฏ และนำไปลอการิทึมฐาน 10 เพื่อให้ค่าที่ได้มีความสำคัญที่แตกต่างกัน เช่น ในกรณีนี้มีเอกสารทั้งหมด 200 ฉบับ และมี 20 ฉบับที่มีคำว่า "computer" ดังนั้น $IDF = \log(200/20)$

เมื่อคำนวณค่า TF-IDF จะเป็นการนำค่า TF คูณกับค่า IDF

จากข้อมูลที่ให้มา เราสามารถคำนวณค่า TF-IDF สำหรับคำว่า "computer" ได้ดังนี้:

- $TF = 5/100$
- $IDF = \log(200/20)$

ดังนั้นค่า TF-IDF = $(5/100) * \log(200/20)$


คำตอบที่ถูกต้องคือ ตัวเลือก a).

หน้า 22: การใช้ TF-IDF ร่วมกับ Wikipedia

- ในหน้านี้จะอธิบายถึงการเข้าถึงคอร์ปัสและดัชนีหารที่สร้างขึ้นจากบทก่อนหน้านี้ เพื่อสร้างโมเดล TF-IDF ด้วย Gensim

โค้ดตัวอย่าง:

python

 Copy code

```
from gensim.models.tfidfmodel import TfidfModel

# สร้างโมเดล TF-IDF ใหม่จากคอร์ปัส
tfidf = TfidfModel(corpus)
```

หน้า 23: การสร้างคอร์ปัสและดิกชันนารี (ต่อจากหน้า 22)

```
articles = []
for i in range(10):
    # อ่านไฟล์บทความ
    f = open(f".\ch3\wiki\wiki_article_{i}.txt", "r")
    article = f.read()
    # ทำการ Tokenize บทความ
    tokens = word_tokenize(article)
    # แปลง tokens ให้เป็นตัวอักษรเล็ก
    lower_tokens = [t.lower() for t in tokens]
    # กรองเฉพาะคำที่เป็นตัวอักษร
    alpha_only = [t for t in lower_tokens if t.isalpha()]
    # ลบ stop words
    no_stops = [t for t in alpha_only if t not in stopwords.words('english')]
    # สร้าง WordNetLemmatizer
    wordnet_lemmatizer = WordNetLemmatizer()
    # ทำ Lemmatize กับทุกคำ
    lemmatized = [wordnet_lemmatizer.lemmatize(t) for t in no_stops]
    # เพิ่มรายการ lemmatized ลงใน articles
    articles.append(lemmatized)

# สร้างดิกชันนารีจาก articles
dictionary = Dictionary(articles)
# สร้างคอร์ปัสจากดิกชันนารี
corpus = [dictionary.doc2bow(a) for a in articles]
# บันทึกเอกสารที่สอง
doc = corpus[1]
```

อธิบายได้:

1. อ่านไฟล์บทความ: อ่านเนื้อหาจากไฟล์ .txt และบันทึกในตัวแปร article
2. Tokenize: ทำการแบ่งข้อความในบทความเป็น tokens (คำแต่ละคำ) ด้วย word_tokenize()
3. แปลงเป็นตัวอักษรเล็ก: ใช้ lower() เพื่อแปลงคำทุกคำให้เป็นตัวอักษรเล็ก

4. กรองคำที่เป็นตัวอักษร: ใช้ `isalpha()` เพื่อตรวจสอบและคงไว้เฉพาะคำที่เป็นตัวอักษรล้วน
5. ลบ Stop Words: ลบคำที่ไม่จำเป็นออกไปจากรายการ เช่น "the", "is", "in"
6. Lemmatization: ใช้ `WordNetLemmatizer` เพื่อแปลงคำให้เป็นรากศัพท์
7. บันทึกในรายการ: เพิ่มรายการ `lemmatized` ลงในลิสต์ `articles`
8. สร้างดิกชันนารี: ใช้ `Dictionary` เพื่อสร้างดิกชันนารีจากรายการ `articles`
9. สร้างคอร์ปัส: แปลงเอกสารเป็น Bag-of-Words (BoW) ด้วย `doc2bow()`

หน้า 24: การคำนวณน้ำหนัก TF-IDF ด้วย Gensim

```
python

# สร้างโมเดล TF-IDF ใหม่จากคอร์ปัส
tfidf = TfidfModel(corpus)

# คำนวณน้ำหนัก TF-IDF ของเอกสาร
tfidf_weights = tfidf[doc]

# แสดงน้ำหนัก 5 คำแรก
print(tfidf_weights[:5])
```

อธิบายโค้ด:

- โค้ดนี้จะสร้างโมเดล TF-IDF ใหม่จากคอร์ปัสที่มีอยู่ จากนั้นคำนวณน้ำหนัก TF-IDF ของเอกสารที่สอง (`doc`) และแสดงน้ำหนัก 5 คำแรกในเอกสารนั้น

```
[(0, 0.04637388957601683),
 (1, 0.04637388957601683),
 (2, 0.04637388957601683),
 (3, 0.04637388957601683),
 (5, 0.04637388957601683)]
```

หน้า 25: การเรียงลำดับน้ำหนัก TF-IDF

python

```
# เรียงลำดับน้ำหนักจากมากไปน้อย
sorted_tfidf_weights = sorted(tfidf_weights, key=lambda w: w[1], reverse=True)

# แสดงคำที่มีน้ำหนักสูงสุด 5 คำ
for term_id, weight in sorted_tfidf_weights[:5]:
    print(dictionary.get(term_id), weight)
```

อธิบายโค้ด:

- โค้ดนี้จะเรียงลำดับน้ำหนัก TF-IDF ของคำในเอกสารจากมากไปน้อย และแสดงคำที่มีน้ำหนักสูงสุด 5 คำพร้อมกับค่าน้ำหนักของแต่ละคำ

ที่เราฟรีเซ็น

ตัวอย่างของ Toolkits และ Libraries

1. NLTK (Natural Language Toolkit):

- **การใช้งาน:** NLTK เป็นไลบรารีที่มีเครื่องมือสำหรับการทำงานกับภาษาธรรมชาติ ซึ่งเหมาะสำหรับการศึกษาและวิจัยเกี่ยวกับการประมวลผลภาษาธรรมชาติ (NLP). มันช่วยให้นักพัฒนาสามารถทำงานกับข้อความและดำเนินการต่างๆ เช่น การตัดคำ (tokenization), การแท็กชนิดของคำ (part-of-speech tagging), และการวิเคราะห์ทางภาษา (parsing).
- **ประโยชน์:**
 - **การศึกษา:** เป็นเครื่องมือที่ดีสำหรับการเรียนรู้และทดลองเกี่ยวกับ NLP
 - **การวิจัย:** ใช้ในการวิจัยและพัฒนาโมเดลการประมวลผลภาษาที่ซับซ้อน
 - **การพัฒนา:** ช่วยในการพัฒนาโปรเจกต์ที่ต้องการการวิเคราะห์และทำความเข้าใจข้อความ
- **ฟังก์ชันที่สามารถทำได้:**
 - การตัดคำและแยกประเภท

- การวิเคราะห์โครงสร้างประโยค
- การจัดหมวดหมู่คำและการทำความเข้าใจความหมายของข้อความ

2. SpaCy:

- **การใช้งาน:** SpaCy เป็นไลบรารีสำหรับการประมวลผลภาษาธรรมชาติที่เน้นประสิทธิภาพสูงและความเร็วในการทำงาน. มันออกแบบมาเพื่อใช้ในแอปพลิเคชันที่ต้องการความเร็วและความแม่นยำในการวิเคราะห์ข้อมูลภาษาธรรมชาติ.
- **ประโยชน์:**
 - **ประสิทธิภาพสูง:** สามารถประมวลผลข้อความจำนวนมากได้อย่างรวดเร็ว
 - **ความแม่นยำ:** ใช้โมเดลที่ได้รับการฝึกฝนมาอย่างดีเพื่อให้ได้ผลลัพธ์ที่แม่นยำ
 - **การใช้งานในเชิงพาณิชย์:** เหมาะสำหรับการพัฒนาแอปพลิเคชันที่ต้องการการวิเคราะห์ภาษาขั้นสูง
- **ฟังก์ชันที่สามารถทำได้:**
 - การระบุเอนทิตี (Named Entity Recognition)
 - การสร้างเวกเตอร์ของคำ (Word Embeddings)
 - การวิเคราะห์ข้อความในเชิงลึก เช่น การตรวจจับความรู้สึกและการจัดประเภท

3. TextBlob:

- **การใช้งาน:** TextBlob เป็นไลบรารีที่ออกแบบมาเพื่อการประมวลผลข้อความที่ง่ายและสะดวกในการใช้งาน. มันเหมาะสำหรับการประมวลผลภาษาที่ไม่ซับซ้อนและการทำงานที่ต้องการผลลัพธ์ที่รวดเร็ว.
- **ประโยชน์:**
 - **ใช้งานง่าย:** มี API ที่เข้าใจง่ายสำหรับการทำงานกับข้อความ
 - **การวิเคราะห์พื้นฐาน:** เหมาะสำหรับการวิเคราะห์ข้อความพื้นฐาน เช่น การวิเคราะห์ความรู้สึก

- การแปลภาษา: สนับสนุนการแปลข้อความระหว่างภาษา
- ฟังก์ชันที่สามารถทำได้:
 - การวิเคราะห์ความรู้สึก
 - การแปลข้อความ
 - การตัดคำและการจัดหมวดหมู่คำ

ตัวอย่างของบริการ Cloud

1. Google Cloud Natural Language API:

- การใช้งาน: Google Cloud Natural Language API ให้บริการการวิเคราะห์ข้อความจาก Google Cloud. มันใช้เทคโนโลยีการเรียนรู้ของเครื่องเพื่อทำความเข้าใจและวิเคราะห์ข้อความที่มีความซับซ้อน.
- ประโยชน์:
 - การวิเคราะห์เชิงลึก: ให้การวิเคราะห์ที่ลึกซึ้งเกี่ยวกับข้อความ
 - การสนับสนุนหลายภาษา: รองรับการวิเคราะห์ข้อความในหลายภาษา
 - ความสามารถในการปรับขนาด: สามารถจัดการกับข้อความจำนวนมากได้
- ฟังก์ชันที่สามารถทำได้:
 - การวิเคราะห์ความรู้สึก
 - การระบุเอนทิตี
 - การวิเคราะห์โครงสร้างข้อความ

2. Amazon Comprehend:

- การใช้งาน: Amazon Comprehend เป็นบริการ NLP ของ AWS ที่ใช้การเรียนรู้ของเครื่องเพื่อวิเคราะห์และสกัดข้อมูลจากข้อความ. มันสามารถทำการวิเคราะห์ข้อความเพื่อค้นหาข้อมูลที่มีค่าและการเชื่อมโยงต่างๆ.

- ประโยชน์:
 - การวิเคราะห์ที่ชาญฉลาด: ใช้การเรียนรู้ของเครื่องเพื่อให้ข้อมูลเชิงลึกจากข้อความ
 - การตรวจจับหัวข้อ: สามารถตรวจจับหัวข้อหลักในข้อความ
 - การสนับสนุนหลายภาษา: รองรับหลายภาษา
- ฟังก์ชันที่สามารถทำได้:
 - การวิเคราะห์ความรู้สึก
 - การระบุเอนทิตี
 - การตรวจจับหัวข้อ

3. Azure AI Language:

- การใช้งาน: Azure AI Language เป็นบริการที่จัดการจาก Microsoft Azure ที่ช่วยในการพัฒนาแอปพลิเคชัน NLP. มันมีเครื่องมือที่ช่วยในการวิเคราะห์และสร้างข้อความ.
- ประโยชน์:
 - การพัฒนาแอปพลิเคชัน: ช่วยในการสร้างและปรับปรุงแอปพลิเคชันที่เกี่ยวข้องกับการวิเคราะห์ข้อความ
 - การวิเคราะห์ที่ครอบคลุม: ให้ความสามารถในการวิเคราะห์และสร้างข้อความ
 - การรวมบริการ: สามารถรวมเข้ากับบริการอื่นๆ ของ Azure ได้
- ฟังก์ชันที่สามารถทำได้:
 - การวิเคราะห์ความรู้สึก
 - การสร้างข้อความ
 - การระบุเอนทิตีและการจำแนกประเภท

1. Amazon Alexa

- การทำงาน:
 - Amazon Alexa เป็นผู้ช่วยเสมือนที่พัฒนาโดย Amazon และทำงานผ่านอุปกรณ์ Echo ของ Amazon และอุปกรณ์ที่รองรับ Alexa. Alexa ใช้การประมวลผลภาษาธรรมชาติ (NLP) เพื่อเข้าใจและตอบสนองต่อคำสั่งเสียงจากผู้ใช้.
- การใช้:
 - ควบคุมอุปกรณ์อัจฉริยะ: สามารถควบคุมอุปกรณ์ในบ้าน เช่น ไฟ, เครื่องปรับอากาศ, และอุปกรณ์สมาร์ตอื่นๆ.
 - สั่งงานและการค้นหา: ใช้เพื่อสั่งงานต่างๆ เช่น เล่นเพลง, ตั้งการแจ้งเตือน, และให้ข้อมูลข่าวสาร.
 - การสร้างสรรค์: ใช้ Alexa Skills Kit (ASK) เพื่อสร้างสรรค์ฟังก์ชันใหม่ๆ ที่สามารถเพิ่มเข้าไปใน Alexa.
- ข้อดี:
 - การบูรณาการที่กว้างขวาง: รองรับการบูรณาการกับอุปกรณ์สมาร์ตหลายชนิด.
 - การทำงานร่วมกับบริการของ Amazon: เช่น Amazon Music, Amazon Prime, และอื่นๆ.
 - การสร้างสรรค์ที่เปิดกว้าง: ช่วยให้ผู้พัฒนาสามารถสร้างสรรค์ทักษะใหม่ๆ ได้.
- ข้อเสีย:
 - ข้อกังวลด้านความเป็นส่วนตัว: การบันทึกและส่งข้อมูลเสียงไปยัง Amazon อาจก่อให้เกิดความกังวลด้านความเป็นส่วนตัว.
 - การทำงานในพื้นที่ที่ไม่รองรับ: บางฟังก์ชันอาจไม่สามารถใช้ได้บางภูมิภาค.

2. Google Assistant

- การทำงาน:
 - Google Assistant เป็นผู้ช่วยเสมือนที่พัฒนาโดย Google, ทำงานบนอุปกรณ์ที่รองรับ Google เช่น สมาร์ทโฟน, Google Home, และอุปกรณ์อื่นๆ. ใช้เทคโนโลยีการประมวลผลภาษาธรรมชาติและการเรียนรู้ของเครื่องเพื่อให้การตอบสนองที่แม่นยำ.
- การใช้:
 - ควบคุมอุปกรณ์อัจฉริยะ: สามารถควบคุมอุปกรณ์ที่เชื่อมต่อ Google Home ได้.
 - ค้นหาข้อมูล: ใช้สำหรับค้นหาข้อมูลจาก Google, รับข่าวสาร, หรือค้นหาสถานที่.
 - การทำงานร่วมกับแอป: สามารถเชื่อมโยงกับแอปต่างๆ และให้ความสามารถในการทำงานที่เกี่ยวข้องกับแอปนั้นๆ.
- ข้อดี:
 - การรวมกับบริการของ Google: เช่น Google Calendar, Gmail, และ Google Maps.
 - การทำงานกับ AI ที่พัฒนาอย่างต่อเนื่อง: พัฒนาการตอบสนองที่แม่นยำมากขึ้นตามการใช้.
 - การรองรับหลายภาษา: รองรับหลายภาษาทั่วโลก.
- ข้อเสีย:
 - การควบคุมที่จำกัดในบางพื้นที่: บางฟังก์ชันอาจไม่สามารถใช้งานได้บางภูมิภาค.
 - ข้อกังวลด้านความเป็นส่วนตัว: ข้อมูลเสียงที่บันทึกอาจมีข้อกังวลเกี่ยวกับความเป็นส่วนตัว.

3. Siri

- การทำงาน:
 - Siri เป็นผู้ช่วยเสมือนที่พัฒนาโดย Apple, ทำงานบนอุปกรณ์ Apple เช่น iPhone, iPad, Mac, และ HomePod. ใช้เทคโนโลยีการประมวลผลภาษาธรรมชาติเพื่อให้บริการตอบสนองต่อคำสั่งเสียงของผู้ใช้.
- การใช้:

- ควบคุมอุปกรณ์ Apple: สามารถใช้เพื่อควบคุมอุปกรณ์ต่างๆ ของ Apple เช่น ตั้งการแจ้งเตือน, ส่งข้อความ, และเปิดแอปพลิเคชัน.
- ค้นหาข้อมูล: ใช้เพื่อค้นหาข้อมูล, ตั้งการแจ้งเตือน, และการช่วยในการทำงาน.
- การทำงานร่วมกับแอป: สามารถเชื่อมโยงกับแอปต่างๆ ที่รองรับ Siri.
- ข้อดี:
 - การรวมกับระบบของ Apple: ทำงานได้อย่างราบรื่นกับระบบปฏิบัติการ iOS, macOS, และอุปกรณ์ Apple อื่นๆ.
 - การควบคุมด้วยเสียงที่เป็นธรรมชาติ: ให้การตอบสนองที่เป็นธรรมชาติและเข้าใจได้ง่าย.
 - ความปลอดภัยและความเป็นส่วนตัว: Apple ให้ความสำคัญกับความเป็นส่วนตัวของผู้ใช้.
- ข้อเสีย:
 - ข้อจำกัดในการรองรับหลายแพลตฟอร์ม: ไม่สามารถใช้งานร่วมกับอุปกรณ์ที่ไม่ใช่ของ Apple.
 - การใช้งานที่จำกัดในบางพื้นที่: บางฟังก์ชันอาจไม่สามารถใช้ได้บางภูมิภาค.

1. Cosine Similarity (ความคล้ายคลึงแบบโคไซน์)

- ความหมาย:
 - ใช้เพื่อวัดความคล้ายคลึงของสองเวกเตอร์โดยคำนวณจากมุมระหว่างเวกเตอร์ โดยไม่สนใจขนาดของเวกเตอร์ ค่าจะอยู่ในช่วง -1 (แตกต่างกันโดยสิ้นเชิง) ถึง 1 (คล้ายคลึงกันโดยสิ้นเชิง) และ 0 หมายถึงไม่มีความคล้ายคลึง
- สูตร:

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

โดยที่ $\mathbf{A} \cdot \mathbf{B}$ คือผลคูณเชิงสเกลาร์ของเวกเตอร์ \mathbf{A} และ \mathbf{B} , และ $\|\mathbf{A}\|$ กับ $\|\mathbf{B}\|$ คือขนาดของเวกเตอร์ \mathbf{A} และ \mathbf{B}

- การใช้งาน:

- ใช้ในการวิเคราะห์ข้อความและประมวลผลภาษาธรรมชาติ (NLP) เพื่อตรวจสอบความคล้ายคลึงของเอกสารหรือข้อความตามการแทนค่าด้วยเวกเตอร์
- ตัวอย่าง:
 - หากเวกเตอร์ A และ B แทนเอกสารสองฉบับในเวกเตอร์พื้นที่การนับคำ ความคล้ายคลึงแบบโคไซน์สามารถใช้เพื่อวัดความคล้ายคลึงกันของเนื้อหาของเอกสารเหล่านั้น

2. Euclidean Distance (ระยะทางยูคลิด)

- ความหมาย:
 - วัดระยะทางตรงระหว่างสองจุดในพื้นที่ยูคลิด ซึ่งเป็นระยะทางที่ตรงที่สุดระหว่างสองจุด
- สูตร:

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

โดยที่ x_i และ y_i คือพิกัดของจุดในพื้นที่ n มิติ

- การใช้งาน:

ใช้ในการจัดกลุ่ม การจำแนกประเภท และการรู้จำรูปแบบ เพื่อวัดความคล้ายคลึงระหว่างจุดข้อมูล
- ตัวอย่าง:
 - หากคุณมีจุดสองจุด (3, 4) และ (7, 1) ในพื้นที่ 2 มิติ ระยะทางยูคลิดระหว่างพวกเขาคือ:

$$\sqrt{(7 - 3)^2 + (1 - 4)^2} = \sqrt{16 + 9} = \sqrt{25} = 5$$

3. Jaccard Similarity (ความคล้ายคลึงแบบแจ็กการ์ด)

- ความหมาย:
 - วัดความคล้ายคลึงระหว่างสองชุด โดยการหารขนาดของการตัดกันด้วยขนาดของการรวมกัน ค่าจะอยู่ในช่วง 0 ถึง 1 ซึ่ง 1 หมายถึงชุดเหมือนกันและ 0 หมายถึงไม่มีการทับซ้อน
- สูตร:

$$\text{Jaccard Similarity} = \frac{|A \cap B|}{|A \cup B|}$$

โดยที่ $|A \cap B|$ คือจำนวนของสมาชิกในการตัดกันของชุด A และ B , และ $|A \cup B|$ คือจำนวนของสมาชิกในการรวมกันของชุด A และ B

- การใช้งาน:

ใช้ในการจัดกลุ่มและการดึงข้อมูลเพื่อวัดความคล้ายคลึงของชุดหรือคุณลักษณะไบนารี

- ตัวอย่าง:

- หากชุด $A = \{1, 2, 3\}$ และชุด $B = \{2, 3, 4\}$ ความคล้ายคลึงแบบแจ็กการ์ดคือ:

$$\frac{|\{2, 3\}|}{|\{1, 2, 3, 4\}|} = \frac{2}{4} = 0.5$$

4. Manhattan Distance (ระยะทางแมนฮัตตัน)

- ความหมาย:

- วัดระยะทางระหว่างสองจุดในเส้นทางที่อยู่ในรูปแบบตาราง โดยเป็นผลรวมของความแตกต่างของพิกัด

- สูตร:

$$\text{Manhattan Distance} = \sum_{i=1}^n |x_i - y_i|$$

โดยที่ x_i และ y_i คือพิกัดของจุดในพื้นที่ n มิติ

- การใช้งาน:

ใช้อัลกอริธึมการค้นหาเส้นทางในตารางและการจัดกลุ่มที่การเคลื่อนที่จำกัดอยู่ที่เส้นทางแนวตั้งและแนวนอน

- ตัวอย่าง:

- สำหรับสองจุด $(1, 2)$ และ $(4, 6)$ ในพื้นที่ 2 มิติ ระยะทางแมนฮัตตันคือ:

$$|4 - 1| + |6 - 2| = 3 + 4 = 7$$

