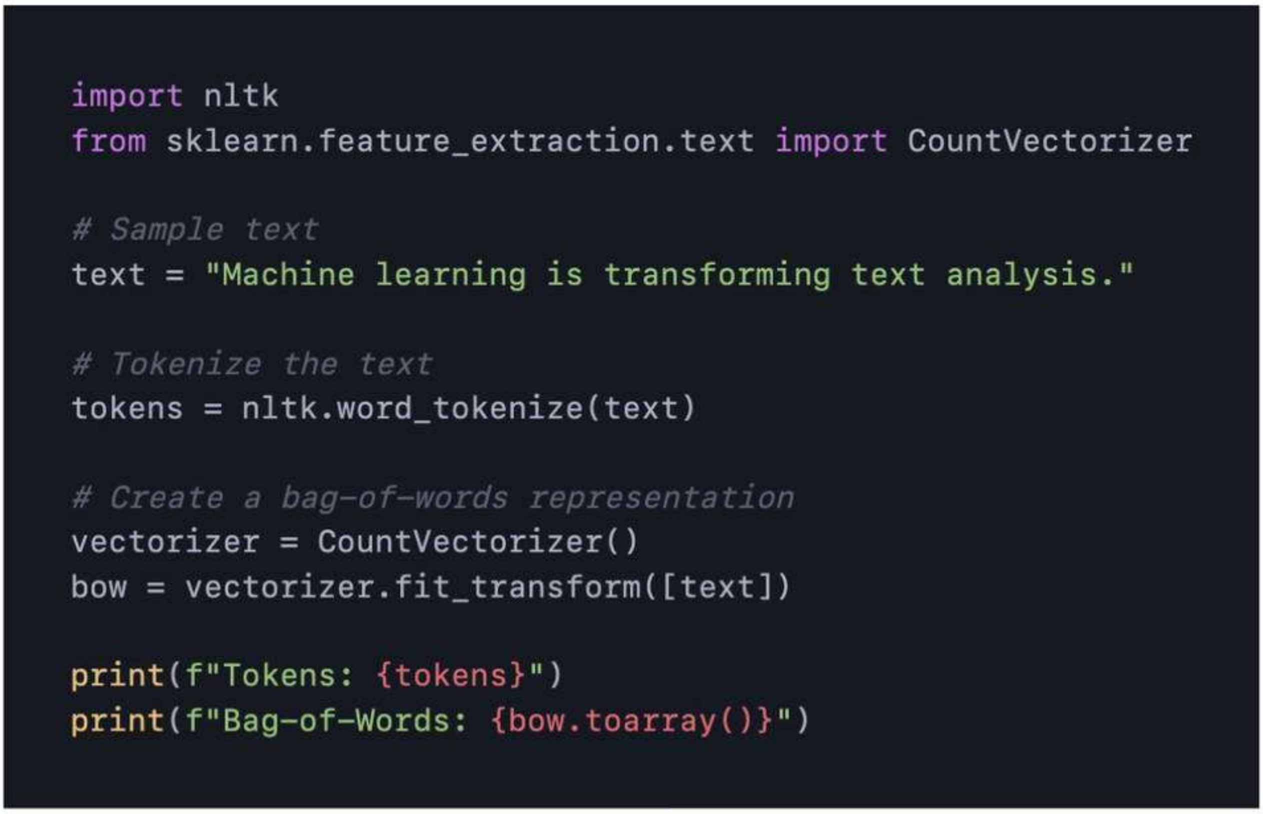
Common Machine Learning Techniques for Text Analysis in Python

With Code Examples

save for later

Text Analysis with Machine Learning in Python

Machine learning techniques have revolutionized text analysis, enabling computers to understand and process human language with remarkable accuracy. This presentation explores common ML methods for text analysis using Python, providing practical examples and code snippets to illustrate key concepts.

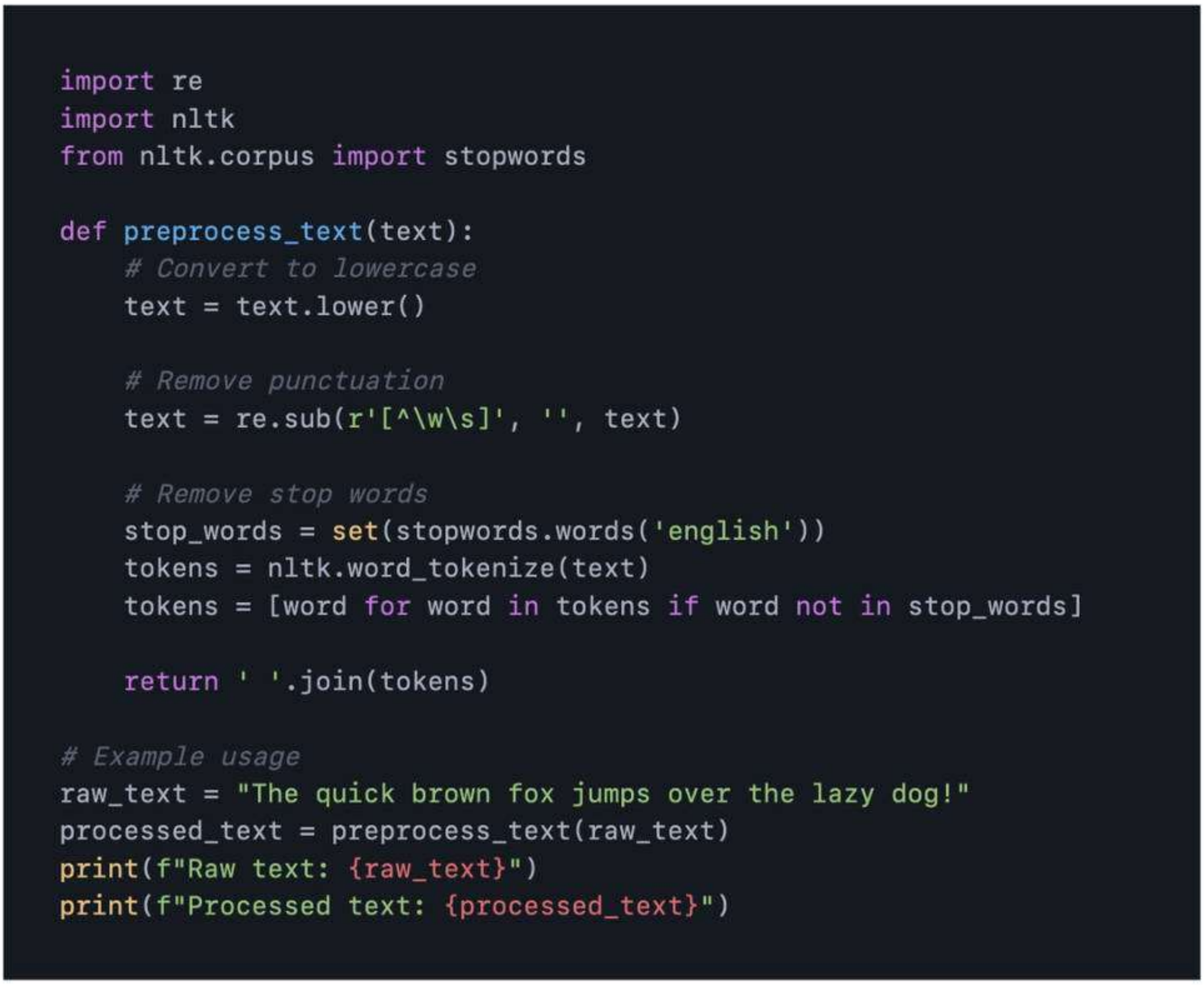


```python
import nltk
from sklearn.feature_extraction.text import CountVectorizer

# Sample text
text = "Machine learning is transforming text analysis."

# Tokenize the text
tokens = nltk.word_tokenize(text)

# Create a bag-of-words representation
vectorizer = CountVectorizer()
bow = vectorizer.fit_transform([text])

print(f"Tokens: {tokens}")
print(f"Bag-of-Words: {bow.toarray()}")
```

Swipe next ⟶

Text Preprocessing

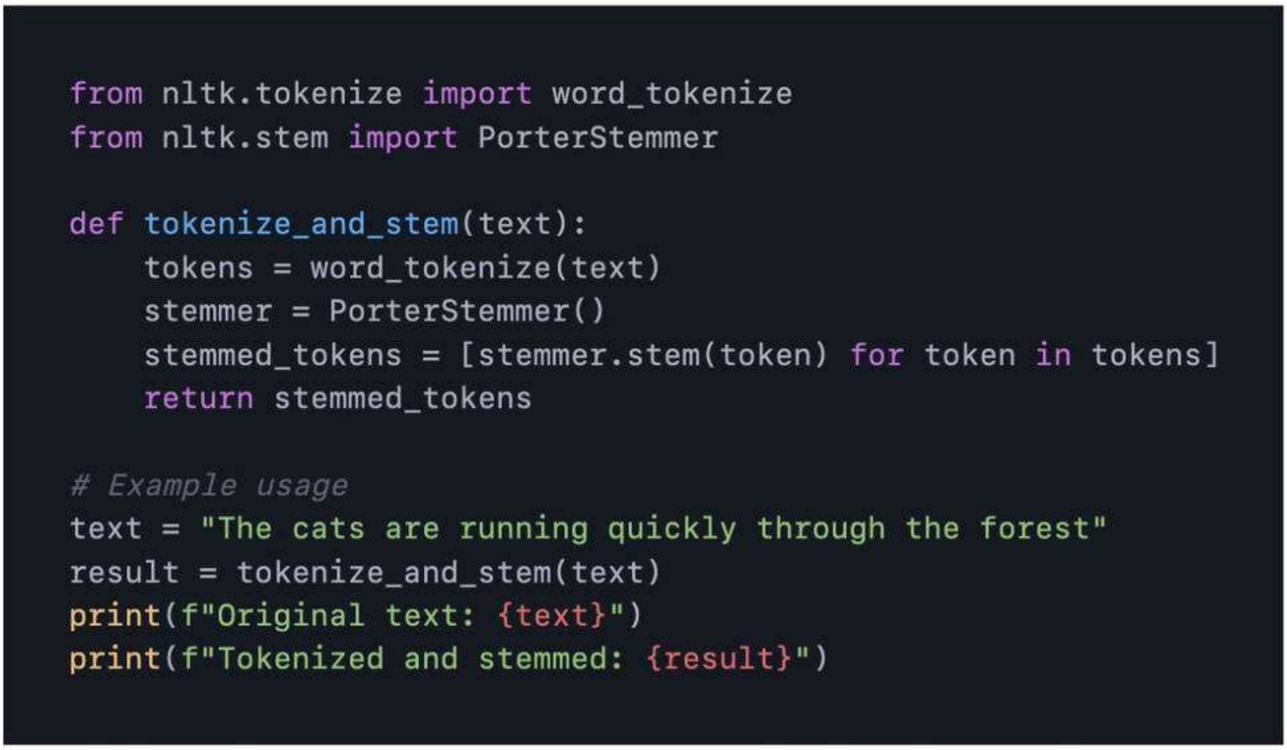
Before applying machine learning algorithms, it's crucial to preprocess the text data. This step involves cleaning and normalizing the text to improve the quality of analysis. Common preprocessing techniques include lowercasing, removing punctuation, and eliminating stop words.



```python
import re
import nltk
from nltk.corpus import stopwords

def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()

    # Remove punctuation
    text = re.sub(r'[^\w\s]', '', text)

    # Remove stop words
    stop_words = set(stopwords.words('english'))
    tokens = nltk.word_tokenize(text)
    tokens = [word for word in tokens if word not in stop_words]

    return ' '.join(tokens)

# Example usage
raw_text = "The quick brown fox jumps over the lazy dog!"
processed_text = preprocess_text(raw_text)
print(f"Raw text: {raw_text}")
print(f"Processed text: {processed_text}")
```

Tokenization and Stemming

Tokenization is the process of breaking text into individual words or subwords. Stemming reduces words to their root form, which can help in grouping similar words together. These techniques are fundamental in preparing text for further analysis.

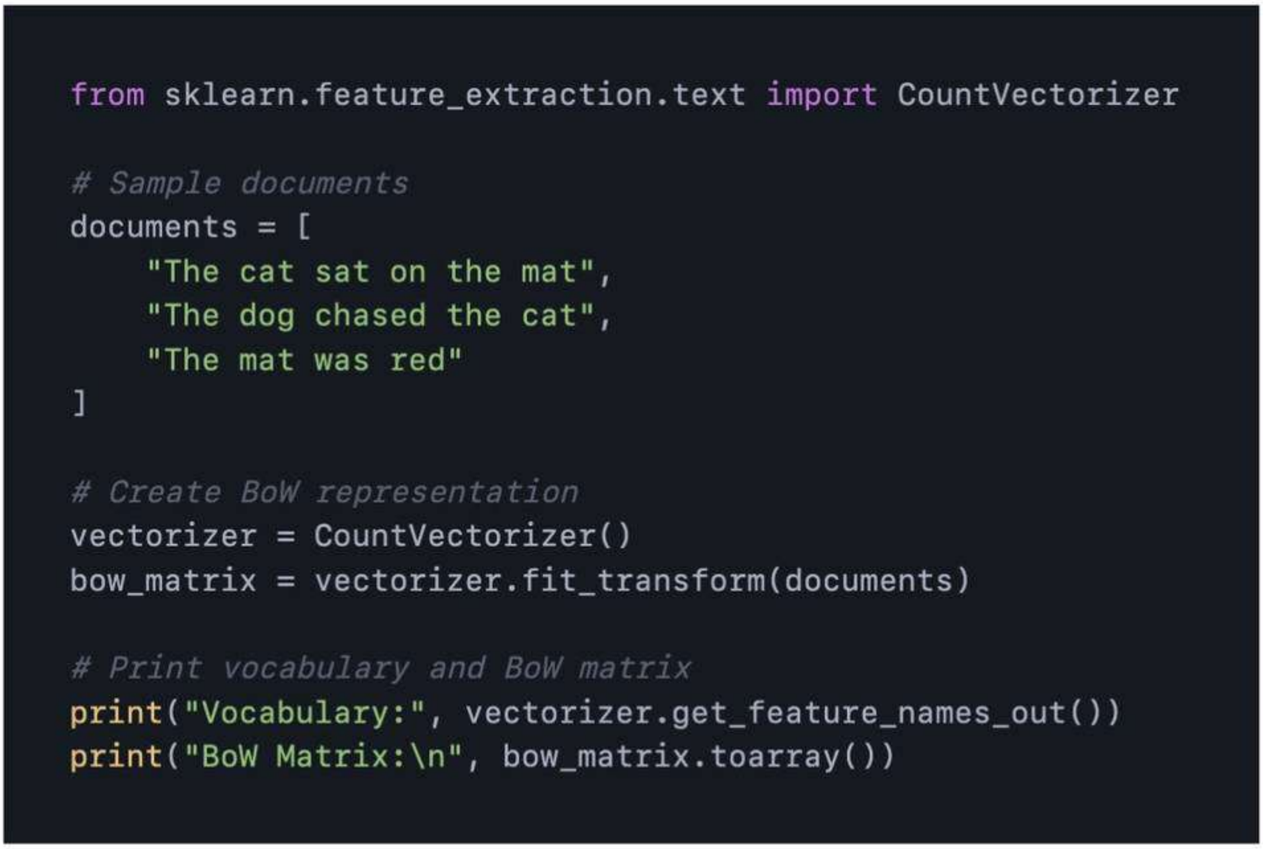


```python
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

def tokenize_and_stem(text):
    tokens = word_tokenize(text)
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(token) for token in tokens]
    return stemmed_tokens

# Example usage
text = "The cats are running quickly through the forest"
result = tokenize_and_stem(text)
print(f"Original text: {text}")
print(f"Tokenized and stemmed: {result}")
```

save for later

Bag of Words (BoW)

The Bag of Words model is a simple yet effective method for representing text data. It creates a vocabulary of unique words and represents each document as a vector of word frequencies. This approach is widely used in text classification and clustering tasks.

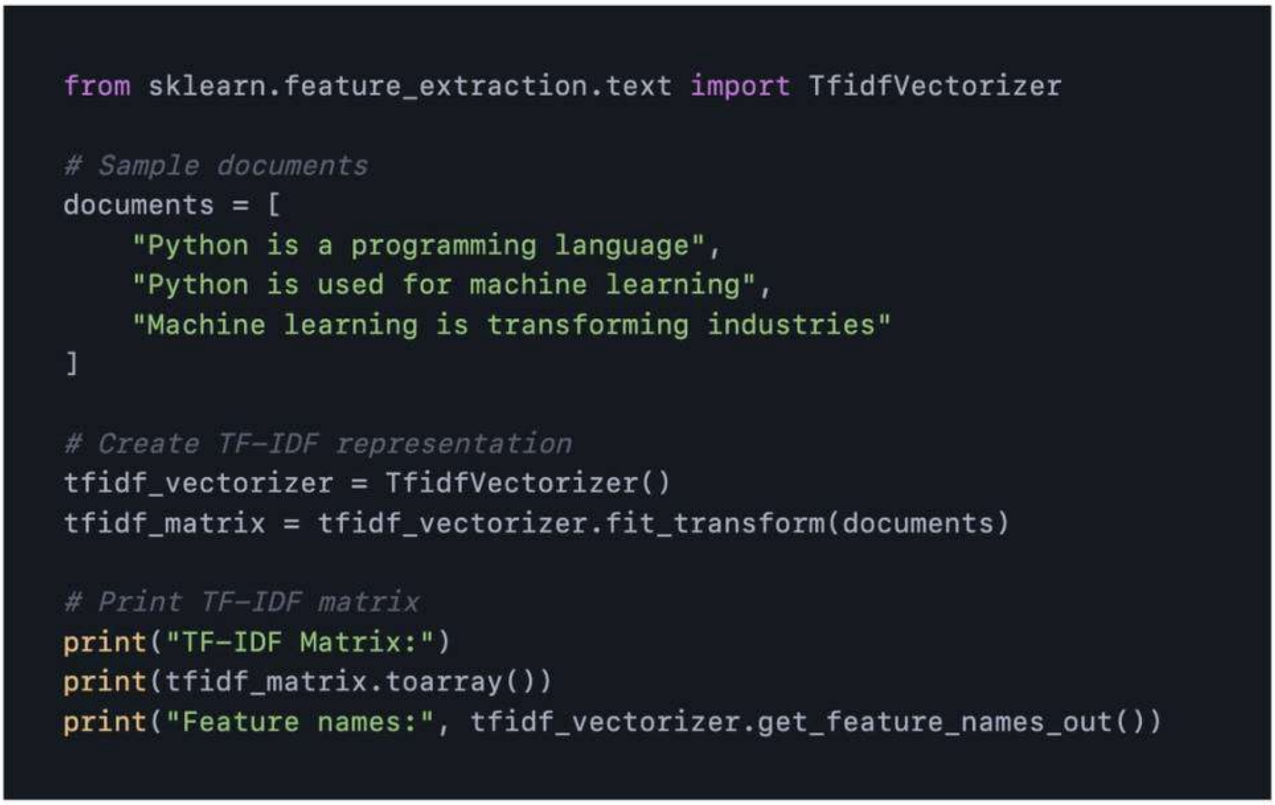


```python
from sklearn.feature_extraction.text import CountVectorizer

# Sample documents
documents = [
    "The cat sat on the mat",
    "The dog chased the cat",
    "The mat was red"
]

# Create BoW representation
vectorizer = CountVectorizer()
bow_matrix = vectorizer.fit_transform(documents)

# Print vocabulary and BoW matrix
print("Vocabulary:", vectorizer.get_feature_names_out())
print("BoW Matrix:\n", bow_matrix.toarray())
```

Swipe next ⟶

TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF is an advanced text representation technique that considers both the frequency of a word in a document and its importance across the entire corpus. It helps to identify words that are particularly characteristic of a document.

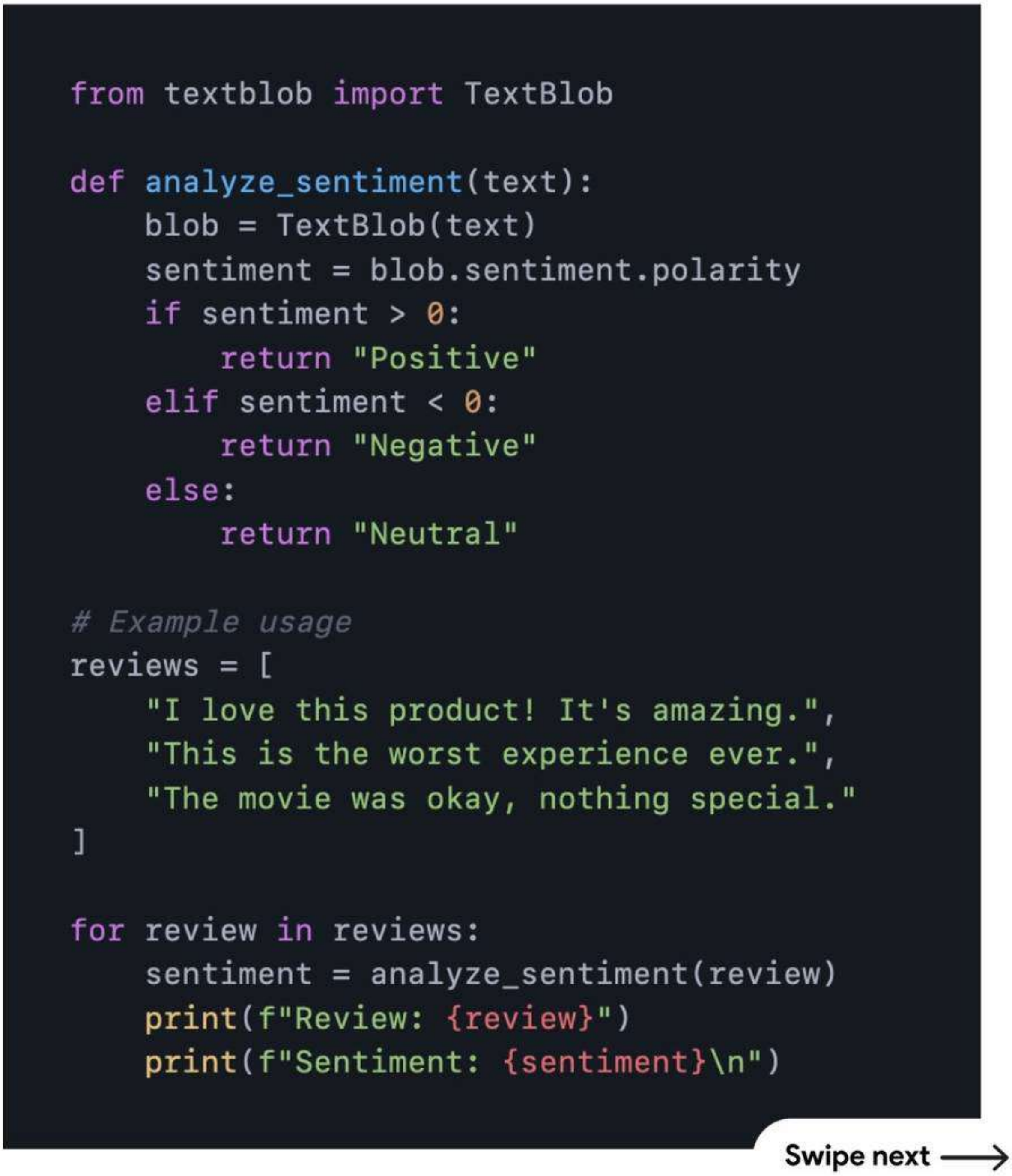


```python
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample documents
documents = [
    "Python is a programming language",
    "Python is used for machine learning",
    "Machine learning is transforming industries"
]

# Create TF-IDF representation
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)

# Print TF-IDF matrix
print("TF-IDF Matrix:")
print(tfidf_matrix.toarray())
print("Feature names:", tfidf_vectorizer.get_feature_names_out())
```

Swipe next ⟶

Sentiment Analysis

Sentiment analysis is the task of determining the emotional tone behind a piece of text. It's widely used in social media monitoring, customer feedback analysis, and market research. Here's a simple example using the TextBlob library.



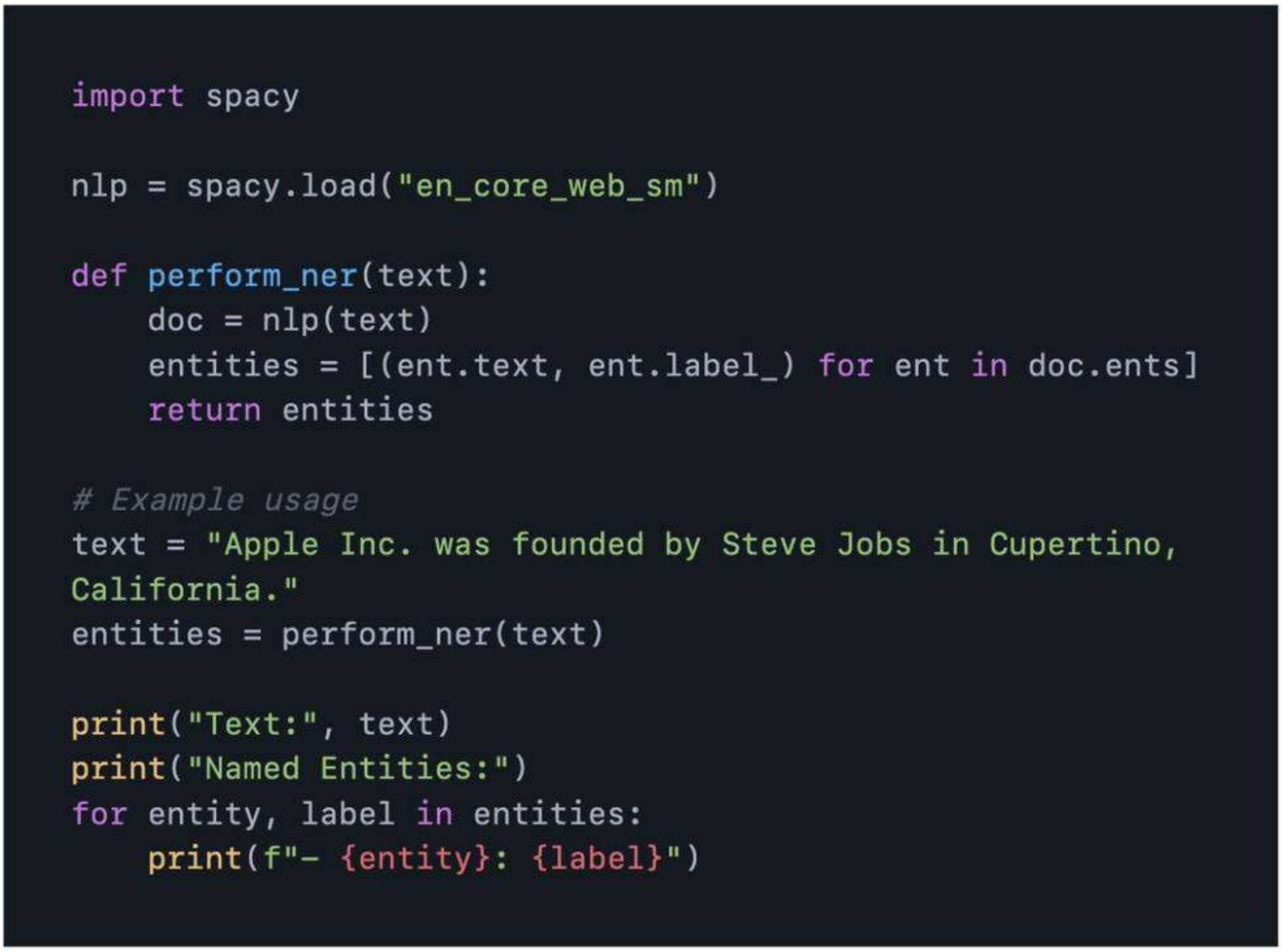
```python
from textblob import TextBlob

def analyze_sentiment(text):
    blob = TextBlob(text)
    sentiment = blob.sentiment.polarity
    if sentiment > 0:
        return "Positive"
    elif sentiment < 0:
        return "Negative"
    else:
        return "Neutral"

# Example usage
reviews = [
    "I love this product! It's amazing.",
    "This is the worst experience ever.",
    "The movie was okay, nothing special."
]

for review in reviews:
    sentiment = analyze_sentiment(review)
    print(f"Review: {review}")
    print(f"Sentiment: {sentiment}\n")
```

Swipe next ⟶

save for later

Named Entity Recognition (NER)

Named Entity Recognition is the task of identifying and classifying named entities (e.g., person names, organizations, locations) in text. It's crucial for information extraction and text understanding. Here's an example using spaCy.



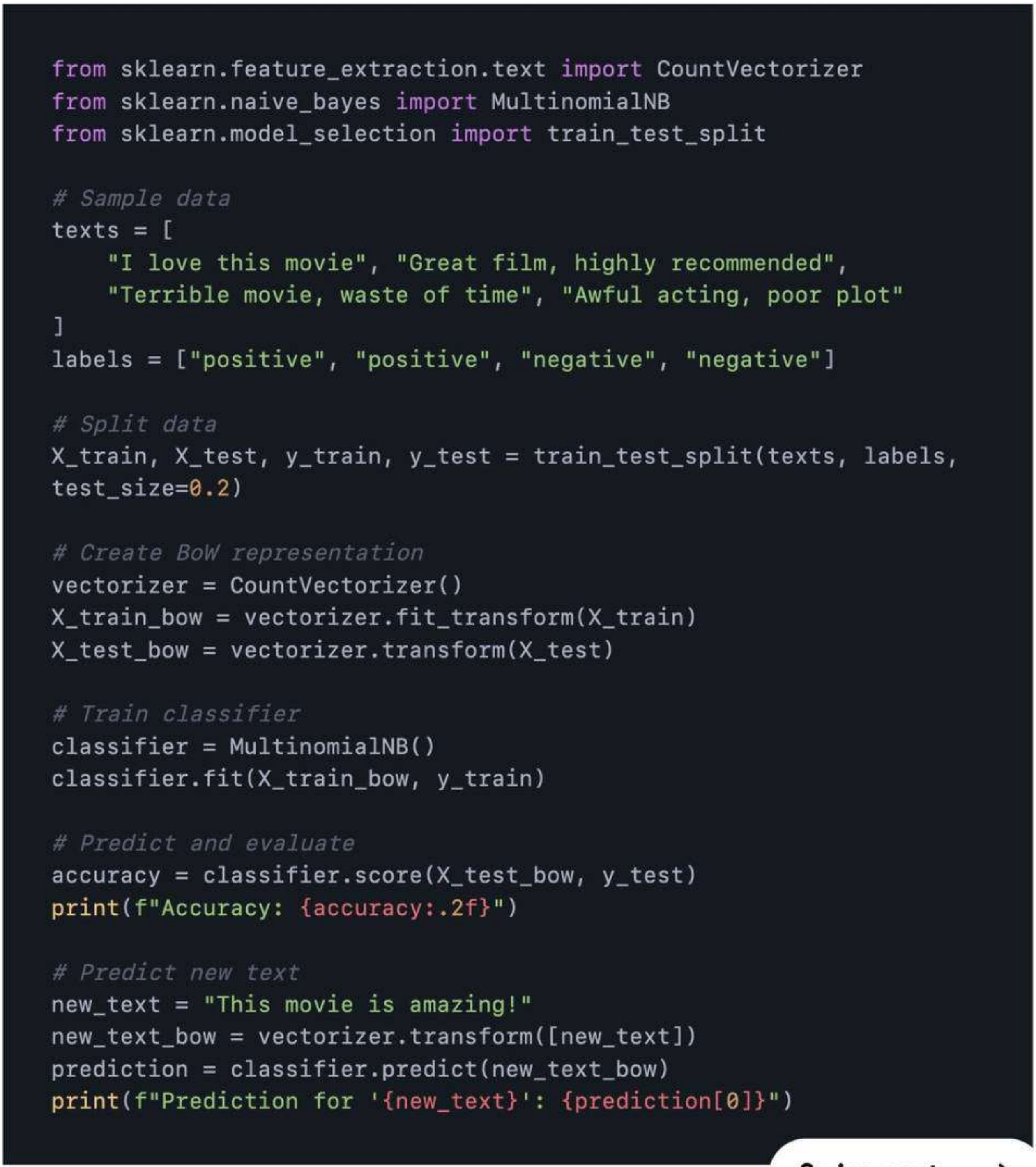
```python
import spacy

nlp = spacy.load("en_core_web_sm")

def perform_ner(text):
    doc = nlp(text)
    entities = [(ent.text, ent.label_) for ent in doc.ents]
    return entities

# Example usage
text = "Apple Inc. was founded by Steve Jobs in Cupertino,
California."
entities = perform_ner(text)

print("Text:", text)
print("Named Entities:")
for entity, label in entities:
    print(f"- {entity}: {label}")
```

Swipe next ⟶

Text Classification

follow for more

Text classification is the task of assigning predefined categories to text documents. It's used in spam detection, topic labeling, and sentiment analysis. Here's an example using Naive Bayes classifier for a simple text classification task.

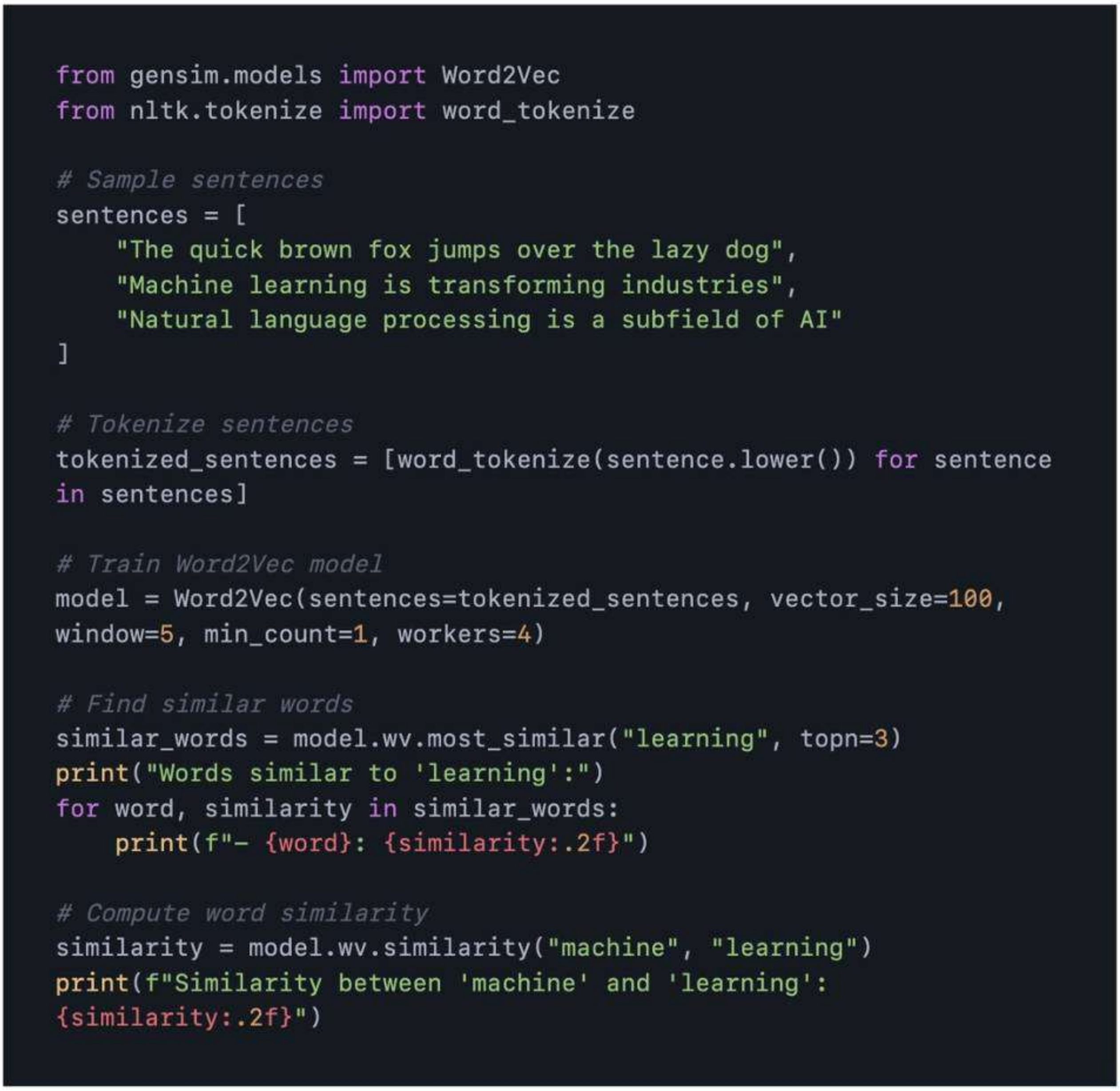


```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split

# Sample data
texts = [
    "I love this movie", "Great film, highly recommended",
    "Terrible movie, waste of time", "Awful acting, poor plot"
]
labels = ["positive", "positive", "negative", "negative"]

# Split data
X_train, X_test, y_train, y_test = train_test_split(texts, labels,
test_size=0.2)

# Create BoW representation
vectorizer = CountVectorizer()
X_train_bow = vectorizer.fit_transform(X_train)
X_test_bow = vectorizer.transform(X_test)

# Train classifier
classifier = MultinomialNB()
classifier.fit(X_train_bow, y_train)

# Predict and evaluate
accuracy = classifier.score(X_test_bow, y_test)
print(f"Accuracy: {accuracy:.2f}")

# Predict new text
new_text = "This movie is amazing!"
new_text_bow = vectorizer.transform([new_text])
prediction = classifier.predict(new_text_bow)
print(f"Prediction for '{new_text}': {prediction[0]}")
```

Swipe next →

Word Embeddings with Word2Vec

Word embeddings are dense vector representations of words that capture semantic relationships. Word2Vec is a popular method for creating word embeddings. Here's an example using the Gensim library to train a Word2Vec model.



```
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize

# Sample sentences
sentences = [
    "The quick brown fox jumps over the lazy dog",
    "Machine learning is transforming industries",
    "Natural language processing is a subfield of AI"
]

# Tokenize sentences
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence
in sentences]

# Train Word2Vec model
model = Word2Vec(sentences=tokenized_sentences, vector_size=100,
window=5, min_count=1, workers=4)

# Find similar words
similar_words = model.wv.most_similar("learning", topn=3)
print("Words similar to 'learning':")
for word, similarity in similar_words:
    print(f"- {word}: {similarity:.2f}")

# Compute word similarity
similarity = model.wv.similarity("machine", "learning")
print(f"Similarity between 'machine' and 'learning':
{similarity:.2f}")
```

Swipe next ⟶

save for later

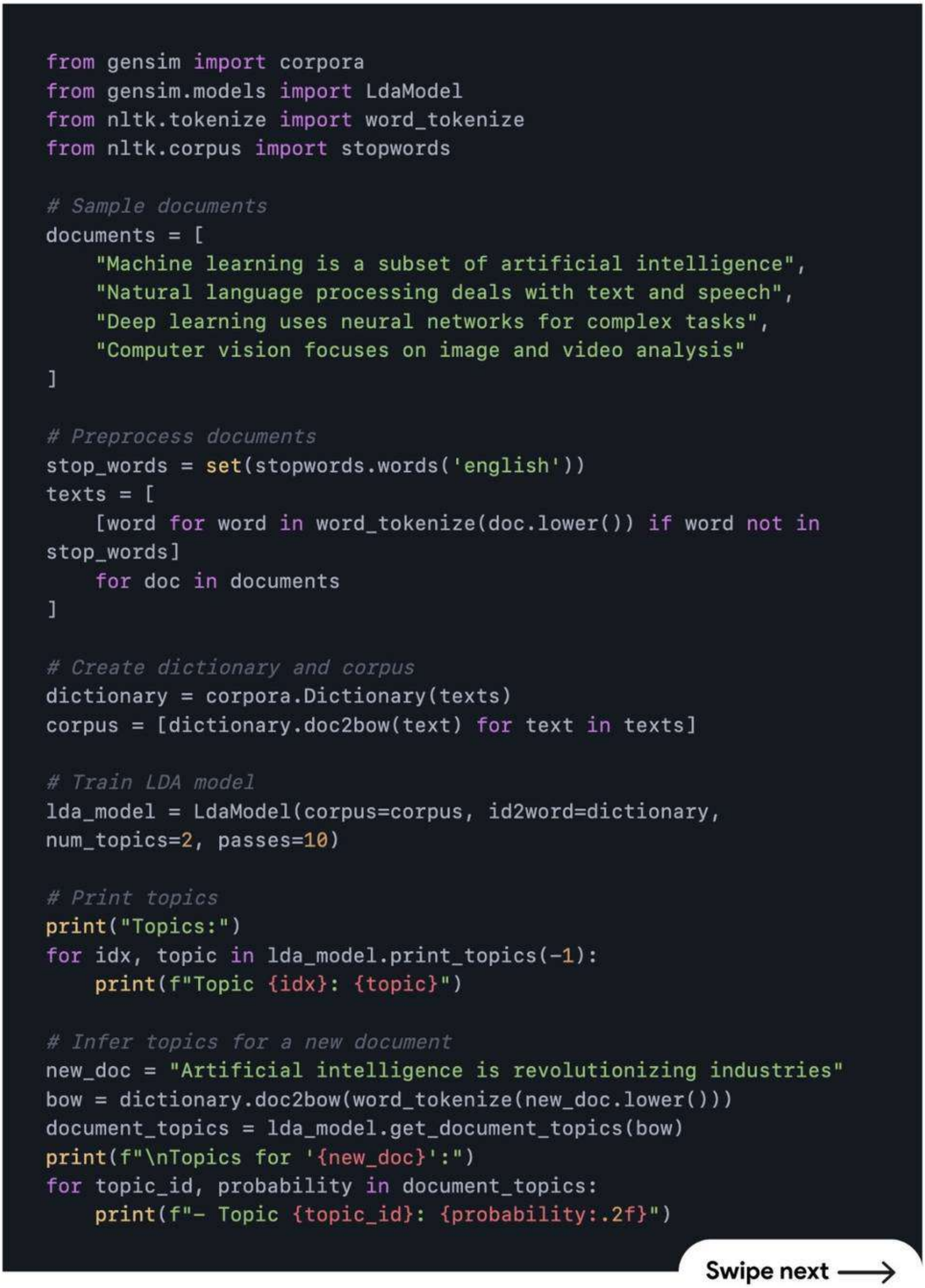
Topic Modeling with Latent Dirichlet Allocation (LDA)

Topic modeling is a technique for discovering abstract topics in a collection of documents. Latent Dirichlet Allocation (LDA) is a popular algorithm for topic modeling. Here's an example using the Gensim library.

Swipe next ⟶

Example

follow for more



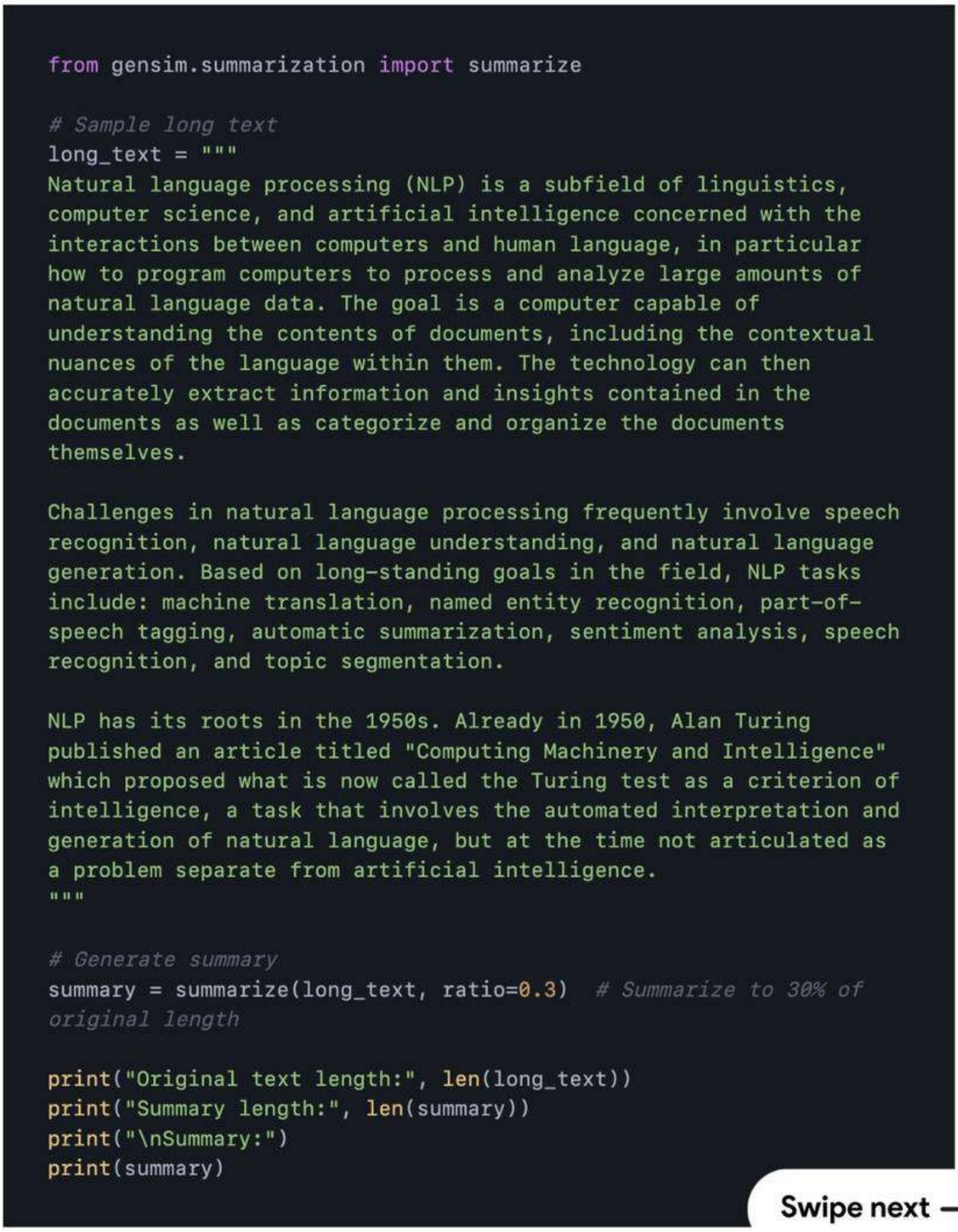
```python
from gensim import corpora
from gensim.models import LdaModel
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

# Sample documents
documents = [
    "Machine learning is a subset of artificial intelligence",
    "Natural language processing deals with text and speech",
    "Deep learning uses neural networks for complex tasks",
    "Computer vision focuses on image and video analysis"
]

# Preprocess documents
stop_words = set(stopwords.words('english'))
texts = [
    [word for word in word_tokenize(doc.lower()) if word not in
stop_words]
    for doc in documents
]

# Create dictionary and corpus
dictionary = corpora.Dictionary(texts)
corpus = [dictionary.doc2bow(text) for text in texts]

# Train LDA model
lda_model = LdaModel(corpus=corpus, id2word=dictionary,
num_topics=2, passes=10)

# Print topics
print("Topics:")
for idx, topic in lda_model.print_topics(-1):
    print(f"Topic {idx}: {topic}")

# Infer topics for a new document
new_doc = "Artificial intelligence is revolutionizing industries"
bow = dictionary.doc2bow(word_tokenize(new_doc.lower()))
document_topics = lda_model.get_document_topics(bow)
print(f"\nTopics for '{new_doc}':")
for topic_id, probability in document_topics:
    print(f"- Topic {topic_id}: {probability:.2f}")
```

Swipe next ⟶

save for later

Text Summarization

Text summarization is the process of creating a concise and coherent summary of a longer text while preserving its key information. Here's an example of extractive summarization using the TextRank algorithm implemented in the Gensim library.



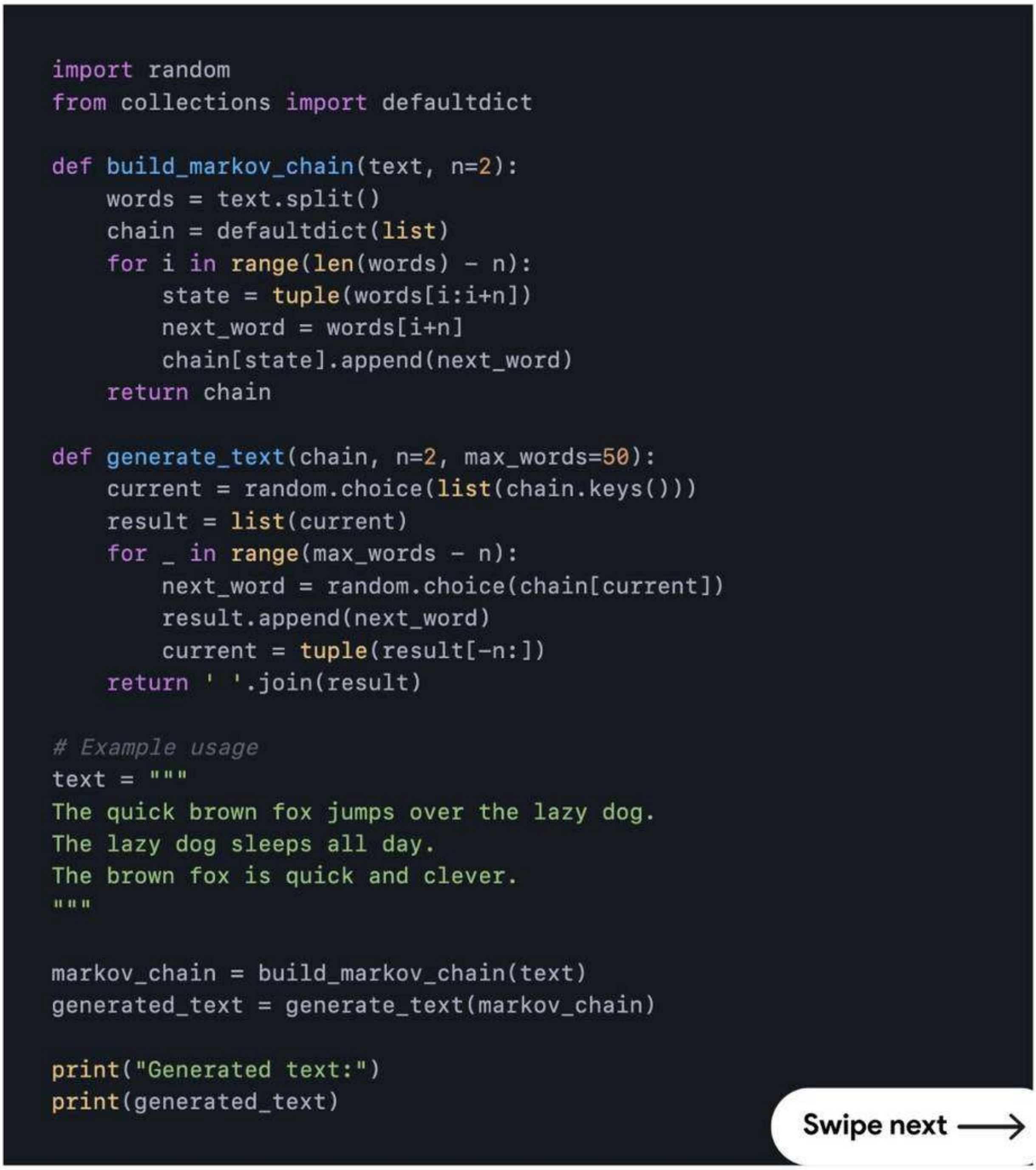
```
from gensim.summarization import summarize

# Sample long text
long_text = """
Natural language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with the
interactions between computers and human language, in particular
how to program computers to process and analyze large amounts of
natural language data. The goal is a computer capable of
understanding the contents of documents, including the contextual
nuances of the language within them. The technology can then
accurately extract information and insights contained in the
documents as well as categorize and organize the documents
themselves.

Challenges in natural language processing frequently involve speech
recognition, natural language understanding, and natural language
generation. Based on long-standing goals in the field, NLP tasks
include: machine translation, named entity recognition, part-of-
speech tagging, automatic summarization, sentiment analysis, speech
recognition, and topic segmentation.

NLP has its roots in the 1950s. Already in 1950, Alan Turing
published an article titled "Computing Machinery and Intelligence"
which proposed what is now called the Turing test as a criterion of
intelligence, a task that involves the automated interpretation and
generation of natural language, but at the time not articulated as
a problem separate from artificial intelligence.
"""

# Generate summary
summary = summarize(long_text, ratio=0.3)  # Summarize to 30% of
original length

print("Original text length:", len(long_text))
print("Summary length:", len(summary))
print("\nSummary:")
print(summary)
```

Swipe next →

Text Generation with Markov Chains

follow for more

Markov Chains can be used for simple text generation tasks. While not as sophisticated as modern deep learning approaches, they provide an intuitive introduction to probabilistic text generation. Here's a basic implementation:



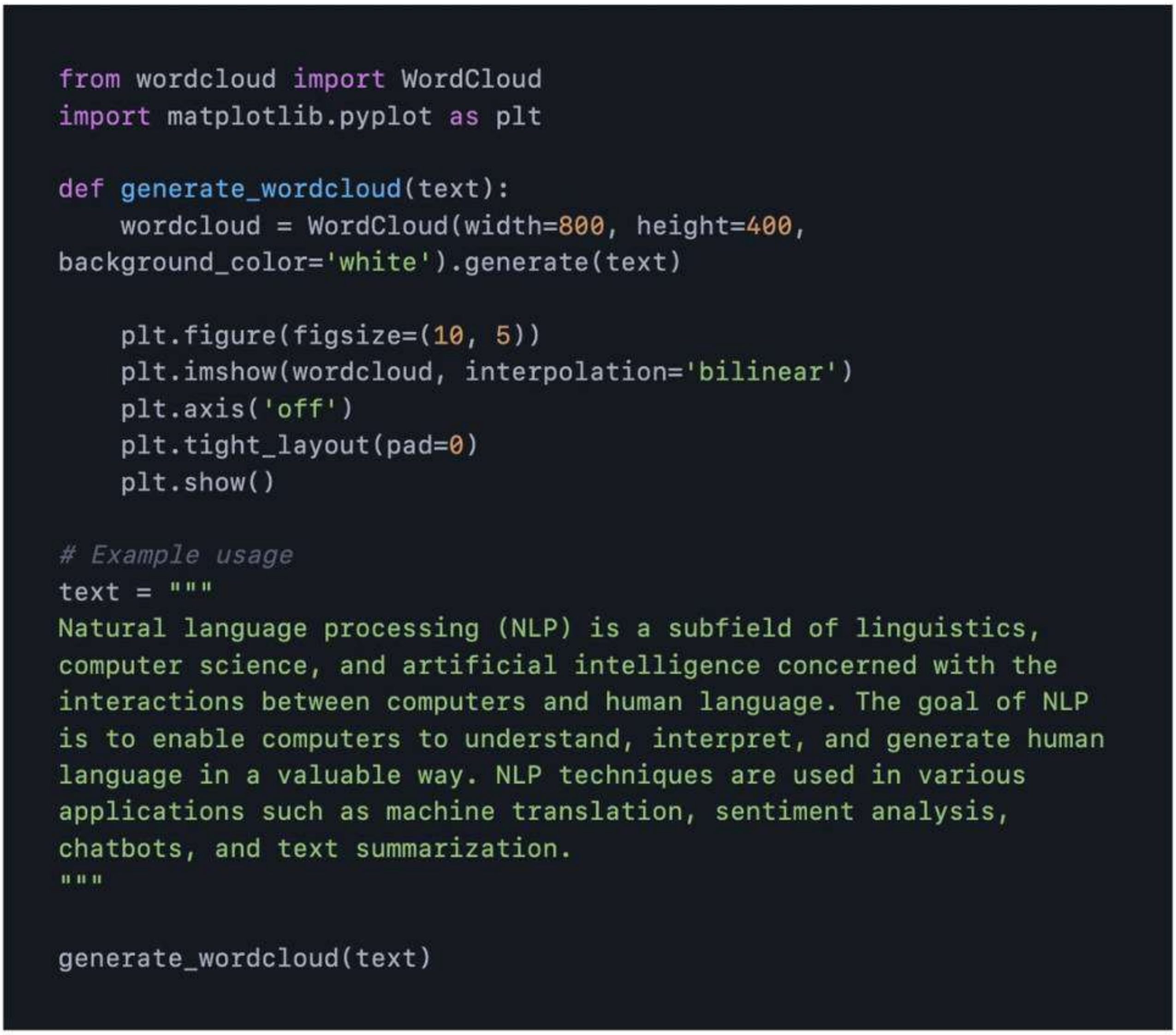
```python
import random
from collections import defaultdict

def build_markov_chain(text, n=2):
    words = text.split()
    chain = defaultdict(list)
    for i in range(len(words) - n):
        state = tuple(words[i:i+n])
        next_word = words[i+n]
        chain[state].append(next_word)
    return chain

def generate_text(chain, n=2, max_words=50):
    current = random.choice(list(chain.keys()))
    result = list(current)
    for _ in range(max_words - n):
        next_word = random.choice(chain[current])
        result.append(next_word)
        current = tuple(result[-n:])
    return ' '.join(result)

# Example usage
text = """
The quick brown fox jumps over the lazy dog.
The lazy dog sleeps all day.
The brown fox is quick and clever.
"""

markov_chain = build_markov_chain(text)
generated_text = generate_text(markov_chain)

print("Generated text:")
print(generated_text)
```

Swipe next ⟶

follow for more

Word Cloud Generation

Word clouds are visual representations of text data where the size of each word indicates its frequency or importance. They're useful for quickly perceiving the most prominent terms in a text corpus. Here's how to create a word cloud using the WordCloud library:



```python
from wordcloud import WordCloud
import matplotlib.pyplot as plt

def generate_wordcloud(text):
    wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(text)

    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    plt.tight_layout(pad=0)
    plt.show()

# Example usage
text = """
Natural language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with the
interactions between computers and human language. The goal of NLP
is to enable computers to understand, interpret, and generate human
language in a valuable way. NLP techniques are used in various
applications such as machine translation, sentiment analysis,
chatbots, and text summarization.
"""

generate_wordcloud(text)
```

Swipe next ⟶

Additional Resources

For those interested in diving deeper into machine learning techniques for text analysis, here are some valuable resources:

1. "Neural Network Methods for Natural Language Processing" by Yoav Goldberg (2017) ArXiv: https://arxiv.org/abs/1703.02620
2. "A Survey of Deep Learning Techniques for Natural Language Processing" by Diksha Khurana et al. (2020) ArXiv: https://arxiv.org/abs/2101.00468
3. "Efficient Estimation of Word Representations in Vector Space" by Tomas Mikolov et al. (2013) ArXiv: https://arxiv.org/abs/1301.3781