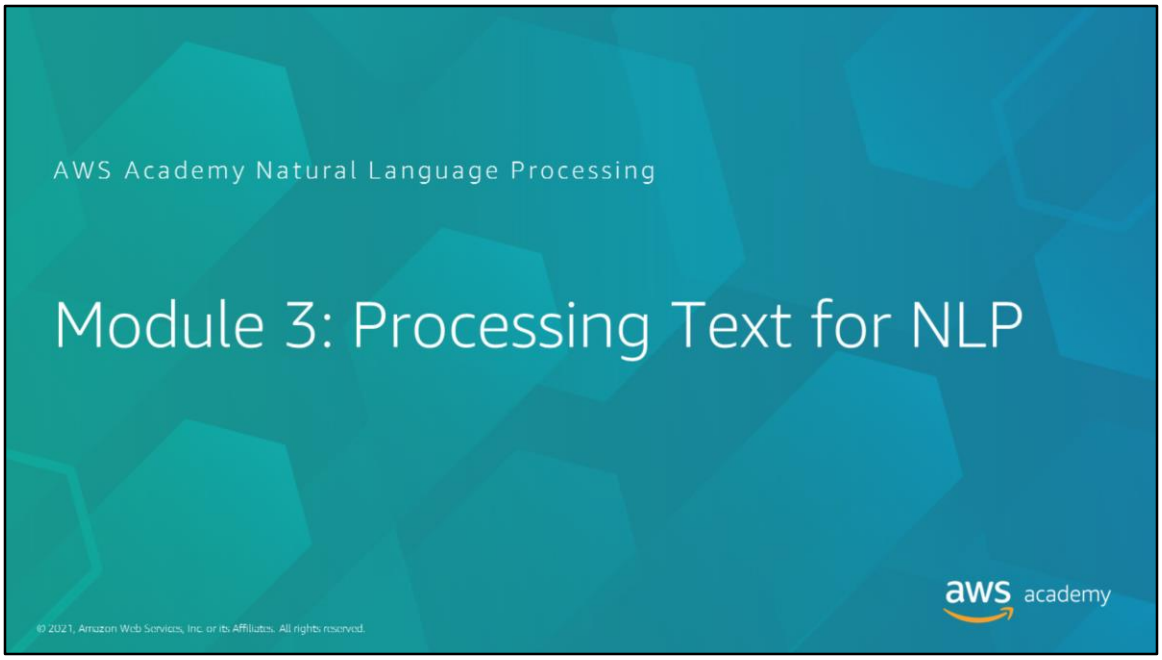
AWS Academy Natural Language Processing
Module 03 Student Guide
Version 0.1.0

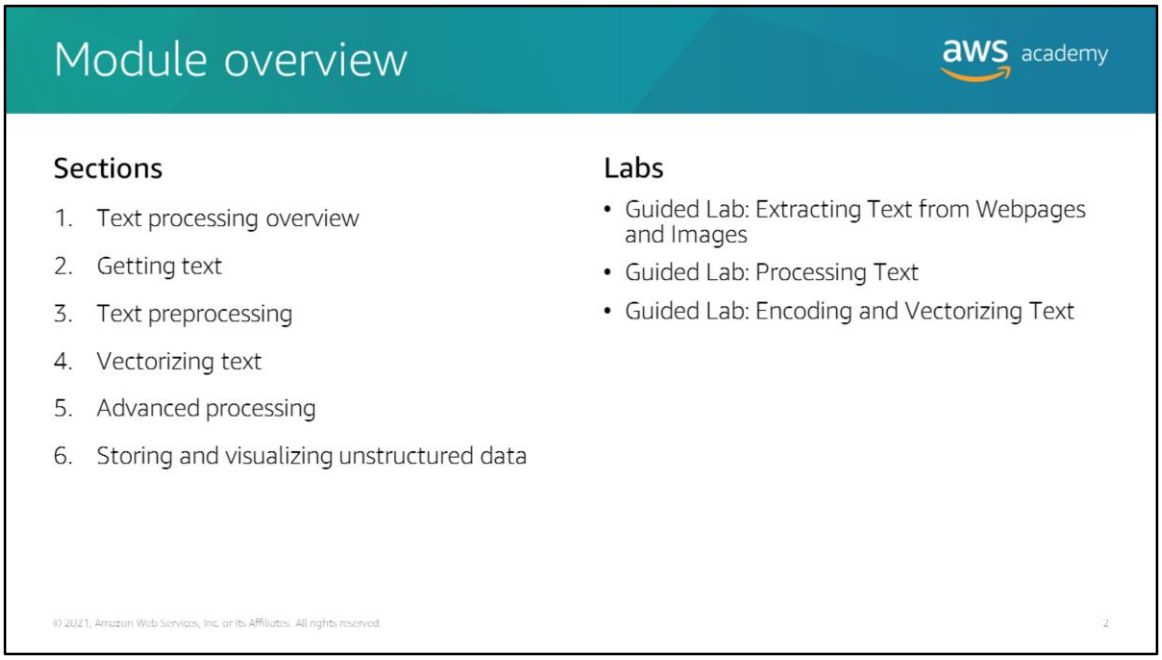
200-ACMNLP-01-EN-SG

AWS Academy Natural Language Processing

Contents



Welcome to Module 3: Processing Text for NLP.



In this module, you will learn about the following:

1. Text processing overview – In this section, you will look at the major steps to take in processing text for your NLP application.

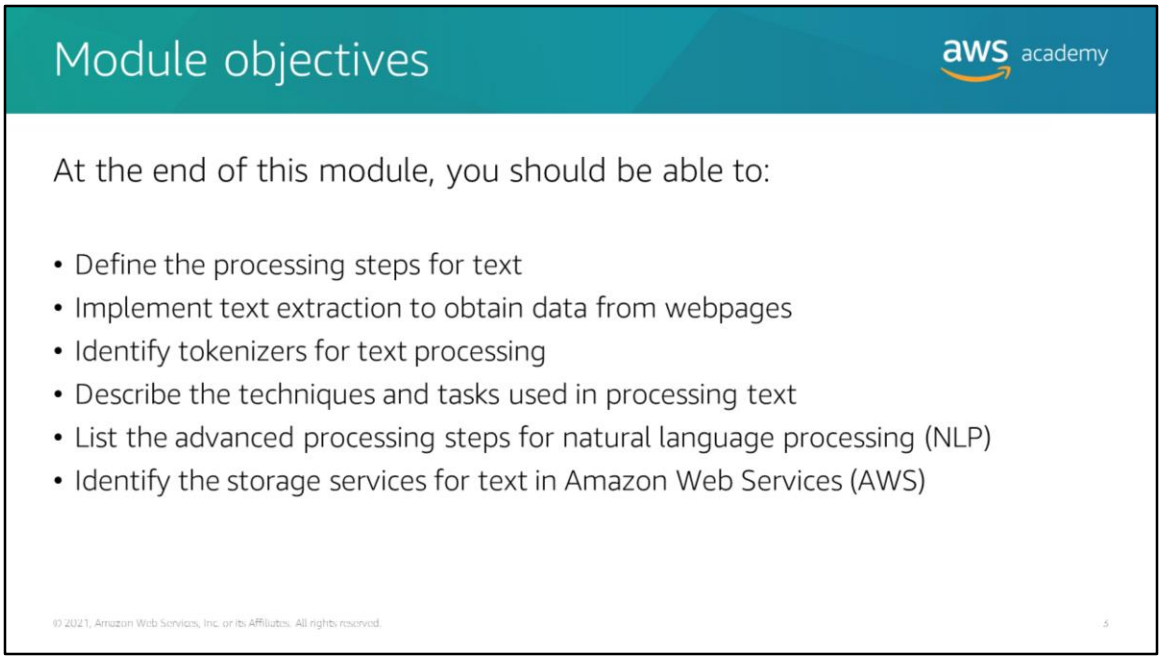
2. Getting text – This section describes how to extract text from various sources.

3. Text preprocessing – In this section, you will learn about some of the most common ways to process text.

4. Vectorizing text – In this section, you will examine several of the more common models for converting text into numbers.

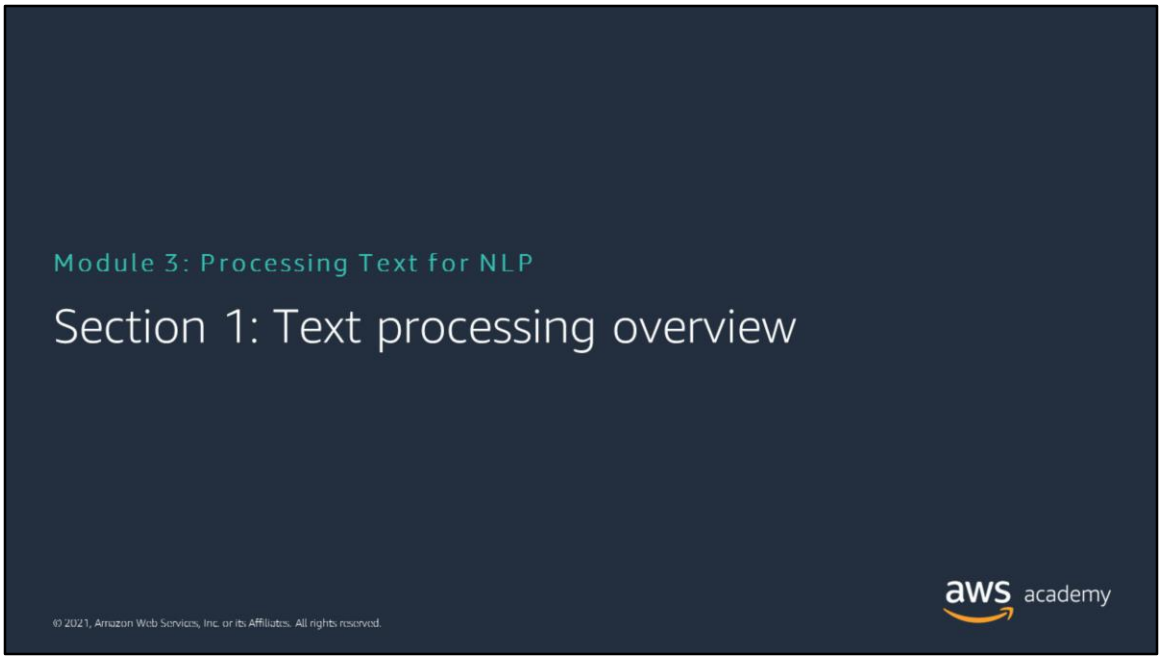
5. Advanced processing – This section explores a few more advanced ways to process text.

6. Storing and visualizing unstructured data – This section describes the differences between structured and unstructured data. It also describes the features of three AWS services that you can use to store text data for your NLP application.

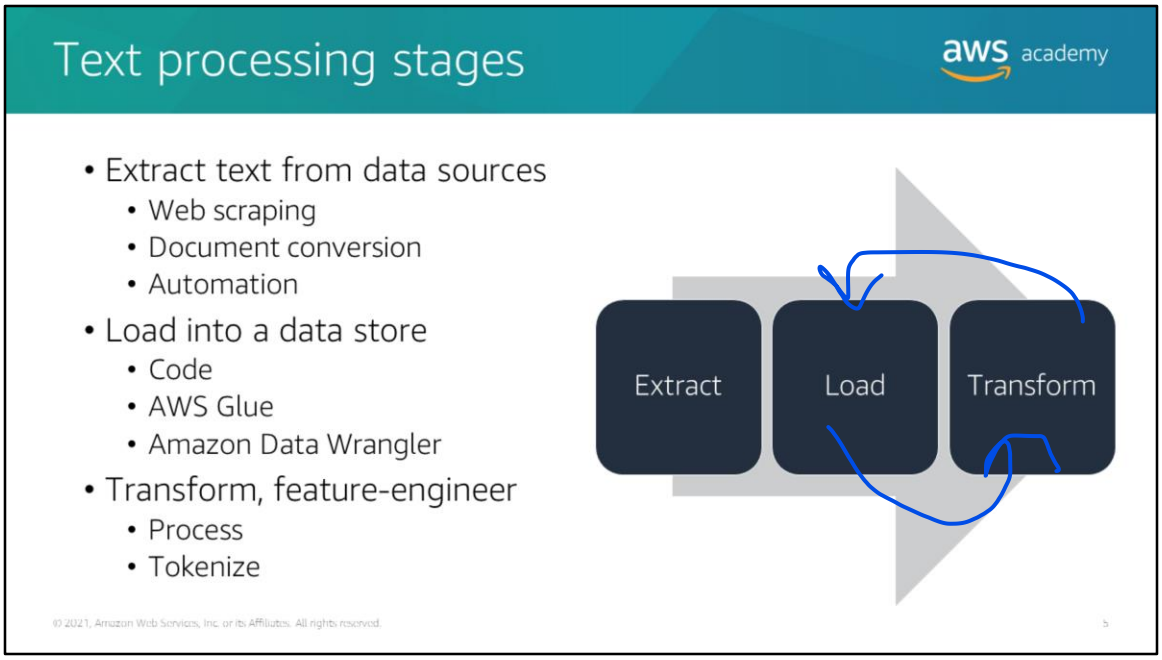


At the end of this module, you should be able to:

- Define the processing steps for text
- Implement text extraction to obtain data from webpages
- Identify tokenizers for text processing
- Describe the techniques and tasks that are used in processing text
- List the advanced processing steps for natural language processing (NLP)
- Identify the storage services for text in Amazon Web Services (AWS)

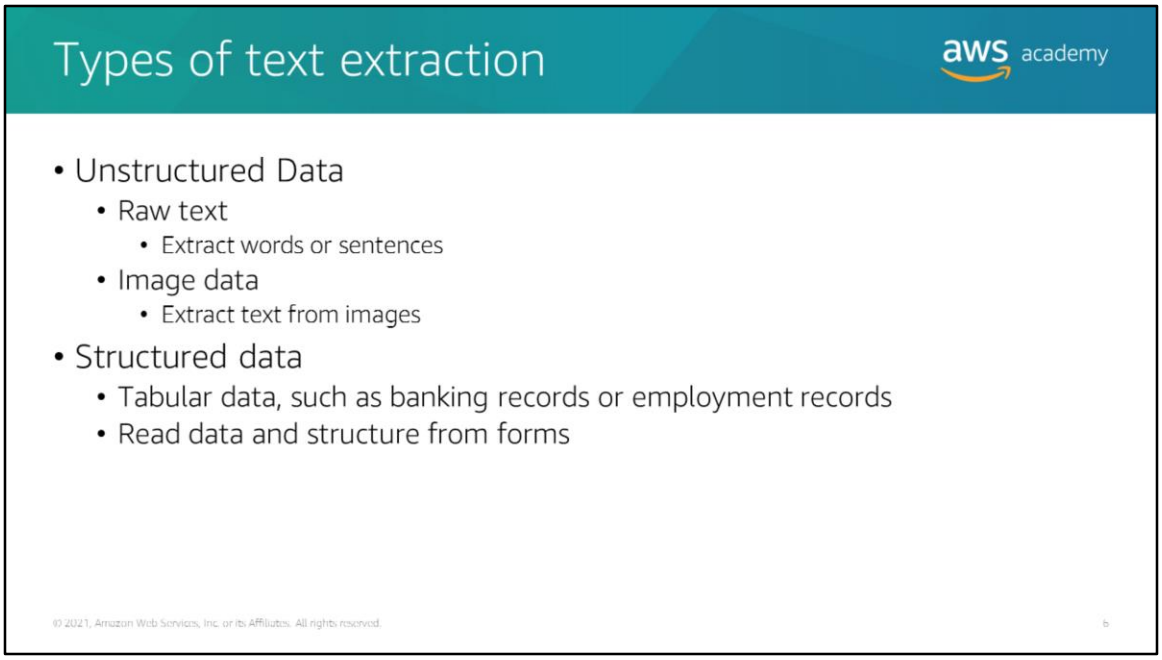


Introducing Section 1: Text processing overview.

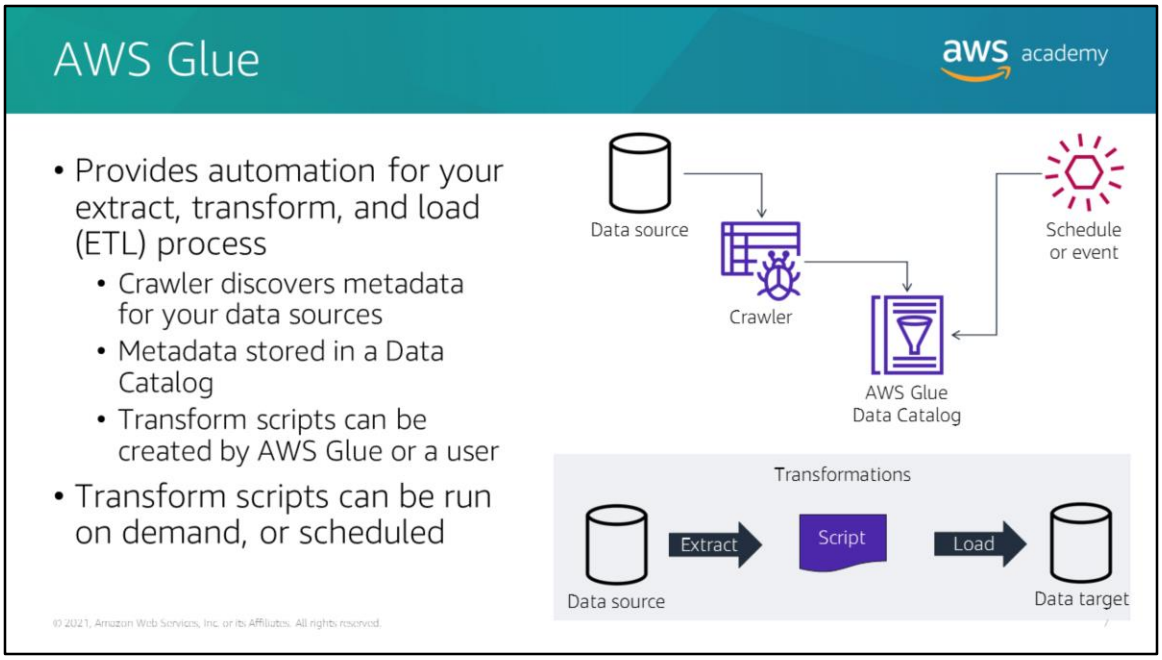


As you saw in the previous module, the first stage of your NLP application is to load and process text. You can think of this stage as consisting of three substeps:

- First, you must extract data from the data sources. For example, you can pull text from websites or other web resources. This process is known as web scraping. If you load data from documents, you must convert it into a form that your loading component requires. For most real-world applications, you will want to automate the extraction process.
- After you extract the text, you load it into the transform pipeline. In this module, you will learn how to do this process by using Python libraries, but you can also automate the process by using Amazon Textract.
- Finally, you must transform the text into numeric representation for use by your machine learning (ML) model.

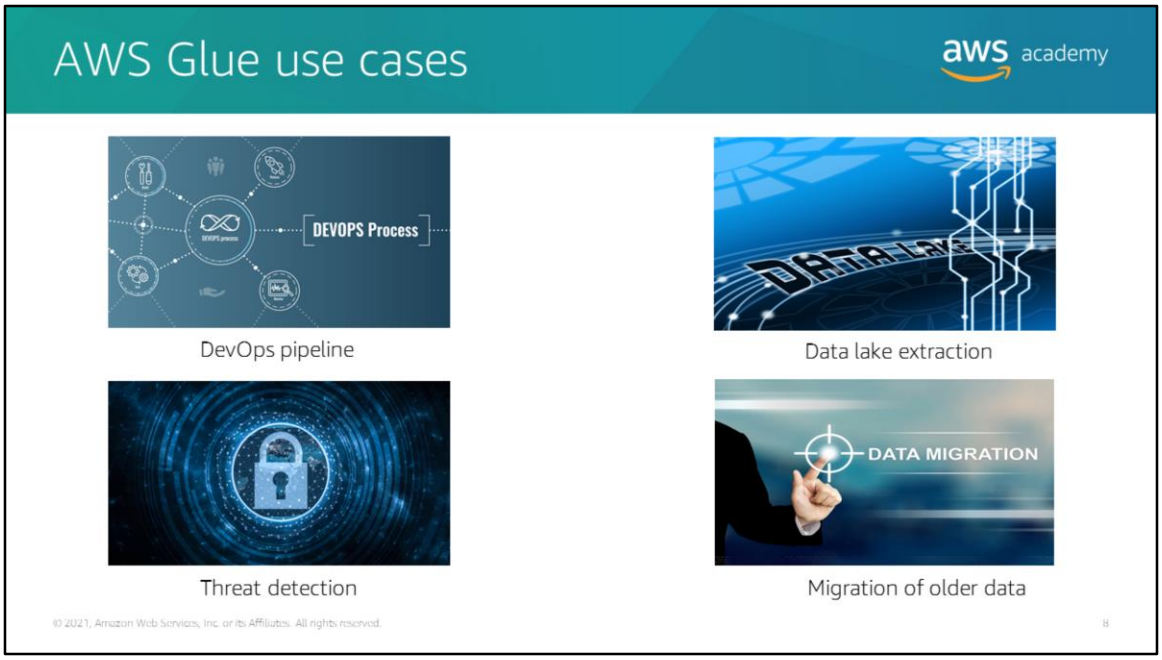


You can extract individual words, sentences, or entire documents of text. You can also extract text data from images. For example, a media company might want to find all images that contain the name of a particular brand or product. Another common use case is to extract data from forms or other types of structured data. For example, a human resources department might want to find all employee records that contain a particular set of skills or experiences.



You can use AWS Glue to automate your ETL processes. The diagram illustrates the major components for an ETL system that is based on AWS Glue.

- First, you define a *crawler*, which discovers metadata based on your data. AWS Glue then stores this metadata in a *Data Catalog*.
- Next, use the script that AWS Glue creates for automating the ETL process. You can also create your own script.
- When you have your data transformation script, you can either run it on demand, or you can create a schedule for running it. The script extracts data from your data source, transforms it, and loads it to the data target.

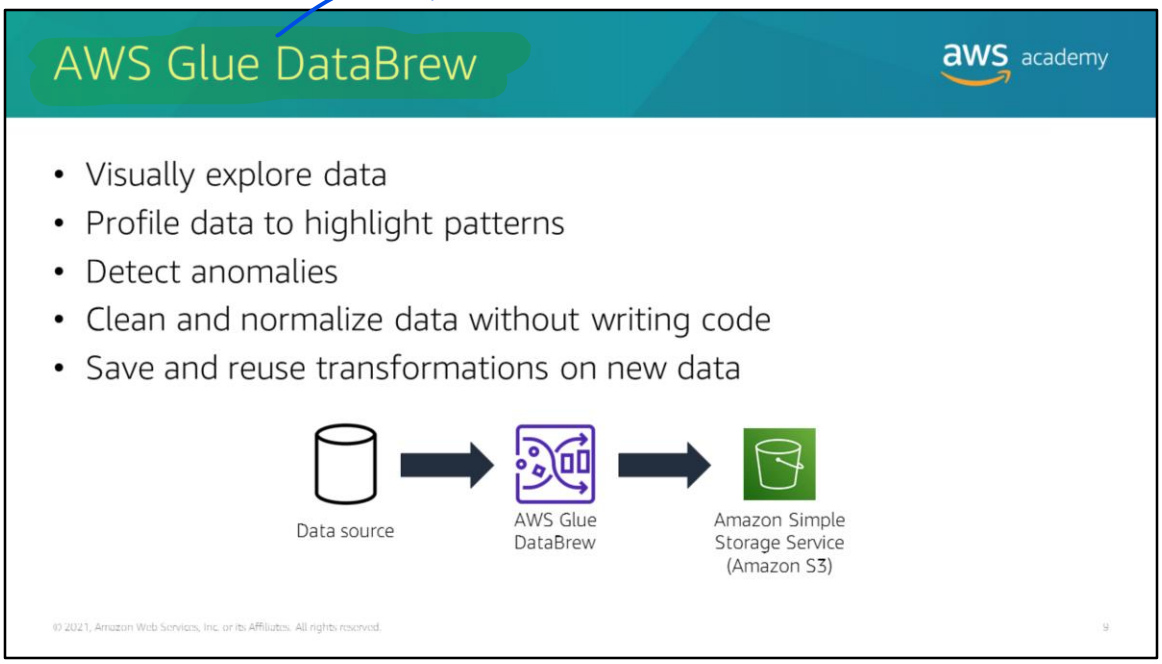


You can use AWS Glue for various ETL jobs. Some common examples are listed here, but they represent only a small sample.

For example, you can:

- Build a DevOps pipeline to automate the troubleshooting and analysis of customer logs for a web-hosting service
- Automate extracting information from a data lake to inform analytics for online retail
- Extract data from legacy data stores to forecast future sales
- Discover security threats by analyzing data breaches and web traffic

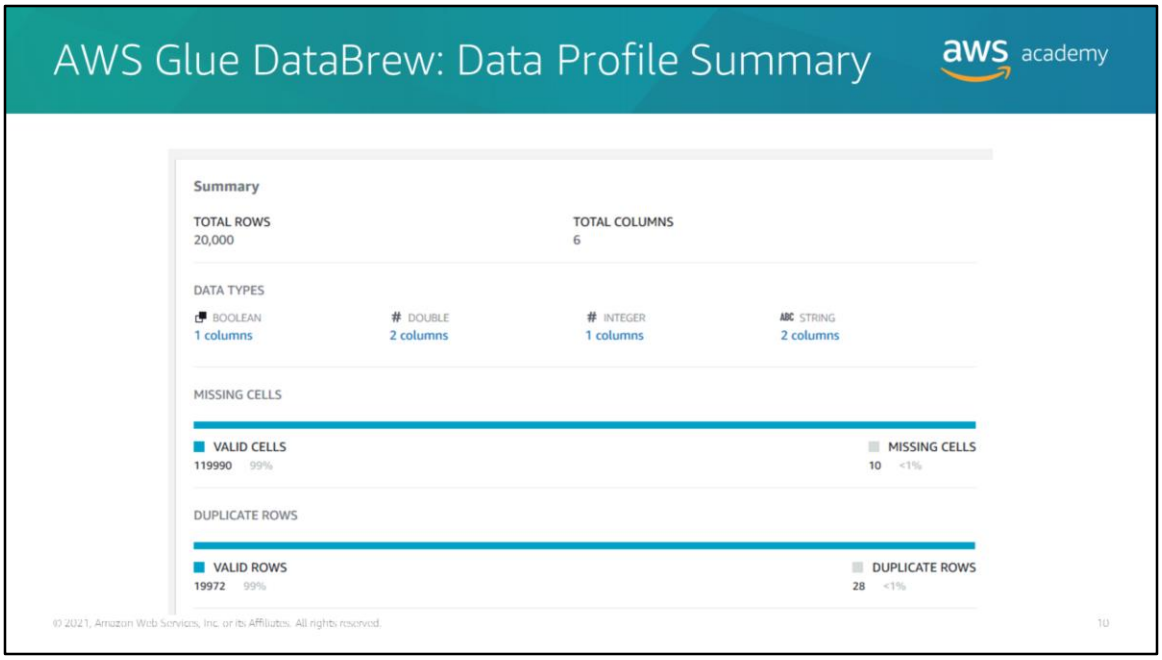
can do all the ADA tools and steps to any dataset using GUI



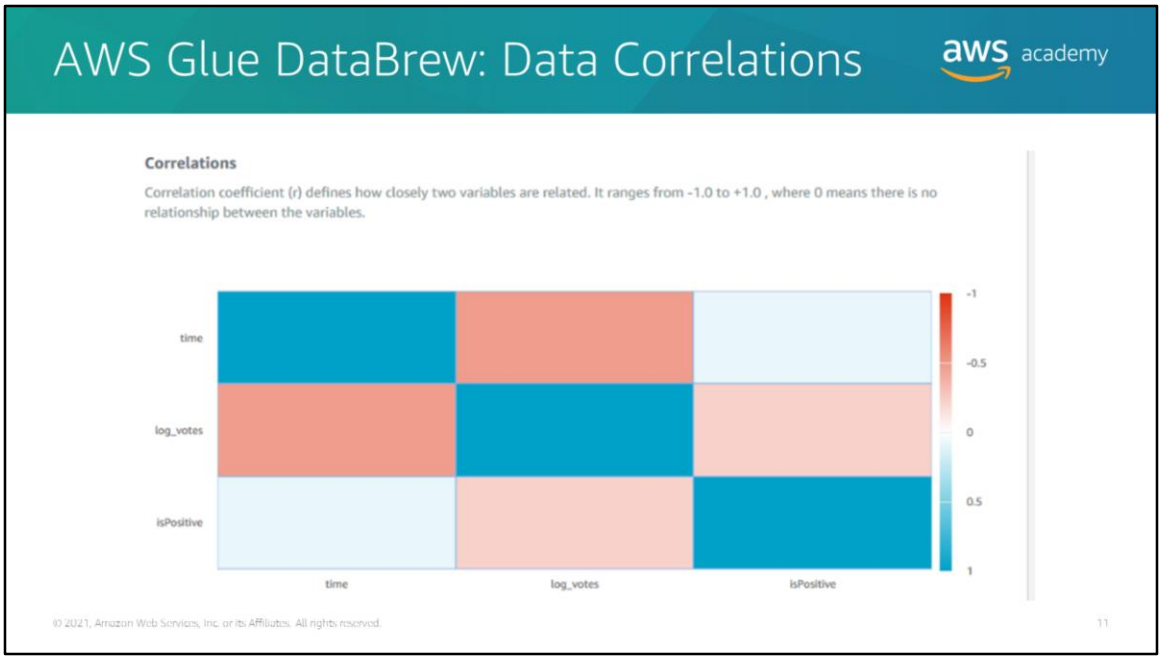
AWS Glue DataBrew is a fully managed AWS service you can use to visually prepare, clean, and normalize data without writing any code. With over 250 built-in transformations to automate data preparation tasks, this service can save time when you prepare data for ML. Such tasks might include filtering anomalies, converting data to standard formats, and correcting invalid values.

You can use AWS Glue DataBrew features to do these tasks:

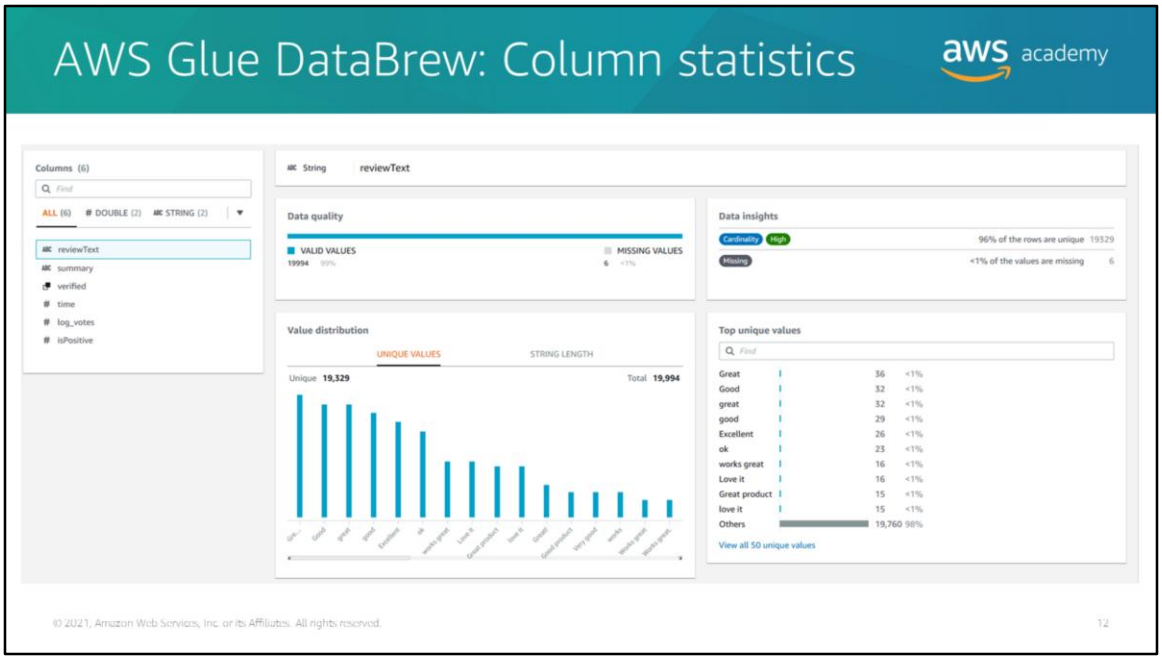
- **Explore data in columns with rich statistics –** You can find anomalies and patterns in the data for each column with rich visualizations.
- **Clean and normalize data by using over 250 built-in transformations –** You can remove or replace null values, standardize date-time columns to conform to a standard, and create encodings. You can merge, pivot, split, or join data, You can also create aggregates and work with representative samples of the data to determine the correct transformations to use.
- **Build and experiment with recipes by using one or more transformation steps –** Each transformation is added as a step to build recipes. You can save, publish, and version recipes to share with other collaborators.
- **Run jobs to apply the recipe on all incoming data –** You can run a job to apply the recipe to larger datasets. You can specify data formats, compression modes, and partition keys as you output prepared data to Amazon Simple Storage Service (Amazon S3).



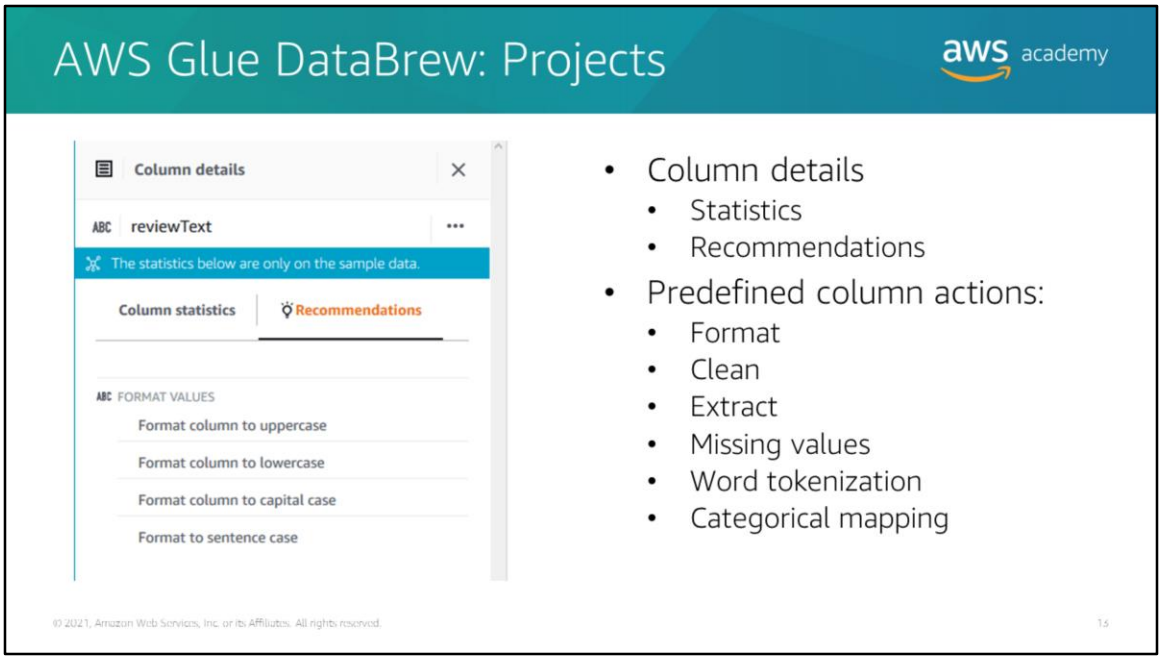
When you import data into AWS Glue DataBrew, you can view key information about your dataset, including missing cells and duplicate rows.



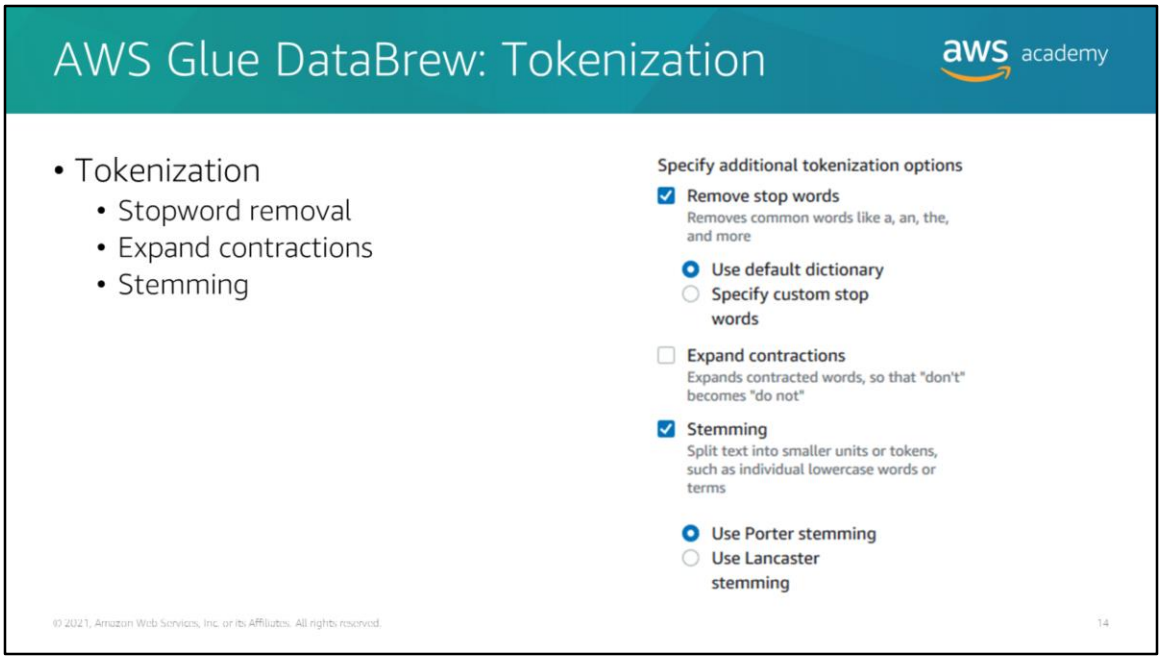
DataBrew also gives you a quick data correlation chart so that you can see key relationships between features.



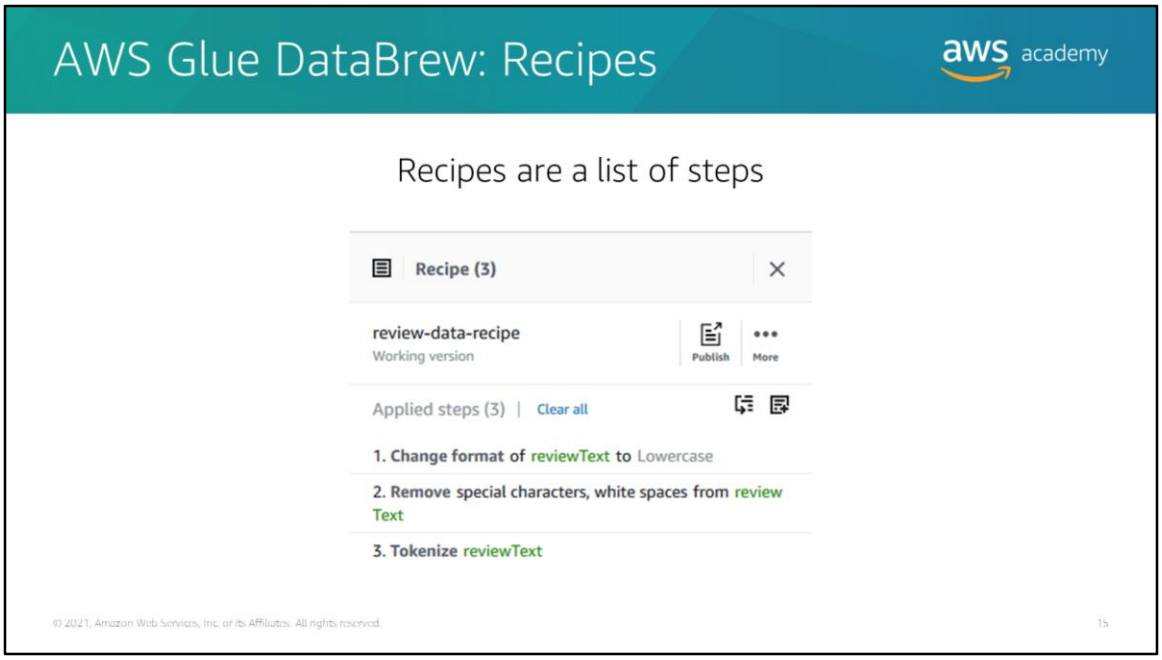
You can view all the common statistics on column data by examining the columns. For text data, this information includes top unique values and value-distribution data.



You can create a project after you import the data. You can use a project to define a set of recipes, which are actions that you can perform on the data. DataBrew also provides some recommendations that are based on the column's data type. For processing text, you can select some of the common processing tasks that you will learn about in this module. These tasks might include formatting, cleaning, extracting, handling missing values, word tokenization, and categorical mapping.

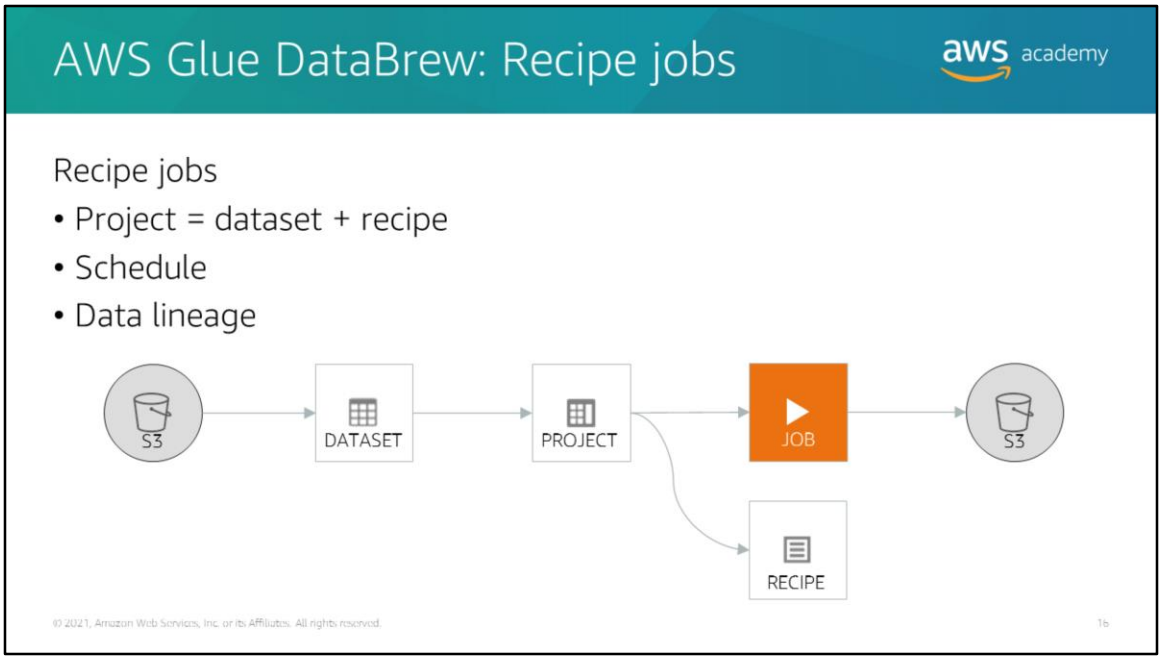


AWS Glue DataBrew can also tokenize text by removing stopwords, expanding contractions, and stemming. You will learn more about these topics later, in Section 3 of this module. These tokenization tasks are all done without code, and can speed up data processing for an NLP project.

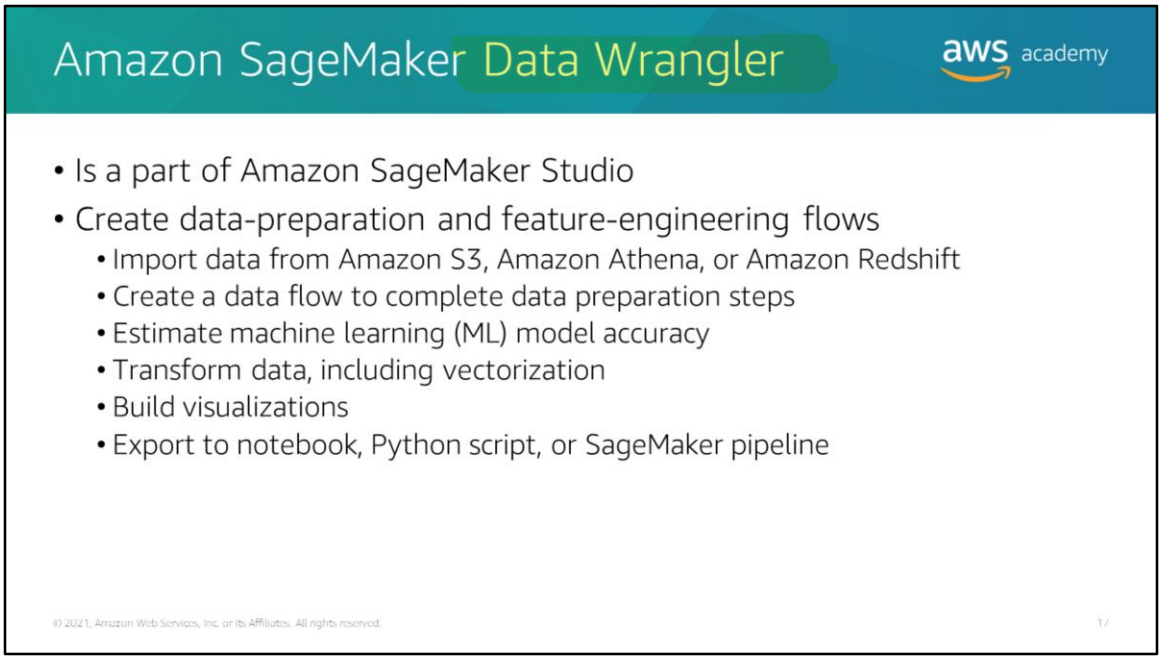


The data-processing steps are compiled into a recipe. The example recipe includes these steps:

1. It changes the text in the **reviewText** column to lowercase.
2. It removes any special characters and white space from the **reviewText** column.
3. It tokenizes the **reviewText** column.

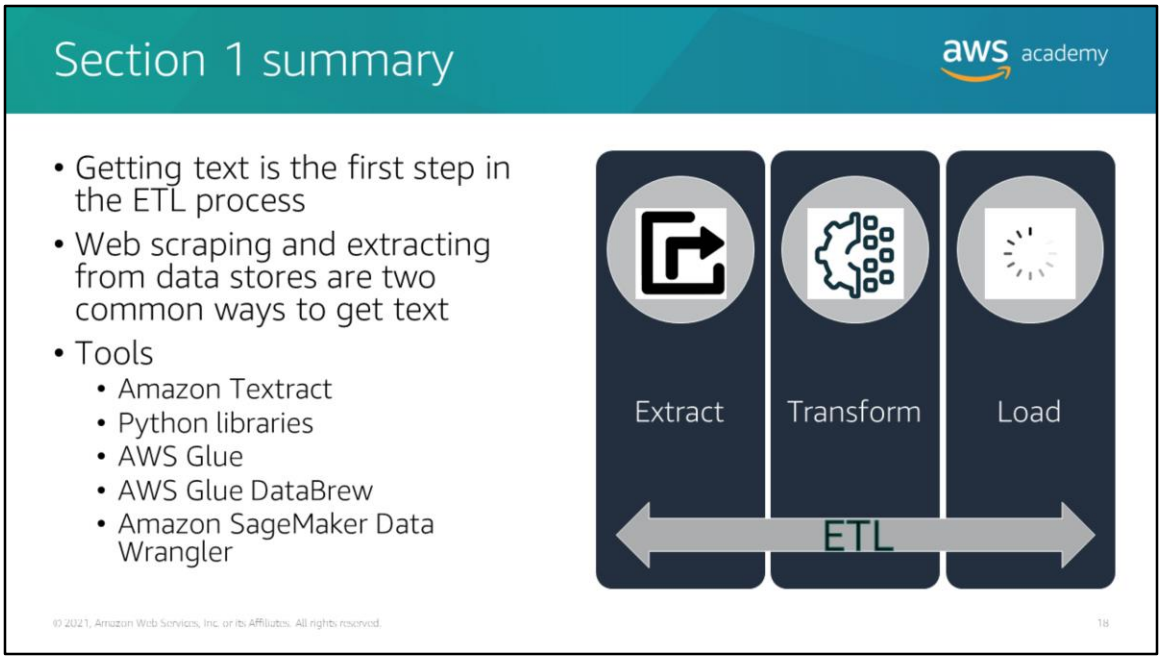


You can create a *recipe job* from a recipe. This recipe job can be run manually, or on a schedule that you choose. DataBrew maintains a view of the data lineage, which includes information about the origin of the data, what happens to the data, and where the data moves to over time. This information can be useful for troubleshooting issues.



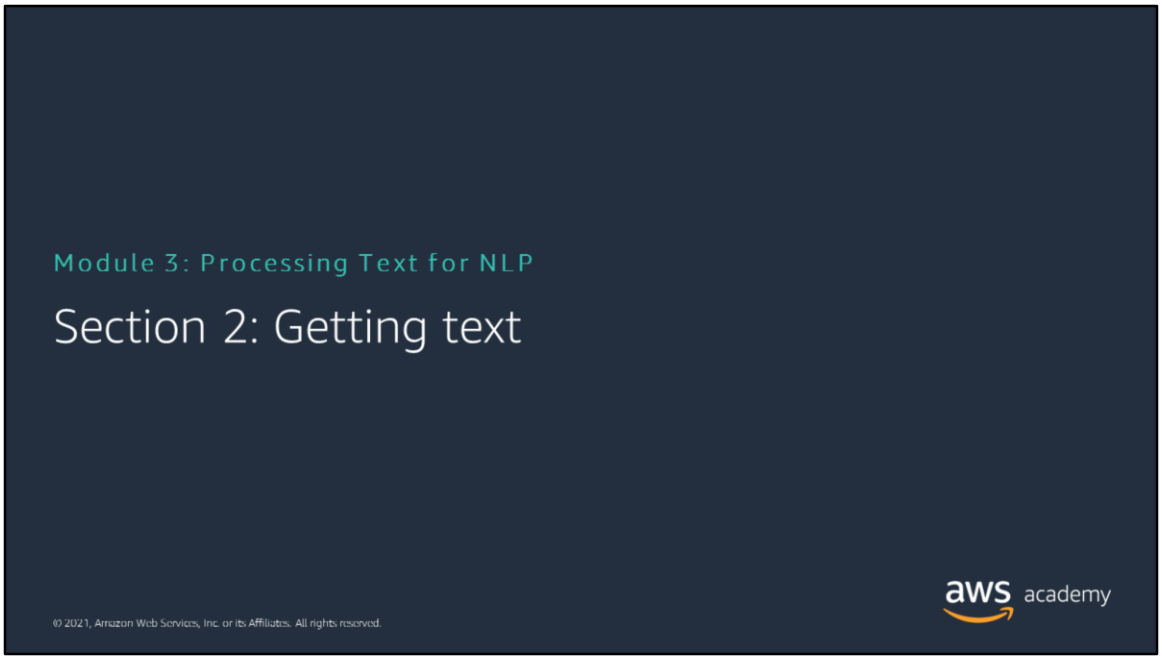
Amazon SageMaker Data Wrangler is another tool that you can use to capture text for your NLP application:

- First, you can import data from Amazon S3, Amazon Athena, or Amazon Redshift.
- Next, you can create a data flow to complete the data preparation steps.
- You can use one of several built-in transformers to perform operations, such as manipulating strings, cleaning data, or vectorizing your text dataset.
- In addition, you can build visualizations, such as scatter plots and histograms of your data.
- Finally, you can export your flow to a Jupyter Notebook, a Python script, or a SageMaker pipeline.



You can think of getting text into the ML pipeline as being the *extract* part of an ETL process. You learned about two general patterns for getting text: Web scraping and extracting text from data stores. You also learned about some tools that you can use for getting text. These tools include Python libraries like Beautiful Soup and Scrapy, Amazon Textract, AWS Glue, AWS Glue DataBrew, and Amazon SageMaker Data Wrangler.

You will get hands-on experience with Beautiful Soup and Amazon Textract in Guided Lab 3.1.

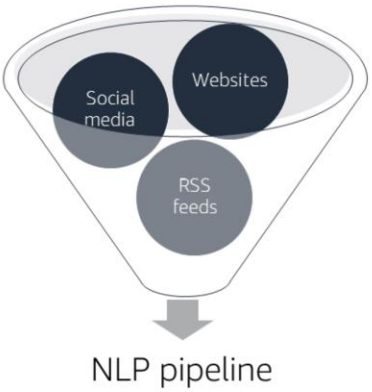


Introducing Section 2: Getting text.

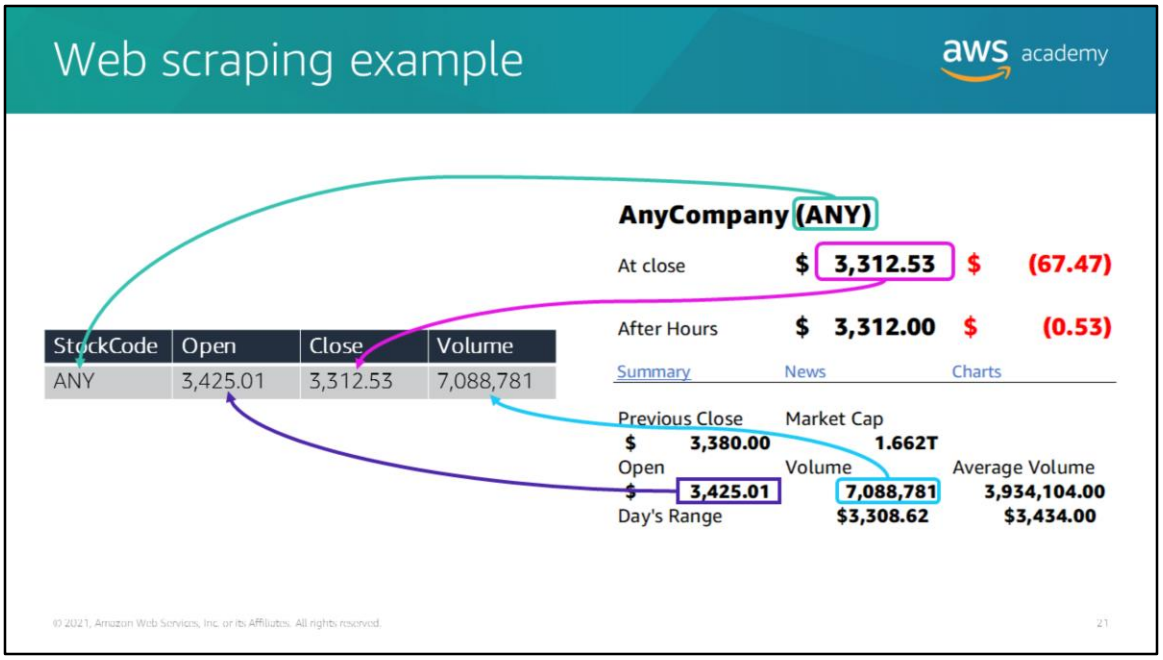
In this section, you will learn about some common ways to extract text.

Extracting text from the web

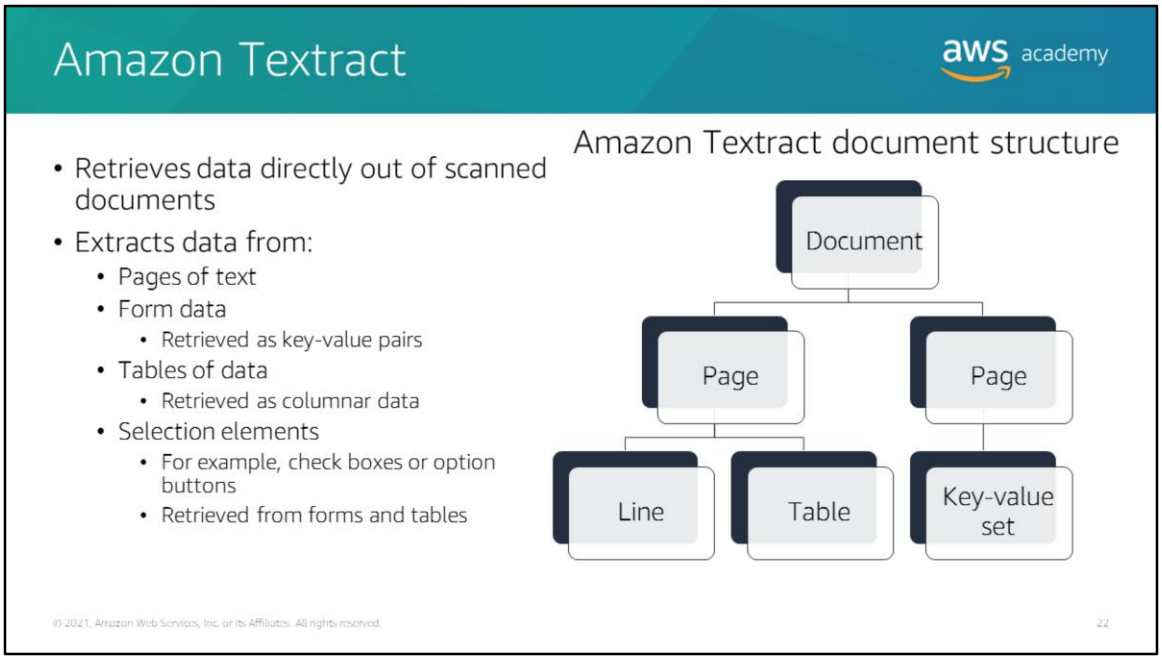
- You can automate data capture for your NLP application.
- Example use cases –
  - Marketing analysis: Gather product information.
  - Hiring: Harvesting resumes.
  - News analysis: Twitter feeds.
- Python libraries –
  - Beautiful Soup: Text-parsing tools for Hypertext Markup Language (HTML) and Extensible Markup Language (XML).
  - Scrapy: A set of tools for more complex projects.



Extracting text from the web is a common way of getting text into your NLP pipeline. Beautiful Soup and Scrapy are two of the more common tools for getting text. In the lab, you will use Beautiful Soup, but both tools follow the same general process.

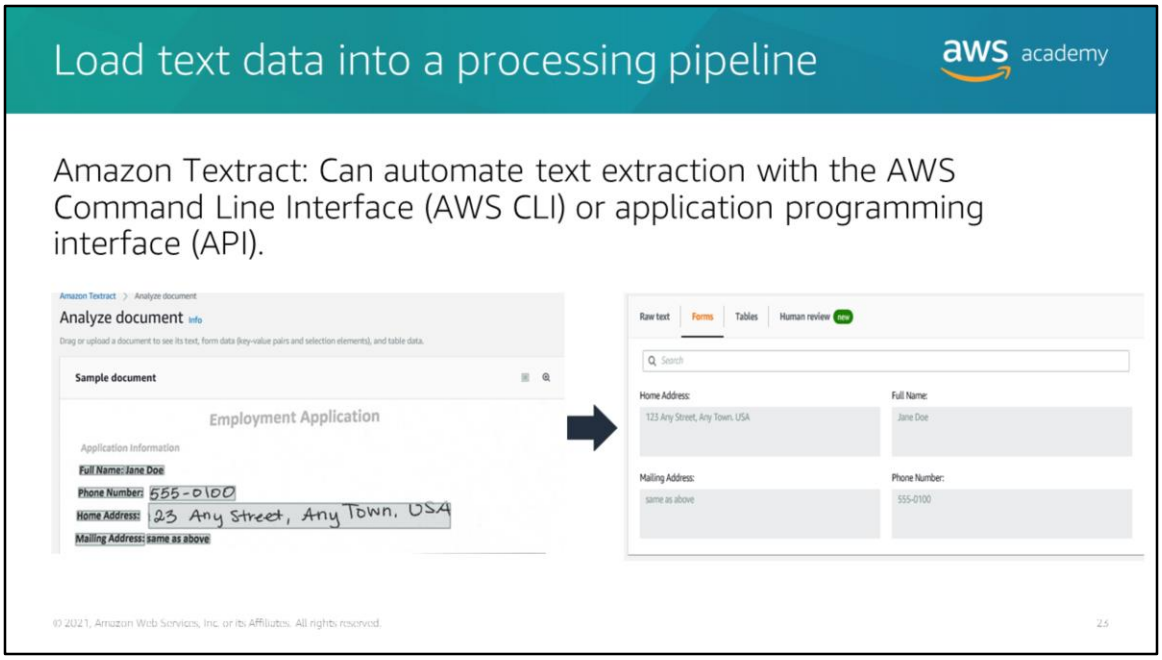


In this example, you might want to extract some key information, such as stock pricing, from a webpage. The information that you need is included among other data. By using text extraction tools and some basic HTML knowledge, you can write a script to extract the information that you need. You can also store the information in a format that is more appropriate to solving your business problem.



Amazon Textract retrieves data directly from scanned documents. You can interact with Amazon Textract by importing scanned images in the AWS Management Console. Alternatively, you can automate the import process by using the AWS Command Line Interface (AWS CLI) or the application programming interface (API).

Amazon Textract organizes text as individual blocks that it detects in a document. Documents are made up of pages. Pages can be of different types, such as individual lines of text, tables of text, or key-value pairs.

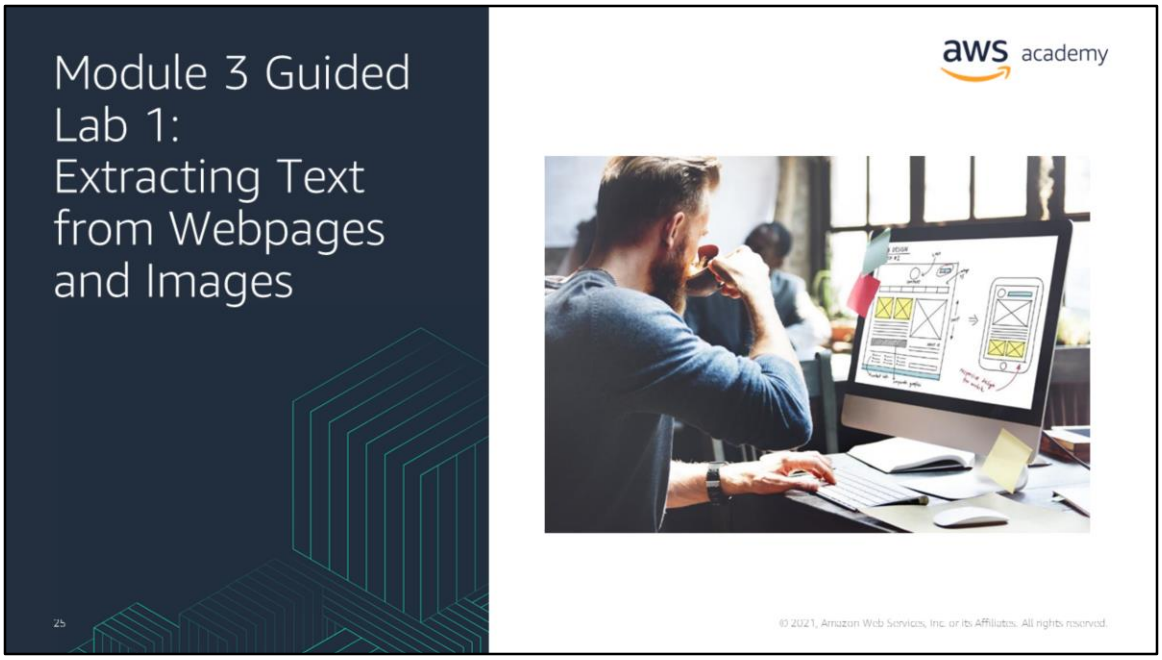


After you extract text, you load it into a processing pipeline. In lab 3.1 in this module, you will load data into a DataFrame by using the Python pandas library. You can also use the Amazon Textract service. Recall that with Amazon Textract, you can automate the loading process by using the AWS CLI or by calling the API directly.

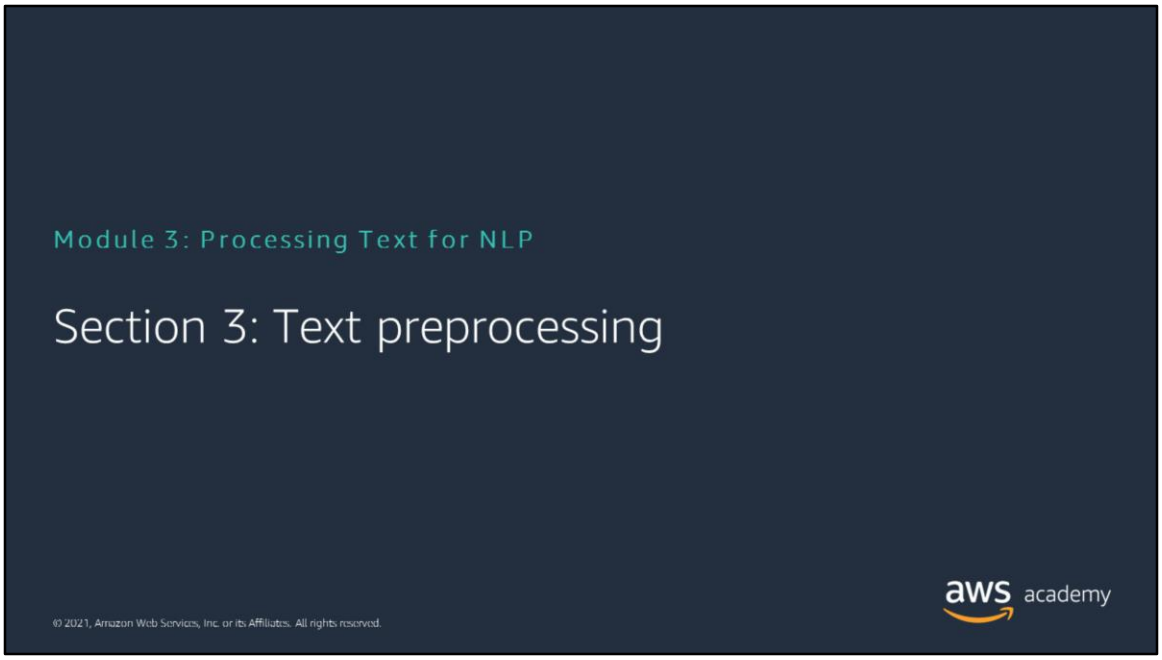
These screen captures show the Amazon Textract Analyze Document page in the AWS Management Console. This example shows how text was loaded from an employee application form. In addition to viewing the text as a form, you can view it as raw text, or as a table.

In this section, you learned how to capture text for your ML pipeline. You can use Python libraries, such as Beautiful Soup, or Amazon Textract.

You also learned how Amazon Textract retrieves text directly from scanned documents. It can extract raw text and tabular data. It can also extract information from form controls, such as check boxes or option buttons.

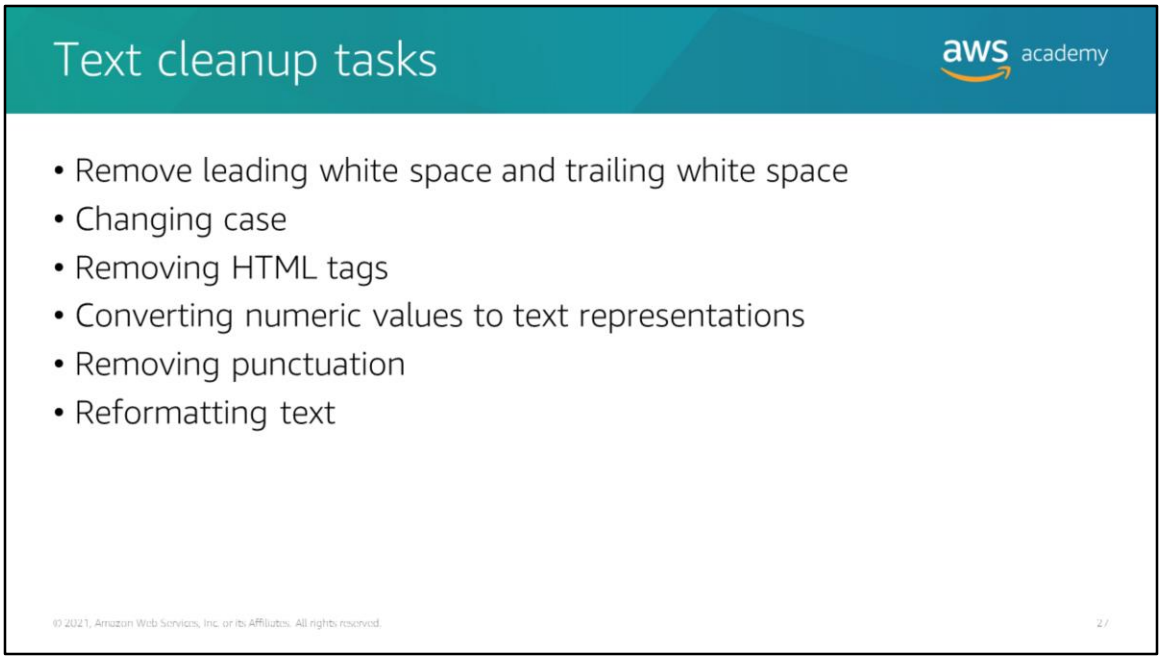


You will now complete Module 3 – Guided Lab 1: Extracting Text from Webpages and Images.



Introducing Section 3: Text preprocessing.

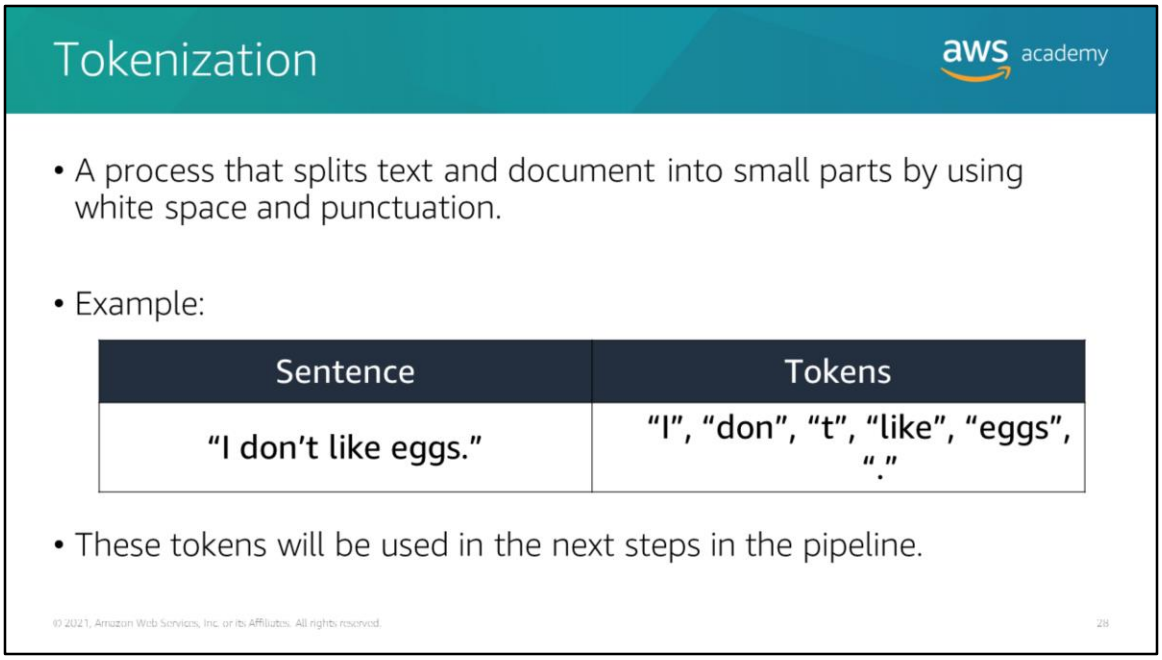
In this section, you learn about tokenization, stopword removal, stemming, and lemmatization.



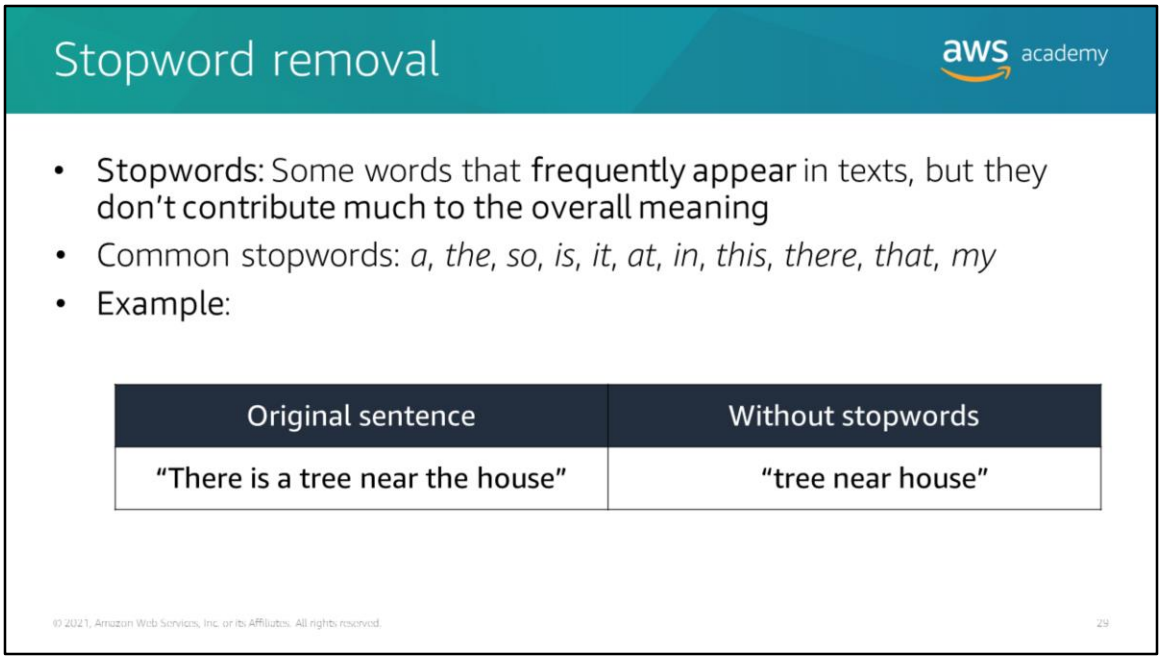
Text data usually needs some cleanup before you can start processing it. You can perform typical cleanup tasks by using tools such as AWS Glue DataBrew or Amazon SageMaker Data Wrangler, or by using Python code. Regardless of which tools you use for your project, general text-cleanup tasks usually include:

- Removing leading white space and trailing white space
- Changing the case to lowercase or uppercase
- Removing HTML, XML, or other markup tags
- Converting numerical values to text (for example changing *$9.99* to *nine dollars and ninety-nine cents*)
- Removing punctuation
- Reformatting the text

As with all tasks, you must experiment to see which method gives you the best results.



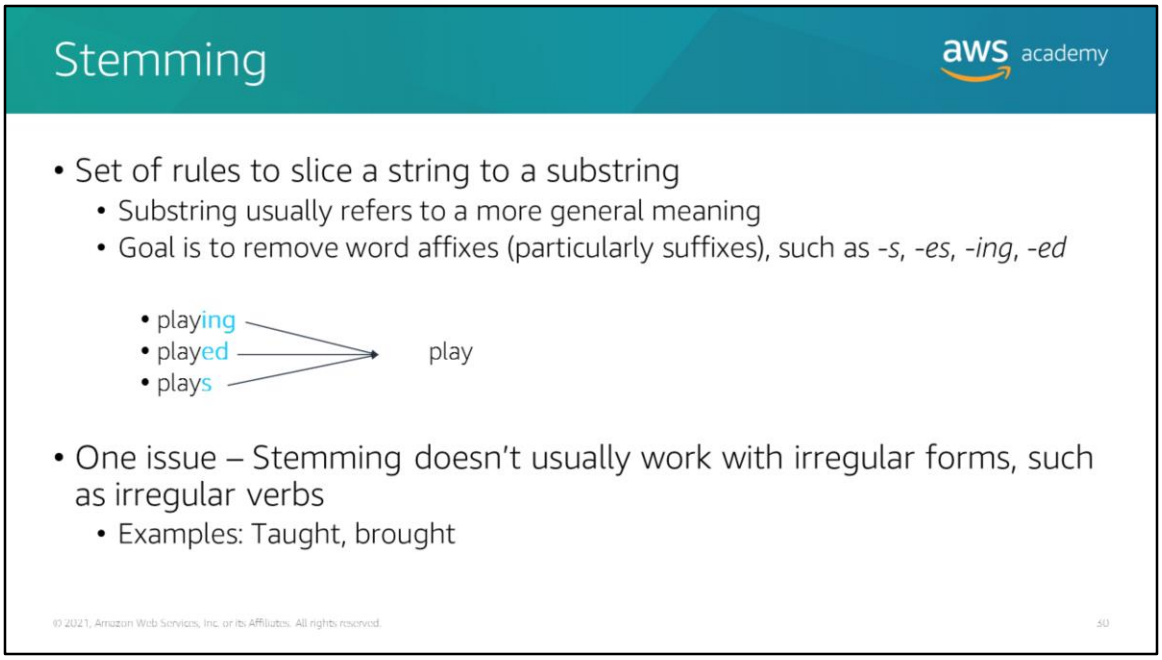
Tokenization splits text into smaller parts by using white space and punctuation. The sentence *"I don't like eggs"* could be tokenized into the following tokens: *"I", "Don"*, *"t"* , *"like"* , *"eggs"*, and *"."*. The tokenizer that you use determines how the text is split into tokens. The generated tokens are then used in the next steps of the processing pipeline.



Stopwords are words that frequently appear in sentences, buy that also don't contribute much to the overall meaning. Removing these stopwords will reduce the number of words that must converted to numbers for the ML algorithm to learn. In the example, the meaning of the sentence isn't lost by removing the stopwords.

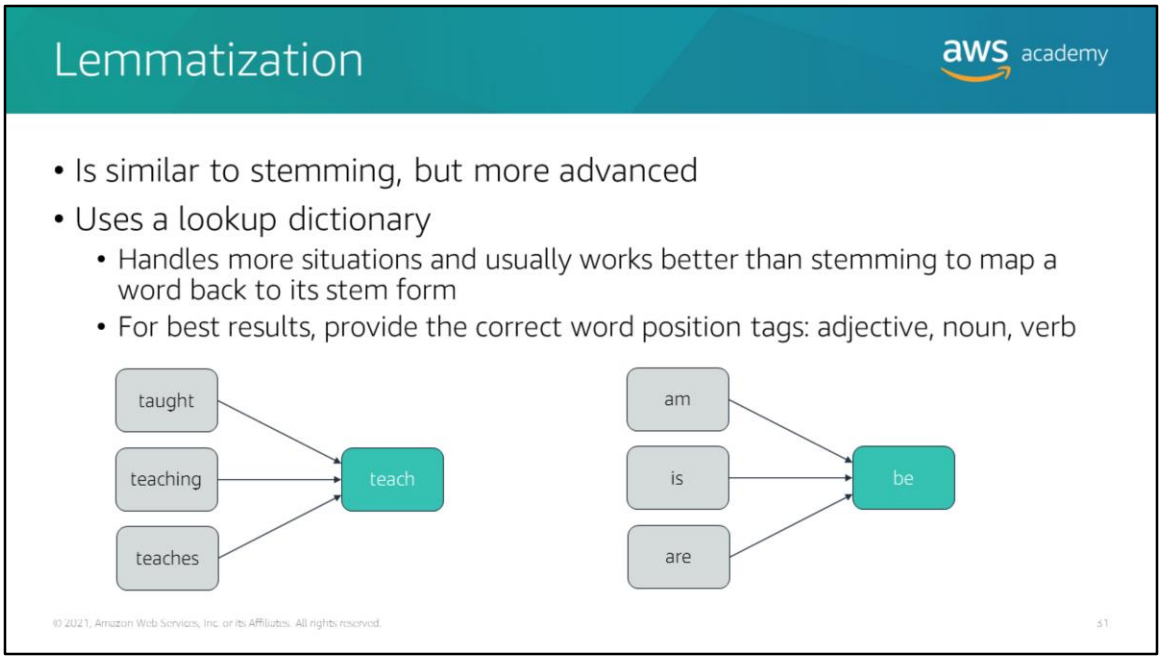
However, be careful to keep stopwords that are appropriate for the business problem that you are trying to solve. You do not need to remove all stopwords. For example, when you perform sentiment analysis, you will probably want to keep words such as *not* because they could change the sentiment of the sentence.

You can make your own list of stopwords. Many ML packages—such as the Natural Language Toolkit (NLTK)—provide curated lists of stopwords. It's often easier to start with those curated lists and exclude specific words that you want to keep. Like with most machine learning tasks, you must experiment with stopword removal to see which strategy gives you the best results.



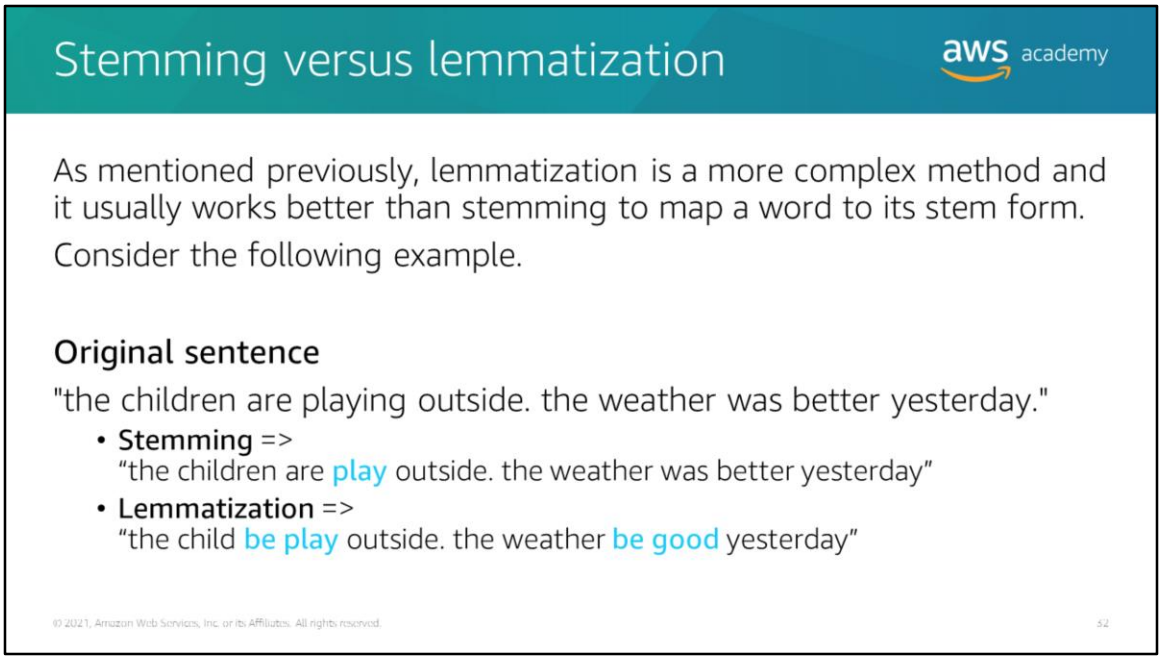
The English language contains many different forms of a word that can be used grammatically – for example, *play*, *plays*, *playing*, and *played*. When you build a model, you want to make sure that words with the same meaning have the same impact on the algorithm. A simple way to achieve this result is to remove the trailing suffixes such as *s*, *es*, *ing*, and *ed*. Two main stemmers are currently in use, the Porter and Snowball. Porter implements a set of rules to perform on the word. Snowball modifies these rules and can give slightly better results.

Stemming works for many words, but not with irregular verbs such as *taught* and *teach*, or *brought* and *bring*. Stemming does not take into account whether a word is a noun or a verb. For example, *saw* could be either *see* or *saw*, depending on whether the token is a verb or a noun. To solve these issues, you must use another technique, which is called lemmatization.



Lemmatization is another technique that is used in NLP. Lemmatization works by considering the morphological analysis of the words. Thus, *taught*, *teaching*, and *teaches* can be mapped to the word *teach*.

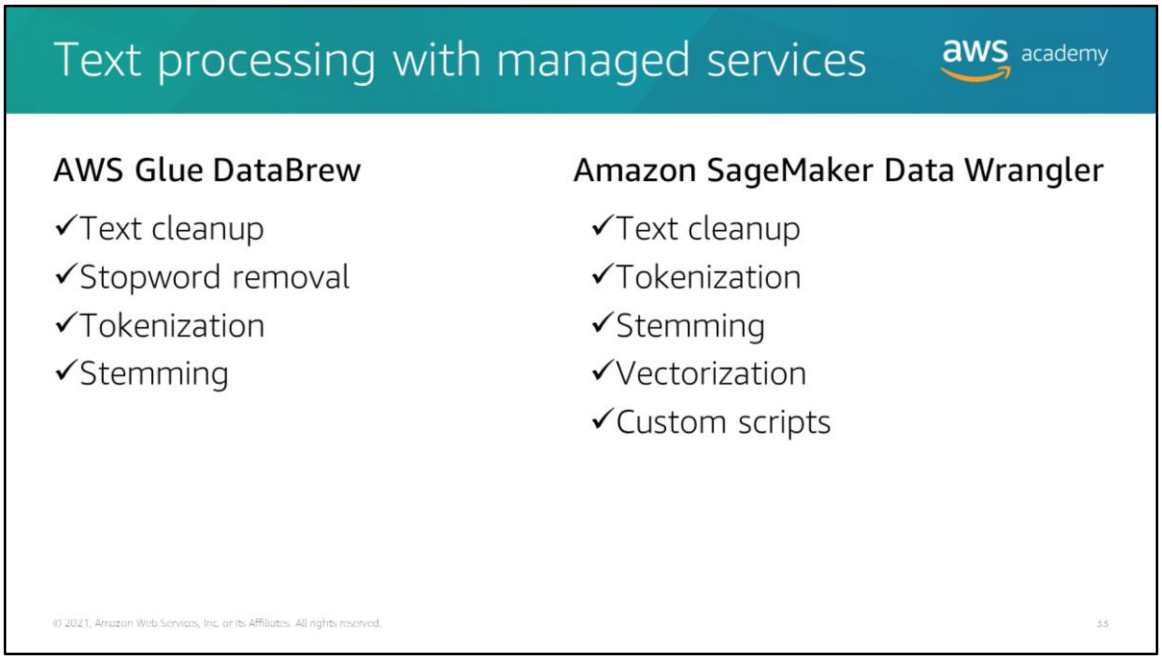
NLP tools provide functions to perform lemmatization, which are derived from human-produced, language-specific dictionaries. These dictionaries map a word back to its canonical form, known in NLP as a lemma, and account for the word's context. These lookup dictionaries are part of the NLP tools that you will use, such as those found in the Python NLTK.



It is easier to build a stemmer for a language than it is to build a lemmatizer for that language. Stemming tends to be faster because it performs only some string manipulation. A lemmatizer is slower because it must look up each word, and you might also need to calculate which part of speech a word is.

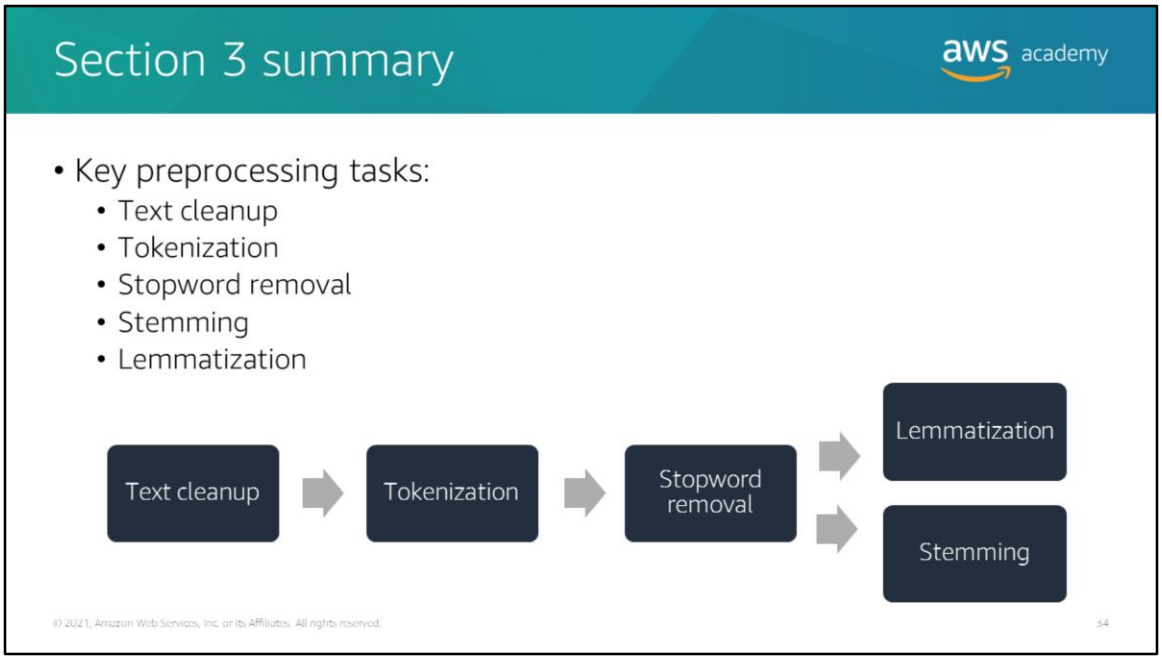
Whether stemming or lemmatizing gives you the best results depends on the type of problem that you want to solve. You must also consider the tradeoffs that you might need to make in terms of performance versus accuracy.

In the example, the lemmatized version has fewer tokens, and the words *are* and *was* are mapped to *be*. This mapping could improve the results of the model that you are training, depending on the problem that you are working on.



Many text-processing tasks can be performed by using managed services. Both AWS Glue DataBrew and Amazon SageMaker Data Wrangler can perform many of the cleanup tasks that you have learned about in this section. Which tool you use will depend on your data processing requirements.

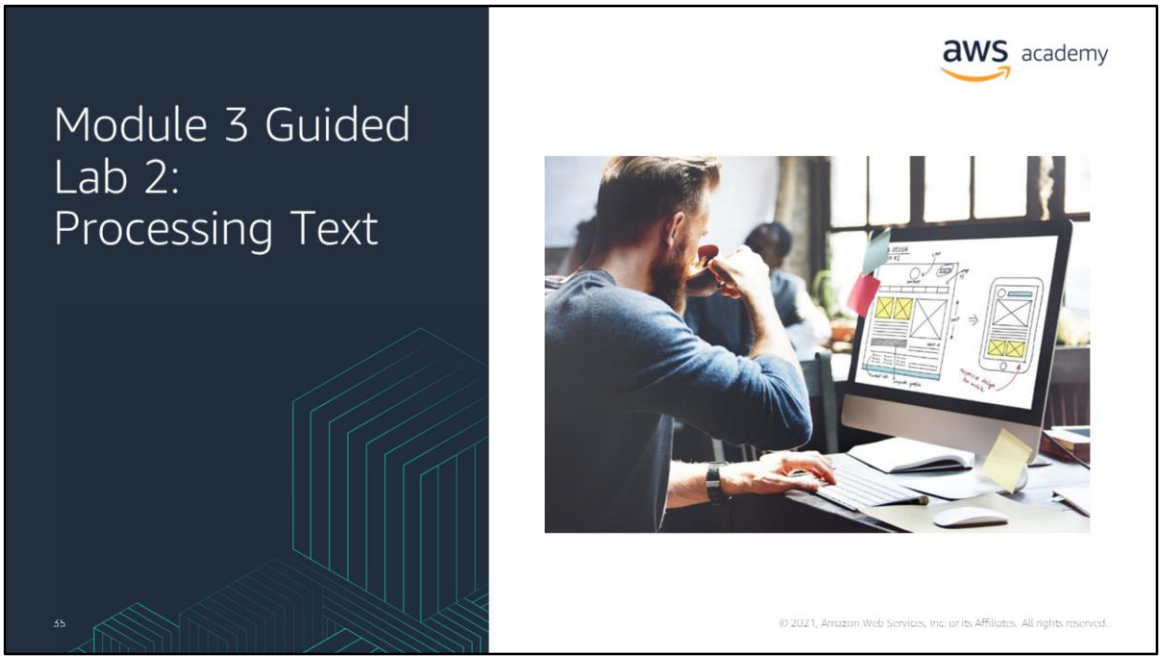
One advantage of Data Wrangler is the ability to use custom scripts. These scripts can be snippets of Python code that manipulate the data.



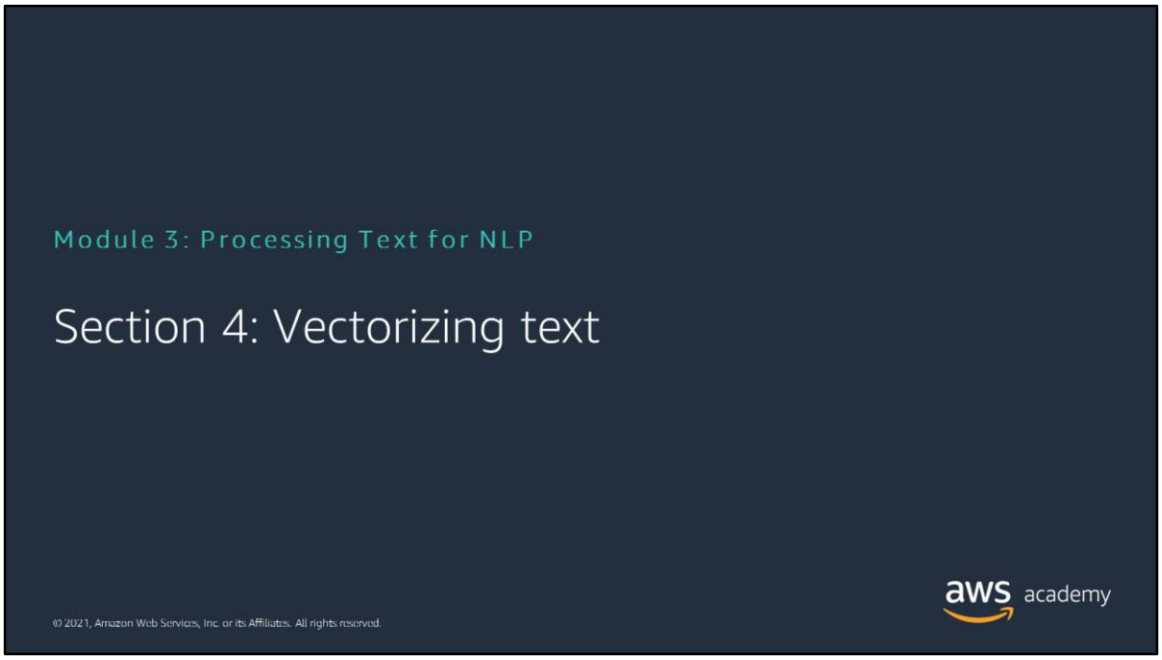
In this section, you learned about some common text-preprocessing tasks:

- Text cleanup involved converting the case to a standard case, removing white space, removing HTML tags, and converting numbers to text.
- Tokenization covered converting sentences to words and punctuation.
- Stopword removal looked at removing commonly used words from sentences to reduce the number of features.
- Stemming looked at simple parsing techniques to return a word to its root form.
- Lemmatization looked at using human-curated lists to map words back to a common form.

You might do some of these tasks when you prepare your text. Not all NLP tasks need all these steps, and you should experiment with each problem to determine which tasks give you the best results.

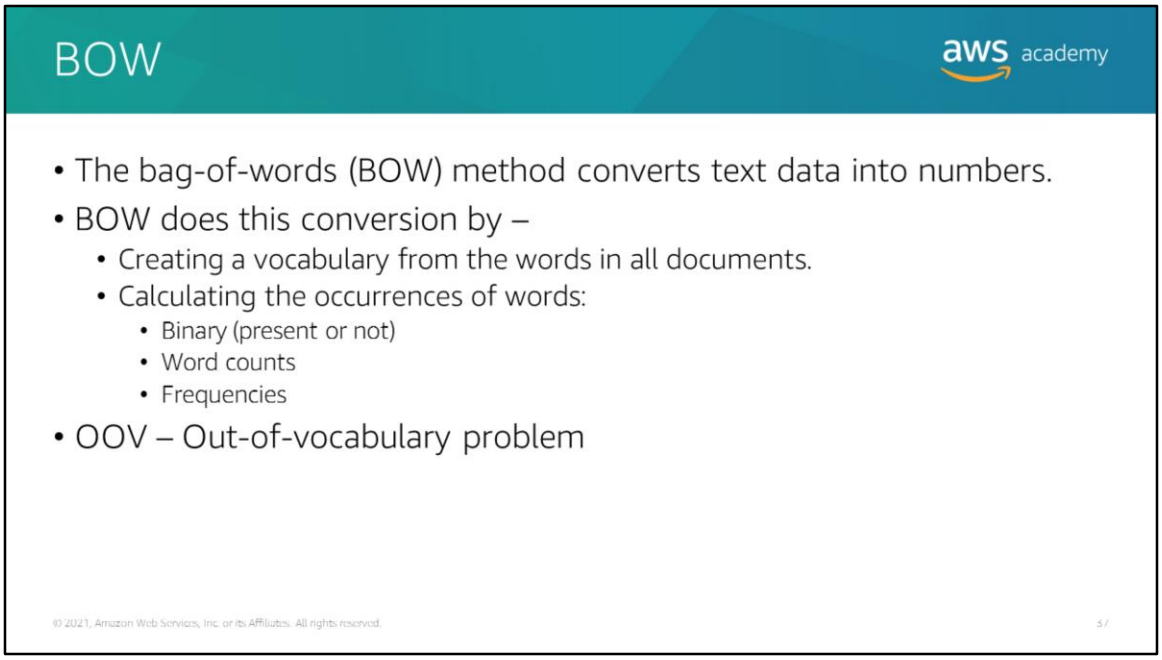


You will now complete Module 3 – Guided Lab 2: Processing Text.



Introducing Section 4: Vectorizing text.

In this section, you learn how to convert text data into a numerical representation.

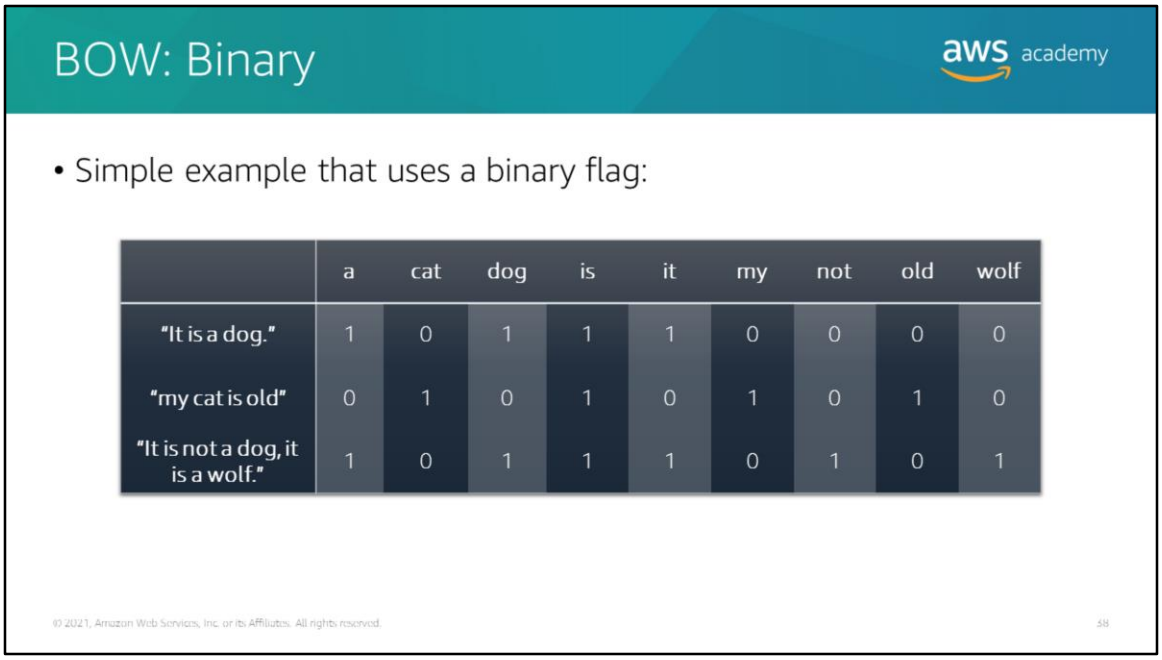


Bag-of-words (BOW) converts text data into numbers. BOW does this conversion by creating a vocabulary from all the words in all the documents. It then calculates the occurrences. Occurrences can be counted in three ways:

- Binary – Indicates whether the word is present or not present.
- Word counts – Counts how many times the word appears in the text.
- Frequencies – Provides a count of the words, normalized across the document.

One challenge of BOW is how to handle new words that were not part of the original vocabulary. This issue is known as out-of-vocabulary (OOV), and you will come back to it later in this section.

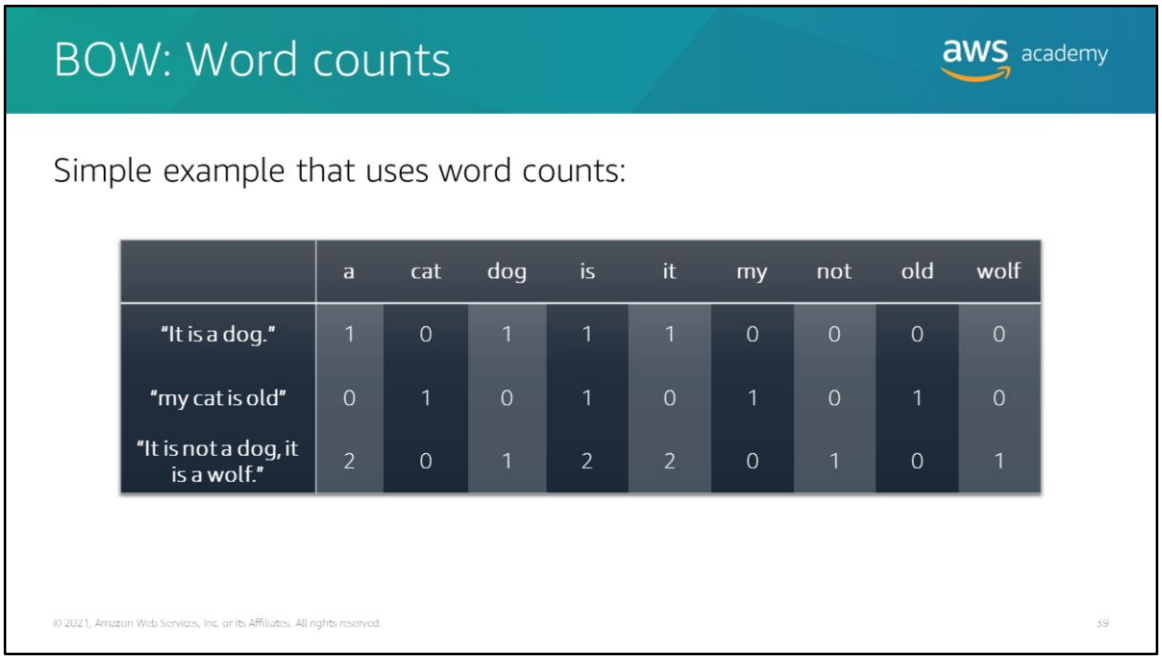
Next, you walk through an example of BOW over the next few slides.



In the example, you have three sentences:
- "It is a dog."
- "my cat is old"
- "It is not a dog, it is a wolf."

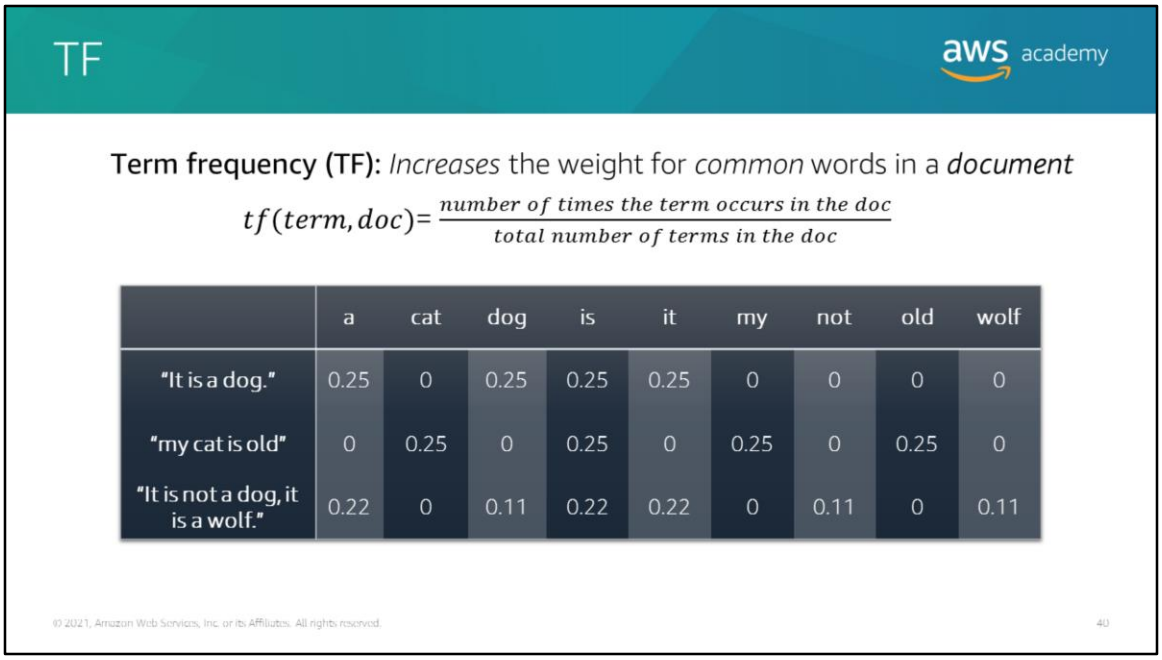
With a simple binary BOW, the column has a *1* if the sentence contains the word. Otherwise, the column has a *zero* (*0*).

Note: The output has no context, order, or meaning. The popularity or rareness of a term is not considered. The output only indicates whether the word is in the text.



A word count indicates whether a word is present. It also provides a count of the number of times that the word appears.

This method is slightly better than binary in that the number of word occurrences are captured, but the output still has no context, order, or meaning. The rareness of a term is not considered.

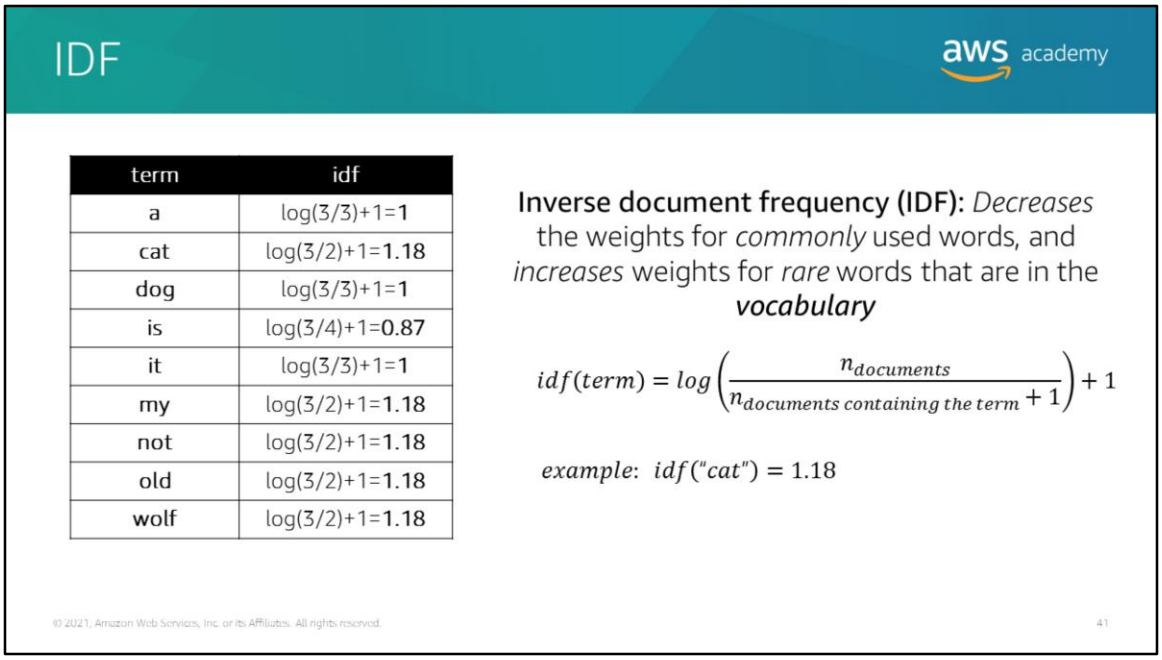


A term frequency (TF) increases the weight for common words in a document.

The formula on the slide is straightforward. For the text “It is a dog,” the terms are a, dog, is, and it. Because the document contains four words, each word gets a score of .25.

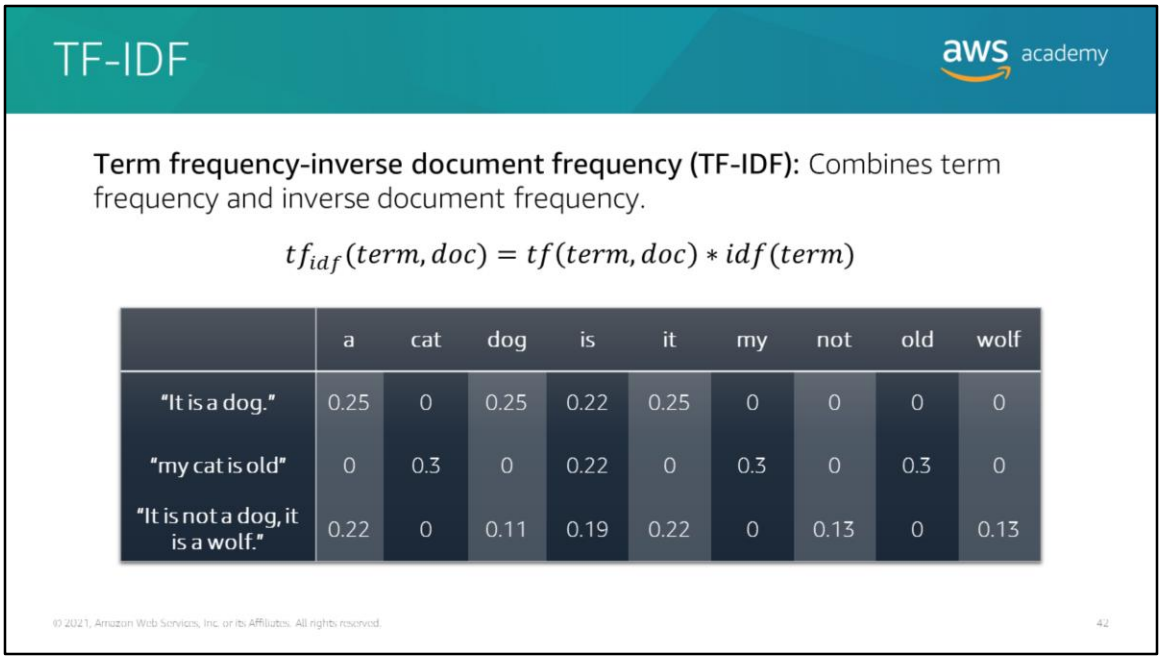
The text “It is not a dog, it is a wolf” has a total of nine words. The term wolf appears once. Applying the formula, you get 1/9=0.11. If you apply the formula to the word is, which occurs twice, you get 2/9 = .22.

You might wonder whether it is useful to increase the weight of what appears to be stopwords. The answer would depend on the problem, but remember that the results would be different if you remove the stopwords.



Inverse document frequency (IDF) increases the weights for rare words that are in the entire vocabulary, not just in the line of text.

The formula is shown, with the calculations for each term. The terms that appear less frequently have a higher IDF score.



	a	cat	dog	is	it	my	not	old	wolf
"It is a dog."	0.25	0	0.25	0.22	0.25	0	0	0	0
"my cat is old"	0	0.3	0	0.22	0	0.3	0	0.3	0
"It is not a dog, it is a wolf."	0.22	0	0.11	0.19	0.22	0	0.13	0	0.13

TF-IDF multiplies the TF and IDF metrics together.

If you apply the formula to the term *wolf*, you get these results:

tf = 1/9 = 0.11
idf = log(3/2)+1 = 1.18.

tf * idf = 0.11 * 1.18 = 0.13

TF-IDF highlights terms that are more unique in a document. For example, consider the term *dog*, which appears in two documents. It has a TF-IDF of .25 in the first document and .11 in the third document. Why does this result occur?

- The first document contains few words, so *dog* represents a larger portion of the document.
- The third document contains more words, so *dog* represents a smaller portion of the document.

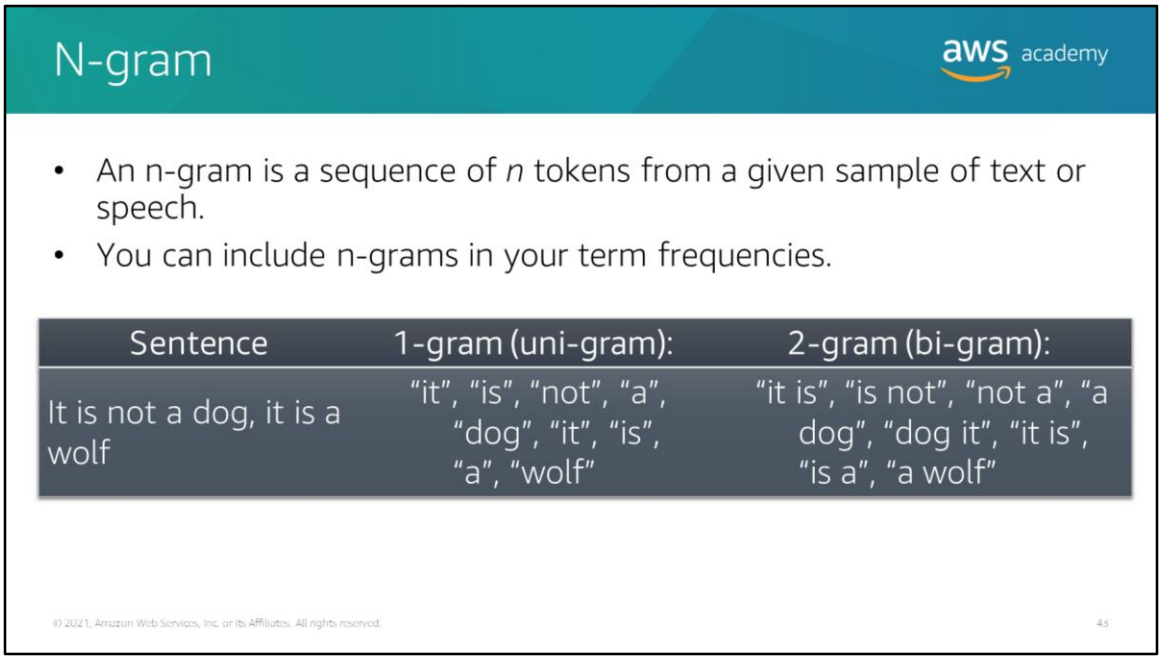
Now compare this case to the term *cat*, which has a score of .3. The term appears in only one document, and it is 25 percent of the document. The term *wolf* appears in only one document, but has a score of .13. The term *wolf* gets this result because *wolf* is one of nine terms in the document, so it is less important than the term *cat*.

What is the significance? If you were searching for the term *dog*, the first document (with a

higher score for *dog*) is probably more relevant than the third document. Equally significant, though the term *wolf* appears in the third document, the document is less likely to be about *wolf* because it contains many more terms.

If you asked about the second document, you could say that it is about *cat*, *my*, or *old*. In these small documents—and in this small corpus—it's difficult to demonstrate how the frequency impacts the results. You could imagine many documents that use the terms *my* and *old*—for example, *my old dog* and *my old wolf*. In this case, the scores for those words would drop, but the score for *cat* would remain high.

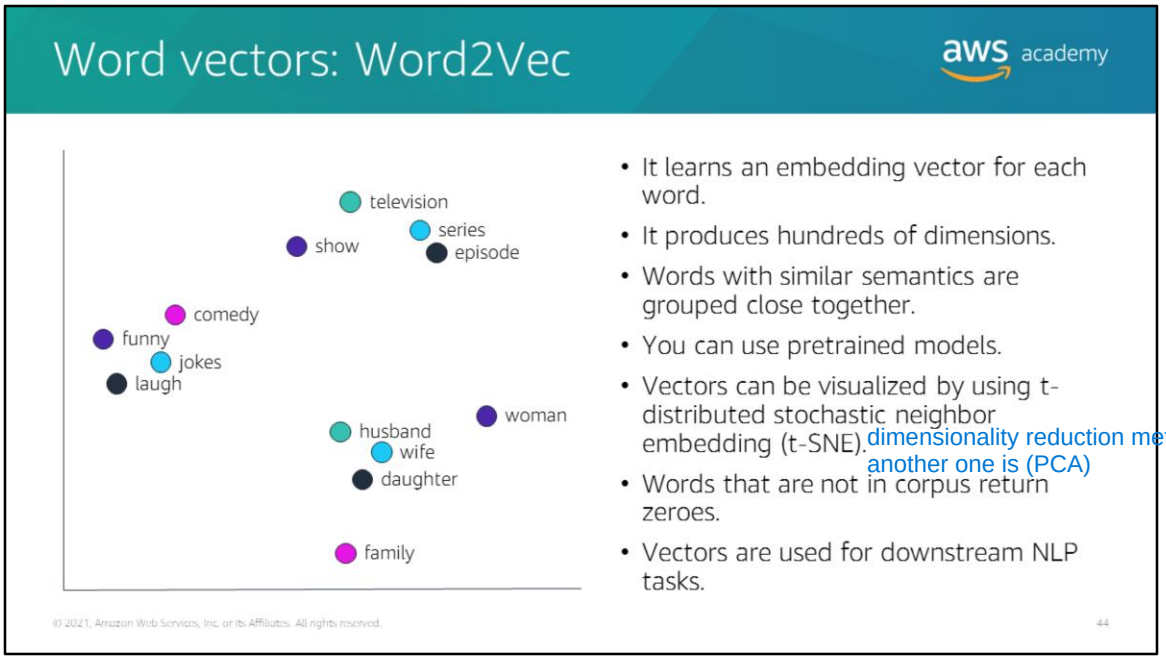
TF-IDF is often used for document search and keyword extraction.



An n-gram is a sequence of tokens from the text. 1-grams are single words, and 2-grams or (bi-grams) contain two words. You can also have 3-grams, 4-grams, and n-grams.

You can calculate the term frequency for an n-gram the same way that you would calculate the term frequency of a word.

You can use n-grams to determine the next word in a sequence. This technique is useful for detecting words that can be chunked together (such as *Pacific Ocean)* and analyzed as a single term (instead of analyzing *Pacific* and *Ocean* separately*).* It can also be used for checking spelling. For example, the text *Pacific Ocan* could be corrected to *Pacific Ocean* if it is highly probable that the word *Ocean* occurs after the word *Pacific*.



dimensionality reduction method
another one is (PCA)

[#ILT, #DIGITAL]B
One problem with BOW models is that you don't get any information about the meaning or the context of the captured word. Thus, BOW does not capture relationships between words, such as contextual closeness. This relationship could be as simple as the terms *cat*, *dog*, and *wolf*, which often refer to animals in the context of pets. (Some people have pet wolves.) Word vectors represent words as multidimensional and continuous floating point numbers. Semantically similar words are mapped to nearby points in the vector. Therefore, words such as *cat*, *dog*, and *wolf* should be close together, but *lightbulb* should be distant.

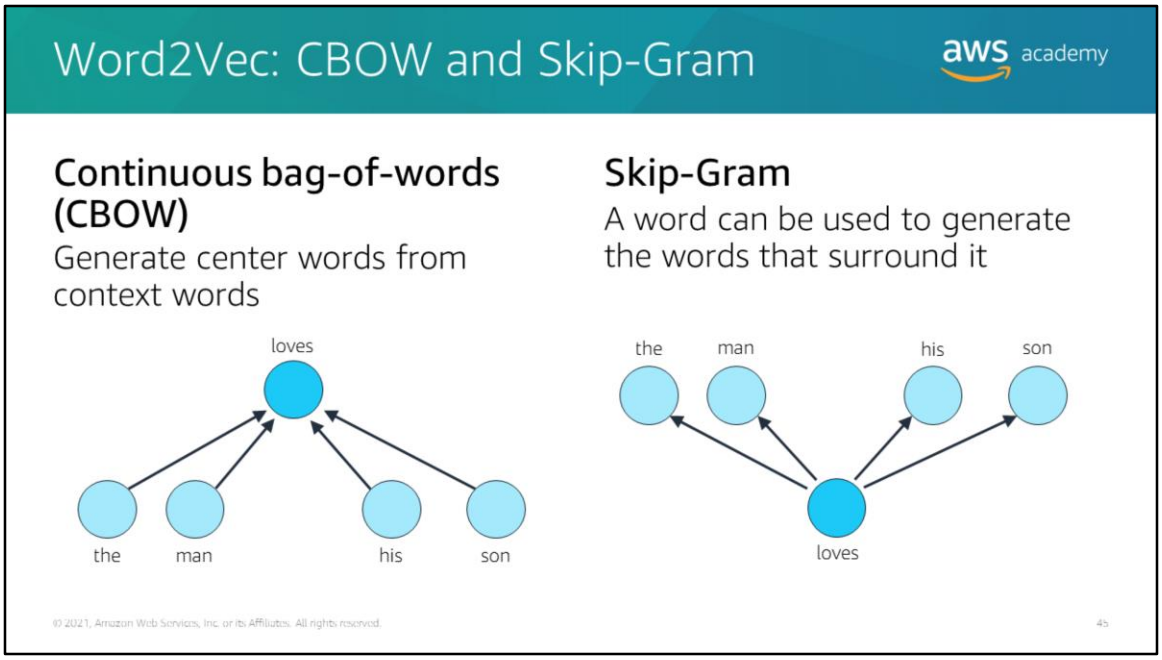
Word2vec takes an input collection of text, called a corpus, of words and produces a vector space of several hundred dimensions. Each unique word is assigned a corresponding vector based on the contexts in which it appears in the corpus. These vectors can then be used in downstream NLP tasks. Word2vec is itself an ML model that uses a neural network that must be trained. You can also use a pretrained model. One such model is GloVe, which has models that are trained from large amounts of data from Twitter and Wikipedia. You can find these models as part of the Gensim GitHub repository at Gensim-data.

It is difficult to visualize high-dimensional data because, at most, you can visually understand three-dimensional (3D) space. You can use tools, such as t-distributed stochastic neighbor embedding (t-SNE), to help visualize data. t-SNE converts similarities between data points to find a representation of those points in a lower-dimensional space. Therefore, you can plot these points, which can help you to visually understand the relationships.

Principal component analysis (PCA) is another method for reducing the number of dimensions. Both models reduce the number of dimensions. The key difference between t-SNE and PCA is that PCA creates a linear map of the variables that are in a higher-dimension space to a position in a lower-dimension space. It only uses the variables that have the greatest impact on the outcome of the model. The t-SNE model creates a similar mapping. However, instead of a linear mapping, it calculates the similarity of words in both the high and low dimension, and then minimizes the difference between the two sets of variables.

When you use a trained model, any words that were not part of the original corpus will return a zero matrix.
The vectors that are generated by vectorization are typically used for downstream NLP tasks, such as sentiment analysis, named entity recognition, and machine translation.

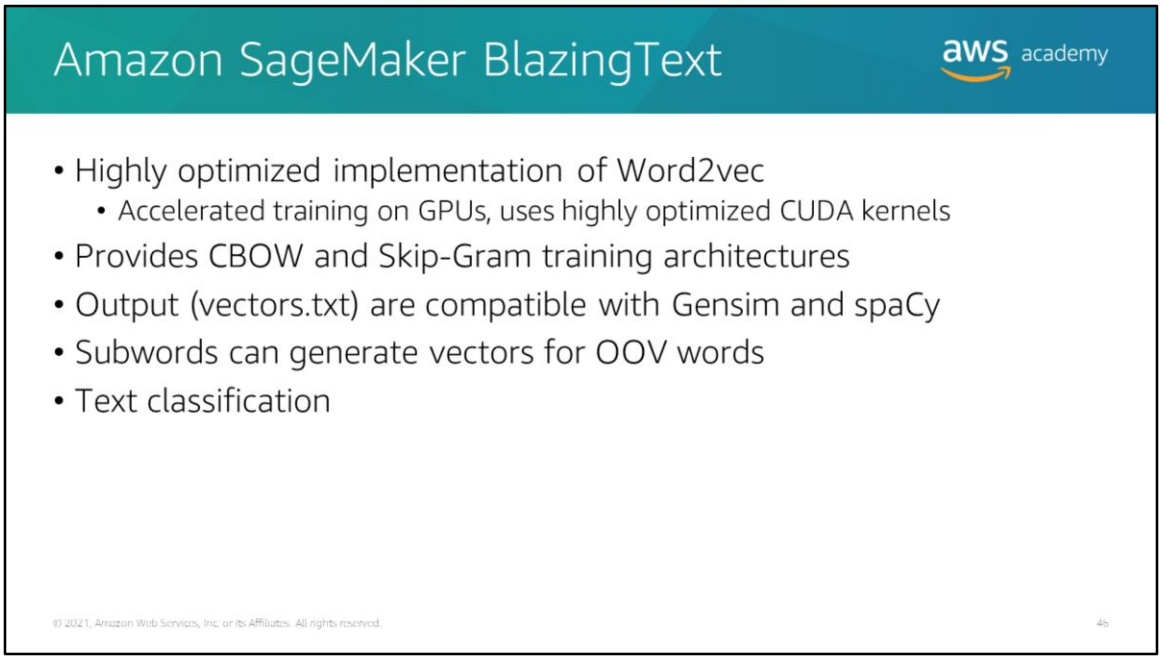
Word2vec has two variants: Continuous bag-of-words (CBOW) and Skip-Gram.



Continuous bag-of-words (CBOW) tries to predict the target (or center word) from the surrounding words. In this example, CBOW would predict the word *loves* from the context words *the*, *man*, *his*, and *son*.

Skip-Gram is the inverse of CBOW. Skip-Gram predicts the surrounding context words from the target word. It would predict *the*, *man*, *his*, and *son* from the target word *loves*.

Both variants return similar results, so it's worth experimenting with which method works best for the business problem that you are trying to solve.



The Amazon SageMaker BlazingText algorithm provides highly optimized implementations of the Word2vec and text classification algorithms. As mentioned previously, the Word2vec algorithm is useful for many downstream natural language processing (NLP) tasks, such as sentiment analysis, named entity recognition, and machine translation.

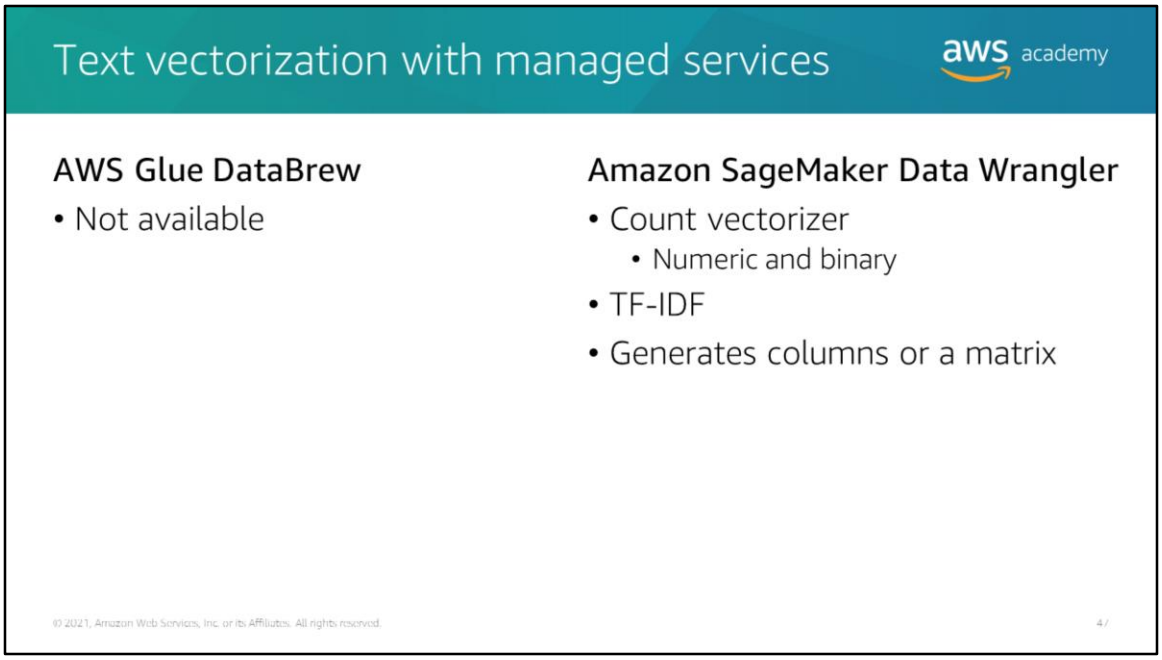
With the BlazingText algorithm, you can scale to large datasets. Similar to Word2Vec, BlazingText provides the Skip-Gram and CBOW training architectures. The BlazingText implementation of the supervised multi-class, multi-label text classification algorithm extends the fastText text classifier to use GPU acceleration with custom CUDA kernels. You can train a model on more than one billion words in a couple of minutes by using multi-core CPUs or a GPU. You can achieve performance on par with the deep learning text classification algorithms.

The SageMaker BlazingText algorithms provide the following features:

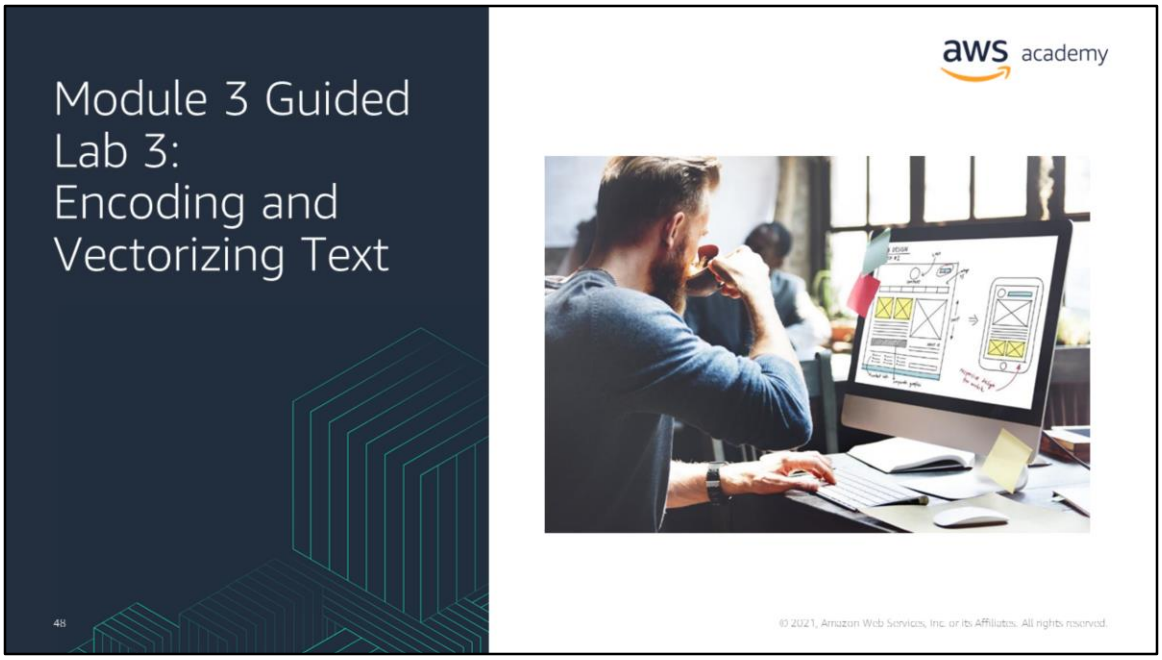
- Accelerated training of the fastText text classifier on multi-core CPUs or a GPU.
- Accelerated training of Word2vec on GPUs that use highly optimized CUDA kernels.
- Enriched word vectors with subword information by learning vector representations for character n-grams. With this approach, BlazingText can generate meaningful vectors for out-of-vocabulary (OOV) words by representing their vectors as the sum of the character n-gram (subword) vectors.

For Word2vec training, the model artifacts consist of *vectors.txt* (which contains words-to-

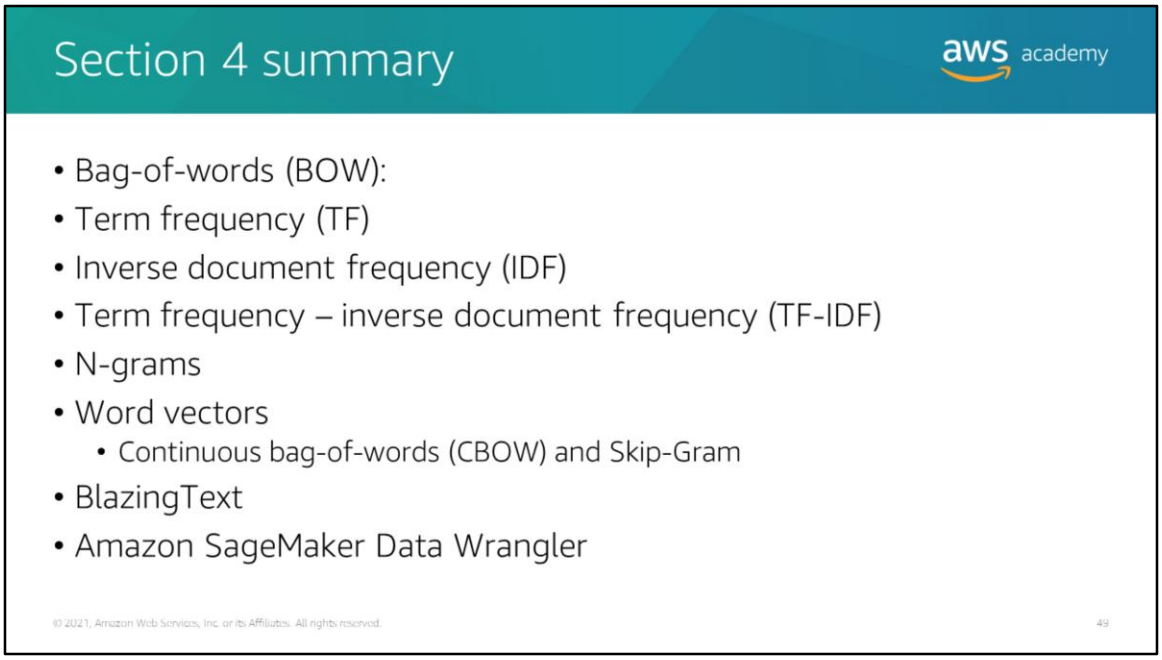
vectors mapping) and *vectors.bin* (a binary that BlazingText uses for hosting, inference, or both). *vectors.txt* stores the vectors in a format that is compatible with other tools, such as Gensim and spaCy.



AWS Glue DataBrew does not include transformations for text vectorization. However, Data Wrangler implements a count vectorizer with TF, and it can generate up to 100 new columns or a matrix.



You will now complete Module 3 – Guided Lab 3: Encoding and Vectorizing Text.

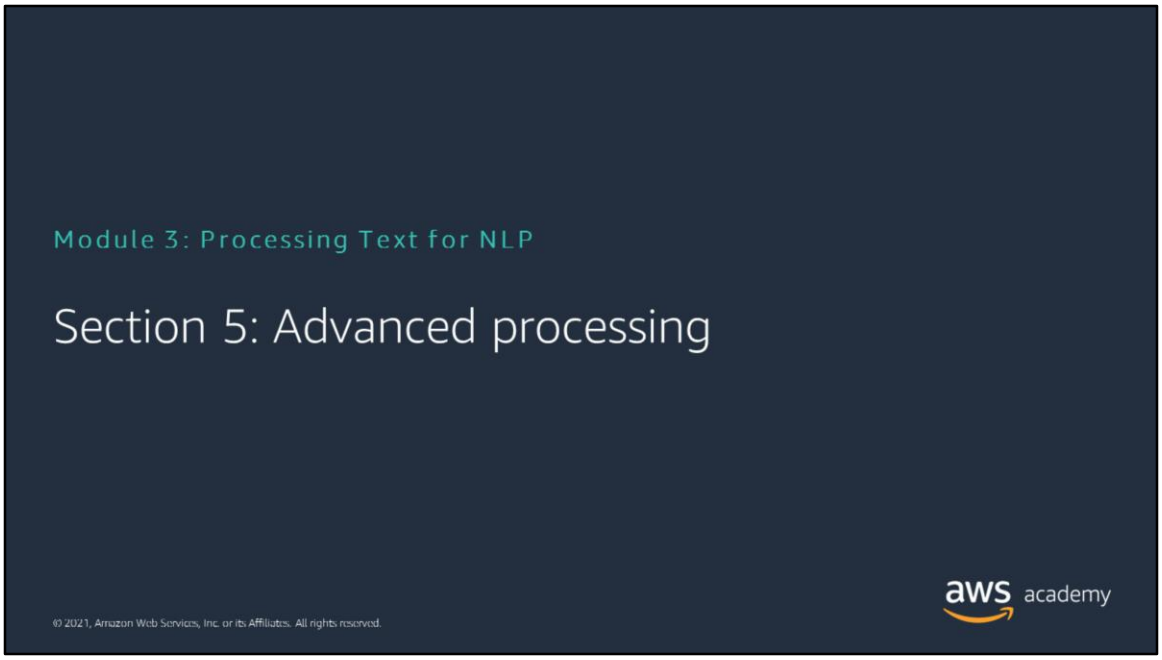


In this section, you learned about seven different models for converting text into a model that you can use with machine learning.

- First, you learned about the bag-of-words (BOW) model. BOW treats each word in the text as a unique entity in the *bag*, which is the collection of all the words.
- The next model was term frequency (TF). TF calculates the frequency of a word in the overall collection. It divides the number of instances for that word by the total number of words in the collection.
- Inverse document frequency (IDF) is a more complex approach to frequency. IDF increases the weight for rare words in the collection of words and decreases the weight of common words.
- Term frequency-inverse document frequency (TF-IDF) is a further refinement of IDF, which combines TF and IDF.
- Next, you learned about the n-gram, which is a model for breaking the text into tokens of various lengths. For example, a uni-gram model creates tokens for individual words, and a bi-gram model creates tokens for two-word pairs.
- Word vectors are a model that assigns a floating point number to each word in a multi-dimensional model. The advantage of word vectors is they can capture the context for how the words are used in the document.
- Continuous bag-of-words (CBOW) and Skip-Gram are two similar models that predict the

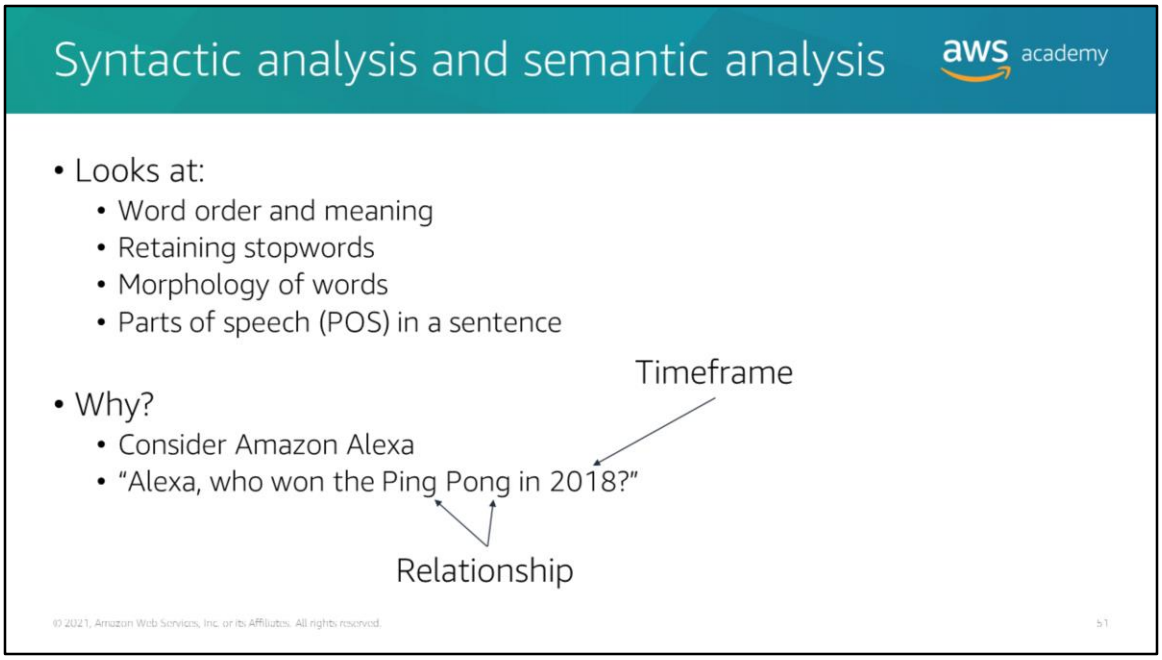
words that surround a particular word in a document. CBOW predicts the central word, and Skip-Grams predict the surrounding words based on the word that is in the center of a group of words.

- Finally, you learned about Amazon SageMaker BlazingText, which is an optimized implementation of Word2vec and other algorithms.



Introducing Section 5: Advanced processing.

In this section, you learn about more advanced techniques for processing text.



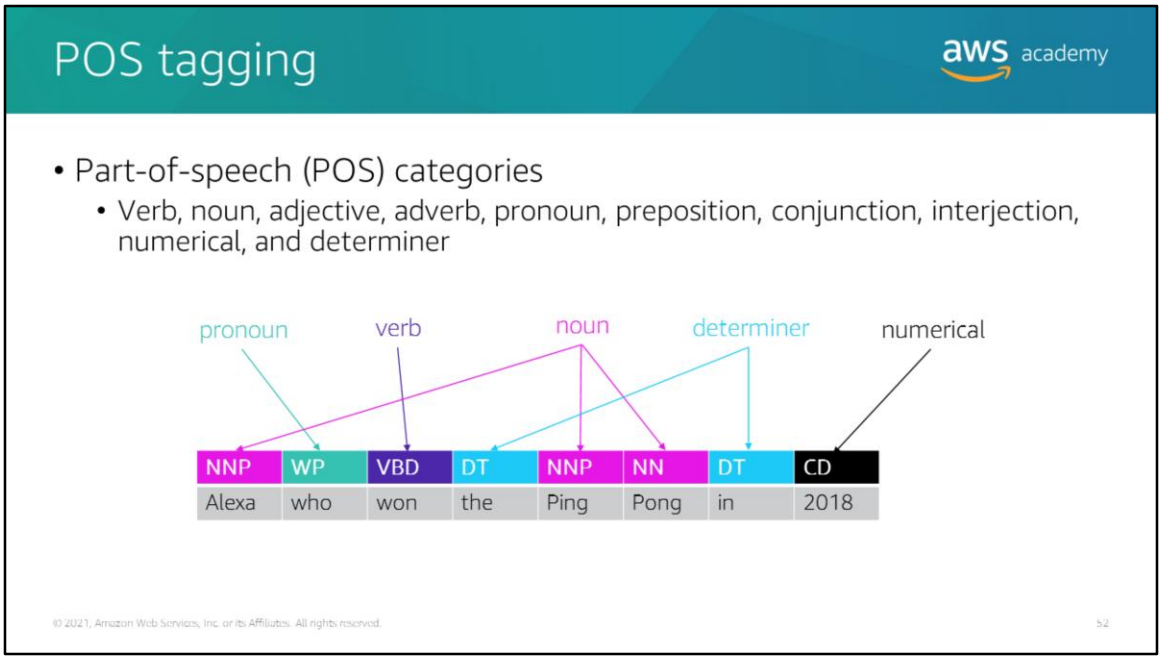
Syntactic analysis and semantic analysis are two techniques for understanding language. Syntax is the grammatical structure, and semantics represent the meaning. A sentence could be grammatically correct, but make no sense. For example, *dogs love to run* makes sense, but *dogs ride bananas* does not make sense. Both sentences are grammatically correct, but only the first is semantically correct.

When building applications such as chatbots, personal assistants, or search engines, it might be important to look at:

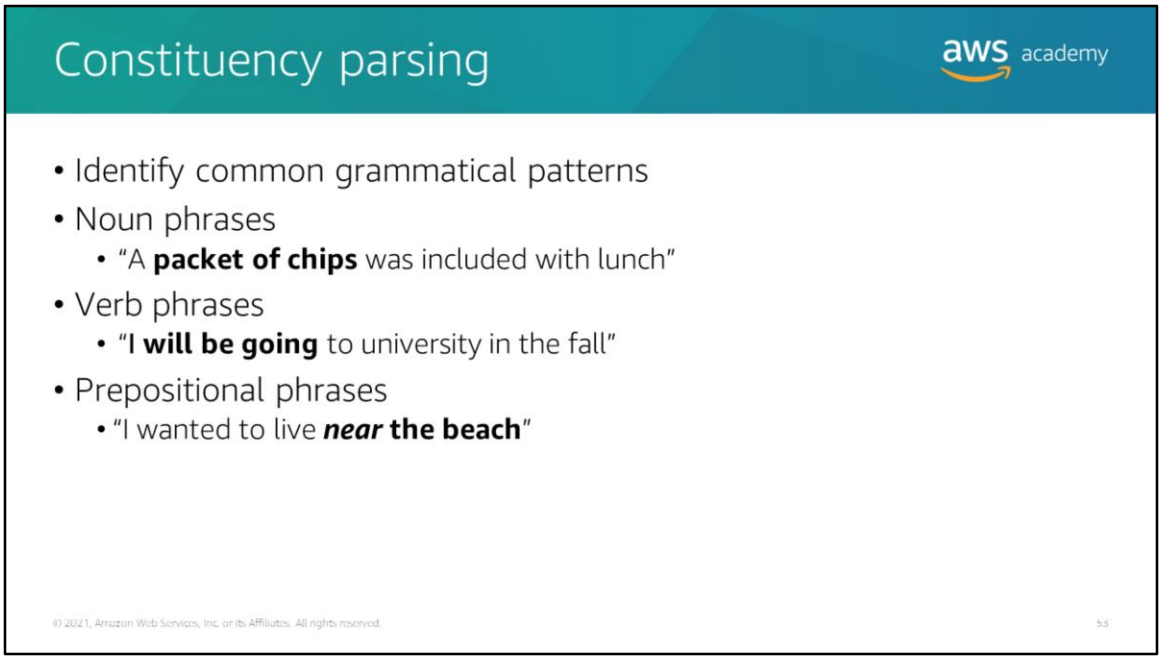
- Word order and meaning – Syntactical analysis looks at how words depend on the surrounding words. Changing the word order might make it difficult to understand the sentence. For example, consider Amazon Alexa. The phrase "Alexa, in the who won Ping Pong in 2018" has the same words as the example, but it does not make sense.
- Retaining stopwords – Removing stopwords might be good for reducing the dimensions. However, removing stopwords can also alter the meaning of a sentence.
- Understanding the morphology of words – Stemming and lemmatization will change the words and modify the grammar of the sentence.
- Identifying parts of speech in a sentence – Identifying the correct parts of speech (POS) is important for understanding the correct meaning of a word. For example, *My dog loves to fetch his stick* and *I must stick to this schedule* both contain the word *stick*. In the first example, *stick* is a noun. In the second example, *stick* is a verb.

All these criteria are important when you work out meaning. For example, suppose that you

ask, *"Alexa, who won the Ping Pong in 2018?"* Alexa must understand the structure of the question to be able to answer it. The phrase *Ping Pong* relates to an event, and *in 2018* refers to a timeframe. The event and the timeframe have a relationship between them. You can find these dependencies by parsing the text.

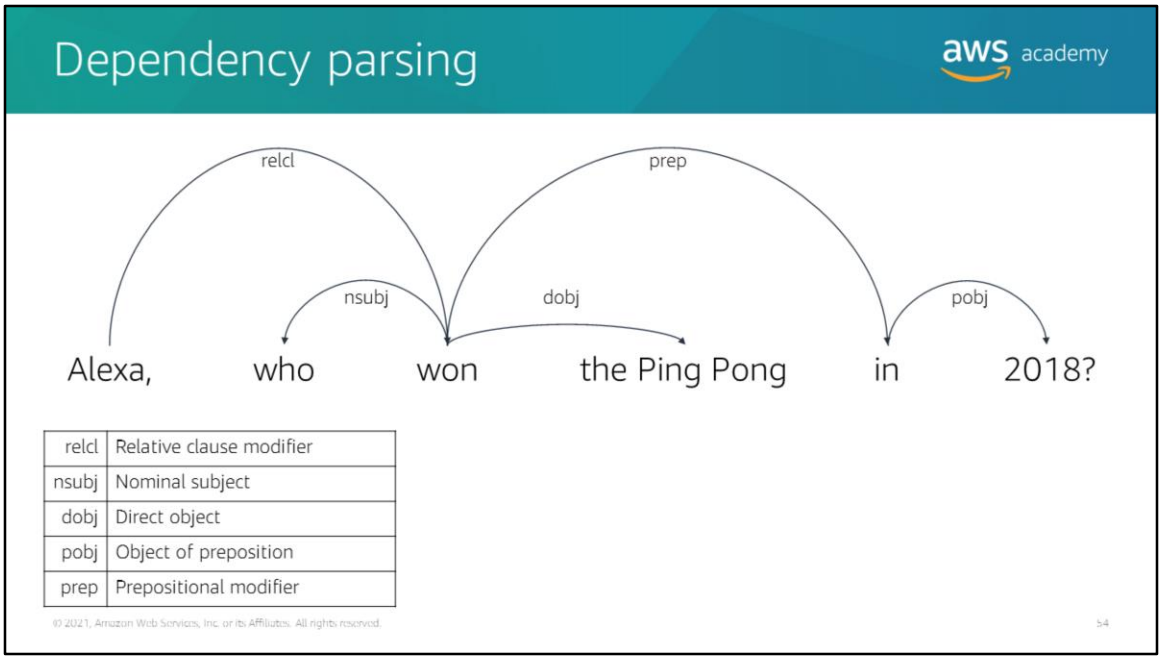


Part-of-speech (POS) tagging means labeling words with the appropriate parts of speech. Parts of speech can include verbs, nouns, adjectives, adverbs, pronouns, prepositions, conjunctions, interjections, numericals, or determiners. POS tagging is usually done with the help of a pretrained tagger. Most taggers use the list of tags from the popular Penn Treebank tag set (POS tags used in the Penn Treebank Project). To identify the correct tag, the previous and next words are considered. Understanding the structure of the sentence is a key step when you want to extract information from text, such as names, events, and locations. All other techniques use POS tags to parse a sentence.



To understand common grammatical patterns, you must divide words into groups called constituents. This division is based on the role of the word in the sentence. The English language has three main constituencies that are used in NLP:

- Noun phrases – Phrases that are built around a single noun. For example: "A ***packet* of chips** was included with lunch."
- Verb phrases – Verbal parts of a clause. For example, "She ***had been living*** in Hong Kong."
- Prepositional phrases – Phrases where the preposition always comes at the beginning. For example: "I wanted to live ***near*** **the beach**."

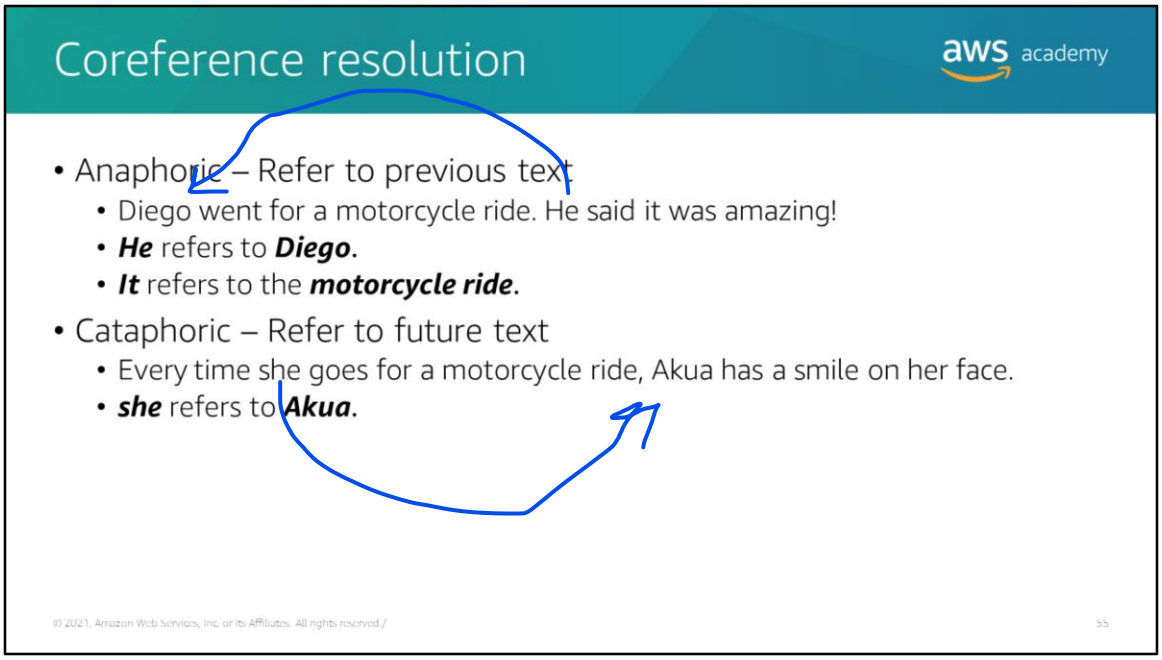


When the parts of speech are tagged—and noun, verb, and preposition phrases are detected—you can understand the meaning of the text. Typically, English has a subject-verb-object (SVO) word order. The basic idea is that a sentence is about something, and it usually involves a subject, a verb, and an object. Most sentences are more complex, but the same parsing rules can be applied.

The example on the slide shows the SVO:

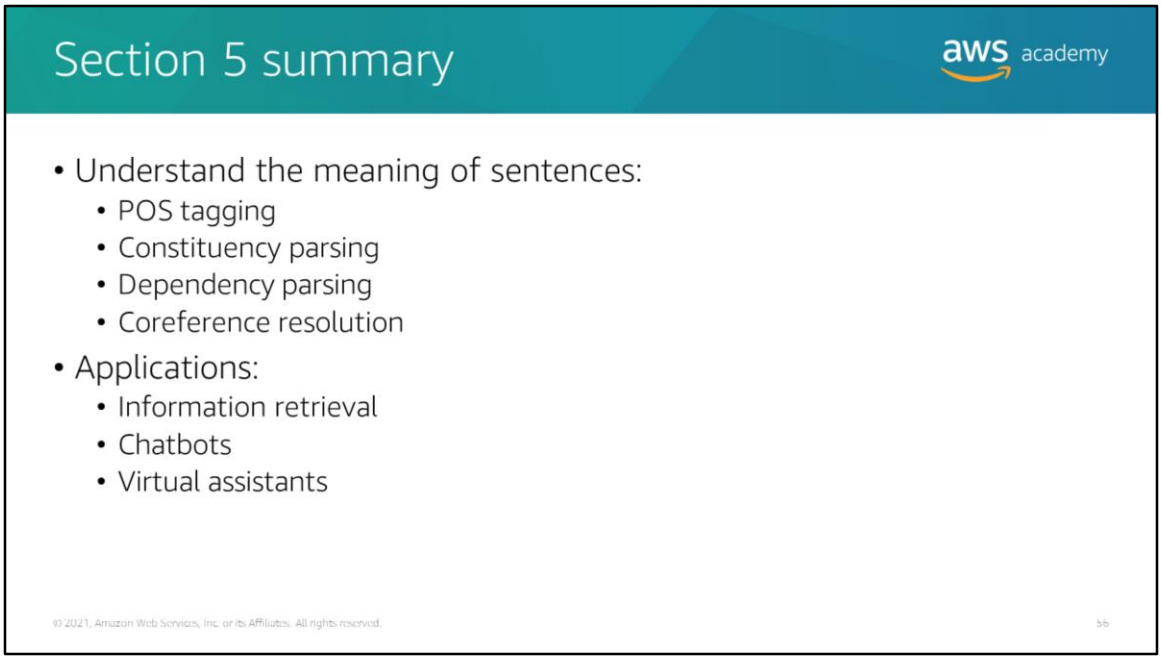
- **Who** is the subject. In this case Alexa deduces that the who must be a table tennis (ping pong) team based on the object **the Ping Pong**.
- **Won** is the verb.
- **The Ping Pong in 2018** is the object.

With this information, Alexa can begin to answer the question.



The goal of coreference resolution is to identify mentions in text that refer to the same underlying object.

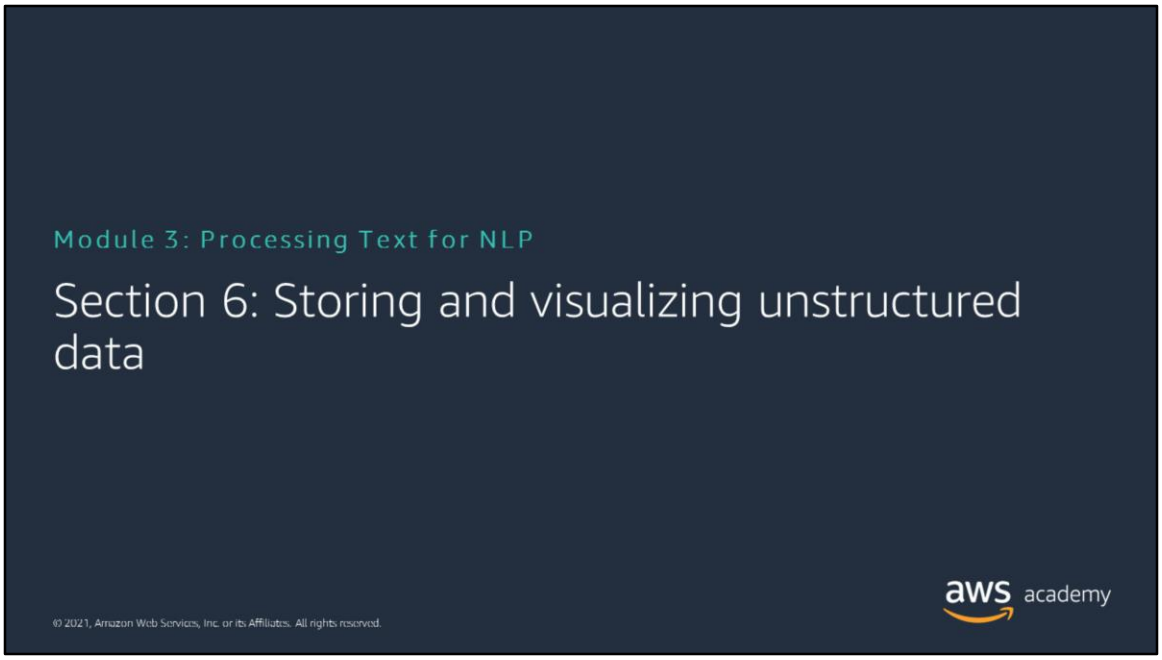
Coreference can be anaphoric, which refers to the previous text. Coreference can also be cataphoric, which refers to future text. It is important for information systems, chatbots, and virtual assistants (such as Amazon Alexa) to understand what *he*, *she,* or *it* refers to in text.



In this section, you learned about some of the more advanced NLP processing tasks, which included:

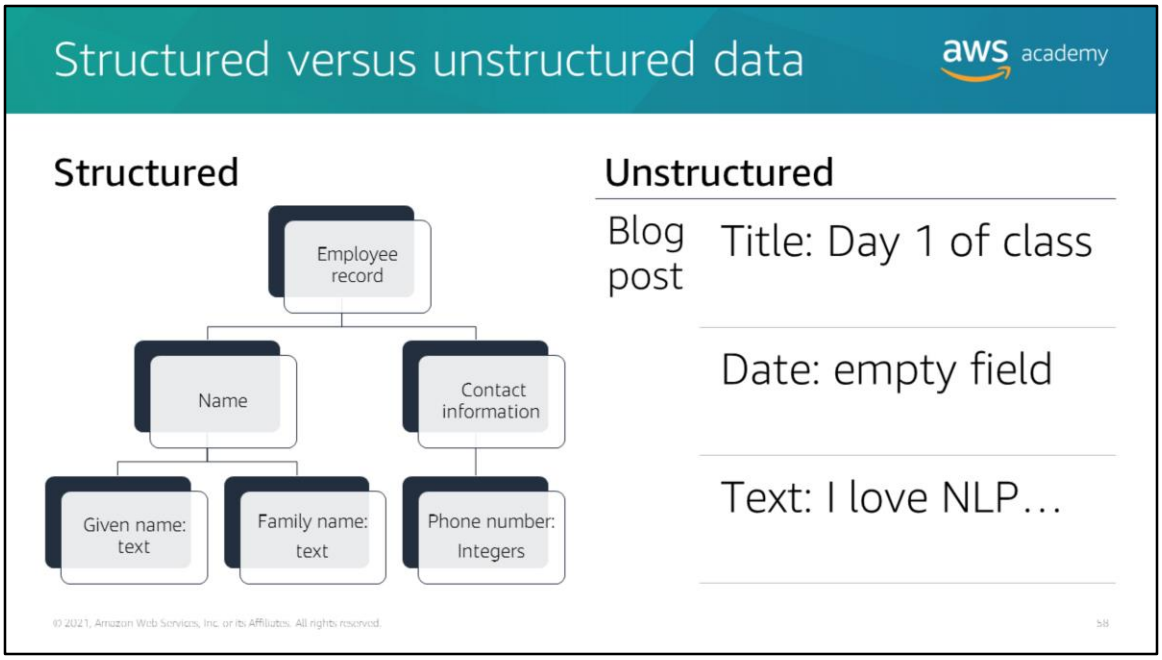
- POS tagging – So you can understand the individual components of a sentence
- Constituency parsing – To identify phrases from words
- Dependency parsing – To understand the relationship between the words and phrases
- Coreference resolution – To identify entity mentions within the text

You will further examine some of these processes and how they are used in later modules that cover information retrieval and topic modeling. Although chatbots and virtual assistants are not covered in this course, these techniques are also useful for building these tools.



Introducing Section 6: Storing and visualizing unstructured data.

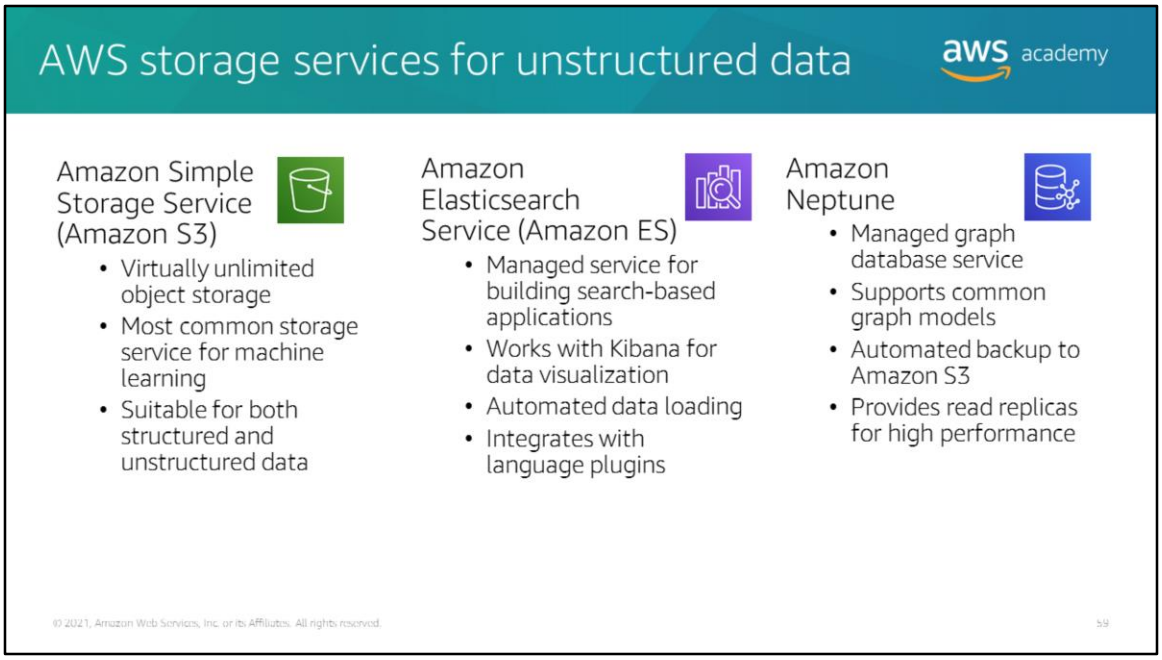
In this section, you will learn how to store and visualize unstructured data.



Structured data is organized in a defined schema. An employee record with a defined set of fields is an example of structured data. In addition to a set of fields, schemas also specify the type of data that is allowed in the fields. You can think of structured data as data you would store in a traditional database management system (DBMS) that supports structured query language (SQL).

Unstructured data does not follow a set schema. Some records might have four fields, and other records might have 14 fields. Unstructured data can include many different data types, such as text, numeric, or even binary objects.

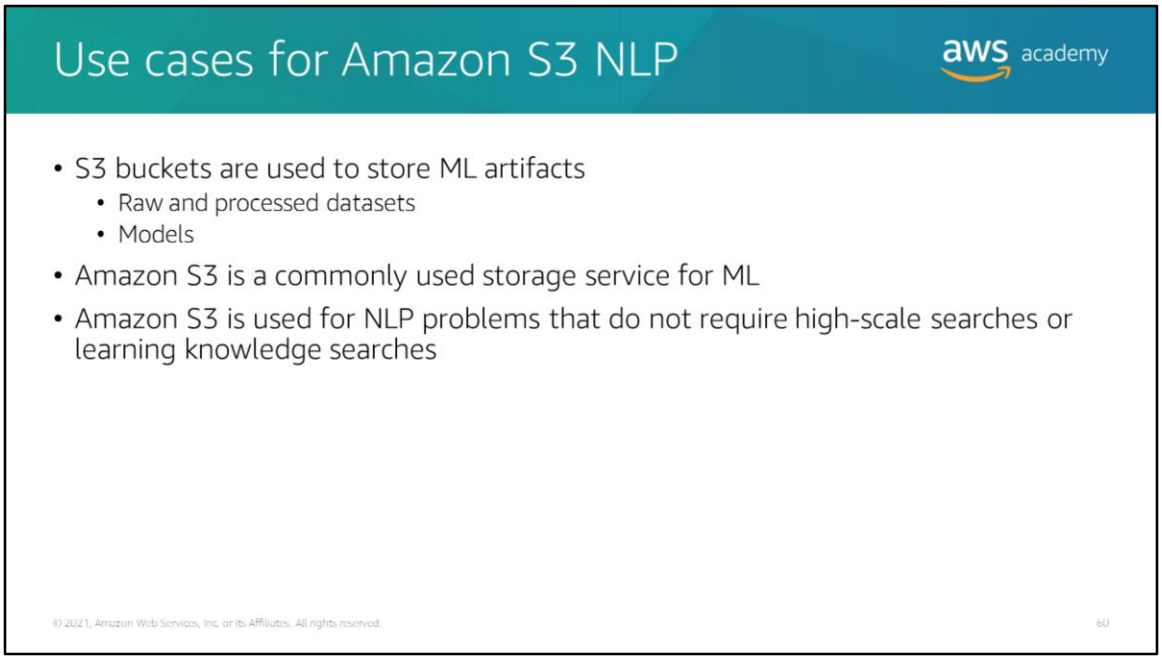
Many NLP datasets are unstructured. You will now learn about three AWS services that you can use to store unstructured data.



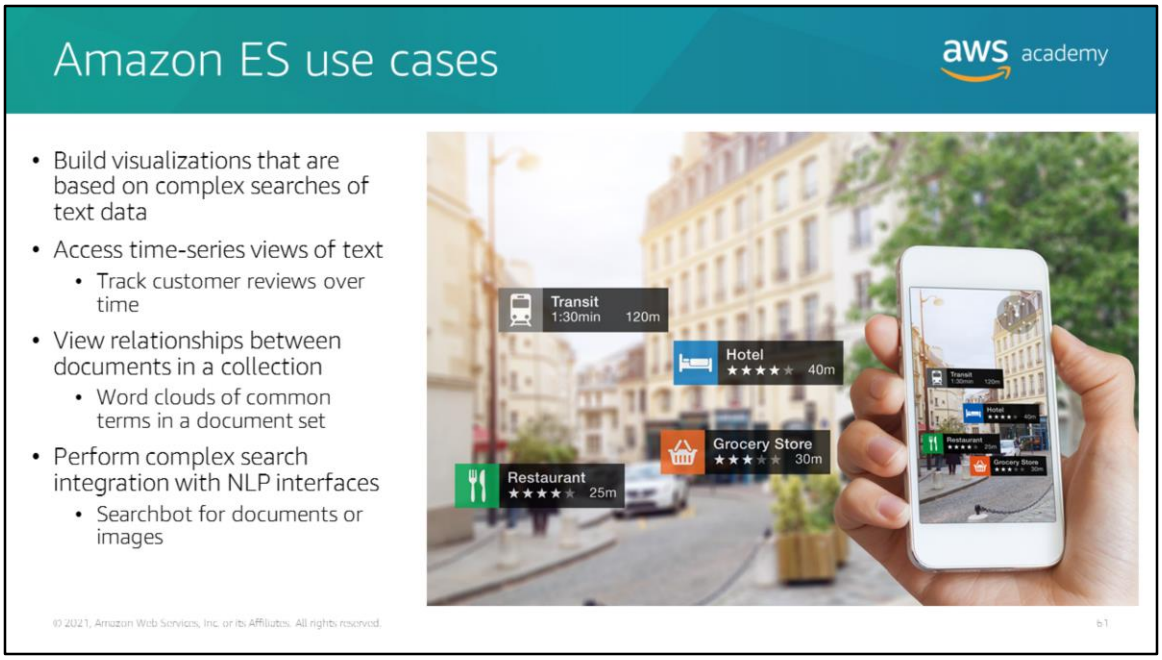
Amazon Simple Storage Service (Amazon S3) offers virtually unlimited object storage. The storage capacity and reliability of Amazon S3 makes it an ideal storage solution for most NLP applications. You can use Amazon S3 to store both structured and unstructured data for ML solutions. Structured data for traditional database applications will usually use another storage solution, such as Amazon Relational Database Service (Amazon RDS).

Search-based applications gather data together across large datasets to surface information. You can build complex search applications with Amazon Elasticsearch Service (Amazon ES). The service works with Kibana, which is an open-source visualization system. Most search-based applications depend on highly dynamic data, which means that you need a way to update in near real time. Amazon ES also works with services (like Amazon Kinesis Data Streams and Amazon DynamoDB) to address the need for near-continuous updates to data.

Knowledge graphs are a special form of search-based application. Amazon Neptune provides a database service that is designed to work with commonly used graph models. The service provides automated backup to Amazon S3. When you set up an Amazon Neptune cluster, you can include read replicas, which provide higher performance for read-intensive applications.



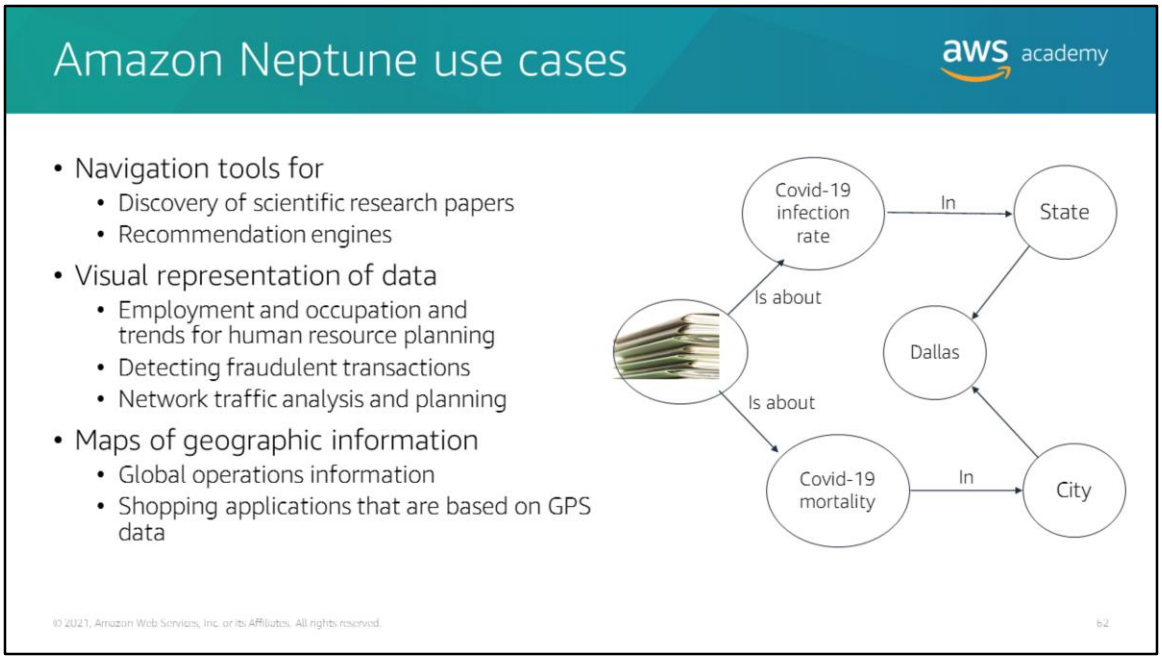
Amazon S3 is the most commonly used storage service for machine learning applications. Amazon S3 is often used as a storage location for training data and other model artifacts. Although Amazon S3 is sufficient for the many NLP applications, it might not be appropriate for scenarios that require high-performance searches over large datasets.



Search applications are one of the most common forms of NLP problems. The application must be able to understand the context of the search to return meaningful search results. For example, if you search for the word *apple* without any knowledge of context, the search might return records for a computer company, a fruit, and a music label.

Search applications are becoming common, as these three examples illustrate:

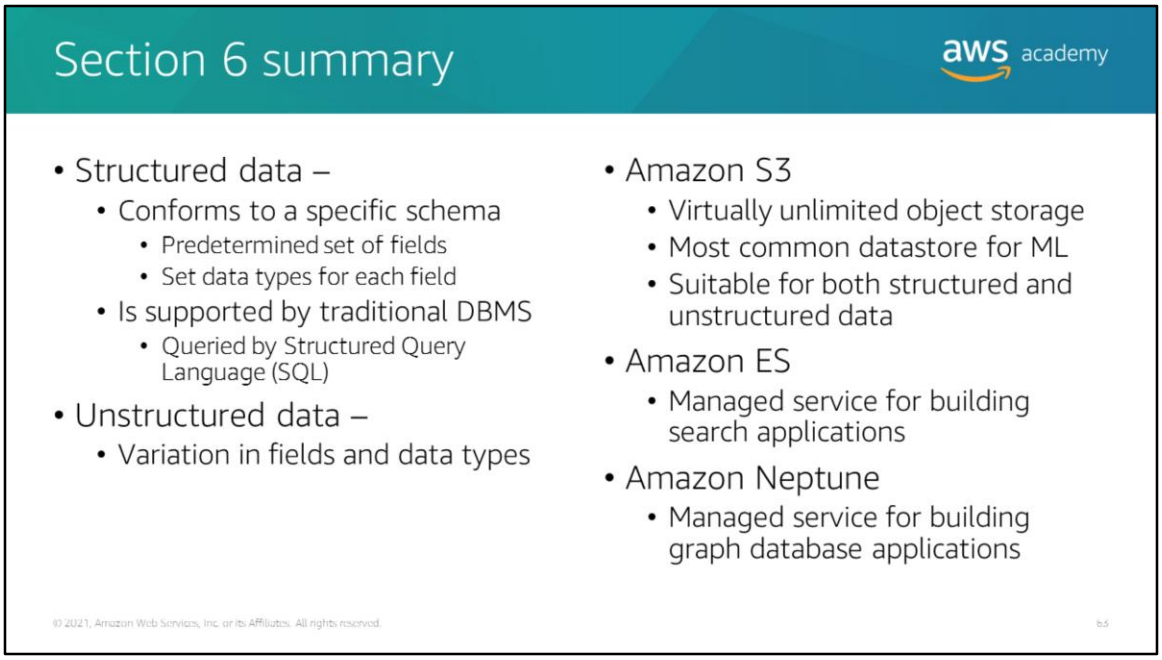
- An application that shows text data as a time series – For example, displaying customer reviews for a product over time.
- Visualizations of relationships in a collection of text data – For example, a word cloud of common terms in a collection of scientific journal articles.
- A combination of search with an NLP interface – For example, a bot based on searching documents or images.



All database systems have some capability to model relationships. However, graph databases are better at handling datasets where the relationships are complex or might not be fully understood in the design stage. With the rapid growth of data in the era of cloud computing, graph databases have emerged to explore relationships so that users can acquire new knowledge and insights.

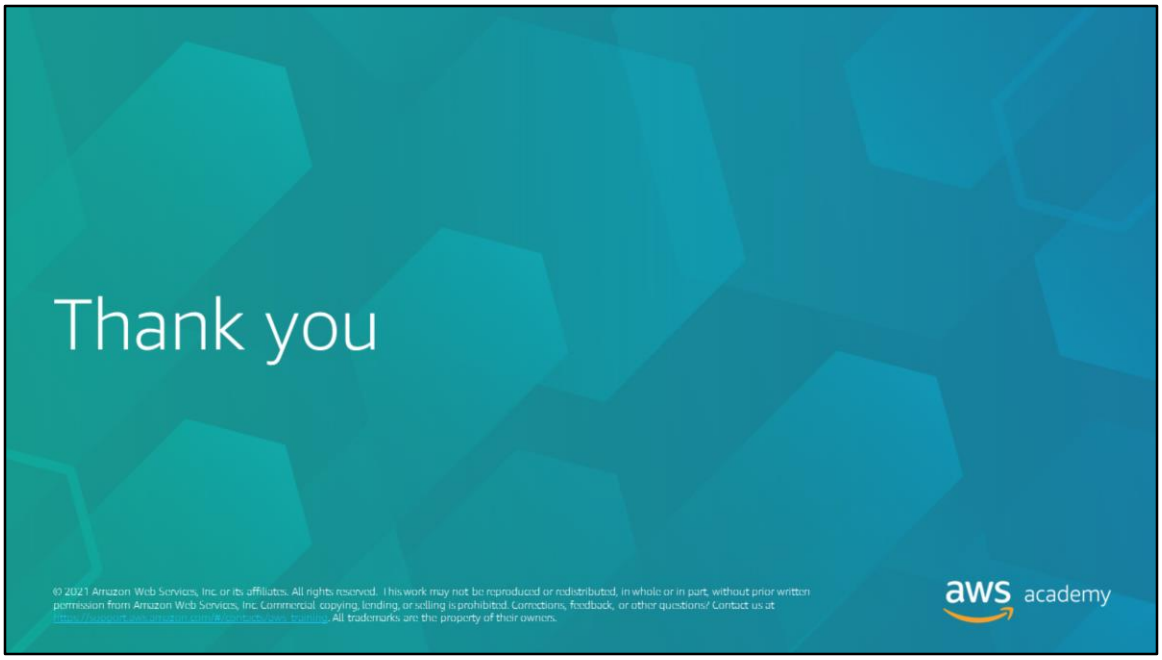
The following examples illustrate use cases for Amazon Neptune:

- Navigation tools – An example could include researchers who are looking for connections between scientific papers and recommendation engines.
- Visual representations of data – Examples include identifying employment and occupational trends for human-resources planning, detecting fraud, and analyzing network traffic.
- Geographic visualizations – Possible examples include global operations for a multinational company, and building shopping applications based on GPS data.



In this section, you learned about the differences between structured and unstructured data. You also learned about three different AWS services that you can use to store data for NLP problems.

- The first storage service is Amazon S3, which is integrated with many of the Amazon Machine Learning services. It provides virtually unlimited object storage, and 11 9s of reliability.
- Amazon ES is a managed service that you can use to build applications that require complex search patterns. You can build search applications with both static and streaming data.
- Finally, you learned about Amazon Neptune, which is designed to support applications that are based on graph databases. Graph databases consider the relationships between the data to be as important as the data itself. Graph databases are useful for handling data with complex relationships, or even relationships that are unknown in the design and development stage.



Thank you for completing this module. In the next module, you will learn how to implement sentiment analysis.

