

3.3 - Build a conversational language understanding model

- [Overview](#)
- [Learning objectives](#)
- [Prerequisites](#)
- [Introduction](#)
- [Understand prebuilt capabilities of the Azure AI Language service](#)
 - [Pre-configured features](#)
 - [Summarization](#)
 - [Named entity recognition](#)
 - [Personally identifiable information \(PII\) detection](#)
 - [Key phrase extraction](#)
 - [Sentiment analysis](#)
 - [Language detection](#)
 - [Learned features](#)
 - [Conversational language understanding \(CLU\)](#)
 - [Custom named entity recognition](#)
 - [Custom text classification](#)
 - [Question answering](#)
- [Understand resources for building a conversational language understanding model](#)
 - [Build your model](#)
 - [Use Language Studio](#)
 - [Use the REST API](#)
 - [Authentication](#)
 - [Request deployment](#)
 - [Get deployment status](#)
 - [Query your model](#)
 - [Query using SDKs](#)
 - [Query using the REST API](#)
 - [Sample response](#)
- [Define intents, utterances, and entities](#)
- [Use patterns to differentiate similar utterances](#)
- [Use pre-built entity components](#)
- [Train, test, publish, and review a conversational language understanding model](#)
- [Exercise - Build an Azure AI services conversational language understanding model](#)
 - [Provision an Azure AI Language resource](#)
 - [Create a conversational language understanding project](#)
 - [Create intents](#)
 - [Label each intent with sample utterances](#)
 - [Train and test the model](#)
 - [Add entities](#)
 - [Add a learned entity](#)

- [Add a _list_ entity.](#)
 - [Add a _prebuilt_ entity.](#)
 - [Retrain the model](#)
 - [Use the model from a client app](#)
 - [Prepare to develop an app in Visual Studio Code](#)
 - [Configure your application](#)
 - [Add code to the application](#)
 - [Clean up resources](#)
 - [More information](#)
 - [Knowledge Check](#)
 - [Summary](#)
-

Overview

The Azure AI Language **conversational language understanding** service (CLU) enables you to train a model that apps can use to extract meaning from natural language.

Learning objectives

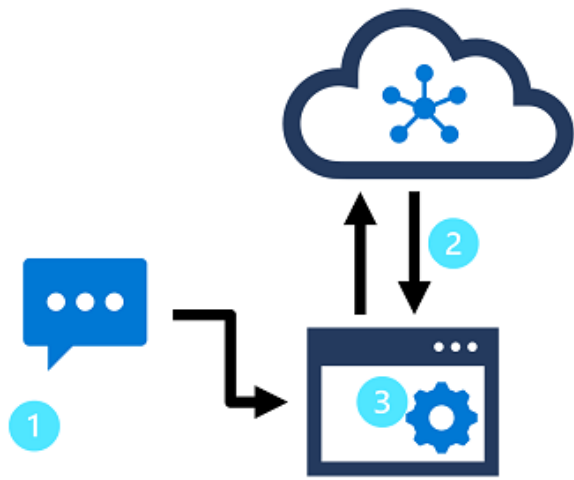
After completing this module, you'll be able to:

- Provision Azure resources for Azure AI Language resource
 - Define intents, utterances, and entities
 - Use patterns to differentiate similar utterances
 - Use pre-built entity components
 - Train, test, publish, and review an Azure AI Language model
-

Introduction

Natural language processing (NLP) is a common AI problem in which software must be able to work with text or speech in the natural language form that a human user would write or speak. Within the broader area of NLP, *natural language understanding* (NLU) deals with the problem of determining semantic meaning from natural language - usually by using a trained language model.

A common design pattern for a natural language understanding solution looks like this:



In this design pattern:

1. An app accepts natural language input from a user.
2. A language model is used to determine semantic meaning (the user's *intent*).
3. The app performs an appropriate action.

Azure AI Language enables developers to build apps based on language models that can be trained with a relatively small number of samples to discern a user's intended meaning.

In this module, you'll learn how to use the service to create a **natural language understanding app** using Azure AI Language.

After completing this module, you'll be able to:

- Provision an Azure AI Language resource.
- Define intents, entities, and utterances.
- Use patterns to differentiate similar utterances.
- Use pre-built entity components.
- Train, test, publish, and review a model.

Understand prebuilt capabilities of the Azure AI Language service

The Azure AI Language service provides various features for understanding human language. You can use each feature to better communicate with users, better understand incoming communication, or use them together to provide more insight into what the user is saying, intending, and asking about.

Azure AI Language service features fall into two categories: **Pre-configured features, and Learned features**. Learned features require building and training a model to correctly predict appropriate labels, which is covered in upcoming units of this module.

This unit covers most of the capabilities of the Azure AI Language service, but head over to the [Azure AI Language service documentation](#) for a full list, including quick-starts and a full explanation of everything available.

Using these features in your app requires sending your query to the appropriate endpoint. The endpoint used to query a specific feature varies, but all of them are prefixed with the Azure AI Language resource you created in your Azure account, either **when building your REST request or defining your client using an SDK**. Examples of each can be found in the next unit.

Pre-configured features

The Azure AI Language service provides certain features without any model labeling or training. Once you create your resource, you can send your data and use the returned results within your app.

The following features are all pre-configured.

Summarization

Summarization is available for both documents and conversations, and will summarize the text into key sentences that are predicted to encapsulate the input's meaning.

Named entity recognition

Named entity recognition can extract and identify entities, such as people, places, or companies, allowing your app to recognize different types of entities for improved natural language responses. For example, given the text "The waterfront pier is my favorite Seattle attraction", *Seattle* would be identified and categorized as a location.

Personally identifiable information (PII) detection

PII detection allows you to identify, categorize, and redact information that could be considered sensitive, such as email addresses, home addresses, IP addresses, names, and protected health information. For example, if the text "email@contoso.com" was included in the query, the entire email address can be identified and redacted.

Key phrase extraction

Key phrase extraction is a feature that quickly pulls the main concepts out of the provided text. For example, given the text "Text Analytics is one of the features in Azure AI Services.", the service would extract "*Azure AI Services*" and "*Text Analytics*".

Sentiment analysis

Sentiment analysis identifies how positive or negative a string or document is. For example, given the text "Great hotel. Close to plenty of food and attractions we could walk to", the service would identify that as *positive* with a relatively high confidence score.

Language detection

Language detection takes one or more documents, and identifies the language for each. For example, if the text of one of the documents was "Bonjour", the service would identify that as *French*.

Learned features

Learned features require you to label data, train, and deploy your model to make it available to use in your application. These features allow you to customize what information is predicted or extracted.

Note: **Quality of data greatly impacts the model's accuracy.** Be intentional about what data is used, how well it is tagged or labeled, and how varied the training data is. For details, see [recommendations for labeling data](#), which includes valuable guidelines for tagging data. Also see the [evaluation metrics](#) that can assist in learning where your model needs improvement.

Conversational language understanding (CLU)

CLU is one of the core custom features offered by Azure AI Language. **CLU helps users to build custom natural language understanding models to predict overall intent and extract important information from incoming utterances.** CLU does require data to be tagged by the user to teach it how to predict intents and entities accurately.

The exercise in this module will be building a CLU model and using it in your app.

Custom named entity recognition

Custom entity recognition takes custom labeled data and extracts specified entities from unstructured text. For example, if you have various contract documents that you want to extract involved parties from, you can train a model to recognize how to predict them.

Custom text classification

Custom text classification enables users to classify text or documents as custom defined groups. For example, you can train a model to look at news articles and identify the category they should fall into, such as *News* or *Entertainment*.

Question answering

Question answering is a mostly pre-configured feature that provides answers to questions provided as input. The data to answer these questions comes from documents like FAQs or manuals.

For example, say you want to make a virtual chat assistant on your company website to answer common questions. You could use a company FAQ as the input document to create the question and answer pairs. Once deployed, your chat assistant can pass input questions to the service, and get the answers as a result.

For a complete list of capabilities and how to use them, see the Azure AI Language [documentation](#).

Understand resources for building a conversational language understanding model

To use the Language Understanding service to develop a NLP solution, you'll need to create a Language resource in Azure. That resource will be used for both authoring your model and processing prediction requests from client applications.

Tip: This module's lab covers building a model for conversational language understanding. For more focused modules on custom text classification and custom named entity recognition, see the custom solution modules in the [Develop natural language solutions](#) learning path.

Build your model

For features that require a model for prediction, you'll need to build, train and deploy that model before using it to make a prediction. This building and training will teach the Azure AI Language service what to look for.

First, you'll need to create your Azure AI Language resource in the [Azure portal](#). Then:

1. Search for **Azure AI services**.
2. Find and select **Language Service**.
3. Select **Create** under the **Language Service**.
4. Fill out the necessary details, choosing the region closest to you geographically (for best performance) and giving it a unique name.

Once that resource has been created, you'll need a key and the endpoint. You can find that on the left side under **Keys and Endpoint** of the resource overview page.


Use Language Studio

For a more visual method of building, training, and deploying your model, you can use [Language Studio](#) to achieve each of these steps. On the main page, you can choose to create a **Conversational language understanding** project. Once the project is created, then go through the same process as above to build, train, and deploy your model.

Language Studio

- ☆ Featured
- 📄 Extract information
- ☰ Classify text
- 🗨️ Understand questions and conversational language
- 📄 Summarize text
- 🌐 Translate text


Retrieve the most appropriate answer to questions using question answering (CQA) or classify intents and extract entities for conversational utterances using conversational language understanding (CLU). Use orchestration workflow to create one project that routes queries between multiple CQA and CLU projects. [Learn more about understanding conversational language.](#)



Answer questions

Use the prebuilt question answering API to get answers to questions over unstructured text.

[Try it out](#)




Custom question answering

Next generation of QnAMaker

Customize the list of questions and answers extracted from your content corpus to provide a conversational experience that suits your needs.

[Open Custom question answering](#)



Conversational language understanding

Next generation of LUIS

Classify utterances into intents and extract information with entities to build natural language into apps, bots, and IoT devices.

[Open Conversational language understanding](#)

The lab in this module will walk through using Language Studio to build your model. If you'd like to learn more, see the [Language Studio quickstart](#)

Use the REST API

One way to build your model is through the REST API. The pattern would be to create your project, import data, train, deploy, then use your model.

These tasks are done asynchronously; you'll need to submit a request to the appropriate URI for each step, and then send another request to get the status of that job.

For example, if you want to deploy a model for a conversational language understanding project, you'd submit the deployment job, and then check on the deployment job status.

Authentication

For each call to your Azure AI Language resource, you authenticate the request by providing the following header.

Key	Value
Ocp-Apim-Subscription-Key	The key to your resource

Request deployment

Submit a **POST** request to the following endpoint.

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMENT-NAME}?api-version={API-VERSION}
```

Placeholder	Value	Example
{ENDPOINT}	The endpoint of your Azure AI Language resource	https://<your-subdomain>.cognitiveservices.azure.com
{PROJECT-NAME}	The name for your project. This value is case-sensitive	myProject
{DEPLOYMENT-NAME}	The name for your deployment. This value is case-sensitive	staging
{API-VERSION}	The version of the API you're calling	2022-05-01

Include the following `body` with your request.

```
{
  "trainedModelLabel": "{MODEL-NAME}",
}
```

Placeholder	Value
{MODEL-NAME}	The model name that will be assigned to your deployment. This value is case-sensitive.

Successfully submitting your request will receive a `202` response, with a response header of `operation-location`. This header will have a URL with which to request the status, formatted like this:

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMENT-NAME}/jobs/{JOB-ID}?api-version={API-VERSION}
```

Get deployment status

Submit a **GET** request to the URL from the response header above. The values will already be filled out based on the initial deployment request.

```
{ENDPOINT}/language/authoring/analyze-conversations/projects/{PROJECT-NAME}/deployments/{DEPLOYMENT-NAME}/jobs/{JOB-ID}?api-version={API-VERSION}
```

Placeholder	Value
{ENDPOINT}	The endpoint for authenticating your API request
{PROJECT-NAME}	The name for your project (case-sensitive)
{DEPLOYMENT-NAME}	The name for your deployment (case-sensitive)
{JOB-ID}	The ID for locating your model's training status, found in the header value detailed above in the deployment request
{API-VERSION}	The version of the API you're calling

The response body will give the deployment status details. The `status` field will have the value of *succeeded* when the deployment is complete.

```
{
  "jobId": "{JOB-ID}",
  "createdDateTime": "String",
  "lastUpdatedDateTime": "String",
  "expirationDateTime": "String",
  "status": "running"
}
```

For a full walkthrough of each step with example requests, see the [conversational understanding quickstart](#).

Query your model

To query your model for a prediction, you can use SDKs in C# or Python, or use the REST API.

Query using SDKs

To query your model using an SDK, you first need to create your client. Once you have your client, you then use it to call the appropriate endpoint.

C#

```
var languageClient = new TextAnalyticsClient(endpoint, credentials);
```



```
var response = languageClient.ExtractKeyPhrases(document);
```

Python

```
language_client = TextAnalyticsClient(
    endpoint=endpoint,
    credential=credentials)
response = language_client.extract_key_phrases(documents = documents)[0]
```

Other language features, such as the conversational language understanding, require the request be built and sent differently.

C#

```
var data = new
{
    analysisInput = new
    {
        conversationItem = new
        {
            text = userText,
            id = "1",
            participantId = "1",
        }
    },
    parameters = new
    {
        projectName,
        deploymentName,
        // Use Utf16CodeUnit for strings in .NET.
        stringIndexType = "Utf16CodeUnit",
    },
    kind = "Conversation",
};
Response response = await client.AnalyzeConversationAsync(RequestContent.Create(data));
```

Python

```
result = client.analyze_conversation(
    task={
        "kind": "Conversation",
        "analysisInput": {
            "conversationItem": {
                "participantId": "1",
                "id": "1",
                "modality": "text",
                "language": "en",
                "text": query
            },

```

```

        "isLoggingEnabled": False
    },
    "parameters": {
        "projectName": cls_project,
        "deploymentName": deployment_slot,
        "verbose": True
    }
}
)

```

Query using the REST API

To query your model using REST, create a **POST** request to the appropriate URL with the appropriate body specified. For built in features such as language detection or sentiment analysis, you'll query the `analyze-text` endpoint.

Tip: **Remember each request needs to be authenticated with your Azure AI Language resource key** in the `Ocp-Apim-Subscription-Key` header

```
{ENDPOINT}/language/:analyze-text?api-version={API-VERSION}
```

Placeholder	Value
{ENDPOINT}	The endpoint for authenticating your API request
{API-VERSION}	The version of the API you're calling

Within the body of that request, you must specify the `kind` parameter, which tells the service what type of language understanding you're requesting.

If you want to detect the language, for example, the JSON body would look something like the following.

```

{
  "kind": "LanguageDetection",
  "parameters": {
    "modelVersion": "latest"
  },
  "analysisInput": {
    "documents": [
      {
        "id": "1",
        "text": "This is a document written in English."
      }
    ]
  }
}

```

Other language features, such as the conversational language understanding, require the request be routed to a different endpoint. For example, the **conversational language understanding request would**

be sent to the following.

```
{ENDPOINT}/language/:analyze-conversations?api-version={API-VERSION}
```

Placeholder	Value
{ENDPOINT}	The endpoint for authenticating your API request
{API-VERSION}	The version of the API you're calling

That request would include a JSON body similar to the following.

```
{
  "kind": "Conversation",
  "analysisInput": {
    "conversationItem": {
      "id": "1",
      "participantId": "1",
      "text": "Sample text"
    }
  },
  "parameters": {
    "projectName": "{PROJECT-NAME}",
    "deploymentName": "{DEPLOYMENT-NAME}",
    "stringIndexType": "TextElement_V8"
  }
}
```

Placeholder	Value
{PROJECT-NAME}	The name of the project where you built your model
{DEPLOYMENT-NAME}	The name of your deployment

Sample response

The query response from an SDK will in the object returned, which varies depending on the feature (such as in `response.key_phrases` or `response.Value`). The REST API will return JSON that would be similar to the following.

```
{
  "kind": "KeyPhraseExtractionResults",
  "results": {
    "documents": [{
      "id": "1",
      "keyPhrases": ["modern medical office", "Dr. Smith", "great staff"],
      "warnings": []
    }],
    "errors": [],
  }
}
```

```
    "modelVersion": "{VERSION}"
  }
}
```

For other models like conversational language understanding, a sample response to your query would be similar to the following.

```
{
  "kind": "ConversationResult",
  "result": {
    "query": "String",
    "prediction": {
      "topIntent": "intent1",
      "projectKind": "Conversation",
      "intents": [
        {
          "category": "intent1",
          "confidenceScore": 1
        },
        {
          "category": "intent2",
          "confidenceScore": 0
        }
      ],
      "entities": [
        {
          "category": "entity1",
          "text": "text",
          "offset": 7,
          "length": 4,
          "confidenceScore": 1
        }
      ]
    }
  }
}
```

The SDKs for both Python and C# return JSON that is very similar to the REST response.

For full documentation on features, including examples and how-to guides, see the [Azure AI Language documentation](#) documentation pages.

Define intents, utterances, and entities

Utterances are the **phrases that a user might enter** when interacting with an application that uses your language model. An *intent* represents a task or action the user **wants to perform**, or more simply

the *meaning* of an utterance. **You create a model by defining intents and associating them with one or more utterances.**

For example, consider the following list of intents and associated utterances:

- **GetTime:**
 - "What time is it?"
 - "What is the time?"
 - "Tell me the time"
- **GetWeather:**
 - "What is the weather forecast?"
 - "Do I need an umbrella?"
 - "Will it snow?"
- **TurnOnDevice**
 - "Turn the light on."
 - "Switch on the light."
 - "Turn on the fan"
- **None:**
 - "Hello"
 - "Goodbye"

In your model, **you must define the intents that you want your model to understand**, so spend some time considering the *domain* your model must support and the kinds of actions or information that users might request. In addition to the intents that you define, every model includes a **None intent** that you should use to explicitly identify utterances that a user might submit, but for which there is no specific action required (for example, conversational greetings like "hello") or that **fall outside of the scope of the domain** for this model.

After you've identified the intents your model must support, it's important to capture various different example utterances for each intent. Collect utterances that you think users will enter; including utterances **meaning the same thing but that are constructed in different ways**. Keep these guidelines in mind:

- Capture multiple different examples, or alternative ways of saying the same thing
- Vary the length of the utterances from short, to medium, to long
- Vary the location of the *noun* or *subject* of the utterance. Place it at the beginning, the end, or somewhere in between
- Use correct grammar and incorrect grammar in different utterances to offer good training data examples
- The precision, consistency and completeness of your labeled data are key factors to determining model performance.
 - Label **precisely**: Label each entity to its right type always. Only include what you want extracted, avoid unnecessary data in your labels.
 - Label **consistently**: The same entity should have the same label across all the utterances.
 - Label **completely**: Label all the instances of the entity in all your utterances.

Entities are used to add specific context to intents. For example, you might define a **TurnOnDevice** intent that can be applied to multiple devices, and use entities to define the different devices.

Consider the following utterances, intents, and entities:

Utterance	Intent	Entities
What is the time?	GetTime	
What time is it in <i>London</i> ?	GetTime	Location (London)
What's the weather forecast for <i>Paris</i> ?	GetWeather	Location (Paris)
Will I need an umbrella <i>tonight</i> ?	GetWeather	Time (tonight)
What's the forecast for <i>Seattle tomorrow</i> ?	GetWeather	Location (Seattle), Time (tomorrow)
Turn the <i>light</i> on.	TurnOnDevice	Device (light)
Switch on the <i>fan</i> .	TurnOnDevice	Device (fan)

You can split entities into a few different component types:

- **Learned** entities are the most flexible kind of entity, and should be used in most cases. You define a learned component with a suitable name, and then associate words or phrases with it in training utterances. When you train your model, it learns to match the appropriate elements in the utterances with the entity.
- **List** entities are useful when you need an entity with a specific set of possible values - for example, days of the week. You can include synonyms in a list entity definition, so you could define a **DayOfWeek** entity that includes the values "Sunday", "Monday", "Tuesday", and so on; each with synonyms like "Sun", "Mon", "Tue", and so on.
- **Prebuilt** entities are useful for common types such as numbers, datetimes, and names. For example, when prebuilt components are added, you will **automatically detect values** such as "6" or organizations such as "Microsoft". You can see this article for a list of [supported prebuilt entities](#).

Use patterns to differentiate similar utterances

In some cases, a model might contain **multiple intents for which utterances are likely to be similar**. You can use the pattern of utterances to disambiguate the intents while minimizing the number of sample utterances.

For example, consider the following utterances:

- "Turn on the kitchen light"
- "Is the kitchen light on?"
- "Turn off the kitchen light"

These utterances are syntactically similar, with only a few differences in words or punctuation. However, they represent three different intents (which could be named **TurnOnDevice**, **GetDeviceStatus**, and **TurnOffDevice**).

Additionally, the intents could apply to a wide range of entity values. In addition to "kitchen light", the intent could apply to "living room light", "television", or any other device that the model might need to support.

To correctly train your model, **provide a handful of examples of each intent** that specify the different formats of utterances.

- **TurnOnDevice:**
 - "Turn on the {DeviceName}"
 - "Switch on the {DeviceName}"
 - "Turn the {DeviceName} on"
- **GetDeviceStatus:**
 - "Is the {DeviceName} on[?]"
- **TurnOffDevice:**
 - "Turn the {DeviceName} off"
 - "Switch off the {DeviceName}"
 - "Turn off the {DeviceName}"

When you teach your model with each different type of utterance, the Azure AI Language service can learn how to categorize intents correctly based off format and punctuation.

Use pre-built entity components

You can create your own language models by defining all the intents and utterances it requires, but often you can use prebuilt components to **detect common entities such as numbers, emails, URLs, or choices**.

For a full list of prebuilt entities the Azure AI Language service can detect, see the list of [supported prebuilt entity components](#).

Using prebuilt components allows you to let the Azure AI Language service automatically detect the specified type of entity, and not have to train your model with examples of that entity.

To add a prebuilt component, you can create an entity in your project, then select **Add new prebuilt** to that entity to detect certain entities.

Azure AI | Language Studio

Language Studio > Conversational Language Understanding projects > Clock - Schema definition > Location - Entity components

Location

Each entity is made of multiple components that you can define. You can define one or multiple components. If you have more than one component, the logic that combines their responses is defined in the Overlap method tab.

Entity components Entity Options

Edit Delete Required component ⓘ

Learned Not Required

The Learned component uses the entity labels you label your utterances with to train a machine learned model. The model learns to predict where the entity is based on the context within the utterance. This component is only defined if you add labels by [labeling utterances](#). If you do not label any utterances with this entity, it will not have a Learned component.

Prebuilt (0) Not Required

A prebuilt component gets your entity to extract common types such as numbers, dates, times, and others.

Prebuilt name ▾	Description ▾
+ Add new prebuilt	

No items found.

Regular expression (0) Not Required

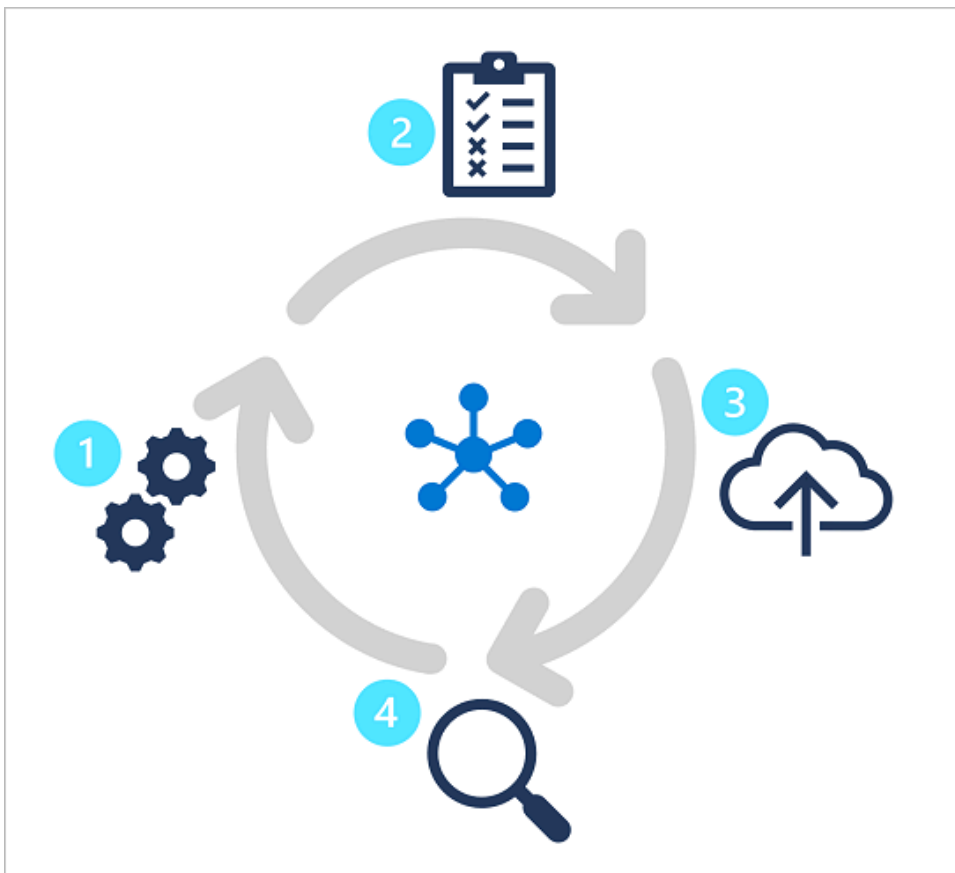
A regex component matches regular expressions for common patterns. You can associate a key to each expression.

Regex key ↑ ▾	Expression ▾
+ Add expression	

You can have up to **five prebuilt components per entity**. Using prebuilt model elements can significantly reduce the time it takes to develop a conversational language understanding solution.

Train, test, publish, and review a conversational language understanding model

Creating a model is an iterative process with the following activities:



1. Train a model to **learn intents and entities from sample utterances**.
2. Test the model interactively or using a testing dataset with known labels
3. Deploy a trained model to a public endpoint so client apps can use it
4. Review predictions and **iterate on utterances** to train your model

By following this iterative approach, you can improve the language model over time based on user input, helping you develop solutions that reflect the way users indicate their intents using natural language.

Exercise - Build an Azure AI services conversational language understanding model

NOTE: The conversational language understanding feature of the Azure AI Language service is currently in preview, and subject to change. In some cases, model training may fail - if this happens, try again.

The Azure AI Language service enables you to define a *conversational language understanding* model that applications can use to interpret natural language input from users, predict the users *intent* (what they want to achieve), and identify any *entities* to which the intent should be applied.

For example, a conversational language model for a clock application might be expected to process input such as:

What is the time in London?

This kind of input is an example of an *utterance* (something a user might say or type), for which the desired *intent* is to get the time in a specific location (an *entity*); in this case, London.

NOTE The task of a conversational language model is to predict the user's intent and identify any entities to which the intent applies. It is not the job of a conversational language model to actually perform the actions required to satisfy the intent. For example, a clock application can use a conversational language model to discern that the user wants to know the time in London; but the client application itself must then implement the logic to determine the correct time and present it to the user.

Provision an Azure AI Language resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource in your Azure subscription.

1. Open the Azure portal at <https://portal.azure.com>, and sign in using the Microsoft account associated with your Azure subscription.
2. In the search field at the top, search for **Azure AI services**. Then, in the results, select **Create** under **Language Service**.
3. Select **Continue to create your resource**.
4. Provision the resource using the following settings:
 - **Subscription:** *Your Azure subscription.*
 - **Resource group:** *Choose or create a resource group.*
 - **Region:** *Choose any available region*
 - **Name:** *Enter a unique name.*
 - **Pricing tier:** Select **F0** (free), or **S** (standard) if F is not available.
 - **Responsible AI Notice:** Agree.
5. Select **Review + create**, then select **Create** to provision the resource.
6. Wait for deployment to complete, and then go to the deployed resource.
7. View the **Keys and Endpoint** page. You will need the information on this page later in the exercise.

Create a conversational language understanding project

Now that you have created an authoring resource, you can use it to create a conversational language understanding project.

1. In a new browser tab, open the Azure AI Language Studio portal at <https://language.cognitive.azure.com/> and sign in using the Microsoft account associated with your Azure subscription.
2. If prompted to choose a Language resource, select the following settings:
 - **Azure Directory:** The Azure directory containing your subscription.
 - **Azure subscription:** Your Azure subscription.
 - **Resource type:** Language.
 - **Language resource:** The Azure AI Language resource you created previously.

If you are not prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

1. On the bar at the top of the page, select the **Settings (⚙)** button.
2. On the **Settings** page, view the **Resources** tab.
3. Select the language resource you just created, and click **Switch resource**.
4. At the top of the page, click **Language Studio** to return to the Language Studio home page

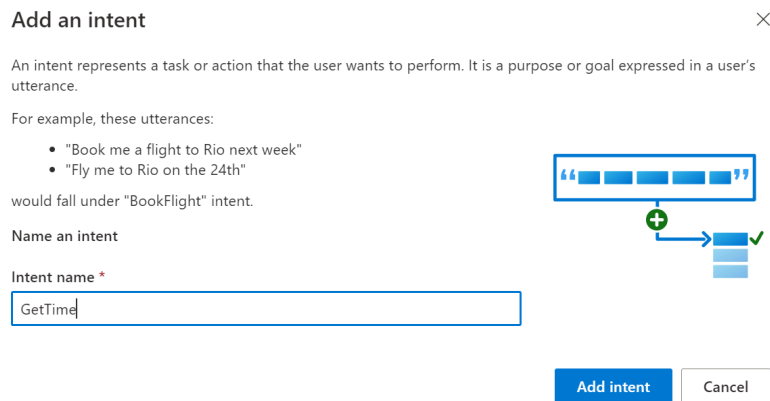
- At the top of the portal, in the **Create new** menu, select **Conversational language understanding**.
- In the **Create a project** dialog box, on the **Enter basic information** page, enter the following details and then select **Next**:
 - Name:** `Clock`
 - Utterances primary language:** English
 - Enable multiple languages in project?:** *Unselected*
 - Description:** Natural language clock
- On the **Review and finish** page, select **Create**.

Create intents

The first thing we'll do in the new project is to define some intents. The model will ultimately predict which of these intents a user is requesting when submitting a natural language utterance.

Tip: When working on your project, if some tips are displayed, read them and select **Got it** to dismiss them, or select **Skip all**.

- On the **Schema definition** page, on the **Intents** tab, select **+ Add** to add a new intent named `GetTime`.



Add an intent

An intent represents a task or action that the user wants to perform. It is a purpose or goal expressed in a user's utterance.

For example, these utterances:

- "Book me a flight to Rio next week"
- "Fly me to Rio on the 24th"

would fall under "BookFlight" intent.

Name an intent

Intent name *

`GetTime`

Add intent Cancel

- Verify that the **GetTime** intent is listed (along with the default **None** intent). Then add the following additional intents:

- `GetDay`
- `GetDate`

[Language Studio](#) > [Conversational Language Understanding projects](#) > [Clock - Schema definition](#)

Schema definition

Add intents and entities to your schema. Intents are tasks or actions the user wants to perform. Entities are terms relevant to the user's intent and can be extracted to help fulfill the user's intent.

Intents			Entities	
+ Add Delete			Search	
<input type="radio"/>	Intents ↑ ↓	Labeled utterances ↓	Entities used with this intent ↓	
<input type="radio"/>	<code>GetDate</code>	0		
<input type="radio"/>	<code>GetDay</code>	0		
<input type="radio"/>	<code>GetTime</code>	0		
<input type="radio"/>	<code>None</code>	0		

Label each intent with sample utterances

To help the model predict which intent a user is requesting, you must label each intent with some sample utterances.

1. In the pane on the left, select the **Data Labeling** page.

Tip: You can expand the pane with the >> icon to see the page names, and hide it again with the << icon.

1. Select the new **GetTime** intent and enter the utterance `what is the time?`. This adds the utterance as sample input for the intent.

2. Add the following additional utterances for the **GetTime** intent:

- `what's the time?`
- `what time is it?`
- `tell me the time`

Language Studio > Conversational Language Understanding projects > Clock - Data labeling

Training set Testing set

Save changes Upload utterance file ... Filter

We'll use these utterances to create your conversation model during training. A separate set of utterances can test the performance of your model. [Learn more about splitting data between training and testing sets.](#)

Intent	Utterance
<input checked="" type="radio"/> GetTime	tell me the time
<input type="radio"/> GetTime	what time is it?
<input type="radio"/> GetTime	what's the time?

0/500

3. Select the **GetDay** intent and add the following utterances as example input for that intent:

- `what day is it?`
- `what's the day?`
- `what is the day today?`
- `what day of the week is it?`

4. Select the **GetDate** intent and add the following utterances for it:

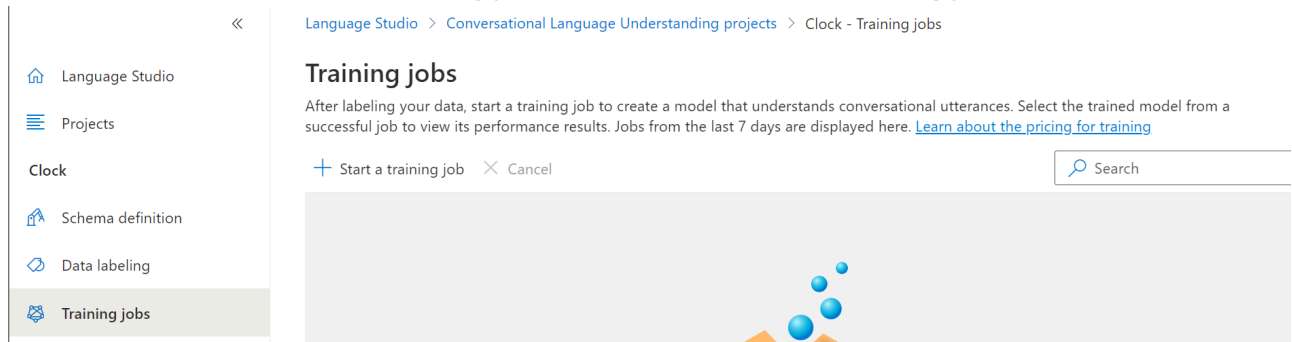
- `what date is it?`
- `what's the date?`
- `what is the date today?`
- `what's today's date?`

5. After you've added utterances for each of your intents, select **Save changes**.

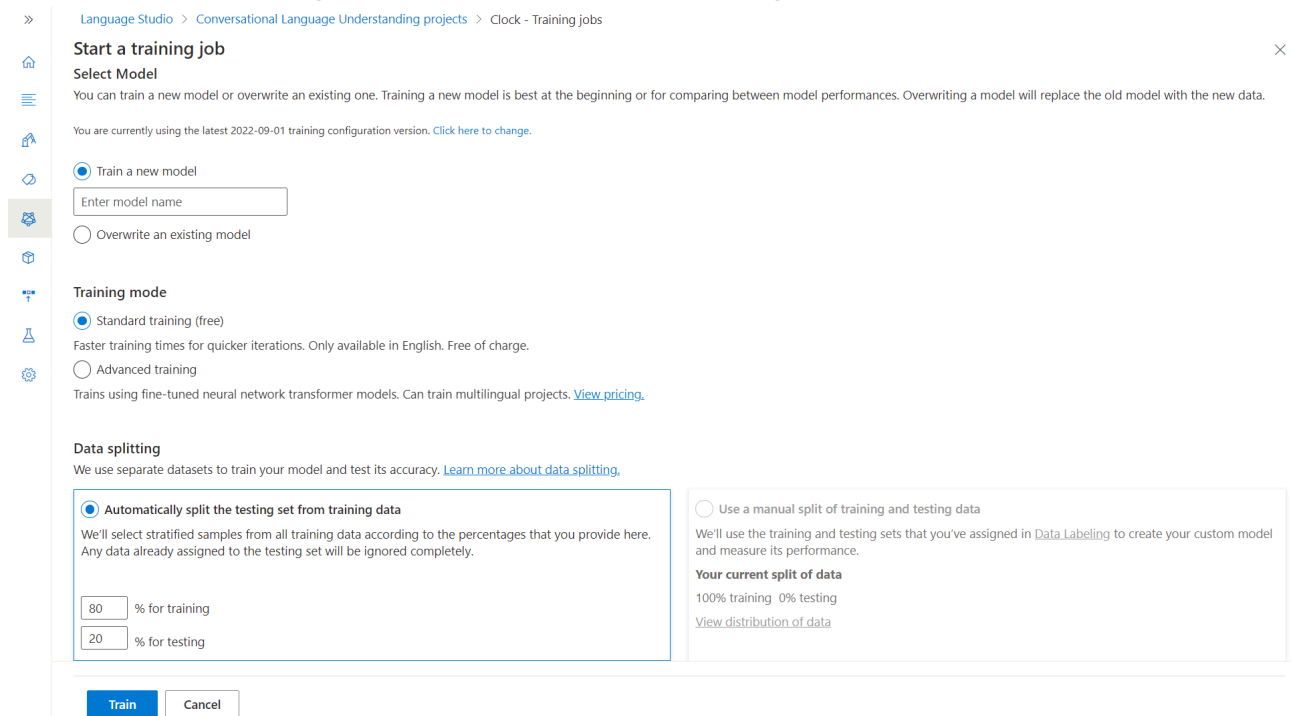
Train and test the model

Now that you've added some intents, let's train the language model and see if it can correctly predict them from user input.

1. In the pane on the left, select **Training jobs**. Then select **+ Start a training job**.



2. On the **Start a training job** dialog, select the option to train a new model, name it **Clock**.
Select **Standard training mode** and the default **Data splitting** options.

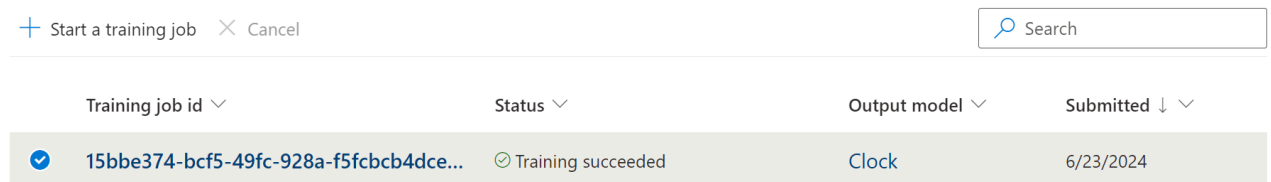


3. To begin the process of training your model, select **Train**.
4. When training is complete (which may take several minutes) the job **Status** will change to **Training succeeded**.

[Language Studio](#) > [Conversational Language Understanding projects](#) > Clock - Training jobs

Training jobs

After labeling your data, start a training job to create a model that understands conversational utterances. Select the trained model from a successful job to view its performance results. Jobs from the last 7 days are displayed here. [Learn about the pricing for training](#)






5. Select the **Model performance** page, and then select the **Clock** model. Review the overall and per-intent evaluation metrics (*precision*, *recall*, and *F1 score*) and the *confusion matrix* generated by the evaluation that was performed when training (note that due to the small number of sample utterances, not all intents may be included in the results).

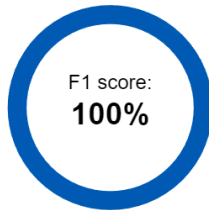
Model Type

Intents

Status:  Trained successfully

F1 score:  100%
Precision:  100%
Recall:  100%

Finished training on: 6/23/2024, 12:39:52 AM
Total training time: 0 hour(s), 1 minute(s), 10 second(s)




Training data splitting type:	Percentage
Number of training utterances:	9 (81.82%)
Number of testing utterances:	2 (18.18%)



[Learn more about how to improve your model](#)


Guidance

To improve your model's performance, review these issues and follow any recommendations.

 **Not enough intents labels in training set**
We recommend a minimum of 15 utterances per intent in your training data.

3 intents with low data.

[View Details](#)

 **Missing intents in test set.**
When the testing data lacks labeled instances for an intents, the model's test performance may become less comprehensive due to untested scenarios.

2 missing intents.

[View Details](#)

 Intents in training set are clearly distinct.

[View Details](#)

NOTE To learn more about the evaluation metrics, refer to the [documentation](#)

- Go to the **Deploying a model** page, then select **Add deployment**.
- On the **Add deployment** dialog, select **Create a new deployment name**, and then enter `production`.

8. Select the **Clock** model in the **Model** field then select **Deploy**. The deployment may take some time.

Language Studio > Conversational Language Understanding projects > Clock - Deploying a model

Deploying a model

Choose which model to deploy
Deploy your project to different regions

Deployments Regions

Add deployment Get SDK

Add deployment

Create or select an existing deployment name

You can create a new deployment name or overwrite an existing deployment name to add a trained model to them

☒ Create a new deployment name

production

☐ Overwrite an existing deployment name

By selecting an existing deployment name, you will override any deployed model to this deployment name

Assign trained model to your deployment name

Add a trained model to your selected deployment name

Model

Clock

Deployment regions

Select the regions to deploy in based on the available assigned resources.

Regions

East US - AI-LEARN-LANG-1

Deploy Cancel

9. When the model has been deployed, select the **Testing deployments** page, then select the **production** deployment in the **Deployment name** field.
10. Enter the following text in the empty textbox, and then select **Run the test**:
what's the time now?

Review the result that is returned, noting that it includes the predicted intent (which should be **GetTime**) and a confidence score that indicates the probability the model calculated for the predicted intent. The JSON tab shows the comparative confidence for each potential intent (the one with the highest confidence score is the predicted intent)

Azure AI | Language Studio

Language Studio > Conversational Language Understanding projects > Clock - Testing deployments

Testing deployments

Test a deployment by providing a sample utterance to find out what intent and entities get recognized. These are the same predictions as when making API calls against your model in code. [Learn about the info in these API requests and responses.](#)

Run the test

Deployment name

production

Enter your own text, or upload a text document

what's the time now?

Clear text box

Result JSON

Show entities cards Sort Filter

Intent

Top intent

GetDate

Confidence: 89.54%

Entities

11. Clear the text box, and then run another test with the following text:

tell me the time

Again, review the predicted intent and confidence score.

12. Try the following text:

what's the day today?

Hopefully the model predicts the **GetDay** intent.

Add entities

So far you've defined some simple utterances that map to intents. Most real applications include more complex utterances from which specific data entities must be extracted to get more context for the intent.

Add a learned entity

The most common kind of entity is a *learned* entity, in which **the model learns to identify entity values based on examples**.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+** **Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Location` and ensure that the **Learned** tab is selected. Then select **Add entity**.

Add an entity

An entity is an item or element that is relevant to the user's intent or goal.

For example, these utterances:

- "Book me a flight to Rio next week"
- "Fly me to Rio on the 24th"

"Rio" would be under "Destination" entity, and "24th" would be "Date" entity.

Name an entity

Entity name *

Location

An entity can be extracted by different methods. They can be learned through context, matched from a list, or detected by a prebuilt. Every entity in your project is composed by one or more of these methods that are defined as your entity's components.

Learned List Prebuilt Regex

The learned component uses the entity labels you label your utterances with to train a machine learned model. The model learns to predict where the entity is based on the context within the utterance.

"Book 2 business tickets from **Cairo** to **Seattle**"

source destination

Add entity

Cancel

3. After the **Location** entity has been created, return to the **Data labeling** page.

4. Select the **GetTime** intent and enter the following new example utterance:

what time is it in London?

5. When the utterance has been added, select the word **London**, and in the drop-down list that appears, select **Location** to indicate that "London" is an example of a location.

○ Intent ▾ Utterance ▾

GetTime ▾ Write your example utterance and press enter 0/500

○ GetTime what time is it in London

Search for an entity

Location

○ GetDate View in labeling pane

6. Add another example utterance for the **GetTime** intent:

Tell me the time in Paris?

7. When the utterance has been added, select the word **Paris**, and map it to the **Location** entity.

○ Intent ▾ Utterance ▾

GetTime ▾ Write your example utterance and press enter 0/500

✓ GetTime tell me the time in Paris
Loc...

8. Add another example utterance for the **GetTime** intent:

what's the time in New York?

9. When the utterance has been added, select the words **New York**, and map them to the **Location** entity.

10. Select **Save changes** to save the new utterances.

○ Intent ▾ Utterance ▾

GetTime ▾ Write your example utterance and press enter 0/500

✓ GetTime what's the time in New York?
Locati...

○ GetTime tell me the time in Paris
Loc...

○ GetTime what time is it in London?
Loca...

Add a *list* entity

In some cases, valid values for an entity can be restricted to a list of specific terms and synonyms; which can help the app identify instances of the entity in utterances.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+** **Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name **Weekday** and select the **List** entity tab. Then select **Add entity**.

Name an entity

Entity name *

Weekday

An entity can be extracted by different methods. They can be learned through context, matched from a list, or detected by a prebuilt. Every entity in your project is composed by one or more of these methods that are defined as your entity's components.

Learned **List** Prebuilt Regex

The list component represents a fixed, closed set of related words along with their synonyms. The component performs an exact text match against the list of values you provide.

Book 2 business tickets to **Arrabury Airport**

airport

The "airport" entity includes a list component of airport names mapped to their airport IDs
AAA = ("Ana Airport", "AAA"), AAB = ("Arrabury Airport", "AAB") ...

Add entity Cancel

3. On the page for the **Weekday** entity, in the **Learned** section, ensure **Not required** is selected. Then, in the **List** section, select **+ Add new list**. Then enter the following value and synonym and select **Save**:

|List key | synonyms|

|---|---|

| Sunday | Sun |

Edit Delete

Required component ⓘ

> Learned	<input checked="" type="radio"/> Not Required
> Prebuilt (0)	<input type="radio"/> Not Required
> Regular expression (0)	<input type="radio"/> Not Required
∨ List (0)	<input type="radio"/> Not Required

A list component adds additional phrases to match against, and also returns the list key of the match.

List key ↑ ∨ Synonyms ∨

+ Add new list

Sunday Sun × Type in value and press enter...

Save Cancel

4. Repeat the previous step to add the following list components:

|Value | synonyms|

|---|---|

| Monday | Mon |

| Tuesday | Tue, Tues |

| Wednesday | Wed, Weds |

| Thursday | Thur, Thurs |

| Friday | Fri |

| Saturday | Sat |

▼ List (7)

🔍 ☐ Not Required

A list component adds additional phrases to match against, and also returns the list key of the match.

☐ List key ↑ ▼

Synonyms ▼

+ Add new list

☐ Friday

Friday × Fri ×

☐ Monday

Monday × Mon ×

☐ Saturday

Saturday × Sat ×

☐ Sunday

Sunday × Sun ×

☐ Thursday

Thursday × Thur × Thurs ×

5. After adding and saving the list values, return to the **Data labeling** page.
6. Select the **GetDate** intent and enter the following new example utterance:
what date was it on Saturday?
7. When the utterance has been added, select the word **Saturday**, and in the drop-down list that appears, select **Weekday**.

☐ GetDate

what date was it on Saturday?

Search for an entity

Weekday

Location

View in labeling pane

☐ GetTime

what's the time in N

8. Add another example utterance for the **GetDate** intent:
what date will it be on Friday?
9. When the utterance has been added, map **Friday** to the **Weekday** entity.
10. Add another example utterance for the **GetDate** intent:
what will the date be on Thurs?
11. When the utterance has been added, map **Thurs** to the **Weekday** entity.
12. Select **Save changes** to save the new utterances.

GetDate ▼

Write your example utterance and press enter

0/500

☐

GetDate

what will the date be on Thurs?

Wee...

☐

GetDate

what date will it be on Friday?

Week...

☐

GetDate

what date was it on Saturday?

Weekday

Add a *prebuilt* entity

The Azure AI Language service provides a set of *prebuilt* entities that are commonly used in conversational applications.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+** **Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Date` and select the **Prebuilt** entity tab. Then select **Add entity**.

Entity name *

Date

An entity can be extracted by different methods. They can be learned through context, matched from a list, or detected by a prebuilt. Every entity in your project is composed by one or more of these methods that are defined as your entity's components.

Learned List **Prebuilt** Regex

The prebuilt component allows you to select from a library of ready-built common types such as numbers, datetimes, names and others. When added, a prebuilt component is automatically detected.

"Book **2** adult business tickets to **New York City**"

number geography

The "number" and "geography" entities use pre-built components from the library of common types that are available.

Add entity

Cancel

3. On the page for the **Date** entity, in the **Learned** section, ensure **Not required** is selected. Then, in the **Prebuilt** section, select **+ Add new prebuilt**.
4. In the **Select prebuilt** list, select **DateTime** and then select **Save**.

Edit Delete Required component ⓘ

Search

Choice.Boolean
Description: boolean choice
Example: yes

DateTime
Description: Date time expressions
Example: May 11th

Email
Description: Email addresses
Example: user@example.net

General.Event
Description: Important events
Example: World War two

General.Organization
Description: Companies and corporati...
Example: Microsoft

DateTime

Not Required

label your utterances with to train a machine learned model. The model learns to predict where the entity is based on the only defined if you add labels by [labeling utterances](#). If you do not label any utterances with this entity, it will not have a

Not Required

common types such as numbers, dates, times, and others.

Description ▾

Save Cancel

5. After adding the prebuilt entity, return to the **Data labeling** page

6. Select the **GetDay** intent and enter the following new example utterance:
what day was 01/01/1901?
7. When the utterance has been added, select **01/01/1901**, and in the drop-down list that appears, select **Date**.
8. Add another example utterance for the **GetDay** intent:
what day will it be on Dec 31st 2099?
9. When the utterance has been added, map **Dec 31st 2099** to the **Date** entity.
10. Select **Save changes** to save the new utterances.

☐ GetDay

what day will it be on Dec 31st 2099?
Date

☐ GetDay

what day was 01/01/1901?
Date

Retrain the model

Now that you've modified the schema, you need to retrain and retest the model.

1. On the **Training jobs** page, select **Start a training job**.
2. On the **Start a training job** dialog, select **overwrite an existing model** and specify the **Clock** model. Select **Train** to train the model. If prompted, confirm you want to overwrite the existing model.

Select Model

You can train a new model or overwrite an existing one. Training a new model is best at the beginning or for comparing between model performances. Overwriting a model will replace the old model with the new data.

You are currently using the latest 2022-09-01 training configuration version. [Click here to change.](#)

☐ Train a new model

☒ Overwrite an existing model

Clock ▾

3. When training is complete the job **Status** will update to **Training succeeded**.
4. Select the **Model performance** page and then select the **Clock** model. Review the evaluation metrics (*precision*, *recall*, and *F1 score*) and the *confusion matrix* generated by the evaluation that was performed when training (note that due to the small number of sample utterances, not all intents may

be included in the results).

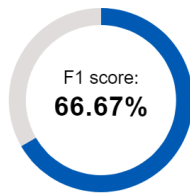
Clock

Overview Model Performance Test set details Dataset distribution Confusion matrix

Model Type

Intents

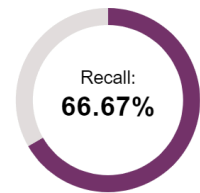
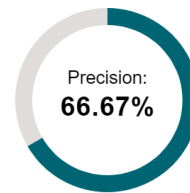
Status: ✔ Trained successfully



F1 score: ⓘ 66.67%
Precision: ⓘ 66.67%
Recall: ⓘ 66.67%

Finished training on: 6/23/2024, 1:04:52 AM
Total training time: 0 hour(s), 1 minute(s), 10 second(s)

Training data	Percentage
splitting type:	
Number of training utterances:	16 (84.21%)
Number of testing utterances:	3 (15.79%)



[Learn more about how to improve your model](#)

- On the **Deploying a model** page, select **Add deployment**.
- On the **Add deployment** dialog, select **Override an existing deployment name**, and then select **production**.
- Select the **Clock** model in the **Model** field and then select **Deploy** to deploy it. This may take some time.

Add deployment

×

Create or select an existing deployment name

You can create a new deployment name or overwrite an existing deployment name to add a trained model to them



Create a new deployment name



Override an existing deployment name

By selecting an existing deployment name, you will override any deployed model to this deployment name

Deployment name

production

Assign trained model to your deployment name

Add a trained model to your selected deployment name

Model

Clock

Deployment regions

Select the regions to deploy in based on the available assigned resources.

Regions

East US - AI-LEARN-LANG-1

Deploy

Cancel


- When the model is deployed, on the **Testing deployments** page, select the **production** deployment under the **Deployment name** field, and then test it with the following text:
`what's the time in Edinburgh?`
- Review the result that is returned, which should hopefully predict the **GetTime** intent and a **Location** entity with the text value "Edinburgh".
- Try testing the following utterances:
`what time is it in Tokyo?`
`what date is it on Friday?`
`what's the date on Weds?`

what day was 01/01/2020?

what day will Mar 7th 2030 be?

Testing deployments

Test a deployment by providing a sample utterance to find out what intent and entities get recognized. These are the same predictions as when making API calls against your model in code.

 Run the test

Deployment name

production

Enter your own text, or upload a text document

what time is it in Tokyo?

Result JSON

Intent

Top intent

GetTime

Confidence: 87.85%

Entities

Location

Tokyo

Confidence: 100.00%

Original text

what time is it in Tokyo?
Loc...

Use the model from a client app

In a real project, you'd iteratively refine intents and entities, retrain, and retest until you are satisfied with the predictive performance. Then, when you've tested it and are satisfied with its predictive performance, you can use it in a client app by calling its REST interface or a runtime-specific SDK.

Prepare to develop an app in Visual Studio Code

You'll develop your language understanding app using Visual Studio Code. The code files for your app have been provided in a GitHub repo.

Tip: If you have already cloned the **mslearn-ai-language** repo, open it in Visual Studio code. Otherwise, follow these steps to clone it to your development environment.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the <https://github.com/MicrosoftLearning/mslearn-ai-language> repository to a local folder (it doesn't matter which folder).
3. When the repository has been cloned, open the folder in Visual Studio Code.

Note: If Visual Studio Code shows you a pop-up message to prompt you to trust the code you are opening, click on **Yes, I trust the authors** option in the pop-up.

4. Wait while additional files are installed to support the C# code projects in the repo.

Note: If you are prompted to add required assets to build and debug, select **Not Now**.

Configure your application

Applications for both C# and Python have been provided, as well as a sample text file you'll use to test the summarization. Both apps feature the same functionality. First, you'll complete some key parts of the application to enable it to use your Azure AI Language resource.

1. In Visual Studio Code, in the **Explorer** pane, browse to the **Labfiles/03-language** folder and expand the **CSharp** or **Python** folder depending on your language preference and the **clock-client** folder it contains. Each folder contains the language-specific files for an app into which you're going to integrate Azure AI Language question answering functionality.
2. Right-click the **clock-client** folder containing your code files and open an integrated terminal. Then install the Azure AI Language conversational language understanding SDK package by running the appropriate command for your language preference:

C#:

```
dotnet add package Azure.AI.Language.Conversations --version 1.1.0
```

Python:

```
pip install azure-ai-language-conversations
```

3. In the **Explorer** pane, in the **clock-client** folder, open the configuration file for your preferred language
 - **C#:** appsettings.json
 - **Python:** .env
4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal).
5. Save the configuration file.

Add code to the application

Now you're ready to add the code necessary to import the required SDK libraries, establish an authenticated connection to your deployed project, and submit questions.

1. Note that the **clock-client** folder contains a code file for the client application:
 - **C#:** Program.cs
 - **Python:** clock-client.py (View here: [clock-client.py](#))

Open the code file and at the top, under the existing namespace references, find the comment **Import namespaces**. Then, under this comment, add the following language-specific code to import the namespaces you will need to use the Text Analytics SDK:

C#: Programs.cs

```
// import namespaces using Azure; using Azure.AI.Language.Conversations;
```


Python: clock-client.py (View here: [clock-client.py](#))

```
# Import namespaces from azure.core.credentials import AzureKeyCredential from
azure.ai.language.conversations import ConversationAnalysisClient
```

2. In the **Main** function, note that code to load the prediction endpoint and key from the configuration file has already been provided. Then find the comment **Create a client for the Language service model** and add the following code to create a prediction client for your Language Service app:

C#: Programs.cs

```
// Create a client for the Language service model Uri endpoint = new
Uri(predictionEndpoint); AzureKeyCredential credential = new
AzureKeyCredential(predictionKey); ConversationAnalysisClient client = new
ConversationAnalysisClient(endpoint, credential);
```

Python: clock-client.py (View here: [clock-client.py](#))

```
# Create a client for the Language service model client = ConversationAnalysisClient(
ls_prediction_endpoint, AzureKeyCredential(ls_prediction_key))
```

3. Note that the code in the **Main** function prompts for user input until the user enters "quit". Within this loop, find the comment **Call the Language service model to get intent and entities** and add the following code:

C#: Programs.cs

```
// Call the Language service model to get intent and entities var projectName = "Clock";
var deploymentName = "production"; var data = new { analysisInput = new {
conversationItem = new { text = userText, id = "1", participantId = "1", } }, parameters
= new { projectName, deploymentName, // Use Utf16CodeUnit for strings in .NET.
stringIndexType = "Utf16CodeUnit", }, kind = "Conversation", }; // Send request Response
response = await client.AnalyzeConversationAsync(RequestContent.Create(data)); dynamic
conversationalTaskResult =
response.Content.ToDynamicFromJson(JsonPropertyNames.CamelCase); dynamic
conversationPrediction = conversationalTaskResult.Result.Prediction; var options = new
JsonSerializerOptions { WriteIndented = true };
Console.WriteLine(JsonSerializer.Serialize(conversationalTaskResult, options));
Console.WriteLine("-----\n"); Console.WriteLine(userText); var topIntent
= ""; if (conversationPrediction.Intents[0].ConfidenceScore > 0.5) { topIntent =
conversationPrediction.TopIntent; }
```

Python: clock-client.py (View here: [clock-client.py](#))

```
# Call the Language service model to get intent and entities cls_project = 'Clock'
deployment_slot = 'production' with client: query = userText result =
client.analyze_conversation( task={ "kind": "Conversation", "analysisInput": {
"conversationItem": { "participantId": "1", "id": "1", "modality": "text", "language":
"en", "text": query }, "isLoggingEnabled": False }, "parameters": { "projectName":
cls_project, "deploymentName": deployment_slot, "verbose": True } } ) top_intent =
result["result"]["prediction"]["topIntent"] entities = result["result"]["prediction"]
["entities"] print("view top intent:") print("\ttop intent: {}".format(result["result"]
["prediction"]["topIntent"])) print("\tcategory: {}".format(result["result"]
["prediction"]["intents"][0]["category"])) print("\tconfidence score:
{}\n".format(result["result"]["prediction"]["intents"][0]["confidenceScore"]))
print("view entities:") for entity in entities: print("\tcategory:
{}".format(entity["category"])) print("\ttext: {}".format(entity["text"]))
```

```
print("\tconfidence score: {}".format(entity["confidenceScore"])) print("query:
{}".format(result["result"]["query"]))
```

The call to the Language service model returns a prediction/result, which includes the top (most likely) intent as well as any entities that were detected in the input utterance. Your client application must now use that prediction to determine and perform the appropriate action.

4. Find the comment **Apply the appropriate action**, and add the following code, which checks for intents supported by the application (**GetTime**, **GetDate**, and **GetDay**) and determines if any relevant entities have been detected, before calling an existing function to produce an appropriate response.

C#: Programs.cs

```
// Apply the appropriate action switch (topIntent) { case "GetTime": var location =
"local"; // Check for a location entity foreach (dynamic entity in
conversationPrediction.Entities) { if (entity.Category == "Location") {
//Console.WriteLine($"Location Confidence: {entity.ConfidenceScore}"); location =
entity.Text; } } // Get the time for the specified location string timeResponse =
GetTime(location); Console.WriteLine(timeResponse); break; case "GetDay": var date =
DateTime.Today.ToShortDateString(); // Check for a Date entity foreach (dynamic entity
in conversationPrediction.Entities) { if (entity.Category == "Date") {
//Console.WriteLine($"Location Confidence: {entity.ConfidenceScore}"); date =
entity.Text; } } // Get the day for the specified date string dayResponse =
GetDay(date); Console.WriteLine(dayResponse); break; case "GetDate": var day =
DateTime.Today.DayOfWeek.ToString(); // Check for entities // Check for a Weekday entity
foreach (dynamic entity in conversationPrediction.Entities) { if (entity.Category ==
"Weekday") { //Console.WriteLine($"Location Confidence: {entity.ConfidenceScore}"); day
= entity.Text; } } // Get the date for the specified day string dateResponse =
GetDate(day); Console.WriteLine(dateResponse); break; default: // Some other intent (for
example, "None") was predicted Console.WriteLine("Try asking me for the time, the day,
or the date."); break; }
```

Python: clock-client.py (View here: [clock-client.py](#))

```
# Apply the appropriate action if top_intent == 'GetTime': location = 'local' # Check
for entities if len(entities) > 0: # Check for a location entity for entity in entities:
if 'Location' == entity["category"]: # ML entities are strings, get the first one
location = entity["text"] # Get the time for the specified location
print(GetTime(location)) elif top_intent == 'GetDay': date_string =
date.today().strftime("%m/%d/%Y") # Check for entities if len(entities) > 0: # Check for
a Date entity for entity in entities: if 'Date' == entity["category"]: # Regex entities
are strings, get the first one date_string = entity["text"] # Get the day for the
specified date print(GetDay(date_string)) elif top_intent == 'GetDate': day = 'today' #
Check for entities if len(entities) > 0: # Check for a Weekday entity for entity in
entities: if 'Weekday' == entity["category"]: # List entities are lists day =
entity["text"] # Get the date for the specified day print(GetDate(day)) else: # Some
other intent (for example, "None") was predicted print('Try asking me for the time, the
day, or the date.')
```

5. Save your changes and return to the integrated terminal for the **clock-client** folder, and enter the following command to run the program:

- **C#:** dotnet run
- **Python:** python clock-client.py

Tip: You can use the **Maximize panel size** (^) icon in the terminal toolbar to see more of the console text.

6. When prompted, enter utterances to test the application. For example, try:

Hello

What time is it?

What's the time in London?

What's the date?

What date is Sunday?

What day is it?

What day is 01/01/2025?

Note: The logic in the application is deliberately simple, and has a number of limitations. For example, when getting the time, only a restricted set of cities is supported and daylight savings time is ignored. The goal is to see an example of a typical pattern for using Language Service in which your application must:

1. Connect to a prediction endpoint.
2. Submit an utterance to get a prediction.
3. Implement logic to respond appropriately to the predicted intent and entities.

7. When you have finished testing, enter *quit*.

Clean up resources

If you're finished exploring the Azure AI Language service, you can delete the resources you created in this exercise. Here's how:

1. Open the Azure portal at <https://portal.azure.com>, and sign in using the Microsoft account associated with your Azure subscription.
2. Browse to the Azure AI Language resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

To learn more about conversational language understanding in Azure AI Language, see the [Azure AI Language documentation](#).

Knowledge Check

1. Your app must interpret a command such as "turn on the light" or "switch the light on". What do these phrases represent in a language model? *

☐ Intents.

☒ Utterances.

✓ That's correct. Utterances are example phrases that indicate a specific intent.

☐ Entities.

2. Your app must interpret a command to book a flight to a specified city, such as "Book a flight to Paris." How should you model the city element of the command? *

☐ As an intent.

☐ As an utterance.

☒ As an entity.

✓ That's correct. The city is an entity to which the intent (booking a flight) should be applied.

3. Your language model needs to detect an email when present in an utterance. What is the simplest way to extract that email? *

☐ Use Regular Expression entities.

☒ Use prebuilt entity components.

✓ That's correct. When a language model needs to detect a common entity, use prebuilt components to have the Azure AI Language service automatically detect the entity.

☐ Use Learned entity components.

Summary

In this module, you learned how to create a conversational language understanding model.

Now that you've completed this module, you can:

- Provision an Azure AI Language resource
- Define intents, entities, and utterances
- Use patterns to differentiate similar utterances
- Use pre-built entity components
- Train, test, publish, and review a model

To learn more about language understanding, see the [Azure AI Language documentation](#).
