Create a language understanding model with the Language service

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**.
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:

e. On the bar at the top if the page, select the **Settings (⚙)** button.
    f. On the **Settings** page, view the **Resources** tab.
    g. Select the language resource you just created, and click **Switch resource**.
    h. At the top of the page, click **Language Studio** to return to the Language Studio home page
3. At the top of the portal, in the **Create new** menu, select **Conversational language understanding**.
4. 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`
5. 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**.

1. On the **Schema definition** page, on the **Intents** tab, select **+ Add** to add a new intent named `GetTime`.
2. Verify that the **GetTime** intent is listed (along with the default **None** intent). Then add the following additional intents:
    - `GetDay`
    - `GetDate`

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`
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**.
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).

   NOTE To learn more about the evaluation metrics, refer to the documentation

6. Go to the **Deploying a model** page, then select **Add deployment**.
7. 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.
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)

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**.
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.
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.
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.

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**.
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`

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`

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**.
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.

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**.
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**.
5. After adding athe 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.

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.
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).
5. On the **Deploying a model** page, select **Add deployment**.
6. On the **Add deployment** dialog, select **Override an existing deployment name**, and then select **production**.

7. Select the **Clock** model in the **Model** field and then select **Deploy** to deploy it. This may take some time.
8. 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?
   ```

9. Review the result that is returned, which should hopefully predict the **GetTime** intent and a **Location** entity with the text value "Edinburgh".
10. 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?
    ```

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.
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 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#:

   CodeCopy

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

   Python:

   CodeCopy

   ```
   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

    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

    CodeCopy

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

    Python: clock-client.py

    CodeCopy

```
# 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

    CodeCopy

```
// 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

CodeCopy

```
# 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

CodeCopy

```
// 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

CodeCopy

```
 # 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

   CodeCopy

```
 // 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

CodeCopy

```
# 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:

a. Connect to a prediction endpoint.
    b. Submit an utterance to get a prediction.
    c. 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.