4.2 - Create a Custom Skill for Azure AI Search

Overview

Use the power of artificial intelligence to enrich your data and find new insights.

In this module you will learn how to:

- Implement a custom skill for Azure AI Search

- Integrate a custom skill into an Azure AI Search skillset

Introduction

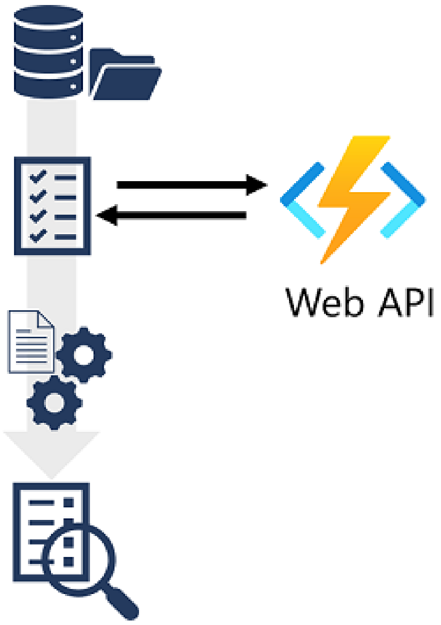
You can use the **predefined skills in Azure AI Search** to greatly enrich an index by extracting additional information from the source data.

However, there may be occasions when you have specific data extraction needs that cannot be met with the predefined skills and require some custom functionality.

For example:

- Integrate the Form Recognizer service to extract data from forms
- Consume an Azure Machine Learning model to integrate predicted values into an index
- Any other custom logic

To support these scenarios, you can implement custom skills as web-hosted services (such as Azure Functions) that support the required interface for integration into a skillset.



In this module you will learn how to:

- Implement a custom skill for Azure AI Search
- Integrate a custom skill into an Azure AI Search skillset

Note: This module assumes you already know how to create and use an Azure AI Search solution that includes built-in skills. If not, complete the Create an Azure AI Search solution module first.

Define the custom skill schema

Your custom skill must implement the expected schema for input and output data that is expected by skills in an Azure AI Search skillset.

Input Schema

The input schema for a custom skill defines a JSON structure containing a record for each document to be processed. Each document has a unique identifier, and a data payload with one or more inputs, like this:

```
{
    "values": [
      {
        "recordId": "<unique_identifier>",
        "data":
           {
             "<input1_name>":  "<input1_value>",
             "<input2_name>": "<input2_value>",
             ...
           }
      },
      {
        "recordId": "<unique_identifier>",
        "data":
           {
             "<input1_name>":  "<input1_value>",
             "<input2_name>": "<input2_value>",
             ...
           }
      },
      ...
    ]
}
```

Output schema

The schema for the results returned by your custom skill reflects the input schema. It is assumed that the output contains a record for each input record, with either the results produced by the skill or details of any errors that occurred.

```
{
    "values": [
      {
        "recordId": "<unique_identifier_from_input>",
        "data":
           {
             "<output1_name>":  "<output1_value>",
              ...
           },
         "errors": [...],
         "warnings": [...]
      },
      {
        "recordId": "< unique_identifier_from_input>",
        "data":
           {
             "<output1_name>":  "<output1_value>",
              ...
           },
        "errors": [...],
        "warnings": [...]
      },
      ...
    ]
}
```

The output value in this schema is a ***property bag*** that can contain any JSON structure, reflecting the fact that index fields aren't necessarily simple data values, but can contain complex types.

Add a custom skill

To integrate a custom skill into your indexing solution, you must add a skill for it to a skillset using the **Custom.WebApiSkill** skill type.

The skill definition must:

- Specify the **URI to your web API endpoint**, including parameters and headers if necessary.
- Set the context to specify at which point in the document hierarchy the skill should be called
- Assign input values, usually from existing document fields
- Store output in a new field, optionally specifying a target field name (otherwise the output name is used)

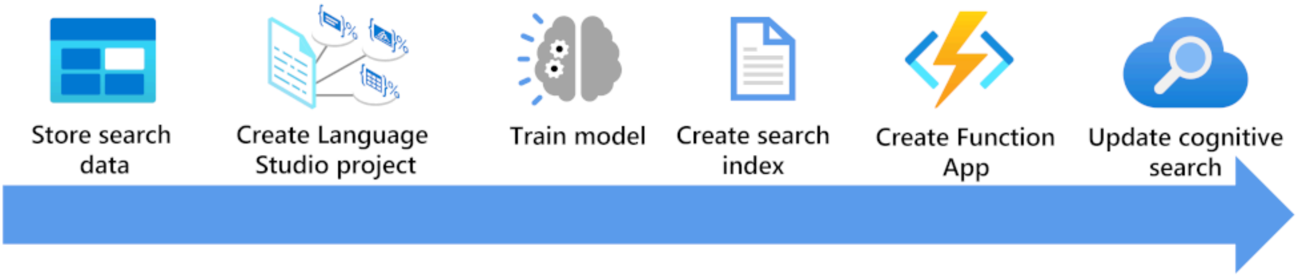
```
{
    "skills": [
      ...,
      {
        "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
        "description": "<custom skill description>",
        "uri": "https://<web_api_endpoint>?<params>",
        "httpHeaders": {
            "<header_name>": "<header_value>"
        },
        "context": "/document/<where_to_apply_skill>",
        "inputs": [
          {
            "name": "<input1_name>",
            "source": "/document/<path_to_input_field>"
          }
        ],
        "outputs": [
          {
            "name": "<output1_name>",
            "targetName": "<optional_field_name>"
          }
        ]
      }
  ]
}
```

Your skillset can include both built-in skills and custom skills, depending on your use case. The `skills` array seen in the above example will contain all of your skills, with the custom skills code running outside of the search service (as applicable) which we'll see in the exercise later in this module.

The following units explore specific kinds of custom skills through the use additional Azure resources.

Custom text classification skill

Custom text classification allows you to map a passage of text to different user defined classes. For example, you could train a model on the synopsis on the back cover of books to automatically identify a books genre. You then use that identified genre to enrich your online shop search engine with a genre facet.



Here, you'll see what you need to consider to enrich a search index using a custom text classification model:

- Store your documents so they can be accessed by Language Studio and Azure AI Search indexers.
- Create a custom text classification project.
- Train and test your model.
- Create a search index based on your stored documents.
- Create a **function app that uses your deployed trained model.**
- Update your search solution, your index, indexer, and custom skillset.

Store your data

Azure Blob storage can be accessed from both Language Studio and Azure AI Services. The container needs to be accessible, so the simplest option is to choose Container, but it's also possible to use private containers with some additional configuration.

Along with your data, you also need a way to assign classifications for each document. Language Studio provides a graphical tool that you can use to classify each document one at a time manually.

You can choose between two different types of project.

- If a document maps to a single class use a single label classification project.
- If you want to map a document to more than one class, use the multi label classification project.

If you don't want to manually classify each document, you can label all your documents before you create your Azure AI Language project. This process involves creating a labels JSON document in this format:

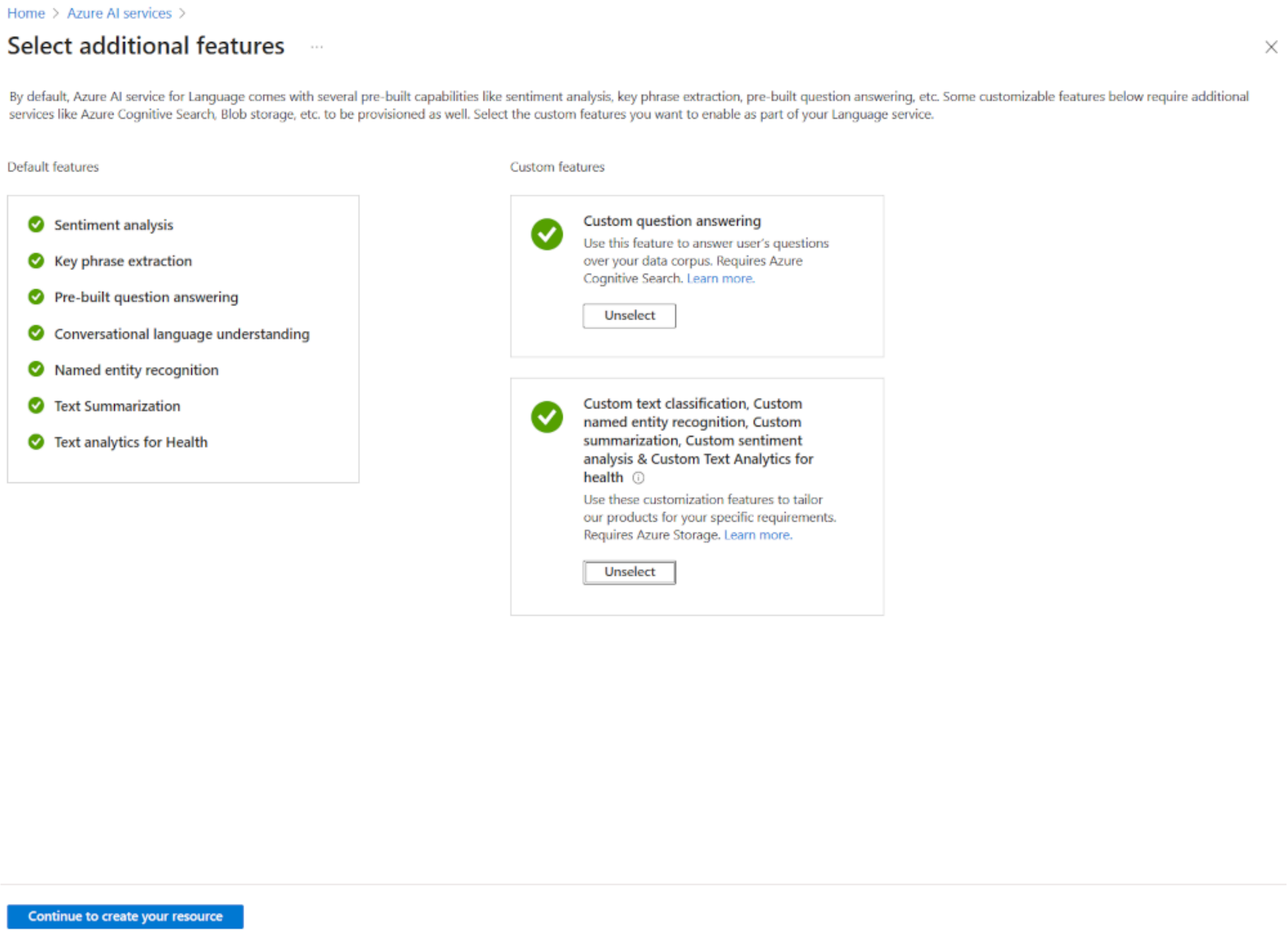
```
{
    "projectFileVersion": "2022-05-01",
    "stringIndexType": "Utf16CodeUnit",
    "metadata": {
      "projectKind": "CustomMultiLabelClassification",
      "storageInputContainerName": "{CONTAINER-NAME}",
      "projectName": "{PROJECT-NAME}",
      "multilingual": false,
      "description": "Project-description",
      "language": "en-us"
    },
    "assets": {
      "projectKind": "CustomMultiLabelClassification",
      "classes": [
        {
          "category": "Class1"
        },
        {
          "category": "Class2"
        }
      ],
      "documents": [
          {
              "location": "{DOCUMENT-NAME}",
              "language": "{LANGUAGE-CODE}",
              "dataset": "{DATASET}",
              "classes": [
                  {
                      "category": "Class1"
                  },
                  {
                      "category": "Class2"
                  }
              ]
          }
      ]
  }
```

You add as many classes as you have to the `classes` array. You add an entry for each document in the `documents` array including which classes the document matches.

Create your Azure AI Language project

There are two ways to create your Azure AI Language project. If you start using Language Studio without first creating a language service in the Azure portal, Language Studio will offer to create one for you.

The most flexible way to create an Azure AI Language project is to first create your language service using the Azure portal. If you choose this option, you get the option to add custom features.



By default, Azure AI service for Language comes with several pre-built capabilities like sentiment analysis, key phrase extraction, pre-built question answering, etc. Some customizable features below require additional services like Azure Cognitive Search, Blob storage, etc. to be provisioned as well. Select the custom features you want to enable as part of your Language service.

Default features

- ✓ Sentiment analysis
- ✓ Key phrase extraction
- ✓ Pre-built question answering
- ✓ Conversational language understanding
- ✓ Named entity recognition
- ✓ Text Summarization
- ✓ Text analytics for Health

Custom features

✓ Custom question answering
Use this feature to answer user's questions over your data corpus. Requires Azure Cognitive Search. Learn more.

✓ Custom text classification, Custom named entity recognition, Custom summarization, Custom sentiment analysis & Custom Text Analytics for health ⓘ
Use these customization features to tailor our products for your specific requirements. Requires Azure Storage. Learn more.

As you are going to create a custom text classification, select that custom feature when creating your language service. You'll also link the language service to a storage account using this method.

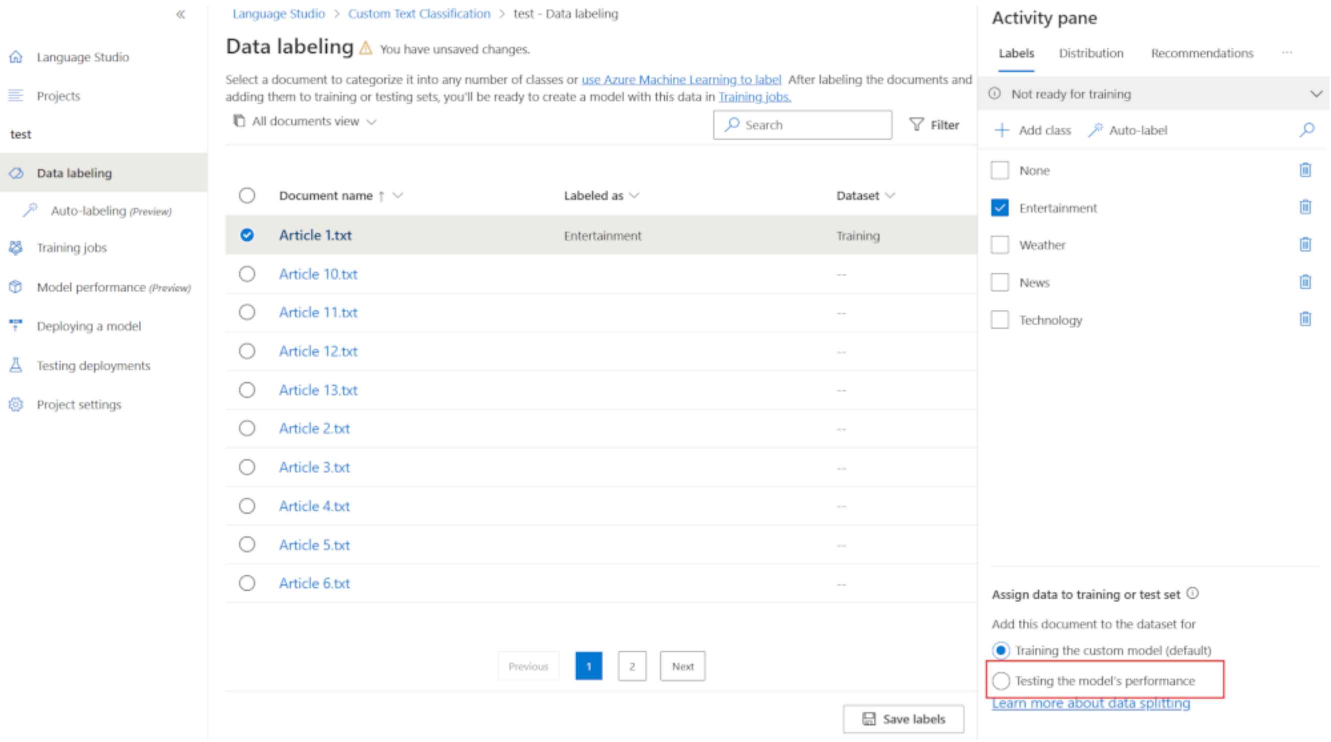
Once the resource has been deployed, you can navigate directly to the Language Studio from the overview pane of the language service. You can then create a new custom text classification project.

> Note: If you have created your language service from Language Studio you might need to follow these steps. Set roles for your Azure Language resource and storage account to connect your storage container to your custom text classification project.

Train your classification model

As with all AI models, you need to have identified data that you can use to train it. The model needs to see examples of how to map data to a class and have some examples it can use to test the model.

You can choose to let the model automatically split your training data, by default it will use 80% of the documents to train the model and 20% to blind test it. If you have some specific documents that you want to test your model with, you can label documents for testing.



Select a document to categorize it into any number of classes or use Azure Machine Learning to label After labeling the documents and adding them to training or testing sets, you'll be ready to create a model with this data in Training jobs.

	Document name ↑	Labeled as	Dataset
●	Article 1.txt	Entertainment	Training
○	Article 10.txt		--
○	Article 11.txt		--
○	Article 12.txt		--
○	Article 13.txt		--
○	Article 2.txt		--
○	Article 3.txt		--
○	Article 4.txt		--
○	Article 5.txt		--
○	Article 6.txt		--

□ None
☑ Entertainment
□ Weather
□ News
□ Technology

Assign data to training or test set

Add this document to the dataset for

● Training the custom model (default)

○ Testing the model's performance

Learn more about data splitting

In Language Studio, in your project, select **Data labeling**. You'll see all your documents. Select each document you'd like to add to the testing set, then select **Testing the model's performance.** Save your updated labels and then create a new training job.

Create search index

There isn't anything specific you need to do to create a search index that will be enriched by a custom text classification model. Follow the steps in Create an Azure AI Search solution. You'll be updating the index, indexer, and custom skill after you've created a function app.

Create an Azure function app

You can choose the language and technologies you want for your function app. The app needs to be able to **pass JSON to the custom text classification endpoint**, for example:

```
{
    "displayName": "Extracting custom text classification",
    "analysisInput": {
        "documents": [
            {
                "id": "1",
                "language": "en-us",
                "text": "This film takes place during the events of Get Smart. Bruce and Lloyd have been testing out an invisibility cloak, but during a party, Maraguayan agent Isabelle steals it for El Presidente. Now, Bruce and Lloyd must find the cloak on their own because the only non-compromised agents, Agent 99 and Agent 86  are in Russia"
            }
        ]
      },
    "tasks": [
        {
        "kind": "CustomMultiLabelClassification",
        "taskName": "Multi Label Classification",
        "parameters": {
            "project-name": "movie-classifier",
            "deployment-name": "test-release"}
        }
    ]
}
```

Then process the JSON response from the model, for example:

```
{
  "jobId": "be1419f3-61f8-481d-8235-36b7a9335bb7",
  "lastUpdatedDateTime": "2022-06-13T16:24:27Z",
  "createdDateTime": "2022-06-13T16:24:26Z",
  "expirationDateTime": "2022-06-14T16:24:26Z",
  "status": "succeeded",
  "errors": [],
  "displayName": "Extracting custom text classification",
  "tasks": {
    "completed": 1,
    "failed": 0,
    "inProgress": 0,
    "total": 1,
    "items": [
      {
        "kind": "CustomMultiLabelClassificationLROResults",
        "taskName": "Multi Label Classification",
        "lastUpdateDateTime": "2022-06-13T16:24:27.7912131Z",
        "status": "succeeded",
        "results": {
          "documents": [
            {
              "id": "1",
              "class": [
                {
                  "category": "Action",
                  "confidenceScore": 0.99
                },
                {
                  "category": "Comedy",
                  "confidenceScore": 0.96
                }
              ],
              "warnings": []
            }
          ],
          "errors": [],
          "projectName": "movie-classifier",
          "deploymentName": "test-release"
        }
      }
    ]
  }
}
```

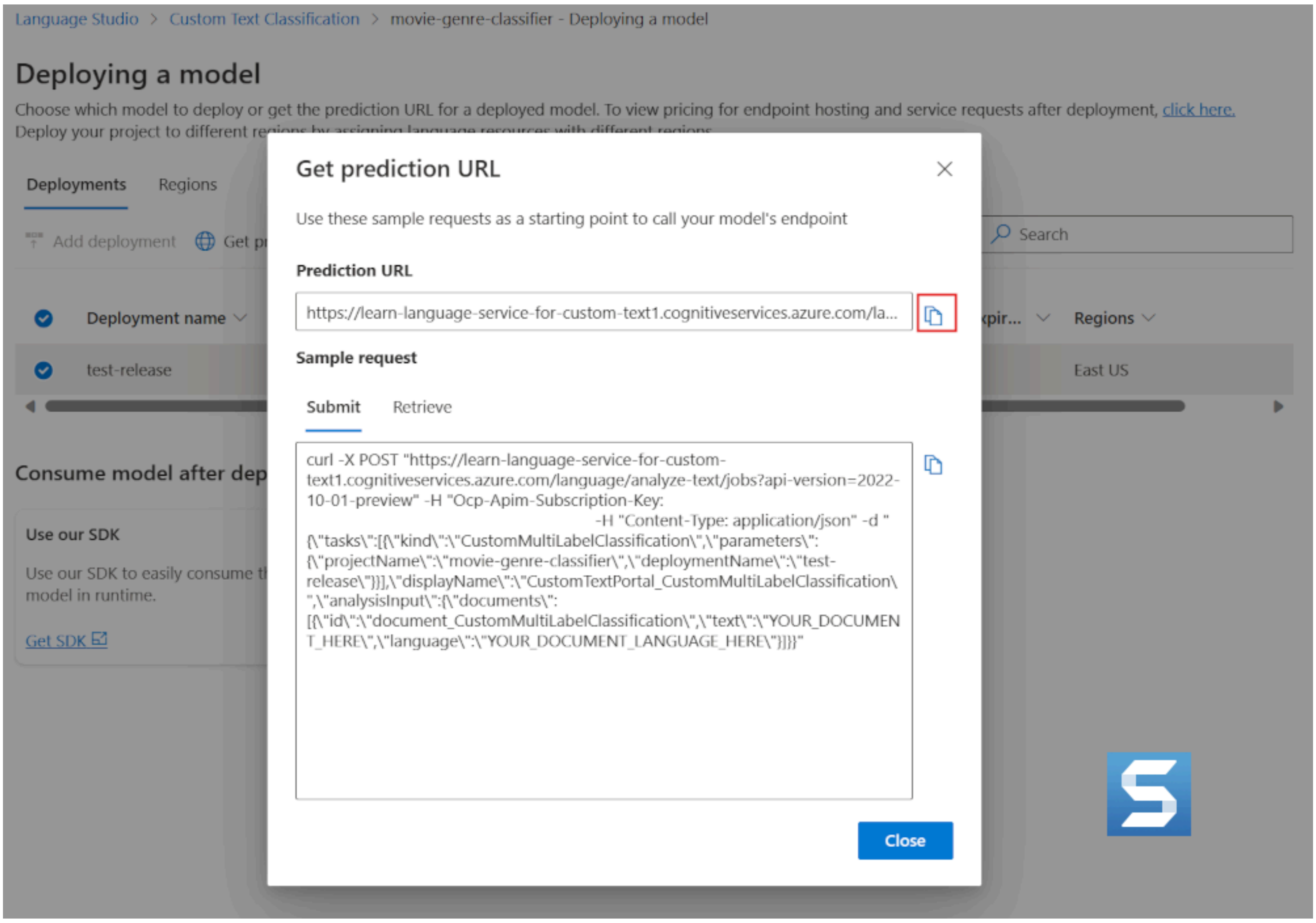
The function then returns a **structured JSON message back to a custom skillset in AI Search**, for example:

```
[{"category": "Action", "confidenceScore": 0.99}, {"category": "Comedy",
"confidenceScore": 0.96}]
```

There are five things the function app needs to know:

1. The text to be classified.
2. The endpoint for your trained custom text classification deployed model.
3. The primary key for the custom text classification project.
4. The project name.
5. The deployment name.

The text to be classified is passed from your custom skillset in AI Search to the function as input. The remaining four items can be found in Language Studio.



Choose which model to deploy or get the prediction URL for a deployed model. To view pricing for endpoint hosting and service requests after deployment, click here.
Deploy your project to different regions by assigning language resources with different regions.

Use our SDK to easily consume t
model in runtime.

Get prediction URL

Use these sample requests as a starting point to call your model's endpoint

Prediction URL

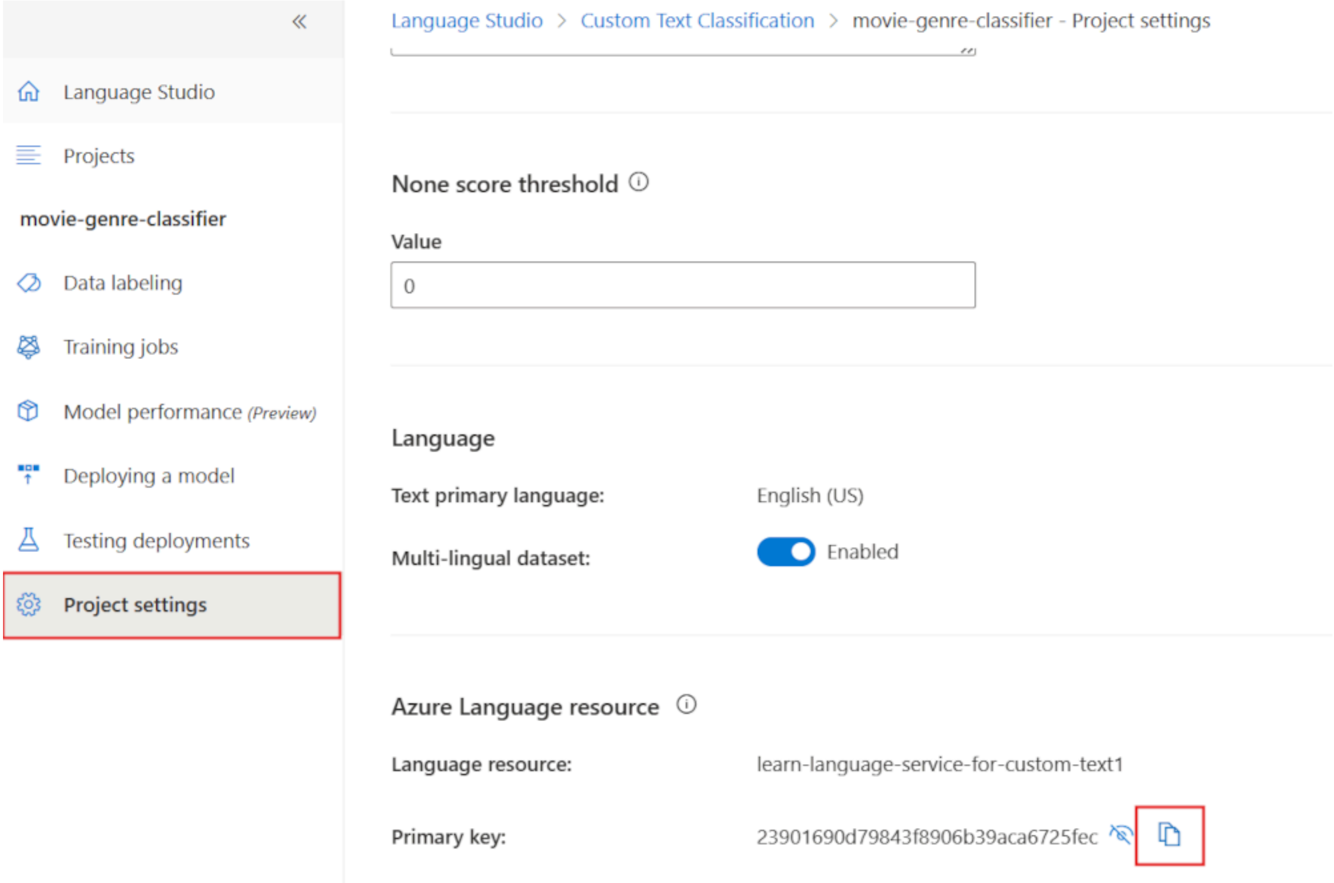
https://learn-language-service-for-custom-text1.cognitiveservices.azure.com/la...

Sample request

Submit Retrieve

```
curl -X POST "https://learn-language-service-for-custom-
text1.cognitiveservices.azure.com/language/analyze-text/jobs?api-version=2022-
10-01-preview" -H "Ocp-Apim-Subscription-Key:
                                                            -H "Content-Type: application/json" -d "
{\"tasks\":[{\"kind\":\"CustomMultiLabelClassification\",\"parameters\":
{\"projectName\":\"movie-genre-classifier\",\"deploymentName\":\"test-
release\"}}],\"displayName\":\"CustomTextPortal_CustomMultiLabelClassification\
",\"analysisInput\":{\"documents\":
[{\"id\":\"document_CustomMultiLabelClassification\",\"text\":\"YOUR_DOCUMEN
T_HERE\",\"language\":\"YOUR_DOCUMENT_LANGUAGE_HERE\"}]}}"
```

The endpoint and deployment name is on the deploying a model pane.



None score threshold ⓘ

Value: 0

Language

Text primary language: English (US)

Multi-lingual dataset: Enabled

Azure Language resource ⓘ

Language resource: learn-language-service-for-custom-text1

Primary key: 23901690d79843f8906b39aca6725fec

The project name and primary key are on the project settings pane.

Update your Azure AI Search solution

There are three changes in the Azure portal you need to make to enrich your search index:

1. You need to **add a field to your index to store the custom text classification enrichment.**
2. You need to **add a custom skillset to call your function app with the text to classify.**
3. You need to **map the response from the skillset into the index.**

(i) Add a field to an existing index

In the Azure portal, go to your AI Search resource, select the index and you'll add JSON in this format:

```
{
  "name": "classifiedtext",
  "type": "Collection(Edm.ComplexType)",
  "analyzer": null,
  "synonymMaps": [],
  "fields": [
    {
      "name": "category",
      "type": "Edm.String",
      "facetable": true,
      "filterable": true,
      "key": false,
      "retrievable": true,
      "searchable": true,
      "sortable": false,
      "analyzer": "standard.lucene",
      "indexAnalyzer": null,
      "searchAnalyzer": null,
      "synonymMaps": [],
      "fields": []
    },
    {
      "name": "confidenceScore",
      "type": "Edm.Double",
      "facetable": true,
      "filterable": true,
      "retrievable": true,
      "sortable": false,
      "analyzer": null,
      "indexAnalyzer": null,
      "searchAnalyzer": null,
      "synonymMaps": [],
      "fields": []
    }
  ]
}
```

This JSON adds a compound field to the index to store the class in a `category` field that is searchable. The second `confidenceScore` field stores the confidence percentage in a double field.

(ii) Edit the custom skillset

In the Azure portal, select the skillset and add JSON in this format:

```
{
  "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
  "name": "Genre Classification",
  "description": "Identify the genre of your movie from its summary",
  "context": "/document",
  "uri": "https://learn-acs-lang-serives.cognitiveservices.azure.com/language/analyze-
text/jobs?api-version=2022-05-01",
  "httpMethod": "POST",
  "timeout": "PT30S",
  "batchSize": 1,
  "degreeOfParallelism": 1,
  "inputs": [
    {
      "name": "lang",
      "source": "/document/language"
    },
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "text",
      "targetName": "class"
    }
  ],
  "httpHeaders": {}
}
```

This `WebApiSkill` skill definition specifies that the language and the contents of a document are passed as inputs to the function app. The app will return JSON text named `class`.

(iii) Map the output from the function app into the index

The last change is to map the output into the index. In the Azure portal, select the indexer and edit the JSON to have a new output mapping:

```
{
  "sourceFieldName": "/document/class",
  "targetFieldName": "classifiedtext"
}
```

The indexer now knows that the output from the function app `document/class` should be stored in the `classifiedtext` field. As this has been defined as a compound field, the function app has to return a JSON array containing a `category` and `confidenceScore` field.

You can now search an enriched search index for your custom classified text.

Machine learning custom skill

Using a machine learning custom skill works the same as adding any other custom skill to a search index.

Here, you'll see how using the `AmlSkill` custom skill is different and explore the considerations of how to effectively use it.

Custom Azure Machine Learning skill schema

When you enrich a search index with an **Azure Machine Learning (AML)** custom skill, the enrichment happens at the document level. The skillset used by your document indexer needs to include an `AmlSkill`. The schema for this skill is:

```
{
      "@odata.type": "#Microsoft.Skills.Custom.AmlSkill",
      "name": "AML name",
      "description": "AML description",
      "context": "/document",
      "uri": "https://[Your AML endpoint]",
      "key": "Your AML endpoint key",
      "resourceId": null,
      "region": null,
      "timeout": "PT30S",
      "degreeOfParallelism": 1,
      "inputs": [
        {
          "name": "field name in the AML model",
          "source": "field from the document in the index"
        },
    {
     "name": "field name in the AML model",
     "source": "field from the document in the index"
    },

   ],
   "outputs": [
    {
     "name": "result field from the AML model",
     "targetName": "result field in the document"
    }
   ]
  }
```

Take note that the custom skill doesn't include settings for `batchSize` as the AML model will process a single document at a time.

The remaining settings that control the performance of the skill are `timeout` and `degreeOfParallelism`. The above schema has set 30 seconds as the timeout value. The degree of parallelism should start at one. Depending on your infrastructure, you might be able to increase this number.

The best way to manage the efficiency of an AML skill is to **scale up the Kubernetes inference cluster appropriately to manage your workload.**

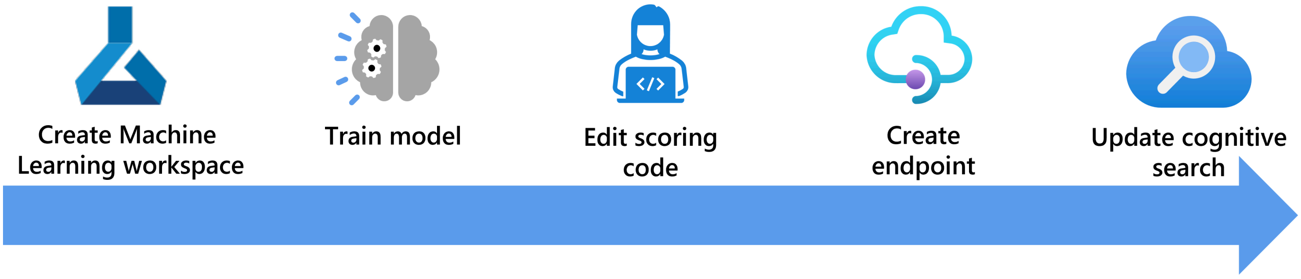
The index for the document needs a field to store the results from the AML model. You'll then add an output field mapping to store the results from the custom skill set to the field on the document in the index.

The JSON to do this output field mapping is:

```
"outputFieldMappings": [
  {
   "sourceFieldName": "/result field in the document",
   "targetFieldName": "result field from the AML model"
  }
 ]
```

You create your Azure Machine Learning model using developer tools like the Python SDK, REST APIs, or Azure CLI.

Another option is to take advantage of the Azure AI Machine Learning studio, a graphical user interface that lets you create, train, and deploy models without writing any code.



With a model created, you alter how the scoring code calls the model to allow it to be used by your custom search skill.

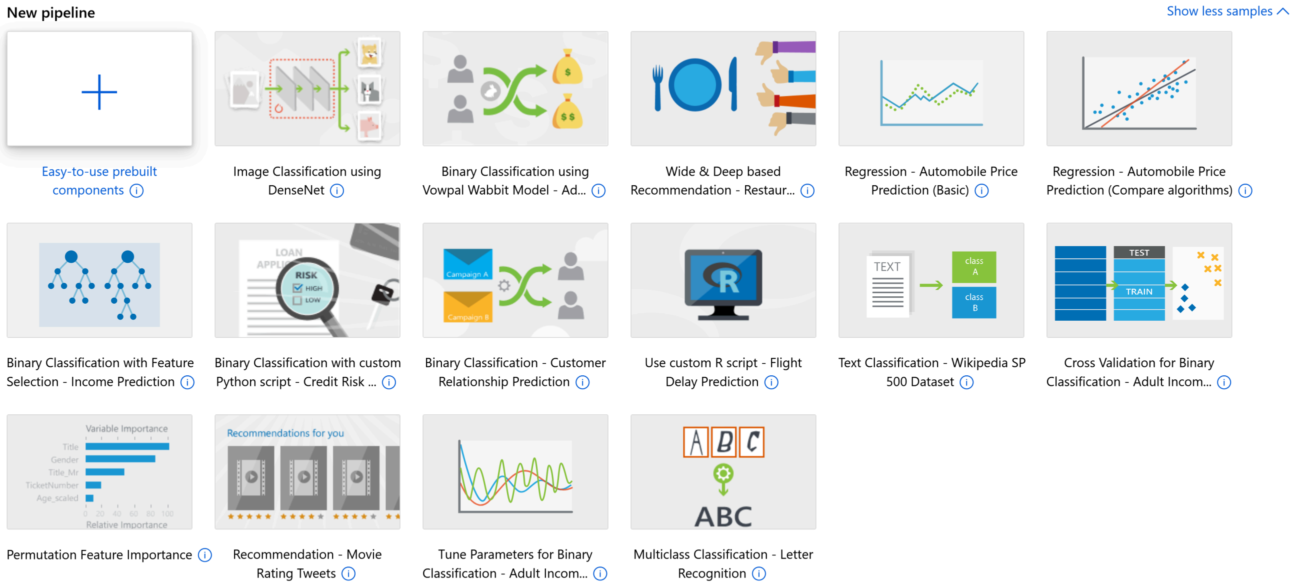
The last steps are to create a **Kubernetes cluster to host an endpoint for your model.**

Create an AML workspace

When you create the AML workspace, Azure will also create storage accounts, a key store, and application insights resources. The AML workspace Overview pane gives you a link to launch the Azure AI Machine Learning Studio.

Create and train a model in Azure Machine Learning studio

Azure AI Machine Learning Studio lets you use a designer to use drag and drop to create pipelines that create and train models. There's an even easier way to create models by using prebuilt templates.


New pipeline
Show less samples
Easy-to-use prebuilt components
Image Classification using DenseNet
Binary Classification using Vowpal Wabbit Model - Ad...
Wide & Deep based Recommendation - Restaur...
Regression - Automobile Price Prediction (Basic)
Regression - Automobile Price Prediction (Compare algorithms)
Binary Classification with Feature Selection - Income Prediction
Binary Classification with custom Python script - Credit Risk ...
Binary Classification - Customer Relationship Prediction
Use custom R script - Flight Delay Prediction
Text Classification - Wikipedia SP 500 Dataset
Cross Validation for Binary Classification - Adult Incom...
Permutation Feature Importance
Recommendation - Movie Rating Tweets
Tune Parameters for Binary Classification - Adult Incom...
Multiclass Classification - Letter Recognition

However you choose to create your models, they **need to be registered in Azure AI Machine Learning Studio** so that you can deploy the model to a web service.

Alter how the model works to allow it to be called by the AML custom skill

The models you train will normally use many examples of the data. The datasets will have many rows and be split and used to train and test the model. The code that handles this data and passes it to the model needs to be changed to handle single rows.

The JSON response from the model should also contain only the output prediction.

For example, if your data is an array of JSON objects:

```
[
    {
        "attribute-1": null,
        "attribute-2": null
    },
    {
        "attribute-1": null,
        "attribute-2": null
    },
    {
        "attribute-1": null,
        "attribute-2": null
    }
]
```

The Python scoring code will have to process the data a row at a time:

```
data = json.loads(data)
for row in data:
  for key, val in row.items():
    input_entry[key].append(decode_nan(val))
```

To change the input dataset to a single record:

```
{
"attribute-1": null,
"attribute-2": null
}
```

The Python code will need to change to:

```
data = json.loads(data)
for key, val in data.items():
  input_entry[key].append(decode_nan(val))
```

For the response from the scoring code, the default code returns the whole JSON document:

```
return json.dumps({"result": result.data_frame.values.tolist()})
```

The custom skill needs to be able to map a single response from the model. So the code should return JSON that is only the last attribute.

```
output = result.data_frame.values.tolist()
# return the last column of the the first row of the dataframe
return { "predicted_outcome": output[0][-1] }
```

Create an endpoint for your model to use

The model is deployed to an endpoint. Azure AI Machine Learning Studio supports deploying a model to a **real-time endpoint, a batch endpoint, or a web service.**

At the moment, the custom `AmlSkill` skill in Azure AI Search **only supports web service endpoints.**

The other restriction is that the endpoint has to be an Azure Kubernetes Service (AKS) cluster. Container instances aren't supported.

If you have experience in creating and managing AKS clusters, you can manually create the clusters in the Azure portal and reference them when you create your endpoint. However, an easier option is to let Azure AI Machine Learning Studio create and manage the cluster for you.

If you navigate to the compute section of the studio, you can **create inference clusters**. AML studio will then guide you through choosing the size of the cluster and even enable HTTPS and create a domain name for you. It will be in the format of `location.cloudapp.azure.com:443`.

Connect the AML custom skill to the endpoint

With everything above in place, you need to update your Azure AI Search service. The steps to do so are similar to examples shown in this module, so we won't detail the specifics here but will list out the steps to follow.

1. First, to enrich your search index you'll add a new field to your index to include the output for the model.
2. Then you'll update your index skillset and add the `#Microsoft.Skills.Custom.AmlSkill` custom skill.
3. Next, you'll change your indexer to map the output from the custom skill to the field you created on the index.
4. The last step is to rerun your indexer to enrich your index with the AML model.

Exercise - Create a Custom Skill for Azure AI Search

Azure AI Search uses an enrichment pipeline of AI skills to extract AI-generated fields from documents and include them in a search index. There's a comprehensive set of built-in skills that you can use, but if you have a specific requirement that isn't met by these skills, you can create a custom skill.

In this exercise, you'll create a custom skill that tabulates the frequency of individual words in a document to generate a list of the top five most used words, and add it to a search solution for Margie's Travel - a fictitious travel agency.

Create Azure resources

> **Note**: If you have previously completed the Create an Azure AI Search solution exercise, and still have these Azure resources in your subscription, you can skip this section and start at the **Create a search solution** section. Otherwise, follow the steps below to provision the required Azure resources.

1. In a web browser, open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the top search bar, search for *Azure AI services*, select **Azure AI services multi-service account**, and create an Azure AI services multi-service account resource with the following settings:
   - **Subscription**: *Your Azure subscription*
   - **Resource group**: *Choose or create a resource group (if you are using a restricted subscription, you may not have permission to create a new resource group - use the one provided)*
   - **Region**: *Choose from available regions geographically close to you*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Once deployed, go to the resource and on the **Overview** page, note the **Subscription ID** and **Location**. You will need these values, along with the name of the resource group in subsequent steps.
4. In Visual Studio Code, expand the **Labfiles/02-search-skill** folder and select **setup.cmd**. You will use this batch script to run the Azure command line interface (CLI) commands required to create the Azure resources you need.
5. Right-click the the **02-search-skill** folder and select **Open in Integrated Terminal**.
6. In the terminal pane, enter the following command to establish an authenticated connection to your Azure subscription.
   `az login --output none`

7. When prompted, select or sign into your Azure subscription. Then return to Visual Studio Code and wait for the sign-in process to complete.
8. Run the following command to list Azure locations.
   `az account list-locations -o table`
9. In the output, find the **Name** value that corresponds with the location of your resource group (for example, for *East US* the corresponding name is *eastus*).
10. In the **setup.cmd** script, modify the **subscription_id**, **resource_group**, and **location** variable declarations with the appropriate values for your subscription ID, resource group name, and location name. Then save your changes.
11. In the terminal for the **02-search-skill** folder, enter the following command to run the script:
    `./setup`

    > **Note**: If the script fails, ensure you saved it with the correct variable names and try again.

12. When the script completes, review the output it displays and note the following information about your Azure resources (you will need these values later):
    - Storage account name
    - Storage connection string
    - Search service endpoint
    - Search service admin key
    - Search service query key
13. In the Azure portal, refresh the resource group and verify that it contains the Azure Storage account, Azure AI Services resource, and Azure AI Search resource.

Create a search solution

Now that you have the necessary Azure resources, you can create a search solution that consists of the following components:

- A **data source** that references the documents in your Azure storage container.
- A **skillset** that defines an enrichment pipeline of skills to extract AI-generated fields from the documents.
- An **index** that defines a searchable set of document records.
- An **indexer** that extracts the documents from the data source, applies the skillset, and populates the index.

In this exercise, you'll use the Azure AI Search REST interface to create these components by submitting JSON requests.

1. In Visual Studio Code, in the **02-search-skill** folder, expand the **create-search** folder and select **data_source.json** (data_source). This file contains a JSON definition for a data source named **margies-custom-data**.
2. Replace the **YOUR_CONNECTION_STRING** placeholder with the connection string for your Azure storage account, which should resemble the following:

   ```
   DefaultEndpointsProtocol=https;AccountName=ai102str123;AccountKey=12345abcdefg...==;
   EndpointSuffix=core.windows.net
   ```

*You can find the connection string on the **Access keys** page for your storage account in the Azure portal.*

3. Save and close the updated JSON file.
4. In the **create-search** folder, open **skillset.json** (skillset). This file contains a JSON definition for a skillset named **margies-custom-skillset**.
5. At the top of the skillset definition, in the **cognitiveServices** element, replace the **YOUR_AI_SERVICES_KEY** placeholder with either of the keys for your Azure AI Services resources.

   *You can find the keys on the **Keys and Endpoint** page for your Azure AI Services resource in the Azure portal.*
6. Save and close the updated JSON file.
7. In the **create-search** folder, open **index.json** (index). This file contains a JSON definition for an index named **margies-custom-index**.
8. Review the JSON for the index, then close the file without making any changes.
9. In the **create-search** folder, open **indexer.json** (indexer). This file contains a JSON definition for an indexer named **margies-custom-indexer**.
10. Review the JSON for the indexer, then close the file without making any changes.
11. In the **create-search** folder, open **create-search.cmd**. This batch script uses the cURL utility to **submit the JSON definitions to the REST interface for your Azure AI Search resource.**
12. Replace the **YOUR_SEARCH_URL** and **YOUR_ADMIN_KEY** variable placeholders with the **Url** and one of the **admin keys** for your Azure AI Search resource.

    You can find these values on the **Overview** and **Keys** pages for your Azure AI Search resource in the Azure portal.
13. Save the updated batch file.
14. Right-click the the **create-search** folder and select **Open in Integrated Terminal**.
15. In the terminal pane for the **create-search** folder, enter the following command run the batch script. `./create-search`
16. When the script completes, in the Azure portal, on the page for your Azure AI Search resource, select the **Indexers** page and wait for the indexing process to complete.

    You can select **Refresh** to track the progress of the indexing operation. It may take a minute or so to complete.

Search the index

Now that you have an index, you can search it.

1. At the top of the blade for your Azure AI Search resource, select **Search explorer**.
2. In Search explorer, in the **Query string** box, enter the following query string, and then select **Search**.

   ```
   search=London&$select=url,sentiment,keyphrases&$filter=metadata_author eq 'Reviewer'
   and sentiment eq 'positive'
   ```

   This query retrieves the **url**, **sentiment**, and **keyphrases** for all documents that mention *London* authored by *Reviewer* that have a positive **sentiment** label (in other words, positive reviews that mention London)

Create an Azure Function for a custom skill

The search solution includes a number of built-in AI skills that enrich the index with information from the documents, such as the sentiment scores and lists of key phrases seen in the previous task.

You can **enhance the index further by creating custom skills**. For example, it might be useful to identify the words that are used most frequently in each document, but no built-in skill offers this functionality.

To implement the **word count functionality as a custom skill, you'll create an Azure Function** in your preferred language.

> **Note**: In this exercise, you'll create a simple **Node.JS function** using the code editing capabilities in the Azure portal. In a production solution, you would typically use a development environment such as Visual Studio Code to create a function app in your preferred language (for example C#, Python, Node.JS, or Java) and publish it to Azure as part of a DevOps process.

1. In the Azure Portal, on the **Home** page, create a new **Function App** resource with the following settings:
    - **Subscription**: *Your subscription*
    - **Resource Group**: *The same resource group as your Azure AI Search resource*
    - **Function App name**: *A unique name*
    - **Publish**: Code
    - **Runtime stack**: Node.js
    - **Version**: 18 LTS
    - **Region**: *The same region as your Azure AI Search resource*
2. Wait for deployment to complete, and then go to the deployed Function App resource.
3. On the **Overview** page select **Create in Azure portal** option to create a new function with the following settings:
    - **Setup a development environment**"
        - **Development environment**: Develop in portal
    - **Select a template**"
        - **Template**: HTTP Trigger
    - **Template details**:
        - **New Function**: wordcount
        - **Authorization level**: Function
4. Wait for the *wordcount* function to be created. Then in its page, select the **Code + Test** tab.
5. Replace the default function code (custom skill to get top 10 most frequent words) with the following code:

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.body && req.body.values) {

        let vals = req.body.values;

        // Array of stop words to be ignored
        const stopwords = [
            '', 'i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
            "youre", "youve", "youll", "youd", 'your', 'yours', 'yourself',
            'yourselves', 'he', 'him', 'his', 'himself', 'she', "shes", 'her',
            'hers', 'herself', 'it', "its", 'itself', 'they', 'them',
            'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom',
            'this', 'that', "thatll", 'these', 'those', 'am', 'is', 'are', 'was',
            'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do',
            'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or',
            'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with',
            'about', 'against', 'between', 'into', 'through', 'during', 'before',
            'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out',
            'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here',
            'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each',
            'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not',
            'only', 'own', 'same', 'so', 'than', 'too', 'very', 'can', 'will',
            'just', "dont", 'should', "shouldve", 'now', "arent", "couldnt",
            "didnt", "doesnt", "hadnt", "hasnt", "havent", "isnt", "mightnt", "mustnt",
            "neednt", "shant", "shouldnt", "wasnt", "werent", "wont", "wouldnt"
        ];

        let res = { "values": [] };

        for (let rec in vals) {
            // Get the record ID and text for this input
            let resVal = { recordId: vals[rec].recordId, data: {} };
            let txt = vals[rec].data.text;

            // Remove punctuation and numerals
            txt = txt.replace(/[^ A-Za-z_]/g, "").toLowerCase();

            // Get an array of words
            let words = txt.split(" ");

            // Count instances of non-stopwords
            let wordCounts = {};
            for (let i = 0; i < words.length; ++i) {
                let word = words[i];
                if (!stopwords.includes(word)) {
                    wordCounts[word] = (wordCounts[word] || 0) + 1;
                }
            }

            // Convert wordCounts to an array
            let topWords = [];
            for (let word in wordCounts) {
                topWords.push([word, wordCounts[word]]);
            }

            // Sort in descending order of count
            topWords.sort((a, b) => b[1] - a[1]);

            // Get the first ten words
            resVal.data.text = topWords.slice(0, 9).map(value => value[0]);

            res.values[rec] = resVal;
        }

        context.res = {
            body: JSON.stringify(res),
            headers: {
                'Content-Type': 'application/json'
            }
        };

    } else {
        context.res = {
            status: 400,
            body: { "errors": [{ "message": "Invalid input" }] },
            headers: {
                'Content-Type': 'application/json'
            }
        };
    }
};
```

6. Save the function and then open the **Test/Run** pane.
7. In the **Test/Run** pane, replace the existing **Body** with the following JSON, which reflects the schema expected by an Azure AI Search skill in which records containing data for one or more documents are submitted for processing:

~~~
{
 "values": [
     {
         "recordId": "a1",
         "data": {
             "text": "Tiger, tiger burning bright in the darkness of the night.",
             "language": "en"
         }
     },
     {
         "recordId": "a2",
         "data": {
             "text": "The rain in spain stays mainly in the plains! That's where
you'll find the rain!",
             "language": "en"
         }
     }
 ]
}

  ```
~~~

8. Click **Run** and view the HTTP response content that is returned by your function. This reflects the schema expected by Azure AI Search when consuming a skill, in which a response for each document is returned. In this case, the response consists of up to 10 terms in each document in descending order of how frequently they appear:

```
{
 "values": [
     {
```

~~~
         "recordId": "a1",
         "data": {
             "text": [
                 "tiger",
                 "burning",
                 "bright",
                 "darkness",
                 "night"
             ]
         }
     },
     {
         "recordId": "a2",
         "data": {
             "text": [
                 "rain",
                 "spain",
                 "stays",
                 "mainly",
                 "plains",
                 "thats",
                 "youll",
                 "find"
             ]
         }
     }
 ]
}

  ```
~~~

9. Close the **Test/Run** pane and in the **wordcount** function blade, click **Get function URL**. Then copy the URL for the default key to the clipboard. You'll need this in the next procedure.

Add the custom skill to the search solution

Now you need to include your function as a custom skill in the search solution skillset, and map the results it produces to a field in the index.

1. In Visual Studio Code, in the **02-search-skill/update-search** folder, open the **update-skillset.json** file (update-skillset.json). This contains the JSON definition of a skillset.
2. Review the skillset definition. It includes the same skills as before, as well as a new **WebApiSkill** skill named **get-top-words**.
3. Edit the **get-top-words** skill definition to set the **uri** value to the URL for your Azure function (which you copied to the clipboard in the previous procedure), replacing **YOUR-FUNCTION-APP-URL**.
4. At the top of the skillset definition, in the **cognitiveServices** element, replace the **YOUR_AI_SERVICES_KEY** placeholder with either of the keys for your Azure AI Services resources.
   *You can find the keys on the **Keys and Endpoint** page for your Azure AI Services resource in the Azure portal.*
5. Save and close the updated JSON file.

6. In the **update-search** folder, open **update-index.json** (update-index.json). This file contains the JSON definition for the **margies-custom-index** index, with an additional field named **top_words** at the bottom of the index definition.
7. Review the JSON for the index, then close the file without making any changes.
8. In the **update-search** folder, open **update-indexer.json** (update-indexer.json). This file contains a JSON definition for the **margies-custom-indexer**, with an additional mapping for the **top_words** field.
9. Review the JSON for the indexer, then close the file without making any changes.
10. In the **update-search** folder, open **update-search.cmd**. This batch script uses the cURL utility to submit the updated JSON definitions to the REST interface for your Azure AI Search resource.
11. Replace the **YOUR_SEARCH_URL** and **YOUR_ADMIN_KEY** variable placeholders with the **Url** and one of the **admin keys** for your Azure AI Search resource. *You can find these values on the **Overview** and **Keys** pages for your Azure AI Search resource in the Azure portal.*
12. Save the updated batch file.
13. Right-click the the **update-search** folder and select **Open in Integrated Terminal**.
14. In the terminal pane for the **update-search** folder, enter the following command run the batch script. `./update-search`
15. When the script completes, in the Azure portal, on the page for your Azure AI Search resource, select the **Indexers** page and wait for the indexing process to complete.
    *You can select **Refresh** to track the progress of the indexing operation. It may take a minute or so to complete.*

Search the index

Now that you have an index, you can search it.

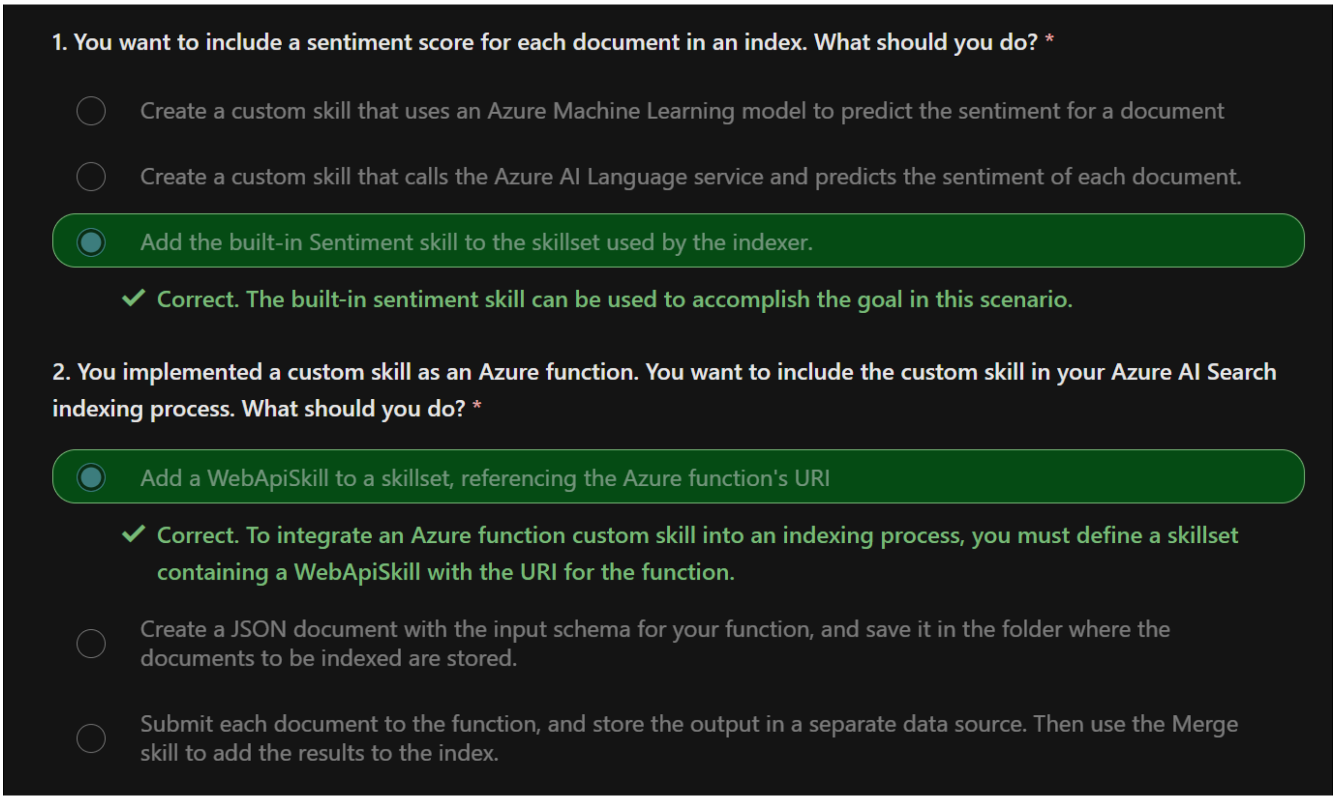
1. At the top of the blade for your Azure AI Search resource, select **Search explorer**.
2. In Search explorer, change the view to **JSON view**, and then submit the following search query: `{"search": "Las Vegas", "select": "url,top_words" }`

   This query retrieves the **url** and **top_words** fields for all documents that mention *Las Vegas*.

More information

To learn more about creating custom skills for Azure AI Search, see the Azure AI Search documentation.

Knowledge Check



1. You want to include a sentiment score for each document in an index. What should you do? *

○ Create a custom skill that uses an Azure Machine Learning model to predict the sentiment for a document

○ Create a custom skill that calls the Azure AI Language service and predicts the sentiment of each document.

● Add the built-in Sentiment skill to the skillset used by the indexer.

✓ Correct. The built-in sentiment skill can be used to accomplish the goal in this scenario.

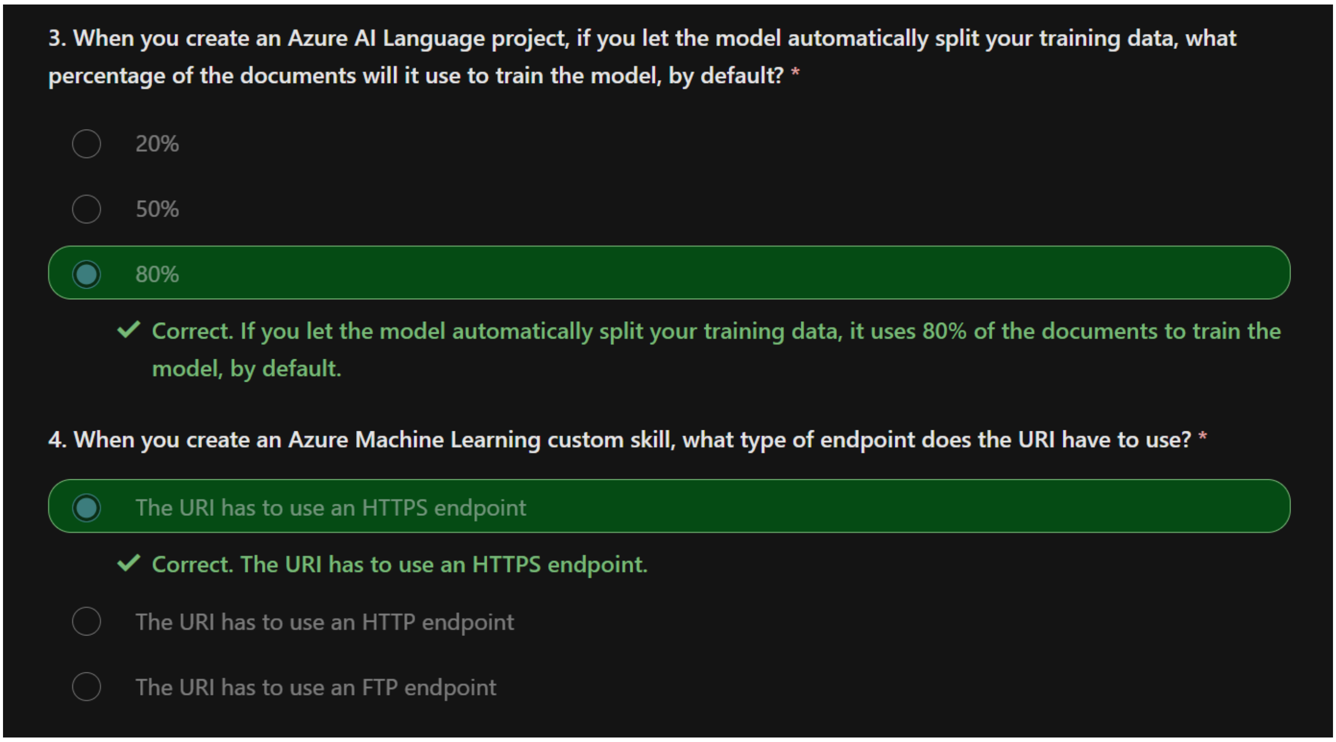
2. You implemented a custom skill as an Azure function. You want to include the custom skill in your Azure AI Search indexing process. What should you do? *

● Add a WebApiSkill to a skillset, referencing the Azure function's URI

✓ Correct. To integrate an Azure function custom skill into an indexing process, you must define a skillset containing a WebApiSkill with the URI for the function.

○ Create a JSON document with the input schema for your function, and save it in the folder where the documents to be indexed are stored.

○ Submit each document to the function, and store the output in a separate data source. Then use the Merge skill to add the results to the index.



3. When you create an Azure AI Language project, if you let the model automatically split your training data, what percentage of the documents will it use to train the model, by default? *

○ 20%

○ 50%

● 80%

✓ Correct. If you let the model automatically split your training data, it uses 80% of the documents to train the model, by default.

4. When you create an Azure Machine Learning custom skill, what type of endpoint does the URI have to use? *

● The URI has to use an HTTPS endpoint

✓ Correct. The URI has to use an HTTPS endpoint.

○ The URI has to use an HTTP endpoint

○ The URI has to use an FTP endpoint

Summary

In this module, you learned how to implement a custom skill for an Azure AI Search solution.

Custom skills can enrich your index in various ways, such as with custom Language models, machine learning, or document intelligence.

Custom skills prevents consumers of your Azure AI Search service from having to do repetitive processing or multiple service calls. This optimization saves both time and cost while delivering a better solution for search.

For more information about Azure AI Search, take a look at the service documentation.

✍️ Compiled by Kenneth Leung (2025)