**Python Language Modules**

The Python environment in Machine Learning uses the Anaconda environment, which is easy to use and also includes some of the more important and popular Python packages, including NumPy, SciPy, and scikit-learn. To run Python code using these packages, just write your code in the text editor of the Execute Python Script module.

The Python Language Modules category includes the following module:

***Execute Python Script:*** Executes a Python script from an Machine Learning experiment

**Execute Python Script**

With Python, you can perform tasks that aren't currently supported by existing Studio (classic) modules such as:

* Visualizing data using matplotlib
* Using Python libraries to enumerate datasets and models in your workspace
* Reading, loading, and manipulating data from sources not supported by the Import Data module

Machine Learning Studio (classic) uses the Anaconda distribution of Python, which includes many common utilities for data processing.

**How to use Execute Python Script**

The Execute Python Script module contains sample Python code that you can use as a starting point. To configure the Execute Python Script module, you provide a set of inputs and Python code to execute in the Python script text box.

1. Add the Execute Python Script module to your experiment.
2. Scroll to the bottom of the Properties pane, and for Python Version, select the version of the Python libraries and runtime to use in the script.

* Anaconda 2.0 distribution for Python 2.7.7
* Anaconda 4.0 distribution for Python 2.7.11
* Anaconda 4.0 distribution for Python 3.5 (default)

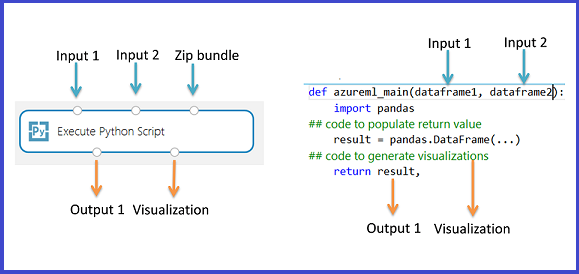
We recommend that you set the version before typing any new code. If you change the version later, a prompt asks you to acknowledge the change.

1. Add and connect on Dataset1 any datasets from Studio (classic) that you want to use for input. Reference this dataset in your Python script as DataFrame1.

Use of a dataset is optional, if you want to generate data using Python, or use Python code to import the data directly into the module.

This module supports addition of a second Studio (classic) dataset on Dataset2. Reference the second dataset in your Python script as DataFrame2.

Datasets stored in Studio (classic) are automatically converted to pandas data.frames when loaded with this module.



1. To include new Python packages or code, add the zipped file containing these custom resources on Script bundle. The input to Script bundle must be a zipped file already uploaded to your workspace.

Any file contained in the uploaded zipped archive can be used during experiment execution. If the archive includes a directory structure, the structure is preserved, but you must prepend a directory called src to the path.

1. In the Python script text box, type or paste valid Python script.

The Python script text box is pre-populated with some instructions in comments, and sample code for data access and output. You must edit or replace this code. Be sure to follow Python conventions about indentation and casing.

* The script must contain a function named azureml\_main as the entry point for this module.
* The entry point function can contain up to two input arguments: Param<dataframe1> and Param<dataframe2>
* Zipped files connected to the third input port are unzipped and stored in the directory, .\Script Bundle, which is also added to the Python sys.path.

Therefore, if your zip file contains mymodule.py, import it using import mymodule.

* A single dataset can be returned to Studio (classic), which must be a sequence of type pandas.DataFrame. You can create other outputs in your Python code and write them directly to Azure storage, or create visualizations using the Python device.

1. Run the experiment, or select the module and click Run selected to run just the Python script.

All of the data and code is loaded into a virtual machine, and run using the specified Python environment.

**Results**

The module returns these outputs:

* Results Dataset. The results of any computations performed by the embedded Python code must be provided as a pandas data.frame, which is automatically converted to the Machine Learning dataset format, so that you can use the results with other modules in the experiment. The module is limited to a single dataset as output. For more information, see Data Table.
* Python Device. This output supports both console output and display of PNG graphics using the Python interpreter.

**How to attach script resources**

The Execute Python Script module supports arbitrary Python script files as inputs, provided they are prepared in advance and uploaded to your workspace as part of a .ZIP file.

***Upload a ZIP file containing Python code to your workspace***

1. In the experiment area of Machine Learning Studio (classic), click Datasets, and then click New.
2. Select the option, From local file.
3. In the Upload a new dataset dialog box, click the dropdown list for Select a type for the new dataset, and select the Zip file (.zip) option.
4. Click Browse to locate the zipped file.
5. Type a new name for use in the workspace. The name you assign to the dataset becomes the name of the folder in your workspace where the contained files are extracted.
6. After you have uploaded the zipped package to Studio (classic), verify that the zipped file is available in the Saved Datasets list, and then connect the dataset to the Script Bundle input port. All files that are contained in the ZIP file are available for use during run time: for example, sample data, scripts, or new Python packages. If your zipped file contains any libraries that are not already installed in Machine Learning Studio (classic), you must install the Python library package as part of your custom script. If there was a directory structure present, it is preserved. However, you must alter your code to prepend the directory src to the path.

***Debugging Python code***

The Execute Python Script module works best when the code has been factored as a function with clearly defined inputs and outputs, rather than a sequence of loosely related executable statements.

This Python module does not support features such as Intellisense and debugging. If the module fails at runtime, you can view some error details in the output log for the module. However, the full Python stack trace is not available. Thus we recommend that users develop and debug their Python scripts in a different environment and then import the code into the module.

Some common problems that you can look for:

* Check the data types in the data frame you are returning back from ***azureml\_main.*** Errors are likely if columns contain data types other than numeric types and strings.
* Remove NA values from your dataset, using ***dataframe.dropna()*** on export from Python script. When preparing your data, use the Clean Missing Data module.
* Check your embedded code for indentation and whitespace errors. If you get the error, "IndentationError: expected an indented block

***Known limitations***

* The Python runtime is sandboxed and does not allow access to the network or to the local file system in a persistent manner.
* All files saved locally are isolated and deleted once the module finishes. The Python code cannot access most directories on the machine it runs on, the exception being the current directory and its subdirectories. When you provide a zipped file as resource, the files are copied from your workspace to the experiment execution space, unpacked, and then used. Copying and unpacking resources can consume memory.
* The module can output a single data frame. It's not possible to return arbitrary Python objects such as trained models directly back to the Studio (classic) runtime. However, you can write objects to storage or to the workspace. Another option is to use pickle to serialize multiple objects into a byte array and then return the array inside a data frame.

**R Language Modules**

**Requirements when using R**

Before using R script in Machine Learning Studio (classic), observe the following requirements:

* If you imported data that uses CSV or other formats, you cannot read the data directly in CSV format from your R code. Instead, use Convert to Dataset to prepare the data, before using it as input to an R module.
* When you attach any Machine Learning dataset as input to an R module, the dataset is automatically loaded into the R workspace as a data frame, with the variable name dataset. However, you can define additional data frames, or change the name of the default dataset variable within your R script.
* The R modules run in a protected and isolated environment within your private workspace. Within your workspace, you can create data frames and variables for use by multiple modules. However, you cannot load R data frames from a different workspace, or read variables created in a different workspace, even if that workspace is open in an Azure session. Also, you cannot use modules that have a Java dependency, or that require direct network access.

**Optimization for R scoring tasks**

The implementation of R in the Machine Learning Studio (classic) and workspace environment includes two principal components. One component coordinates script execution, and the other provides high-speed data access and scoring. The scoring component is optimized to enhance scalability and performance.

Therefore, R workspaces in Machine Learning Studio (classic) also support two kinds of scoring tasks, each optimized for different requirements. You typically use scoring on a file-by-file basis when you are building an experiment. You typically use the request response service (RRS) for very fast scoring, when you are scoring as part of a web service.

**R package and version support**

Machine Learning Studio (classic) includes over 500 of the most popular R packages. The R packages that you can select from depend on which R version you select for your experiment:

* CRAN R
* Microsoft R Open (MRO 3.2.2 or MRO 3.4.4)

Whenever you create an experiment, you must choose a single R version to run on, for all modules in your experiment.

**List of packages per version**

You can also add the following code to an Execute R Script module in your experiment, and run it to get a dataset containing package names and versions. Be sure to set the R version in the module properties to generate the correct list for your intended environment.

*data.set <- data.frame(installed.packages())*

*maml.mapOutputPort("data.set")*

**Extend experiments by using the R language**

There are many ways that you can extend your experiment by using custom R script or by adding R packages. Here are some ideas to get you started:

* Use R code to perform custom math operations. For example, there are R packages to solve differential equations, generate random numbers, or run Monte Carlo simulations.
* Apply custom transformations for data. For example, you might use an R package to perform interpolation on time series data, or perform linguistic analysis.
* Work with different data sources. The R script modules support an additional set of inputs, which can include data files, in zipped format. You might use zipped data files, along with R packages designed for such data sources, to flatten hierarchical data into a flat data table. You might also use these to read data from Excel and other file formats.
* Use custom metrics for evaluation. For example, rather than use the functions provided in Evaluate, you could import an R package, and then apply its metrics.

The following example demonstrates the overall process for how you can install new packages and use custom R code in your experiment.

**Split columns by using R**

Sometimes the data requires extensive manipulation to extract features. Suppose you have a text file that contains an ID followed by values and notes, all separated by spaces. Or suppose that your text file contains characters that are not supported by Machine Language Studio (classic).

The following sample illustrates how to install the needed packages, and split apart columns. You would add this code to the **Execute R Script** module.

*#install dependent packages*

*install.packages("src/concat.split.multiple/data.table\_1.9.2.zip", lib=".", repos = NULL, verbose = TRUE)*

*(success.data.table <- library("data.table", lib.loc = ".", logical.return = TRUE, verbose = TRUE))*

*install.packages("src/concat.split.multiple/plyr\_1.8.1.zip", lib=".", repos = NULL, verbose = TRUE)*

*(success.plyr <- library("plyr", lib.loc = ".", logical.return = TRUE, verbose = TRUE))*

*install.packages("src/concat.split.multiple/Rcpp\_0.11.2.zip", lib=".", repos = NULL, verbose = TRUE)*

*(success.Rcpp <- library("Rcpp", lib.loc = ".", logical.return = TRUE, verbose = TRUE))*

*install.packages("src/concat.split.multiple/reshape2\_1.4.zip", lib=".", repos = NULL, verbose = TRUE)*

*(success.reshape2 <- library("reshape2", lib.loc = ".", logical.return = TRUE, verbose = TRUE))*

*#install actual packages*

*install.packages("src/concat.split.multiple/splitstackshape\_1.2.0.zip", lib=".", repos = NULL, verbose = TRUE)*

*(success.splitstackshape <- library("splitstackshape", lib.loc = ".", logical.return = TRUE, verbose = TRUE))*

*#Load installed library*

*library(splitstackshape)*

*#Use library method to split & concat*

*data <- concat.split.multiple(maml.mapInputPort(1), c("TermsAcceptedUserClientIPAddress", "EmailAddress"), c(".", "@"))*

*#Print column names to console*

*colnames(data)*

*#Redirect data to output port*

*maml.mapOutputPort("data")*

**Execute R Script**

By adding R code to this module, you can perform a variety of customized tasks that are not available in Studio (classic). For example:

* Create custom data transformations
* Use your own metrics for evaluating predictions
* Build models using algorithms that are not implemented as standalone modules in Studio (classic)

**R versions supported in Studio (classic)**

Studio (classic) supports both the typical distribution of R that is available from CRAN, and Microsoft R Open (MRO), which includes all the base R packages, plus the Revo packages. You can specify which version of R to use in an experiment. However, you cannot install any other version of R into your workspace. We recommend that you determine which packages you need before choosing a distribution of R. Some packages are not compatible with both CRAN R and Microsoft R Open.

**How to configure Execute R Script**

To configure the Execute R Script module, you provide a set of optional inputs and the R code that is to be run in the workspace. You can also add files containing additional R code, if you prepare them in a zipped archive file for attachment to the Script bundle input.

To install any additional packages, include them in the zipped archive file.

1. Add the Execute R Script module to your experiment. You can find this module in Machine Learning Studio (classic), in the R Language Modules group.
2. Connect any inputs needed by the script. Inputs can include data, R packages that you added to your workspace in zipped file format, and additional R code.

* Dataset1: The first input is where you attach your main dataset (optional). The input dataset must be formatted as a CSV, TSV, or ARFF file, or you can connect an Machine Learning dataset.
* Dataset2: The second input (optional) supports addition of a second dataset. This dataset also must be formatted as a CSV, TSV, or ARFF file, or you can connect an Machine Learning dataset.
* Script Bundle: The third input, which is optional, takes a file in the .ZIP format. The zipped file can contain multiple files and multiple file types. For example, the zipped archive might contain R code in a script file, R objects for use by the script, an R package that itself was included in .ZIP format, or datasets in one of the supported formats.

1. Type R script into the R Script text box. This is the easiest way to work with the datasets on the input nodes.

To help you get started, the R Script text box is prepopulated with the following sample code, which you can edit or replace.

*# Map 1-based optional input ports to variables*

*dataset1 <- maml.mapInputPort(1) # class: data.frame*

*dataset2 <- maml.mapInputPort(2) # class: data.frame*

*# Contents of optional Zip port are in ./src/*

*# source("src/yourfile.R");*

*# load("src/yourData.rdata");*

*# Sample operation*

*colnames(dataset2) <- c(dataset1['nombre\_columna'])$nombre\_columna;*

*data.set = dataset2;*

*# You'll see this output in the R Device port.*

*# It'll have your stdout, stderr and PNG graphics device(s).*

*# Select data.frame to be sent to the output Dataset port*

*maml.mapOutputPort("data.set");*

1. Random Seed: Type a value to use inside the R environment as the random seed value. This parameter is equivalent to calling set.seed(value) in R code.
2. R Version: Select the version of R to load in the workspace.

* CRAN R 3.1.0: The Comprehensive R Archive Network Web site is the repository for the open source R language.
* Microsoft R Open 3.2.2: MRO is the enhanced distribution of R from Microsoft Corporation. It is an open source platform based on the open source R engine and fully compatible with all R packages, scripts and applications that work with the same version of R. However, MRO provides improved performance in comparison to the standard R distribution due to its use of high-performance, multi-threaded math libraries.
* You cannot install any other version of R into your workspace.
* Machine Learning supports multiple versions of R, but only one version can be used in any experiment.

1. Run the experiment, or select the Execute R Script module and click Run selected.

**Results**

The module can return multiple outputs.

* To get a dataset back, your R code should return a single R data.frame.
* You can display images in the R graphics device, which is displayed in the Machine Learning Studio (classic) log area.
* To persist images, you can write them to a file, or serialize them to a tabular format.
* You can save objects to your workspace.
* Standard messages and errors from R are returned to the module's log.

**Result Dataset**

This output contains the data frame that is generated by the R code in the module.

You can output only one data frame. Other tabular objects must be converted to a data frame using R functions. The data frame output by the module's R code is automatically converted to the internal Data Table format.

* To verify that the returned object is compatible with Studio (classic), use is.data.frame, which must return True.
* To return other R objects, try serializing the object into a byte array, or use a function that returns the desired data as a data.frame.

**R Device**

The R device supports both console output (standard output and standard error) and display of PNG graphics using the R interpreter.

* To view messages sent to the R console (Standard Output and Standard Error), right-click the module after it has finished running, select R Device, and select Visualize.
* To view graphics generated on the R Device port, right-click the module after it has finished running, select R Device, and select Visualize. To save images generated by the Execute R Script module, right-click the image and save a local copy.

**Create R Model**

To create an untrained model from an R script. You can base the model on any learner that is included in an R package in the Machine Learning environment. After you create the model, you can use Train Model to train the model on a dataset, like any other learner in Machine Learning. The trained model can be passed to Score Model to use the model to make predictions. The trained model can then be saved, and the scoring workflow can be published as a web service. In addition to using the Create R Model to save and re-use custom R modules, you can create your own implementation of a modeling and data management process using R, upload the files in zipped format to your workspace, and then register the package as a custom module.

**How to configure Create R Model**

Use of this module requires intermediate or expert knowledge of R. The module supports use of any learner that is included in the R packages already installed in Machine Learning.

This sample from the Azure AI Gallery implements a two-class Naïve Bayes classifier by using the popular e1070 package: + Create R Model. We recommend that you copy the example to your workspace and follow along.

1. Add these modules to your experiment: Create R Model, Train Model, Score Model.
2. In the Properties pane of Create R Model, provide these scripts:

* Trainer R script: The R script that you provide here is used to train the model. When you run the experiment, it is deployed to the Train Model module.
* Scorer R script: The R script that you provide on this input is for scoring only. when you run the experiment, it is deployed to the Score Model module.

1. The sample experiment also include the Execute Python Script module, which is used to plot graphs for model evaluation. This module is optional when publishing to a web service but useful when developing the experiment.

* To view the charts from the Python script, right-click the Python module, select Python Device, and select Visualize.
* To view just the model metrics, right-click the Python module, select Python Dataset, and select Visualize.

**Training script**

The following example demonstrates the type of code you might use in Trainer R script.

This script loads an R package, creates model using a learner from the package, and configures the feature and label columns using the predefined constants and functions provided in Create R Model.

*library(e1071)*

*features <- get.feature.columns(dataset)*

*labels <- as.factor(get.label.column(dataset))*

*train.data <- data.frame(features, labels)*

*feature.names <- get.feature.column.names(dataset)*

*names(train.data) <- c(feature.names, "Class")*

*model <- naiveBayes(Class ~ ., train.data)*

* The first line loads the R package, e1071, which contain the Naïve Bayes classifier algorithm we want to use. Since this is one of the packages pre-installed in the Machine Learning environment, you don’t need to download or install the package.
* The next lines get the feature columns and the label column from the dataset, and combine them into a new R data frame that is named train.data:

*features <- get.feature.columns(dataset)*

*labels <- as.factor(get.label.column(dataset))*

*train.data <- data.frame(features, labels)*

*feature.names <- get.feature.column.names(dataset)*

* Note use of these predefined functions:
* get.label.columns() returns the column that is selected as the class label in the Train Model module.
* get.feature.columns()selects the columns that were designated as features in the dataset.

By default, all columns except the label column are considered features in Studio (classic). Therefore, to mark specific columns as features, use Edit Metadata, or select a set of columns within the R script.

* get.feature.column.names(dataset) gets feature column names from the dataset.
* The names from the combined dataset are designated as the names for columns in train.data, and a temporary name Class is created for the label column.

*names(train.data) <- c(feature.names, "Class")*

* The final line of the code defines the Naïve Bayes classifier algorithm as a function of the variables (features) and outcomes (labels) in the train.data data frame.

model <- naiveBayes(Class ~ ., train.data)

Throughout the model creation, training, and scoring scripts, you must use the variable name model.

**Scoring script**

The following code illustrates the type of R code that you would provide in Scorer R script.

*library(e1071)*

*probabilities <- predict(model, dataset, type="raw")[,2]*

*classes <- as.factor(as.numeric(probabilities >= 0.5))*

*scores <- data.frame(classes, probabilities)*

* The first line loads the package.
* The second line computes the predicted probabilities for the scoring dataset by using the trained model from the training script, designated by the required variable name, model.
* The third line applies a threshold of 0.5 to probabilities when assigning the predicted class labels.
* The final line combines the class labels and probabilities into the output data frame, scores.
* The data frame that gets passed to the Score Model module must have the name scores.