MWSUG 2019 - Paper AL114

Deploying SAS® Viya® Docker images to the cloud - a step by step guide

Alan Zablocki, Ph.D., RedMane Technology, Chicago, IL

ABSTRACT

In this paper, we describe the steps to deploy a pre-built SAS® Viya® Docker image to the cloud. We use Microsoft Azure as the cloud environment to deploy our single programming-only image. We explain how to push massive Docker images to a private Azure Container Registry (ACR) and how to launch Azure Container Instance (ACI) to host a Docker image. We also show how to share code and data between Azure Storage and the SAS Viya Docker image, providing a complete data science working environment. Finally, we demonstrate how to use our SAS Viya environment for machine learning with code examples using SASPy, SAS Scripting Wrapper for Analytics Transfer (SWAT), TensorFlow and R.

INTRODUCTION

In our previous paper, "Deploying SAS Viya to Docker – a practical guide for data scientists", we showed how to build and deploy a local SAS Viya Docker image using sas-container-recipes, an open source GitHub project. Our local deployment had persistent storage and support for open source data science tools such as Jupyter Notebook, Python, and R. This enabled the local user to install additional packages and libraries for both Python and R.

In this paper, we show how to deploy this image to the cloud using Microsoft Azure (see De Capite 2018 for a discussion on Azure and other cloud providers). We show how to tag and push large Docker images to the Azure Container Registry (ACR), and how to create an Azure Container Instance (ACI) using Azure CLI, a command-line tool for managing Azure resources. We discuss the differences in persistent storage between the cloud and a local deployment, and outline the various error messages a user may encounter while deploying a container instance.

SETTING UP YOUR AZURE CLOUD ACCOUNT

In this section, we provide a complete introduction to creating the Azure services needed to deploy a Docker image to the cloud. We also explain the various authentication methods required for a successful launch.

CREATING AN AZURE SUBSCRIPTION

First, you will need to create an Azure account. Once you have an account, you will need to create an Azure subscription. This could be the free Azure subscription or a new pay-as-you-go one. If you are using a company account, you may need an administrator to create a subscription for you, and then give you owner permissions. To create a subscription, navigate to Subscriptions (see Figure 1).

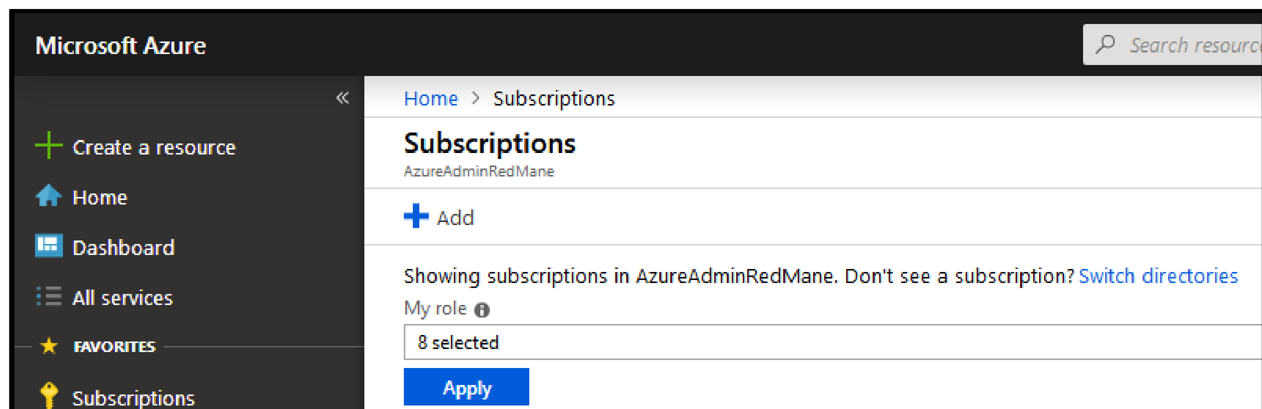


Figure 1: To add a subscription, find subscriptions under the All services tab in the top left of the menu.

After creating a subscription, you will need to install the Azure CLI

AZURE CLI

Although Azure's User Interface (UI) makes some actions quite simple to carry out, we have found that not all options are available through the UI. Unfortunately, some of the missing options are needed for a successful deployment, such as the ability to set "pull" permissions for the ACR. Follow these instructions to install Azure CLI on RedHat, Fedora or CentOS: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli-yum?view=azure-cli-latest.

Azure login and selecting subscriptions

Once you install the Azure CLI, log into Azure using the command *az login* and your Microsoft account credentials. This will open a browser, ask you to confirm your user email and password and take you back to the command prompt, at which point you will see a list of your subscriptions (see Figure 2).

```
[admin@RM-SAS-DOCKER-01 ~]$ az login
Note, we have launched a browser for you to login. For old experience with device cod
e, use "az login --use-device-code"
This tool has been deprecated, use 'gio open' instead.
See 'gio help open' for more info.
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "id": "your_first_subsc_number",
    "isDefault": false,
    "name": "Free Trial",
    "state": "Disabled",
    "tenantId": "your_first_tenant_id",
    "user": {
      "name": "your_user_email",
      "type": "user"
    }
  },
  {
    "cloudName": "AzureCloud",
    "id": "your_second_subsc_number",
    "isDefault": false,
    "name": "some_name",
    "state": "Enabled",
    "tenantId": "your_second_tenant_id",
    "user": {
      "name": "your_user_email",
      "type": "user"
    }
  }
]
```

Figure 2: The JSON output of the az login command will show all your subscriptions.

If you have more than one subscription, you can switch subscriptions using the *az account set* command. Remove the symbols < and > and run:

```
az account set --subscription <your_first_subsc_number>
```

CREATING AZURE SERVICES

The next two services that we will discuss can be created using the UI as well as the Azure CLI. In this section, we expand on the discussion in De Capite 2018, and show how to create a resource group and a

storage account using the command line. We recommend creating a text file with all the commands and pasting the commands in the command line as you follow this guide. You will want to set these five bash shell variables first:

```
ACI_SUBSCRIPTION=your_first_subsc_number
ACI_PERS_RESOURCE_GROUP=your_resource_group
ACI_PERS_STORAGE_ACCOUNT_NAME=chosenstoragename
ACI_PERS_SHARE_NAME=chosensharename
ACR_NAME=chosenregistryname
ACI_PERS_LOCATION=centralus
```

The variables above are your subscription ID, the name of your resource group, the storage account name, the file share name in your storage account, the registry name where you will push your Docker image and the location of all your services respectively.

Creating a resource group

To create a new resource group, set the correct subscription and run:

```
az account set --subscription $ACI_SUBSCRIPTION
az group create $ACI_PERS_RESOURCE_GROUP
```

Creating a storage account

To create a storage account, run:

```
az storage account create --resource-group $ACI_PERS_RESOURCE_GROUP --name
$ACI_PERS_STORAGE_ACCOUNT_NAME --location $ACI_PERS_LOCATION --sku
Standard_LRS --kind StorageV2 --access-tier Hot --subscription
$ACI_SUBSCRIPTION
```

You can modify storage kind and access tier options to match your requirements. In addition to the shell variables set above, we also set the STORAGE_ACCOUNT and STORAGE_KEY variables. You will use them when you set up persistent storage for your instance. Run the following code to set the necessary variables:

```
STORAGE_ACCOUNT=$(az storage account list --resource-group
$ACI_PERS_RESOURCE_GROUP --query
"[?contains(name,'$ACI_PERS_STORAGE_ACCOUNT_NAME')].[name]" --output tsv)
STORAGE_KEY=$(az storage account keys list --resource-group
$ACI_PERS_RESOURCE_GROUP  --account-name $STORAGE_ACCOUNT --query
"[0].value" --output tsv)
```

Once you create the storage account, navigate to storage account settings and make sure that secure transfer is not enabled under the Configuration options (see Figure 3).

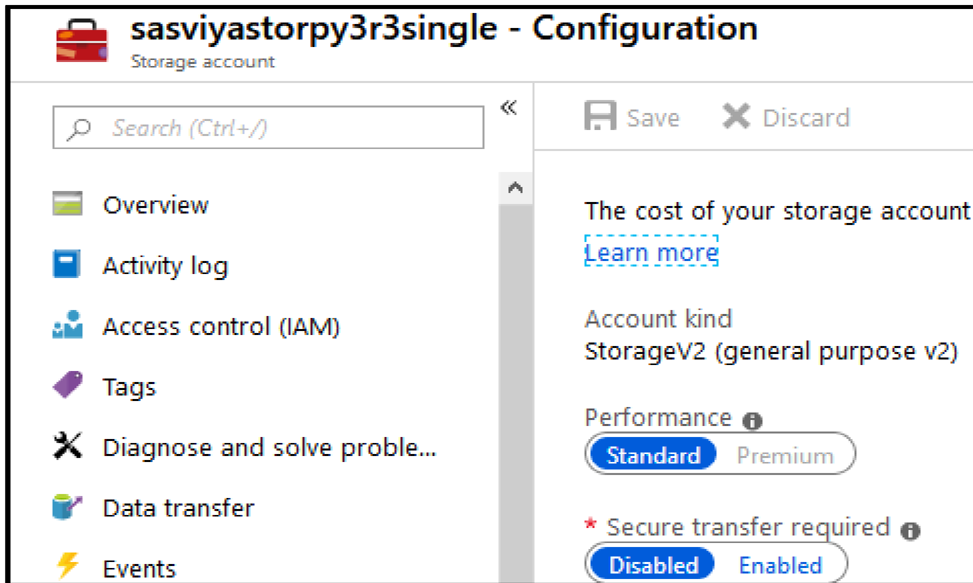


Figure 3: Set secure transfer to disabled under storage account settings.

Creating a file share

To create the file share where Docker will mount the folders inside the user */home* directory (in our example, cas and sasdemo), run:

```
az storage share create --name $ACI_PERS_SHARE_NAME --account-name
$ACI_PERS_STORAGE_ACCOUNT_NAME
```

If you navigate to *portal.azure.com*, you will now see that the resource group and the storage account have been created, and that the storage account contains a file share.

DEPLOYING YOUR DOCKER IMAGE TO AZURE CONTAINER INSTANCES

In the previous section, we set up a subscription, a resource group and a storage account to hold the data and code for the Docker image. In this section, we set up the ACR and we show how to push the image to the ACR. Finally, we walk through how to deploy and launch a SAS Viya Docker image in ACI.

CREATING AN AZURE CONTAINER REGISTRY

To create a container registry, navigate to Container registries and add a registry. Ensure that you enable the Admin user as shown in Figure 4. You will be able to launch multiple instances from a single registry that holds your Docker image.

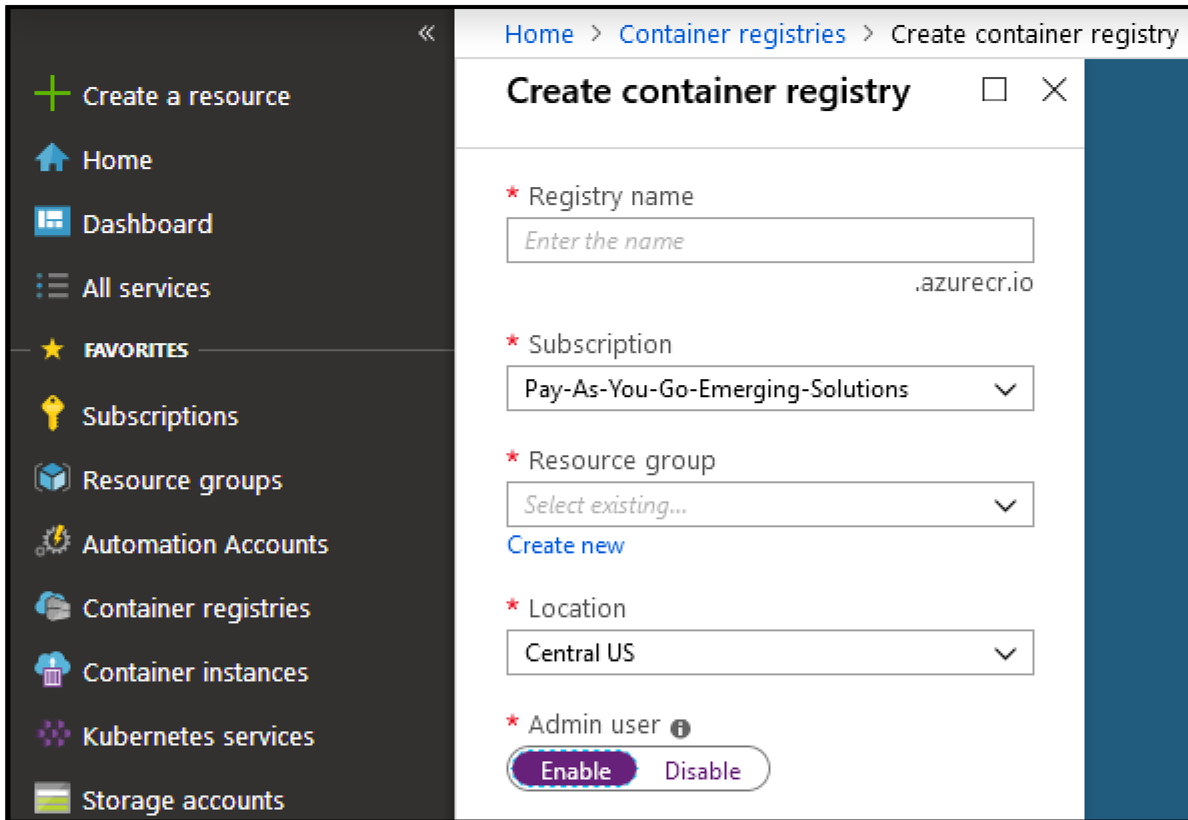


Figure 4: The container registry is created with the admin user enabled.

To create a registry via the Azure CLI, run:

```
az acr create --resource-group $ACI_PERS_RESOURCE_GROUP --name $ACR_NAME --
sku Standard --admin-enabled true --location $ACI_PERS_LOCATION
```

Note that when you create services in Azure, you may encounter errors if some of your chosen names do not conform with the various Azure naming conventions (sometimes the symbols - and _ are allowed, while other times, they are not).

PUSHING YOUR DOCKER IMAGES TO AZURE CONTAINER REGISTRY

Now that your container registry is set up, you will copy or push the Docker image to it. There are two ways to push a Docker image to an ACR: using *az acr login* and using *docker login*. For large images, we strongly advise the use of *docker login* over Azure CLI. When using *az acr login*, the authentication token expires before the push is complete, and the push fails.

Pushing using Docker login

To push the image to the ACR, first find out the image ID of your image with the command docker images. In our case, the image ID is 56c02aae8c9a. Next, create an alias to the fully qualified path of your registry using the *docker tag* command (see https://docs.docker.com/engine/reference/commandline/tag/ for more information). Once you have "tagged" the registry, login using *docker login*. Note that the username and password for the registry are not your Azure credentials but are instead found under Access Keys in the Settings as shown in Figure 5. Note that our registry name (stored under the shell variable ACR_NAME) is sassinglepy3r3 and the image name in the ACR will also be sassinglepy3r3 (indicated by the string immediately after azurecr.io/ in the docker tag command).

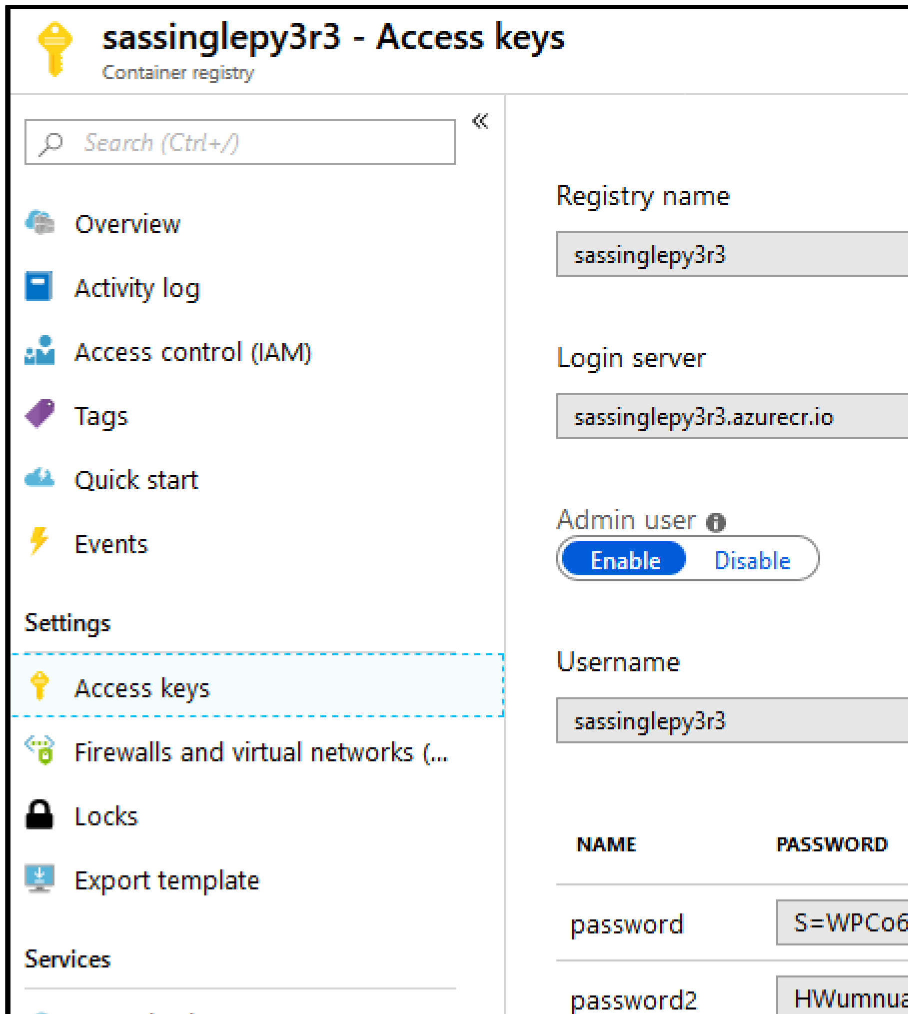


Figure 5: To push the Docker image to the registry you will need the username and password found under Access keys tab in the container registry settings.

The complete set of commands to push your Docker image is:

```
docker tag 56c02aae8c9a sassinglepy3r3.azurecr.io/sassinglepy3r3
docker login sassinglepy3r3.azurecr.io
docker push sassinglepy3r3.azurecr.io/sassinglepy3r3
```

The push will take some time to complete, and depending on the speed of your network, can take as long as an hour. If you need to interrupt the push for whatever reason, you can restart it. Layers that were pushed previously will show the message “Layer already exists”. Once the push is complete, you can look at the size of your image in the ACR Overview. The size in the ACR will be quite a bit smaller than the size on your local machine. In Figure 6, we show the compression when uploading our single SAS Viya image. The 25GB image on a local CentOS machine is just under 10GB in ACR.

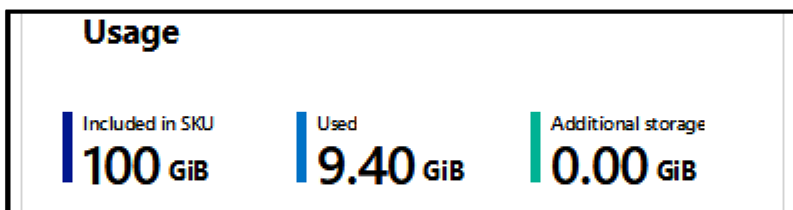


Figure 6: The size of the Docker image is compressed from 25GB to just under 10GB in ACR.

AZURE CONTAINER INSTANCES PRE-REQUISITES

You are almost ready to create and launch your Docker instance. Before you do that however, you will need to give permissions for the instance to pull the image from the ACR. This is not possible when using the UI. You will need a unique service principal for the instance you are launching and the associated username and password. These credentials will be needed when creating the instance using the command line.

Creating a service principal (non-optional)

To create a service principal, use the ACR name and choose a service principal name:

```
ACR_NAME=sassinglepy3r3
SERVICE_PRINCIPAL_NAME=acr-service-principal-sassinglepy3r3
```

Although you can use the same image to set up separate instances (use the same shell variable ACR_NAME), make sure that for each new instance you create, you use a unique SERVICE_PRINCIPAL_NAME. If you do not you might see a message about an existing application instance, and that it will be patched. This can lead to you being unable to restart a previous instance that was set up with that same SERVICE_PRINCIPAL_NAME.

To create the authentication variables, run:

```
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)
SP_PASSWD=$(az ad sp create-for-rbac --name http://$SERVICE_PRINCIPAL_NAME
--scopes $ACR_REGISTRY_ID --role acrpull --query password --output tsv)
SP_APP_ID=$(az ad sp show --id http://$SERVICE_PRINCIPAL_NAME --query appId
--output tsv)
=sassinglepy3r3
```

Notice that --role is set to acrpull which grants pull only permissions. Other options are acrpush (push and pull permissions) and owner (push, pull and assign roles).

Creating the instance

To create a container instance, you will need the previous shell variables and the following information:

- --name : instance name (this can to be anything)
- --image : full address of the registry including the name after the forward slash (/)
- --ip-address : set to Public
- --ports : list of ports, currently maximum of 5 (see comments below)
- --dns-name-label : this will create the fully qualified domain name (FQDN)
- --cpu : number of CPUs
- --memory : RAM in GB
- --os-type : must be Linux
- --azure-file-mount-path : The directory in the Docker container that will map to our file share

Unfortunately, ACI does not currently support port mapping, and it only supports a maximum of 5 ports. At the time of writing, the sas-container-recipes project may have changed the default Jupyter port from 8888 to 8080, so this may free up a port. Until this is confirmed you can choose to launch your image with port 5570, allowing the use of CAS, or launch with port 8787, the default RStudio-server port.

To launch an instance with CAS port 5570, use:

```
az container create --resource-group $ACI_PERS_RESOURCE_GROUP --name
sassinglepy3r3 --image sassinglepy3r3.azurecr.io/sassinglepy3r3 --ip-
address Public --ports 8080 5570 8888 20 443 80 --protocol TCP --dns-name-
```

```
label sassinglepy3r3 --cpu 4 --memory 16 --os-type Linux --registry-
username $SP_APP_ID --registry-password $SP_PASSWD --registry-login-server
sassinglepy3r3.azurecr.io --azure-file-volume-account-name
$ACI_PERS_STORAGE_ACCOUNT_NAME --azure-file-volume-account-key $STORAGE_KEY
--azure-file-volume-share-name $ACI_PERS_SHARE_NAME --azure-file-volume-
mount-path /home/
```

To launch an instance with port 8787, use:

```
az container create --resource-group $ACI_PERS_RESOURCE_GROUP --name
sassinglepy3r3 --image sassinglepy3r3.azurecr.io/sassinglepy3r3 --ip-
address Public --ports 8080 8787 8888 20 443 80 --protocol TCP --dns-name-
label sassinglepy3r3 --cpu 4 --memory 16 --os-type Linux --registry-
username $SP_APP_ID --registry-password $SP_PASSWD --registry-login-server
sassinglepy3r3.azurecr.io --azure-file-volume-account-name
$ACI_PERS_STORAGE_ACCOUNT_NAME --azure-file-volume-account-key $STORAGE_KEY
--azure-file-volume-share-name $ACI_PERS_SHARE_NAME --azure-file-volume-
mount-path /home/
```

The command prompt will show the message Starting, followed by Running. After a few minutes, you should be able to see the new instance listed in the UI under container instances (see Figure 7). If your new instance is not showing, check the terminal for any error messages.

Container instances
AzureAdminRedMane
Add Edit columns Refresh Assign tags
Subscriptions: All 2 selected – Don't see a subscription? Open Directory + Subscription settings
Filter by name... | All subscriptions | All resource groups | All locations
3 items

NAME	RESOURCE GROUP	LOCATION	STATUS	OS TYPE
sasdemo4cp	SASViya_Docker	Central US	Failed	Linux
sassinglepy3r2cas	SASViya_Docker	Central US	Running	Linux
sassinglepy3r3	SASViya_Docker	Central US	Creating	Linux

Figure 7: The container instance list, showing the instance we are creating, an instance that is currently running, and an instance with a failed status, which was stopped normally. The UI need not always show the correct or most recent status (even if you hit refresh).

Once the instance is created, a JSON output will appear in the command line with various instance properties, such as IP address and ports. If you click on the instance name (see Figure 7), you will be taken to the overview panel, where you can see its status and any warning messages as well as RAM, CPU and network traffic plots for the instance.

In our experience the UI may show a status that is often misleading, such as failed or repairing. It is typical to see a warning message such as the one shown in Figure 8. In this case, the instance creation is proceeding normally, but because pulling such a large Docker image takes time, the service returns a "waiting state and may not be running" message. Common instance states are waiting and repairing when the image pull is attempted. The state failed shows up often as well despite the creation proceeding normally. Clicking on the warning message takes you to a list of actions or events in the container instance creation.

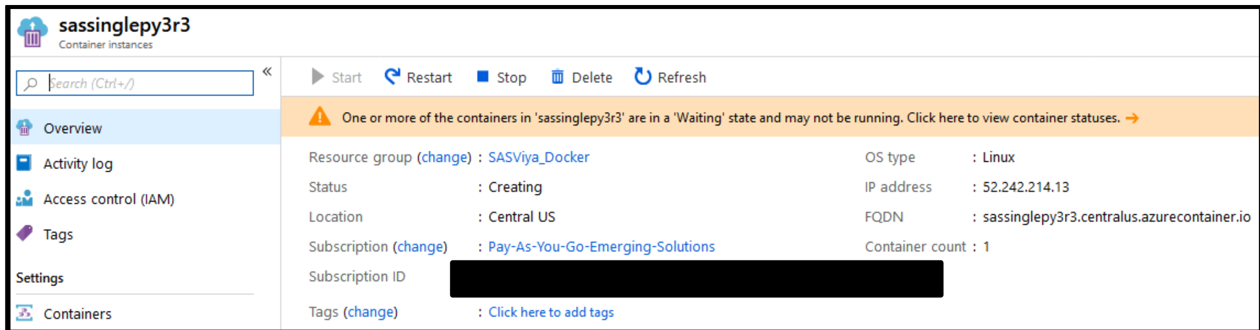


Figure 8: Overview panel showing the correct status for newly created instance. A warning message about waiting containers can be confusing at times.

Quite frequently the image pull may fail, and it can take a few attempts before the pull succeeds and the container starts running. It is not unusual to see counts for the image pull as high as 5-10. In Figure 9, we show the instance events with just a single image pull, before the instance starts running.

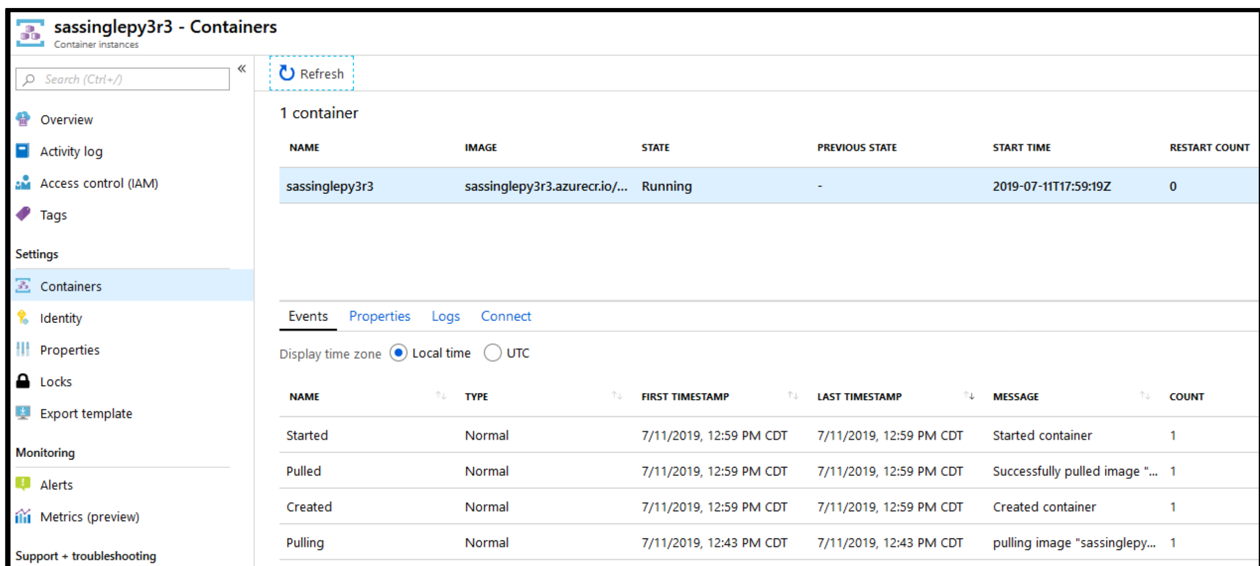
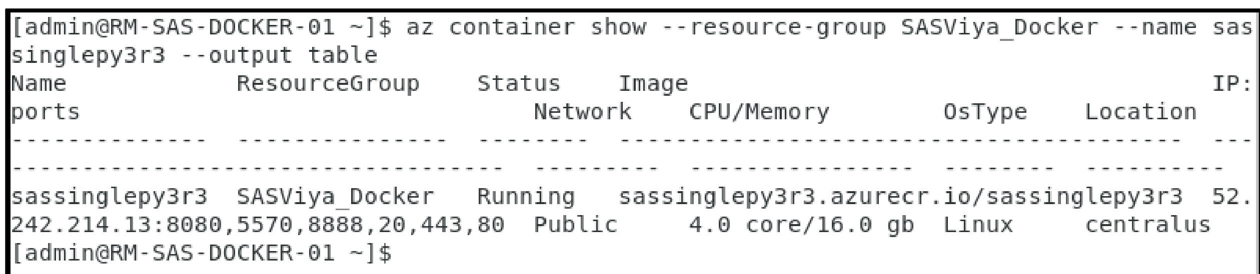


Figure 9: Container instance events, with a single image pull required to start our instance. Frequently, it can take 5-10 pulls before the instance starts running.

Another useful way to check the true status of the instance launch is with the Azure CLI command *az container show*. To check the status of an instance run:

```
az container show --resource-group $ACI_PERS_RESOURCE_GROUP --name
$ACR_NAME --output table
```

New instances will show a status of pending, which will change to running once launched (see Figure 10).



```
[admin@RM-SAS-DOCKER-01 ~]$ az container show --resource-group SASViya_Docker --name sas
singlepy3r3 --output table
Name           ResourceGroup    Status    Image                                       IP:
ports                                Network    CPU/Memory        OsType    Location
-------------  ---------------  --------  ----------------------------------------  ---
-----------------------------------  ---------  ----------------  --------  ----------
sassinglepy3r3  SASViya_Docker   Running   sassinglepy3r3.azurecr.io/sassinglepy3r3  52.
242.214.13:8080,5570,8888,20,443,80  Public     4.0 core/16.0 gb  Linux     centralus
[admin@RM-SAS-DOCKER-01 ~]$
```

Figure 10: Instance status, including the number of CPUs, the amount of RAM, IP address and ports.

Finally, you can also use the CPU and Memory plots in the instance overview panel to see if the instance launched correctly. Once the instance is up and running (and reachable), you will see a characteristic step in the memory usage indicating a running Docker container. The memory usage should be between 1-3GB for a freshly launched image as shown in Figure 11.

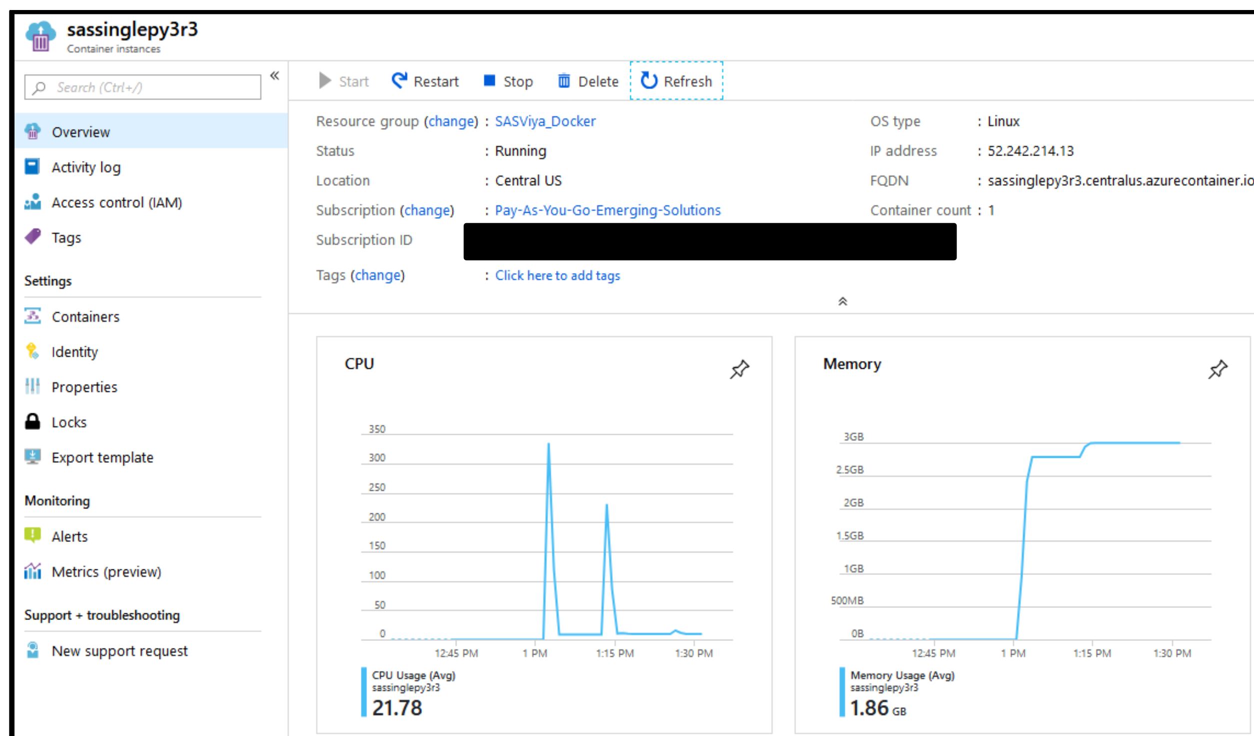


Figure 11: Instance overview, showing the step-like increase in RAM indicating a running Docker container.

Although the above image launched very quickly (15-20 minutes), it can take up to 90 minutes. The most frequent reason for the delay is the initial image pull. If the IP address is not reachable after 90 minutes, try restarting the instance in the UI.

Credentials summary

To ensure a successful deployment, you will need to use three separate sets of credentials:

- First, log into Azure with Microsoft Account credentials using *az login*
- Second, to push the image, use the username and access key for the ACR using *docker login*
- Third, use a service principal with its own username(id) and password for the instance to pull the image from ACR

INSTANCE COSTS

Based on our instance, the daily cost for keeping an instance is about $8. It costs roughly $6 per day to run the instance, with an additional $2 per day coming from ACR and the Storage account. One way to minimize these costs, is to use scheduling automation runbooks that can stop and start instances during working hours, yielding a savings of around 60%.

ERRORS AND UNUSUAL BEHAVIORS

In addition to the errors outlined in the previous section, we also experienced these behaviors:

Jupyter kernel stops or does not start

In Azure, you can use the FQDN to access Jupyter without explicitly including the port. If you use *http://sassinglepy3r3.centralus.azurecontainer.io/Jupyter* to launch Jupyter, the kernel will likely die or

never start. Instead, use the IP address while explicitly specifying the port 8888, as in *http://<ip-address>:8888/Jupyter*. A recent change in the sas-container-recipes GitHub project may mean that Jupyter is now accessible on port 8080. This means that the FQDN address for Jupyter may work, while previously, the kernel would crash.

Persistent storage location changes for Jupyter Notebook folder

In our previous paper, “Deploying SAS Viya to Docker – a practical guide for data scientists”, we deployed a local SAS Viya Docker image. In that Docker image, the sasdemo user had the default folder for Jupyter Notebooks set to */home/sasdemo/jupyter*. When we deployed to Azure, Jupyter notebooks were saved under */home/cas/jupyter* instead. As a result, user saved packages for R and Python were saved in two different folders.

USING YOUR DATA SCIENCE ENVIRONMENT

With your SAS Viya Docker instance live, you can begin to use it for various data science tasks. Before we dive into some code examples, we discuss persistent storage, libraries and kernel tests.

PERSISTENT STORAGE FOR JUPYTER NOTEBOOK

Recall that when you created your instance, you used the flag *--azure-file-mount-path* to map the Azure storage file share, here called sasdockerpy3r3single, to the */home* folder in the Docker image. Therefore, the Azure file share contains two folders from the Docker */home* directory, namely cas and sasdemo, as shown in Figure 12. Anything saved to these two folders will persist, even when you stop or restart your Azure container instance.

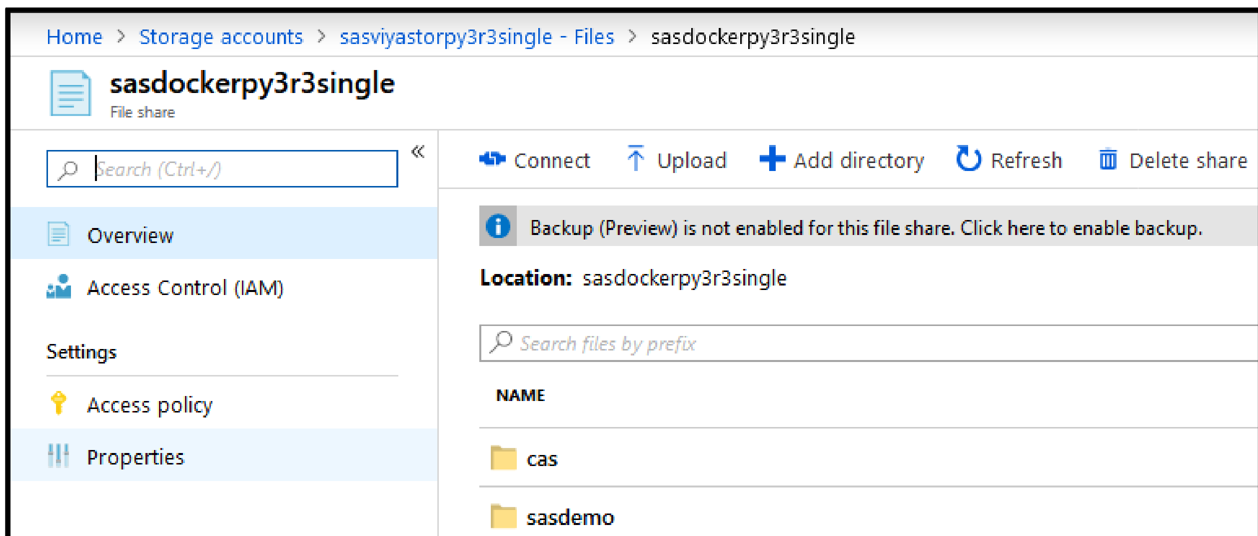


Figure 12: The Azure file share holds the contents of the Docker image /home directory.

The default directory for your Azure Docker instance, where Jupyter stores notebooks is *cas/jupyter/* as shown in Figure 13. This is different from a local deployment, where the default directory was *sasdemo/jupyter*. The file share corresponds to the user’s */home* folder. Any Jupyter notebook-regardless of kernel-will be saved under */cas/jupyter*.

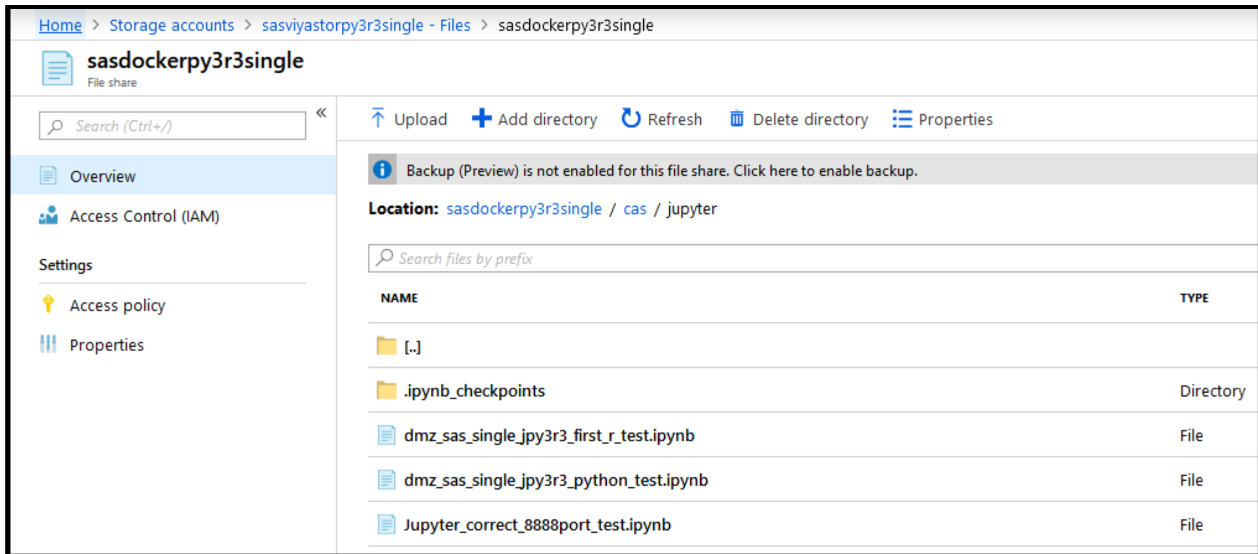


Figure 13: Jupyter Notebooks are saved under cas folder instead of sasdemo when deploying in Azure.

PERSISTENT STORAGE FOR PYTHON LIBRARIES

Any library installed with pip using the *--user* flag, will be installed under */cas/.local/lib/python3.6/site-packages* as shown in Figure 14. We do not advise installing with pip as a root user inside the Docker image.

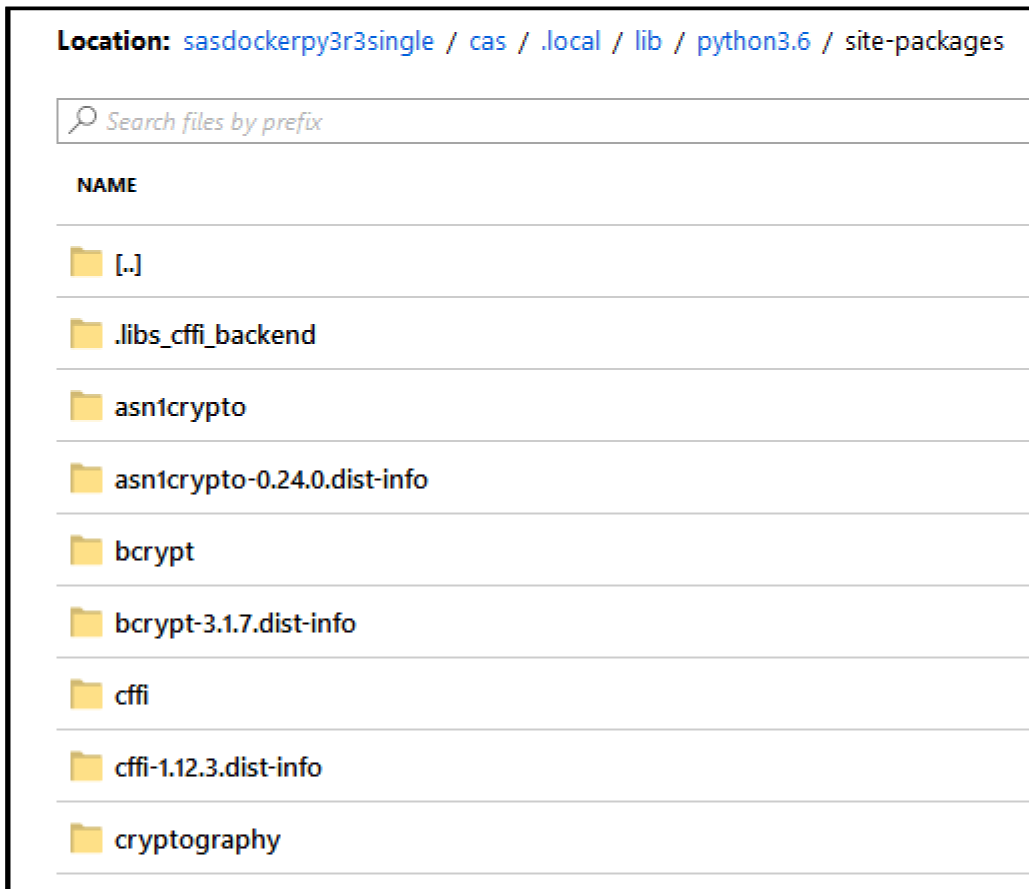


Figure 14: Python libraries installed with the flag --user persist and are saved in the cas folder.

PERSISTENT STORAGE FOR R LIBRARIES

The default library paths for R packages differ between RStudio and the R kernel in Jupyter. In RStudio, the default library path is set to a local user folder (*/home/sasdemo*), while R sessions in Jupyter and the bash shell use a system folder (*/usr*). In this section, we show how to configure Jupyter to use a local library path, so that installed packages persist in Azure storage. In the next section, we discuss RStudio settings.

RStudio Server

Before you can run R code in RStudio server, you will have to start the server. To do this, you will need to exec into the image. In Azure, you can do this by navigating to containers under the instance settings and clicking the connect tab. Choose the /bin/bash option as shown in Figure 15.

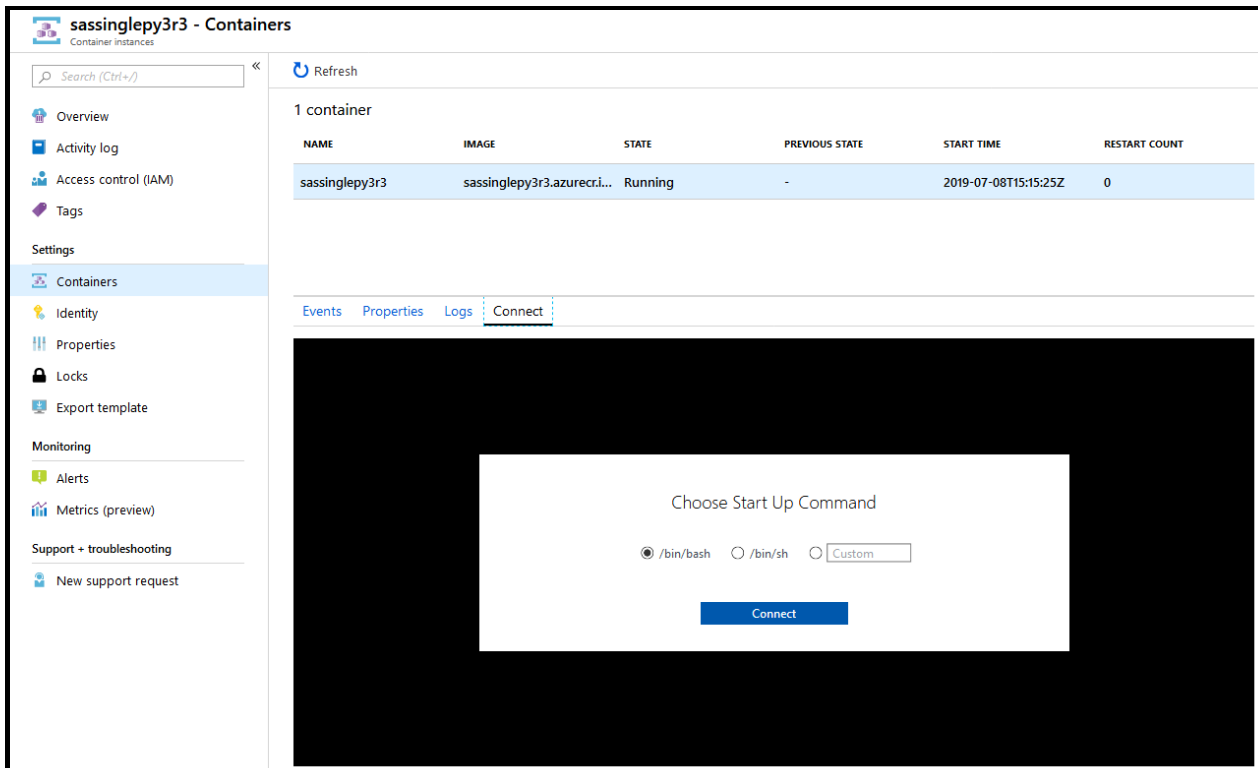


Figure 15: You can exec into an Azure container instance and start services such as RStudio server under Settings > Containers > Connect.

Once you connect to your running Docker instance (you will log in as a root user, shown by #), run the command sudo rstudio-server start to start RStudio on port 8787. If you launched your instance with port 5570 (to allow CAS connections), you will need to either re-configure the server or launch a new instance with port 8787 instead of port 5570. To re-configure the server, create the file */etc/rstudio/rserver.conf* (if it doesn't already exist) and add an entry for port 8787:

```
www-port=5570
```

In the future, Azure may allow more than five ports or allow port mapping, which should make this part of the setup easier. If you navigate to *<ip-address>:8787*, you should see the login screen with the default sasdemo user and password (see Figure 16).

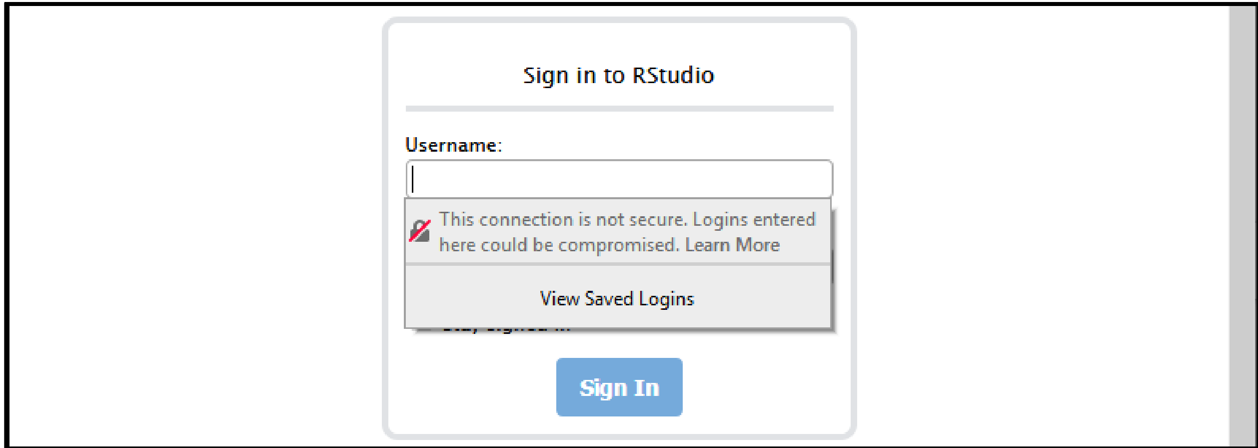


Figure 16: If the RStudio server is running and port 8787 is open/available, you should see the login prompt.

Once you login to RStudio server, you will see the familiar RStudio interface where you can load libraries and execute R code (see Figure 17). Use the *.libPaths()* command to show all the library paths accessible in the R session.

```
52.230.228.3:8787

File  Edit  Code  View  Plots  Session  Build  Debug  Profile  Tools  Help
Go to file/function    Addins

R packages available

Packages in library '/home/sasdemo/R/x86_64-redhat-linux-gnu-library/3.5':

esquisse                   Explore and Visualize Your Data Interactively
ggthemes                   Extra Themes, Scales and Geoms for 'ggplot2'
miniUI                     Shiny UI Widgets for Small Screens
modeltools                 Tools and Classes for Statistical Models
NLP                        Natural Language Processing Infrastructure
shinyWidgets               Custom Inputs Widgets for Shiny
slam                       Sparse Lightweight Arrays and Matrices
tm                         Text Mining Package
topicmodels                Topic Models

Packages in library '/usr/lib64/R/library':

askpass                    Safe Password Entry for R, Git, and SSH
assertthat                 Easy Pre and Post Assertions

Console  Terminal  Jobs
~/
>
> .libPaths()
[1] "/home/sasdemo/R/x86_64-redhat-linux-gnu-library/3.5" "/usr/lib64/R/library"
[3] "/usr/share/R/library"
> library()
Warning message:
In library() : library '/usr/share/R/library' contains no packages
> library(topicmodels)
```

Figure 17: RStudio server interface accessible in the Docker container on port 8787, which must be open when you first create the instance in Azure.

Note how RStudio automatically configured the path */home/sasdemo/R/x86_64-redhat-linux-gnu-library/3.5* to store R packages. This is the path that you need to add to a Jupyter R session.

Configuring Jupyter Notebook R kernel

By default, an R session in Jupyter Notebook can only access two library paths (see Figure 18). The path set by RStudio is not yet available. To fix this, add a *.Rprofile* file to the */home/cas* folder in the Docker

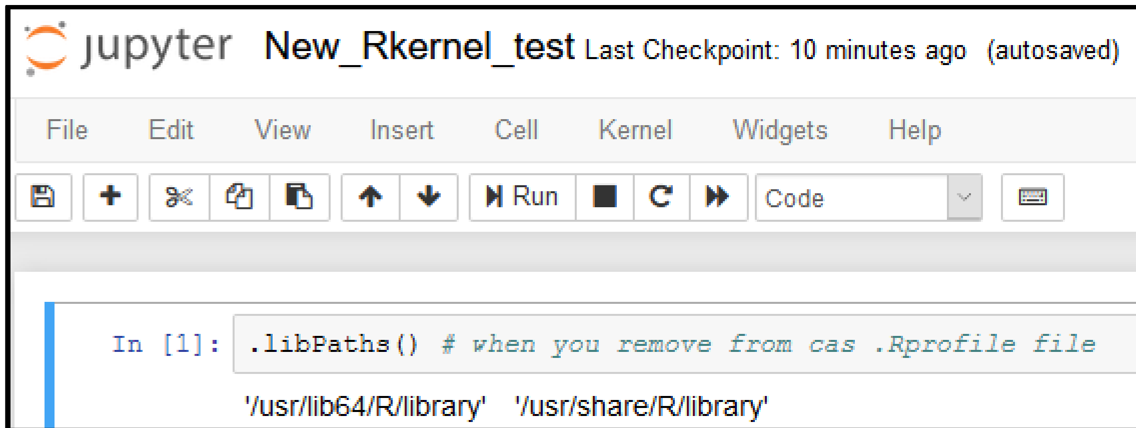


Figure 18: The default library paths when you first launch Jupyter with IRkernel.

image. You can do this by either uploading the file to the cas folder in the Azure file share or by logging into the image and opening a new file with a text editor (e.g., *vi .Rprofile*) and typing:

```
.libPaths("/home/sasdemo/R/x86_64-redhat-linux-gnu-library/3.5")
```

Remember to make sure that the path corresponds to your system and your version of R. In this paper, we use R version 3.5. Save the file and restart the R kernel. Running the *.libPaths()* command should now return the same three libraries that we saw when we used RStudio server. It's worth noting that in a local deployment, the *.Rprofile* file was in the */home/sasdemo* folder, and not */home/cas*. This is due to the directory swap we described in Persistent storage location changes for Jupyter Notebook folder.

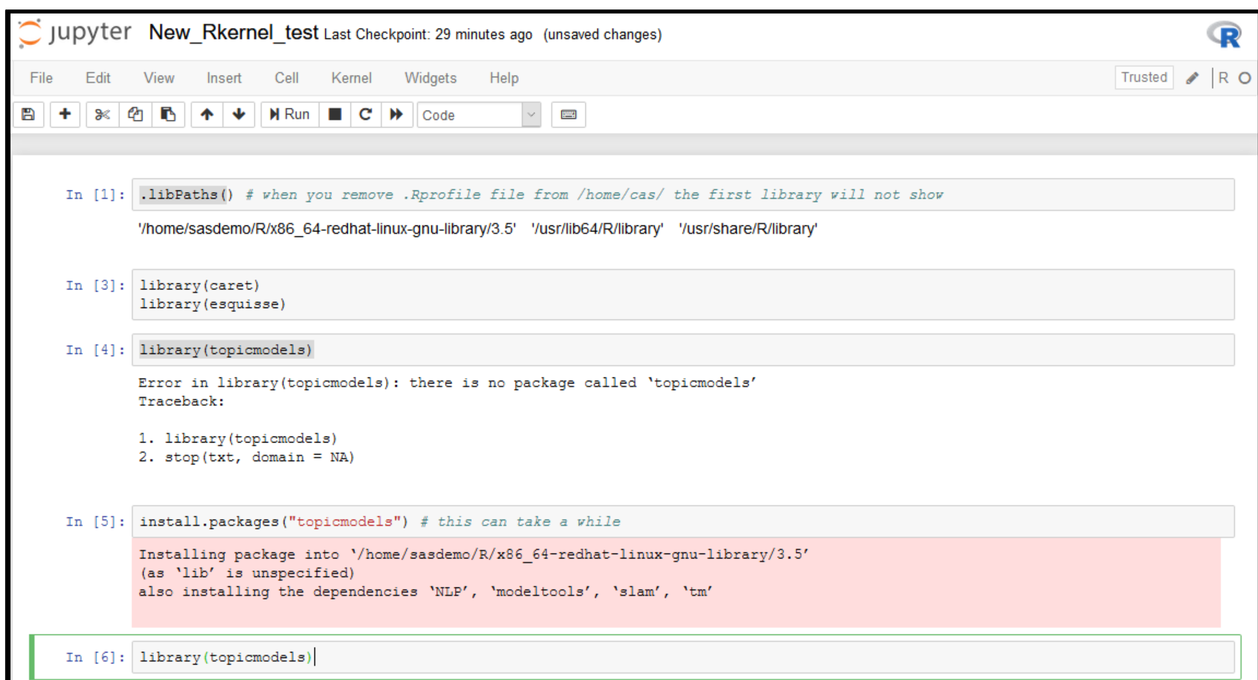


Figure 19: Library paths now include the local library folder, if you use the *.Rprofile* file in the cas folder.

With the local path now available you can load packages that are installed in any of the three libraries, and you can install new packages, such as topicmodels, as shown in Figure 19.

If you did not install RStudio server or if the server is not available in your instance due to port settings, you should see a prompt from Jupyter about setting a local path when you attempt to install a new package. You should see the option to set the exact same folder path as we saw with RStudio, namely */home/sasdemo/R/x86_64-redhat-linux-gnu-library/3.5*. If you are not given this option, you can create your own folder (for example *sasdemo/my_libs/R/x86_64-redhat-linux-gnu-library/3.5*) and use that path in the *.Rprofile* file instead.

DATA SCIENCE EXAMPLES

In this section, we demonstrate how to use your SAS Viya environment for data analytics and machine learning with short code examples using SASPy, SAS Scripting Wrapper for Analytics Transfer (SWAT), TensorFlow, and R.

SAS code in SAS Studio

To access SAS Studio, use the IP address found in the instance overview, e.g., *http://52.242.214.13/* or the FQDN, which is *http://sassinglepy3r3.centralus.azurecontainer.io/* in our example (see Figure 20).

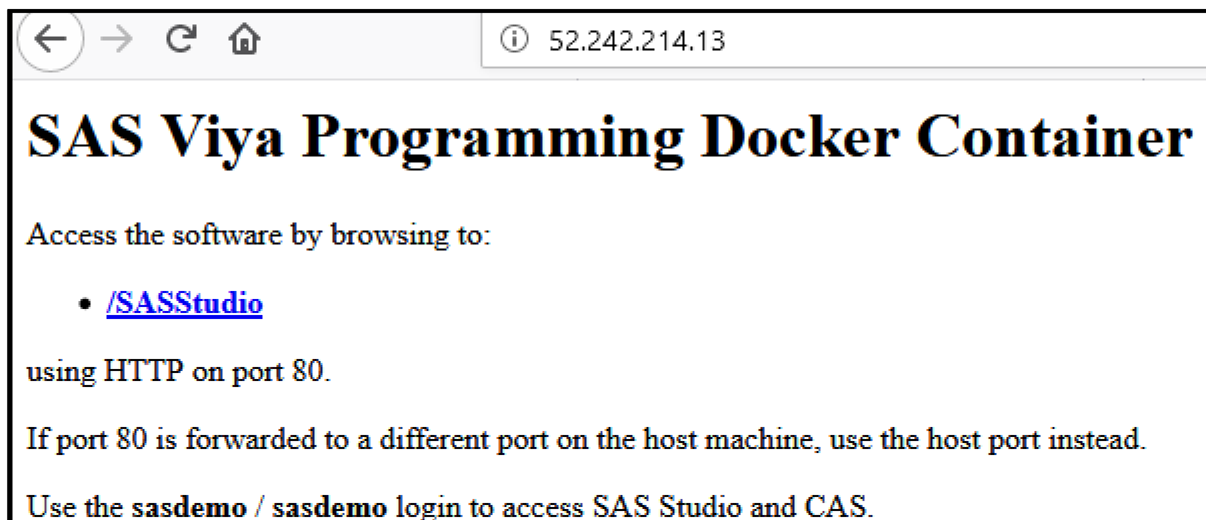


Figure 20: Welcome page with an access link to SAS Studio.

To log into a SAS Viya session, click the link on the access page or go *to http://52.242.214.13/SASStudio* (see Figure 21).

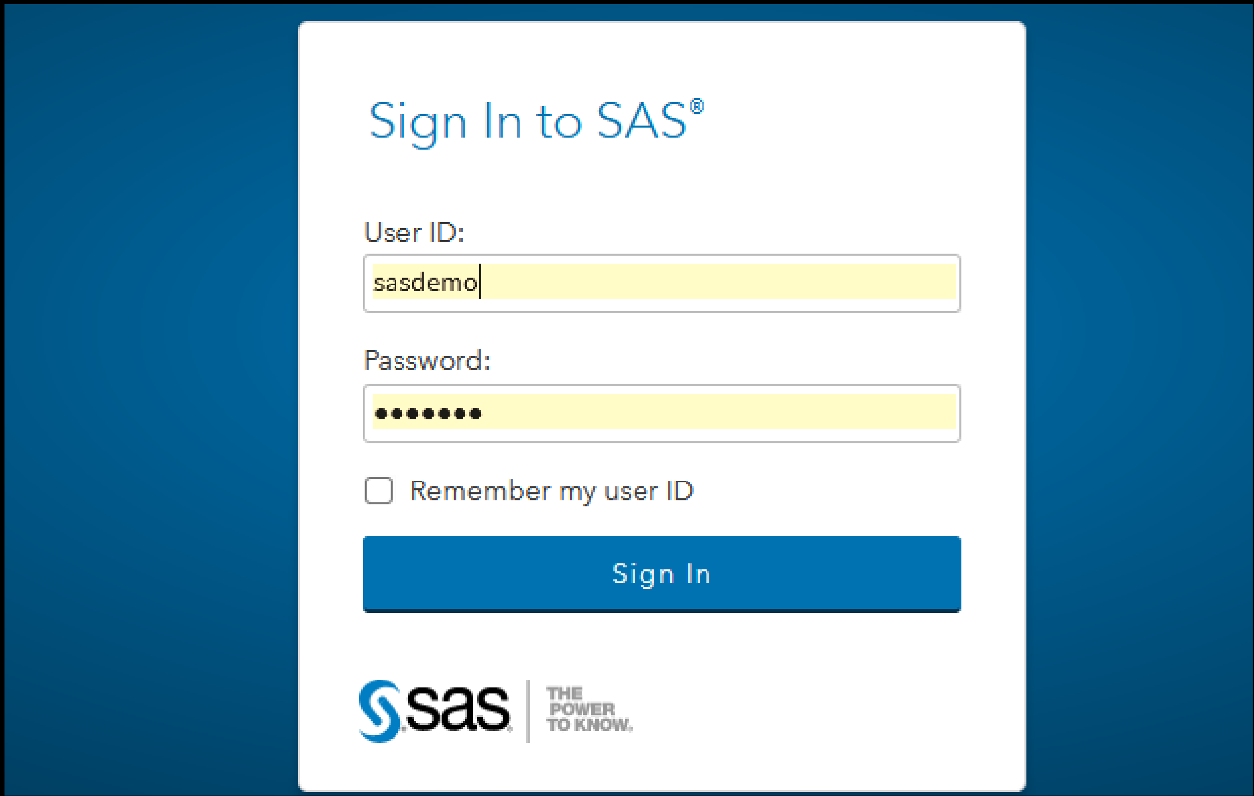


Figure 21: SAS login screen.

Once logged in, you can start coding in SAS (see Figure 22). Since you configured persistent storage, your data and code will be accessible through Azure Storage.

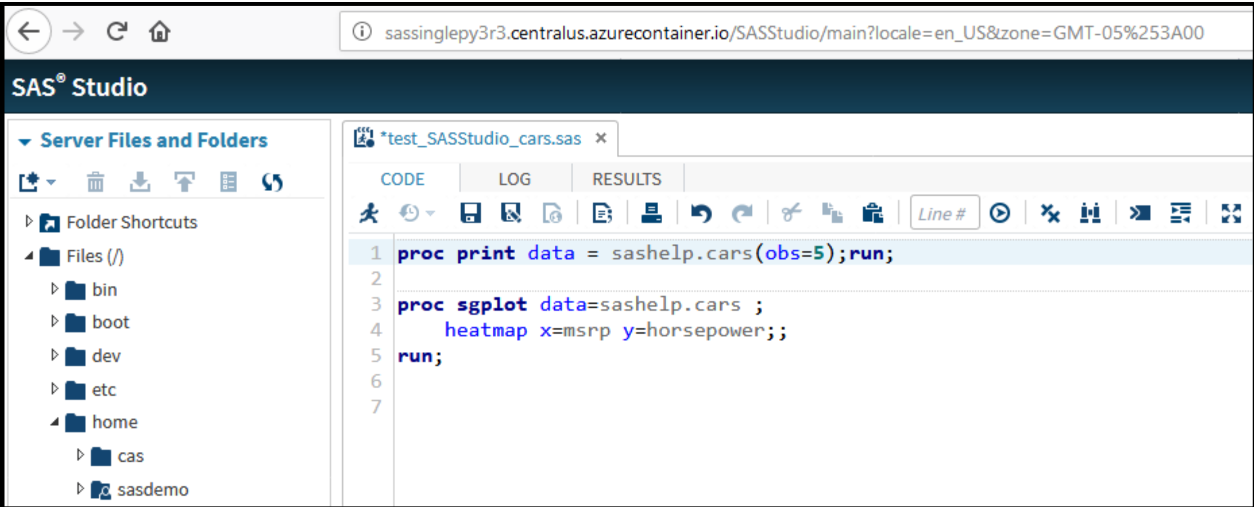


Figure 22: SAS code example using SAS Studio.

Licensed SAS products can be found under the Task and Utilities menu, where for example, you can access SAS Viya Supervised Learning methods such as Linear Regression (see Figure 23).

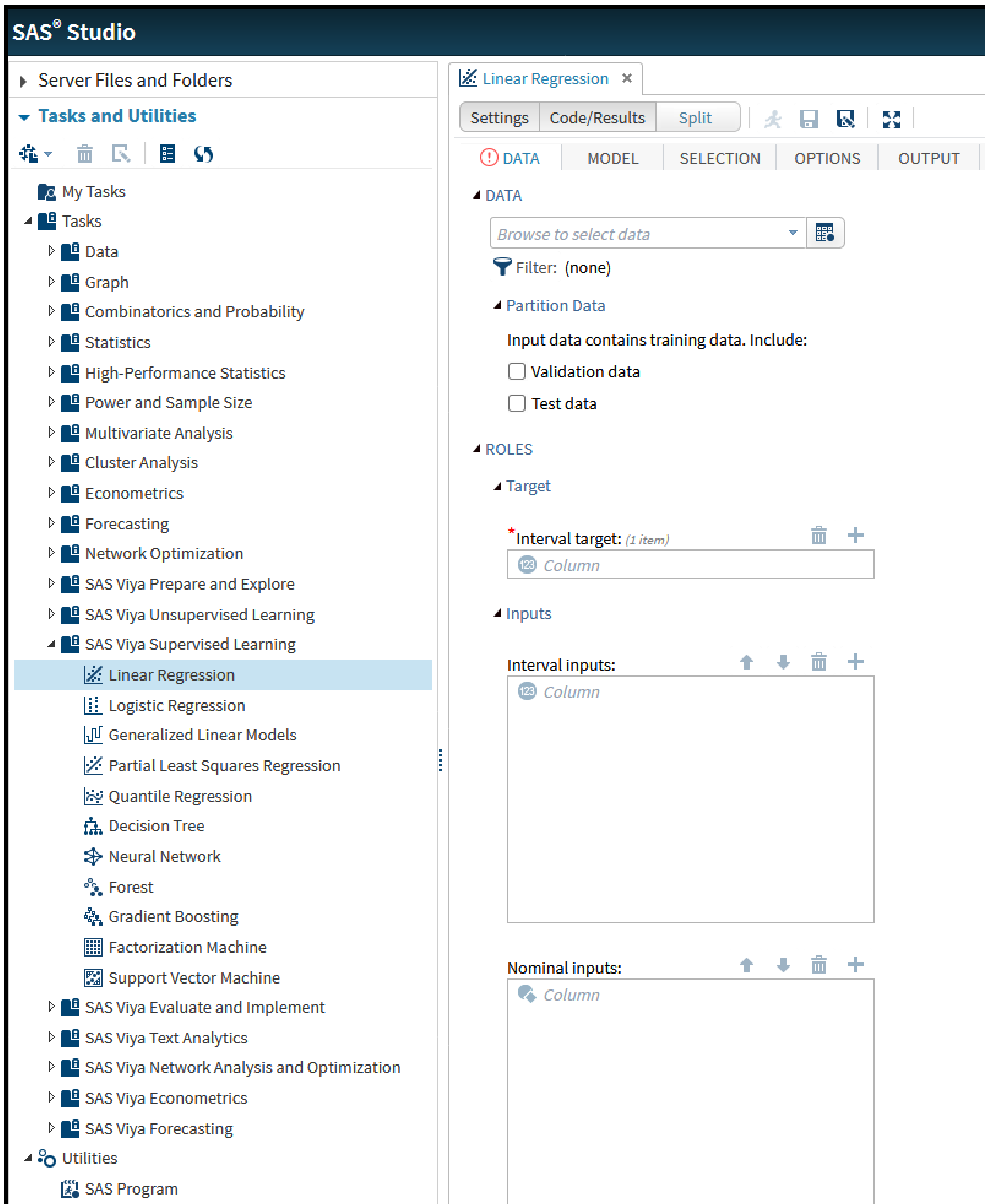


Figure 23: Access to various SAS products including SAS® Visual Data Mining and Machine Learning.

SAS code in Jupyter

You can execute SAS code using the SAS kernel in Jupyter Notebook (see Figure 24), a development environment and a visualization tool in one. This makes it easy to show code and results side by side.

jupyter test_SAS_kernel_cars Last Checkpoint: Last Tuesday at 10:22 AM (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Run Code

In [3]:
```
proc print data = sashelp.cars(obs=2);run;
```

Out[3]:

The SAS System

Obs	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Ho
1	Acura	MDX	SUV	Asia	All	$36,945	$33,337	3.5	6	
2	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2.0	4	

In [2]:
```
proc sgplot data=sashelp.cars ;
    heatmap x=msrp y=horsepower;;
run;
```

Out[2]:

Figure 24: SAS code example with SAS kernel in Jupyter Notebook.

Python and SASPy

SASPy is an open source Python package that allows access to SAS datasets and products. In your SAS Viya Docker image, it is already installed and configured for use in Jupyter Notebooks.

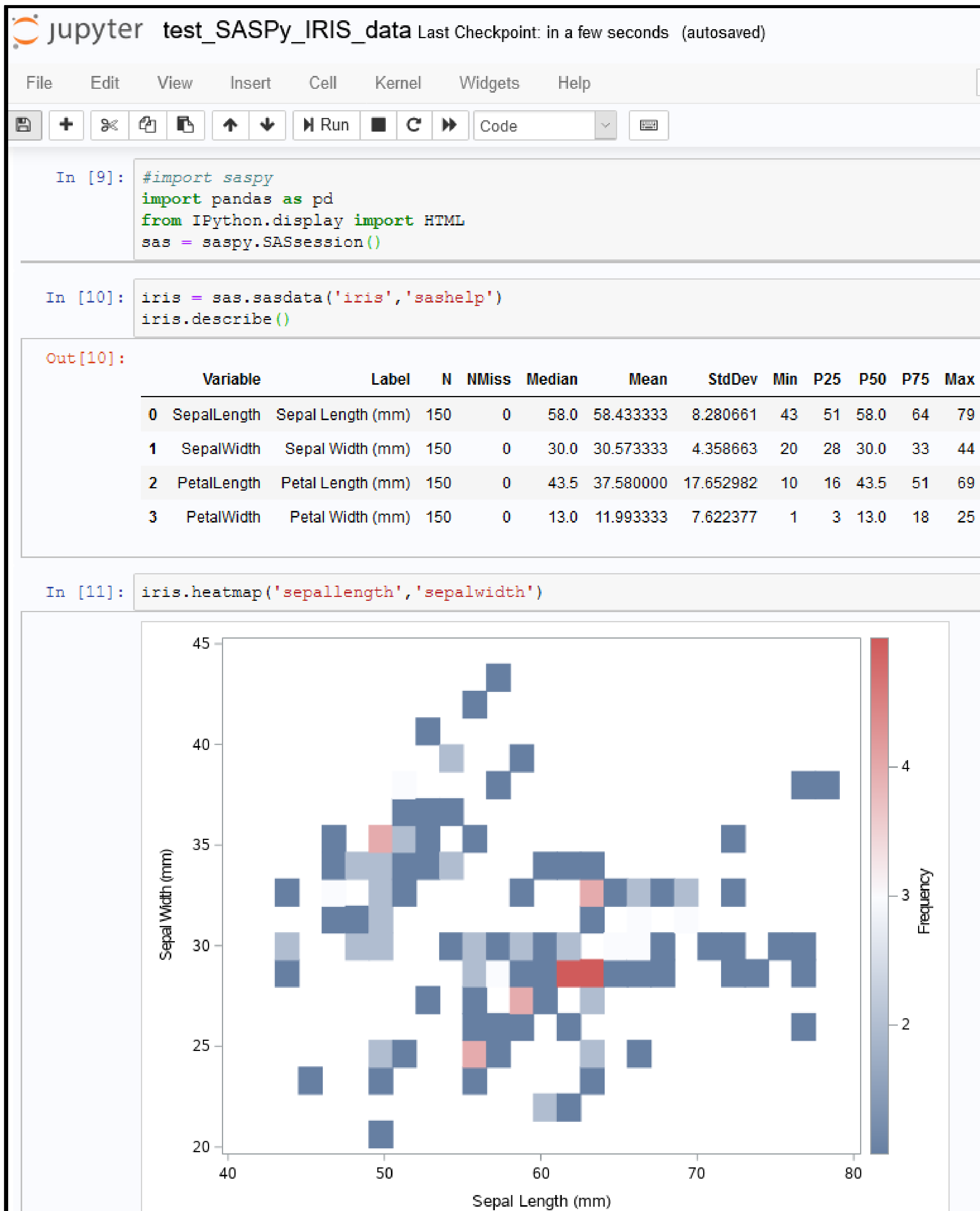


Figure 25: SASPy code example using Python 3.6.

When using the sashelp library, you may encounter permission issues when partitioning data for training and testing. Although you can use custom SAS or Python code to avoid this to some extent, you can use a simple trick to circumvent the issue if you are using SASPy. You can save any dataset from the sashelp library and make a local copy:

```
iris.to_csv('/tmp/iris.csv')
iriscsv = sas.read_csv('/tmp/iris.csv', 'iris_csv')
# now you can partition a local copy or iris with not errors
iris_part = iriscsv.partition(fraction=.7, var='species')
```

Python and SWAT

The SAS SWAT package is an interface to the SAS Cloud Analytics Services (CAS) engine. Recall that when we launched our Azure container instance, we specifically included port 5570, the default port for communication with the CAS server. In Figure 26, we show how to connect to CAS with SWAT and how to import a remote dataset into a SAS session.

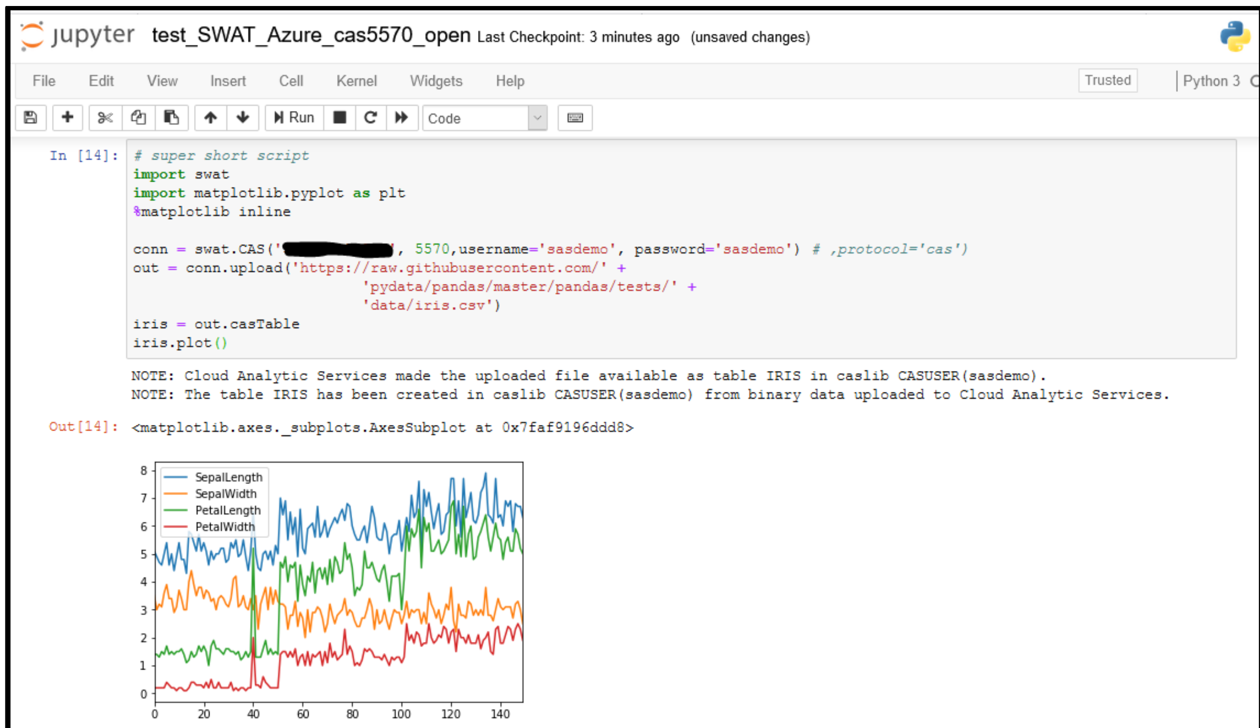


Figure 26: Code example showing how to connect to CAS on port 5570 using the Python SWAT package.

TensorFlow example

In this section we use Tensorflow to build a neural network to classify images from the MNIST dataset (see https://www.tensorflow.org/tutorials for more TensorFlow examples). Since we did not install TensorFlow during the initial image build, we can use pip to install it locally with this command:

```
!pip3 install --user tensorflow==1.5 --no-cache-dir
```

We use pip3 explicitly and we disable the cache to prevent pip from hanging. We also force TensorFlow version 1.5, although you can choose to install the newest version. The install takes quite a bit longer than on a local machine. It may take as long as 20 minutes. Allow the process to finish, and do not exit the install or you might corrupt the libraries. In Figure 27, we show the system paths including the */home/cas/.local* folder, where new libraries are saved. Once the install completes, restart the kernel. You should now be able to import tensorflow.

Figure 27: Python paths, including the */home/cas/.local* path where pip installs new libraries.

In Figure 28, we test our TensorFlow install by training a neural network on the MNIST dataset.

Figure 28: Building a small neural network using TensorFlow in the Docker instance.

Machine Learning with R and Jupyter Notebook

Your new data science environment comes with IRkernel, the R kernel for Jupyter Notebook. In Figure 29, we use the R package ggplot, to visualize the iris dataset. We use the repr package to resize the ggplot output images.

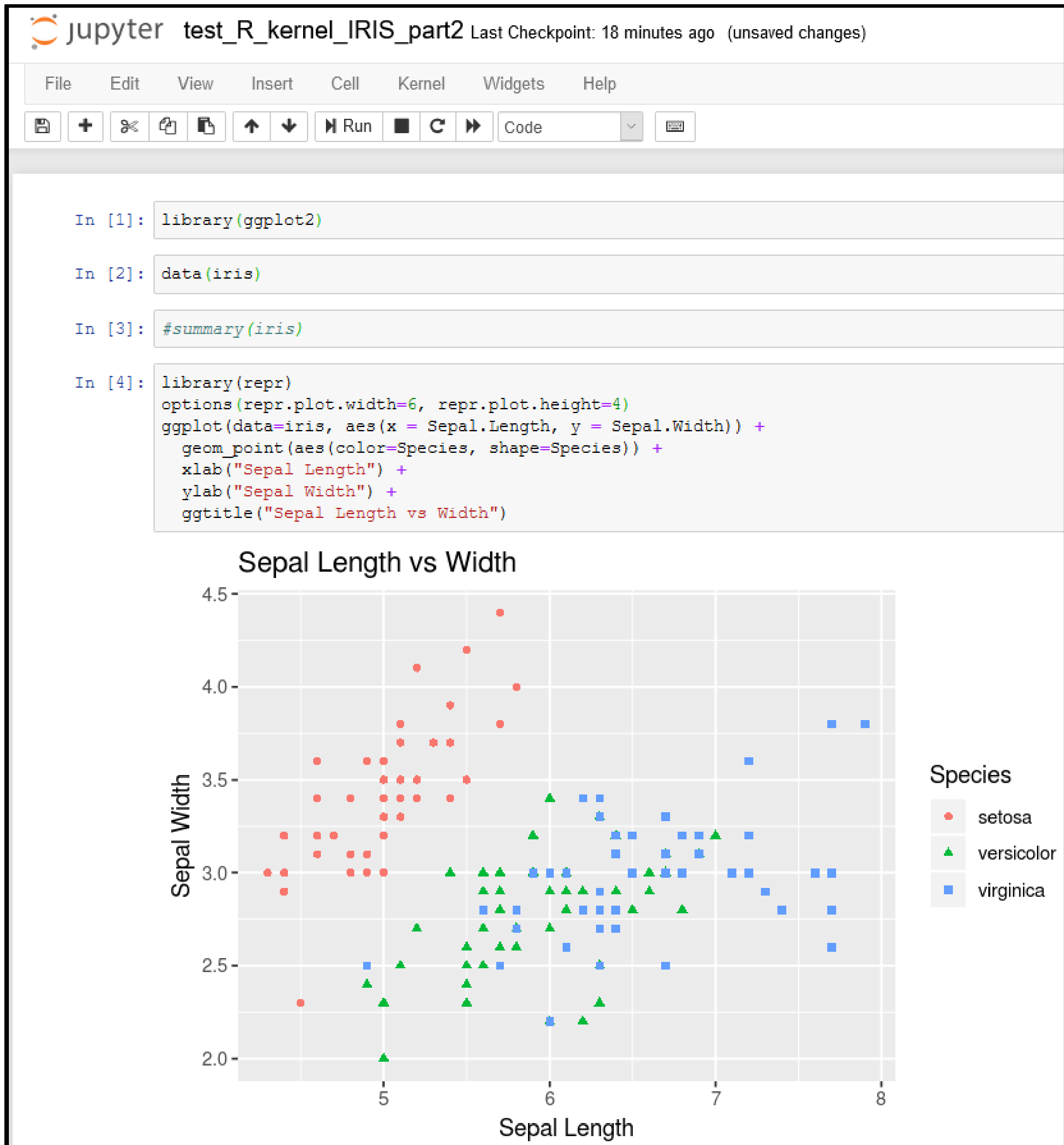


Figure 29: R code example in Jupyter Notebook.

In addition to visualization, we can also train various machine learning models to predict the type of iris species. In Figure 30, we show a minimal code example, which trains five different models and uses cross-validation.

```
#install.packages("kernlab")
#install.packages("randomForest")
library(repr)
options(repr.plot.width=6, repr.plot.height=4)
library(caret)
data(iris)
validation_index <- createDataPartition(iris$Species, p=0.80, list=FALSE)
validation <- iris[-validation_index,]
iris <- iris[validation_index,]
control <- trainControl(method="cv", number=10)
metric <- "Accuracy"
set.seed(1234)
fit.lda <- train(Species~., data=iris, method="lda", metric=metric, trControl=control)
set.seed(1234)
fit.cart <- train(Species~., data=iris, method="rpart", metric=metric, trControl=control)
set.seed(1234)
fit.knn <- train(Species~., data=iris, method="knn", metric=metric, trControl=control)
set.seed(1234)
fit.svm <- train(Species~., data=iris, method="svmRadial", metric=metric, trControl=control)
set.seed(1234)
fit.rf <- train(Species~., data=iris, method="rf", metric=metric, trControl=control)
results <- resamples(list(lda=fit.lda, cart=fit.cart, knn=fit.knn, svm=fit.svm, rf=fit.rf))
#summary(results)
predictions <- predict(fit.lda, validation)
confusionMatrix(predictions, validation$Species)
```

Figure 30: Complete code example in R, illustrating data partitioning, training 5 machine learning models and obtaining predictions using the caret package.

We apply our 5 models to the validation data and find that that Linear Discriminant Analysis (LDA) has the best accuracy (97.5% accuracy), followed by the k-nearest neighbor algorithm. We achieve 100% accuracy on validation data (see the confusion matrix in Figure 31).

```
predictions <- predict(fit.lda, validation)
confusionMatrix(predictions, validation$Species)

Confusion Matrix and Statistics

            Reference
Prediction   setosa versicolor virginica
  setosa         10          0         0
  versicolor      0         10         0
  virginica       0          0        10
```

Figure 31: Predictions for the validation data and the confusion matrix for the different classes. In this case, we achieve perfect classification.

CONCLUSION

In this paper, we've shown how to deploy a single SAS Viya Docker image to the Azure cloud. We described in detail how to set up an Azure account for Docker deployments and how to push large Docker images to the Azure container registry. We also explained how to successfully deploy an Azure container instance using the Azure CLI. Finally, we showed examples of using SAS, Python, and R, in the SAS Viya data science environment. We focused on running pure SAS code in Jupyter Notebook using the SAS kernel, accessing SAS procedures using SASPy and SWAT and using pure Python and R to perform basic machine learning in Jupyter Notebook.

REFERENCES

De Capite, D. 2018. “Docker Toolkit for Data Scientists - How to Start Doing Data Science in. Minutes!” *Proceedings of SAS Global 2018 Conference*, 1875-2018. Denver, CO. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1875-2018.pdf

SAS Container Recipes https://github.com/sassoftware/sas-container-recipes

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Alan Zablocki
RedMane Technology
alan_zablocki@redmane.com
www.alanzablocki.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.