|  |
| --- |
| Databricks |
| Databricks Connect |
| Workshop Walkthrough |

|  |
| --- |
| Bill Kellett  3-10-2021 |

Table of Contents

[Introduction 2](#_Toc65743003)

[Project Overview 2](#_Toc65743004)

[Prerequisites 2](#_Toc65743005)

[Workshop Steps 3](#_Toc65743006)

[Local Dev Machine Setup and Run 4](#_Toc65743007)

[Databricks Job Submission 9](#_Toc65743008)

[Install External Libraries 9](#_Toc65743009)

[Install Workshop Library 9](#_Toc65743010)

[Configure and Run Job 9](#_Toc65743011)

[Code Walkthrough 10](#_Toc65743012)

[Overall Structure 10](#_Toc65743013)

[Initialization Module 10](#_Toc65743014)

[Cleanup Module 10](#_Toc65743015)

[Main Module 10](#_Toc65743016)

[Summary 11](#_Toc65743017)

# Introduction

This document provides a hands-on introduction to building a non-trivial project using Databricks Connect (or dbconnect, as it is usually called). Dbconnect is a solution for Databricks developers who do not wish to use notebooks for development. It enables developers to work in their IDE of choice on their local machines, while still using a Databricks cluster for interactive testing.

This document is a practical walkthrough of a project. For a more complete discussion of dbconnect, its pros and cons, installation, etc. see <https://docs.databricks.com/dev-tools/databricks-connect.html>

Our approach in this workshop is to begin with an existing notebook-based project. We’ll then show how to convert the notebook-based project to a Python code-based project using dbconnect. This approach lets us compare and contrast the two styles in great detail. The notebook-based project is available at <https://github.com/billkellett/databricks-linear-regression-workshop/blob/main/notebooks/ml-linear-regression-workshop.dbc> . Please download and run this notebook and spend some time understanding it. We’ll then rebuild all the same functionality in pure python code.

## Project Overview

Both the notebook and code versions of the workshop project have the exact same functionality:

* Download a file that contains detailed data on home sales
* Create a Delta Lake table from the downloaded data
* Develop a linear regression model that trains on this data in order to predict house prices
* Track the model development process using MLflow
* Determine the best model and store it as an MLflow artifact
* Instantiate the stored model from MLflow
* Use the instantiated model to predict house prices
* Clean up all resources

## Prerequisites

To get the most value from this walkthrough, you should have familiarity with the following topics:

* Databricks in general, and notebook development in particular. The notebook will be the reference point we use throughout in order to highlight the differences in coding techniques.
* Python in general. All development in this workshop will be in Python.
* A Python IDE. We’ll be using PyCharm throughout this document, but you should be fine with any popular IDE. To make things more generic, we’ll do many common functions using terminal commands like git and pip. This will avoid over-reliance on a particular IDE.
* Dbconnect installation. We assume that you have dbconnect installed, as well as any necessary IDE configuration needed for dbconnect. Doc: <https://docs.databricks.com/dev-tools/databricks-connect.html>
* Git installation. We’ll be downloading the project from Github.
* Databricks personal access tokens. You must create a token.
* Databricks cluster creation. For this workshop you must use Databricks Runtime 7.3 LTS ML. A single-node cluster is fine. Note that dbconnect expects this runtime, and its version of Python.
* Databricks Experiments. You must create a new Experiment in the Shared namespace of your workspace and take note of its Experiment ID.

## Workshop Steps

After you complete the prerequisite steps noted above, we’ll start the workshop. Here’s what we’ll be doing:

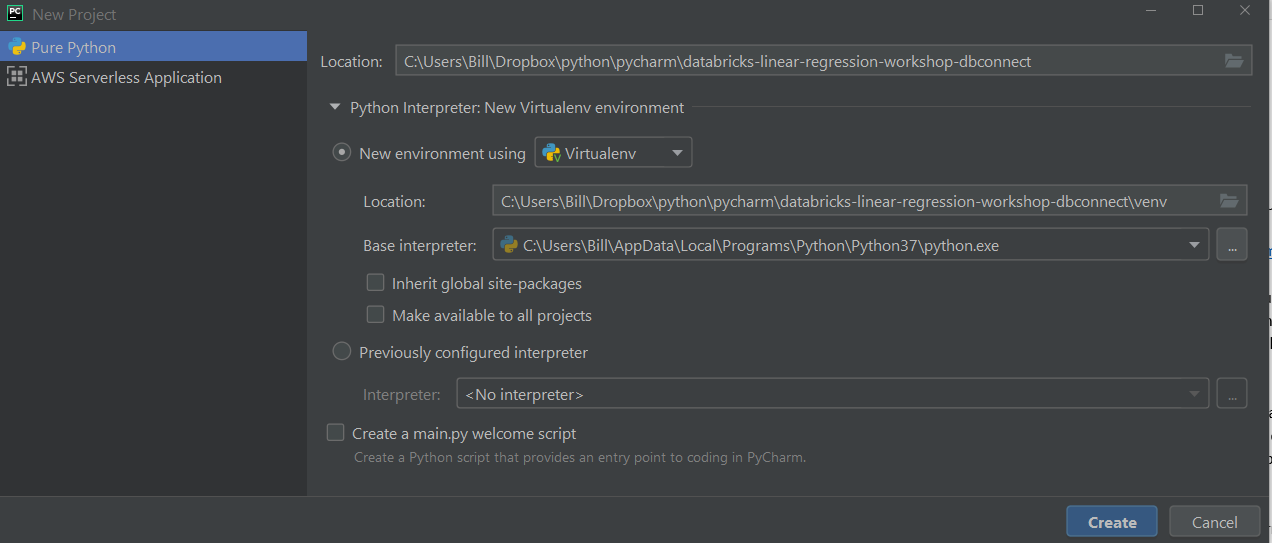
* **Local Dev Machine Setup and Run:** we’ll clone the project from github and get our IDE set up. Then we’ll run against a Databricks cluster using dbconnect. With dbconnect, all Spark processes will run on the cluster, but all “plain old” Python code will run on the local dev machine.
* **Databricks Job Submission:** Next we’ll make sure the code runs when we submit it as a Databricks Jog. In this mode, all our code will run on Databricks.
* **Code walkthrough:** Once we make sure everything is working, we’ll walk through the code in detail. We’ll compare the notebook version of the code to the pure Python version, and explain the purpose of any differences.

# Local Dev Machine Setup and Run

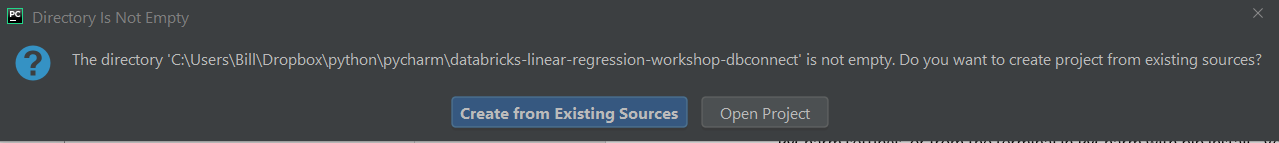
To download the starting resources for the Python project:

* Navigate to one directory level above where you want your project to reside, then:
  + … git clone <https://github.com/billkellett/databricks-linear-regression-workshop-dbconnect>
* Using your IDE, create a new project using the sources you just cloned (below are directions for PyCharm:
  + Click New Project, and select the directory you created above.
  + Specify the virtual environment:
    - Using Virtualenv
    - Base interpreter must be Python3.7.x – you may have to install it if you don’t have it

Below is a screen shot showing the work we did above:

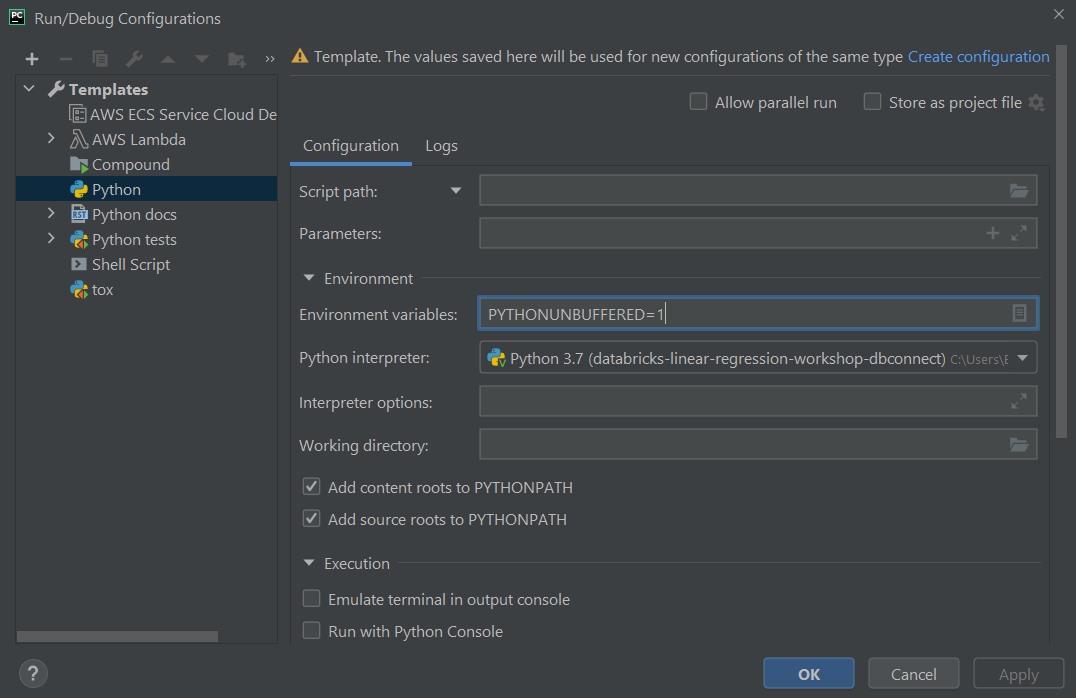


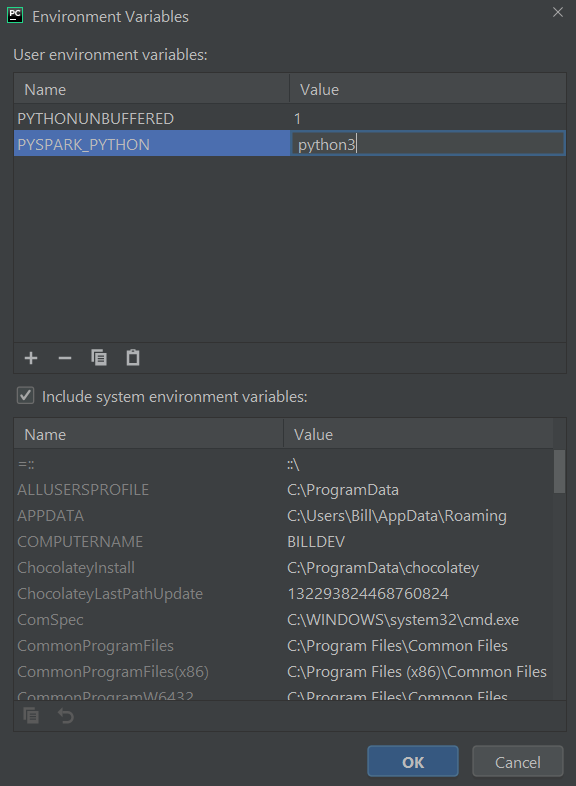
Click Create, then Create from Existing Sources:



Add a new environment variable at Run > Edit Configurations > drop down Templates > Python > drop down Environment > click Edit button on Environment variables and add:

* PYSPARK\_PYTHON=python3





Click OK, Apply, OK.

In PyCharm, open Terminal. You’ll see the prompt begins with (venv), showing that you are in your virtual environment. Enter the following to prepare the IDE to use dbconnect:

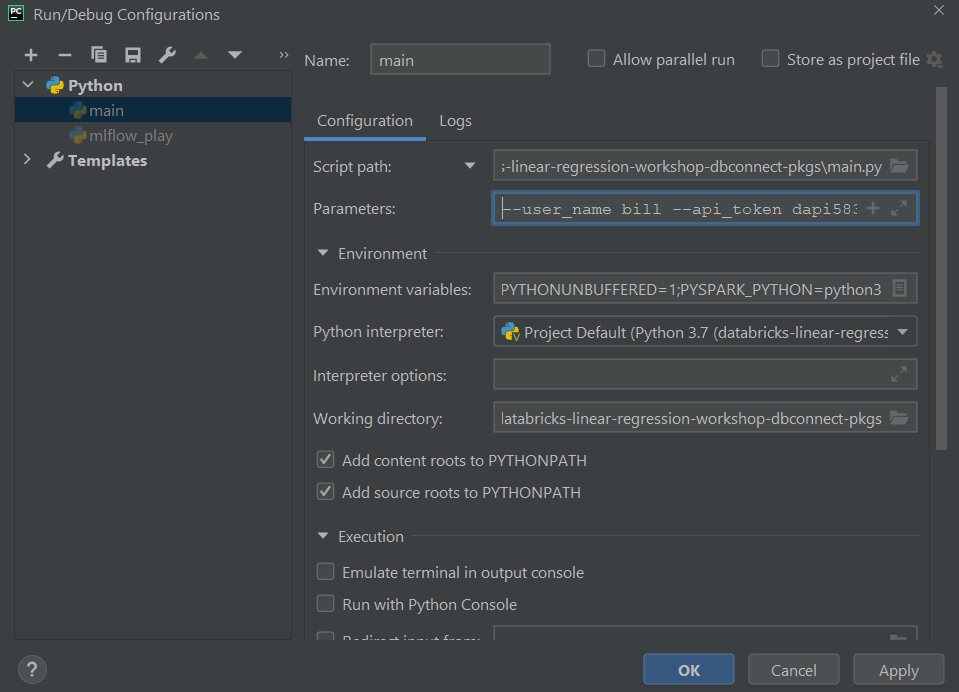
* … pip uninstall pyspark
* … pip install -U databricks-connect==7.3.\*

Now take a look at the requirements.txt file that was downloaded as part of the git clone. It contains all the library dependencies needed to run this project. Let’s recreate it, then use it to make sure we have everything installed. On your PyCharm terminal (venv) enter the following:

* … pip freeze > requirements.txt (don’t worry about any warning messages)
* … pip install -r requirements.txt

Now must set the run-time parameters our code expects. Navigate to Run > Edit Configurations > Python > main > and enter data into the Parameters field:

* --user\_name <any unique string> --api\_token <databricks personal access token> --run\_cleanup\_at\_eoj y --mlflow\_experiment\_id <your mlflow experiment ID>



Click Apply and OK. Note that the –user\_name parameter can be any string you like. It’s just there to ensure that we create a Delta Lake database with a unique name, as well as a unique DBFS file path.

Now you can run main.py. You’ll see some warning messages, but the job should complete successfully with exit code 0.

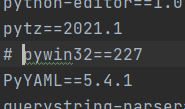
# Databricks Job Submission

Now we’re ready to setup a version of the workshop that will run completely on Databricks as a Job.

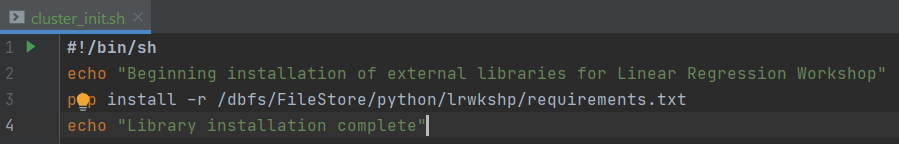
## Install External Libraries

Our first task is to make sure that all external libraries are installed on the cluster every time it starts. We’ll do this with a cluster initialization script.

Begin in Pycharm by opening requirements.txt in your project. Remember that requirements.txt was created (or recreated) when you ran pip freeze earlier. If you are developing on Windows, you will see a line beginning with “pywin32.” This is a windows library, and we must comment it out for Databricks; otherwise, the installation will fail.

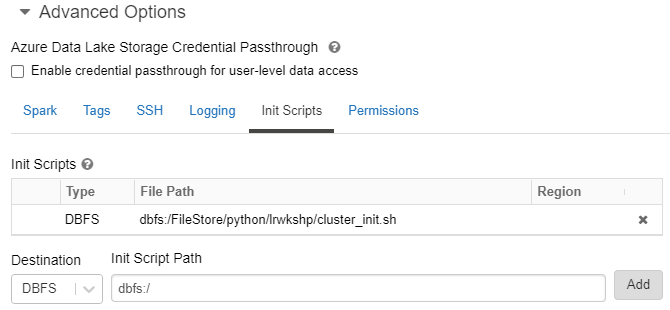


Now open cluster\_init.sh in your project. This is a shell script that we will use to initialize the cluster.



Use the Databricks UI to load both of the above files into DBFS at /FileStore/python/lrwkshp/

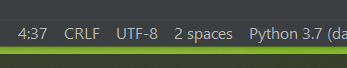
Now you can specify the initialization script on the cluster you will be using:



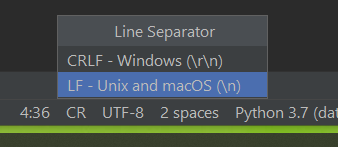
NOTE: If you are developing on a Windows machine, Windows line formatting may cause a problem with your initialization script. The cluster may come up successfully, but if you cannot import your external libraries, check stderr on the cluster initialization log. You may see this error:

ERROR: Could not open requirements file: [Errno 22] Invalid argument: '/dbfs/FileStore/python/lrwkshp/requirements.txt\r'

If this occurs, open the script in PyCharm and check the line format on the bottom status line:



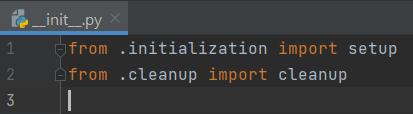
CRLF indicates Windows line formatting. Click on the CRLF.



Change the format to LF, Save, and reinstall your script on the cluster.

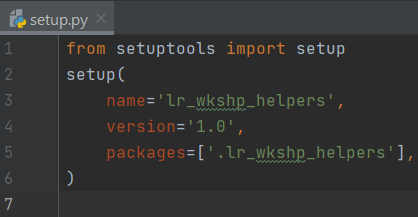
## Install Workshop Library

Our project has two helper modules, cleanup.py and initialization.py. We want to put these into a private library that we upload to the Databricks cluster. Note that these modules are in a directory named lr\_wkshp\_helpers, which also contains this \_\_initi\_\_.py file:



There are many different ways to set up an \_\_init\_\_.py file, but this way will work for our project. For a great discussion on setting up \_\_init\_\_.py, see <https://towardsdatascience.com/whats-init-for-me-d70a312da583>

We also need to create a setup.py file to help build our library and render it as a wheel (.whl) file. Again, there are many options for creating wheels, and a great discussion is here: <https://towardsdatascience.com/how-to-create-a-wheel-file-for-your-python-package-and-import-it-in-another-project-b09f7fbfc466> . The file included with this project will work for us:



Now we’re ready to create the wheel. In PyCharm, open a (venv) terminal and enter:

* … pip install wheel setuptools
* … python setup.py bdist\_wheel

Note that you run python (not python3) above. On Windows, you may see an error at the end of this process:



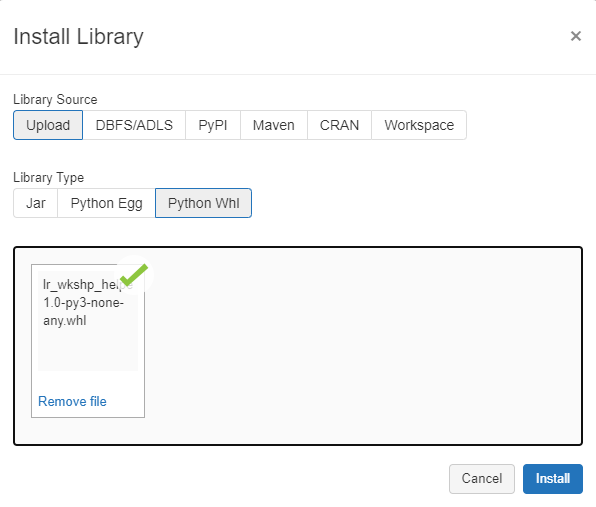
Don’t worry, the process was still successful.

The wheel can now be found in your project in a new folder named dist.



Now we’ll install the wheel at the cluster level in Databricks. Note that the cluster must be running in order to install a library:

* Open the cluster and click the Libraries tab
* Drop in the wheel and click Install

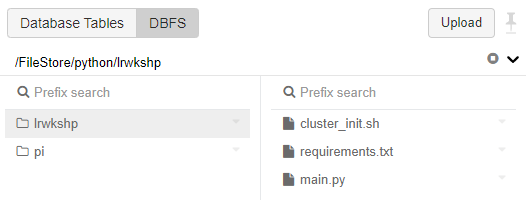


When the installation is complete, you can test it by opening a notebook, connecting to the cluster, and entering:

* … import lr\_wkshp\_helpers

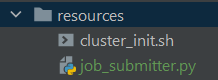
If the import succeeds, the library is installed. Note that you do not specify the entire name of the wheel in the import statement. Just use the name of your package folder (skip the “-1.0-py3-non-any.whl” part).

Now we’ll take our main code that works on dbconnect from our local machine, and deploy it as a Databricks job. First, load main.py to DBFS at /FileStore/python/lrwkshp/



## Configure and Run Job

Currently, the Databricks Jobs UI only lets us submit Python jobs via spark-submit. However, we can use the Databricks REST API to submit a Python job. Open job\_submitter.py, which is found in your project’s resources folder.



In job\_submitter.py, configure the following values:

* Your Databricks domain
* Your Databricks Personal Access Token
* Set parameter values:
  + User name (any unique string)
  + Personal Access Token
  + MLflow Experiment ID

Run job\_submitter.py, and a Job ID is returned. Now you can go to the Databricks Jobs UI and run that job. It should complete successfully.

# Code Walkthrough

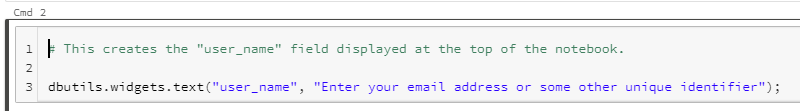
At this point, we have a pure Python version of our linear regression project. We have tested it using dbconnect and an IDE, and we have submitted and run it as a Databricks Job.

Now we’re ready to walk through the code and compare it with the notebook-based version of the project ( <https://github.com/billkellett/databricks-linear-regression-workshop> ). We’ll take a detailed look at what parts of the code had to be changed as we moved from notebook to IDE with dbconnect.

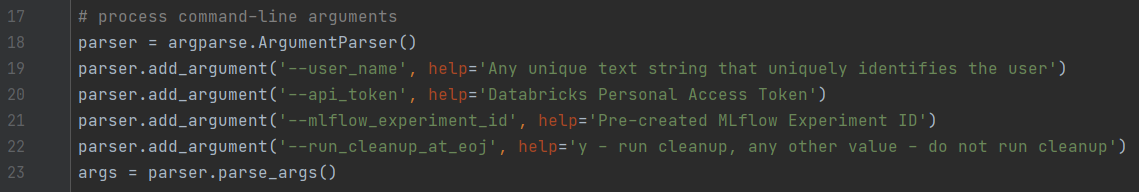
Keep in mind one important design principle that we’ll attempt to use throughout this project. Dbconnect runs spark commands on the cluster, but runs other “plain-old” Python code on the development machine. However, when we submit our completed project as a production, of course, the “plain-old” Python will run on the cluster driver node. This can sometimes create issues, especially when we are doing local file operations. We want to solve this problem without resorting to any sort of “dev-or-prod” conditional processing.

## Overall Structure

Let’s begin by looking at some high-level changes to our structure. First, notebooks can accept run-time parameters as widgets, and when the notebook is submitted as a job, the widgets api can still be used to retrieve run-time parameters defined for the job:

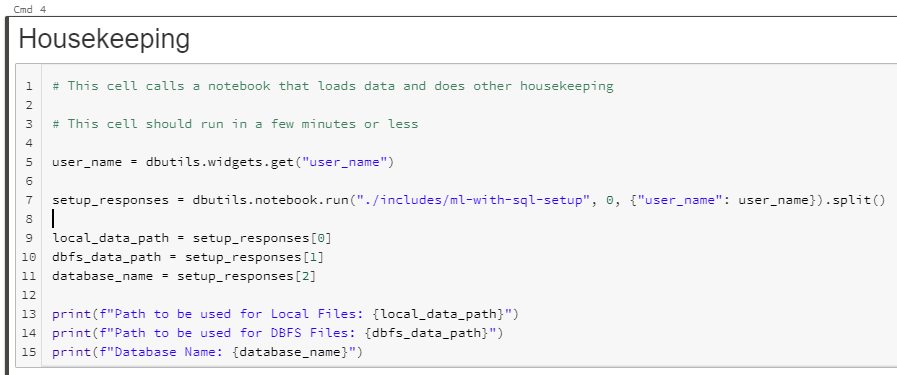


Dbconnect, though, does not support widgets. We’ll need to use traditional run-time parameters instead. In addition, we need to define several more run-time parameters in main.py that are not needed in the notebook version:

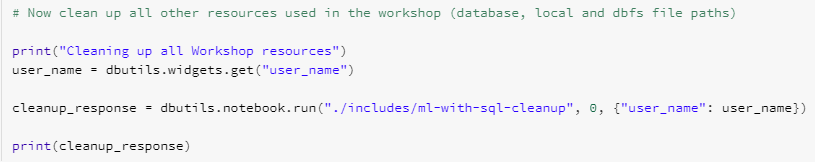


We need to add api\_token because we are executing outside of the Databricks notebook environment. Notebooks always get their own experiment\_id, but we need to define this ourselves in our Python code. The run-time option to perform cleanup (or not) is necessary in the Python code; in the notebook code we can use dbutils.notebook.exit(0) to make sure we don’t accidentally delete databases and files when doing a Run All.

We also need to change the way we call our setup and cleanup modules. In the notebook, we simply call a different notebook for setup…

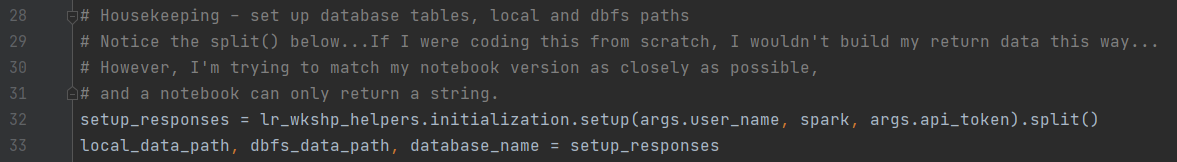


… and for cleanup…



Note above that notebooks can only return a String, so we parse the string to extract our multiple return values.

In our dbconnect version, we copy this String parsing to keep our code as similar as possible. However, we have to call the modules in a more traditional Python manner:



… and…

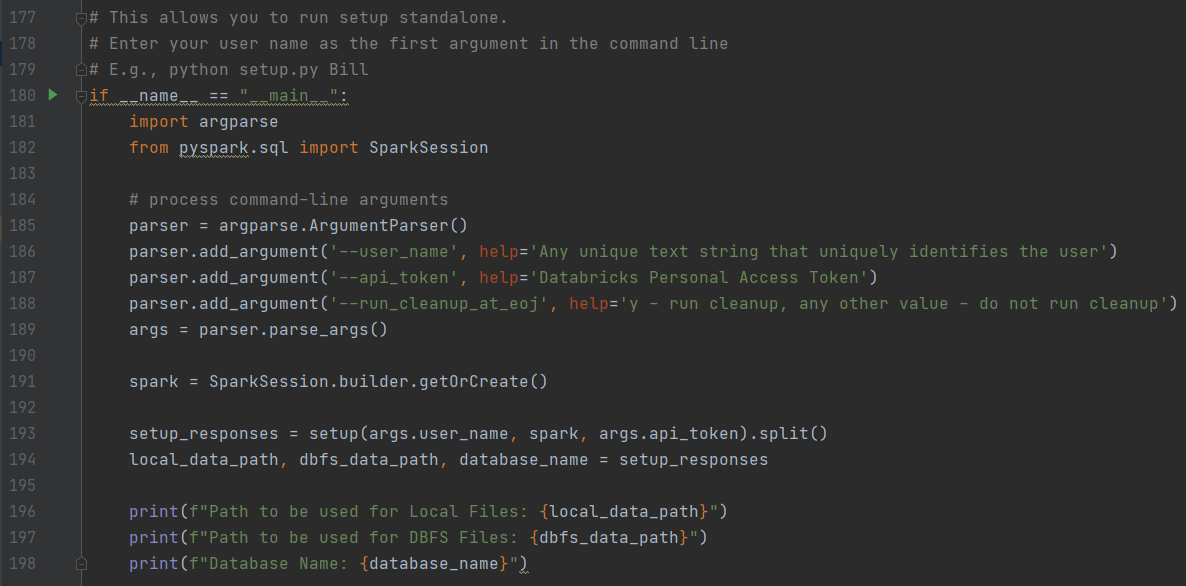
Note the following:

* We changed the notebook name “setup” to “initialization” to avoid a naming conflict with setup.py, which is the name setuptools expects for a configuration file when creating a wheel.
* We had to pass the spark context to the called modules, which is not necessary in notebooks.

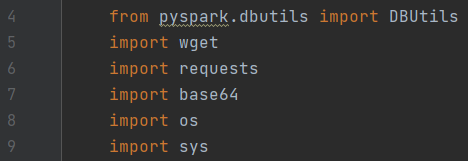
## Initialization Module

Now let’s examine any differences in our startup modules. We have already noted that we must pass the Spark context to our dbconnect version.

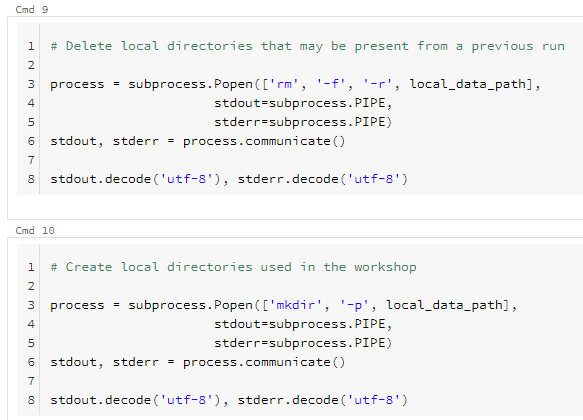
First, we use a standard Python technique that lets us run the module standalone if desired. This is not necessary for a notebook:



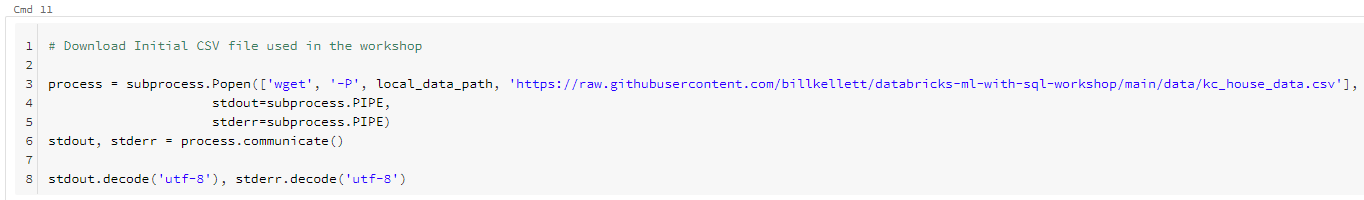
We must do some additional imports:



In the notebook, we create sub-processes to run shell commands to remove and recreate driver-local paths for files:



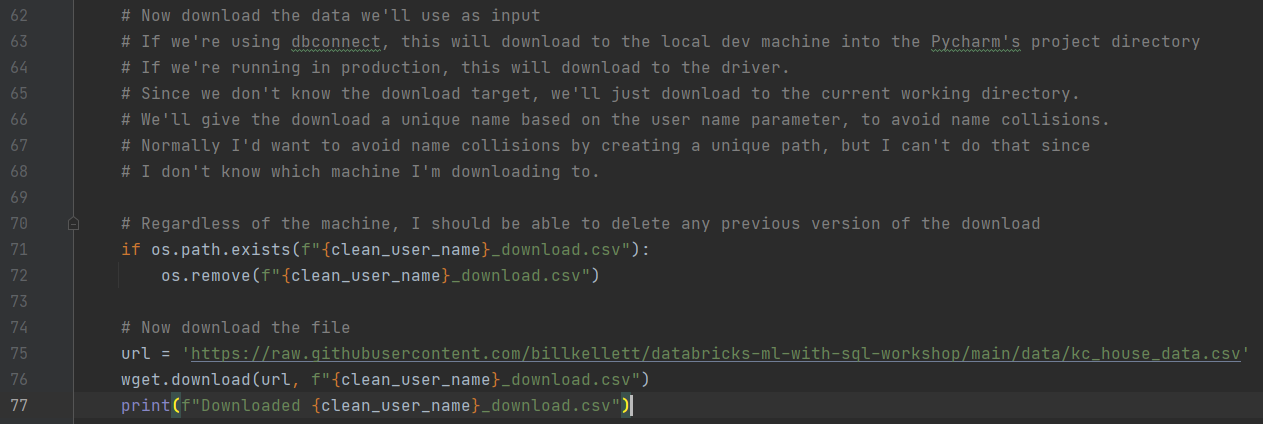
Now we want to download the data we’ll be using as input. This is an interesting problem; the notebook can use wget to download the file to a specific directory on the driver node:



Keep in mind, though, that dbconnect only runs spark commands on the cluster. Any other “plain-old” Python code will run on the local dev machine. However, when we submit the finished project as a Databricks job, that behavior will change, and data will be downloaded to the driver node. That creates a problem for downloading to a specific path, especially if we are doing development on a Windows machine. To solve this in our Python code, we’ll use Python wget and avoid explicitly specifying a path. However, Python wget won’t work on the driver node unless we specify the following sys configuration parameter:

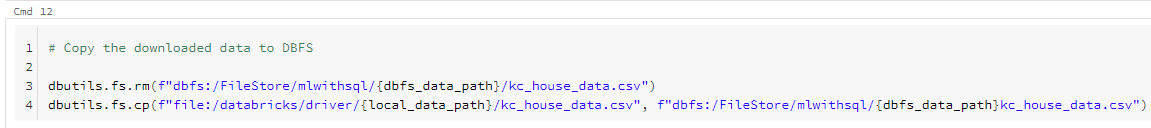


Now we can simply download to the current working directory:



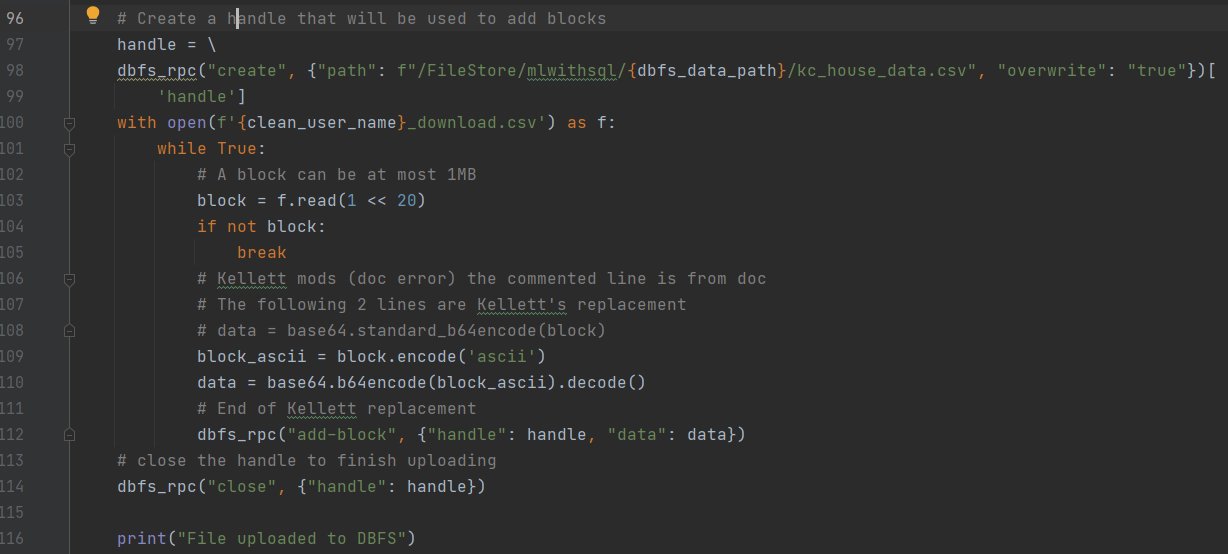
This will work regardless of whether the code executes on the cluster driver or the development machine.

Now we have to solve the problem of moving the downloaded file to DBFS. In the notebook version, we can simply use dbutils:



In the dbconnect version, however, we can’t specify the explicit path of the local file, because we don’t know whether it’s on the local development machine or the cluster driver node. To ensure that the upload will work regardless of the source machine, we’ll use the REST API:





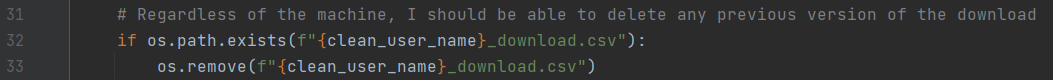
The only other changes we need to make are to use spark.sql() in our Python code whenever the notebook uses the %sql magic command.

## Cleanup Module

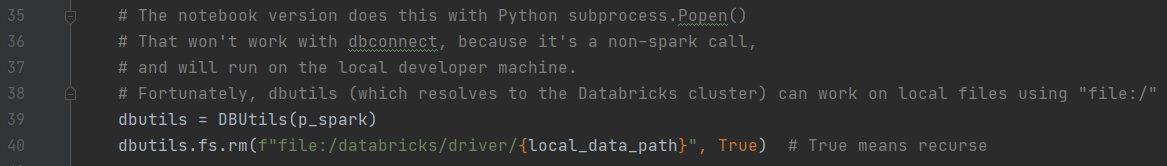
Changes to the cleanup module are very similar to the changes needed in the initialization module. For example, we need to pass in the spark context. In addition, we need to:



… in order to delete the downloaded local file:

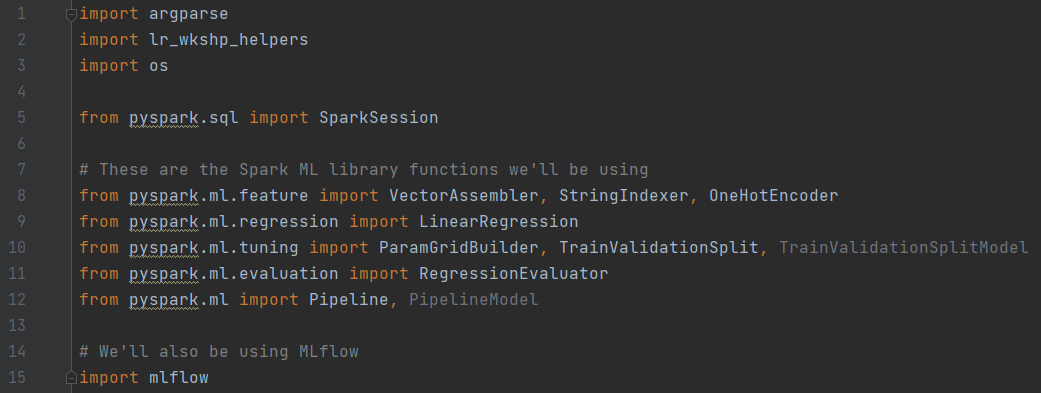


We must also use dbutils instead of Python’s subprocess:



## Main Module

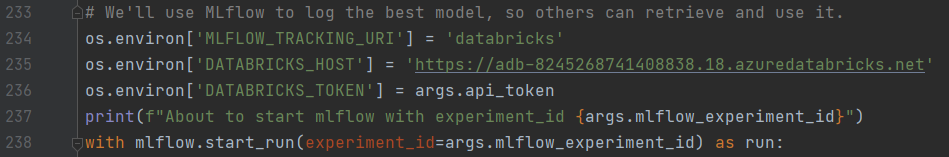
Now let’s look at our main module, main.py. We need quite a few more imports than we do in the notebook version:



We must explicitly get the spark context, which is not necessary in a notebook:



A notebook automatically gets its own MLflow Experiment defined. In our dbconnect Python code, we have to name the Experiment explicitly. We have pre-defined an Experiment in the Shared area of our Databricks Workspace, and we use that Experiment ID:



Also note that the notebook version of the project will automatically log a great deal of information for us. The pure Python version will only log what we explicitly tell it to log, which in our case is the best model itself.

# Summary

We have created a pure Python project that uses dbconnect for development. It is equivalent to a notebook-based project that we used as our model. Some of the most important considerations when using dbconnect include:

* Replace widgets with run-time parameters
* Be wary of local file operations. Remember that dbconnect uses the local dev machine for “plain-old” Python during development. However, when the job is submitted as a Databricks Job, that same code will run on the cluster driver node. This can impact local file operations.
* Use the REST API when dbutils is problematic because of the dev-machine/cluster-driver issue.
* Remember that notebooks automatically have an experiment ID defined for them. When using pure Python, you must define and use an Experiment ID explicitly.