



Automatic Machine Learning with H2O Driverless AI & IBM Db2

Kelly Schlamb

Executive IT Specialist,
IBM Cognitive Systems

 kschlamb@ca.ibm.com
 @KSchlamb



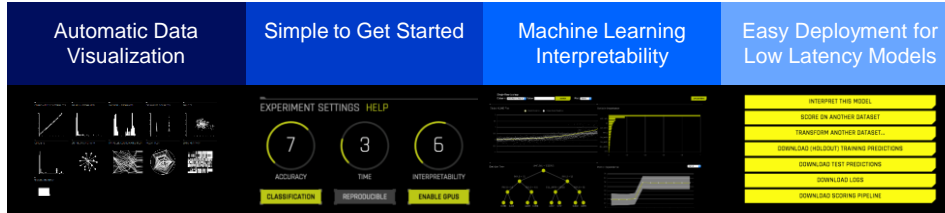
In this deck you'll see how to access data in a Db2 (Db2 for Linux, UNIX and Windows) database for the purposes of training a machine learning model, as well as being able to use that model to score other data stored in the Db2 database.

For those that are familiar with Db2, the term "Driverless" in Driverless AI does not refer to the drivers we're familiar with for Db2 – i.e. client drivers. This is simply intended to mean that you don't need an expert data scientist "at the wheel" to get started working on machine learning problems with H2O Driverless AI. While DAI can certainly be a powerful tool in the hands of an experienced data scientist, DAI lowers the bar of entry that also puts machine learning into the hands of the many.

Thanks for Eric Gudgion at H2O.ai for his assistance.

H2O Driverless AI – Automatic Machine Learning

- You don't need to be an expert data scientist to get started
- Import your dataset, select the column to predict and build your model... it's as simple as that!
- 21 day free trial (<https://www.h2o.ai/try-driverless-ai/>)



H2O Driverless AI is an artificial intelligence (AI) platform for automatic machine learning. It automates some of the most difficult and time-consuming machine learning tasks such as feature engineering, model validation, model tuning, model selection and model deployment. It aims to achieve the highest predictive accuracy, comparable to expert data scientists, but in much shorter time... all thanks to end-to-end automation. Driverless AI also offers automatic visualizations and machine learning interpretability (MLI).

Driverless AI runs on commodity hardware but it was also specifically designed to take advantage of graphical processing units (GPUs), including multi-GPU servers such as the IBM Power Systems AC922 server, the best server for Enterprise AI... period.

You can get started with a 21 day free trial license (<https://www.h2o.ai/try-driverless-ai/>). Provision on your own on-premises server or in the cloud. GPUs will accelerate the model training process but they aren't necessary.

IBM Db2 - The AI database

Analytics performance **powered by cost optimizer, MPP**
OLTP performance with scale up & out high volume low latency transactions with **SMP, pureScale**
Enterprise grade resiliency & scalability **with pureScale, HADR**
Comprehensive security **with RCAC, LBAC, Encryption, SSL**

Powered by AI



Confidence-based query results
leveraging ML-SQL



Up to 10x better query performance
powered by an ML-Optimizer



No data movement & single view on all data
delivered by Data Virtualization



Auto resource optimization
delivered by Adaptive Workload Management

Built for AI



Faster data exploration
by using NLQ in Augmented Data Explorer



Build AI based applications
with Python, GO, JSON and Jupyter notebooks



Model Complex Relationships
by using Db2 Graph and SQL



Blockchain Ready
using Db2 Blockchain Connector

What do we mean by “Db2 – The AI database”? We are making Db2 both “Powered by AI” as well as “Built for AI”.

Let's first look at Powered by AI. We all know Db2 is quite well established for security, resiliency and scalability. We are now taking it further. We are adding ML capabilities to the industry leading cost optimizer. In addition we will be working on ML-based self healing capabilities. What this means is that in the future Db2 will self-tune, self-optimize and self-manage itself!

What do mean by Built for AI? We are adding a ton of capabilities that will make it easier to develop AI-based applications. This will include NLQ-based tools for data discovery, Graph support, ML-based SQL, and support for modern languages and frameworks. Customers will be able to pull data out of block chain and combine with data outside for analytics and deeper insights.

And of course, Db2 has been a reliable, secure, highly available relational database management system (RDBMS) that clients have been depending on for decades to host their most mission critical transactional, operational and warehouse/analytics data. In the world of AI and machine learning where trusted data is king (or queen 😊) it can be the foundation of an enterprise's AI strategy, on top of which AI platforms and tools such as Watson Studio & Watson Machine Learning, IBM Cloud Pak for Data, and H2O Driverless AI can be used to build models from that data.

H2O Driverless AI & IBM Db2



- Access to Db2 is via JDBC – added in Driverless AI 1.7.1 (August 2019)
- Integration points:
 1. Create datasets in DAI based on data in Db2
 - Doesn't have to be all data in a single table – data is extracted via a SQL SELECT statement, which can join multiple tables, transform data, be selective about which data to include, etc.
 2. Use model to score against data in the database
- DAI must have network connectivity to the Db2 database
- Db2 credentials required (examples shown)
 - Server hostname/IP address: 169.62.167.22
 - Port #: 50000
 - Database name: BLUDB
 - User ID: db2inst1
 - Password: MyDb2PassW0rd

The ability to access data in a database such as Db2 using JDBC was added in Driverless AI (DAI) version 1.7.1 (released August 19th, 2019).

You can create new datasets in DAI by extracting data from a Db2 database (or any other database with a JDBC type 4 driver) and once a model has been built using this data, the model can be used to score other data in the database.

The DAI instance must have network connectivity to Db2. How this is accomplished is beyond the scope of this presentation and may require that you work with your network administrators.

When connecting to a Db2 database, you require 5 pieces of information, as shown. This is typical for any kind of connection to Db2 and is not specific to DAI.

H2O Driverless AI & IBM Db2 Installation



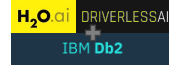
- DAI and Db2 installation steps are **not covered** in this presentation
- **Helpful DAI install links:**
 - Install documentation: <http://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/installing.html>
 - 21 day trial license: <https://www.h2o.ai/try-driverless-ai/>
- **Helpful Db2 install links:**
 - Install documentation: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.5.0/com.ibm.db2.luw.qb.server.doc/doc/c0060076.html
 - Download trial software: <https://www.ibm.com/analytics/db2/trials>
 - Db2 Docker Image: <https://hub.docker.com/r/ibmcom/db2>
 - Free Db2 on Cloud “Lite” instance: <https://cloud.ibm.com/catalog/services/db2>

This presentation assumes that you already have H2O Driverless AI as well as an instance of a Db2 server installed.

If you do not then some pointers to where you can get the software and installation instructions are shown.

If you do not already have a Db2 server and database setup, the easiest approach to getting one is to request a free instance of the Db2 on Cloud “Lite” plan at the link shown. It includes 200 MB of data storage.

Db2 JDBC Driver



- DAI requires use of a **JDBC Type 4 driver** to access a database
- Db2 JDBC type 4 driver file name: **db2jcc4.jar** (also db2jcc.jar)
- Db2 JDBC driver file included in various Db2 installation packages:
 - IBM Data Server Driver for JDBC and SQLJ
 - IBM Data Server Driver Package
 - IBM Data Server Runtime Client
 - IBM Data Server Client
- JDBC and SQLJ driver package by version:
 - <https://www-01.ibm.com/support/docview.wss?uid=swg21363866>
- All IBM data server client and driver packages by version:
 - <https://www.ibm.com/support/pages/download-fix-packs-version-ibm-data-server-client-packages>

To be able to access Db2 or other databases, H2O DAI requires that the database's JDBC Type 4 driver be installed on the H2O DAI server. Db2 has a Type 4 driver called db2jcc4.jar.

The Db2 JDBC driver file is included in various Db2 installation packages. The smallest package – and the one that we will use in this presentation – is the IBM Data Server Driver for JDBC and SQLJ package. The links (for all versions and fix packs of this package) can be found here: <https://www-01.ibm.com/support/docview.wss?uid=swg21363866>.

Links to all of the other IBM data server client and driver packages can be found here:
<https://www.ibm.com/support/pages/download-fix-packs-version-ibm-data-server-client-packages>.

The Db2 documentation includes a page that discusses the different IBM Data Server Client and Driver Types:
https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.swg.im.dbclient.install.doc/doc/c0022612.html.

The Db2 Server install also includes the JDBC driver files but it is unlikely that you will be running H2O DAI on the same server (although you could copy the db2jcc4.jar file from an existing Db2 Server over to the server on which H2O DAI is running).

In addition to the **db2jcc4.jar** file, there is also a **db2jcc.jar** file. Both of them are JDBC Type 4 drivers. Specifically, db2jcc.jar includes functions in the JDBC 3.0 and earlier specifications. db2jcc4.jar includes functions in JDBC 4.0 and later, as well as JDBC 3.0 and earlier specifications. Note that you may also see the JDBC driver described as the JCC driver (“IBM Java Combined Client Driver”).

IBM Data Server Driver and Client Types



Version 11.5 GA

Package	Description
IBM Data Server Driver Package (DS Driver) * Includes JDBC driver	This package contains drivers and libraries for various programming language environments. It provides support for Java (JDBC and SQLJ), C/C++ (ODBC and CLI), .NET drivers and database drivers for open source languages like PHP and Ruby. It also includes an interactive client tool called CLPPlus that is capable of executing SQL statements, scripts and can generate custom reports.
IBM Data Server Driver for JDBC and SQLJ (JCC Driver) * Includes JDBC driver (smallest package to download)	Provides support for JDBC and SQLJ for client applications developed in Java. Supports JDBC 3 and JDBC 4 standard. Also called as JCC driver.
IBM Data Server Driver for ODBC and CLI (CLI Driver)	This is the smallest of all the client packages and provides support for Open Database Connectivity (ODBC) and Call Level Interface (CLI) libraries for the C/C++ client applications.
IBM Data Server Runtime Client * Includes JDBC driver	This package is a superset of Data Server Driver package. It includes many DB2 specific utilities and libraries. It includes DB2 Command Line Processor (CLP) tool.
IBM Data Server Client * Includes JDBC driver	This is the all in one client package and includes all the client tools and libraries available. It includes add-ins for Visual Studio.
IBM Database Add-Ins for Visual Studio	This package contains the add-ins for Visual Studio for .NET tooling support.
IBM .NET Driver NuGet	This package contains the .NET Driver NuGet packages for various platforms.

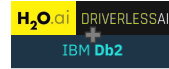
<https://www.ibm.com/support/pages/download-initial-version-115-clients-and-drivers>

This is a description of the different IBM data server driver and client types, some of which contains the JDBC driver. This is taken from here: <https://www.ibm.com/support/pages/download-initial-version-115-clients-and-drivers>.

Additionally, the Db2 documentation describes the different types here:

https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.swg.im.dbclient.install.doc/doc/c0022612.html.

Installing Db2 JDBC Driver on H2O DAI Server



- Download the “*IBM Data Server Driver for JDBC and SQLJ*” package
 - Example of downloaded file: `db2_db2driver_for_jdbc_sqlj_v11.5.tar.gz`
- Extract files from the package:
 - `tar -xzvf db2_db2driver_for_jdbc_sqlj_v11.5.tar.gz`
- Extract driver file from embedded ZIP file:
 - `unzip jdbc_sqlj/db2_db2driver_for_jdbc_sqlj.zip`
- Locate driver file (`db2jcc4.jar`) in the output
- Copy driver file into a location that is accessible by H2O DAI:
 - `mkdir /opt/h2oai/dai/home/Db2`
 - `cp jdbc_sqlj/db2jcc4.jar /opt/h2oai/dai/home/Db2/`
 - `chown -R dai:dai /opt/h2oai/dai/home/Db2/`

The smallest package to download is the IBM Data Server Driver for JDBC and SQLJ package. Follow the links provided on the previous pages. Many different versions of the package exist, corresponding to different version and fixpack levels for Db2. You may want to choose one that matches your Db2 server level, but the drivers should have some level of upward and backward compatibility. In this example we'll use the latest (at the time of writing) Db2 11.5 GA driver.

Note that you will need an IBM ID to be able to login to download the package. You have the option of using Download Directory or HTTP to download. If you want to download it directly to your H2O DAI server (versus downloading to your desktop and then copying it over) then you can choose to “Download using http” and then copy the link from the “Download now” button (e.g. on Windows, right click on the button and then choose to copy the link). Strip the “https://” from the front of the link and use it with the curl command (also specifying an output file name) on your H2O DAI server. For example: “**curl**

```
iwm.dhe.ibm.com/sdfdl/v2/regs2/sm Kane/IDSDIS/Xa.2/Xb.YzLNfpLA2wBbX1X3w35547Km7_x9f4iEpaGQ-  
gVbVEc/Xc.db2_db2driver_for_jdbc_sqlj_v11.5.tar.gz/Xd./Xf.LPr.D1vk/Xg.10385009/Xi.swg-  
idsdjs/XY.regsrvs/XZ.Kyez_SaouRnK2a6fEv8Fb4OPjkk/db2_db2driver_for_jdbc_sqlj_v11.5.tar.gz -o
```

db2_db2driver_for_jdbc_sqlj_v11.5.tar.gz”. You may have to install the curl executable if it's not available on your server.

To make it easier to contain and cleanup all of the extracted files, download the initial file into a temporary location and extract from there (e.g. `/tmp` or `~/TEMP_DIR`). You may have to install the unzip executable if it's not available on your server.

In addition to the **db2jcc4.jar** file, there is also a **db2jcc.jar** file. Both of them are JDBC Type 4 drivers. Specifically, `db2jcc.jar` includes functions in the JDBC 3.0 and earlier specifications. `db2jcc4.jar` includes functions in JDBC 4.0 and later, as well as JDBC 3.0 and earlier specifications. Note that you may also see the JDBC driver described as the JCC driver (“IBM Java Combined Client Driver”).

Update H2O DAI Configuration



- Update the DAI configuration file (e.g. /etc/dai/config.toml) as below
- Add JDBC as one of the enabled data connectors:

```
enabled_file_systems = "<existingListOfMethods>, jdbc"
```

- Create a Db2 JDBC configuration:

```
jdbc_app_configs = '{"db2": {"url": "jdbc:db2://<hostname>:<port#>/<dbname>",  
                             "jarpath": "/<location>/db2jcc4.jar",  
                             "classpath": "com.ibm.db2.jcc.DB2Driver"}}'
```

- If DAI isn't already being started such that it's pointing to the config.toml file (or a copy of it), then add this to the user's .bashrc file:

```
export DRIVERLESS_AI_CONFIG_FILE="/etc/dai/config.toml"
```

- Stop and restart the DAI service

The JDBC connector is not enabled by default. Follow the steps here to enable it as a connector and to create a Db2 JDBC configuration.

Depending on whether you have made changes to the config.toml file in the past, these lines might be commented out. You will need to uncomment them.

The value provided for JDBC_APP_CONFIGS is a JSON string. The first string ("db2" in this case) corresponds to the name of the configuration that you'll see later in the H2O DAI graphical interface.

From a Db2 connection perspective, you need to specify the Db2 server's host name, it's port, and the database name. The user ID and password will be provided later when you generate the dataset.

Note that the JDBC_APP_CONFIGS setting is shown over multiple lines here. That is just for example purposes, but it must actually all be specified as part of a single line in the config.toml file (or it won't be accepted, or H2O DAI might not even start).

Example line: jdbc_app_configs = '{"db2": {"url": "jdbc:db2://db2srvr.mycomp.com:50000/BLUDB", "jarpath": "/opt/h2oai/dai/home/Db2/db2jcc4.jar", "classpath": "com.ibm.db2.jcc.DB2Driver"}}'

There are two other related settings that can be configured: JDBC_APP_JVM_ARGS (extra jvm args for jdbc connector) and JDBC_APP_CLASSPATH (alternative classpath for jdbc connector). They have defaults if not defined, so they aren't necessary to use (unless you have a need to configure them).

For those not familiar with **TOML**, it's a configuration file format that is intended to be easy to read and write due to obvious semantics which aim to be "minimal". This is the format used for DAI's configuration.

H2O DAI: Importing Datasets from Db2

1. Click "Add Dataset"

2. Click "JDBC"

3. Select the entry we added to the config file (e.g. "db2")

4. Enter user ID

5. Enter password

6. Provide name for imported dataset

7. Provide SQL query to use for data import

8. Click to start the import process

9. Imported dataset

Name	Path	Size	Rows	Columns	Status
Db2TitanicTrainingData	...ingData.1569634150.6128998.bin	54KB	791	12	(Click for Actions)

These are the steps for importing data from Db2 into H2O DAI. In this example we're using the Titanic Survivor dataset that has been loaded into Db2. Later on in this presentation we'll go through examples of how to use the scoring pipeline for a model built from this dataset.

- 1) You start by going to the Datasets page and then you click on the "+ ADD DATASET (OR DRAG & DROP)" button.
- 2) Next, click on the "JDBC" connector icon. If you do not see "JDBC" listed then you either missed adding it to the config.toml file, the config.toml file is not being used by DAI, or the DAI service was not stopped and restarted.
- 3) Click on the "SELECT JDBC CONNECTION" button and then choose the name of the configuration we just added (e.g. "db2"). If you do not see the configuration listed then you made a mistake adding it. Note that if you copied it from somewhere (like from this presentation) then there might be issues around the apostrophes/single quotes. Rewrite the configuration line.
- 4) By clicking on the "db2" configuration, it will fill in all of the fields on the left side (JDBC JRL, JDBC Driver, etc.). You must now fill in all of the fields on the right. Start by entering the user ID that you will be connecting to the Db2 database with.
- 5) Next, fill in the password associated with that user.
- 6) Provide a name that you want the new dataset to be called.
- 7) Now, specify a SQL SELECT statement that corresponds with the data that you want to extract out of Db2 (note that the data is being pulled into DAI – when experiments are run against it in the future, it runs against the local copy of the data, it does not in anyway query the data in Db2 at that time). The SELECT statement can be as simple as "SELECT * FROM <TABLE>", which will retrieve all of the rows from the given table. However, you can also be selective (via a WHERE clause), join tables, etc. The column names will be returned along with the data, which will be passed along into the new dataset.
- 8) Click the "CLICK TO MAKE QUERY" button to start the import process.
- 9) Once complete, you will be brought to the Datasets page and you will see the new dataset there. At this point you can explore it, visualize it, and run experiments on it.

Using the Imported Dataset

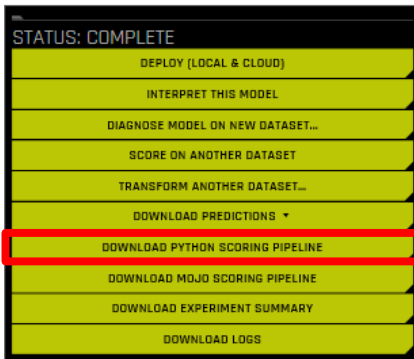
- The imported dataset is no different than any other dataset, which means you can explore the dataset's details, visualize it, and make predictions from it



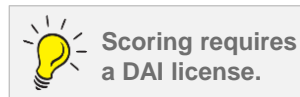
After the dataset has been created, it is just like any other dataset that you've imported from the file system or that you uploaded from your desktop.

The column names come from the names of the columns that are passed back in the result set from the SELECT statement. If you are joining tables in the SELECT statement then make sure that every column in the result set is given a unique name (using the "AS" clause), or the import will fail.

Python Scoring Pipeline



- Download Python scoring pipeline (**scorer.zip**)
 - Package contains an exported model and Python 3.6 source code examples for productionizing models built using H2O Driverless AI.
- Move file to server from where scoring will be performed (it can be the DAI server itself)
- Unzip file (unzips to **scoring_pipeline/**)



<http://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/scoring-standalone-python.html>

After an experiment completes, you have the option of downloading the Python scoring pipeline. This will download a file called `scorer.zip`, which contains the exported model as well as example Python 3.6 source code examples within instructions on how to run them.

Copy the `scorer.zip` file to the server on which you're going to do your scoring. This could be a server where Driverless AI is already installed (in which case it can share the license file) or it could be a separate server. Note that the scoring module requires a valid Driverless AI license for it to work.

Once the file has been copied to the server, unzip it. All of the files will be extracted into a directory called `scoring_pipeline`.

See the Driverless AI documentation for more information on the standalone Python scoring pipeline:
<http://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/scoring-standalone-python.html>.

Python Scoring Pipeline



Notable files in scoring_pipeline:

- **README.txt**
 - Describes the package and how to run the examples
- **example.py**
 - Sample Python script showing how to call the scoring APIs
- **run_example.sh**
 - Runs the example.py script
 - It also sets up a virtualenv (isolated Python environment) with the prerequisite libraries
- **requirements.txt**
 - All the prerequisite libraries for the scoring module to work correctly are listed in here
- **scoring_h2oai_experiment_61704de6_e191_11e9_8883_06cb1baf45d7-1.0.0-py3-none-any.whl**
 - Contains the Python scoring module, bundled into a standalone wheel file

These are some of the notable files found in the scoring_pipeline directory after you unzip scorer.zip.

In the following slides we will use this environment to run a Python script that retrieves data from Db2 and uses the scoring module to score it.

By default, the run_example.sh script creates a virtual environment using virtualenv and pip within which the python code is executed. It can also leverage conda (Anaconda/Miniconda) to create a conda virtual environment and install the required package dependencies.

virtualenv is a tool to create isolated Python environments and manage Python packages for different projects. Using virtualenv allows you to avoid installing Python packages globally which could break system tools or other projects.

Installation of Python 3.6 and other prerequisites are described in README.txt. In my own testing I was using Ubuntu 18.04 and so I used the first command shown below.

Installing Python3.6 on Ubuntu 16.10+:

```
$ sudo apt install python3.6 python3.6-dev python3-pip python3-dev \  
python-virtualenv python3-virtualenv libopenblas-dev
```

Installing Python3.6 on Ubuntu 16.04:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa  
$ sudo apt-get update  
$ sudo apt-get install python3.6 python3.6-dev python3-pip python3-dev \  
python-virtualenv python3-virtualenv libopenblas-dev
```

Python Scoring Pipeline – Scoring Db2 Data



- Update **requirements.txt** file to include **ibm_db** (Db2 Python library):

```
...
datatable-0.9.0.dev119-cp36-cp36m-linux_x86_64.whl
h2o4gpu-0.3.2-cp36-cp36m-linux_x86_64.whl
h2oai-1.7.1-cp36-cp36m-linux_x86_64.whl
scoring_h2oai_experiment_61704de6_e191_11e9_8883_06cb1baf45d7-1.0.0-py3-none-any.whl
ibm_db
```

- If you are using your own Python script (versus using the **example.py** script) then copy it into **scoring_pipeline** and update the **run_example.sh** script to call it instead:

```
run_using_pip(){
...
source env/bin/activate
python Db2DAISingleRowScoring.py
deactivate
}
```

This assumes that you are using the `scoring_pipeline/run_example.sh` script to execute your Python script. You can put your source code directly into `example.py` or change `run_example.sh` to call your Python script which contains the Db2 work and the DAI scoring API calls (sample applications will be shown later on in this presentation).

The `requirements.txt` file contains a list of Python packages/modules that the scoring module and the example source code depend on to run. Doing work with Db2 requires the use of the `ibm_db` module, which is not included in the `requirements.txt` file. Therefore, you must add it to the file as shown.

Note that when you run the `run_example.sh` script for the first time, it's going to use `virtualenv` to create a private Python environment that it will use as part of that first invocation of the script and subsequent ones. If you run `run_example.sh` before adding `ibm_db` to it, the environment will already be built and `ibm_db` will not be installed. So, if you have Python code that attempts an import of this module, it will fail.

To fix this (and any subsequent situation where you want to install additional packages/modules that aren't already installed in the environment) then you can delete the **env** subdirectory within `scoring_pipeline`. This is where all of the files associated with the private Python environment are stored. The result is that `virtualenv` will be used to create a new environment, and any additions to the `requirements.txt` file will be picked up then.

Python Scoring Pipeline – Scoring Db2 Data (cont.)



- Put the **Driverless AI license** into a file and point the environment variable to it:

```
export DRIVERLESS_AI_LICENSE_FILE=~/license.sig
```

- Execute the **run_example.sh** script:

```
bash run_example.sh
```

- The first time that this runs it will create a Python virtual environment (in the **env** subdirectory) and install various packages – subsequent runs won't do this and will be faster.

As noted previously, scoring requires a Driverless AI license. You can put the license into a file and point DRIVERLESS_AI_LICENSE_FILE to it or you can set DRIVERLESS_AI_LICENSE_KEY to the license key value itself.

At this point you can execute the run_example.sh script which will setup the private Python environment (only the first time that the script is run) and then call the example.py script or your script (if you updated run_example.sh to include yours).

Python Pseudocode #1 – Single Row Scoring



```
# Load the required libraries. The scoring_h2oai_experiment* module name will be specific to your model.
from scoring_h2oai_experiment_61704de6_e191_11e9_8883_06cb1baf45d7 import Scorer
import ibm_db

# Create an instance of the Scorer.
scorer = Scorer()

# Connect to Db2 database.
connectionID = ibm_db.connect(<connString>, "", "")

# Execute a SELECT statement that retrieves all of the table columns that will be needed for scoring.
resultSet = ibm_db.exec_immediate(connectionID, <selectStatement>)

# Fetch rows and pack them into a Pandas data frame. Keep reading rows until there are none left.
# For each row read, score it and print the results.
while <still data to be retrieved>
    dataRecord = ibm_db.fetch_tuple(resultSet)
    rowScore = scorer.score(<dataRetrievedFromFetch>)

# Disconnect from the database.
returnCode = ibm_db.close(connectionID)
```

ibm_db API documentation: <https://github.com/ibmdb/python-ibmdb/wiki/APIs>

Here is an example of the flow of a Python script that:

1. Imports the minimal modules (the scoring module and the Db2 module).
2. Creates an instance of the Driverless AI Scorer object.
3. Connects to a Db2 database.
4. Executes a SQL SELECT statement that returns all of the columns/features that the scoring API for your scoring module needs as input. This doesn't return any data but sets up what's called a "cursor" to subsequently scan through the rows in the result set.
5. Iteratively fetches one row at a time from the result set of the SELECT statement.
6. For each row returned, calls the single row scoring API. Note that each invocation of this API is fairly slow. Therefore, it would be more realistic to use the batch scoring API for a group of rows all at once.

The `ibm_db` API documentation can be found here: <https://github.com/ibmdb/python-ibmdb/wiki/APIs>. There you will see descriptions, parameter lists and examples for the Db2 APIs used here.

Python Pseudocode #2 – Batch Scoring



```
# Load the required libraries. The scoring_h2oai_experiment* module name will be specific to you.
import pandas as pd
from scoring_h2oai_experiment_61704de6_e191_11e9_8883_06cb1baf45d7 import Scorer
import ibm_db

# Create an instance of the Scorer.
scorer = Scorer()

# Connect to Db2 database.
connectionID = ibm_db.connect(<connString>, "", "")

# Execute a SELECT statement that retrieves all of the table columns that will be needed for scoring.
resultSet = ibm_db.exec_immediate(connectionID, <selectStatement>)

# Fetch rows and pack them into a Pandas data frame. Keep reading rows until there are none left.
while <still data to be retrieved>
    dataRecord = ibm_db.fetch_tuple(resultSet)
    <Add row into data frame 'df'>

# Score all of the data in the data frame.
results = scorer.score_batch(df)

# Display the scoring results. Alternatively, use Db2 insert/update APIs to store the scoring results back in Db2.
print(results)

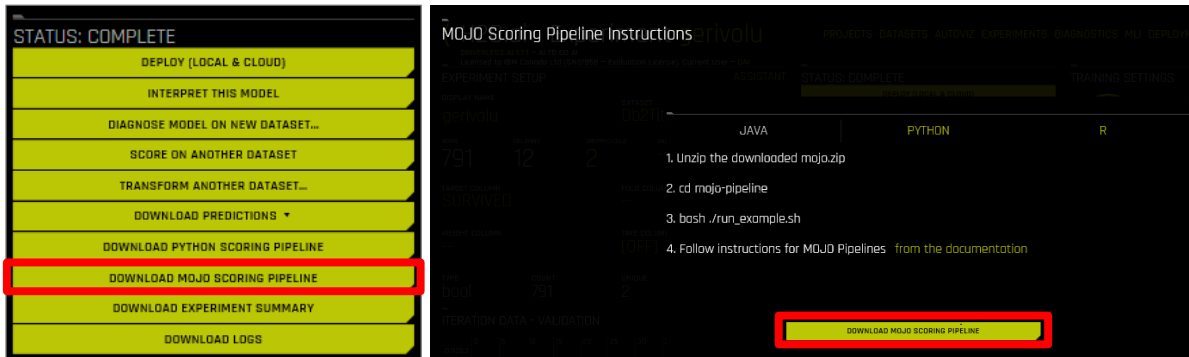
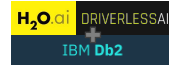
# Disconnect from the database.
returnCode = ibm_db.close(connectionID)
```

Here is an example of the flow of a Python script that:

1. Imports the minimal modules (the scoring module and the Db2 module).
2. Creates an instance of the Driverless AI Scorer object.
3. Connects to a Db2 database.
4. Executes a SQL SELECT statement that returns all of the columns/features that the scoring API for your scoring module needs as input. This doesn't return any data but sets up what's called a "cursor" to subsequently scan through the rows in the result set.
5. Iteratively fetches one row at a time from the result set of the SELECT statement and packs them into a Pandas data frame.
6. Calls the batch scoring API with all of the rows within the data frame. If the number of rows being returned from Db2 has the potential for being very large, it might be more practical to read X number of rows at once (e.g. 100 or 1000) and call the batch scoring API for each group of rows, versus doing a single batch scoring call for all rows in the result set.

The `ibm_db` API documentation can be found here: <https://github.com/ibmdb/python-ibmdb/wiki/APIs>. There you will see descriptions, parameter lists and examples for the Db2 APIs used here.

MOJO (Java) Scoring Pipeline



Scoring requires
a DAI license.

- Write your own Java application (see documentation)
- Or use the H2O.ai-provided DAImojoRunner_DB app

<http://docs.h2o.ai/driverless-ai/latest-stable/docs/userguide/scoring-mojo-pipelines.html>

After an experiment completes, you have the option of downloading the MOJO (Model Objects, Optimized) scoring pipeline. It is a two click process – click “DOWNLOAD MOJO SCORING PIPELINE” from the experiment screen and then click on “DOWNLOAD MOJO SCORING PIPELINE” from the instructions screen. This will download a file called mojo.zip, which contains the following files:

- run_example.sh: A bash script to score a sample test set.
- pipeline.mojo: Standalone scoring pipeline in MOJO format.
- mojo2-runtime.jar: MOJO Java runtime.
- example.csv: Sample test set (synthetic, of the correct format).

Copy this file to the server on which you’ll be running the application to do the scoring.

The instructions on screen show how to run an example program with the scoring module using a sample of data in the CSV file.

As with the Python scoring pipeline, scoring with the MOJO scoring pipeline requires that a Driverless AI license be available to it.

The license can be specified in the following ways:

- * Environment variable:
 - 'DRIVERLESS_AI_LICENSE_FILE' : A location of file with a license
 - 'DRIVERLESS_AI_LICENSE_KEY' : A license key
- * System properties of JVM (-D option):
 - 'ai.h2o.mojos.runtime.license.file' : A location of license file.
 - 'ai.h2o.mojos.runtime.license.key' : A license key.
- * Classpath
 - The license is loaded from resource called '/license.sig'
 - The default resource name can be changed via system property 'ai.h2o.mojos.runtime.license.filename'

MOJO Scoring Pipeline – DAIMojoRunner_DB



- **DAIMojoRunner_DB** is an H2O.ai-provided application that uses your MOJO scoring pipeline (pipeline.mojo) to access databases through JDBC
 - Provided as a pair of jar files
- Includes ability to retrieve data for scoring as well as pushing scoring results back into the database
- For Db2, requires the use of the Db2 **JDBC type 4 driver** (db2jcc4.jar)
- Uses a **properties file** for database and model input
- Very fast!



DAIMojoRunner_DB is not currently available for direct download but can be requested from H2O.ai.

H2O.ai provides a Java application called DAIMojoRunner_DB that can use an exported MOJO scoring pipeline (pipeline.mojo) to make predictions against data in a JDBC-accessible database as well as insert those predictions back into the database. At this point in time (Oct. 2019) this application is not currently available for direct download, but it is available from H2O.ai upon request.

DAIMojoRunner_DB is provided as a zip file (e.g. DAIMojoRunner_DB_2.9.zip). Create a directory and unzip it within this directory. The following files are included:

- DAIMojoRunner_DB.jar
- lib/mojo2-runtime-1.7.0.jar

Like the requirement on a JDBC type 4 driver for importing Db2 data into a Driverless AI dataset, the driver is also required for the DAIMojoRunner_DB application to run and access a Db2 database. Make sure that the db2jcc4.jar file is in a location that is accessible by the user running the application.

You can have multiple scorers running, using slightly different SELECT statements, so you can get many servers running against the database at the same time if you wish. Each server would be running the same pipeline.mojo scoring pipeline, but each properties file would have a slightly different SQL statement (e.g. one server could run (where ID >= 0 and ID < 1000) while another runs (where ID >= 1000 and ID < 2000)).

MOJO Scoring Pipeline – Properties File



- Configurable properties (not an exhaustive list; defaults described in speaker notes):
 - **ModelName:** Full location and file name of pipeline.mojo scoring module
 - **SQLConnectionString:** JDBC database connection string (can include user ID and password)
 - **SQLUser:** User ID to use when connecting to database
 - **SQLPassword:** Password (base64 encoded – see speaker notes)
 - **SQLPrompt:** Prompt for password instead of including in properties file
 - **SQLKey:** Unique row identifier (e.g. id) to associate with output predictions
 - **SQLSelect:** Select statement to retrieve key (id) field and all columns required for scoring
 - **SQLPrediction:** Include all predictions, or specify index (e.g. 0 or 1) of prediction to include in output
 - **SQLWrite:**
 - **csv:** Output key/id column plus prediction column(s)
 - **insert into <table>() values():** Inserts key/id value plus prediction value(s) into database table
 - **update <table> set where <key/id>=:** Updates table, inserting prediction value(s) into it

The DAIMOjoRunner_DB app uses a properties/configuration file to tell it what to do.

Some of the major properties are shown here. You will see examples of these properties in use throughout the example scenarios that follow.

A few extra points of clarification and usage:

- Instead of specifying the user ID and password as separate properties, you can include them in the SQLConnectionString instead (e.g. "jdbc:db2://<hostname>:<port>/<dbname>;user=<userID>;password=<password>;")
- When specifying a password with SQLPassword, do not specify your password as-is. It is expected to be base64-encoded. Use the output of the following Linux command with SQLPassword: `echo <YourPassword> | base64 -i`
- If you don't want to hardcode the encoded password then you can specify SQLPrompt=true. You must still include SQLUser=<userID> to specify the user who will be connecting.
- For SQLPrediction, if you leave it blank then it will write out all target variables (output features of the model). However, you can specify a number that represents the index number of the target variable you want to write out. For example, SQLPrediction=1 means that it will write out SURVIVED.1 in the case of a Titanic survival model.
- When creating your SQLSelect statement you can get an example of what it is expecting to see in the output of the INSPECT of the model.
- SQLKey is optional for both CSV and INSERT operations. If you specify a column for it then it will be included in the CSV output and in the insert statements that get generated. It is required for UPDATE where it is used to find the row that needs updating with the prediction values, for each row that is read and scored.
- SQLWrite has a few different variations. You can specify a value of "csv", in which case the app will simply output the key/id column plus the predictions (e.g. SURVIVED.0, SURVIVED.1), which can then be used to load into the database at a later time. You can have the app actually insert this data (key/id column plus the predictions) into a table in the database. And you can also have it do an update – which would typically involve adding the predictions into the base table from which you queried the rows from. Rather than explaining the syntax here, it's best to see the examples coming up.

The default properties file name is DAIMOjoRunner_DB.properties. However, you can override the file name on the command line using the option -Dpropertiesfilename=<filename>.

The defaults for these properties are as follows:

- ModelName: pipeline.mojo
- SQLConnectionString: no default
- SQLUser: no default
- SQLPassword: no default
- SQLPrompt: no default
- SQLKey: no default
- SQLSelect: no default
- SQLPrediction: no default
- SQLSavePrediction: 0
- SQLWrite: no default
- SQLFieldSeparator: ", "

MOJO Scoring Pipeline – Runtime Properties



- You can also set other property values on the java command line using the “**-Dproperty=value**” syntax
- Configurable runtime properties (not an exhaustive list; defaults described in speaker notes):
 - **inspect**: If set to true then it will display information about the model.
 - **threads**: Number of worker threads
 - **verbose**: If set to true then includes extra debug lines showing what the threads are doing.
 - **logging**: If set to true then also display a line for each row in the input data showing all retrieved values from the row plus the output predictions.
 - **save**: If false then executes inserts/updates against the database; if true it simply displays the insert/update statements
 - **propertiesfilename**: Specifies the name of the properties file that has the SQLxxxxxx properties.
 - **stats**: Displays stats about number of rows read, number of rows scored, number of errors
 - **errors**: Displays information about any scoring errors that happen
 - **wait**: Used for demos; if set to true then the program loads the model but then prompts before scoring

In addition to specifying the ModelName and SQLxxxxxx properties in the properties file as shown on the previous slide, there are some other configurable runtime parameters that can be adjusted to dictate what the program does.

For example, if you want to specify the verbose option then you would just -dverbose=true on the java command.

You will see examples of these properties in use throughout the example scenarios that follow.

The defaults for these properties are as follows:

- inspect: false
- capacity: defaults to number of available processors reported in the environment plus 75%
- threads: defaults to number of available processors reported in the environment
- verbose: false
- logging: false
- save: false
- propertiesfilename: “DAIMojoRunner_DB.properties” in the current working directory
- stats: false
- errors: false
- wait: false

MOJO Scoring Pipeline – Example Scenario



- The upcoming examples are based on the Titanic dataset
- Given information about a passenger (age, sex, fare, number of siblings on the ship, etc.) can you predict whether they were likely to survive the disaster? (model created earlier)

```
CREATE TABLE TITANIC_TRAIN (PASSENGER_ID INT, SURVIVED INT,  
                             PCLASS INT, NAME CHAR(100), SEX CHAR(10), AGE INT,  
                             SIBSP INT, PARCH INT, TICKET CHAR(50), FARE FLOAT,  
                             CABIN CHAR(50), EMBARKED CHAR(50));  
  
CREATE TABLE TITANIC_TEST (PASSENGER_ID INT,  
                             PCLASS INT, NAME CHAR(100), SEX CHAR(10), AGE INT,  
                             SIBSP INT, PARCH INT, TICKET CHAR(50), FARE FLOAT,  
                             CABIN CHAR(50), EMBARKED CHAR(50),  
                             "SURVIVED.0" FLOAT, "SURVIVED.1" FLOAT);  
  
CREATE TABLE TITANIC_RESULTS (PASSENGER_ID INT,  
                                "SURVIVED.0" FLOAT, "SURVIVED.1" FLOAT);
```

The Titanic dataset can be found here: <https://www.kaggle.com/c/titanic/data>. Note that this requires some massaging to get it into a Db2 database (for example, some of the names contain an apostrophe/single quote, which means that it will confuse the Db2 import/load utility if not within double quotes).

I created the tables as shown on the slide. Here are some details on what each table is for:

- TITANIC_TRAIN: Contains the data that was imported into H2O Driverless AI which was then used to build a model (as shown earlier in this presentation)
- TITANIC_TEST: Contains data that will be queried and scored by DAIMOjoRunner_DB, using the model that was built earlier. It also includes two columns (SURVIVED.0 and SURVIVED.1) into which we'll store the predictions for each row. SURVIVED.0 is the confidence value (score) of the passenger not surviving and SURVIVED.1 is the confidence value of the passenger surviving. They should both add up to 1.
- TITANIC_RESULTS: As an alternative to storing the data in-place in the TITANIC_TEST table, you can insert it into another table. That is how this table will be used.

The color-coding in the CREATE TABLE statements is intended to highlight a few important things:

- The table that was used to train the model includes a column called SURVIVED (in blue above), which indicates whether the passenger actually survived or not. This column does not exist in the test table, as it's the data in this table that will be scored.
- Predictions (scoring results) will be stored back in tables. One upcoming example will update the existing TITANIC_TEST table and in another example we'll store the predictions in table TITANIC_RESULTS. As mentioned above, you can see that each of these tables has two columns called SURVIVED.0 and SURVIVED.1 (in green). What's very important to note about this is that each of these columns needs to be in double quotes. As you'll see in the upcoming INSPECT example, the output features (columns) of the model are SURVIVED.0 and SURVIVED.1. However, by default Db2 doesn't like periods (.) in its column names... it confuses the SQL parser. However, you are allowed to have periods in the column names, you just have to surround the name with double quotes. Also note that Db2 is generally case-insensitive (SQL gets converted to uppercase), but when you use a column name surrounded in double quotes in an INSERT or UPDATE SQL statement (which the DAIMOjoRunner_DB app does) and the table was created with the column name in double quotes, then the same case must be used to reference the column in all of the SQL statements against it. Since DAIMOjoRunner_DB will specify "SURVIVED.0" (with those double quotes) in its INSERT and UPDATE statements, the CREATE TABLE statement must use the same case. It's not an issue here since all of the SQL shown on this slide is in uppercase, but some people like to write their SQL in lowercase, which is fine – except that "SURVIVED.0" and "SURVIVED.1" must always be written in uppercase when you create and access the table.

DAIMojoRunner_DB:

Inspect Model (Get Model Details)



- Specify **-Dinspect=true** to display details of the input model (the location of which is specified in the properties file)
- Sample properties file:

```
ModelName=/home/kschlamb/mojo-pipeline/pipeline.mojo
```

- Sample java command to run:

```
$ export DRIVERLESS_AI_LICENSE_FILE=/home/kschlamb/license.sig
# java -Dinspect=true -Dpropertiesfilename=properties-inspect \
  -XX:+UseG1GC -XX:+UseStringDeduplication -Xms5g -Xmx5g \
  -cp /tmp/db2jcc4.jar:DAIMojoRunner_DB.jar daimojorunner_db.DAIMojoRunner_DB
```

One of the capabilities of the DAIMojoRunner_DB application is the ability to inspect (display details about) the model stored in the pipeline.mojo file.

This is done by specifying a command similar to the one shown on the screen. Note the **-Dinspect=true** option, which specifies that you want it to display model information.

The only thing that is needed in the properties file is the **ModelName** line, which specifies the location of the model.

The java command is pointing to the properties file “properties-inspect”. That is the name of the first file where the **ModelName** property is listed. This can be named anything.

DAIMojoRunner_DB:

Inspect Model (Get Model Details)



- Sample output:

```
Details of Model: /home/kschlamb/mojo-pipeline/pipeline.mojo
UUID: 61704de6-e191-11e9-8883-06cblbaf45d7
Input Features
0 = Name: PCLASS Type: Float32
1 = Name: SEX Type: Str
2 = Name: AGE Type: Float32
3 = Name: SIBSP Type: Float32
4 = Name: PARCH Type: Float32
5 = Name: TICKET Type: Str
6 = Name: FARE Type: Float32
7 = Name: CABIN Type: Str
8 = Name: EMBARKED Type: Str
Output Features
0 = Name: SURVIVED.0 Type: Float64
1 = Name: SURVIVED.1 Type: Float64
Suggested configuration for properties file:

select <add-table-index>, PCLASS, SEX, AGE, SIBSP, PARCH, TICKET, FARE, CABIN, EMBARKED from <add-table-name>

update <add-table-name> set where <add-table-index>=

Change the values in <> above and manually test before using them in the program.

The System has 7GB available physically. This program is using 4GB Consider adjusting -Xms and -Xmx to no more than 5GB
The System has 2 Processors.
```

Here is sample output of the INSPECT option.

Note that it shows the input features (columns) and the output features for the model. It also provided skeleton SQL to be used in the properties file when fetching data from the database.

It also suggests the JVM runtime parameters based on the machine you ran the command on. This can be helpful when determining what to set the java parameters to for heap (-Xms and -Xmx) as well as the -Dthreads settings.

DAIMojoRunner_DB:

Exporting Predictions to CSV



- Sample properties file:

```
ModelName=/home/kschlamb/mojo-pipeline/pipeline.mojo
SQLConnectionString=jdbc:db2://127.0.0.1:50000/TESTDB
SQLUser=db2inst1
SQLPassword=XTlQYXNzdzByZAo=
SQLWrite=CSV
SQLKey=PASSENGER_ID
SQLSelect=select passenger_id, pclass, sex, age, sibsp, parch, \
          ticket, fare, cabin, embarked from titanic_test
```

- Sample java command to run:

```
$ export DRIVERLESS_AI_LICENSE_FILE=/home/kschlamb/license.sig
# java -Dpropertiesfilename=properties-csv \
      -Dthreads=2 -XX:+UseG1GC -XX:+UseStringDeduplication -Xms5g -Xmx5g \
      -cp /tmp/db2jcc4.jar:DAIMojoRunner_DB.jar daimojorunner_db.DAIMojoRunner_DB
```

The SQLWrite=CSV option is used to generate a CSV file that can be later used to import/load into a database table. The rows are output to the screen (i.e. stdout) and so you need to capture the output of the app to create the CSV file.

DAIMojoRunner_DB:

Exporting Predictions to CSV



- Sample output:

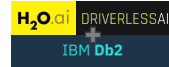
```
PASSENGER_ID,SURVIVED.0,SURVIVED.1
793,0.5799871583779652,0.4200128416220347
794,0.5317727625370026,0.46822723746299744
795,0.6853587826093037,0.3146412173906962
796,0.5869313081105549,0.413068691889445
792,0.44562995433807373,0.5543700456619263
797,0.6082452833652496,0.39175471663475037
798,0.6697674791018169,0.33023252089818317
799,0.6520744959513347,0.34792550404866535
801,0.5717853208382924,0.4282146791617076
802,0.609515090783437,0.3904849092165629
803,0.4034782449404398,0.5965217550595602
800,0.6379381318887074,0.36206186811129254
804,0.49901189406712854,0.5009881059328715
805,0.6863619337479274,0.31363806625207263
806,0.5966482957204182,0.4033517042795817
808,0.615371863047282,0.3846281369527181
807,0.5974470873673756,0.4025529126326243
810,0.4312458038330078,0.5687541961669922
...
```

Here is sample output of the CSV option.

Included is the PASSENGER_ID value (as it was specified in the SQLKey property) and the two output features (column values): SURVIVED.0 and SURVIVED.1.

DAIMojoRunner_DB:

Updating Predictions in Source Table



- Sample properties file:

```
ModelName=/home/kschlamb/mojo-pipeline/pipeline.mojo
SQLConnectionString=jdbc:db2://127.0.0.1:50000/TESTDB
SQLUser=db2inst1
SQLPassword=XTlQYXNzdzByZAo=
SQLKey=PASSENGER_ID
SQLSelect=select passenger_id, pclass, sex, age, sibsp, parch, ticket, fare, cabin,
embarked from titanic_test
SQLWrite=update titanic_test set where PASSENGER_ID=
```

- Sample java command to run:

```
$ export DRIVERLESS_AI_LICENSE_FILE=/home/kschlamb/license.sig
# java -Dpropertiesfilename=properties-update -Dstats=true -Derrors=true \
-Dthreads=2 -XX:+UseG1GC -XX:+UseStringDeduplication -Xms5g -Xmx5g \
-cp /tmp/db2jcc4.jar:DAIMojoRunner_DB.jar daimojorunner_db.DAIMojoRunner_DB
```

The SQLWrite=update option is used to insert the predictions (SURVIVED.0 and SURVIVED.1) into the same table from which the rows were read (TITANIC_TEST in this example). As it is updating existing data rows in the table (by inserting predictions into existing rows), it is actually an update operation that is performed.

You must specify the update statement in the right format: update <table> set where <id_column>=. At runtime, DAIMojoRunner_DB will automatically fill in the column names and their values between “set” and “where” and will automatically append the key/id value after the statement. For example, here is an update statement that it generates for this titanic example:

- update titanic_test set "SURVIVED.0"='0.44562995433807373',"SURVIVED.1"='0.5543700456619263' where PASSENGER_ID='792'

Note that the value of SQLKey (PASSENGER_ID in this example) must match the case of the same column name in the SQLWrite's update statement. This is how the tool matches things up. If they are different then this will fail.

The key is needed because you want to be able to do a lookup of the row that needs updating (e.g. where PASSENGER_ID='792'). This would typically correspond to a primary key or some other unique index that is defined on the table.

DAIMojoRunner_DB:

Updating Predictions in Source Table



- Sample output if you choose to display statements versus executing them (-Dsave=true)

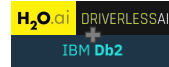
```
update titanic_test set "SURVIVED.0"='0.5799871583779652',"SURVIVED.1"='0.4200128416220347' where PASSENGER_ID='793'
update titanic_test set "SURVIVED.0"='0.5317727625370026',"SURVIVED.1"='0.46822723746299744' where PASSENGER_ID='794'
update titanic_test set "SURVIVED.0"='0.6853587826093037',"SURVIVED.1"='0.3146412173906962' where PASSENGER_ID='795'
update titanic_test set "SURVIVED.0"='0.44562995433807373',"SURVIVED.1"='0.5543700456619263' where PASSENGER_ID='792'
update titanic_test set "SURVIVED.0"='0.5869313081105549',"SURVIVED.1"='0.413068691889445' where PASSENGER_ID='796'
update titanic_test set "SURVIVED.0"='0.6082452833652496',"SURVIVED.1"='0.39175471663475037' where PASSENGER_ID='797'
update titanic_test set "SURVIVED.0"='0.6697674791018169',"SURVIVED.1"='0.33023252089818317' where PASSENGER_ID='798'
update titanic_test set "SURVIVED.0"='0.6520744959513347',"SURVIVED.1"='0.34792550404866535' where PASSENGER_ID='799'
update titanic_test set "SURVIVED.0"='0.6379381318887074',"SURVIVED.1"='0.36206186811129254' where PASSENGER_ID='800'
update titanic_test set "SURVIVED.0"='0.609515090783437',"SURVIVED.1"='0.3904849092165629' where PASSENGER_ID='802'
update titanic_test set "SURVIVED.0"='0.5717853208382924',"SURVIVED.1"='0.4282146791617076' where PASSENGER_ID='801'
update titanic_test set "SURVIVED.0"='0.4034782449404398',"SURVIVED.1"='0.5965217550595602' where PASSENGER_ID='803'
update titanic_test set "SURVIVED.0"='0.49901189406712854',"SURVIVED.1"='0.5009881059328715' where PASSENGER_ID='804'
update titanic_test set "SURVIVED.0"='0.6863619337479274',"SURVIVED.1"='0.31363806625207263' where PASSENGER_ID='805'
update titanic_test set "SURVIVED.0"='0.5966482957204182',"SURVIVED.1"='0.4033517042795817' where PASSENGER_ID='806'
update titanic_test set "SURVIVED.0"='0.615371863047282',"SURVIVED.1"='0.3846281369527181' where PASSENGER_ID='808'
update titanic_test set "SURVIVED.0"='0.5974470873673756',"SURVIVED.1"='0.4025529126326243' where PASSENGER_ID='807'
update titanic_test set "SURVIVED.0"='0.5869313081105549',"SURVIVED.1"='0.413068691889445' where PASSENGER_ID='809'
update titanic_test set "SURVIVED.0"='0.683991089463234',"SURVIVED.1"='0.31600891053676605' where PASSENGER_ID='811'
update titanic_test set "SURVIVED.0"='0.6133703192075093',"SURVIVED.1"='0.38662968079249066' where PASSENGER_ID='812'
...
Total selected rows 100
Thread-0 Rows Read 53 Scored 53 Error 0 Queue Empty false
Thread-3 Rows Read 47 Scored 47 Error 0 Queue Empty true
```

Here is sample output of the UPDATE option.

This is using the -Dsave=true option to display the update statements versus executing them. It's also using the -Dstats=true option to show how many rows were read and scored.

DAIMojoRunner_DB:

Inserting Predictions to Separate Table



- Sample properties file:

```
ModelName=/home/kschlamb/mojo-pipeline/pipeline.mojo
SQLConnectionString=jdbc:db2://127.0.0.1:50000/TESTDB
SQLUser=db2inst1
SQLPassword=XTlQYXNzdzByZAo=
SQLKey=PASSENGER_ID
SQLSelect=select passenger_id, pclass, sex, age, sibsp, parch, ticket, fare, cabin,
embarked from titanic_test
SQLWrite=insert into titanic_results() values()
```

- Sample java command to run:

```
$ export DRIVERLESS_AI_LICENSE_FILE=/home/kschlamb/license.sig
# java -Dpropertiesfilename=properties-insert -Dstats=true -Derrors=true \
-Dthreads=2 -XX:+UseG1GC -XX:+UseStringDeduplication -Xms5g -Xmx5g \
-cp /tmp/db2jcc4.jar:DAIMojoRunner_DB.jar daimojorunner_db.DAIMojoRunner_DB
```

The SQLWrite=insert option is used to insert the predictions (SURVIVED.0 and SURVIVED.1) into a separate table (TITANIC_RESULTS in this example). This table does need to be in the same database as the data table, though, because it's all done under the same connection string, which specifies which database you're connecting to.

You must specify the insert statement in the right format: insert into <table>() values(). At runtime, DAIMojoRunner_DB will automatically fill in the column names within the first set of parenthesis and their values within the second set of parenthesis. For example, here is an insert statement that it generates for this titanic example:

- insert into
titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('792','0.44562995433807373','0.5543700456619263')

The string PASSENGER_ID in the insert statement is coming directly from the SQLKey property. It will use exactly what you specify in SQLKey in the insert statements. If you recall the discussion earlier about how Db2 handles case in SQL statements, you'll know that if a column name is in double quotes (as it is here) then the table must have been created in exactly the same way... which we did. However, what this also means is that the app will **not** work if you don't use uppercase text in SQLKey (e.g. SQLKey=passenger_id will **not** work – each insert will fail with an SQL0206N error, stating that this column does not exist in the table).

DAIMojoRunner_DB:

Inserting Predictions to Separate Table



- Sample output if you choose to display statements versus executing them (-Dsave=true)

```
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('793','0.5799871583779652','0.4200128416220347')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('794','0.5317727625370026','0.46822723746299744')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('795','0.6853587826093037','0.3146412173906962')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('796','0.5869313081105549','0.413068691889445')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('792','0.44562995433807373','0.5543700456619263')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('798','0.6697674791018169','0.33023252089818317')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('797','0.6082452833652496','0.39175471663475037')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('800','0.6379381313887074','0.36206186811129254')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('801','0.5717853208382924','0.4282146791617076')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('802','0.609515090783437','0.3904849092165629')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('799','0.6520744959513347','0.34792550404866535')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('804','0.49901189406712854','0.5009881059328715')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('805','0.6863619337479274','0.31363806625207263')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('806','0.5966482957204182','0.4033517042795817')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('803','0.4034782449404398','0.5965217550595602')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('808','0.615371863047282','0.3846281369527181')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('809','0.5869313081105549','0.413068691889445')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('807','0.5974470873673756','0.4025529126326243')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('811','0.683991089463234','0.31600891053676605')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('812','0.6133703192075093','0.38662968079249066')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('813','0.5820269385973613','0.41797306140263873')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('814','0.5592739482720692','0.440726051727307')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('815','0.5859753489494324','0.41402465105056761')
insert into titanic_results("PASSENGER_ID","SURVIVED.0","SURVIVED.1")values('816','0.6604404648145039','0.33955353518549603')
...
Total selected rows 100
Thread-3 Rows Read 46 Scored 46 Error 0 Queue Empty true
Thread-0 Rows Read 54 Scored 54 Error 0 Queue Empty true
```

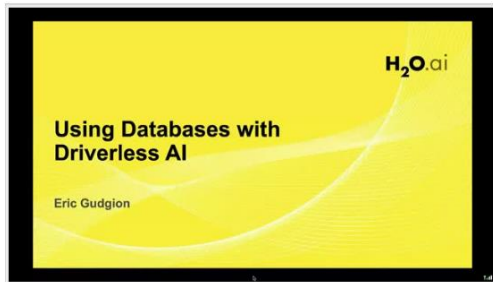
Here is sample output of the INSERT option.

This is using the -Dsave=true option to display the insert statements versus executing them. It's also using the -Dstats=true option to show how many rows were read and scored.

For More Information on DAIMojoRunner_DB

<https://www.h2o.ai/webinars/?commid=367565>

Learn How to Easily Use AI Against Your Production Database



Eric Gudgion, H2O.ai

Sep 10 2019 | 49 mins

Play

H2O Driverless AI is an award-winning automatic machine learning platform. With Driverless AI, everyone including expert and junior data scientists, domain scientists, and data engineers can develop trusted machine learning models.... [more](#)

If you would like to hear more about DAIMojoRunner_DB then watch this webinar from H2O.ai.

IBM