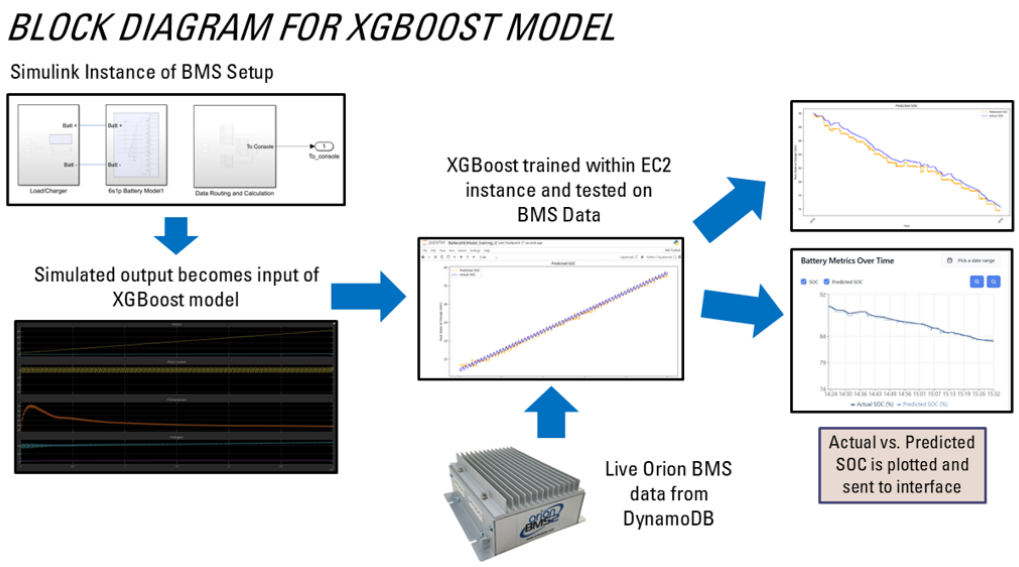
How-to Manual for Our AI Model – Battery Diagnostic and Prognostic Tool Application

by Joshua Natal

***Introduction***

This manual walks you through setting up AWS CLI v2, SSH'ing into our EC2 instance, launching Jupyter to train and test our XGBoost model, and finally generating synthetic battery data in Simulink. Here is a quick overview of our Battery Diagnostic and Prognostic process to help you get started: To train and test an XGBoost model (or swap in an LSTM) in a Jupyter environment hosted on AWS EC2, you must first create realistic battery charge-discharge profiles in Simulink, then extract and preprocess that data into time-series CSVs. Lastly, you must deploy the trained model and continuously gather new data for ongoing predictions. Refer to the figure for an example of the workflow for the model. Follow each section in order and you'll be ready to pick up where we left off for the Battery Diagnostic and Prognostic Tool Application.




***Getting Started – Prerequisites***

Before installing AWS CLI, ensure you have a compatible Python environment set up. We recommend Python 3.8–3.11 in a virtual environment or Conda environment to avoid dependency conflicts. You'll need to install key libraries after activating your environment:

```
pip install xgboost pandas numpy matplotlib scikit-learn jupyterlab
```

Next, make sure you have AWS CLI v2 installed. Link:

https://awscli.amazonaws.com/AWSCLIV2.msi

If you run into trouble, this is the webpage for troubleshooting (in case if you have an older version or AWS CLI, for example:

https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

Install command to put in PowerShell:

```
# Windows (Chocolatey)

choco install awscli
```

Ensure you install AWS CLI v2 first because it gives you command-line access to key pairs, EC2 instances, S3 buckets, and so on. Once you've run aws configure, you can spin up and connect to your instance without opening the AWS web console. That makes the rest of this workflow much easier to work with.

***Getting into the EC2 instance***

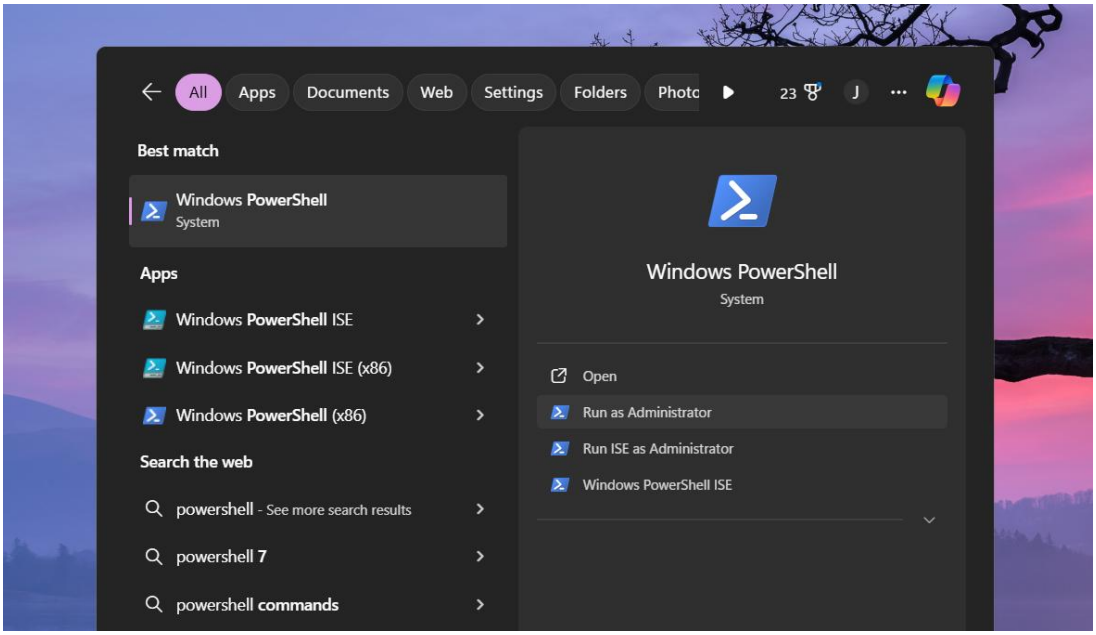
Instead of embedding long-lived Access Keys in ~/.aws/credentials, we advise giving your EC2 instance an IAM role with least-privilege permissions (S3 read/write, EC2 describe, etc.) for security and consistency. If keys are required, think about using a vault or environment variables (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY) to prevent unencrypted storage of secrets. Common SSH problems include incompatible usernames (ec2-user vs. Ubuntu), closed ports (make sure your security group permits inbound TCP 8888 and SSH port 22), or wrong file permissions on your.pem (chmod 400 key.pem).

1. Make sure to go on the Amazon account to get the necessary credentials, such as

- Access Key ID
- Secret Key
- Region

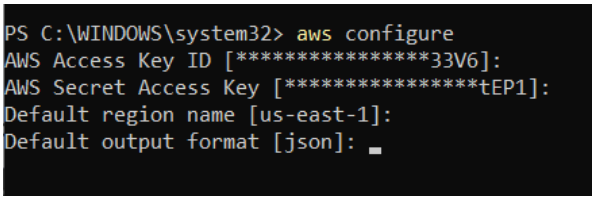
You will also need to download a .pem file which will help you SSH into the EC2 instance.

2. After getting the right credentials from the Amazon account, go into Powershell (run as administrator)

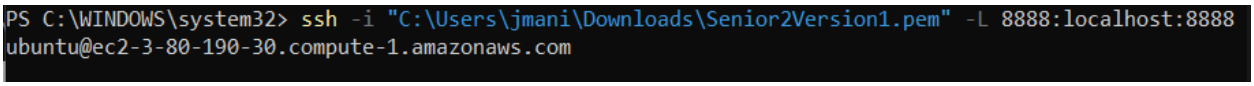



and run the command `aws configure`, like so


3. It will ask for your Access Key ID, Secret Access Key, region name, and output format.




4. Once the credentials are established, you can ssh into the instance using the command




where the file path in blue quotations is the location of your .pem file, and the 8888 localhost followed by the ec2 instance defines your specific compute instance

After that, you will see a lot of text indicating status updates for the instance and whatnot

```
PS C:\WINDOWS\system32> ssh -i "C:\Users\jmani\Downloads\Senior2Version1.pem" -L 8888:localhost:8888
ubuntu@ec2-3-80-190-30.compute-1.amazonaws.com
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-1024-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Thu Apr 24 21:20:42 UTC 2025

  System load:  0.0               Processes:             101
  Usage of /:   22.2% of 28.89GB  Users logged in:       0
  Memory usage: 29%               IPv4 address for eth0: 172.31.22.58
  Swap usage:   0%

 * Ubuntu Pro delivers the most comprehensive open source security and
   compliance features.

   https://ubuntu.com/aws/pro

Expanded Security Maintenance for Applications is not enabled.

17 updates can be applied immediately.
1 of these updates is a standard security update.
To see these additional updates run: apt list --upgradable

7 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm
```

```
*** System restart required ***
Last login: Wed Apr 16 18:58:06 2025 from 131.94.186.11
ubuntu@ip-172-31-22-58:~$
```

5. On the command line, type

- `cd certs,` followed by
- `cd ~/.jupyter/,` followed by
- `jupyter notebook`

```
*** System restart required ***
Last login: Wed Apr 16 18:58:06 2025 from 131.94.186.11
ubuntu@ip-172-31-22-58:~$ cd certs
ubuntu@ip-172-31-22-58:~/certs$ cd ~/.jupyter/
ubuntu@ip-172-31-22-58:~/.jupyter$ jupyter notebook
[I 2025-04-24 21:22:12.000 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2025-04-24 21:22:12.004 ServerApp] jupyter_server_terminals | extension was successfully linked.
[I 2025-04-24 21:22:12.008 ServerApp] jupyterlab | extension was successfully linked.
[I 2025-04-24 21:22:12.013 ServerApp] notebook | extension was successfully linked.
[I 2025-04-24 21:22:12.674 ServerApp] notebook_shim | extension was successfully linked.
[I 2025-04-24 21:22:12.724 ServerApp] notebook_shim | extension was successfully loaded.
[I 2025-04-24 21:22:12.726 ServerApp] jupyter_lsp | extension was successfully loaded.
[I 2025-04-24 21:22:12.728 ServerApp] jupyter_server_terminals | extension was successfully loaded.
[I 2025-04-24 21:22:12.741 LabApp] JupyterLab extension loaded from /home/ubuntu/.local/lib/python3.1
0/site-packages/jupyterlab
[I 2025-04-24 21:22:12.741 LabApp] JupyterLab application directory is /home/ubuntu/.local/share/jupy
ter/lab
[I 2025-04-24 21:22:12.742 LabApp] Extension Manager is 'pypi'.
[I 2025-04-24 21:22:12.861 ServerApp] jupyterlab | extension was successfully loaded.
[I 2025-04-24 21:22:12.865 ServerApp] notebook | extension was successfully loaded.
[I 2025-04-24 21:22:12.866 ServerApp] Serving notebooks from local directory: /home/ubuntu/.jupyter
```

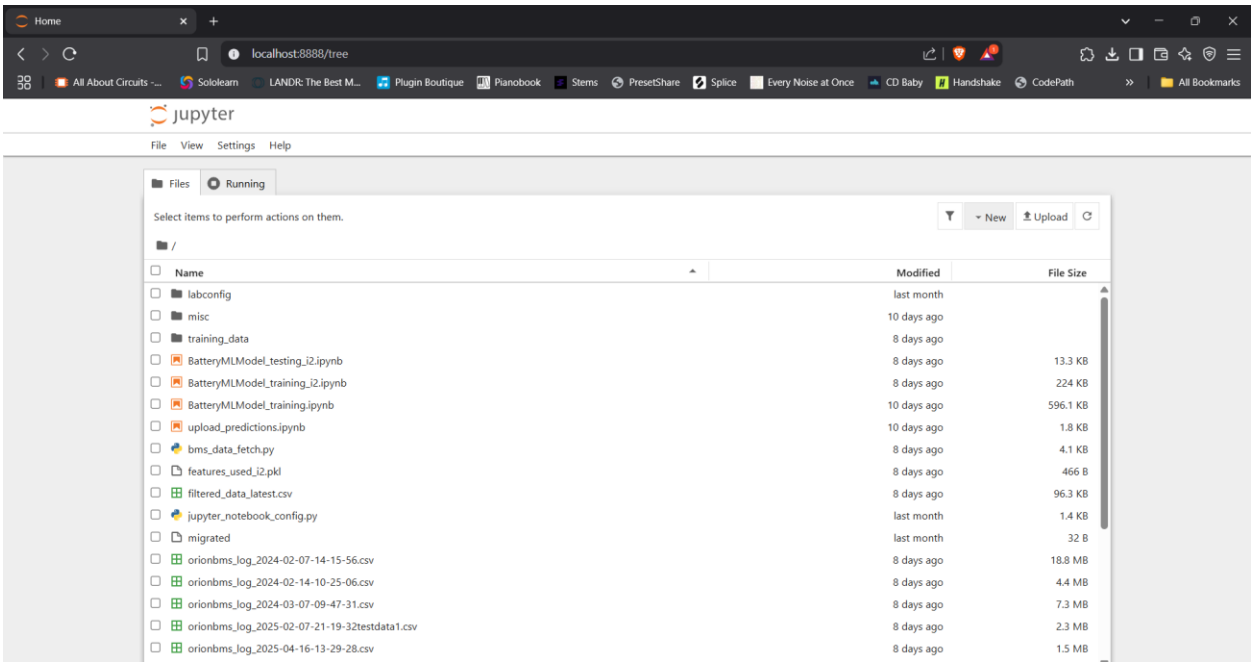
after that, you should be in.

```
2ea45e4ec874eb2e2ca4a2
[I 2025-04-24 21:22:12.866 ServerApp] Use Control-C to stop this server and shut down all kernels (tw
ice to skip confirmation).
[W 2025-04-24 21:22:12.877 ServerApp] No web browser found: Error('could not locate runnable browser'
).
[C 2025-04-24 21:22:12.877 ServerApp]

    To access the server, open this file in a browser:
        file:///home/ubuntu/.local/share/jupyter/runtime/jpserver-20070-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/tree?token=e9688afe93340c87338cfac3582ea45e4ec874eb2e2ca4a2
        http://127.0.0.1:8888/tree?token=e9688afe93340c87338cfac3582ea45e4ec874eb2e2ca4a2
[I 2025-04-24 21:22:12.897 ServerApp] Skipped non-installed server(s): bash-language-server, dockerfi
le-language-server-nodejs, javascript-typescript-langserver, jedi-language-server, julia-language-ser
ver, pyright, python-language-server, python-lsp-server, r-languageserver, sql-language-server, texla
b, typescript-language-server, unified-language-server, vscode-css-languageserver-bin, vscode-html-la
nguageserver-bin, vscode-json-languageserver-bin, yaml-language-server
```

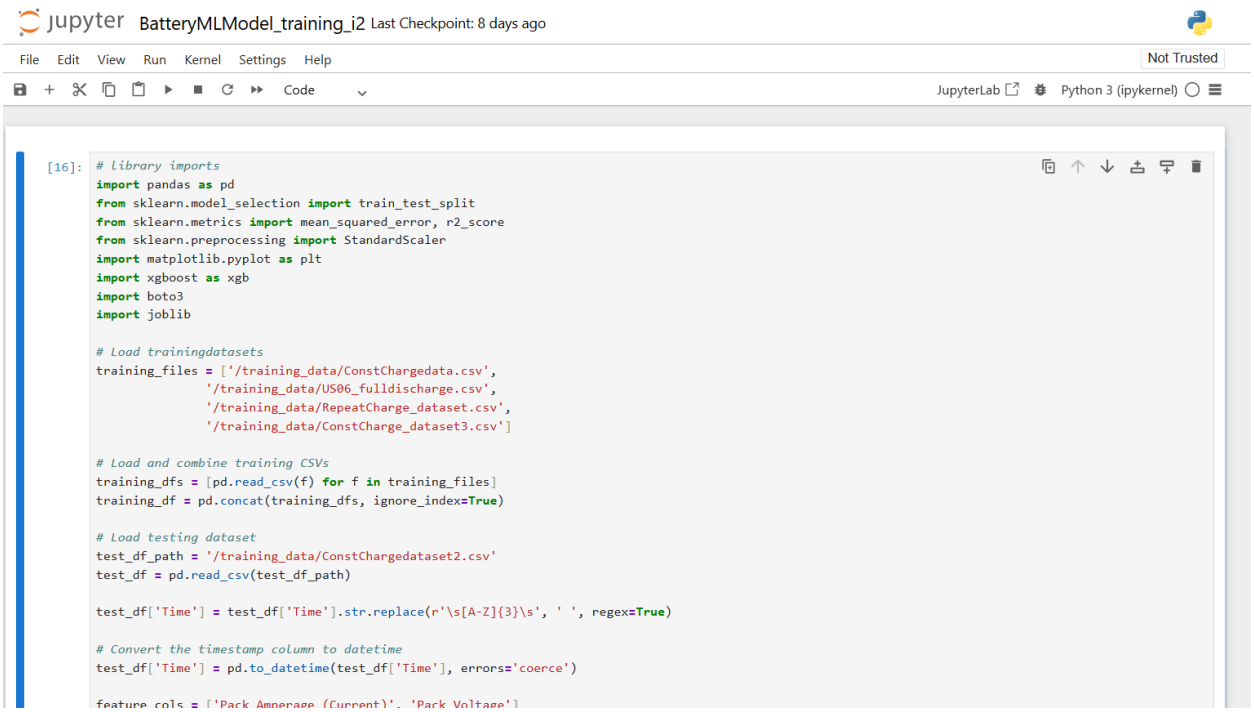
Copy and paste one of the two links into your browser to open the instance. Notice that the local host address contains the four digits 8888, not 8889. If you see 8889, close it out and run the SSH command again to ensure you get in.

Once you're in, the Jupyter notebook should look like the photo below:




Now, it is not formally documented within the instance, but the notebooks **BatteryMLModel_training_i2.ipynb** and **BatteryMLModel_testing_i2.ipynb** contain copies of my XGBoost model and a loop to fetch data from the raw data table and make predictions. **upload_predictions.ipynb** is a manual way to directly upload a CSV file stored in the main directory to append data without the continuous loop. I'll run through the most important file to give you an idea of how the model is written.

### *Training the Model - BatteryMLModel_training_i2.ipynb:*




I've made countless cookie-cutter iterations of the code for our model, but it generally follows the same flow, with distinct points to swap out data and update it for increased accuracy.

some library imports you'll need:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import xgboost as xgb
import boto3
import joblib
```

A list of training data files, you can add more, but only if you have a limited sample rate. E.g. if your data samples data 10 times a second, chances are you may only be able to train on 3 files so preprocessing your data is crucial.

```
# Load trainingdatasets
training_files = ['/training_data/ConstChargedata.csv',
                  '/training_data/US06_fulldischarge.csv',
                  '/training_data/RepeatCharge_dataset.csv',
                  '/training_data/ConstCharge_dataset3.csv']
```

Processing the files and placing them in data frames, as well as DateTime handling, works in this block:

```
# Load and combine training CSVs
training_dfs = [pd.read_csv(f) for f in training_files]
training_df = pd.concat(training_dfs, ignore_index=True)

# Load testing dataset
test_df_path = '/training_data/ConstChargedataset2.csv'
test_df = pd.read_csv(test_df_path)

test_df['Time'] = test_df['Time'].str.replace(r'\s[A-Z]{3}\s', ' ', regex=True)

# Convert the timestamp column to datetime
test_df['Time'] = pd.to_datetime(test_df['Time'], errors='coerce')

feature_cols = ['Pack Amperage (Current)', 'Pack Voltage']
target_cols = ['Pack State of Charge (SOC)']
```

Additionally, it's good to set your features and target to be the same names within your raw data, whether it's Simulink data or Orion BMS data, because if you don't, it will not run due to heterogeneous names.

Since I used an XGBoost model (a variant of a decision tree model), we "simulate" temporal dependencies by creating lag and rolling features here.

```
def create_lag_features(training_df, feature_cols, lags):
    for col in feature_cols:
        for lag in lags:
           training_df[f'{col}_lag{lag}'] = training_df[col].shift(lag)
    return training_df

def create_rolling_features(training_df, feature_cols, windows):
    for col in feature_cols:
        for window in windows:
            training_df[f'{col}_roll_mean_{window}'] = training_df[col].rolling(window=window).mean()
            training_df[f'{col}_roll_std_{window}'] = training_df[col].rolling(window=window).std()
    return training_df

training_df = create_lag_features(training_df, feature_cols, lags = [1, 2, 3])
training_df = create_rolling_features(training_df, feature_cols, windows=[3, 5])
training_df.dropna(inplace=True)
```

In the next  image you can see More spaghetti code to select features at the top. It then sets training and testing features based on the columns selected within your CSV, and (optionally) scales the data to prepare it for the model. The command

**`model = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=100, learning_rate=0.1, max_depth=3)`**

Can be configured to make the model more nuanced by setting how many estimators, or trees, to use, the learning rate, and other hyperparameters. Consider changing this if you know your data is good and want to lower your error rate or other key factors.

The results are stored in another data frame.

```
all_features = [col for col in training_df.columns if col not in target_cols and col != 'Time']
common_features = [col for col in all_features if col in test_df.columns]
x_train = training_df[common_features]
x_test = test_df[common_features]
y_train = training_df[target_cols]
y_test = test_df[target_cols]  # Assuming test_df has the same target column

# Normalize input features
scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

# Initialize the model
model = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=100, learning_rate=0.1, max_depth=3)

# Train the model
model.fit(x_train, y_train)

# Make predictions on the test set
y_pred = model.predict(x_test)

# Create a results DataFrame to combine timestamps, actual, and predicted values
results_df = pd.DataFrame({
    'Time': test_df['Time'],
    'Actual SOC': y_test.values.flatten(),
    'Predicted SOC': y_pred.flatten()
})
```

I also added some functionality to save the model to import it to the testing model.

```
# Sort the results by Timestamp to maintain the time sequence for plotting
results_df.sort_values('Time', inplace=True)

# Save the trained model
joblib.dump(model, 'xgb_model_i2.pkl')

# Save the scaler used for normalization
joblib.dump(scaler, 'scaler_i2.pkl')

# Save common features
joblib.dump(common_features, 'features_used_i2.pkl')
```

Lastly, this code block saves the predictions to a CSV, which can be used to upload your predictions manually if you want. Additionally, it calculates the MSE (mean squared error) and R^2 (correlation) scores to help you understand how accurate the model is. Generally, you want a low error (<5%) and a high correlation (>0.95). Please research to understand overfitting and underfitting, and other concepts that help you gauge if you did a good job training the model.

```
# Define the save path
csv_save_path = "pack_soc_predictions1.csv"

# Save to CSV
results_df.to_csv(csv_save_path, index=False)

print(f"Saved CSV file: {csv_save_path}")

# Evaluate the model's performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error (MSE): {mse}")
print(f"R² Score: {r2}")
```

If you want a more robust model, consider using an LSTM model to capture temporal dependencies. The only caveat is that it is computationally expensive (meaning potential GPU-level processing, especially with a large dataset), so do your research to see if a change is necessary. It also helps to plot all of your results (which are not sent directly to the interface) so you can see visually how accurate the model is.

Training and testing your model in Python covers the prediction side, but you still need realistic data. In the next section, Simulink is used to help to run drive-profile simulations, extract time/current traces, and export CSVs. Those files plug directly into the notebooks to train the model within the notebook instance.

Some last advice: visualizing feature distributions and train/test splits in your notebook is a good idea after loading and combining DataFrames. Before fitting, make sure to plot kernel density estimates or rolling-mean traces to identify outliers. To identify overfitting, draw your learning curves (training vs. validation error) after training. This can be automated with tools like scikit-learn's learning_curve helper. Lastly, save your best model with a timestamped filename (for example, battery_xgb_20250424.pkl) so that you can reverse any performance degradation caused by additional hyperparameter searches.

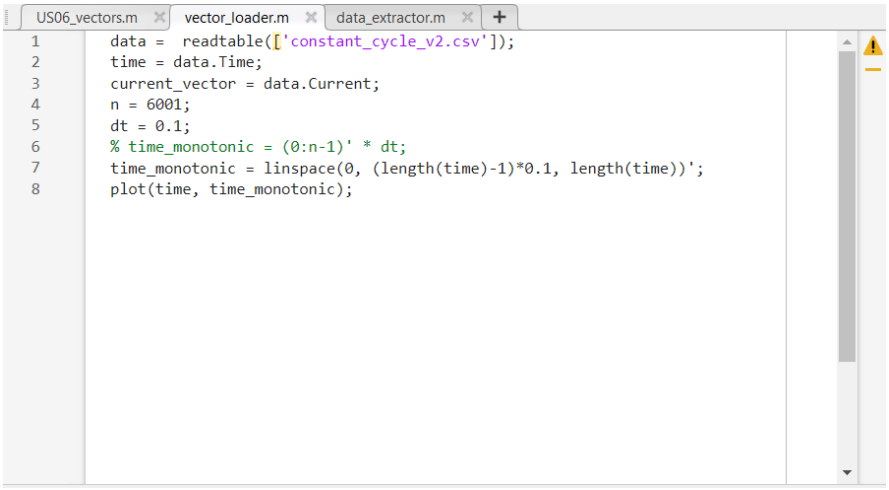
***Simulink***

A few key points:

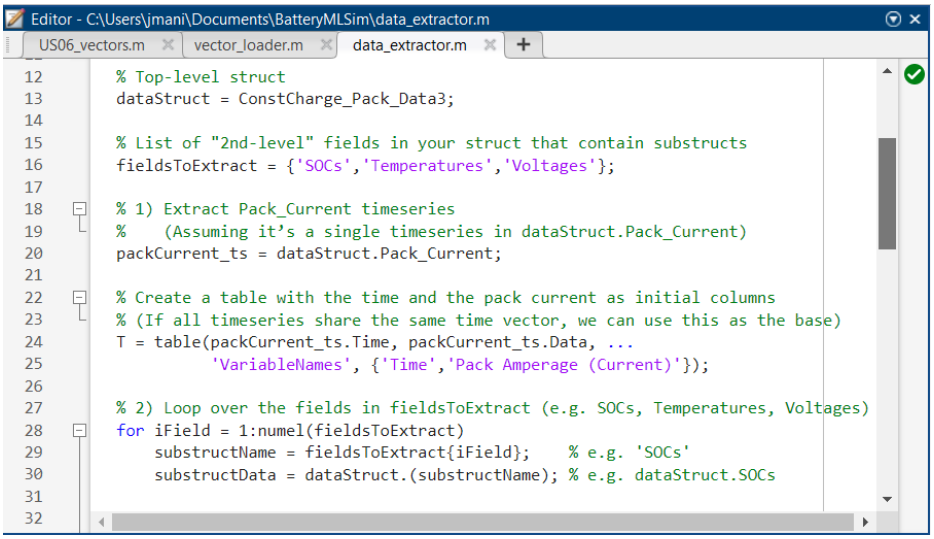
Make sure your Simulink license is active on the EC2 instance (or on your local MATLAB installation) before running the model.

Attached is a compressed MATLAB workspace folder, containing an assortment of drive profiles to generate simulated data as well as the Simulink model I used to train our model. Take a look at the formatting of each file to see which ones are used for current profiles (columns only have time and current, for example)

- **vector_loader takes a given CSV containing current and time and loads them into workspace.**



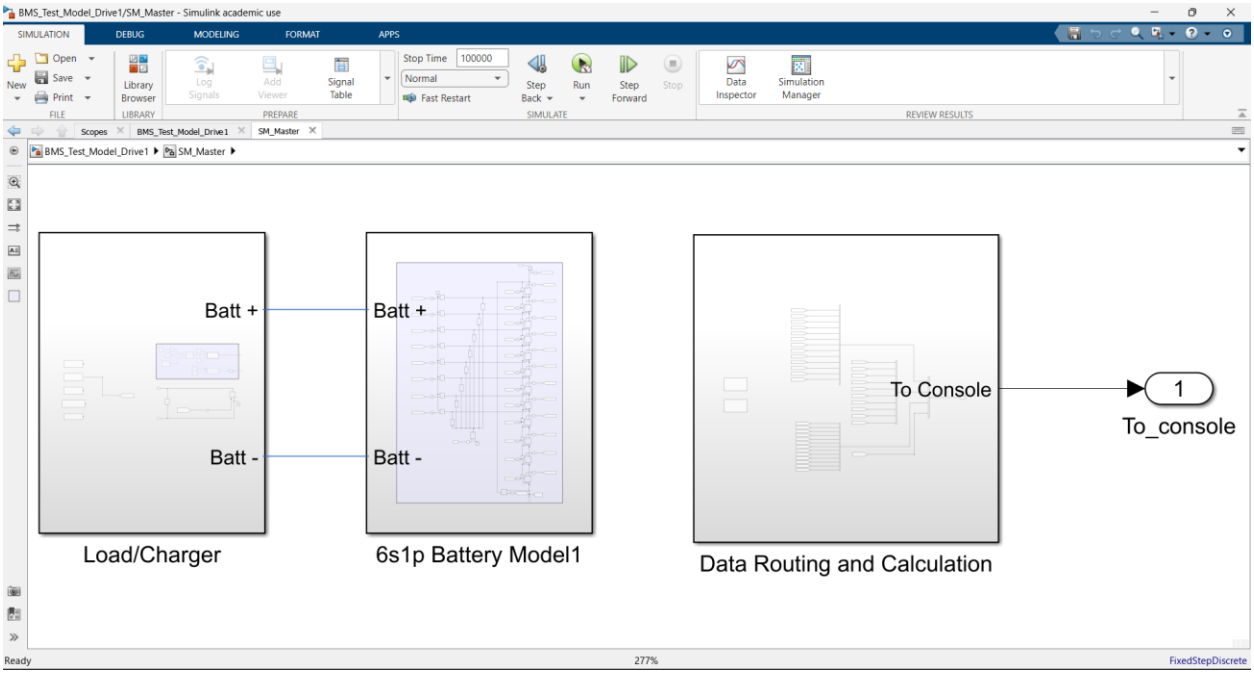

- **data_extractor takes the simulation results of the battery model and exports to a CSV, which becomes the input of the model.**



In the “vector_loader” block, confirm your CSVs contain exactly two columns named time and current; otherwise add a preprocessing script to rename columns automatically. Use the data_extractor script to export traces, but wrap it in a try/except that logs any simulation warnings or solver errors to sim_errors.log.

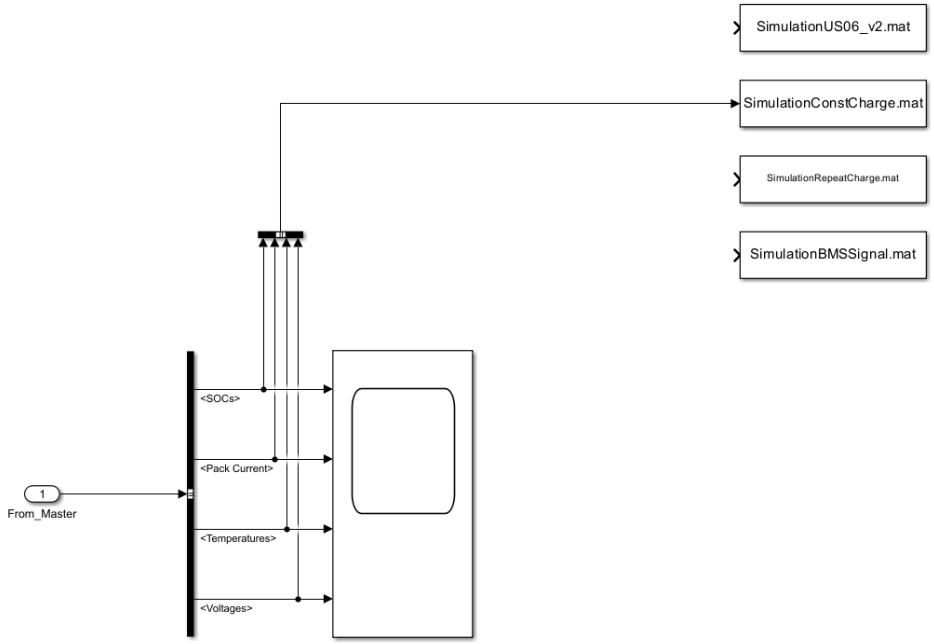
Lastly, standardize CSV output paths (e.g. ./data/raw/drive_profile_01.csv) so downstream scripts always find inputs where expected.

**The model itself:**




- Load/Charger contains the block where you input your current drive profiles.
- The middle block contains the battery model.
- Results are sent to the console.

Within the console itself:




Results are sent to both the scope and a To Workspace block. The name of your file can be loaded into MATLAB and then the data_extractor script can output it to a CSV that you can work with.