Goal: In this lab you will install Jupyter Lab, python and the needed frameworks. After that, you will do a simple guided machine learning application.

Lab 1:

Part 1: Setting Up Python and Installing Packages (Windows, MacOS)

This guide should get you started with getting a usable Python environment installed on your computer for Data Science classes. It will walk you through setting up a virtual environment, activating it, and installing packages.

Step 1: Install Python 3

All of these commands will be run from a terminal (be that iTerm, Terminal, cmd, or Powershell; see the note about PowerShell limitations below).

MacOS

You need a more modern version of Python3 than comes preinstalled; check your version by running the following in a terminal window

```
$> python3 --version
```

If you see a message saying python3 is not installed or a version number that is smaller than `3.8`, you should update your python by installing a stable version from https://python.org.

Windows

If you run Windows 10, there are two possibilities for installing Python 3

Recommended: Microsoft Store

You may be able to install Python 3 directly from the Microsoft Store; this seems to be the easiest way to accomplish this. This only is available on newer builds of Windows 10.

Download and install manually (only option for previous versions of Windows or if the Microsoft Store gives you an error)

Download the installer from here. When you run the installer, it is very important that you check the box at the bottom of the setup window that says "Add Python 3.x to PATH" (the x depends on which version you downloaded)

Note If you install python manually, you should run `python` instead of `python3` in the following steps.

Step 2: Verify that you have Python 3

Open a *new* terminal window (CMD/Powershell, Terminal, xterm, etc), and run the following command: `python3 --version` Depending on what version of python installed, you should see something that looks like:

```
C:\Users\choot>python3 --version
Python 3.8.6
```

or

```
C:\Users\choot>python3 --version
Python 3.9.0
```

The second number doesn't matter; what's important is that you have Python 3.x.y installed; if you see Python 2, you may need to install Python 3 again and make sure your path is set correctly. You should have at least Python 3.8 (JupyterLab requires >= Python 3.5, other packages may have newer dependencies)

Step 3: Create our Data Science Virtual Environment

A Virtual Environment is a way to isolate installed packages and libraries in a way that does not conflict with system installations of utilities. To use a virtual environment we must first create it. If you want to put your virtual environment in a specific location, open a terminal in that directory; I will just put mine in my home directory (`C:\Users\choot`). The virtual environment (which I will call a venv) will be created in a directory where we're currently at; I am going to name mine `ds-venv` (you can change the name if you want).
To create the virtual environment, run

```
python3 -m venv ds-venv
```

This creates a copy of the python environment in the directory ds-venv; check out the contents with `dir` or `ls` depending on your OS

```
C:\Users\choot>dir ds-venv
Directory of C:\Users\choot\ds-venv

10/14/2020  09:17 AM    <DIR>          .
10/14/2020  09:17 AM    <DIR>          ..
10/14/2020  09:17 AM    <DIR>          Include
10/14/2020  09:17 AM    <DIR>          Lib
10/14/2020  09:17 AM               119 pyvenv.cfg
10/14/2020  09:17 AM    <DIR>          Scripts
               1 File(s)            119 bytes
               5 Dir(s)  114,095,161,344 bytes free
```

At this point, our virtual environment is set up. Whenever we want to use python, we need to activate the environment

Step 4: Activate the environment

MacOS/Linux

```
source ds-venv/bin/activate
```

Windows

Make sure you use `cmd` or allow scripts to be run from PowerShell

```
ds-venv\Scripts\Activate
```

You should now see that your terminal has changed:

```
C:\Users\choot>ds-venv\Scripts\Activate

(ds-venv) C:\Users\choot>
```

Step 5: Installing packages

At this point, we can install packages. The normal way to install a package is to run the command:

```
pip install <packagename>
```

You can specify multiple packages to install by separating the list with spaces.

Installing Numpy (Windows only)

A lot of packages we will use require the `numpy` package; unfortunately on Windows the newest version exposes a bug in the Windows math co-routines. Before installing anything else, we can specify the version of numpy to get installed so we have a safe working version:

```
pip install numpy==1.19.3
```

If at any time you need a specific version of a package, you can specify it with the `==version` addition to the package name.

Installing Numpy (Mac)

No version specification is needed. Just run

```
pip install numpy
```

Finish up by Installing JupyterLab, Matplotlib and Pandas

Run

```
pip install jupyterlab matplotlib pandas
```

You should see a lot of downloads; it's grabbing all of the prerequisites from various sources and installing them.

Submission 1 of 10: Progress mark (10)
Screen shot package installation report

When it's done, test that it installed by running `jupyter lab`:

```
(ds-venv) C:\Users\choot>jupyter lab
```

You should get a new browser window to pop up with your running instance of JupyterLab! Success!

Submission 2 of 10: Progress mark (5)
Show running JupyterLab

JupyterLab

One option (if you want to avoid using a terminal) is to open the directory where you created your virtual environment, and run (double click) `jupyter-lab` from either the `bin` or `Scripts` directory, depending on whether you're on MacOS or Windows, respectively. Windows may open a web browser that says it cannot find the file; copy and paste the URLs (usually `httplocalhost:8888/?token=<sometoken>`) and paste it in your browser. You should be able to create shortcuts to this `jupyter-lab` to make it easier to find. Consult your OS and the internet on how to do this correctly.
Otherwise:

1. Open a terminal
2. Activate your virtual environment
3. run `jupyter lab`

Part 2: Basic Steps For Creating a Model

Creating a workspace:

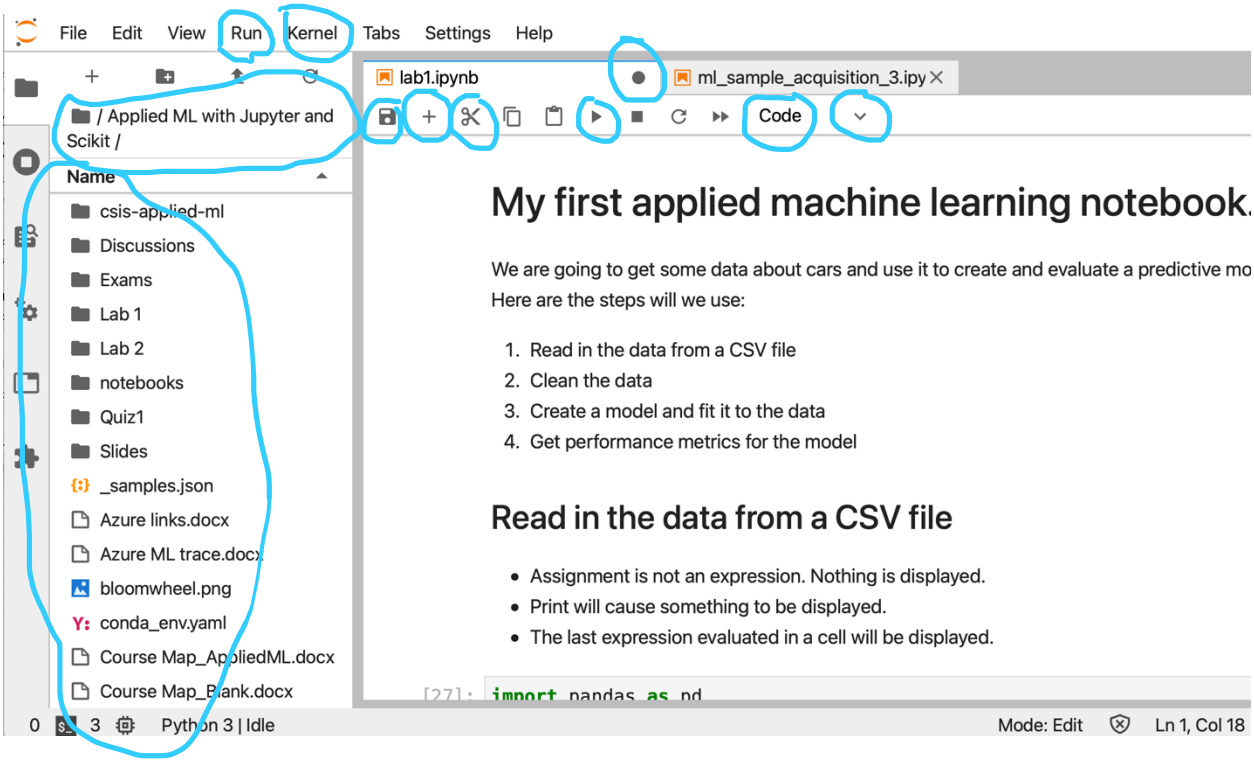
This is not a formally defined part of JupyterLab, but it is often usefull to group together related notebooks and data into a single directory. You can create such a directory or put them in your virtual environments directory. (Note: Windows may require that files be in or under the virtual environment directory.)

Starting a notebook:

While we can create a new notebook, for this exercise copy lab1_starter.ipynb into your workspace directory and then open it.

Submission 3 of 10: Progress mark (10)
JupyterLab with notebook open

It should look something like this:



A few things to notice

Run: This menu allows you to run one or more code cells. Affects are cumulative until restart.
Kernel: This menu allows you to interrupt python and restart it.
Current Folder: Where we currently are
Contents of the Folder: What we have
Lab1.ipynb: The notebook we are currently looking at. (There is another notebook that has been loaded, but is not visible. It has its own kernel.
Filled Circle: We have unsaved content in this notebook.
Disk Icon: We can save this notebook. (We can also do this from the topmost menu.)
Plus Icon: We can add a cell to this notebook.
Scissor Icon: Cut the selected cells.
Play Icon: Run the selected cells.
Code indicator: The type of the selected cell. It will be Code or Markdown. In this case, the selected cell is off screen.
Down caret: A dropdown menu that allows us to change the type of the current cell between code and markdown.

Getting the data:

We need to get the data where we can find it. Copy automobile_price_data3.csv into the same directory (workspace) as lab1_starter.ipynb. In the third cell, enter the following two lines of python code.

```
import pandas as pd

data_frame = pd.read_csv("automobile_price_data3.csv")
```

1. We want to use the pandas framework so we import the contents. To use it we will use the abbreviation pd.

2. We use the function read_csv from pandas. The argument gives the name of the file. This will create a dataframe that we then assign to the variable data_frame.

Run the code cell. No output should appear. This because the cell is executed in a read-eval-print loop. Each line is run one at a time. If a line is an expression and it is the last line, the expression is displayed. Otherwise, something else like print is needed to cause an output.

In this case, the last line is an assignment, which is not an expression.

In the third cell add the following line of code.

```
3**10
```

1. This is the expression 3 to the 10^{th} power.

Run the code cell. You should see 59049

In the third cell add the following line of code.

```
print(data_frame.keys())
```

1. We are asking the data_frame for the list of keys (feature names) and then printing them.

Run the code cell. The value won't show up in the output. (It is no longer the last line.) But you should see a list of names like "make" and "body".

Submission 4 of 10: Progress mark (10)
Keys of Dataframe acquired

Exploring the data:

Now that we have our data in a data frame we can take a quick look for any problems like missing data values.

In the third cell add the following lines of code.

```
print()
data_frame.head(n=10)
```

1. Print an empty line
2. Head displays rows from the front of the data frame. We can specify n or let it use the default. We display the first 10 rows. Even if we randomize the order of the instances in the data frame, it still remembers their original positions. This is the last line in the file and it is an express, so it is displayed. The NaN stands for "Not a Number" and indicate a missing value.

Run the code cell. You should see the start of the data frame. The NaN stands for "Not a Number" and indicates a missing value.

In the third cell add the following line of code.

```
data_frame.info()
```

1. Report on each of the features – what type of values are they, how many values are present. We should look for missing values and make sure that the type is appropriate. This always produces output.

Run the code cell. Make note of the types

In the third cell add the following line of code.

```
data_frame.hist(bins=30)
```

1. Produces histograms of numerical features. In this case, we specify that there will be 30 bins. We can look for any suspect distributions. This always produces output.

Run the code cell. These distributions are not too skew, but are not normal (bell shaped) distributions.

Submission 5 of 10: Progress mark (10)
Histograms shown

Cleaning the data:

We have to do something about the missing data values.

Strategy: Get rid of the data instance.
Strategy: Get rid of the feature.
Strategy: Replace the missing value with a representative value

We saw that the column normalized-losses has a number of NaN so it is a good candidate for removal.

Use the + to create a new cell after the "Clean the data" markdown cell. By default, new cells will be code cells.

In the new cell add the following lines of code.

```
data_frame.drop(axis='columns', labels='normalized-losses', inplace=True)
data_frame.dropna(axis='index', how="any", inplace=True)
```

1. The drop method unconditionally removes a row or column. The axis specifies whether we are getting rid of a row or col. In this case, we are removing the column with the name "normalized-losses". Be cautious. Spelling must match the name of the feature in the data frame. Uppercase/lowercase matters. The inplace argument determines if we return a new transformed data frame or as we specify here, just modify the current one.
2. The dropna will allow use to drop a row or col if it has missing values. Here the axis choice lets us know we are removing rows where ***any*** data value is missing. Again, we will modify the dataframe without creating a new one.

Run the code cell. It shouldn't produce any output. (You could do an info after to verify the operation.)

Run the code cell again. You should see an error. The problem is that we are trying to remove the column a second time, but that feature is no longer in the data frame. This is the same kind of error that you see if you misspell a feature name.

Submission 6 of 10: Progress mark (10)
Screen shot of error. Make sure to include the last line.

Create and train a model:

Our first task is to split the data set into two pieces – a training set and a test set. For the first time we are going to pull in Scikit-learn. We are going to do a simple randomized split of what is left in the data set after cleaning.

Create a new code cell after the Create a Model cell.

In the newest cell add the following lines of code.

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(data_frame,
test_size=0.2, random_state=123)
print(len(train_set), len(test_set))
```

1. We are pulling in the function train_test_split from scikit-learn from the model_selection grouping. Since we explicitly mention the function, we can use it directly without specifying the package name.
2. The first argument of the split must be present and must be a data frame. We specify that the size of the test set is 20%. The split is randomized. I want the split to be the same each time while developing the code, so initialize the randomizer to always start in the same place. If we left off the parameter, we would get a new sequence each time. The result of the split is two data frames which are assigned to the variables train_set and test_set.
3. The len function applied to a data set, returns the number of rows that it has. We print that number for each of the two data sets from the split. If you want, you can use head to examine values from the split.

Run the code cell. The sizes should be 157 and 40

Submission 7 of 10: Progress mark (10)
Screen shot of split code

In the newest cell add the following lines of code.

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
```

1. We are pulling in the class LinearRegression from scikit-learn from the linear_model grouping.
2. We create a new instance of the LinearRegression class using the () and assign the result to the variable reg. We have a model, but it has not yet been trained on data.

Run the code cell. The sizes should be 157 and 40.

In the newest cell add the following lines of code.

```
X = train_set[['weight', 'engine-size', 'bhp', 'mpg']]
y = train_set["price"]
reg.fit(X, y)
```

1. We are extracting the features we want to train the model on. The [[name, name, …]] gives us an array. Each of the desired features are listed. Even if you only want to use a single feature to make predictions from, you must still use [[name]]. The capitalization of X is only there to help us remember that it is 2 dimensional.
2. We are extracting the target (correct) value from the data frame. In this case, we know It is a single vector, so we use single brackets [name]. The lower case on the y is to help us remember that it is a vector.
3. Take the model we created and train it on the specified data. The fit method does the training. Models in scikit-learn have some standard functions like fit that are used for certain tasks. This chunk of three lines is typical pattern we will see often.

Run the code cell. The sizes should be 157 and 40. The fit also returns a value which is the instance of the model.

Submission 8 of 10: Progress mark (10)
Screen shot training code and result

Evaluate the performance of the model.

Now that we have trained the model we can take a look at the internal parameters. In this case, we have a bias value and a coefficient for each of the 4 features we selected. If you give me four feature values, to make a price prediction I multiply each feature by its coefficient and add them to the bias. (This is specific to a linear regression model and other models will have different internal parameters.)

After the performance markdown cell add a new code cell.

In the newest cell add the following lines of code.

```
print("The bias is " , reg.intercept_)
print("The feature coefficients are ", reg.coef_)
```

1. The LinearRegression class will have internal parameters that we can access directly. The bias is given by intercept_ (The trailing underscore is there and indicates a value that you shouldn't change directly. It will change in response to training the model with the fit function.) Print the value with a handy label.
2. The LinearRegress class has an internal parameter which is a numpy array held in the variable coef_.

Run the code cell. The values should be approximately -18000 and 4, 82, 67 and 73. Roughly, features with larger coefficients are more important, but that depends on the range of values for the features.

Submission 9 of 10: Progress mark (10)
Screen model internal parameters print.

In the newest cell add the following lines of code.

```
print("The score for the training set is", reg.score(X,y))
```

1. The LinearRegression class will have internal parameters that we can access directly. The bias is given by intercept_ (The trailing underscore is there and indicates a value that you shouldn't change directly. It will change in response to training the model with the fit function.) Print the value with a handy label.
   The LinearRegress class has an internal parameter which is a numpy array held in the
   The bias is given by intercept_ (The trailing underscore is there and indicates a value

Run the code cell. The score should be about 82%.

In the newest cell add the following lines of code.

```
# Check the performance on the test set
X_test = test_set[['weight', 'engine-size', 'bhp', 'mpg']]
y_test = test_set["price"]
print("The score for the test set is", reg.score(X_test,y_test))
```

1. While we often use the markdown text cells to add in comments, we can still use pythons traditional # to indicate a comment.
2. We select the same features as we did in the training.
3. We select the same target feature as we did in training.
4. Compute the score on the test data. A significant difference in the score likely indicates a problem in the data and/or the choice of model.

Run the code cell. The score should be about 55%.

For a linear regression, the score indicates the predictive power of the model. The closer the score is to 1, the better the model is. A score of 75% indicates that 75% of the variation in the target is explained by the input features. The difference in these scores is worrisome. Changing the random state could give some insight.

Submission 10 of 10: Progress mark (10)
Screen shot of performance