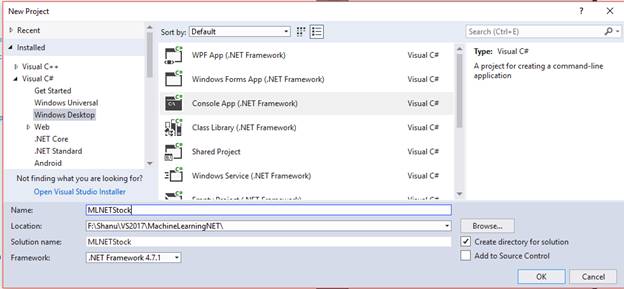
**Getting Started with ML.Net – Stock Prediction**

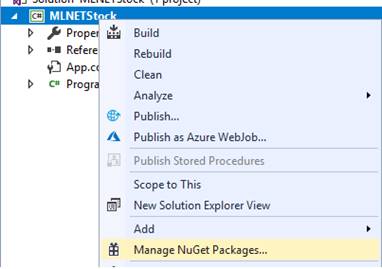
**Step 1 - Create C# Console Application**

After installing the prerequisites, click Start >> Programs >> Visual Studio 2017 >> Visual Studio 2017 on your desktop. Click New >> Project. Select Visual C# >> Windows Desktop >> Console APP (.Net Framework). Enter your project name and click OK.

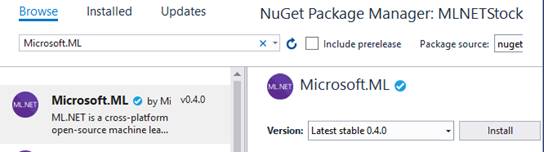


**Step 2 – Add Microsoft ML package**

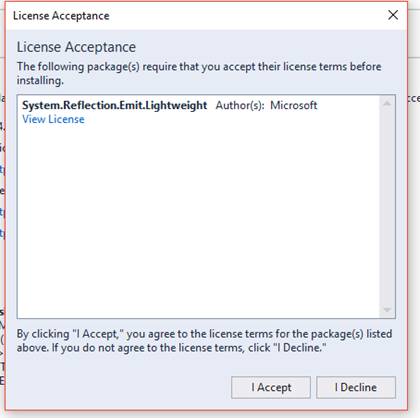
Right-click on your project and click on Manage NuGet Packages.



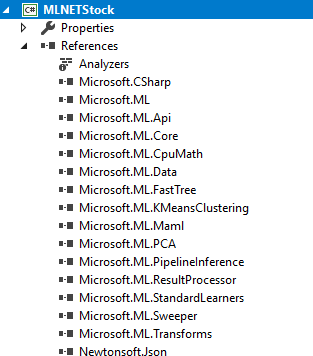
Select Browse tab and search for Microsoft.ML



Click on Install, I accept it and wait until the installation is complete.



We can see the Microsoft.ML package has been installed and all the references for Microsoft.ML have been added to our project references.

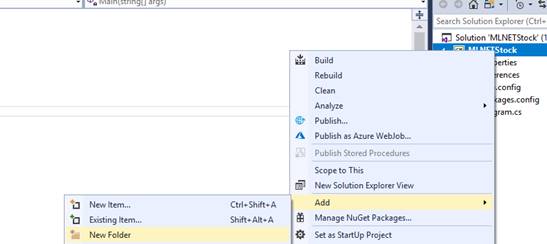


**Step 3 – Creating Train and Evaluate Data**

Now, we need to create a Model training and evaluate the dataset. For creating this we will add two CSV files,  one for training and one for the evaluation. We will create a new folder called data in our project to add our CSV files.

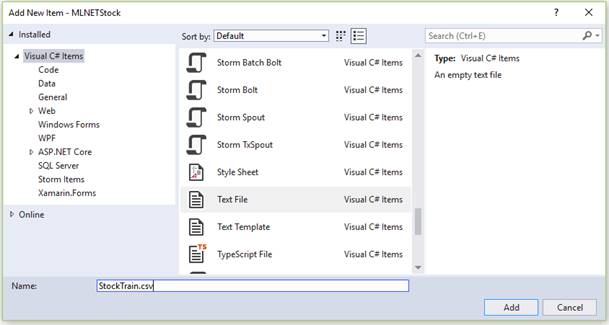
**Add Data Folder**

Right-click the project and Add New Folder and name the folder as “Data”

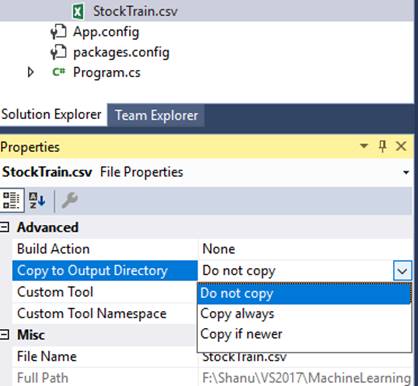


**Creating Train CSV file**

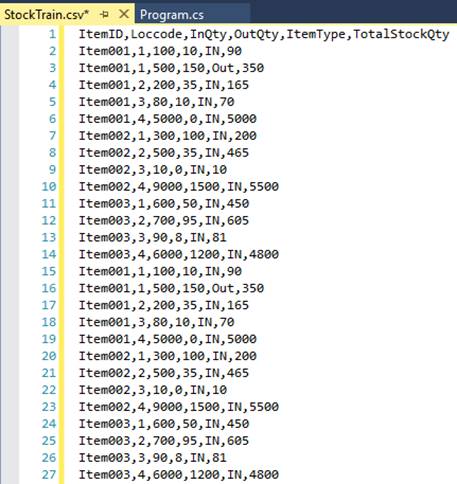
Right-click the Data folder click on Add >> New Item >> select the text file and name it as “StockTrain.csv”



Select the properties of the “StockTrain.csv” change the Copy to Output Directory to “Copy always”



Add your CSV file data like below.



Here, we have added the data with the following fields.

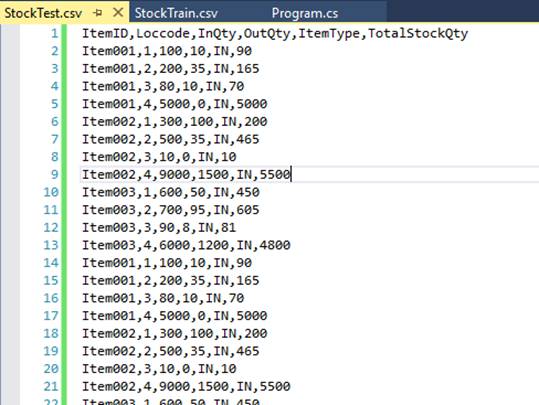
Note that we need to fix the label and the features.

**Label**A Label is a field which we need to predict in our sample. Here TotalStockQty is the Label where we will be predicting the total remaining Stock Quantity of an item. We make the other fields features.

* ItemID - Stock Item ID (Feature)
* Loccode – Warehouse Location of the Item (Feature)
* InQty – Total Item received in location (Feature)
* OutQty – Total Item delivered from the location (Feature)
* ItemType - ItemType “In” means as local manufactured and “out” means as outsourced. (Feature)
* TotalStockQty - Total no of item stock in location. (Label)

**Note**We need a minimum of 100 records of data to be added to train our Model.

Like this we will add one more CSV file and add similar information for evaluating. We have created one more CSV file named  “StockTest.csv”



**Step 4 – Creating Class for Input Data and Prediction**

Now, we need to create a class for Input Data and prediction. For doing this right-click our project and add a new class and name it as “ItemStock.cs”

In our class, first we need to import the Microsoft.ML.Runtime.API

1. using Microsoft.ML.Runtime.Api;

Next, we need to add all our columns like our CSV file in the same order in our class and set the column as 0 to 5.

1. **public** **class** ItemStock
2. {
3. [Column("0")]
4. **public** string ItemID;
6. [Column("1")]
7. **public** **float** Loccode;
9. [Column("2")]
10. **public** **float** InQty;
12. [Column("3")]
13. **public** **float** OutQty;
15. [Column("4")]
16. **public** string ItemType;
18. [Column("5")]
19. **public** **float** TotalStockQty;
20. }

Now, we need to create a prediction class and, in this class, we need to add our prediction column. Prediction column is the same as our label. Our label is “TotalStockQty”. Our aim of this program is to predict and display the final result of the Total Stock quantity from the trained model. We add “TotalStockQty” as a prediction column in our prediction class.

**Note**Important to be noted is in the prediction column we need to set the column name as the “Score” and also set the data type as the float.

In the regression model, the scoring column contains the predicted results.

1. **public** **class** itemStockQtyPrediction
2. {
3. [ColumnName("Score")]
4. **public** **float** TotalStockQty;
5. }

**Step 5 – Program.cs**

To work with ML.NET, we open our “program.cs” file and first, we import all the needed ML.NET references.

1. using Microsoft.ML;
2. using Microsoft.ML.Data;
3. using Microsoft.ML.Models;
4. using Microsoft.ML.Trainers;
5. using Microsoft.ML.Transforms;

Also import the below to your program.cs file.

1. using System.IO;

**Dataset Path**

We set the Traincsv data, Evaluate data and Model data path. For the traindata we give “StockTrain.csv” path and do the same for the evaluate csv file.

The final trained model needs to be saved for evaluating and producing results. For this, we set the modelpath with the “Model.zip” file. The trained model will be saved in the zip file automatically during runtime of the program in our bin folder with all needed files.



1. **static** readonly string \_Traindatapath = Path.Combine(Environment.CurrentDirectory, "Data", "StockTrain.csv");
2. **static** readonly string \_Evaluatedatapath = Path.Combine(Environment.CurrentDirectory, "Data", "StockTest.csv");
3. **static** readonly string \_modelpath = Path.Combine(Environment.CurrentDirectory, "Data", "Model.zip");

Change the Main method to async Task Main method like the below code,

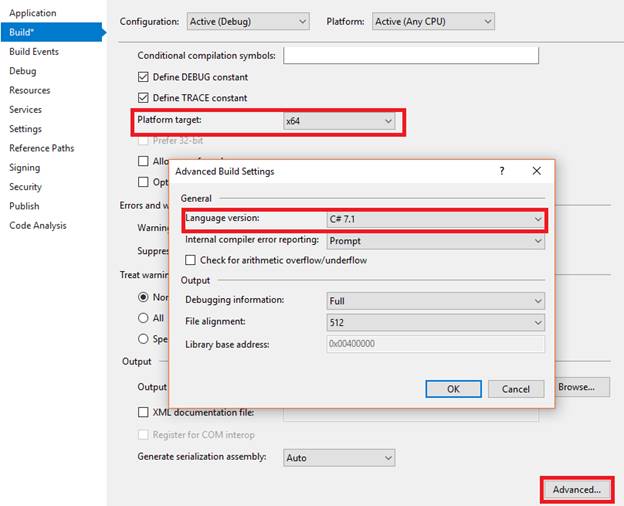
1. **static** async Task Main(string[] args)
2. {
3. }

Before doing this, we need to perform 2 important tasks to successfully run our program

First is to set Platform Target as x64. The ML.NET only runs in x64, for doing this right-click the project and *select properties >> Select Build and change the Platform target to x64.*

 In order to run with our async Task Main method, we need the change the Language version to C#7.1

In the Project Properties >> Build tab >> click on the Advanced button at the bottom and change the Language Version to C#7.1



**Working with Training Model**

First, we need to train the model and save the model to the zip file. For this in our main method we call the predictionModel method and pass the ItemStock and itemStockQtyPrediction class and return the model to the main method:

1. **static** async Task Main(string[] args)
2. {
3. PredictionModel<ItemStock, itemStockQtyPrediction> model = await TrainourModel();
4. }
6. **public** **static** async Task<PredictionModel<ItemStock, itemStockQtyPrediction>> TrainourModel()
7. {
8. }

**Train and Save Model**

In the above method we add the function to train the model and save the model to the zip file.

**LearningPipeline**

In training, the first step will be working the LearningPipeline().

The LearningPipeline loads all the training data to train the model.

**TextLoader**

The TextLoader is used to get all the data from tran CSV file for training and here we set the useHeader: true to avoid reading the first row from the CSV file.

**ColumnCopier**

In the training model, the important part is the predicted values to be defined. In our example, we have set the TotalStockQty as the label. In training model, we need to set the prediction label by using the ColumnCopier. We can see in our below code part in ColumnCopier we have given the TotalStockQty and Label.

**CategoricalOneHotVectorizer**

The training model needs numeric features to transform the categorical data like our ItemID, ItemType into numbers.

**ColumnConcatenator**

Next, we add all our featured columns to be trained and evaluated.

**Adding Learning Algorithm**

**FastTreeRegressor**

The learner will train the model. We have selected regression for our sample and we will be using FastTreeRegressor learner . FastTreeRegressor is one of the regression leraners provided by the ML.NET. Here we add the FastTreeRegressor to our pipeline.

**Train and Save Model**

 Finally, we will train and save the model from this method.

1. **public** **static** async Task<PredictionModel<ItemStock, itemStockQtyPrediction>> TrainourModel()
2. {
3. **var** pipeline = **new** LearningPipeline
4. {
5. **new** TextLoader(\_Traindatapath).CreateFrom<ItemStock>(useHeader: **true**, separator: ','),
6. **new** ColumnCopier(("TotalStockQty", "Label")),
7. **new** CategoricalOneHotVectorizer(
8. "ItemID",
9. "ItemType"),
10. **new** ColumnConcatenator(
11. "Features",
12. "ItemID",
13. "Loccode",
14. "InQty",
15. "OutQty",
16. "ItemType"),
17. **new** FastTreeRegressor()
18. };
20. PredictionModel<ItemStock, itemStockQtyPrediction> model = pipeline.Train<ItemStock, itemStockQtyPrediction>();
22. await model.WriteAsync(\_modelpath);
23. **return** model;
24. }

**Evaluate Model**

Next, we need to evaluate the model by checking the model with the test data and predicting the final results. For this we call the method Evaluate from our main method and pass the trained model to this method.

1. **static** async Task Main(string[] args)
2. {
3. PredictionModel<ItemStock, itemStockQtyPrediction> model = await TrainourModel();
5. Evaluate(model);
6. }

In the Evaluate method, we pass the trained model. We need to evaluate the model with our sample csv test data. In this method using textloader, we load all the data from test data csv file for evaluating the model.

Using RegressionEvaluator method we evaluate the model with the testdata and produce evaluation metrics

We display both the RMS (Root mean Squared) and RSquared metrics value.

RMS is one of evaluation metrics where the lower RMS value is treated as the better model.

RSquared is another evaluation metric where the value will be between 0 to 1. If the value is closer to 1 is the better model.

1. **private** **static** **void** Evaluate(PredictionModel<ItemStock, itemStockQtyPrediction> model)
2. {
3. **var** testData = **new** TextLoader(\_Evaluatedatapath).CreateFrom<ItemStock>(useHeader: **true**, separator: ',');
4. **var** evaluator = **new** RegressionEvaluator();
5. RegressionMetrics metrics = evaluator.Evaluate(model, testData);
7. Console.WriteLine($"Rms = {metrics.Rms}");
8. Console.WriteLine($"RSquared = {metrics.RSquared}");
10. }

**Prediction Results**

Now its time for us to produce the result of predicted results by model. For this we will add one more class and, in this Class, we will give the inputs.

Create a new Class named “ItemStocks.cs“

We add the values to the ItemStock Class which we already created and defined the columns for Model training. Here, we have created stock input and we have given the values as below,

**Note**  
In each Stock, we haven’t given the input for the TotalStockQty and we have set the TotalStockQty = 0 as the model needs to predict the result from the trained model and produce the result.

1. **static** **class** ItemStocks
2. {
3. internal **static** readonly ItemStock stock1 = **new** ItemStock
4. {
5. ItemID = "Item001",
6. Loccode = 1,
7. InQty = 100,
8. OutQty = 10,
9. ItemType = "IN",
10. TotalStockQty = 0 // predict it. Actual Total Stock Quantity is = 90
11. };
13. internal **static** readonly ItemStock stock2 = **new** ItemStock
14. {
15. ItemID = "Item003",
16. Loccode = 4,
17. InQty = 6000,
18. OutQty = 1200,
19. ItemType = "IN",
20. TotalStockQty = 0 // predict it. Actual Total Stock Quantity is = 4800
21. };
22. }

We can see in our StockTrain.csv file we have added the values of above stock1 and we can see as we have given TotalStockQty as 90 and the Model needs to predict this result and produce the result.

Getting Started with Machine Learning DotNet (ML.NET)

Same  for Stock2 --  we have added the values in the StockTrain.csv file , we can see as we have given TotalStockQty as 4800 and the Model needs to predict this result and produce the result.

Getting Started with Machine Learning DotNet (ML.NET)  
  
 **Produce the Model Predicted results**

In our program's main method, we will add the below code at the bottom after Train and Evaluate method call to predict the Stock Quantity details and display the final predicted results from the model to users in the command window.

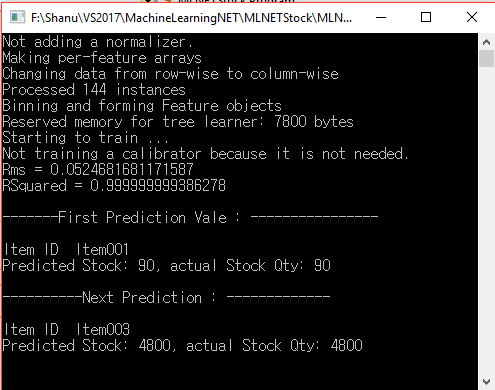
Here we display both Stock1 and Stock2 prediction results. The prediction.TotalStockQty will have the model predicted results and we display the results to the end users of the predicted Stock Quantity with actual input given by us.

1. Console.WriteLine("-------First Prediction Vale : ----------------" );
2. itemStockQtyPrediction prediction = model.Predict(ItemStocks.stock1);
3. Console.WriteLine("Item ID  {0}", ItemStocks.stock1.ItemID);
4. Console.WriteLine("Predicted Stock: {0}, actual Stock Qty: 90 ",      prediction.TotalStockQty);
5. Console.WriteLine(" ");
6. Console.WriteLine("----------Next Prediction : -------------");
7. Console.WriteLine(" ");
8. prediction = model.Predict(ItemStocks.stock2);
9. Console.WriteLine("Item ID  {0}", ItemStocks.stock2.ItemID);
10. Console.WriteLine("Predicted Stock: {0}, actual Stock Qty: 4800 ",   prediction.TotalStockQty);
11. Console.ReadLine();

**Build and Run**

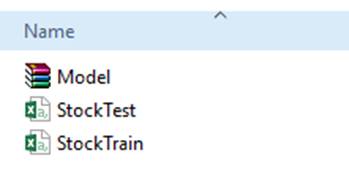
When we run the program, we can see the result in the command window like below.

We can see the Rms value as: 0.052 as we already discussed in this article if the Rms vale is near to 0 then the model is good in prediction and also, we can see the RSquared =0.999. As we already discussed in this article, if the RSquared value is near to 1 then the model is good and we can see the predicted result of both the Stock1 and Stock2 as both have predicted the same Stock quantity as the actual one. Our Model is very good in prediction and the predicted value is exactly the same as the actual value.



**Trained Model Information**

After we build and run the program we can see the Model.zip file in the Data folder inside the Application root bin folder. We can also see our StockTrain and StockTest csv file. The Model will be trained with this csv file and the trained model result will be stored in Model.zip file for predicting the result.



**Conclusion**

ML.NET (Machine Learning DotNet) is a great framework for all the dotnet lovers who are all looking to work with machine learning. Now the only preview version of ML.NET is available and I can’t wait until the release of the public version of ML.NET. Here in this article, I have used the regression model to predict and display the Total Stock Quantity. If you are .Net lovers, not aware about Machine Learning and looking forward to working with machine learning then ML.Net is for you. Hope you all enjoy reading this article and see you soon with another post.