What can you do in ML.NET with C#?

Exploring the different machine learning tasks in ML.NET

Matt Eland Dec. 20, 2021

What is ML.NET?

ML.NET is Microsoft's open source cross-platform machine learning library for .NET applications that allows you to perform **machine learning** tasks using C#, F#, or any other .NET language.

Additionally, ML.NET supports models built in other machine learning frameworks such as TensorFlow, ONNX, Infer.NET and others.

For those who do not yet have deep data science skills and knowledge of the various machine learning algorithms, ML.NET also offers AutoML which automates the training of certain types of machine learning models which helps you focus more on setting up the experiment and working with the trained model.

All of these things combine to make ML.NET a very effective way to work with machine learning tasks using applications you already have and skills you already know.

In this article we'll use C# to explore the high-level tasks that ML.NET can do and where to find more information about each type of machine learning task you might want to perform.

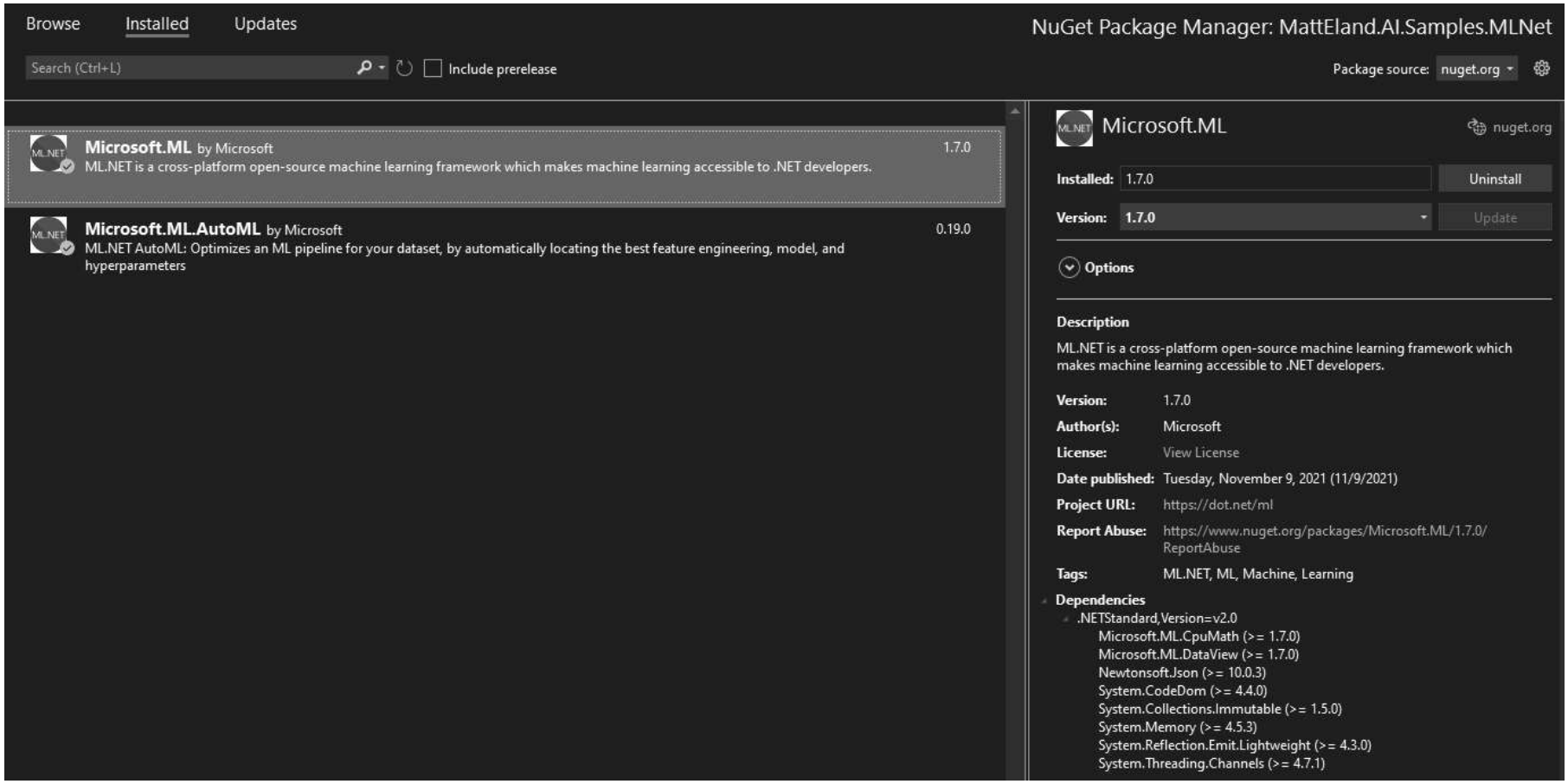
Note: I also recommend reading using AutoML and ML.NET to predict ESRB ratings for video games since it serves as an in-depth intro to ML.NET for a single task.

Note: Typically my articles have YouTube videos embedded in them. The YouTube version of this article is coming soon as time allows. I recommend you subscribe to my channel to get notification when it is updated.

Installing ML.NET

ML.NET can be installed via NuGet Package Manager from Visual Studio for any project that supports .NET Standard (almost all .NET projects can do this).

To add ML.NET to a project, go to NuGet Package Manager and install the latest version of `Microsoft.ML`. I also recommend you install `Microsoft.ML.AutoML` since many examples here feature that code and AutoML is a good way to get started with ML.NET.



See **Microsoft's documentation on NuGet Package Manager** for more details on working with NuGet Package Manager.

Tasks Supporting AutoML

First, I'm going to highlight the five machine learning tasks in ML.NET that are supported using AutoML. These are easier tasks to get into due to their support for AutoML and so I will supply *some* code for each type of task. Supplying full code and details for each one is something best done in a separate article, so if you have interest in more detail into a specific task, **please ask** and I'll be happy to provide additional content as I can.

Binary Classification

Binary Classification tasks involve predicting a categorical label that something should be assigned to given a set of related features. For example, given some characteristics about a loan applicant, a binary classification model would predict whether or not that loan should be approved or rejected.

Binary Classification tasks are limited to predicting a single column that has two possible values. If there are more than two possible values, that is a *multi-class classification* task which we'll discuss next.

Code to run a binary classification experiment using AutoML might look something like the following:

```
public ITransformer PerformBinaryClassification(IDataView trainingData, IDataView validationData)
{
    // Set up the experiment
    MLContext context = new MLContext();
    uint maxSeconds = 10;
    BinaryClassificationExperiment experiment = context.Auto().CreateBinaryClassificationExperiment(maxSeconds);

    // Run the experiment and wait synchronously for it to complete
    ExperimentResult<BinaryClassificationMetrics> result =
        experiment.Execute(trainingData, validationData, labelColumnName: "ShouldApproveLoan");

    // result.BestRun.ValidationMetrics has properties helpful for evaluating model performance
    double accuracy = result.BestRun.ValidationMetrics.Accuracy;
    double f1Score = result.BestRun.ValidationMetrics.F1Score;
    string confusionTable = result.BestRun.ValidationMetrics.ConfusionMatrix.GetFormattedConfusionTable();

    // Return the best performing trained model
    ITransformer bestModel = result.BestRun.Model;
    return bestModel;
}
```

You could then use that trained model to make a prediction via the following code:

```
public LoanPrediction PredictBinaryClassification(ITransformer bestModel, IDataView trainingData, LoanData loan)
{
    MLContext context = new MLContext();

    // Create an engine capable of evaluating one or more loans in the future
    PredictionEngine<LoanData, LoanPrediction> engine =
        context.Model.CreatePredictionEngine<LoanData, LoanPrediction>(bestModel, trainingData.Schema);

    // Actually make the prediction and return the findings
    LoanPrediction prediction = engine.Predict(loan);
    return prediction;
}
```

Here `LoanData` and `LoanPrediction` are classes representing a row in your dataset and the final prediction of your algorithm, respectively.

Note: ML.NET also supports classification without using AutoML, but for brevity, that code is omitted from this article

Multi-Class Classification

Multi-Class classification tasks are very similar to binary classification tasks in that you are trying to predict a categorical value for a single labelled column given a set of features. The key difference between a binary classification problem and a multi-class classification problem is that with a binary classification problem there are only two possible values something might have while in a multi-class classification problem there are three or more possible categories something might fall into.

Code for training a multi-class classification experiment using AutoML might look like the following:

```
public ITransformer PerformMultiClassification(IDataView trainingData, IDataView validationData)
{
    // Set up the experiment
    MLContext context = new MLContext();
    uint maxSeconds = 10;
    MulticlassClassificationExperiment experiment = context.Auto().CreateMulticlassClassificationExperiment(maxSec

    // Run the experiment and wait synchronously for it to complete
    ExperimentResult<MulticlassClassificationMetrics> result =
        experiment.Execute(trainingData, validationData, labelColumnName: "RiskCategory");

    // result.BestRun.ValidationMetrics has properties helpful for evaluating model performance
    string confusionTable = result.BestRun.ValidationMetrics.ConfusionMatrix.GetFormattedConfusionTable();

    // Return the best performing trained model
    ITransformer bestModel = result.BestRun.Model;
    return bestModel;
}
```

Beyond that, the code for working with trained classification models is remarkably similar to the code for working with binary classification models. Also like the binary classification models, it is possible to work with multi-class classification models without using AutoML.

For more code examples around multi-class classification, take a look at my article and video on **using AutoML and ML.NET to predict ESRB ratings for video games** which also serves as an in-depth introduction to ML.NET for a single task.

Regression

Regression tasks involve predicting a numerical value given a set of features. For example, you could use a regression model to predict the price of gas given a known set of other factors or use regression to predict the length of time you might need to spend defrosting your car

in the morning given overnight weather factors.

Any time you need to calculate a single numerical value, you're likely dealing with a regression problem.

The code to perform model training for your regression experiment is similar to that of classification experiments:

```
public ITransformer PerformRegression(IDataView trainingData, IDataView validationData)
{
    // Set up the experiment
    MLContext context = new MLContext();
    uint maxSeconds = 10;
    RegressionExperiment experiment = context.Auto().CreateRegressionExperiment(maxSeconds);

    // Run the experiment and wait synchronously for it to complete
    ExperimentResult<RegressionMetrics> result =
        experiment.Execute(trainingData, validationData, labelColumnName: "Temperature");

    // result.BestRun.ValidationMetrics has properties helpful for evaluating model performance
    double error = result.BestRun.ValidationMetrics.MeanAbsoluteError;

    // Return the best performing trained model
    ITransformer bestModel = result.BestRun.Model;
    return bestModel;
}
```

Note that the validation metrics for regression experiments are completely different than the validation metrics for classification experiments. While classification experiments deal with the probability that something is given the correct category, regression experiments deal with the distance between the predicted numerical value and the actual numerical value for known historical data.

Like both classification model types you don't need to use AutoML when training a regression model, but it can be helpful if your knowledge of individual algorithms is limited.

Recommendation

A recommendation algorithm is a variant of a regression algorithm. With a recommendation algorithm you feed in data about different types of users and different ratings they've given items in the past. Given such a dataset, a recommendation model can predict what rating a user would give to something they've not yet interacted with before based on their similarity to the tastes of other known users.

Recommendation models are popular in movie, music, and product recommendation systems where repeat users are common and everyone benefits from users finding the content they'll like the most.

Recommendation is supported by AutoML and the code for recommendation is very similar to regression code:

```
public ITransformer PerformRecommendation(IDataView trainingData, IDataView validationData)
{
    // Set up the experiment
    MLContext context = new MLContext();
    uint maxSeconds = 10;
    RecommendationExperiment experiment = context.Auto().CreateRecommendationExperiment(maxSeconds);

    // Run the experiment and wait synchronously for it to complete
    ExperimentResult<RegressionMetrics> result =
        experiment.Execute(trainingData, validationData, labelColumnName: "Rating");

    // result.BestRun.ValidationMetrics has properties helpful for evaluating model performance
    double error = result.BestRun.ValidationMetrics.MeanAbsoluteError;

    // Return the best performing trained model
    ITransformer bestModel = result.BestRun.Model;
    return bestModel;
}
```

Under the hood the recommendation algorithms use matrix factorization, which is a more complicated topic. See **Microsoft's tutorial on matrix factorization** for more details on recommendation systems without using AutoML. There's also a fantastic **article from Rubik's Code** exploring the topic in additional depth.

Ranking

Ranking is similar to a recommendation algorithm, but is used to put items into a force-order rank suitable for displaying search results. Ranking systems are suitable for showing a list of ordered recommendations for a specific user or group of users.

The code is similar to the code we've seen before, though the validation metrics are significantly different:

```
public ITransformer PerformRanking(IDataView trainingData, IDataView validationData)
{
    // Set up the experiment
    MLContext context = new MLContext();
    uint maxSeconds = 10;
    RankingExperiment experiment = context.Auto().CreateRankingExperiment(maxSeconds);

    // Run the experiment and wait synchronously for it to complete
    ExperimentResult<RankingMetrics> result =
        experiment.Execute(trainingData, validationData, labelColumnName: "Temperature");

    // result.BestRun.ValidationMetrics has properties helpful for evaluating model performance
    IEnumerable<double> gains = result.BestRun.ValidationMetrics.DiscountedCumulativeGains;
    IEnumerable<double> normalizedGains = result.BestRun.ValidationMetrics.NormalizedDiscountedCumulativeGains;

    // Return the best performing trained model
    ITransformer bestModel = result.BestRun.Model;

    RankingEvaluatorOptions options = new RankingEvaluatorOptions();
    RankingMetrics metrics = context.Ranking.Evaluate(trainingData, labelColumnName: "Label", rowGroupColumnName:
```

```
    return bestModel;
}
```

Other Solution Types

Now let's take a high-level look at each of the five machine learning tasks not currently supported through AutoML. Code for each one of these is going to be more varied than this article can afford so reach out and ask for more content on the ones that interest you the most.

Forecasting Time Series Data

Forecasting involves predicting a batch of future regression values based on historical data. When you are forecasting you are predicting values from some window into the future where each value predicted has a certain level of confidence level.

This works in a similar way to how a weather forecast might work. Weather forecasts are most accurate with a lot of relevant historical data when predicting values in the near future. They can be used to predict values some time in the future, but the accuracy of those predictions goes down significantly as the time range goes on.

Clustering

Clustering is used to group various data points together into groups based on similarities to nearby data points. This can be used to determine what customers are similar to each other for marketing, grouping for recommendations, or other purposes. When working with geographical data this can also be a great way of determining optimal locations for office placements or cell towers.

Clustering typically works by choosing an arbitrary number of clusters and allowing machine learning to follow a K-Means clustering algorithm to optimize the central location of each cluster in order to minimize the overall distance from each data point to the center of its cluster. Clustering algorithms also tend to try to space out clusters from each other when possible.

Anomaly Detection

Anomaly detection can be used to flag individual transactions as unusual for additional investigation. Anomaly detection is often used for virus detection, credit card fraud detection, and identifying unusual network activity. You can think of anomaly detection almost like an

automated form of binary classification where something is either going to be normal or anomalous.

Image Classification

Image classification is similar to binary or multi-class classification but instead of working on numerical features it works on an image to determine what is featured in a given image. Like classification problems, you must provide ML.NET with a wide variety of labelled images in different sizes, lighting, and arrangements featuring the things you are trying to detect in order for it to reliably classify images.

Object Detection

Object detection is like image classification, but instead of telling you that an image is of a specific class, object detection gives you an actual bounding box in the image telling you where that specific object is located. Additionally, object detection is capable of locating multiple objects in a single image, which exceeds the limitations of image classification.

Object detection is a part of **Azure Cognitive Services** that is only available in ML.NET via the Model Builder at the time of this writing.

Conclusion

Hopefully this article helps clarify some of the things that ML.NET can achieve - either with AutoML or without it.

Watch this blog (and **my YouTube channel**) for more content on ML.NET in the future and let me know what you're most curious about learning.

In the meantime, I recommend looking into **Microsoft's documentation on ML.NET** for additional details or checking out their **ML.NET samples on GitHub**

This post was featured as part of the C# Advent 2021 event. Be sure to check out other posts by fantastic members of the .NET community

---

If you found this content helpful, **let me know on Twitter** or **send me an message**. You can also get regular video content by **subscribing on YouTube**.

Matt on Data Science

YouTube 123

ML.NET CSharp Concepts Machine Learning Data Science

About Matt Eland

Matt is a software engineering instructor who raises up future C# developers by day and performs data science experiments by night. Prior to teaching Matt enjoyed a .NET and JavaScript-heavy career in the startup and Software as a Service world where he helped companies grow to maturity and then scale up. Matt is currently pursuing a master's degree studying data science.

« Previous

Installing Anaconda for Python Development
