ML.NET AUTOML

Weiter lernt die Maschine

Microsoft hat AutoML erweitert. Die strategische Ausrichtung ist indes nicht ganz klar.

Mit ML.NET AutoML stellt Microsoft eine Klassenbibliothek zum automatischen Trainieren von Machine-Learning-Modellen zur Verfügung [1][2]. Die AutoML-Logik erledigt typische Arbeitsschritte vor und während des Trainings, etwa die Auswahl des Trainingsalgorithmus. Sie probiert mehrere Algorithmen aus, sodass sie dann das beste Modell präsentieren kann. Ende 2019 hat die dotnetpro die AutoML-Version 0.14.0 vorgestellt und gezeigt, wie sich die Automatisierung noch weitertreiben lässt, indem Sie zusätzlich mehrere Optimierungsmetriken durchlaufen [3].

Inzwischen ist Version 0.20.0 (Preview) aktuell und sie besitzt eine Reihe an Neuerungen [4]. Neben weiteren Algorithmen und Metriken gibt es einige Datenvorbereitungsschritte und man kann Suchräume für Hyperparameter vorgeben. Um die Beispiele nachzuvollziehen, sind die Pakete Microsoft.ML und Microsoft.ML.AutoML per NuGet aus dem ML.NET daily feed zu installieren [5]. Nötig ist darüber hinaus der Namensraum *System.Text.Json* 6.0.0.

Neue Algorithmen und Metriken

Neben allgemeinen Verbesserungen an vorhandenen Optimierungsmetriken und an den Timeouts von Trainingsdurchläufen (Experimenten) gibt es nun eigene Klassen für die Aufgabentypen „Empfehlung“ (Recommendation) und „Priorisierung“ (Ranking). Beide bieten spezielle Trainingsalgorithmen für den jeweiligen Aufgabentyp: *MatrixFactorization* (Empfehlung), *FastTreeRanking* und *LightGbmRanking* (Priorisierung). Außerdem gibt es für den Typ Priorisierung neue Optimierungsmetriken (*Dcg* und *Ndcg*). Für den Typ Empfehlung sucht man eigene Metriken vergeblich. Stattdessen müssen Sie die Metriken des Regression-Typs verwenden. Tabelle 1 fasst den aktuellen Stand zusammen. Ein einfaches Beispiel für ein Experiment ist in Listing 1 zu sehen; dort wird ein Modell trainiert, das eine Prognose für das Vorliegen von bösartigem Brustkrebs abgibt (binäre Klassifikation). Genauere Details finden Sie unter [3].

Datenvorbereitung

Wichtig vor jedem Training eines Modells ist die Aufbereitung der Eingangsdaten. AutoML übernimmt nun einige typische Schritte der Datenvorverarbeitung:

- Features (Merkmale), die für das Training nutzlos sind, werden ignoriert. Das betrifft Features, bei denen alle Werte leer oder identisch sind, aber auch Features mit hoher Kardinalität (zum Beispiel ID-Werte).

Tabelle 1: Trainingsalgorithmen und Optimierungsmetriken von ML.NET AutoML

Aufgabentyp	Trainingsalgorithmen	Optimierungsmetriken
Binäre Klassifikation	*AveragedPerceptronBinary* *FastForestBinary* *FastTreeBinary* *LightGbmBinary* *LinearSvmBinary* *LbfgsLogisticRegressionBinary* *SdcaLogisticRegressionBinary* *SgdCalibratedBinary* *SymbolicSgdLogisticRegressionBinary*	*Accuracy* *AreaUnderRocCurve* *AreaUnderPrecisionRecallCurve* *F1Score* *PositivePrecision* *PositiveRecall* *NegativePrecision* *NegativeRecall*
Mehrklassen-Klassifikation	*AveragedPerceptronOva* *FastForestOva* *FastTreeOva* *LightGbmMulti* *LinearSvmOva* *LbfgsMaximumEntropyMulti* *LbfgsLogisticRegressionOva* *SdcaMaximumEntropyMulti* *SgdCalibratedOva* *SymbolicSgdLogisticRegressionOva*	*MicroAccuracy* *MacroAccuracy* *LogLoss* *LogLossReduction* *TopKAccuracy*
Regression	*FastForestRegression* *FastTreeRegression* *FastTreeTweedieRegression* *LightGbmRegression* *OnlineGradientDescentRegression* *OlsRegression* *LbfgsPoissonRegression* *SdcaRegression*	*MeanAbsoluteError* *MeanSquaredError* *RootMeanSquaredError* *RSquared*
Empfehlung	*MatrixFactorization*	*MeanAbsoluteError* *MeanSquaredError* *RootMeanSquaredError* *RSquared*
Priorisierung	*LightGbmRanking* *FastTreeRanking*	*Dcg (Discounted Cumulative Gains)* *Ndcg (Normalized Discounted Cumulative Gains)*

Listing 1: Ein einfaches AutoML-Experiment

```
using Microsoft.ML;
using Microsoft.ML.AutoML;
using System;

namespace AutoMLExample2 {
  public static class Program {
    public static void Main()   {
      // Initialize context
      var mlContext =
        new MLContext(24); // Arbitrary seed

      // Load data from csv file
      var trainDataView = mlContext.Data.
        LoadFromTextFile<BiopsyData>(
        @"..\..\..\Biopsy\BiopsyTrainData.csv", ';',
        true);

      // Create experiment settings
      var experimentSettings =
        new BinaryExperimentSettings
        {
          MaxExperimentTimeInSeconds = 60,
          OptimizingMetric =
            BinaryClassificationMetric.Accuracy
        };

      // Create experiment
      var experimentA = mlContext.Auto().
        CreateBinaryClassificationExperiment(
        experimentSettings);

      // Run experiment and print model accuracy
      var experimentResultA = experimentA.Execute(
        trainDataView, "class");
      Console.WriteLine($"(A) Accuracy = {experiment"
        + $"ResultA.BestRun.ValidationMetrics."
        + $"Accuracy}");
    }
  }
}
```

Bag-of-Words-Modell/N-Gramme

Das Bag-of-Words-Modell ist eine vereinfachende Darstellung, die in der Verarbeitung natürlicher Sprache Verwendung findet. Dabei wird ein Text als eine Menge (Bag) von Wörtern dargestellt. Grammatik und Wortreihenfolge werden außer Acht gelassen; Wörter können aber mehrfach enthalten sein.

Ein verwandtes Verfahren ist die Zerlegung eines Textes in N-Gramme, also Textfragmente, die aus N Einzelteilen (Buchstaben, Phoneme, Wörter) bestehen. Hierbei bleiben Informationen über die Reihenfolge bestehen.

Beim Machine Learning dienen Modelle und N-Gramme zur Klassifizierung von Texten. Dabei werden Auftreten, Häufigkeit und Anordnung von Wörtern und anderen Textfragmenten als Feature für das Training eines ML-Modells genutzt.

One-Hot-Codierung

Bei der One-Hot-Codierung wird eine Zahl durch n Bits dargestellt, wobei jeweils ein Bit auf 1 gesetzt ist, während die restlichen Bits 0 sind. Beim Machine Learning wird das Format verwendet, um die Zugehörigkeit zu einer von mehreren Kategorien darzustellen. In dieser Darstellung lassen sich Modelle besser optimieren.

- **Beispiel:** Farbe = "rot" wird umgewandelt in Farbe blau = 0, Farbe rot = 1, Farbe grün = 0.

- Einzelne fehlende Werte werden durch den Standardwert des entsprechenden Datentyps ersetzt und es wird ein Feature hinzugefügt, das anzeigt, ob ein Wert gefehlt hat.
- Textwerte mit niedriger Kardinalität werden in One-Hot-Codierung überführt (siehe Kasten One-Hot-Codierung).
- Aus Texten wird ein Bag-of-Words-Modell mit sogenannten N-Grammen aus Buchstaben und Wörtern erzeugt. Diese lassen sich für die Verarbeitung natürlicher Sprachen einsetzen (siehe Kasten Bag-of-Words-Modell/N-Gramme).
- Numerische Werte werden normalisiert.

Andere wichtige Vorbereitungsschritte und Konfigurationsmöglichkeiten fehlen allerdings:

- Fehlende numerische Werte können nicht durch statistische Größen ersetzt werden (beispielsweise Durchschnittswert oder Maximalwert)
- Es werden keine Features aus Datumsangaben extrahiert (etwa Monat, Wochentag)
- Die Methode der Normalisierung lässt sich nicht wählen.

Laut Ankündigung von Microsoft sollen diese Limitierungen in Zukunft zu umgehen sein, und zwar mit einer Verarbeitungspipeline, die mithilfe der *TransformsCatalog*-Klasse (über *MLContext.Transforms*) erzeugt wurde. Das AutoML-Experiment muss dann über die neue Methode *CreateExperiment()* erzeugt und die Pipeline über die *SetPipeline()*-Methode gesetzt werden. Das Experiment lässt sich anschließend über *Run()* (auch asynchron) starten.

Listing 2 zeigt, wie es geht. In der Pipeline werden dort zunächst neun numerische Features (V1 bis V9) einem Label (*class*) zugeordnet und deren Werte anschließend explizit ▶

Listing 2: Eine über MLContext.Transform erzeugte Verarbeitungspipeline

```
using Microsoft.ML;
using Microsoft.ML.AutoML;
using System;

namespace AutoMLExample2 {
  public static class Program {
    public static void Main() {

      var mlContext = new MLContext(24);
        // Arbitrary seed
      var trainDataView = mlContext.Data.
        LoadFromTextFile<BiopsyData>(
        @"..\..\..\Biopsy\BiopsyTrainData.csv", ';',
        true);

      // Create data processing pipeline to normalize
      var preProcessingPipeline = mlContext.Transforms
        // Combine feature columns to single vector
        .Concatenate("Features", new string[] { "V1",
        "V2", "V3", "V4", "V5", "V6", "V7", "V8",
        "V9" })
        // Normalize values using min/max method
        .Append(mlContext.Transforms.NormalizeMinMax(
        "Features"));

      // Append pipeline step for the experiment
      var experimentPipeline = preProcessingPipeline.
        Append(mlContext.Auto().BinaryClassification(
        "class", "Features"));

      // Configure experiment
      var experimentB =
        mlContext.Auto().CreateExperiment()
        .SetPipeline(experimentPipeline)
        .SetDataset(trainDataView).SetEvaluateMetric(
        BinaryClassificationMetric.Accuracy, "class")
        .SetTrainingTimeInSeconds(60);

      // Run experiment and print model accuracy
      var experimentResultB = experimentB.Run();
        // Can also run asynchronously in an async
        // method
      Console.WriteLine(
        $"(B) Accuracy = {experimentResultB.Result."
        + $"Metric}");
    }
  }
}
```

über einen Min/Max-Algorithmus normalisiert. An die Pipeline lassen sich auch weitere Schritte anhängen, zum Beispiel mithilfe der *ReplaceMissingValues()*-Methode, über die man festlegen kann, wie fehlende Werte in den Eingangsdaten ersetzt werden sollen. Die Dokumentation der *TransformsCatalog*-Klasse vermittelt ein Gefühl dafür, was möglich ist [6]. In der Version 0.20.0-preview.22273 haben allerdings noch nicht alle Methoden funktioniert.

Suchraum von Hyperparametern

Hyperparameter sind Parameter, welche die Architektur des ML-Modells beeinflussen (etwa maximale Knotentiefe in einem Entscheidungsbaum) oder den Trainingsprozess steuern (zum Beispiel Lernrate). Sie beeinflussen wesentlich den Trainingserfolg und die Geschwindigkeit des Trainings.

Jeder einzelne Parameter kann viele verschiedene Werte annehmen. Die Menge aller möglichen Kombinationen an Werten ist der Suchraum. Das Ziel ist, die beste Kombination an Werten über alle Parameter hinweg zu finden. Um Ressourcen zu sparen, sollte der Suchraum dabei so klein wie möglich gehalten werden. Wenn also schon vor dem Training bekannt ist, dass bestimmte Hyperparameterwerte ungeeignet sind, ist es sinnvoll, den Suchraum entsprechend einzuschränken.

Anpassung der Hyperparameter

Was aber schon funktioniert, ist die Anpassung des Suchraums der Hyperparameter (siehe Kasten Suchraum von Hypermarametern). Hierzu verwenden Sie die Methode *CreateSweepableEstimator()*. Der Name basiert auf dem Sweep-Verfahren, einem Vorgehen zum Lösen von Problemen der algorithmischen Geometrie. Dabei übergeben Sie ein *SearchSpace*-Objekt sowie eine Lambda-Funktion, in der die Suchraumparameter gesetzt werden. Die so erzeugte Sweeping-Estimator-Pipeline wird dann über *SetPipeline()* verwendet. Listing 3 zeigt dies am Beispiel des Fast-Forest-Algorithmus.

Die Flexibilität, die sich dadurch ergibt, bedeutet allerdings auch mehr Arbeit bei der Konfiguration des Experiments. Man muss nämlich die Konfiguration für jeden Trainingsalgorithmus einzeln festlegen; automatisiert wird an dieser Stelle also nichts. Das müssen Sie mit eigenem Code erledigen, beispielsweise durch entsprechende Schleifen.

Fazit

AutoML bietet neue Algorithmen und Metriken und passt sich nun besser in die Landschaft rund um den *Microsoft.ML*-Namensraum ein. Die automatische Auswahl des Trainingsalgorithmus ist nur einer von mehreren Aspekten; über Pipelines lassen sich passende Verarbeitungsschritte einhängen und Suchräume für Hyperparameter definieren.

Allerdings fühlt sich das alles noch etwas umständlich an. Es wäre wünschenswert, dass AutoML weitere Schritte auto-

Listing 3: Beispiel zum Fast-Forest-Algorithmus

```
...

using Microsoft.ML.AutoML.CodeGen;
using Microsoft.ML.SearchSpace;
using Microsoft.ML.Trainers.FastTree;

...

// Create search space
var searchSpace =
  new SearchSpace<FastForestOption>();

// Initialize sweeping estimator pipeline
var sweepingEstimatorPipeline =
    preProcessingPipeline
    .Append(mlContext.Auto().
    CreateSweepableEstimator((context, param) =>
  {
    var option =
      new FastForestBinaryTrainer.Options()
    {
      NumberOfLeaves = 5, // Maximum number of
                          // leaves per tree
      NumberOfTrees = 5, // Total number of trees
      FeatureFraction = 0.9, // Use only 90% of
                             // features (helps
                             // reducing overfitting)
      LabelColumnName = "class",
      FeatureColumnName = "Features"
    };

    return context.BinaryClassification.
      Trainers.FastForest(option);
  }, searchSpace));

// Configure experiment
var experimentB =
  mlContext.Auto().CreateExperiment()
  .SetPipeline(sweepingEstimatorPipeline)
  .SetDataset(trainDataView)
  .SetEvaluateMetric(
    BinaryClassificationMetric.Accuracy, "class")
  .SetTrainingTimeInSeconds(60);

...
```

matisiert und eine einfache Konfiguration ermöglicht. Derzeit muss noch einiges an Code geschrieben werden, bis eine umfassende Automation steht. Dazu kommt, dass einige ML-Aufgabentypen in AutoML gar nicht zur Verfügung stehen (Clustering, Anomalie-Erkennung, Dimensionsreduktion, Bilderkennung). Das ist wohl auch der Grund, warum es nach mehreren Jahren Entwicklungszeit noch immer keine Version 1.0 gibt. Hier stellt sich die Frage, welchen Stellenwert AutoML als Klassenbibliothek eigentlich hat.

Es scheint, dass Microsoft den Fokus aktuell nicht auf klassische .NET-Entwickler legt, sondern auf Anwender von Tools. So wurden AutoML-Funktionen in den ML.NET-Modell-Generator [7], in ML.NET CLI [8] und in Azure Machine Learning [9] eingebaut und dort auch schneller weiterentwickelt als in der reinen AutoML-Bibliothek. Auffällig ist auch, dass ML.NET nicht Teil der Microsoft-eigenen Zertifizierung zum Data Scientist Associate ist [10]. Von diesem wird stattdessen erwartet, mit Azure Machine Learning, Python und Tools wie Jupyter Notebooks zu arbeiten. Auch zwei wichtige Projekte von Microsoft Research – Neural Network Intelligence [11] und FLAML [12] – arbeiten mit Python.

Dennoch – wenn man bereit ist, die Automation mit eigenem Code weiter in die Hand zu nehmen, bietet AutoML einige interessante Möglichkeiten. Im Übrigen arbeitet Microsoft an einer .NET-Bibliothek zur Anbindung von PyTorch (TorchSharp [13]), was in Zukunft zusätzliche Wege für .NET-Entwickler eröffnen könnte. ■

[1] How to use the ML.NET automated machine learning API, www.dotnetpro.de/SL2209AutoML1
[2] Microsoft.ML.AutoML Namespace, www.dotnetpro.de/SL2209AutoML2
[3] Martin Gossen, Selbst lernt die Maschine, dotnetpro 12/2019, Seite 74 ff., www.dotnetpro.de/A1912AutoML
[4] What's new with ML.NET Automated ML (AutoML) and tooling, www.dotnetpro.de/SL2209AutoML3
[5] ML.NET daily feed, www.dotnetpro.de/SL2209AutoML4
[6] TransformsCatalog Class, www.dotnetpro.de/SL2209AutoML5
[7] What is Model Builder and how does it work?, www.dotnetpro.de/SL2209AutoML6
[8] Automate model training with the ML.NET CLI, www.dotnetpro.de/SL2209AutoML7
[9] Data featurization in automated machine learning (Python SDK azureml), www.dotnetpro.de/SL2209AutoML8
[10] Microsoft Certified: Azure Data Scientist Associate, www.dotnetpro.de/SL2209AutoML9
[11] Neural Network Intelligence, www.dotnetpro.de/SL2209AutoML10
[12] FLAML: A Fast and Lightweight AutoML Library, www.dotnetpro.de/SL2209AutoML11
[13] TorchSharp, www.dotnetpro.de/SL2209AutoML12

Martin Gossen
ist IT-Berater bei der IKS GmbH in Hilden. Er erstellt seit 15 Jahren Softwarelösungen auf Basis von C#, .NET und Microsoft SQL Server. Sie erreichen ihn unter **m.gossen@iks-gmbh.com**.

dnpCode A2209AutoML