ML.NET AUTOML

Selbst lernt die Maschine

Microsofts neue Erweiterung für das ML.NET-Framework vereinfacht das Trainieren von Machine-Learning-Modellen.

Das typische Vorgehen bei der Erstellung von Softwarelösungen mittels Machine Learning (ML) besteht aus folgenden Schritten: Daten importieren, Daten aufbereiten, Trainingsalgorithmus auswählen, Hyperparameter und Optimierungsmetrik festlegen, Modell trainieren, Modell bewerten – und fertig ist die „künstliche Intelligenz".

In der praktischen Umsetzung trifft man dabei allerdings auf diverse Herausforderungen. Allein schon die Auswahl des besten Algorithmus ist nicht trivial, da von diesen mehrere Dutzend zur Verfügung stehen [1]. Kombiniert mit einer Reihe an Hyperparametern (siehe Kasten Hyperparameter) und mehreren Optimierungsmetriken (siehe Kasten Optimierungsmetriken) ergeben sich unzählige Möglichkeiten für die Durchführung des Trainings.

Entsprechend schwierig ist es, die beste Kombination aus allen Aspekten für das zu lösende Problem zu finden. Es können zwar einige Faustregeln angewandt werden, aber selbst erfahrene Datenwissenschaftler müssen oftmals viele Kombinationen austesten, bevor sie mit gutem Gewissen sagen können, dass sie die beste oder zumindest eine sehr gute Lösung gefunden haben.

An dieser Stelle treten „Auto"-Lösungen auf den Plan. Diese Tools sollen einen oder mehrere Schritte bei der Erstellung von ML-Modellen übernehmen. Damit wird ML auch für Entwickler zugänglich, die nur wenig Expertise im Bereich Data Science mitbringen.

Einige namhafte Tools sind derzeit das kostenpflichtige Cloud AutoML von Google [2], die Open-Source-Bibliothek Auto-Keras [3] und seit Kurzem AutoML von Microsoft [4]. Letzteres ist Teil des ML.NET-Frameworks [5]. Es befindet sich noch in einem frühen Entwicklungsstadium, kann aber – wie nachfolgend gezeigt – bereits eingesetzt werden.

Funktionsweise

ML.NET AutoML automatisiert das Durchtesten verschiedener Trainingsalgorithmen mit unterschiedlichen Werten für die jeweiligen Hyperparameter. Man setzt sich als Entwickler also kaum noch mit den Interna der Modelle und des Trainingsvorgangs auseinander, sondern legt nur noch einige allgemeine Parameter fest – etwa ein Ausstiegskriterium (üblicherweise eine Trainingsdauer) oder eine Angabe zum Caching. Man kann auch einzelne Algorithmen vom Training ausschließen, zum Beispiel wenn bekannt ist, dass ein Algorithmus sehr langsam ist oder nur ungenaue Ergebnisse für die vorliegende Problemstellung liefern würde.

Sind die Trainingsdurchläufe beendet, greift man sich das geeignetste Modell heraus und persistiert es. So lässt sich das trainierte Modell jederzeit laden und einsetzen.

ML.NET AutoML unterstützt binäre Klassifikation, Mehrklassen-Klassifikation sowie Regression. Die unterstützten Trainingsalgorithmen zeigt Tabelle 1, während Tabelle 2 die verfügbaren Optimierungsmetriken im Überblick darstellt. Details zu den einzelnen Algorithmen und Metriken finden Sie unter [6] bis [11]. Zu beachten ist, dass AutoML in der aktuellen Vorabversion 0.14.0 noch keine echten neuronalen Netzwerke unterstützt, sondern nur das Averaged Perceptron, einen sehr einfachen Vorläufer von neuronalen Netzen.

Installation und Training

Zur Nutzung von ML.NET AutoML müssen Sie lediglich die Bibliotheken Microsoft.ML und Microsoft.ML.AutoML installieren, zum Beispiel mittels NuGet in Visual Studio. Greifen

Optimierungsmetriken

Eine Optimierungsmetrik ist eine mathematische Funktion, mit welcher der Erfolg eines ML-Modells während der Trainingsphase gemessen wird. Je nach Problemtyp werden unterschiedliche Metriken eingesetzt; so etwa die mittlere Standardabweichung (Mean Absolute Error) bei der Regression oder der relative Anteil korrekt durchgeführter Klassifizierungen (Accuracy) bei der binären Klassifikation. Eine Einführung in das Thema finden Sie unter [14].

Hyperparameter

Hyperparameter (auch Metaparameter) sind Parameter, die den Trainingsprozess eines ML-Modells steuern und deren Werte festgelegt werden, bevor das Training beginnt. Hyperparameter stehen damit im Kontrast zu Parametern, deren Werte während des laufenden Trainingsprozesses gesetzt und optimiert werden. Welche Hyperparameter zur Anwendung kommen, hängt vom eingesetzten Trainingsalgorithmus ab; beispielsweise wird bei Regression der Polynomgrad gesetzt oder bei neuronalen Netzen die Anzahl der Knoten und der versteckten Schichten. Unter [15] finden Sie einen Artikel zur Optimierung von Hyperparametern.

Sie ruhig zur neuesten Version, auch wenn sie noch als instabil gekennzeichnet ist. In der Tat treten bei einzelnen Trainingsdurchläufen gelegentlich Exceptions auf, was aber verschmerzbar ist – durch Abfangen der Exception geht nur das Ergebnis des aktuellen Durchlaufs verloren.

Zur Demonstration der Vorgehensweise dient das Beispiel „Brustkrebsdiagnose", das auch schon in [12] beschrieben und dort mit R Services implementiert wurde. Bei diesem Beispiel liegen Gewebebiopsie-Daten aus der Untersuchung des Tumorgewebes von Brustkrebspatientinnen vor (Tabelle 3).

Tabelle 1: Trainingsalgorithmen in ML.NET AutoML

Trainingsalgorithmus	Binäre Klassifikation [6]	Mehrklassen-Klassifikation [7]	Regression [8]
AveragedPerceptron	X		
AveragedPerceptronOVA		X	
FastForest	X		X
FastForestOVA		X	
FastTree	X		X
FastTreeOVA		X	
FastTreeTweedie			X
LbfgsLogisticRegression	X		
LbfgsLogisticRegressionOVA		X	
LbfgsMaximumEntropy		X	
LbfgsPoissonRegression			X
LightGbm	X	X	X
LinearSupportVectorMachinesOVA		X	
LinearSvm	X		
Ols			X
OnlineGradientDescent			X
SdcaLogisticRegression	X		
SdcaMaximumEntropy		X	
SgdCalibrated	X		
SgdCalibratedOVA		X	
StochasticDualCoordinateAscent			X
SymbolicSgdLogisticRegression	X		
SymbolicSgdLogisticRegressionOVA		X	

Basierend auf diesen Eingangswerten soll das trainierte Modell eine Entscheidung fällen, ob bei einer Person bösartiger Brustkrebs vorliegt oder nicht (binäre Klassifikation).

Listing 1 zeigt anhand einer Konsolenanwendung, wie es geht. Zuerst müssen Sie als Basis ein *MLContext*-Objekt erzeugen. Hier können Sie optional einen Seed (Startwert) übergeben, der sicherstellt, dass die Ergebnisse reproduzierbar sind. Danach können Sie die Trainingsdaten einlesen, zum Beispiel aus einer CSV-Datei. Dabei erfolgt eine Typisierung auf die Eingangsdaten, wie sie in der Klasse *BiopsyData* vorgegeben sind (Listing 2). Jedes Feld muss dabei mit dem *LoadColumn*-Attribut versehen werden. Erlaubte Datentypen innerhalb von *BiopsyData* sind *string*, *float* und *bool*, da AutoML nur mit diesen umgehen kann. Die Eingangsdaten sollten entsprechend angepasst werden (*true/false* statt *malignant/benign*).

Im Anschluss erstellen Sie ein *BinaryExperimentSettings*-Objekt, mit dem das Training konfiguriert wird (Microsoft nennt das Training ein „Experiment"). Einer der Konfigurationswerte ist die zu verwendende Optimierungsmetrik. Um die Automatisierung noch einen Schritt weiter zu treiben, setzen Sie eine Schleife um den darauffolgenden Trainingscode und iterieren dabei durch alle verfügbaren Metriken.

Für das eigentliche Training wird ein *BinaryClassificationExperiment*-Objekt benötigt, das Sie aus *mlContext* erzeugen können. Durch Aufruf der *Execute*-Methode wird ▶

Tabelle 2: Optimierungsmetriken für ML.NET AutoML

Optimierungsmetrik	Binäre Klassifikation [9]	Mehrklassen-Klassifikation [10]	Regression [11]
Accuracy	X		
AreaUnderPrecisionRecallCurve	X		
AreaUnderRocCurve	X		
F1Score	X		
LogLoss		X	
LogLossReduction		X	
MacroAccuracy		X	
MeanAbsoluteError			X
MeanSquaredError			X
MicroAccuracy		X	
NegativePrecision	X		
NegativeRecall	X		
PositivePrecision	X		
PositiveRecall	X		
RootMeanSquaredError			X
RSquared			X
TopKAccuracy		X	

das Training angestoßen. Legen Sie nach jedem Durchlauf das trainierte Modell mitsamt seiner Konfiguration und den ermittelten Messgrößen (Listing 3) in einer Liste ab. Am Ende picken Sie sich das geeignetste Modell durch Sortieren der Liste heraus. Im vorliegenden Beispiel wird zuerst nach negativer Genauigkeit (*NegativePrecision*), dann nach positiver Genauigkeit (*PositivePrecision*) sortiert, weil es wichtig ist, wenige falsch negative Ergebnisse zu bekommen (ein übersehener bösartiger Tumor ist viel problematischer als einige irrtümlich als bösartig erkannte gutartige Tumoren).

Im letzten Schritt persistieren Sie das Gewinnermodell in eine Binärdatei und geben einige Kenngrößen in der Konsole aus. Besonders übersichtlich ist dabei die sogenannte Confusion Matrix, die mehrere Größen tabellenartig wiedergibt [13]. Ein typisches Ergebnis sehen Sie in Bild 1.

Beurteilung

Führt man das Experiment mehrmals durch, fällt schnell auf, dass die einzelnen Durchläufe zu unterschiedlichen Ergeb-

Tabelle 3: Felder im Beispiel „Brustkrebsdiagnose“

Feldname	Datentyp	Bedeutung
ID	integer	Nummer der Gewebeprobe
V1	integer [1 –10]	Verklumpungsstärke
V2	integer [1 –10]	Einheitlichkeit der Zellgröße
V3	integer [1 –10]	Einheitlichkeit der Zellform
V4	integer [1 –10]	Verklebung der Außenränder
V5	integer [1 –10]	Größe einzelner Epithelzellen
V6	integer [1 –10]	Häufigkeit freiliegender Zellkerne
V7	integer [1 –10]	Farbloses Chromatin
V8	integer [1 –10]	Normale Zellkernkörperchen
V9	integer [1 –10]	Häufigkeit an Mitosen
class	character	Diagnose (benign = gutartig, malignant = bösartig)

Listing 1: Automatisches Training mit Ausgabe des Gewinnermodells

```
using Microsoft.ML;
using Microsoft.ML.AutoML;
using Microsoft.ML.Data;
using System;
using System.Collections.Generic;
using System.Linq;

namespace AutoMLExample
{
  public static class Program
  {
    const string trainDataFilename =
      "BiopsyTrainData.csv";
    const string modelFilename = "Model.bin";
    const uint trainingTimeInSeconds = 10;

    static List<TrainingResult> trainingResults =
      new List<TrainingResult>();

    public static void Main(string[] args)
    {
      MLContext mlContext = new MLContext();

      IDataView trainDataView = mlContext.Data.LoadFrom
        TextFile<BiopsyData>(trainDataFilename, ';',
        true);

      var experimentSettings =
        new BinaryExperimentSettings();
      experimentSettings.MaxExperimentTimeInSeconds =
        trainingTimeInSeconds;

      var metricValues = Enum.GetValues(typeof(
        BinaryClassificationMetric));

      // Loop through all metrics
      foreach (BinaryClassificationMetric metricValue
          in metricValues)
      {
        experimentSettings.OptimizingMetric =
          metricValue;

        var experiment = mlContext.Auto().CreateBinary
          ClassificationExperiment(experimentSettings);
        ExperimentResult<BinaryClassificationMetrics>
          experimentResult;

        try
        {
          // Run training
          experimentResult = experiment.Execute(
            trainDataView, "class");
        }
        // Workaround for bug in Execute() Method
        catch (ArgumentOutOfRangeException)
        {
          continue;
        }

        var bestRun = experimentResult.BestRun;
        var metrics = bestRun.ValidationMetrics;

        trainingResults.Add(
          new TrainingResult
```

```
Training Time: 5
Metric: AreaUnderRocCurve
Trainer: AveragedPerceptronBinary
Accuracy: 0,987
AreaUnderPrecisionRecallCurve: 0,99
AreaUnderRocCurve: 0,996
F1Score: 0,979
PositiveRecall: 1
NegativeRecall: 0,982
Positive Precision: 0,958
Negative Precision: 1

TEST POSITIVE RATIO:    0,2911 (23,0/(23,0+56,0))
Confusion table
          ||======================
PREDICTED || positive | negative | Recall
TRUTH     ||======================
 positive ||       23 |        0 | 1,0000
 negative ||        1 |       55 | 0,9821
          ||======================
Precision ||   0,9583 |   1,0000 |
```

Ausgabe der Kenngrößen des Gewinnermodells (Bild 1)

nissen führen. Das ist ein erwartetes Verhalten, da bei der Erzeugung des *MLContext*-Objekts kein Seed benutzt wurde. Interessant ist, dass schon bei kurzen Trainingszeiten von wenigen Sekunden Modelle generiert werden, die keine falsch negativen Ergebnisse liefern. Es ist aber schwierig, Modelle zu finden, die gleichzeitig auch wenige falsch positive Ergebnisse ausgeben. Bei vielfacher Wiederholung des Gesamtexperiments, aber auch bei deutlicher Verlängerung der Trainingsdauer auf 1000 Sekunden pro Iteration erreichte das beste Modell eine positive Genauigkeit von 0,96 (96 %). Dies ist eine leichte Verschlechterung gegenüber dem Referenzwert, der in [12] mit einem neuronalen Netz in kürzester Zeit erreicht wurde (negative Genauigkeit = 1 und positive Genauigkeit = 0,98). Hier macht sich bemerkbar, dass AutoML noch keine neuronalen Netze unterstützt.

Einsatz des Modells

Um das trainierte Modell in der Praxis konkret einzusetzen, sind nur wenige Schritte nötig (Listing 4). Nach dem Laden ▶

```
    {
      ExperimentTimeInSeconds =
        trainingTimeInSeconds,
      Model = bestRun.Model,
      Metric = metricValue,
      Trainer = bestRun.TrainerName,
      Accuracy = metrics.Accuracy,
      AreaUnderPrecisionRecallCurve =
        metrics.AreaUnderPrecisionRecallCurve,
      AreaUnderRocCurve =
        metrics.AreaUnderRocCurve,
      F1Score = metrics.F1Score,
      PositiveRecall = metrics.PositiveRecall,
      NegativeRecall = metrics.NegativeRecall,
      PositivePrecision =
        metrics.PositivePrecision,
      NegativePrecision =
        metrics.NegativePrecision,
      Matrix = metrics.ConfusionMatrix
    });
  }

  trainingResults = trainingResults
    .OrderByDescending(result =>
       result.NegativePrecision)
    .ThenByDescending(result =>
       result.PositivePrecision)
    .ToList<TrainingResult>();

  // Save best model to disk
  var bestTrainingResults = trainingResults[0];
  mlContext.Model.Save(bestTrainingResults.Model,
      trainDataView.Schema, "Model.bin");

  // Print results to console window
  Console.WriteLine($"Training Time: {
    bestTrainingResults.ExperimentTimeInSeconds}");
  Console.WriteLine($"Metric: {
    bestTrainingResults.Metric.ToString()}");
  Console.WriteLine($"Trainer: {
    bestTrainingResults.Trainer}");
  Console.WriteLine($"Accuracy: {
    bestTrainingResults.Accuracy:0.###}");
  Console.WriteLine($"AreaUnderPrecisionRecallCurve:
    {bestTrainingResults.AreaUnderPrecisionRecall
    Curve:0.###}");
  Console.WriteLine($"AreaUnderRocCurve: {
    bestTrainingResults.AreaUnderRocCurve:0.###}");
  Console.WriteLine($"F1Score: {
    bestTrainingResults.F1Score:0.###}");
  Console.WriteLine($"PositiveRecall: {
    bestTrainingResults.PositiveRecall:0.###}");
  Console.WriteLine($"NegativeRecall: {
    bestTrainingResults.NegativeRecall:0.###}");
  Console.WriteLine($"Positive Precision: {
    bestTrainingResults.PositivePrecision:0.###}");
  Console.WriteLine($"Negative Precision: {
    bestTrainingResults.NegativePrecision:0.###}");
  Console.WriteLine();
  Console.WriteLine(bestTrainingResults.Matrix.Get
    FormattedConfusionTable());
 }
}
```

Listing 2: Eingangsfelder in einer Klasse deklarieren

```
using Microsoft.ML.Data;

namespace AutoMLExample
{
  public class BiopsyData
  {
    [LoadColumn(0)]  public float ID;
    [LoadColumn(1)]  public float V1;
    [LoadColumn(2)]  public float V2;
    [LoadColumn(3)]  public float V3;
    [LoadColumn(4)]  public float V4;
    [LoadColumn(5)]  public float V5;
    [LoadColumn(6)]  public float V6;
    [LoadColumn(7)]  public float V7;
    [LoadColumn(8)]  public float V8;
    [LoadColumn(9)]  public float V9;
    [LoadColumn(10)] public bool  @class;
  }
}
```

des Modells erzeugen Sie zunächst ein *PredictionEngine*-Objekt, das sowohl auf die Eingangsdaten (*BiopsyData*) als auch Ausgabedaten (*BiopsyPredicition*, Listing 5) typisiert ist. Die Klasse *BiopsyPredicition* erbt von *BiopsyData*, wobei dies lediglich dazu dient, dass die Ausgabedaten auch die ursprünglichen Eingangswerte enthalten. Entscheidend ist die zu ermittelnde Eigenschaft *IsMalignant* (ist bösartig), die mit dem Attribut *ColumnName("PredictedLabel")* dekoriert werden muss. Im Beispiel wurde darüber hinaus noch die Eigenschaft *Score* hinzugefügt, die die Zuverlässigkeit des Ergebnisses angibt.

Nach diesen Vorbereitungen können Sie das Modell nutzen, indem Sie die *Predict*-Methode von *PredictionEngine* aufrufen. Aus dem zurückgelieferten Objekt *BiopsyPrediction* lässt sich dann das Resultat auslesen.

Fazit

AutoML-Tools sind pragmatische Lösungen für reale Problemstellungen. Der Ansatz des Durchprobierens vieler Lösungsvarianten ist bei nicht trivialen Optimierungsproblemen durchaus sinnvoll. Microsofts Ansatz ML.NET AutoML macht hier schon einiges richtig, und Sie kommen als Entwickler mit sehr wenig Code aus.

Listing 3: Hilfsklasse als Container für die trainierten Modelle

```
using Microsoft.ML;
using Microsoft.ML.AutoML;
using Microsoft.ML.Data;

namespace AutoMLExample
{
  public class TrainingResult
  {
          // Configuration
    public uint                         ExperimentTimeInSeconds        { get; set; }
    public ITransformer                 Model                          { get; set; }
    public string                       Trainer                        { get; set; }
    public BinaryClassificationMetric   Metric                         { get; set; }

          // Metrics
    public double                       Accuracy                       { get; set; }
    public double                       AreaUnderPrecisionRecallCurve  { get; set; }
    public double                       AreaUnderRocCurve              { get; set; }
    public double                       F1Score                        { get; set; }
    public double                       PositiveRecall                 { get; set; }
    public double                       NegativeRecall                 { get; set; }
    public double                       PositivePrecision              { get; set; }
    public double                       NegativePrecision              { get; set; }

    public ConfusionMatrix              Matrix                         { get; set; }
  }
}
```

Listing 4: Einsatz eines trainierten Modells

```
MLContext mlContext = new MLContext();

// Load model
ITransformer model = mlContext.Model.Load(
  modelFilename, out var schema);

// Create PredictionEngine
PredictionEngine<BiopsyData, BiopsyPrediction>
  predictionEngine = mlContext.Model.CreatePrediction
  Engine<BiopsyData, BiopsyPrediction>(model);

BiopsyData inputData = new BiopsyData
{
  ID = 1f,
  V1 = 2f,
  V2 = 4f,
  V3 = 10f,
  V4 = 10f,
  V5 = 10f,
  V6 = 10f,
  V7 = 2f,
  V8 = 2f,
  V9 = 3f
};

BiopsyPrediction prediction = predictionEngine.
  Predict(inputData);

Console.Write($"ID: {prediction.ID}\t");
Console.Write($"Is malignant: {
  prediction.IsMalignant}\t");
Console.Write($"Score: {prediction.Score}\t");
```

Listing 5: Modell mit separater Klasse für das Resultat

```
using Microsoft.ML.Data;

namespace AutoMLExample
{
  public class BiopsyPrediction : BiopsyData
  {
    [ColumnName("PredictedLabel")]
    public bool  IsMalignant { get; set; }

    public float Score       { get; set; }
  }
}
```

Natürlich dauert das Probieren seine Zeit, und Sie verlieren beim Einsatz solcher Tools an Einfluss auf den Trainingsvorgang. Benötigen Sie diesen Einfluss, dann können Sie AutoML immer noch für eine Vorauswahl vielversprechender Modelle nutzen und diese dann mit den weiteren Möglichkeiten des ML.NET Frameworks untersuchen und optimieren.

Es wird interessant sein zu sehen, welche Trainingsalgorithmen Microsoft bis zum offiziellen Release noch hinzufügen wird. Besonders neuronale Netze müssen hier nachgeliefert werden. Nützlich wäre auch, weitere Aspekte des Trainingsprozesses automatisieren zu können, so zum Beispiel eine automatische Korrektur und Ergänzung von Eingangswerten, eine eigenständige Erkennung des Problemtyps (Klassifikation versus Regression) oder die Berücksichtigung des Ressourcenverbrauchs (Speicher und Trainingszeit) einzelner Algorithmen. ■

[1] Top Machine Learning Algorithms You Should Know to Become a Data Scientist, www.dotnetpro.de/SL1912AutoML1

[2] Google Cloud AutoML, www.dotnetpro.de/SL1912AutoML2

[3] Auto-Keras, https://autokeras.com

[4] ML.NET AutoML, www.dotnetpro.de/SL1912AutoML3

[5] ML.NET Documentation, www.dotnetpro.de/SL1912AutoML4

[6] BinaryClassificationTrainer Enum, www.dotnetpro.de/SL1912AutoML5

[7] MulticlassClassificationTrainer Enum, www.dotnetpro.de/SL1912AutoML6

[8] RegressionTrainer Enum, www.dotnetpro.de/SL1912AutoML7

[9] BinaryClassificationMetric Enum, www.dotnetpro.de/SL1912AutoML8

[10] MulticlassClassificationMetric Enum, www.dotnetpro.de/SL1912AutoML9

[11] RegressionMetric Enum, www.dotnetpro.de/SL1912AutoML10

[12] Martin Gossen, Smarte Datenbank, dotnetpro 10/2017, Seite 92 ff., www.dotnetpro.de/A1710MicrosoftML

[13] Confusion matrix, www.dotnetpro.de/SL1912AutoML11

[14] Metrics to Evaluate your Machine Learning Algorithm, www.dotnetpro.de/SL1912AutoML12

[15] Hyperparameters in Deep Learning, www.dotnetpro.de/SL1912AutoML13

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 A1912AutoML