✦ Member-only story

Predict Heart Disease With C# And ML.NET Machine Learning

Mark Farragher · Follow

7 min read · Apr 9, 2019

Twenty years ago, something strange happened to my dad. He was out of breath all the time. Climbing the stairs left him exhausted. Gardening became a struggle. Even simple activities tired him out.

So he went to a specialist for a checkup. The doctor put him on an exercise bike, hooked him up to an EKG machine, and told him to start pedaling.

My dad told me later that he barely got started when the doctor yelled “Stop! stop!” and called an ambulance right away. He was just days away from having a major heart attack!

The doctor who read the EKG and saved my father’s life instantly recognized the signs of a pending heart attack.

But here’s a question: could we train a machine to do the same?

Wearables are getting smarter all the time. We are very close to having watches that can measure heart activity and alert a doctor if something is wrong. This technology can and will save lives!

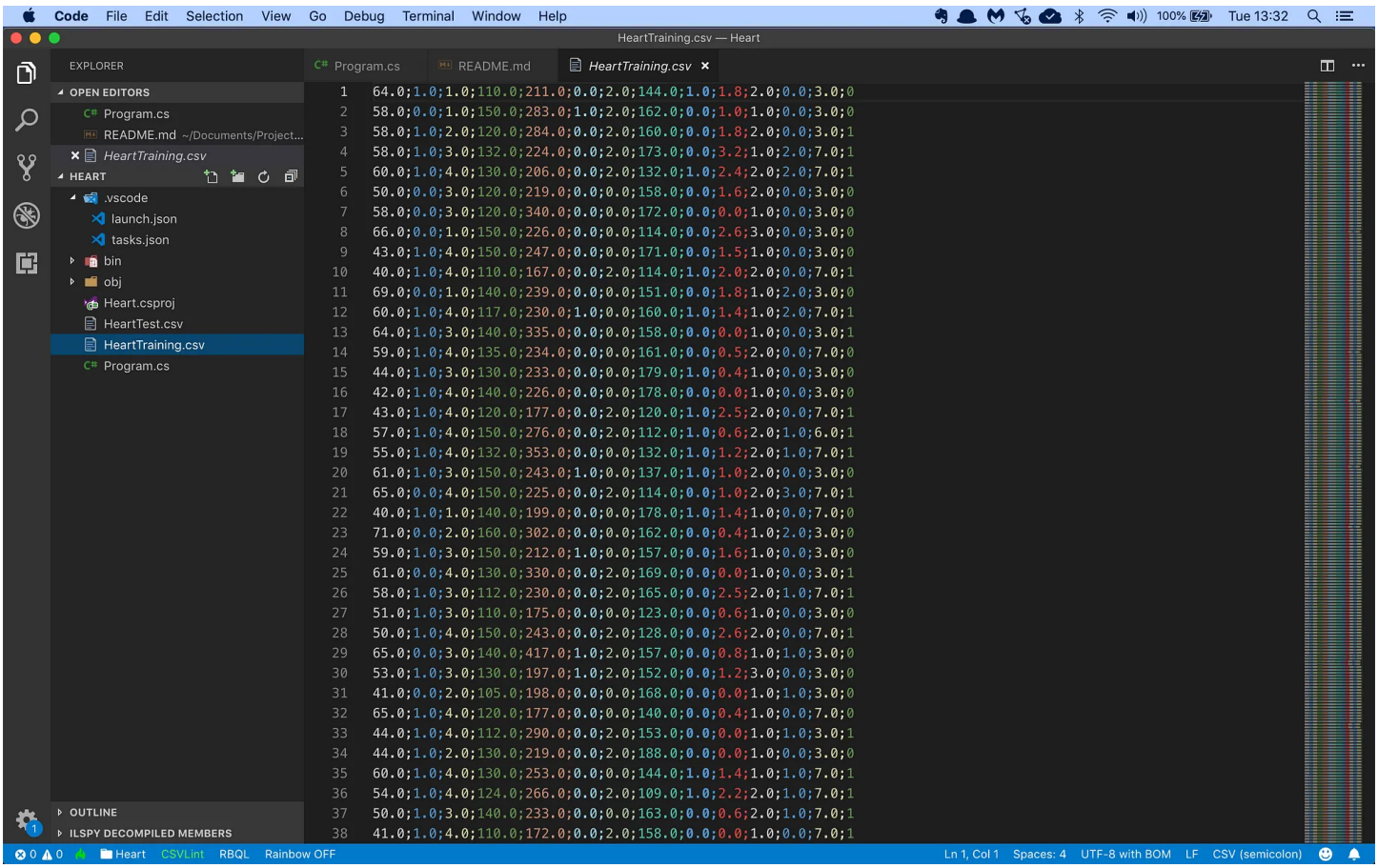
So in this article, I am going to build a C# app with ML.NET and NET Core that reads medical data and predicts if a patient has a risk of heart disease. I will show you how we can emulate the skills of my father's doctor with just 200 lines of code.

ML.NET is Microsoft's new machine learning library. It can run linear regression, logistic classification, clustering, deep learning, and many other machine learning algorithms.

And **NET Core** is the Microsoft multi-platform NET Framework that runs on Windows, OS/X, and Linux. It's the future of cross-platform NET development.

The first thing I need for my app is a data file with patients, their medical info, and their heart disease risk assessment. I will use the famous UCI Heart Disease Dataset which has real-life data from 303 patients.

The training data file looks like this:



It's a CSV file with 14 columns of information:

- Age
- Sex: 1 = male, 0 = female
- Chest Pain Type: 1 = typical angina, 2 = atypical angina , 3 = non-anginal pain, 4 = asymptomatic
- Resting blood pressure in mm Hg on admission to the hospital
- Serum cholesterol in mg/dl
- Fasting blood sugar > 120 mg/dl: 1 = true; 0 = false
- Resting EKG results: 0 = normal, 1 = having ST-T wave abnormality, 2 = showing probable or definite left ventricular hypertrophy by Estes' criteria
- Maximum heart rate achieved
- Exercise induced angina: 1 = yes; 0 = no
- ST depression induced by exercise relative to rest
- Slope of the peak exercise ST segment: 1 = up-sloping, 2 = flat, 3 = down-sloping
- Number of major vessels (0–3) colored by fluoroscopy
- Thallium heart scan results: 3 = normal, 6 = fixed defect, 7 = reversible defect
- Diagnosis of heart disease: 0 = less than 50% diameter narrowing, 1 = more than 50% diameter narrowing

The first 13 columns are patient diagnostic information, and the last column is the diagnosis: 0 means a healthy patient, and 1 means an elevated risk of heart disease.

I will build a binary classification machine learning model that reads in all 13 columns of patient information, and then makes a prediction for the heart disease risk.

Let's get started. Here's how to set up a new console project in NET Core:

```
$ dotnet new console -o Heart
$ cd Heart
```

Next, I need to install the ML.NET base package:

```
$ dotnet add package Microsoft.ML
```

Now I'm ready to add some classes. I'll need one to hold patient info, and one to hold my model's predictions.

I will modify the Program.cs file like this:

```
/// <summary>
/// The HeartData record holds one single heart data record.
/// </summary>
public class HeartData
{
    [LoadColumn(0)] public float Age { get; set; }
    [LoadColumn(1)] public float Sex { get; set; }
    [LoadColumn(2)] public float Cp { get; set; }
    [LoadColumn(3)] public float TrestBps { get; set; }
    [LoadColumn(4)] public float Chol { get; set; }
    [LoadColumn(5)] public float Fbs { get; set; }
    [LoadColumn(6)] public float RestEcg { get; set; }
    [LoadColumn(7)] public float Thalac { get; set; }
    [LoadColumn(8)] public float Exang { get; set; }
    [LoadColumn(9)] public float OldPeak { get; set; }
    [LoadColumn(10)] public float Slope { get; set; }
    [LoadColumn(11)] public float Ca { get; set; }
    [LoadColumn(12)] public float Thal { get; set; }
    [LoadColumn(13)] public bool Label { get; set; }
}

/// <summary>
/// The HeartPrediction class contains a single heart data prediction.
/// </summary>
public class HeartPrediction
{
    [ColumnName("PredictedLabel")] public bool Prediction;
    public float Probability;
    public float Score;
}
```

mlnet-heart-classes.cs hosted with ❤ by **GitHub** **view raw**

The **HeartData** class holds one single patient record. Note how each field is adorned with a **Column** attribute that tell the CSV data loading code which column to import data from.

I'm also declaring a **HeartPrediction** class which will hold a single heart disease prediction.

Now I'm going to load the training data in memory:

```
/// <summary>
/// The application class.
/// </summary>
public class Program
{
    // filenames for training and test data
    private static string trainingDataPath = Path.Combine(Environment.Curre
    private static string testDataPath = Path.Combine(Environment.CurrentDi

    /// <summary>
    /// The main applicaton entry point.
    /// </summary>
    /// <param name="args">The command line arguments.</param>
    public static void Main(string[] args)
    {
        // set up a machine learning context
        var mlContext = new MLContext();

        // load training and test data
        Console.WriteLine("Loading data...");
        var trainingDataView = mlContext.Data.LoadFromTextFile<HeartData>(t
        var testDataView = mlContext.Data.LoadFromTextFile<HeartData>(testD

        // the rest of the code goes here...
    }
}
```

mlnet-heart-loading.cs hosted with ❤ by **GitHub** **view raw**

This code uses the method **LoadFromTextFile** to load the CSV data directly into memory. The class field annotations tell the method how to store the loaded data in the **HeartData** class.

Now I'm ready to start building the machine learning model:

```
// set up a training pipeline
// step 1: concatenate all feature columns
var pipeline = mlContext.Transforms.Concatenate(
    "Features",
    "Age",
    "Sex",
    "Cp",
    "TrestBps",
    "Chol",
    "Fbs",
    "RestEcg",
    "Thalac",
    "Exang",
    "OldPeak",
    "Slope",
    "Ca",
    "Thal")

    // step 2: set up a fast tree learner
    .Append(mlContext.BinaryClassification.Trainers.FastTree(
        labelColumnName: DefaultColumnNames.Label,
        featureColumnName: DefaultColumnNames.Features));

// train the model
Console.WriteLine("Training model...");
var trainedModel = pipeline.Fit(trainingDataView);

// the rest of the code goes here...
```

mlnet-heart-train.cs hosted with ❤ by **GitHub** **view raw**

Machine learning models in ML.NET are built with pipelines, which are sequences of data-loading, transformation, and learning components.

My pipeline has the following components:

- **Concatenate** which combines all input data columns into a single column called Features. This is a required step because ML.NET can only train on a single input column.

- A **FastTree** classification learner which will train the model to make accurate predictions.

The **FastTreeBinaryClassificationTrainer** is a very nice training algorithm that uses gradient boosting, a machine learning technique for classification problems.

With the pipeline fully assembled, I can train the model with a call to **Fit(...)**.

I now have a fully- trained model. So now I need to take the test data, predict the diagnosis for each patient, and calculate the accuracy metrics of my model:

```
// make predictions for the test data set
Console.WriteLine("Evaluating model...");
var predictions = trainedModel.Transform(testDataView);

// compare the predictions with the ground truth
var metrics = mlContext.BinaryClassification.Evaluate(
    data: predictions,
    label: DefaultColumnNames.Label,
    score: DefaultColumnNames.Score);

// report the results
Console.WriteLine($"  Accuracy:          {metrics.Accuracy:P2}");
Console.WriteLine($"  Auc:               {metrics.Auc:P2}");
Console.WriteLine($"  Auprc:             {metrics.Auprc:P2}");
Console.WriteLine($"  F1Score:           {metrics.F1Score:P2}");
Console.WriteLine($"  LogLoss:           {metrics.LogLoss:0.##}");
Console.WriteLine($"  LogLossReduction:  {metrics.LogLossReduction:0.##}")
Console.WriteLine($"  PositivePrecision: {metrics.PositivePrecision:0.##}"
Console.WriteLine($"  PositiveRecall:    {metrics.PositiveRecall:0.##}");
Console.WriteLine($"  NegativePrecision: {metrics.NegativePrecision:0.##}"
Console.WriteLine($"  NegativeRecall:    {metrics.NegativeRecall:0.##}");
Console.WriteLine();

// the rest of the code goes here...
```

mlnet-heart-metrics.cs hosted with ❤ by **GitHub** **view raw**

This code calls **Transform(...)** to set up a diagnosis for every patient in the set, and **Evaluate(...)** to compare these predictions to the ground truth and automatically

calculate all evaluation metrics for me:

- **Accuracy**: this is the number of correct predictions divided by the total number of predictions.
- **AUC**: a metric that indicates how accurate the model is: 0 = the model is wrong all the time, 0.5 = the model produces random output, 1 = the model is correct all the time. An AUC of 0.8 or higher is considered good.
- **AUCPRC**: an alternate AUC metric that performs better for heavily imbalanced datasets with many more negative results than positive.
- **F1Score**: this is a metric that strikes a balance between Precision and Recall. It's useful for imbalanced datasets with many more negative results than positive.
- **LogLoss**: this is a metric that expresses the size of the error in the predictions the model is making. A logloss of zero means every prediction is correct, and the loss value rises as the model makes more and more mistakes.
- **LogLossReduction**: this metric is also called the Reduction in Information Gain (RIG). It expresses the probability that the model's predictions are better than random chance.
- **PositivePrecision**: also called 'Precision', this is the fraction of positive predictions that are correct. This is a good metric to use when the cost of a false positive prediction is high.
- **PositiveRecall**: also called 'Recall', this is the fraction of positive predictions out of all positive cases. This is a good metric to use when the cost of a false negative is high.
- **NegativePrecision**: this is the fraction of negative predictions that are correct.
- **NegativeRecall**: this is the fraction of negative predictions out of all negative cases.

When monitoring heart disease, I definitely want to avoid false negatives because I don't want to be sending high-risk patients home and telling them everything is

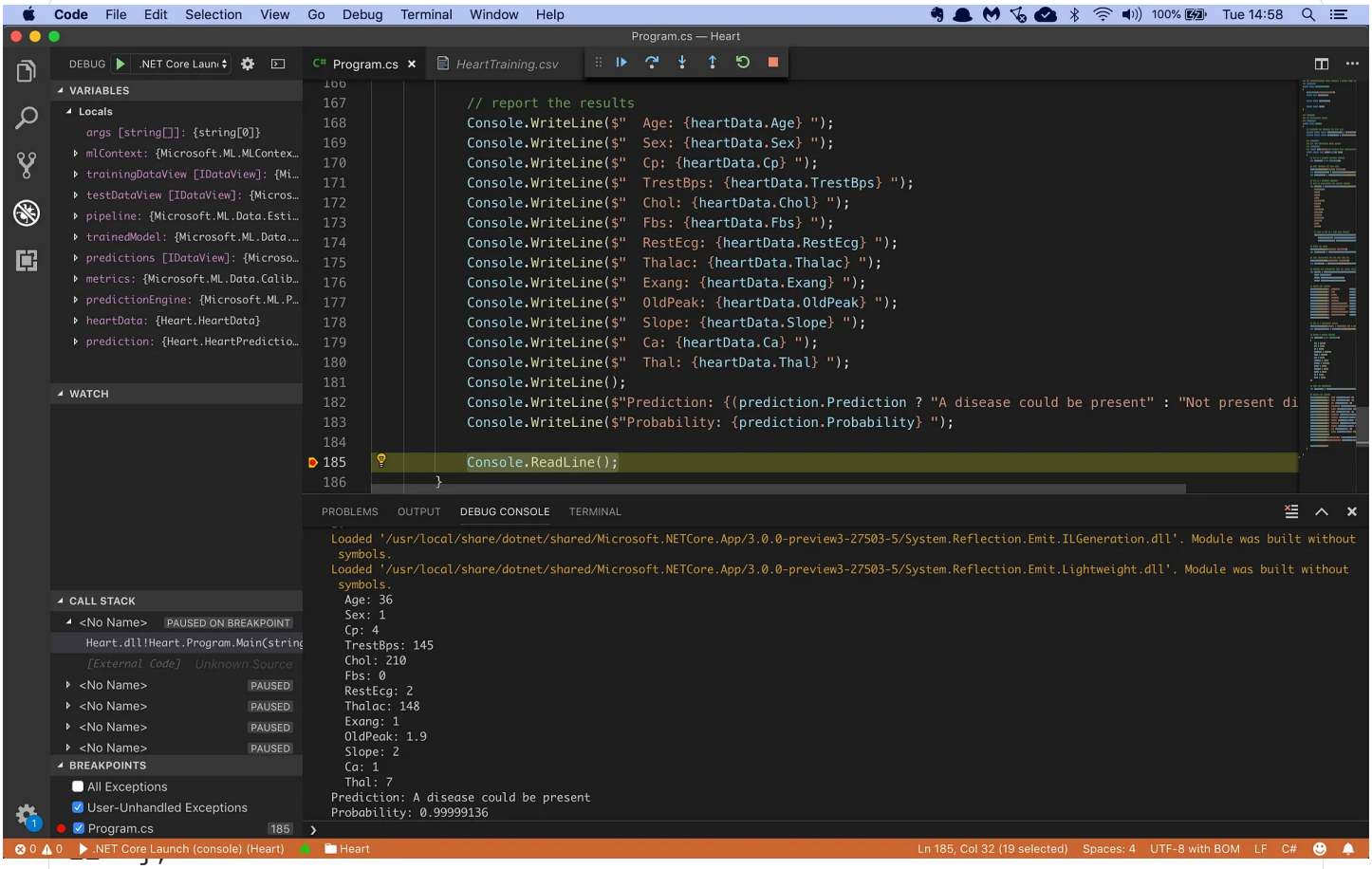
okay.

I also want to avoid false positives, but they are a lot better than a false negative because later tests would probably discover that the patient is healthy after all.

The data set has a nicely balanced distribution of positive and negative labels, so there's no need to use the AUCPRC or F1Score metrics.

So in our case, I'm going to focus on Recall and AUC to evaluate this model.

To wrap up, I'm going to create a new patient record and ask the model to make a prediction:




```
22
23  // make the prediction
24  var prediction = predictionEngine.Predict(heartData);
25
26  // report the results
27  Console.WriteLine($"  Age: {heartData.Age} ");
28  Console.WriteLine($"  Sex: {heartData.Sex} ");
29  Console.WriteLine($"  Cp: {heartData.Cp} ");
30  Console.WriteLine($"  TrestBps: {heartData.TrestBps} ");
31  Console.WriteLine($"  Chol: {heartData.Chol} ");
32  Console.WriteLine($"  Fbs: {heartData.Fbs} ");
33  Console.WriteLine($"  RestEcg: {heartData.RestEcg} ");
34  Console.WriteLine($"  Thalac: {heartData.Thalac} ");
35  Console.WriteLine($"  Exang: {heartData.Exang} ");
36  Console.WriteLine($"  OldPeak: {heartData.OldPeak} ");
37  Console.WriteLine($"  Slope: {heartData.Slope} ");
38  Console.WriteLine($"  Ca: {heartData.Ca} ");
39  Console.WriteLine($"  Thal: {heartData.Thal} ");
40  Console.WriteLine();
41  Console.WriteLine($"Prediction: {(prediction.Prediction ? "A disease could
42  Console.WriteLine($"Probability: {prediction.Probability} ");
```

I use the CreateBinaryPredictionEngine method to set up a prediction engine. The true

```
~/Documents/Projects/MLNET/Heart dotnet run
Loading data...
Training model...
Evaluating model...
  Accuracy:          94.74%
  Auc:               96.43%
  Auprc:             95.48%
  F1Score:           92.31%
  LogLoss:           0.36
  LogLossReduction:  62.56
  PositivePrecision: 1
  PositiveRecall:    0.86
  NegativePrecision: 0.92
  NegativeRecall:    1

Making a prediction for a sample patient...
  Age: 36
  Sex: 1
  Cp: 4
  TrestBps: 145
  Chol: 210
  Fbs: 0
  RestEcg: 2
  Thalac: 148
  Exang: 1
  OldPeak: 1.9
  Slope: 2
  Ca: 1
  Thal: 7

Prediction: A disease could be present
Probability: 0.99999136
```

The results nicely illustrate how to evaluate a binary classifier. I get a precision of 1 which is awesome. It means all positive predictions made by the model are correct.

Done deal, right?

Not so fast. The recall is 0.86, which means that out of all positive cases, my model only predicted 86% correct. The remaining 14% are high-risk heart patients who were told that everything is fine and they can go home.

That's obviously very bad, and it clearly shows how important the recall metric is in

cases where we want to avoid false negatives at all costs.

This article is based on a homework assignment from my machine learning course: Machine Learning with C# and ML.NET.

I'm getting an AUC of 96.43% which is a very good result. It means this model has excellent predictive ability.

Machine Learning | Data Science | Programming | Deep Learning | Artificial Intelligence

Finally, my model is 99.99% confident that my 36-year old male patient with [illegible] pain has a high-risk for heart disease.

Looks like we caught that one in time!

So what do you think?

Are you ready to start writing C# machine learning apps with ML.NET?

Follow

Written by Mark Farragher

1.8K Followers

Meet Your New Trainer https://www.mdfarragher.com

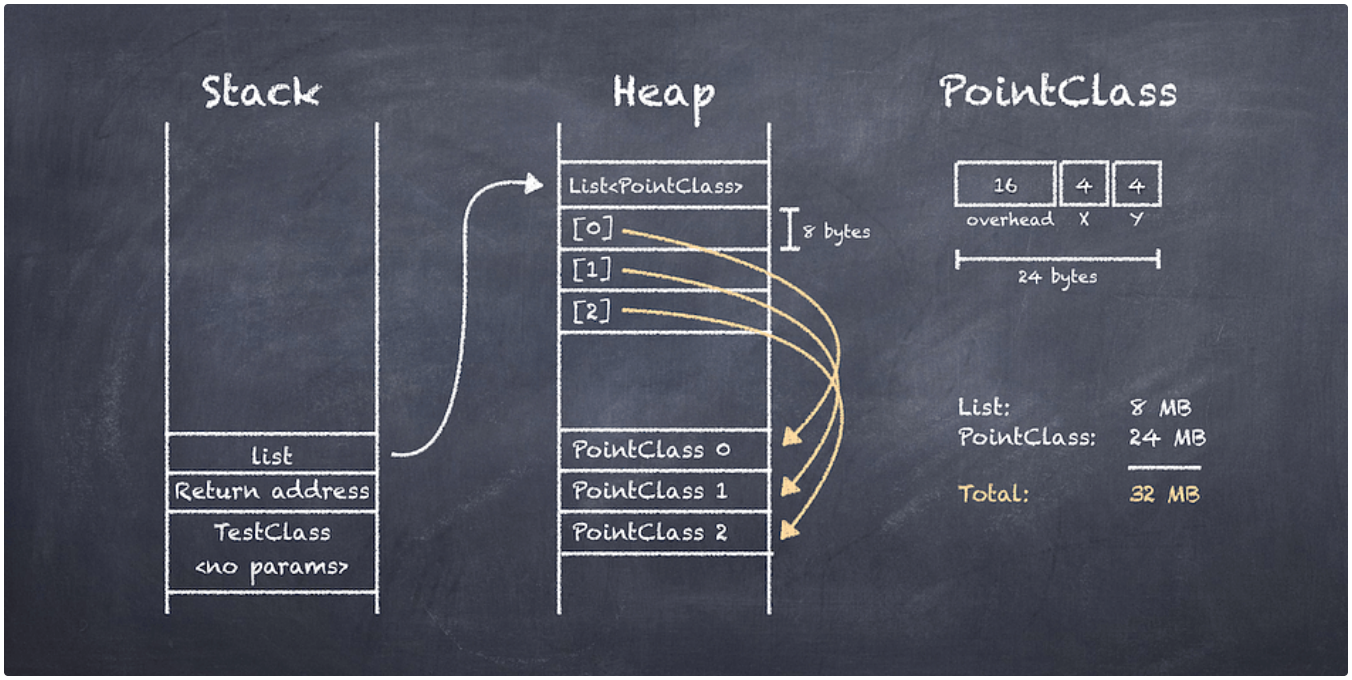
More from Mark Farragher

Mark Farragher

Optical Character Recognition With C#, ML.NET, And NET Core

Watch me build a cross-platform C# app that uses NET Core and ML.NET to recognize handwritten digits in the famous MNIST dataset.

· 6 min read · Apr 22, 2019



Mark Farragher

What Is Faster In C#: A Struct Or A Class?

What do you think is faster: filling an array with one million structs, or filling an array with one million classes?

✦ · 4 min read · Mar 15, 2019

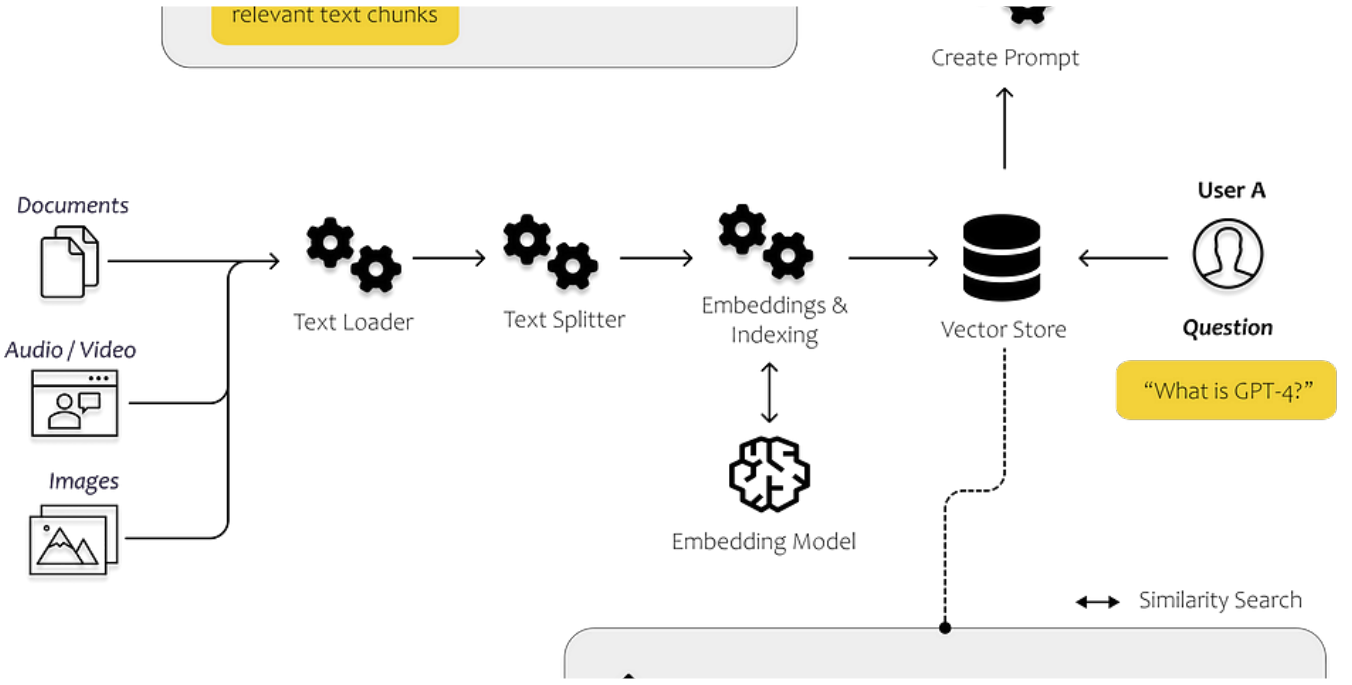
2.9K 15

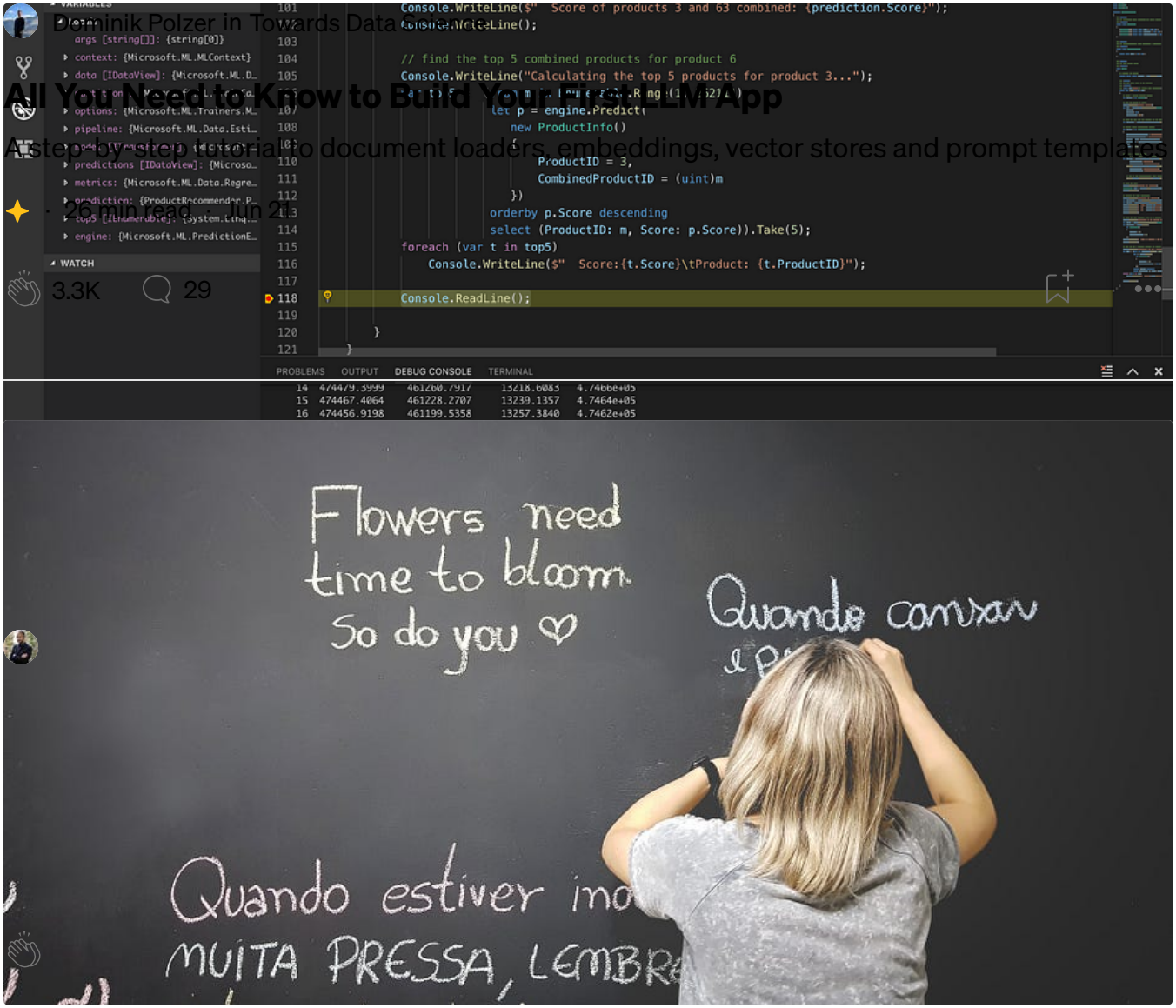
Mark Farragher

Run TensorFlow Machine Learning Code In C# With Almost No Changes

Thanks to SciSharp, we can now copy TensorFlow machine learning code and run it in C# with almost no source code changes.

· 6 min read · Jun 27, 2019





Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

· 26 min read · Jun 21

3.3K 29

Christophe Atten in DataDrivenInvestor

See all from Mark Farragher

A guide to Natural Language Processing — Basics

Learn the basics of Natural Language Processing, how it works, and what its limitations are

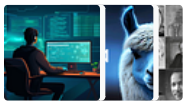
· 9 min read · Feb 27

31

Lists

Predictive Modeling w/ Python

18 stories · 174 saves



Natural Language Processing
444 stories · 82 saves

Practical Guides to Machine Learning
10 stories · 194 saves

ChatGPT prompts
22 stories · 170 saves

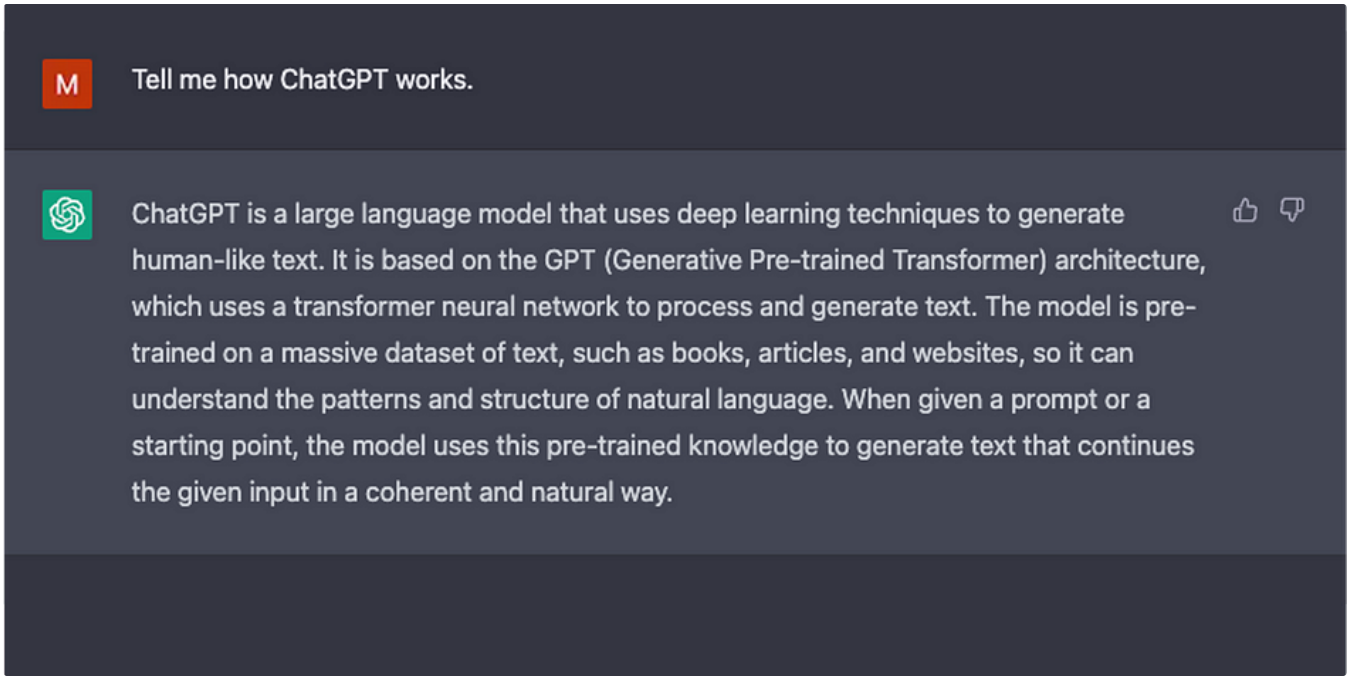
Ignacio de Gregorio

Microsoft Just Showed us the Future of ChatGPT with LongNet

Let's talk about Billions

· 8 min read · Jul 20

2.4K 35



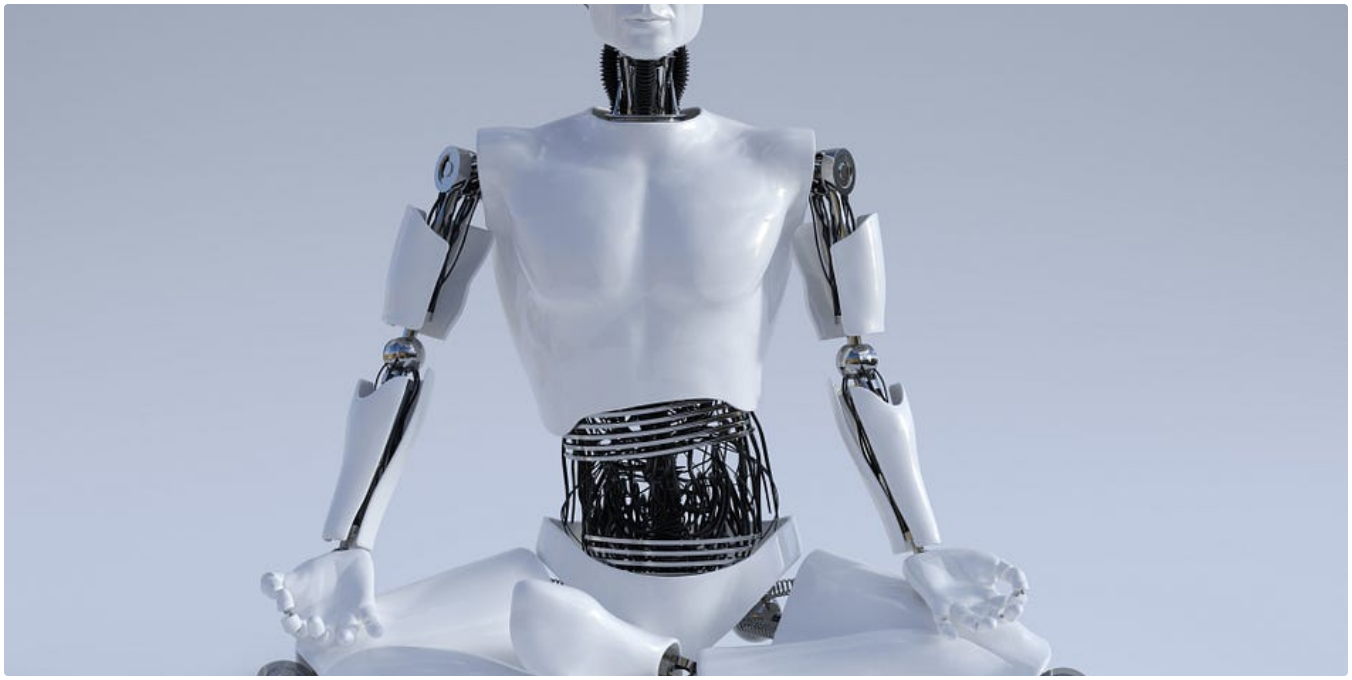
Molly Ruby in Towards Data Science

How ChatGPT Works: The Models Behind The Bot

A brief introduction to the intuition and methodology behind the chat bot you can't stop hearing about.

✦ · 8 min read · Jan 30

7.6K 127



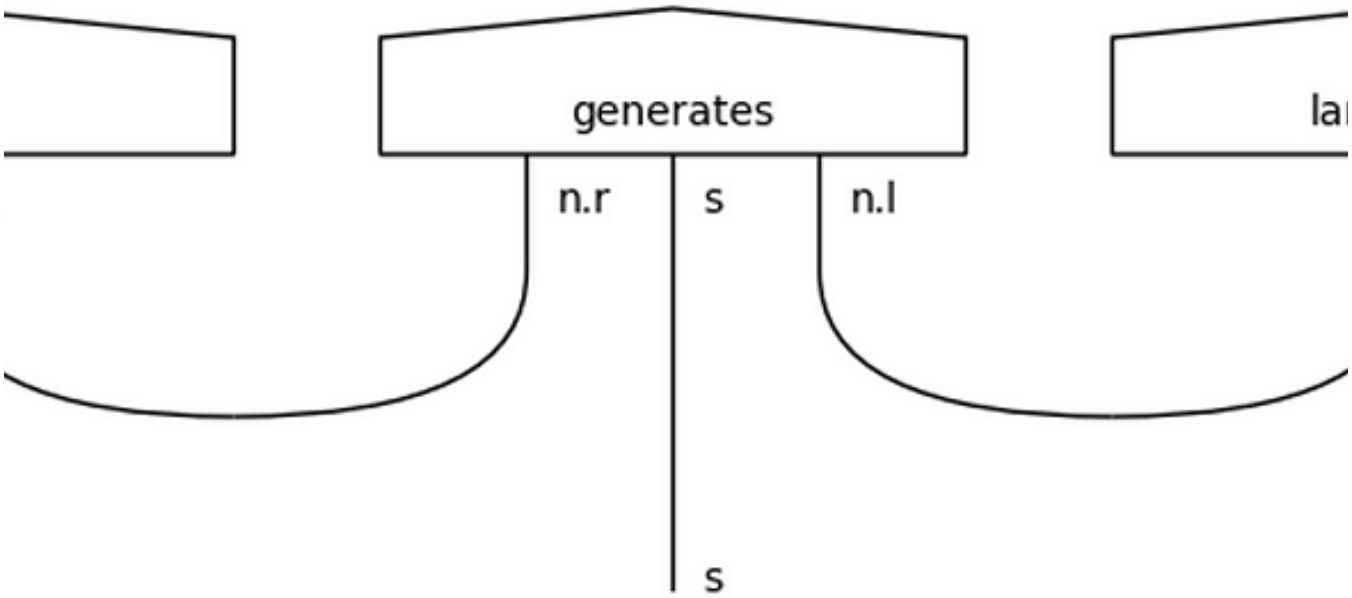
The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

✨ · 7 min read · Mar 17

29K 526



Qiskit in Qiskit

An introduction to Quantum Natural Language Processing

By Amin Karamlou, Marcel Pfaffhauser, and James Wootton

10 min read · Nov 4, 2022

183

See more recommendations