Is this page helpful?
Yes No

Prepare data for building a model

09/10/2019 • 6 minutes to read •

Learn how to use ML.NET to prepare data for additional processing or building a model.

Data is often unclean and sparse. ML.NET machine learning algorithms expect input or features to be in a single numerical vector. Similarly, the value to predict (label), especially when it's categorical data, has to be encoded. Therefore one of the goals of data preparation is to get the data into the format expected by ML.NET algorithms.

Filter data

Sometimes, not all data in a dataset is relevant for analysis. An approach to remove irrelevant data is filtering. The DataOperationsCatalog contains a set of filter operations that take in an IDataView containing all of the data and return an IDataView containing only the data points of interest. It's important to note that because filter operations are not an IEstimator or ITransformer like those in the TransformsCatalog, they cannot be included as part of an EstimatorChain or TransformerChain data preparation pipeline.

Using the following input data which is loaded into an IDataView:

```C#
HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        NumberOfBedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        NumberOfBedrooms=6f,
        Price=600000f
    }
};
```

To filter data based on the value of a column, use the FilterRowsByColumn method.

```C#
// Apply filter
IDataView filteredData = mlContext.Data.FilterRowsByColumn(data, "Price", lowerBound
```

The sample above takes rows in the dataset with a price between 200000 and 1000000. The result of applying this filter would return only the last two rows in the data and exclude the first row because its price is 100000 and not between the specified range.

Replace missing values

Missing values are a common occurrence in datasets. One approach to dealing with missing values is to replace them with the default value for the given type if any or another meaningful value such as the mean value in the data.

Using the following input data which is loaded into an IDataView:

```C#
HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        NumberOfBedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        NumberOfBedrooms=6f,
        Price=float.NaN
    }
};
```

Notice that the last element in our list has a missing value for `Price`. To replace the missing values in the `Price` column, use the ReplaceMissingValues method to fill in that missing value.

> **Important**
>
> ReplaceMissingValue only works with numerical data.

```C#
// Define replacement estimator
var replacementEstimator = mlContext.Transforms.ReplaceMissingValues("Price", replac

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimat
ITransformer replacementTransformer = replacementEstimator.Fit(data);

// Transform data
IDataView transformedData = replacementTransformer.Transform(data);
```

ML.NET supports various replacement modes. The sample above uses the `Mean` replacement mode which will fill in the missing value with that column's average value. The replacement 's result fills in the `Price` property for the last element in our data with 200,000 since it's the average of 100,000 and 300,000.

Use normalizers

Normalization is a data pre-processing technique used to standardize features that are not on the same scale which helps algorithms converge faster. For example, the ranges for values like age and income vary significantly with age generally being in the range of 0-100 and income generally being in the range of zero to thousands. Visit the transforms page for a more detailed list and description of normalization transforms.

Min-Max normalization

Using the following input data which is loaded into an IDataView:

```C#
HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms = 2f,
        Price = 200000f
    },
    new HomeData
    {
        NumberOfBedrooms = 1f,
        Price = 100000f
    }
};
```

Normalization can be applied to columns with single numerical values as well as vectors. Normalize the data in the `Price` column using min-max normalization with the NormalizeMinMax method.

```C#
// Define min-max estimator
var minMaxEstimator = mlContext.Transforms.NormalizeMinMax("Price");

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimat
ITransformer minMaxTransformer = minMaxEstimator.Fit(data);

// Transform data
IDataView transformedData = minMaxTransformer.Transform(data);
```

The original price values `[200000,100000]` are converted to `[ 1, 0.5 ]` using the `MinMax` normalization formula which generates output values in the range of 0-1.

Binning

Binning converts continuous values into a discrete representation of the input. For example, suppose one of your features is age. Instead of using the actual age value, binning creates ranges for that value. 0-18 could be one bin, another could be 19-35 and so on.

Using the following input data which is loaded into an IDataView:

```C#
HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        NumberOfBedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        NumberOfBedrooms=6f,
        Price=600000f
    }
};
```

Normalize the data into bins using the NormalizeBinning method. The `maximumBinCount` parameter enables you to specify the number of bins needed to classify your data. In this example, data will be put into two bins.

```C#
// Define binning estimator
var binningEstimator = mlContext.Transforms.NormalizeBinning("Price", maximumBinCoun

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimat
var binningTransformer = binningEstimator.Fit(data);

// Transform Data
IDataView transformedData = binningTransformer.Transform(data);
```

The result of binning creates bin bounds of `[0,200000,Infinity]`. Therefore the resulting bins are `[0,1,1]` because the first observation is between 0-200000 and the others are greater than 200000 but less than infinity.

Work with categorical data

Non-numeric categorical data needs to be converted to a number before being used to build a machine learning model.

Using the following input data which is loaded into an IDataView:

```C#
CarData[] cars = new CarData[]
{
    new CarData
    {
        Color="Red",
        VehicleType="SUV"
    },
    new CarData
    {
        Color="Blue",
        VehicleType="Sedan"
    },
    new CarData
    {
        Color="Black",
        VehicleType="SUV"
    }
};
```

The categorical `VehicleType` property can be converted into a number using the OneHotEncoding method.

```C#
// Define categorical transform estimator
var categoricalEstimator = mlContext.Transforms.Categorical.OneHotEncoding("VehicleT

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimat
ITransformer categoricalTransformer = categoricalEstimator.Fit(data);

// Transform Data
IDataView transformedData = categoricalTransformer.Transform(data);
```

The resulting transform converts the text value of `VehicleType` to a number. The entries in the `VehicleType` column become the following when the transform is applied:

```text
[
    1, // SUV
    2, // Sedan
    1 // SUV
]
```

Work with text data

Text data needs to be transformed into numbers before using it to build a machine learning model. Visit the transforms page for a more detailed list and description of text transforms.

Using data like the data below that has been loaded into an IDataView:

```C#
ReviewData[] reviews = new ReviewData[]
{
    new ReviewData
    {
        Description="This is a good product",
        Rating=4.7f
    },
    new ReviewData
    {
        Description="This is a bad product",
        Rating=2.3f
    }
};
```

The minimum step to convert text to a numerical vector representation is to use the FeaturizeText method. By using the FeaturizeText transform, a series of transformations is applied to the input text column resulting in a numerical vector representing the lp-normalized word and character ngrams.

```C#
// Define text transform estimator
var textEstimator  = mlContext.Transforms.Text.FeaturizeText("Description");

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimat
ITransformer textTransformer = textEstimator.Fit(data);

// Transform data
IDataView transformedData = textTransformer.Transform(data);
```

The resulting transform would convert the text values in the `Description` column to a numerical vector that looks similar to the output below:

```text
[ 0.2041241, 0.2041241, 0.2041241, 0.4082483, 0.2041241, 0.2041241, 0.2041241, 0.204
```

Combine complex text processing steps into an EstimatorChain to remove noise and potentially reduce the amount of required processing resources as needed.

```C#
// Define text transform estimator
var textEstimator = mlContext.Transforms.Text.NormalizeText("Description")
    .Append(mlContext.Transforms.Text.TokenizeIntoWords("Description"))
    .Append(mlContext.Transforms.Text.RemoveDefaultStopWords("Description"))
    .Append(mlContext.Transforms.Conversion.MapValueToKey("Description"))
    .Append(mlContext.Transforms.Text.ProduceNgrams("Description"))
    .Append(mlContext.Transforms.NormalizeLpNorm("Description"));
```

`textEstimator` contains a subset of operations performed by the FeaturizeText method. The benefit of a more complex pipeline is control and visibility over the transformations applied to the data.

Using the first entry as an example, the following is a detailed description of the results produced by the transformation steps defined by `textEstimator`:

Original Text: This is a good product

Transform	Description	Result
1. NormalizeText	Converts all letters to lowercase by default	this is a good product
2. TokenizeWords	Splits string into individual words	["this","is","a","good","product"]
3. RemoveDefaultStopWords	Removes stopwords like *is* and *a*.	["good","product"]
4. MapValueToKey	Maps the values to keys (categories) based on the input data	[1,2]
5. ProduceNGrams	Transforms text into sequence of consecutive words	[1,1,1,0,0]
6. NormalizeLpNorm	Scale inputs by their lp-norm	[0.577350529, 0.577350529, 0.577350529, 0, 0]

Feedback

Send feedback about

This product | This page

0 Open | 5 Closed

There are no open issues

View on GitHub