Sample Code

Creating a model from tabular data

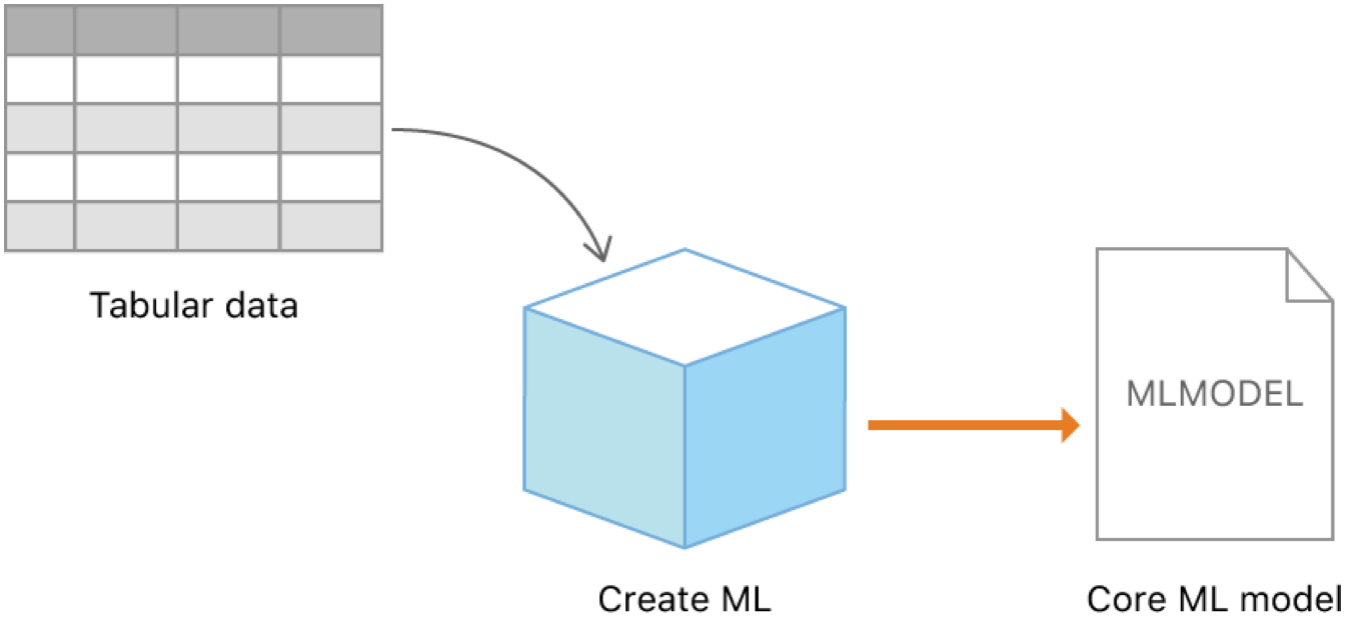
Train a machine learning model by using Core ML to import and manage tabular data.

Download

Xcode 10.1+

Overview

This sample playground uses the Create ML framework to train two Core ML models, a regressor and a classifier.



The playground imports a CSV file, which contains Martian housing data, into a data table. The data table contains the following columns of information about a habitat on Mars:

- Price

- Size (area in acres)
- Number of greenhouses
- Number of solar panels
- Primary purpose

The playground trains the regressor and classifier models, each with a group of columns relevant to that model. Once trained, the regressor is ready to predict the price of a habitat, and the classifier is ready to predict the purpose of a habitat.

The playground concludes by saving each model to a file, ready for integration into an app.

Import the data

Use any `MLDataTable` initializer to import your data in to a data table. In this sample, the playground initializes its first data table with the contents of the CSV file embedded within the playground.

```
/// Create a data table from a CSV file in the playground's `Resources` folder.
let csvFile = Bundle.main.url(forResource: "MarsHabitats", withExtension: "csv")!
let dataTable = try MLDataTable(contentsOf: csvFile)
```

Pass your data table to `print()` to see a pretty-printed sample of its contents in the console.

```
print(dataTable)
```

```
// Prints...
/*
 Columns:
     solar_panels    float
     greenhouses     float
     size            integer
     price           integer
     purpose         string
 Rows: 400
 Data:
 +-------------+-------------+-------------+-------------+-------------+
 | solarPanels | greenhouses | size        | price       | purpose     |
 +-------------+-------------+-------------+-------------+-------------+
 | 5           | 2.5         | 430         | 1500        | farm        |
 | 12          | 3           | 470         | 2990        | general     |
```

```
| 20          | 2           | 460         | 2950        | power       |
| 8           | 3           | 315         | 1990        | farm        |
| 7.5         | 1.5         | 245         | 1500        | general     |
+-------------+-------------+-------------+-------------+-------------+
[400 rows x 5 columns]
*/
```

Isolate the relevant model data

If necessary, generate a new data table that includes only relevant columns for your model.

For example, to predict price, the playground's regressor needs only four columns of the five original columns:

- `price`
- `solarPanels`
- `greenhouses`
- `size`

The playground generates a new data table, specifically tailored for the regressor, by passing an array of column names to the first data table's `subscript(_:)`.

```
let regressorColumns = ["price", "solarPanels", "greenhouses", "size"]
let regressorTable = dataTable[regressorColumns]
```

To predict the purpose of a habitat, the classifier needs a similar group of columns:

- `purpose`
- `solarPanels`
- `greenhouses`
- `size`

```
let classifierColumns = ["purpose", "solarPanels", "greenhouses", "size"]
let classifierTable = dataTable[classifierColumns]
```

Divide the data for training and evaluation

If you intend to evaluate your model after training, reserve some of your data table's rows for evaluation by separating them from the training data. By evaluating your model with data not present for training, your model's evaluation better represents its real-world performance.

The playground creates two data tables per model, one for evaluation and the other for training, by using the `randomSplit(by:seed:)` method of `MLDataTable`. The method returns a tuple of two new data tables, each generated by randomly dividing the rows of the original data table. The size of the first data table in the tuple is determined by the `proportion` parameter, a floating point value between `0.0` and `1.0`. The second data table in the tuple contains the remaining data rows.

In this example, the playground sets aside 20% of each model's data rows for evaluation, leaving the remaining 80% for training.

```
let (regressorEvaluationTable, regressorTrainingTable) = regressorTable.randomSplit(
let (classifierEvaluationTable, classifierTrainingTable) = classifierTable.randomSpl
```

The amount of data your model needs for evaluation versus training will vary with each app. Generally, training your model with more examples leads to better performance.

Train the regressor

The playground trains the regressor by providing its initializer with the training data table and the name of the target column. The `targetColumn` parameter determines what information you want the model to provide in its predictions. The playground tells the regressor to predict the price of a habitat by specifying `price` as the model's target column.

```
let regressor = try MLRegressor(trainingData: regressorTable, targetColumn: "price")
```

During training, Create ML automatically sets aside a small percentage of the training data to use for validating the model's progress during the training phase. The training process gauges the model's performance based on the validation data. Depending on the validation accuracy, the training algorithm can adjust values within the model or even stop the training process, if the accuracy is high enough. Because the split is done randomly, you might get a different result each time you train a model.

Evaluate the regressor

To see how accurately the regressor performed during training and validation, the playground gets the `maximumError` property of its `trainingMetrics` and `validationMetrics` properties.

```
/// The largest distances between predictions and the expected values
let worstTrainingError = regressor.trainingMetrics.maximumError
let worstValidationError = regressor.validationMetrics.maximumError
```

> **Note**
>
> `MLRegressorMetrics` also has a `rootMeanSquaredError` property.

The playground evaluates the regressor's performance by passing its evaluation data table.

```
/// Evaluate the regressor
let regressorEvalutation = regressor.evaluation(on: regressorEvaluationTable)

/// The largest distance between predictions and the expected values
let worstEvaluationError = regressorEvalutation.maximumError
```

If your regressor's evaluation performance isn't good enough, you may need to:

- Retrain with more rows of data.
- Choose a different, specific regressor type (see "Supporting Types" under `MLRegressor`.
- Make other adjustments (see Improving Your Model's Accuracy).

Train the classifier

The playground trains the classifier to predict the purpose of a habitat. It does so by targeting a classifier on the `purpose` column of its training data table.

```
let classifier = try MLClassifier(trainingData: classifierTrainingTable,
                                  targetColumn: "purpose")
```

A classifier can only predict values provided in its training data, unlike a regressor, which can predict numeric values beyond those in its training data. For example, the playground's classifier can only predict a value of `"power"`, `"farm"`, or `"general"` because those are the only values in the `purpose` column.

> **Note**
>
> As an alternative, you may also train a classifier with numerical values instead of textual labels (strings) demonstrated here. As with textual labels, classifiers can only return specific numerical values from its training data, and will not interpolate or extrapolate new values, unlike a regressor.

Evaluate the classifier

To see how accurately the classifier performed during training and validation, the playground gets the `classificationError` property of its `trainingMetrics` and `validationMetrics` properties.

```
/// Classifier training accuracy as a percentage
let trainingError = classifier.trainingMetrics.classificationError
let trainingAccuracy = (1.0 - trainingError) * 100

/// Classifier validation accuracy as a percentage
let validationError = classifier.validationMetrics.classificationError
let validationAccuracy = (1.0 - validationError) * 100
```

Just as with the regressor, the playground evaluates the classifier's performance by passing its evaluation data table.

```
/// Evaluate the classifier
let classifierEvaluation = classifier.evaluation(on: classifierEvaluationTable)

/// Classifier evaluation accuracy as a percentage
let evaluationError = classifierEvaluation.classificationError
let evaluationAccuracy = (1.0 - evaluationError) * 100
```

If your classifier's evaluation performance isn't good enough, you may need to:

- Retrain with more rows of data.
- Choose a different, specific classifier type (see "Supporting Types" under `MLClassifier`).
- Make other adjustments (see Improving Your Model's Accuracy).

Save the model

When you are satisfied with your model's performance, save it to a file for later use in your app. The playground saves the price regressor, along with metadata, to the user's desktop with its `write(to:metadata:)` method.

```
let regressorMetadata = MLModelMetadata(author: "Maria Ruiz",
                                        shortDescription: "Predicts the price of a h
                                        version: "1.0")
/// Save the trained regressor model to the Desktop.
try regressor.write(to: desktopPath.appendingPathComponent("MarsHabitatPricer.mlmode
                    metadata: regressorMetadata)
```

The playground similarly saves the purpose classifier to the user's desktop.

```
let classifierMetadata = MLModelMetadata(author: "Maria Ruiz",
                                         shortDescription: "Predicts the purpose of
                                         version: "1.0")

/// Save the trained classifier model to the Desktop.
try classifier.write(to: desktopPath.appendingPathComponent("MarsHabitatPurposeClass
                     metadata: classifierMetadata)
```

> **Note**
>
> To see the author, version, and description of a model's `MLModelMetadata`, select the model in Xcode's project navigator after you add it to an app.

Add the model to an app

To add your model to an app, see Integrating a Core ML Model into Your App.

See Also

Creating a data table

`init(contentsOf: URL, options: MLDataTable.ParsingOptions) throws`

Creates a data table from an imported JSON or CSV file.

`init(dictionary: [String : any MLDataValueConvertible]) throws`

Creates a data table from a dictionary of column names and data values.

`init(namedColumns: [String : MLUntypedColumn]) throws`

Creates a data table from a dictionary of column names and untyped columns.

`init()`

Creates an empty table containing no rows or columns.

`struct ParsingOptions`

The options for parsing a comma-separated values (CSV) file into a data table for a machine learning model.