Using MLJ

Lesson 2: Model Composition

Authors: Anthony Blaom

Goals

MLJ's design enables flexible **model composition**. Here we learn:

1. What a **composite model** is
2. How to construct model **pipelines**
3. How composite models help us avoid **data leakage**
4. About the **model wrapper**, `TransformedTargetModel`
5. About other model wrappers
6. Other kinds of model composition

Prerequisites

1. **Lesson 1: Basics** (supervised learning, machines, models, evaluation)
2. Prior exposure to common ML pre-processing operations, such as one-hot encoding and standardization
3. Familiarity with cross-validation and the concept of **data leakage**

Getting more help

The **Resources** page linked below contains:

- Slides for this presentation
- Julia code for the demos
- Links to general MLJ learning resources

https://github.com/JuliaAI/MLJ.jl/tree/dev/examples/using_mlj

What is a composite model?

A **composite model** is a model that has other models as hyper-parameters.

```
julia> forest = EnsembleModel(DecisionTreeClassifier())
ProbabilisticEnsembleModel(
  model = DecisionTreeClassifier(
        max_depth = -1,
        min_samples_leaf = 1,
        min_samples_split = 2,
        min_purity_increase = 0.0,
        n_subfeatures = 0,
        post_prune = false,
        merge_purity_threshold = 1.0,
        display_depth = 5,
        feature_importance = :impurity,
        rng = Random.TaskLocalRNG()),
  atomic_weights = Float64[],
  bagging_fraction = 0.8,
  rng = Random.TaskLocalRNG(),
  n = 100,
  acceleration = CPU1{Nothing}(nothing),
  out_of_bag_measure = Any[])
```

`forest.model.max_depth` is a **nested hyper-parameter**.

Kinds of composite models in MLJ

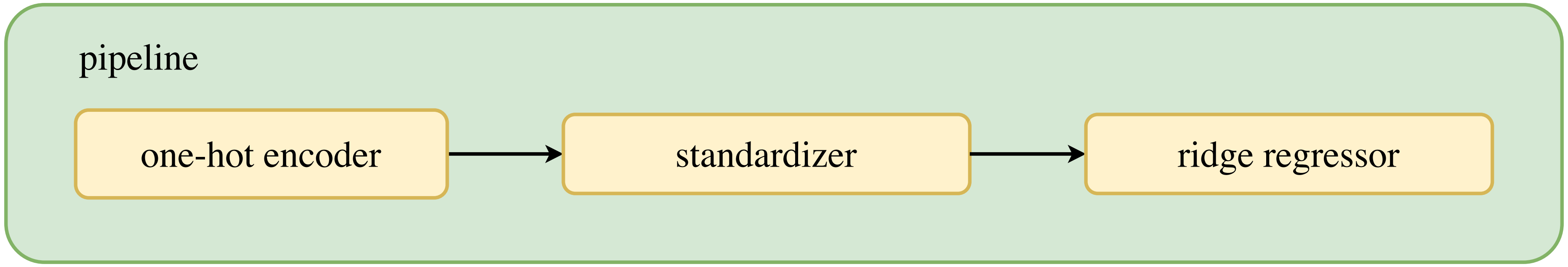
The simplest kinds of model compostion in MLJ:

- **pipelines**
- **model wrappers**

Not discussed here:

- A **model stack** (`Stack`) wraps multiple supervised learners
- **Learning networks** are maximally flexible. Used internally to implement all the above

Model pipelines



Main point: `pipeline` is new, standalone, supervised model, behaving like any other.

For example, you can use `evaluate` to estimate the performance of `pipeline`.

Syntax

```
pipeline = OneHotEncoder() |> Standardization() |> RidgeRegressor()
```

which is syntactic sugar for

```
pipeline = Pipeline(OneHotEncoder(), Standardizer(), RidgeRegressor())
```

which also allows for passing some keyword options.

Data leakage

Why does the following workflow, combining standardization and ridge regression, have **data leakage**?

Step 1. Standardize all the input data:

```
mach = machine(Standardizer(), X) |> fit!
Xstand = transform(mach, X)
```

Step 2. Evaluate the performance of a ridge regressor:

```
evaluate(RidgeRegressor(), Xstand, y, resampling=CV(nfolds=2), measure=rms)
```

Answer: Let `fold1` and `fold2` be the CV folds. When training the ridge regressor on `fold1`, `evaluate` is using data standardized using parameters learned from **all** the data, which includes `fold2`. So `fold2` is a "tainted" dataset, not appropriate for getting an unbiased estimate of the model's performance, which what `evaluate` does to get the first CV score.

Data Leakage

In the following workflow, training on each CV fold includes learning appropriate standardization parameters, but using only data from that fold. So data leakage is avoided:

```julia
pipeline = Standardizer() |>  RidgeRegressor()
evaluate(pipeline, X, y, resampling=CV(nfolds=2), measure=rms)
```

Model wrappers, discussed next, similarly help us to avoid many other sources of data leakage.

A model wrapper for target transformations

Some supervised models perform poorly unless the **target** data is standardized.

Sample task:

1. Learn standarization parameters for target y.
2. Apply standardization to y to get z
3. Train a ridge regressor using some input features X and target z
4. Predict on some new data X_{new} to obtain $\hat{z}$
5. *Inverse* transform $\hat{z}$ to obtain $\hat{y}$.

Notice Steps 2 and 5 both make use of the same learned standardization parameters.

Target transformations

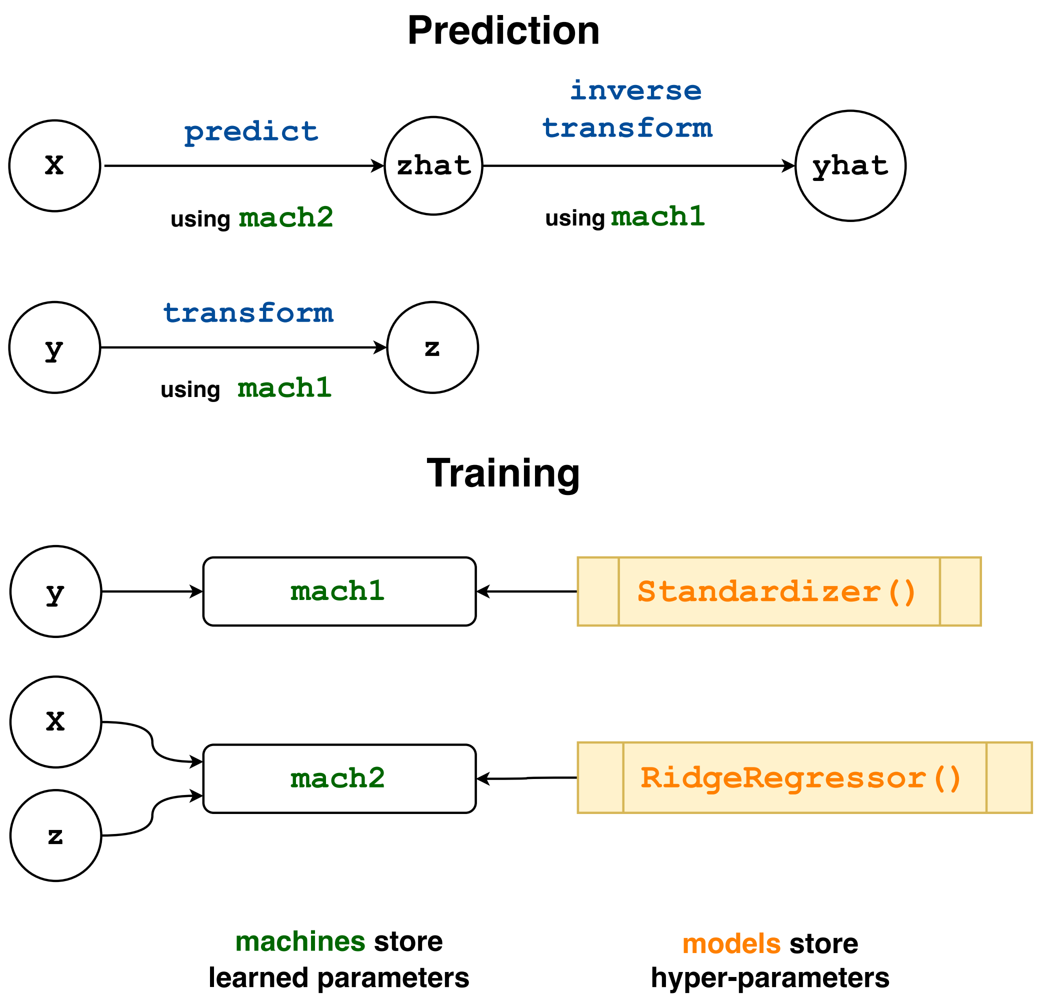
In code:

```
1  pstandardizer = Standardizer()
2  regressor = RidgeRegressor()
3
4  mach1 = machine(standardizer, y) |> fit!
5  z = transform(mach1, y)
6
7  mach2 = machine(regressor, X, z) |> fit!
8  ẑ = predict(mach2, X)
9  ŷ = inverse_transform(mach1, ẑ)
```

The fitted machine `mach1` gets used twice, in lines 10 and 14.

A simple pipeline cannot replicate this workflow.

Target transformations



Target transformations

Model wrappers to the rescue:

```
model = RidgeRegressor()
```

```
wrapped_model = TransformedTargetModel(model, transformer=Standardizer())
```

The `wrapped_model` behaves like `model`, but with target standardization automatically enforced internally, protecting against data leakage.

Live coding

We now demonstrate a supervised learning task making use of both a **pipeline** and the `TransformedTargetModel` **wrapper** to mitigate data leakage.

Other model wrappers in MLJ

- `TunedModel(model)`: for tuning hyperparameters of `model` - see Lesson 3!
- `BalancedModel(model)`: to use `model` in conjunction with oversampling/undersampling algorithms that correct for **class imbalance**
- `EnsembleModel(model)`: to create a **bagged** ensemble of `model` clones (e.g, random forest)
- `IteratedModel(model)`: to wrap an iterative `model` in various iteration controls or callbacks, such as **early stopping** criteria and live inspection of training losses
- `BinaryThresholdPredictor(model)`: for converting a probabilistic predictor into a deterministic one, given a threshold probability for the "positive" outcome.
- `RecursiveFeatureElimination(model)`: for selecting features based on rankings of a supervised `model` that reports feature importances (wrapped model is a transformer)