Using MLJ
Lesson 3: Model Tuning

Authors: Anthony Blaom

Goals

Model **tuning** (or **optimization**) refers to the process of choosing the “best” hyper-parameters of a model. The same principles apply to choosing between *different* models (algorithms).

Here we learn:

1. How to generate a **learning curve**, a visual tool for tuning a **single** model hyperparameter.
2. How to conceptualize model tuning as a **model wrapper**.
3. How to use the `TunedModel` wrapper in practice, to tune **multiple** parameters while guarding against data leakage.
4. How to use `TunedModel` to choose from a **list** of models, of possibly varying type.
5. Why tuning-as-wrapper can lead to **nested resampling**.

Prerequisites

1. **Lesson 1: Basics** (supervised learning, machines, models, evaluation)
2. **Lesson 2: Model Composition** (pipelines and model wrappers)
3. Previous exposure to random forest models helpful.

Getting more help

The **resources page** 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

Live coding

We will now demonstrate the use of MLJ **learning curves** to examine the effect of a single hyper-parameter.

Tuning as model wrapper

We want to automate the process of optimizing one or more hyper-parameters.

Tuning as model wrapper

In MLJ we wrap a supervised `model` in a “tuning strategy” like this:

```
tuned_model =
TunedModel(model; ranges=..., resampling=CV(), measure=..., ...)
```

Fitting

Fitting `tuned_model` to some data does this:

1. Construct a sequence of `model` mutations `model1`, `model2`, `model3`, … , of `model` with hyper-parameters varying over the specified `range`.
2. Calls `evaluate` on each model mutant, using the specified `resampling` strategy and `measure` to get **out-of-sample** estimates of performance
3. Identifies the model with the best performance
4. **Retrains** the best model **using all supplied data**.

Tuning as model wrapper

What does this mean for new predictions?

```julia
mach = machine(tuned_model, X, y) |> fit!
ŷ₁ = predict(mach, Xnew)
```

```julia
best_model = report(mach).best_model
mach = machine(best_model, X, y) |> fit!
ŷ₂ = predict(mach, Xnew)
```

Then `ŷ₁ == ŷ₂`.

Tuning as model wrapper

You can think of `tuned_model` as a “self-tuning” counterpart of `model`.

More precisely, in `tuned_model` the hyper-parameters specified by `range` have been transformed from **hyper-parameters** to **learned parameters**.

Warning

Adding to the learned parameters **adds complexity** and whence the risk of over-fitting.

Live coding

We now demonstrate the use of the `TunedModel` wrapper.

Nested resampling

In our live coding we:

1. Wrapped a model called `pipe` in a tuning strategy:

```
tuned_pipe = TunedModel(pipe, resampling=CV(nfolds=4), ...)
```

Here the `CV` is called **inner resampling** .

2. Evaluated the wrapped model:

```
evaluate(tuned_pipe, X, y; resampling=CV(nfolds=3))
```

Here the `CV` is called **outer resampling** .

Nested resampling

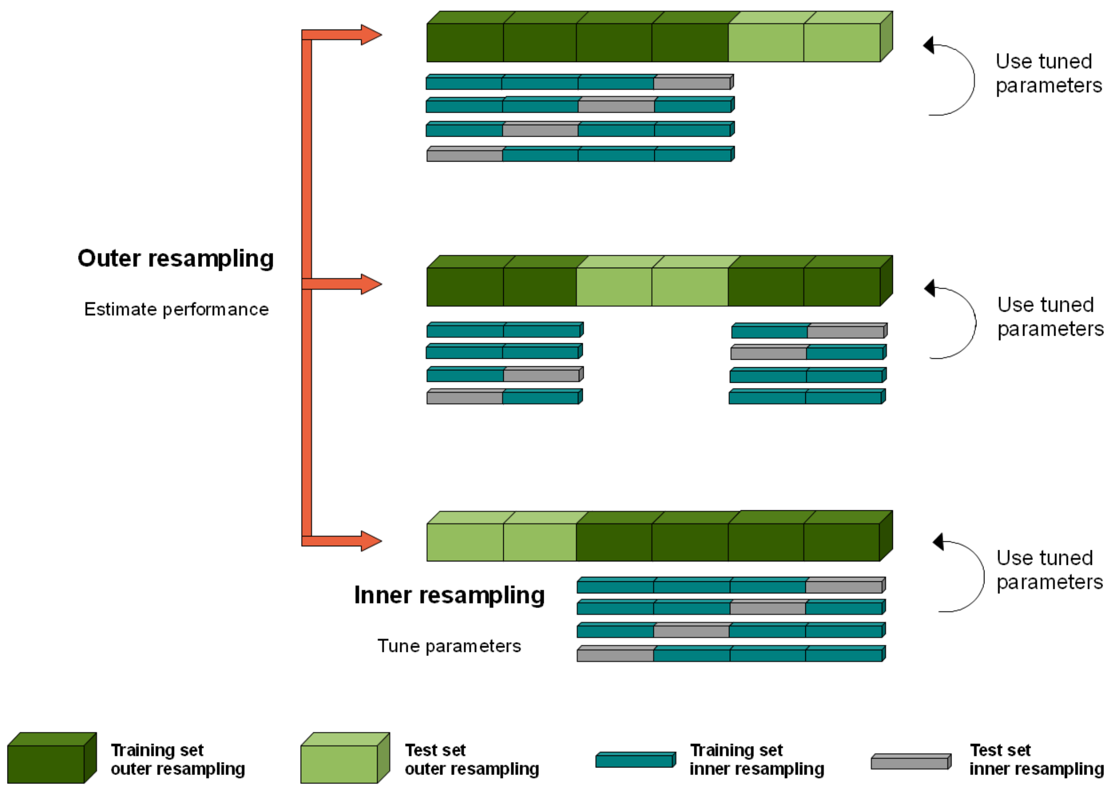

Image source: The `mlr` project.

Final observations

- It's important to remember that a `TunedModel` is *not* a model + pre-optimized hyperparameters. Hyperparameters are not optimized until training the model. In particular, the optimized hyperparameters generally assume *different* values on each training fold in CV estimation of the model's performance.
- While the wrapper approach is the safest way to avoid common data leakage pitfalls in model tuning, nested resampling can make it computationally expensive, especially when used in conjunction with other model wrapping algorithms, such as `IteratedModel` or `Stack`. In those cases, one may decide to "freeze" the hyper-parameters by extracting `best_model` from `report(mach)`.