The Alan Turing Institute

MLJ
A machine learning toolbox for julia

Anthony Blaom et al.

The Alan Turing Institute is the national centre for data science, headquartered at the British Library.

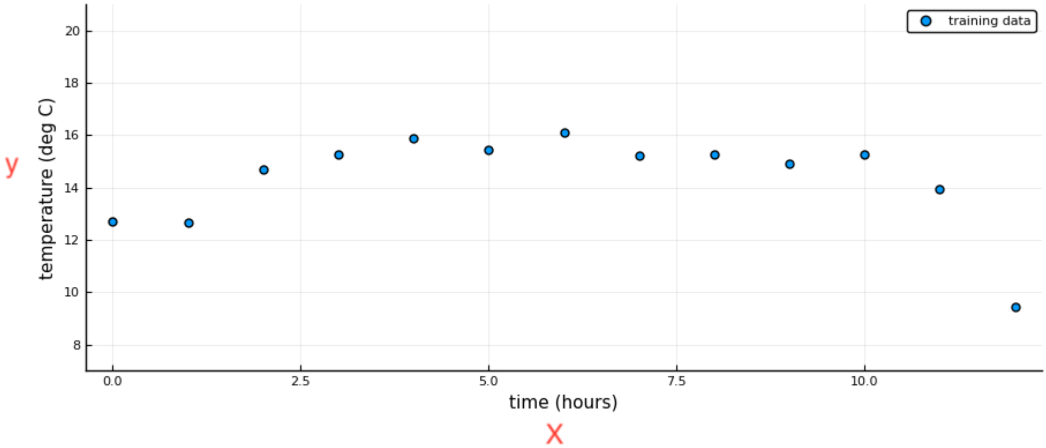
NeSI
New Zealand eScience Infrastructure

Julia computing

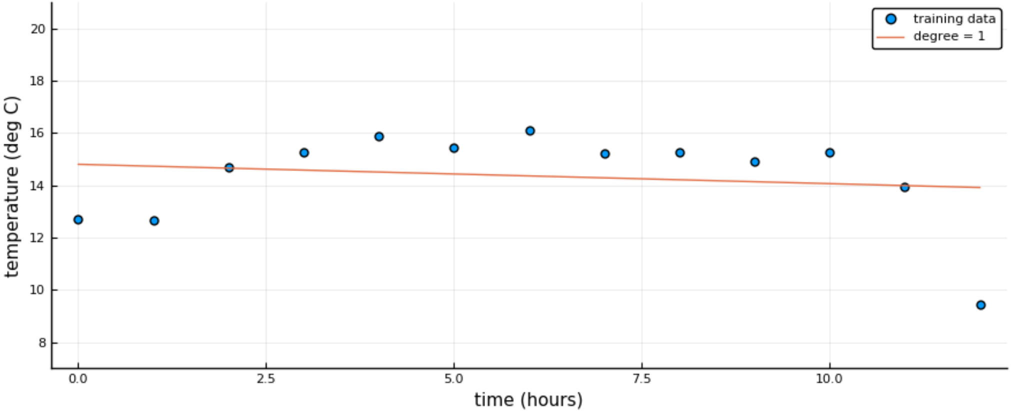
Supervised Learning

Learning to **predict** some target variable **y** from a knowledge of some other variables **X** (the *input features*).

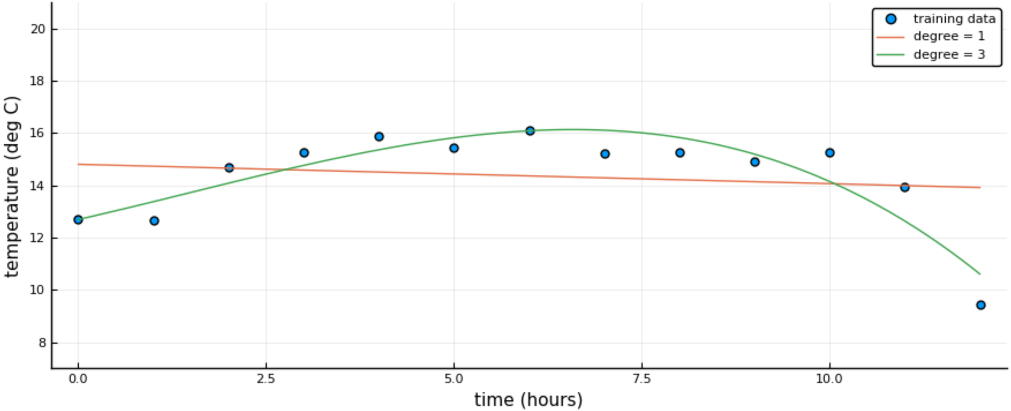
Supervised Learning



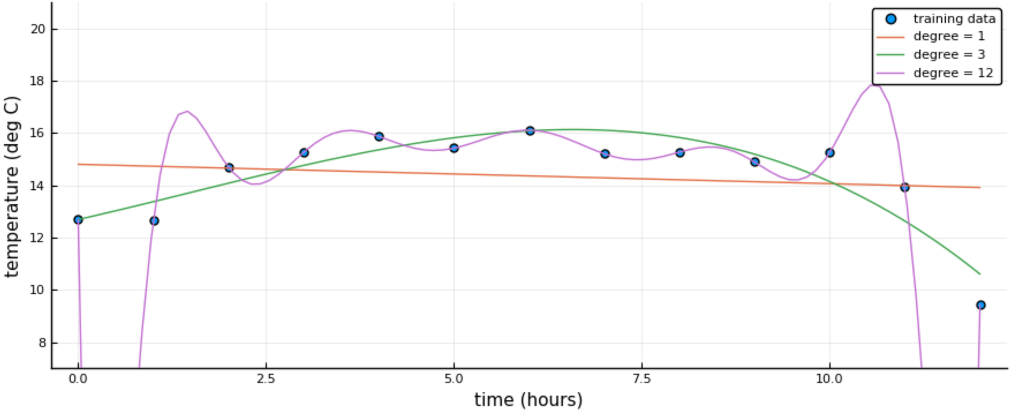
Supervised Learning



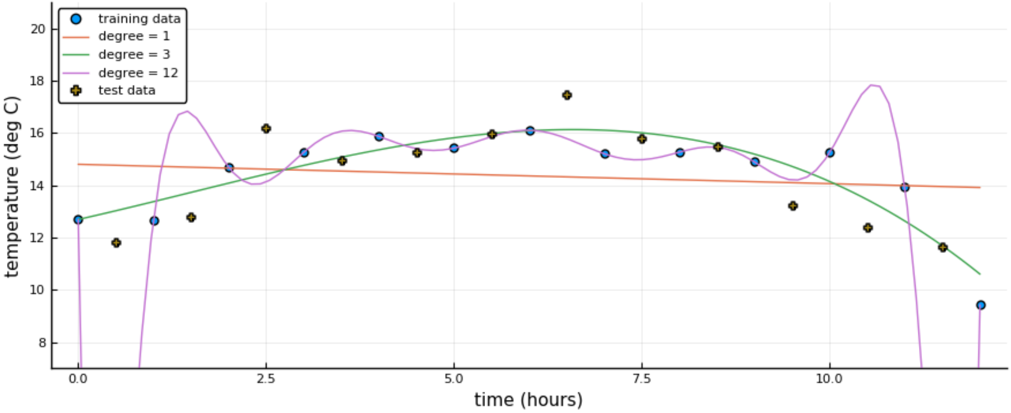
Supervised Learning



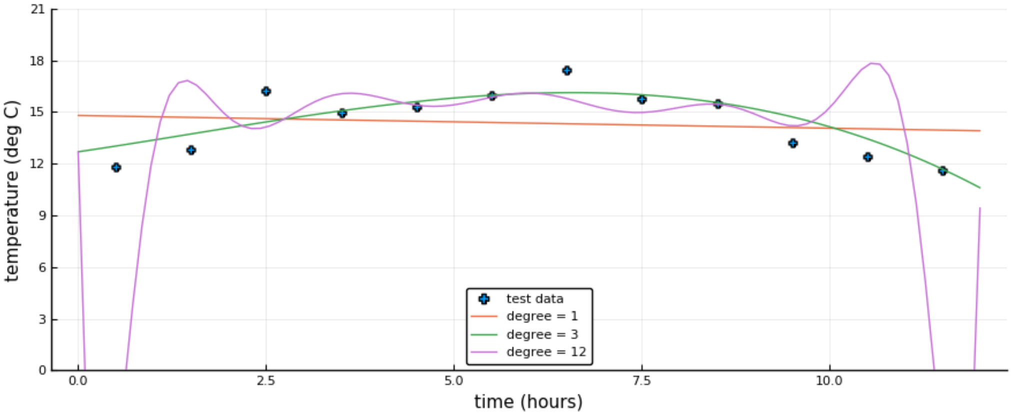
Supervised Learning



Supervised Learning

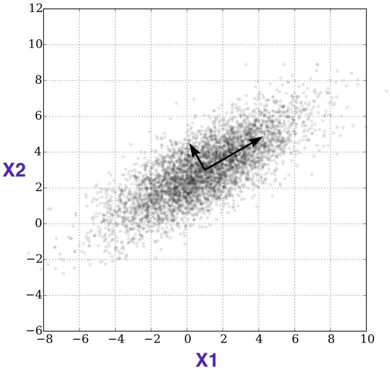


Supervised Learning



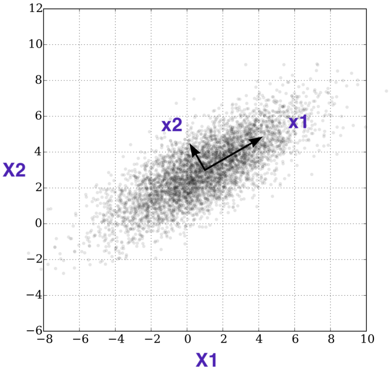
Unsupervised Learning

Learning data **transformations**, e.g., dimension reduction

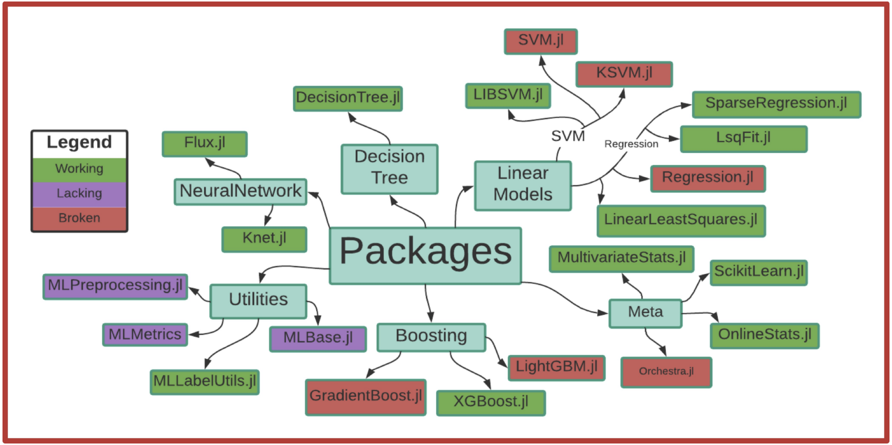


Unsupervised Learning

Learning data **transformations**, e.g., dimension reduction



A plethora of models



Machine learning toolboxes

A machine learning toolbox:

Machine learning toolboxes

A machine learning toolbox:

- Provides a **uniform interface** for **fitting**, **evaluating**, **tuning** and **benchmarking** models.

Machine learning toolboxes

A machine learning toolbox:

- Provides a **uniform interface** for **fitting**, **evaluating**, **tuning** and **benchmarking** models.
- Provides common **preprocessing tasks** (such as data cleaning and type coercion)

Machine learning toolboxes

A machine learning toolbox:

- Provides a **uniform interface** for **fitting**, **evaluating**, **tuning** and **benchmarking** models.
- Provides common **preprocessing tasks** (such as data cleaning and type coercion)
- Allows for model **composition** (aka pipelining)

Toolboxes in other ecosystems

Goals for MLJ

Goals for MLJ

– Want **usability**, interoperability, extensibility and reproducibility

Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task

Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code

Model search and tasks

```julia
using MLJ
```

```julia
models()
```

```
Dict{Any,Any} with 9 entries:
  "MultivariateStats" => Any["ICA", "RidgeRegressor", "KernelPCA", "PCA"]
  "MLJ"               => Any["MLJ.Constant.DeterministicConstantRegressor", "ML…
  "DecisionTree"      => Any["DecisionTreeRegressor", "DecisionTreeClassifier"]
  "ScikitLearn"       => Any["SVMLRegressor", "SVMNuClassifier", "ElasticNet", …
  "LIBSVM"            => Any["EpsilonSVR", "LinearSVC", "NuSVR", "NuSVC", "SVC"…
  "Clustering"        => Any["KMeans", "KMedoids"]
  "GLM"               => Any["OLSRegressor", "GLMCountRegressor"]
  "NaiveBayes"        => Any["GaussianNBClassifier", "MultinomialNBClassifier"]
  "XGBoost"           => Any["XGBoostCount", "XGBoostRegressor", "XGBoostClassi…
```

Model search and tasks

```
task = load_boston()
models(task)
```

```
Dict{Any,Any} with 6 entries:
  "MultivariateStats" => Any["RidgeRegressor"]
  "MLJ"               => Any["MLJ.Constant.DeterministicConstantRegressor", "ML…
  "DecisionTree"      => Any["DecisionTreeRegressor"]
  "ScikitLearn"       => Any["SVMLRegressor", "ElasticNet", "ElasticNetCV", "SV…
  "LIBSVM"            => Any["EpsilonSVR", "NuSVR"]
  "XGBoost"           => Any["XGBoostRegressor"]
```

Quick performance evaluation

```julia
@load DecisionTreeRegressor # load code

tree_ = DecisionTreeRegressor(n_subfeatures=3)
tree = machine(tree_, task)
evaluate!(tree,
          resampling=Holdout(fraction_train=0.7),
          measure=[rms, mav])
```

Quick performance evaluation

```
@load DecisionTreeRegressor # load code

tree_ = DecisionTreeRegressor(n_subfeatures=3)
tree = machine(tree_, task)
evaluate!(tree,
          resampling=Holdout(fraction_train=0.7),
          measure=[rms, mav])

    (MLJ.rms = 8.795939100833767,
     MLJ.mav = 5.785953164160401,)
```

Meta-algorithms as model wrappers

```
forest_ = EnsembleModel(atom=tree_, n=10)
```

Meta-algorithms as model wrappers

```
forest_ = EnsembleModel(atom=tree_, n=10)

r1 = range(forest_, :bagging_fraction, lower=0.4, upper=1.0);
r2 = range(forest_, :(atom.n_subfeatures), lower=1, upper=12)

self_tuning_forest_ = TunedModel(model=forest_,
                                 tuning=Grid(),
                                 resampling=CV(),
                                 ranges=[r1,r2],
                                 measure=rms)
```

Meta-algorithms as model wrappers

```
self_tuning_forest = machine(self_tuning_forest_, task)

evaluate!(self_tuning_forest,
          resampling=CV(),
          measure=[rms,rmslp1])
```

```
(MLJ.rms = [2.91827, 3.40544, 4.60971, 4.54709, 8.12081, 3.79819],
 MLJ.rmslp1 = [0.148546, 0.119118, 0.148812, 0.134863, 0.345141, 0.221093],)
```

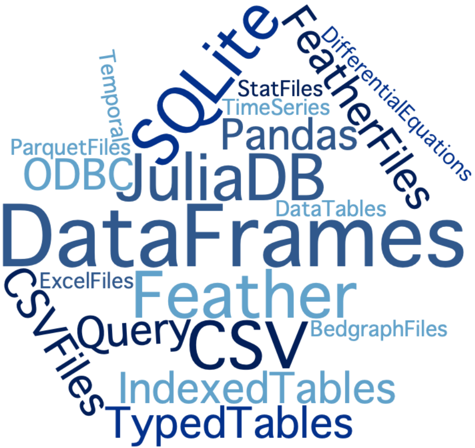
Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code

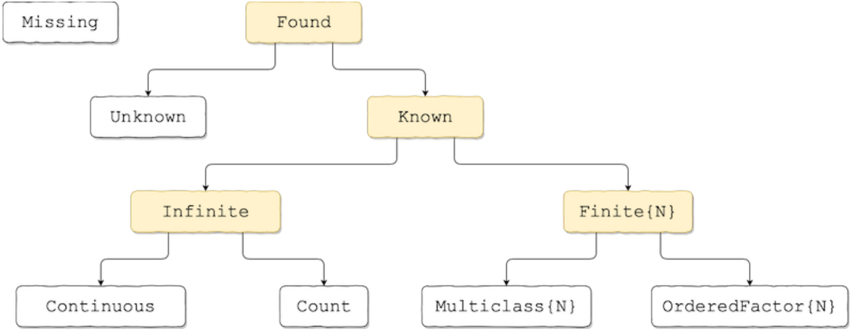
Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format

Use any tabular data format



Scientific types



Categorical data

categorical $\neq$ integer

Categorical data

categorical $\neq$ integer

$$\mathrm{data} = [1, 2, 2, 2, 1, 2, 1, 1, 3, 2]$$

$$\mathrm{train} = [1, 2, 2, 2, 1] \quad \mathrm{eval} = [1, 3, 2]$$

Categorical data

categorical $\neq$ integer

$$\text{data} = [1, 2, 2, 2, 1, 2, 1, 1, 3, 2]$$

$$\text{train} = [1, 2, 2, 2, 1] \quad \text{eval} = [1, 3, 2]$$

MLJ expects `CategoricalArray.CategoricalValue` for categoricals.

Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format

Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format
  - Probabilistic predictions (evaluation, inconsistent representations, ...)

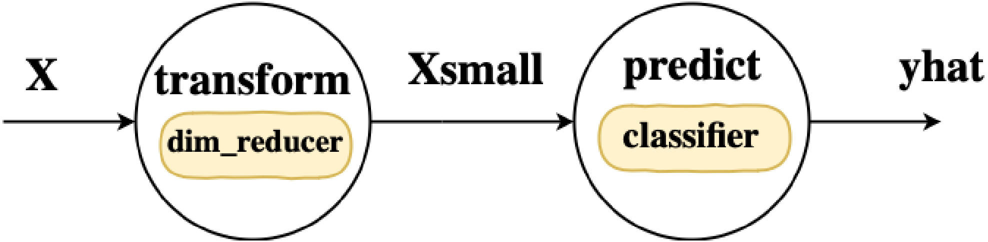
Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format
  - Probabilistic predictions: evaluation, inconsistent representations
  - Limitations of **model composition** API

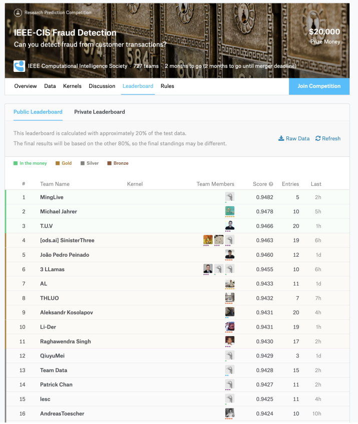
Goals for MLJ

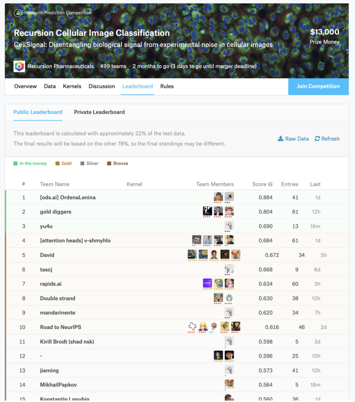
- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format
  - Probabilistic predictions: evaluation, inconsistent representations
  - Limitations of **model composition** API — barrier to innovation!

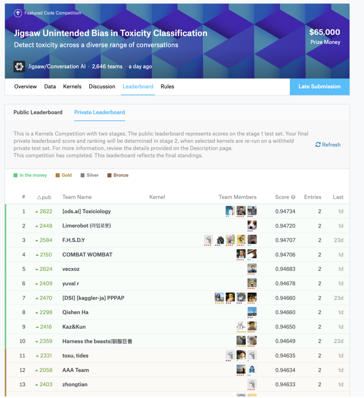
Model composition (aka pipelining)



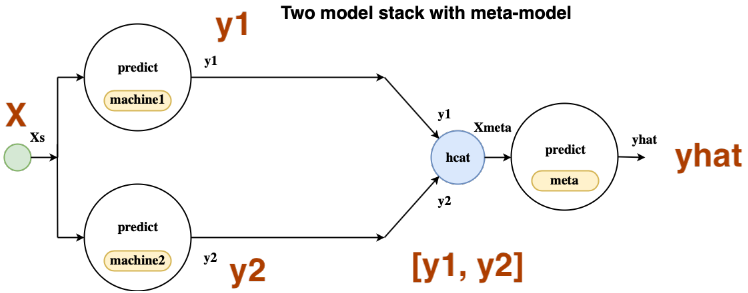
Data science competitions (kaggle)



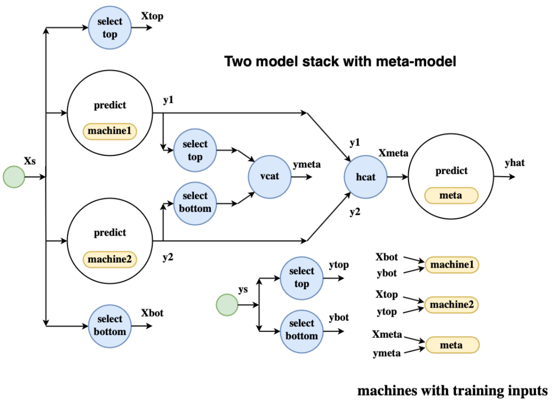




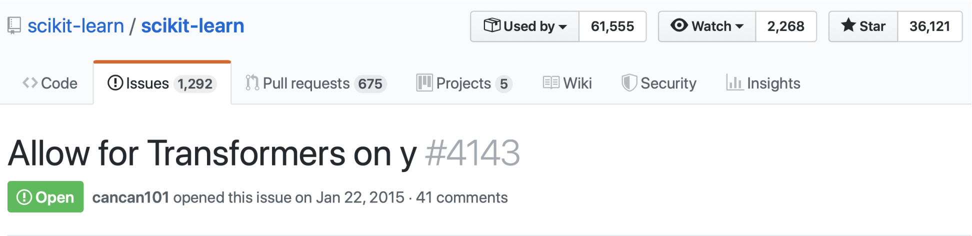
More complicated example



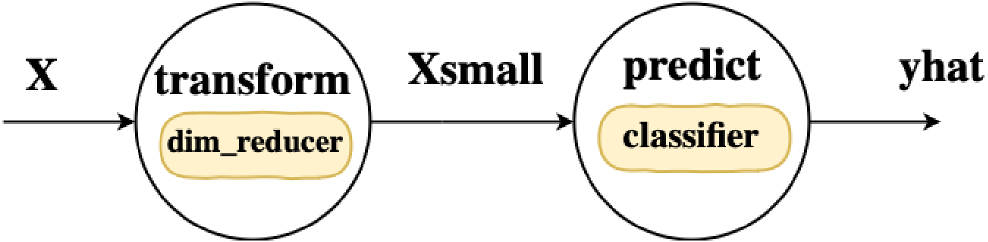
More complicated example



Target transformations



Model composition (aka pipelining)



Model composition (aka pipelining)

Compact syntax for linear pipeline

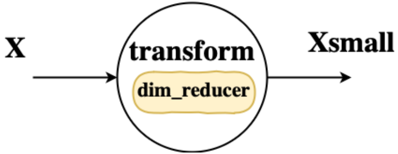
```
composite_ = @pipeline dim_reducer_ classifier_
```

Compact syntax for linear pipeline

```
composite_ = @pipeline dim_reducer_ classifier_
```

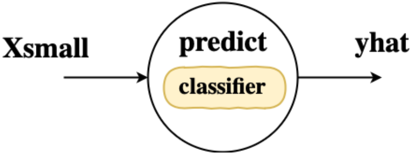
Does not generalize!

The dimension reducer




```
dim_reducer_ = PCA()
dim_reducer = machine(dim_reducer_, X)
fit!(dim_reducer)
Xsmall = transform(dim_reducer, X);
```

The classifier




```
classifier_ = SVC()
classifier = machine(classifier_, Xsmall, y)
fit!(classifier)
ŷ = predict(classifier, Xsmall)
```

Summary of unstreamlined workflow

```
dim_reducer_ = PCA()
dim_reducer = machine(dim_reducer_, X)
fit!(dim_reducer)
Xsmall = transform(dim_reducer, X);

classifier_ = SVC()
classifier = machine(classifier_, Xsmall, y)
fit!(classifier)
ŷ = predict(classifier, Xsmall)
```

Refactoring as learning network: Step 1

```julia
X = source(X)
y = source(y)

dim_reducer_ = PCA()
dim_reducer = machine(dim_reducer_, X)
fit!(dim_reducer)
Xsmall = transform(dim_reducer, X);

classifier_ = SVC()
classifier = machine(classifier_, Xsmall, y)
fit!(classifier)
ŷ = predict(classifier, Xsmall)
```

Refactoring as learning network: Step 2

```julia
X = source(X)
y = source(y)

dim_reducer_ = PCA()
dim_reducer = machine(dim_reducer_, X)
Xsmall = transform(dim_reducer, X);

classifier_ = SVC()
classifier = machine(classifier_, Xsmall, y)
ŷ = predict(classifier, Xsmall)
```

Training a learning network

```
X = source(X)
y = source(y)

dim_reducer_ = PCA()
dim_reducer = machine(dim_reducer_, X)
Xsmall = transform(dim_reducer, X);

classifier_ = SVC()
classifier = machine(classifier_, Xsmall, y)
ŷ = predict(classifier, Xsmall)

fit!(ŷ)
```

Prediction in a learning network

```julia
ŷ(rows=3:4)
```

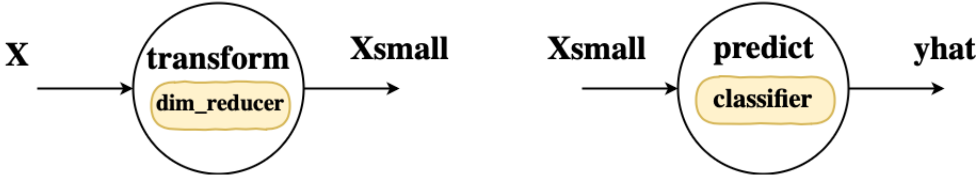
```
2-element Array{CategoricalString{UInt8},1}:
 "versicolor"
 "versicolor"
```

Prediction in a learning network

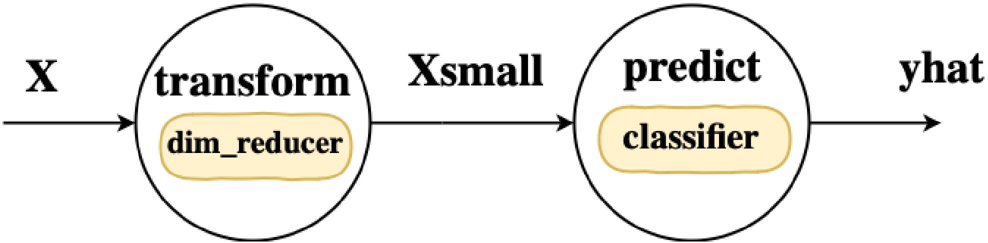
```
Xnew = (SepalLength = [4.0, 5.2],
        SepalWidth = [3.2, 3.0],
        PetalLength = [1.2, 1.5],
        PetalWidth = [0.1, 0.4],)
ŷ(Xnew)
```

```
2-element Array{CategoricalString{UInt8},1}:
 "setosa"
 "setosa"
```

Summarizing



Summarizing



Still a need stand-alone model!

Macro to the rescue

```
composite = @from_network Composite(pca=dim_reducer_, svc=classifier_) <= (X, y, ŷ)
```

Composite is now a model like any other

```
composite_ = @from_network Composite(pca=dim_reducer_, svc=classifier_) <= (X, y, ŷ)

composite = machine(composite_, X2, y2)
fit!(composite)
predict(composite, Xnew)
```

Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format
  - Probabilistic predictions: evaluation, inconsistent representations
  - Limitations of **model composition** API

Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format
  - Probabilistic predictions: evaluation, inconsistent representations
  - Limitations of **model composition** API — barrier to innovation!

Goals for MLJ

- Want **usability**, interoperability, extensibility and reproducibility
- Want avoid common **pain-points**:
  - Identifying all models that solve a given task
  - Routine operations requiring a lot of code
  - Passage from data source to algorithm-specific data format
  - Probabilistic predictions: evaluation, inconsistent representations
  - Limitations of **model composition** API — barrier to innovation!
- Hope that project adds some focus to Julia ML development more generally

Road map

Road map

Enhancing functionality: Adding models

- Wrap the scit-learn (python/C) models (Z. Nugent, D. Arenas)
- **Flux.jl** deep learning (A. Shridhar)
- **Turing.jl** probabilistic programming (M. Trapp)
- **Geostats.jl** (J. Hoffimann)
- Data cleaning? Feature engineering (featuretools?)

Road map

Enhancing functionality: Adding models

- Wrap the scit-learn (python/C) models (Z. Nugent, D. Arenas)
- **Flux.jl** deep learning (A. Shridhar)
- **Turing.jl** probabilistic programming (M. Trapp)
- **Geostats.jl** (J. Hoffimann)
- Data cleaning? Feature engineering (featuretools?)

Road map

Enhancing core functionality

- Systematic benchmarking
- More comprehensive performance evaluation
- Tuning using Bayesian optimization
- Tuning using gradient descent and AD
- Iterative model control

Road map

Broadening scope

- Extend or supplement LossFunctions.jl
- Add sparse data support (NLP)
- Time series

Road map

Scalability

- Online learning support and distributed data
- DAG scheduling (J. Samaroo)
- Automated estimates of cpu/memory requirements

github.com/alan-turing-institute/**MLJ.jl**

Resources for this talk: examples/JuliaCon2019/

Core design: Anthony Blaom, Franz Kiraly, Sebastian Vollmer

Lead contributor: Anthony Blaom

Julia language consultants: Mike Innes, Avik Sengupta

Other contributors, past and present: Dilum Aluthge, Diego Arenas, Edoardo Barp, Gergö Bohner, Michael K. Borregaard, Valentin Churavy, Harvey Devereux, Mosè Giordano, Thibaut Lienart, Mohammed Nook, Piotr Oleśkiewicz, Julian Samaroo, Ayush Shridar, Yiannis Simillides, Annika Stechemesser

turing.ac.uk
@turinginst