# Table of Contents


**Chapter 1: Machine Learning** 1

**Your bookmark** 1

- **Introduction** 1
  - Taxonomies 1
  - The Query-Verse 3
- **Statistical Modelling** 5
  - Loess 6
  - Loss Functions 7
  - Clustering 9
  - Decision Trees 13
- **Bayesian methods** 15
  - Inference 16
    - Naive Bayes 17
    - ApproxBayes 19
  - Markov  Chain Monte Carlo (MCMC) 22
    - Sampling Algorithms 22
      - MCMCChain 23
    - Mamba 25
    - STAN 30
      - JAGS 32
      - Turing 32
- **ML frameworks** 32
  - ML Datasets 33
    - MNIST 34
  - Flux 35
    - Model Zoo 38
    - MNIST 39
    - Metalhead 40
  - Knet 43
    - Tutorial examples 44
  - Tensor Flow 48
- **Summary** 49

**Index** 51

# 1
# Machine Learning

## Introduction

This a new chapter. When the previous version of book was written, Julia version was v0.3/0.4 and there were few (or no) packages seriously targetting at Machine Learning (ML). Situation very different now with some of the heavyweight developers/groups working on approaches to ML

It used to be thought that scientific programning and latterly data science, would represents the kill-app for Julia but now with Julia's compilation to executable code, ability to address GPU (and now TPU) architectures and easily accommodated parallel and networked computing, it is seen in some area that machine learning corresponds to th e true home for Julia. This is now to suggest that the former area are to be neglected, we have seen in previous chapters that the converse is the case.

So we will need to review Julia's approaches to machine learning and this presents a problem in a number of ways.

1. More than any other topics in this book, to discuss the theory(s) underpinning machine learning require a complete text rather than merely a single chapter.
2. A number of packages have come (and gone); some choices seem to have emerged with the production of version 1: i.e. from those which are v1.+ compilant and being actively being worked on.
3. Many packages cut across differing ML approaches which makes individual classifications *tricky.*

In the spirit of this book I have chosen to some packages which seem to be currently important, with of simple examples, and to refer the reader to the documentation and examples, together with online presentations, which are usually very extensive.

# Taxonomies

The concept of teaching machines to learn has occurred is advances in computing hardware and also changes in programming paradigms from conventional imperative coding to parallelism and alternative approaches such as the use of artificial neural networks (ANNs).

Machine learning can be divided into two types:

- Supervised
- Unsupervised

In supervised learning, we start with a dataset that contains training data, where each the outcome (target) of data is known. An example might be facial recognition which we wish to classify be gender or number plate recognition, the bain of my life living within the London congestion zone. The aim of the ML system is to predict future datasets tased on which can be *learned* for the training data; normally the training data is split into 2 groups, say by 90% to 10%, and the latter 10% is used to determine how well our system are learnt.

This is done algorithmically; the task being to identify (fit) a function (f) from inputs (X) to outputs (Y), so that `Y = f(X)`. Once this is determined it can be used on new data (that is, new inputs that are not labeled) to predict the output. Depending on the type of output that needs to be computed, if the outputs are to be assigned to categorical values (like male and female) then we are dealing with a classification algorithm. Alternatively, if the type of outputs are numeric value, the problem is one of regression.

With unsupervised machine learning the situation is very different. There are inputs, but no outputs. In such cases, we use the learning dataset to train our system, the main goal is, generating different clusters of inputs and being able to assign new data to the most appropriate cluster.

For supervised learning, linear or logistic regression, decision trees, or neural networks are often used; with unsupervised learning, the choice may be k-means clustering or *apriori* algorithms.

It is also possible to identify a couple of (sub-) classes in common usage:

- Semi-supervised
- Reinforcement

The names are reasonably suggestive but do, in fact, correspond to the way that most individuals are taught. Semi-supervised, represents a mixture of the two principle approaches, and are applicable when our data contains both known and unknown

outcomes.

Reinforcement learning is somewhat different; the algorithm (in this case) is rewarded about the success of previous decisions. and based on this, the algorithm modifies its strategy in order to maximise outcomes.

Semi-supervision is often the way that children (and ourselves) usually learn, whereas reinforcement is often used when training animals (i.e. Pavlovian).

# The Query-Verse

Before plunging into machine learning *per se*, I want to introduce an important topic which has not yet been discussed: David Anthoffs' group's work at Berkeley on the Queryverse (QV).

We saw in chapter 8, that previous comments on Julia's approaches to visualising data have largely been dispelled with the work on the Plots API (with its associated backends) and on the Gadfly ggplot2 style package; further being enhance by frameworks such as StatsPlots, Makie, VegaLite etc.

On the other-hand data(sets) can come in a variety of forms, all of which need to be handled by Julia. Some (but not all) of these may comprise data sources from: DataFrames.jl, Pandas.jl, IndexedTables.jl, JuliaDB.jl, TimeSeries.jl, Temporal.jl, CSVFiles.jl, ExcelFiles.jl, FeatherFiles.jl, etc., and it is a central aim of the Queryverse framework to be able to read and then (as the name suggests) query these.

This is vital in the understanding of any dataset, prior to selecting and proceeding with its analysis. Further it might be necessary to *clean the data*, a sometimes dubious activity, but at least to eliminate any cases where data is incomplete or missing, as in such instances many procedures may fail.

Further more the results of a query can be required to be output (and saved) into a range of different data structures. In addition to those employed by the the *inputter* packages, these may also be: iterators, dictionaries, tuples, regular and sparse arrays

The 'main' package in the QV is Query.jl, which relies on the various data providers to handle its inputs and outputs. Query.(jl) makes extensive use of Julia's macro features to exhibit a user interface similar to .NET's Linq approach and hence to SQL syntax.

The following is a query of the by now well-trodden IRIS package from Datasets

```
julia> using RDatasets, Query
julia> iris = RDatasets.dataset("datasets","iris")
```

```
5 rows × 5 columns
```

| | SepalLength | SepalWidth | PetalLength | PetalWidth | Species |
|---|---|---|---|---|---|
| | Float64 | Float64 | Float64 | Float64 | Categorical… |
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 3 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 4 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 5 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

The following is a simple select of the sepal dimensions and species types, excluding the type: virginica and for flowers with a sepal width of 3.0 cm or more.

Notice the different ordering of the statements (macros) here as we would expect in SQL. The unseating (@where) and ordering (@orderby) precede the query (@select). The final macro @collect is used to specify the output type.

```
julia> tp = @from df in iris begin
    @where df.SepalWidth >= 3.0 and df.Species != "virginica"
    @orderby df.Species, df.SepalLength
    @select df.SepalLength, df.SepalWidth, df.Species
    @collect
end

julia> tp[1]
(4.3, 3.0, CategoricalString{UInt8} "setosa")
```

Because QV handles a wide variety of Julia data types the baseline collection is an array of tuples:

```
julia> typeof(tp)
Array{Tuple{Float64,Float64,CategoricalString{UInt8}},1

julia> size(tp)
(93,)
```

The index corresponds to the row in the array and since a tuple is returned we can add an additional index to reference the individual data values.

```
julia> for i in 1:5
   @printf "%s :: %.2f %.2f\n" tp[i][3] tp[i][1] tp[i][2]
end
setosa :: 4.30 3.00
setosa :: 4.40 3.00
setosa :: 4.40 3.20
setosa :: 4.60 3.10
```

```
setosa :: 4.60 3.40

# This is not quite the same as above because the data is sampled and
ordered.
```

Query also provides an additional syntax using the pipe (`|>`) operator .

```
julia> df = iris |> @select(1:2,5) |>
   @filter(_.SepalWidth>=3.0) |>
   @filter(_.Species!="setosa") |>
   @take(5) |> DataFrame
10 rows × 3 columns
```

| | SepalLength Float64 | SepalWidth Float64 | Species Categorical… |
|---|---|---|---|
| 1 | 7.0 | 3.2 | versicolor |
| 2 | 6.4 | 3.2 | versicolor |
| 3 | 6.9 | 3.1 | versicolor |
| 4 | 6.3 | 3.3 | versicolor |
| 5 | 5.9 | 3.0 | versicolor |

Here I have excluded the data on "setosa", which is getting a little repetitive.

Note: that with this syntax the `@where` clause is replaced by one (or more) `@filter` macros and the size of the dataset returned specified by using `@take`

Query has a number of additional macros for table joining (`@join`), grouping (`@group`) and creation of local variable (`@let`) in order to facilitate complex queries. Its ability to output to data-frames and indexed tables makes it ideal for data *munging* and cleaning.

In addition the QV promotes the use of VegaLite.jl, a *Grammar of Graphics style* API, and of DataVoyager.jl to interactively explore datasets.

# Statistical Modelling

Classical approaches to ML have relied on application of statistics. Before emergence of high speed CPU's, and computing via the GPU, these where the principal tools. and in those days the term data-mining was in widespread use rather than machine learning.

Is there a difference between statistical modelling and of machine learning, there is some debate even now. Julia ecosphere, at present isa little confused too: JuliaStats describes itself as *Statistics and Machine Learning made easy in Julia* and the packages listed here are all pretty well mainstream SM. However there is a Machine Learning group (JuliaML) and

packages which tackle problems considered more to be mainstream ML are listed here.

Somewhere (in-between) are the Bayesians which we will discuss a little later

It is worth mentioning the existence of the meta-package Statskit, which loads a set of essential packages for statistics in one command Pkg command. Some of these we have met already, other will be discussed here but some will not.

# Loess

To demonstrate the use of a simple model fitting approach I will look at the Loess package, which is essentially is local regression/smoothing technique which uses non-parametric methods to classify data.

These are based on k dimensional (*kd*) tree based approximation and are used in the *netlib* loess (C/Fortran) code, and both in Python and R's loess routines; the Julia module is implemented in native code and is not a wrapper around netlib.

We will generate a random variate in to range [0,10], consisting of 50 values, and map this against a known function to which considerable noise as been added.

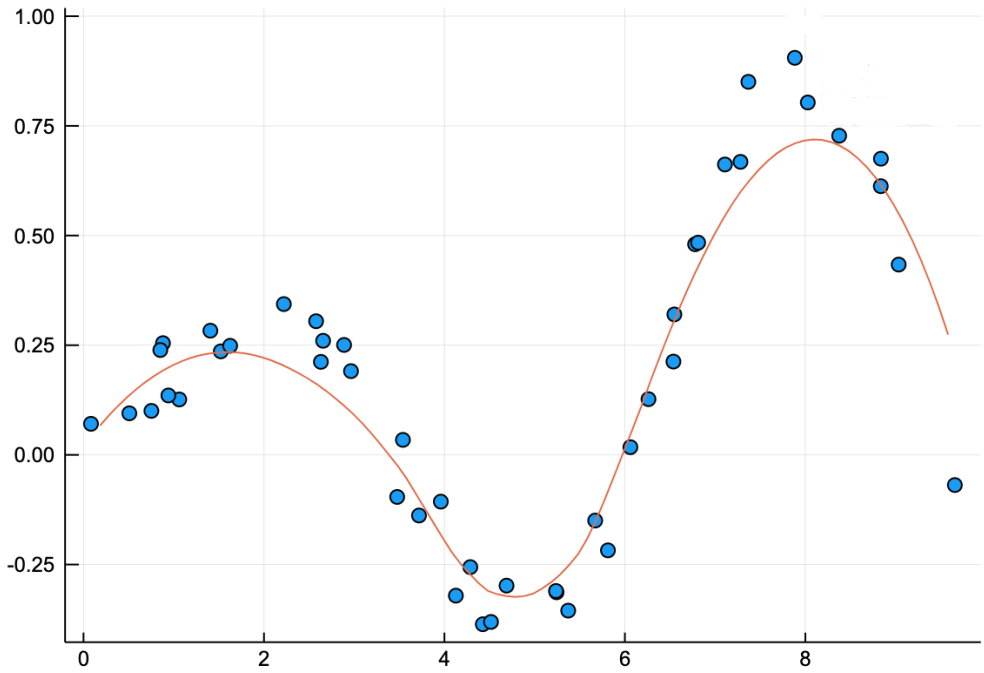
```
julia> using Plots, Loess
julia> gr()
Plots.GRBackend()

julia> x = 10 .* rand(50);
julia> y = 0.1*x .* sin.(x) .+ 0.2 * rand(50);
julia> model = loess(x, y);
```

The loess function is called against these values reorder the the x-values using `collect()` and then apply Loess' `predict()` routine to perform the regression

```
julia> u = collect(minimum(x):0.1:maximum(x));
julia> v = Loess.predict(model, u);
```

To visualise the result we will plot the input data (x,y) as a scatter diagram and the outputs (u,v), imposed as line plot

```
julia> p = scatter(x,y)
julia> plot!(p,u,v)
```




# Loss Functions

LossFunctions(.jl) is a package that provides efficient implementations for a diverse set of loss functions that are commonly used in Machine Learning. It is part of the JuliaML rather than JuliaStats and is NOT added via StatsKit, although it does use StatsKit components to do its work.

Typically, the loss functions lie into the category of supervised losses. These are functions of two variables, a true target `y`, which represents the "truth" (i.e. correct answer) and the predicted output $\hat{y}$, which is what our model *thinks* the truth is.

A supervised loss function takes these two variables as input and returns a value that quantifies how "bad" our prediction is in comparison to the truth, i.e. the lower the loss, the better the prediction.

The Julia package provides an API to query their properties (such convexity) and also methods to compute their values, derivatives, and second derivatives for single

observations and also arbitrarily sized arrays,

Losses are immutable types and has to instantiate one in order to work with them.
For most losses, the constructors do not expect any parameters.

```
julia> L2DistLoss()
LPDistLoss{2}()
```

Losses such as `L2DistLoss()` do not have any constructor parameters or member variables, there is no additional code executed at all. Such *singletons* are only used for dispatch and don't even produce any additional code and therefore are abstractions of zero-cost.

The following shows the use of the L2DistLoss routine.

```
julia> using LossFunctions

# Setup an array of true and predicted values
julia> true_targets = [ 1, 0, -2, 1, 1.5] ;
julia> pred_outputs = [0.5, 2, -1, 1, 2];

# Instantiate the loss function ...
julia> loss = L2DistLoss()

# ... and compute its value plus the 1st/2nd derivatives
# ... display as the adjoint for better clarity
julia> hcat(value(loss, true_targets, pred_outputs),
     deriv(loss, true_targets, pred_outputs),
     deriv2(loss,true_targets, pred_outputs))'

3×5 LinearAlgebra.Adjoint{Float64,Array{Float64,2}}:
 0.25 4.0 1.0 0.0 0.25
 -1.0 4.0 2.0 0.0 1.0
 2.0 2.0 2.0 2.0 2.0
```

We can also work with multidimensional arrays

```
julia> A = rand(3,4)
3×4 Array{Float64,2}:
 0.155142 0.162888 0.905477 0.981411
 0.939674 0.520626 0.802698 0.325044
 0.812418 0.615272 0.585989 0.2044

julia> B = rand(3,4)
3×4 Array{Float64,2}:
 0.97898  0.638196 0.464653  0.0844752
 0.55683  0.499168 0.778739  0.835229
 0.480935 0.640781 0.0751028 0.362013
```

It is possible to define a weighting function (on a per-column basis) and then compute weighted sums and means.

```
julia> W = [2,1,3,2];

julia> v1 = value(L2DistLoss(), A, B, AvgMode.WeightedSum(W))
julia> v2 = value(L2DistLoss(), A, B, AvgMode.WeightedMean(W))
julia> round.([v1, v2], digits=4)
2-element Array{Float64,1}:
 5.6443
 0.2352
```

# Clustering

Clustering techniques provide a methods of classification in cases which we have described above as unsupervised.

The main package is unremarkably called Clustering.jl and

- K-means
- K-medoids
- Affinity Propagation
- Density-based spatial clustering with noise (DBSCAN)
- Markov Clustering Algorithm (MCL)
- Fuzzy C-Means Clustering
- Hierarchical Clustering

The package is well documented on `clusteringjl.readthedocs.io` and the various algorithms are well described there. We will turn attention to actual application of clustering using K-means segmentation; again we first use the familiar IRIS dataset.

```
julia> using RDatasets, Clustering
julia> iris = RDatasets.dataset("datasets", "iris");
```

The dataset can be reduced, extracting the numeric fields [1:4] from the final species [5] and apply the k-means algorithm against this subset.

k-means clustering is a method of quantisation, originally from signal processing, and one which historically become popular in data mining

The aim is to partition n observations into k clusters (n >> k) in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This problem is seen as computationally difficult (*NP-hard*), however there exist some efficient

heuristic algorithms which normally converge quickly to a local optimum.

The algorithm has a loose relationship to the k-nearest neighbour classifier, another popular ML categorisation technique. Julia has naturally an implementation of this too due to Kristoffer Carlsson, viz. NearestNeighbors.jl

We need to choose a value for k and our knowledge of the iris dataset suggests k = 3.

```
julia> features = Matrix(iris[:,[1,2,3,4]])'
julia> result = kmeans( features, 3)
KmeansResult{Float64}([5.90161 5.006 6.85; 2.74839 3.428 3.07368; 4.39355
1.462 5.74211; 1.43387 0.246 2.07105], [2, 2, 2, 2, 2, 2, 2, 2, 2, 2  …  3,
3, 1, 3, 3, 3, 1, 3, 3, 1], [0.01998, 0.20038, 0.17398, 0.27598, 0.03558,
0.45838, 0.17238, 0.00438, 0.65198, 0.14158  …  0.151593, 0.467909,
0.729017, 0.0958033, 0.259488, 0.374224, 0.805468, 0.426856, 0.698435,
0.696436], [62, 50, 38], [62.0, 50.0, 38.0], 78.851441426147, 4, true)
```

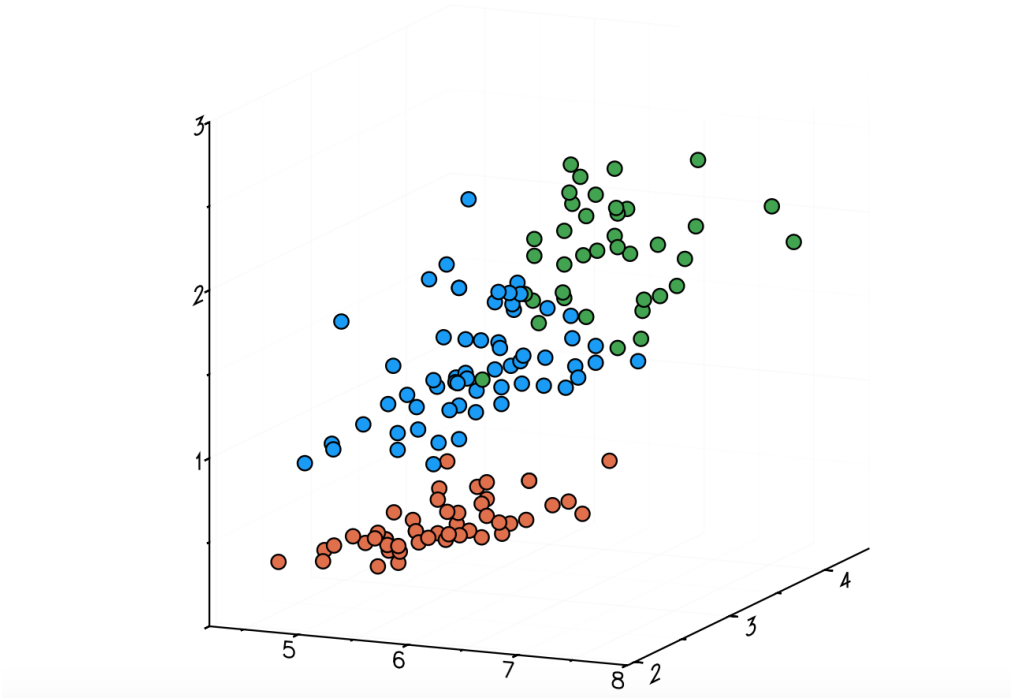
The resultant data structure has number of fields; of particular interest are the `:assignments` and `:costs`. Assignments is an array of equivalent size to that of the input dataset, costs is a measure of the degree of classification and conveniently their sum is also provided as `:total cost`

```
julia> fieldnames(KmeansResult)
(:centers, :assignments, :costs, :counts, :cweights, :totalcost,
:iterations, :converged)

julia> length(result.assignments) == size(iris)[1]
true

 julia> result.assignments
 1×150 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:
 2 2 2 2 2 2 2 2 2 2 2 2 2 … 1 3 3 3 1 3 3 3 1 3 3 1
```

Listing the assignments array indicates the cluster to which each data values has been put. With the IRIS data the algorithm performs well with the *setosa* species but is less successful in distinguishing between *versicolor* and *virginica.* (Recall that there are 50 values for each species and these are ordered in the dataset).

Visualising the data will help; there are 4 metrics we can plot any 3, or present a montage of all 4 separate displays. For our purposes a single graph will serve its purpose, the choice of which is not especially significant

```
julia> using Plots; gr()
julia> scatter(features[1,:], features[2,:], features[4,:],
               color = result.assignments)
```




In cases where a choice for k is not clear it is necessary to run a series of simulations for different k's . We should note however that increasing k without penalty will always reduce the amount of error in the resulting clustering, possibily by splitting existing clusters in smaller ones. So an indicator is when the total cost does NOT decrease by much with increases in k.

As a second example to illustrate this let us look at the dataset of motor cars which we worked with in chapter 6. The data is provided in an Excel file in the Files directory and we will use the Queryverse framework to load it and extracts the fields of interest.
This package loads Query.jl, together with the necessary modules for input, handling and output of separate data formats.

```
julia> using Queryverse
julia> mtcars = load("Files/mtcars.xlsx","data") |> DataFrame

# Reshape the names() to create more compact output
julia> reshape(names(mtcars),1,12)
1×12 Array{Symbol,2}:
 :Model :MPG :Cyl :Disp :HP :DRat :WT :QSec :VS :AM :Gear :Carb
```

The numeric fields which may be significant are :

- **MPG** : miles per (US) gallon
- **Disp** : displacement (cu.in.)
- **HP** : gross horsepower
- **WT** : weight (1000 lbs)

```
julia> df = @from car in mtcars begin
    @orderby car.MPG, car.HP
    @select {MPG=car.MPG, HP=car.HP, WT=car.WT, Disp=car.Disp}
    @collect DataFrame
end
```

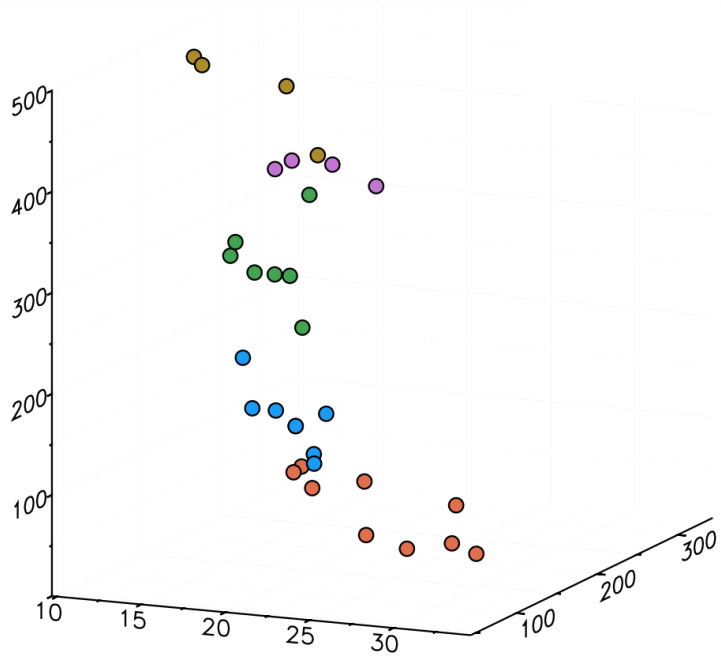
Applying a clustering (k-means) analysis for different values of k :

```
julia> using Clustering
julia> features = Matrix(df[:,2:5])'
julia> results = [kmeans( features, i) for i = 2:6]
julia> for i in 1:length(results)
    println(i+1, " => ", results[i].totalcost)
end
2 => 152386.1254654603
3 => 103396.27222330484
4 => 72463.36499131401
5 => 43990.94213469363
6 => 34666.83812172713
```

Recall the the total cost will continue to decrease with increasing k, but probably there is little to be gained in segmenting in to 6 clusters rather than 4 or 5.

The resultant plot (for k = 5) is shown below,

```
julia> result = kmeans( features, 5)

julia> using Plots; gr()
julia> scatter(features[1,:], features[2,:], features[4,:],
               color = result.assignments)
```




# Decision Trees

DecisionTree.jl is a Julia implementation of random forest algorithms which can be used for classification and regression; it supports pre/post-pruning, random forests and n-fold cross validation. It supports both mixed categorical and numerical data.

Decision trees are a popular method for various machine learning tasks but for our purposes here we will just look at a simple regression task similar to the one used with `Loess.jl` above.

```
# Create a random dataset based on a simple function
julia> Random.seed!(systime())
julia> X = sort(5 * rand(80))
julia> XX = reshape(X, 80, 1)
julia> y = X .* sin.(X)
julia> y[1:5:end] += 3 * (0.5 .- rand(16));

# Fit regression models
julia> using DecisionTree
julia> regr_1 = DecisionTreeRegressor()
julia> regr_2 =
        DecisionTreeRegressor(pruning_purity_threshold=0.05)
```

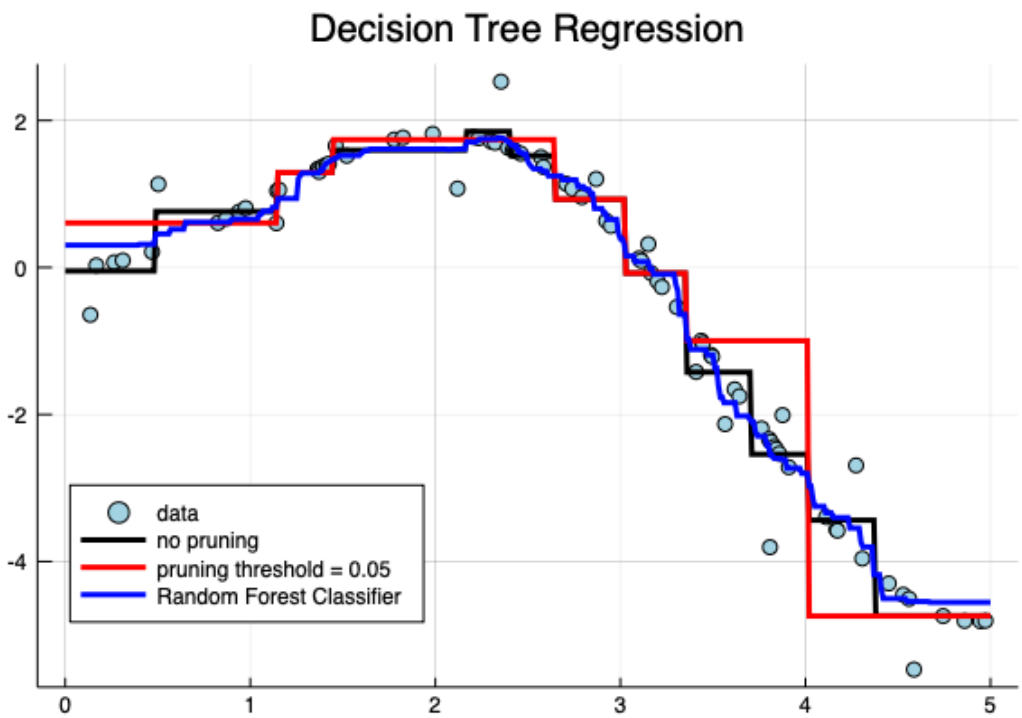
```
julia> regr_3 = RandomForestRegressor(n_trees=20)

julia> DecisionTree.fit!(regr_1, XX, y)
julia> DecisionTree.fit!(regr_2, XX, y)
julia> DecisionTree.fit!(regr_3, XX, y)
```

Random forests and the k-nearest neighbour algorithms (k-NN); both can be viewed as so-called weighted neighborhoods schemes. T hese are models built from a training set for new points x' by looking at the "neighbourhood" of the point, using a weight function W.

$$\hat{y} = \sum_{i=1}^{n} W(x_i, x')\, y_i .$$

The weights $W(x_i, x')$ are equal to 1/k when $x_i$ one of k points closest to x for kNN algorithms, whereas in decision trees the k points are in the same leaf.

```
# Predict
julia> X_test = 0:0.01:5.0
julia> y_1 = DecisionTree.predict(regr_1, hcat(X_test))
julia> y_2 = DecisionTree.predict(regr_2, hcat(X_test))
julia> y_3 = DecisionTree.predict(regr_3, hcat(X_test))

julia> using Plots
julia> gr()
julia> Plots.scatter(X, y, label="data",
                           color="lightblue", legend=:bottomleft)
julia> Plots.plot!(X_test, y_1, color="black",
                          label="no pruning", linewidth=3)
julia> Plots.plot!(X_test, y_2, color="red",
              label="pruning threshold = 0.05", linewidth=3)
julia> Plots.plot!(X_test, y_3, color="blue",
              label="Random Forest Classifier", linewidth=3)
julia> Plots.title!("Decision Tree Regression")
```




Instead of decision trees, linear models have been proposed and evaluated as base estimators in random forests, in particular multinomial logistic regression and naive Bayes classifiers which we will meet next.

Note that is package should not be confused with a couple of other packages `DecisionTrees.jl` (*with a 's'*) and `D3Trees.jl`, both of which are actively being developed at present.

# Bayesian methods

The Rev. Bayes is buried in Bunhill Fields in the City of London, less than 100 m. from my apartment. Visitors crossing the cemetery may pause to view the resting places of John Bunyan, William Blake and Daniel Defoe, never realising than a man as influential as Thomas Bayes lies close by.

Bayes is known for the Bayesian view in the interpretation of statistical data, as opposed to that taken by traditional frequentists.

Bayesian inference derives the posterior probability as a consequence of two antecedents: a prior probability and a "likelihood function" derived from a statistical model for the observed data, that is by computing the posterior according to Bayes' theorem:

$$P(\mathbf{H}|\mathbf{E}) = P(\mathbf{H}) \bullet \frac{P(\mathbf{E}|\mathbf{H})}{P(\mathbf{E})}$$

This is a rearrangement of the definition of conditional probability and so far there is no contraversy, this arises from the interpretation put upon the equation by Bayesianists.

**P(H)** is the prior probability, i.e. an the estimate of the probability of the hypothesis before the data **E**, the current evidence, is observed.

**P(E|H)** is the probability of observing **E**, given **H**, and corresponds to the likelihood. As a function of **E** for fixed **H**, it indicates the compatibility of the evidence with the given hypothesis. The likelihood function is a function of the evidence, E, while the posterior probability is a function of the hypothesis, **H**.

**P(E)** is sometimes termed the marginal likelihood or "model evidence"; it does not affect the relative probabilities of different hypotheses and acts only as a scaling factor and keeps **P(H|E)**, the posterior probaility, in the range [0,1]

The purpose of Bayesian analysis is to determine the posterior based on the priors plus and new information, which updates the likelyhood. The criticism is that a suitable prior has to be selected before the computational process can proceed.

Nevertheless application of Bayesian style analysis can tackle problems which classical methods can not and so with the advent of significant computing power there are become very popular.

The material in this section assumes some knowledge of Bayesian methods. A standard text is the `Bayesian Data Analysis` by Gelman, Carlin, Stern and Rubin

# Inference

As we noted there are two ways to approach inference from a dataset

1. Classical (frequentist) inference tell us that: we should for parameter values that maximise the likelihood function (MLE) and associate parameter's uncertainty with the calculation of standard errors which in turn enable us to construct confidence intervals for the parameters.
2. Bayesian inference views the data are treated as fixed quantities and the *parameters* are treated as random variables. This allows parameters (and models) probabilities to be assigned, and to some minds, make the inferential framework far more intuitive and straightforward, at least in principle!

In the Bayesian world everything is assigned distributions (prior, posterior); so it is possible to incorporate prior information about the parameter which is then updated by using the likelihood function leading to the posterior distribution, which should provide a good description the parameter.

One of the biggest criticisms to the Bayesian approach is the use of the prior distribution. It is necessary to choose a very reasonably informative prior to come up with favourable results and if nothing is known about the parameter - what prior should be chosen choose?

The counter arguments Bayesianists would provide against this view are:

- priors should be chosen before we see the data and it is very often the case that there is some prior information available (e.g. previous studies)
- if we know nothing about the parameter, then we could assign to it a so-called uninformative (or vague) prior;
- if there is not a lot of data available then the posterior distribution would not be influenced by the prior (too much) and vice versa;

Although Bayesian inference has been around for long time it is only the last couple of decades that it has been dominant the way that statistical modelling is approached. Thanks to computational developments such as Markov Chain Monte Carlo (MCMC) doing Bayesian inference is a lot easier and Julia has a number of packages to perform MCMC analyses.

## Naive Bayes

Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naive) independence assumptions between the features. They have been studied extensively since the 1960s when it was usually principally in the text retrieval and still is often used as a baseline method for text categorisation.

Space in this book will not permit us to look in detail at textual analysis but there are some excellent reviews online and also the reader is referred to
the `JuliaText` community group.

NaiveBayes.jl incorporates three differing approaches, to be used depending on the type of dataset.

- **MultinomialNB** - Assumes variables have a multinomial distribution and is good for text classification.
- **GaussianNB** - Assumes variables have a multivariate normal distribution and is better suited for real-valued data.
- **HybridNB** - A hybrid empirical naive Bayes model for a mixture of continuous and discrete features; continuous features are estimated using Kernel Density Estimation.

The following code applies the GaussianNB method to the IRIS dataset.

```
julia> using NaiveBayes
julia> using RDatasets
julia> using StatsBase
julia> iris = dataset("datasets", "iris");
(4, 150)
```

Let's extract the first number columns into a matrix `X` and the species in a separate array `y`

```
# Observations in columns and variables in rows
julia> X = convert(Array, iris[:, 1:4])';
julia> p, n = size(X)


# By default species is a PooledDataArray,
julia> y = [species for species in iris[:, 5]];
```

Create a random (Gaussian) set of numbers and permute them.
90% of the dataset is used for training and the remaining 10% for testing.

```
# Use for 90% training
julia> train_frac = 0.9;
julia> k = floor(Int, train_frac * n);

julia> idxs = randperm(n)
julia> train = idxs[1:k]
julia> test = idxs[k+1:end]
```

Define a GaussianNB (multivariate normal distribution) model and fit the training values in `X` against the species array `y` .

```
julia> model = GaussianNB(unique(y), p)
julia> fit(model, X[:, train], y[train]);
```

From the model we can make predictions against the test set and compare the accuracy of the fit.

```
# Define a routine to count the number of non-zero elements
# (This routine is deprecated in v1.+)

julia> countnz(aa::Array) =
   count([aa[i] != 0 for i in 1:length(aa)])

julia> accuracy =
   round(count(predict(model, X[:,test]) .== y[test]) / countnz(test),
digits = 4);

julia> @printf  "Accuracy: %.4f\n: $accuracy
Accuracy: 0.9333
```

# ApproxBayes

Approximate Bayesian computation (ABC) comprises a set of methods which can be used to estimate the posterior distributions of model parameters.

The likelihood function, *defined above as* ***P(E|H)***, is of central importance in statistical inference, since it quantifies the support data lending to particular values of parameters and to choices among different models.

The ABC approach bypasses the evaluation of the likelihood function and as such, widens the classes of models in which inference can be applied, however assumptions are made whose impact needs to be assessed.

ABC methods approximate the likelihood function by simulations, the outcomes of which are compared with the observed data. In the most basic algorithm - ABC rejection algorithm - a set of parameter points is first sampled from the prior distribution and a dataset is simulated under the statistical model . If the generated is too different from the observed data the sampled parameter value is discarded and the process repeated.

The outcome of ABC rejection is a sample of parameter values approximately distributed according to the desired posterior distribution, and, crucially, obtained without the need to explicitly evaluate the likelihood function.

An alternate scheme is ABCSMC (sequential Monte Carlo), based on the reject algorithm but computationally more efficient.

The Julia package ApproxBayes.jl implements both methods.

As an example we will take a random set of values from a Gamma distribution.

A utility function `ksdist()` is applied to compare this date with a target set and compute a `distance` measure.

```
julia> using ApproxBayes, Distributions
julia> function gammadist(params, constants, targetdata)
   simdata = rand(Gamma(params...), 1000)
   ApproxBayes.ksdist(simdata, targetdata)
end

julia> Random.seed!(systime())
julia> α = 0.1
julia> θ = 1.0
julia> target = rand(Gamma(α,θ), 1000);
```

First let's apply the SMC algorithm.

The Gamma function has two parameters $\alpha$ and $\theta$ and we specify a target distance estimate ($\epsilon$) and two Uniform distributions for the simtta and targetdata.

The following computes the time to run the simulation against the model (S)

```
julia> modelS = ABCSMC(gammadist,   # simulation function
   2,                                # number of parameters
   0.1,                              # target ϵ
   Prior([Uniform(0.0, 0.5),Uniform(0.0, 2.0)]),
)
julia> @time smc = runabc(modelS, target, verbose=true)
Use ABC rejection to get first population
Preparing to run in serial on 1 processor
Running ABC SMC...

Preparing to run in serial on 1 processor
Total number of simulations: 5.34e+02
Cumulative number of simulations = [100, 534]
Acceptance ratio: 1.87e-01
Out[3]:
Tolerance schedule = [0.21, 0.1]

Median (95% intervals):
```

```
Parameter 1: 0.12 (0.06,0.18)
Parameter 2: 0.94 (0.13,1.95)
```

SMC runs the rejection function for the first time to bootstrap the process. We can also apply the rejection routine against the same model specification. This time we need to add on extra backstop of a maximum number of iteration by which the algorithm has to terminate (or exited)

```
julia> modelR = ABCRejection(gammadist, # simulation function
  2,                                    # number of parameters
  0.1,                                  # target ϵ
  Prior([Uniform(0.0, 0.5), Uniform(0.0, 2.0)]); # Prior for each of the
parameters
  maxiterations = 10^6,    # Maximum number of iterations before the
algorithm terminates
)
julia> rej = runabc(modelR, target)
Preparing to run in serial on 1 processor
Number of simulations: 1.15e+03
Acceptance ratio: 8.70e-02

Median (95% intervals):
Parameter 1: 0.10 (0.08,0.14)
Parameter 2: 1.03 (0.20,1.93)
```

Comparing the two runs we see that SMC acceptance ratio is much better than the rejection ratio; this is inversely related to the number of simulations needed for convergence in each method.

```
julia> round(smc.accratio/rej.accratio, digits=4)
2.1517

julia>  (sum(smc.numsims), rej.numsims)
(534, 1149)
```

For other distributions this difference is much more marked.

For example if we chose a Gaussian distribution rather than a Gamma:

```
julia> function normaldist(params, constants, targetdata)
  simdata = rand(Normal(params...), 1000)
  ApproxBayes.ksdist(simdata, targetdata), 1
end

julia> Random.seed!(systime())
julia> μ = 5.0
julia> σ = 1.0
julia> target = rand(Normal(μ, σ), 1000);
```

```
julia> modelS = ABCSMC(normaldist,   # simulation function
    2,                               # number of parameters
    0.1,                             # target ϵ
    Prior([Uniform(0.0, 15.0), Uniform(0.0, 2.0)]),
                                     # prior for each parameter
)

# Similar definition for modelR
```

In this case we see that the convergence of the SMC routine is much quicker.

```
julia> smc.accratio/rej.accratio
28.8165

julia> (sum(smc.numsims), rej.numsims
(534, 15388)
```

# Markov Chain Monte Carlo (MCMC)

A stochastic process is considered to be Markov if any future state does not depend on previous history, i.e. only on the current one. This clearly simplifies the computational process

Markov Chain Monte Carlo (*MCMC*) is a technique for estimating by simulation the expectation of a statistic in a complex model. Successive random selections form a Markov chain, the stationary distribution of which is the target distribution.

It is particularly useful for the evaluation of posterior distributions in Bayesian models. The problem is how to efficiently select a value from a (*statistical*) distribution and this is that which MCMC aims to solve. Why is it a problem? Because we have no idea what actual process is used, so how can that process be simulated.

To illustrate this consider the banker teller example in chapter 6, we do not know by which means customers decide to use the bank, which makes parameter estimation difficult.

## Sampling Algorithms

he Metropolis–Hastings algorithm is a very popular MCMC method for obtaining a sequence of random samples from a probability distribution when direct sampling is difficult. The sequence can be used to approximate the distribution or to compute epected values.

Metropolis–Hastings and other MCMC algorithms are generally used for sampling from

multi-dimensional distributions, especially when the number of dimensions is high. For single-dimensional distributions, there are other methods, such as adaptive rejection sampling, that can directly return independent samples, free from the problem of autocorrelation occuring with MCMC methods.

In the Metropolis–Hastings algorithm, items are selected from an arbitrary "proposal" distribution and are retained or not according to an acceptance rule.

The Gibbs sampler is a special case in which the proposal distributions are conditional distributions of single components of a vector parameter.

## MCMCChain

MCMCChain,jl is a useful package to summarise what is happening in MCMC simulations, providing utility functions for visualisations and diagnostic purposes.

This is a simple example showing how to use the module the StatsPlots module to visually summarize a MCMC simulation.

```
# StatsPlots can produce ggplot2 style graphics
julia> using MCMCChain, StatsPlots
julia> theme(:ggplot2);

# Define the experiment
julia> n_iter = 500;
julia> n_name = 3;
julia> n_chain = 2;
```

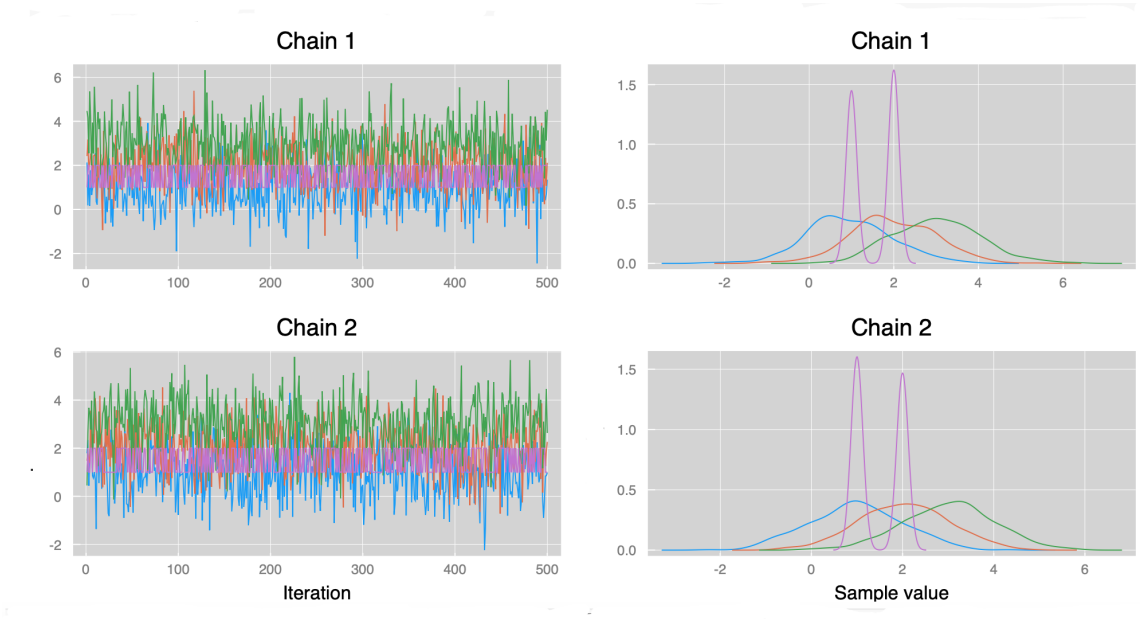
The target set will be a 500x3x2 matrix of Normally distributed (μ=0, σ=1) values, to which an offset (1,2,3) is applied

To this is added an additional columns of values: either 1.0 or 2.0

```
julia> val = randn(n_iter, n_name, n_chain) .+ [1, 2, 3]'
500×3×2 Array{Float64,3}:
[:, :, 1] =
  1.49209  1.7195  2.23688
  1.77179  2.68773 3.64836
  0.762639 2.07804 3.49789

julia> val = hcat(val, rand(1:2, n_iter, 1, n_chain))
500×4×2 Array{Float64,3}:
[:, :, 1] =
  1.49209  1.7195  2.23688 1.0
  1.77179  2.68773 3.64836 1.0
  0.762639 2.07804 3.49789 2.0
```

We can construct a chains model from the val matrix and plot the results for each chain as:

```
julia> chn = Chains(val);
julia> p = plot(chn, colordim = :parameter)
```




The package provides a variety of convergence diagnostic routines:

- Discrete
- Gelman, Rubin and Brooks
- Geweke
- Heidelberger and Welch
- Raftery and Lewis

As an example we can run the `GRB diagnostic` against other simple randomised dataset.

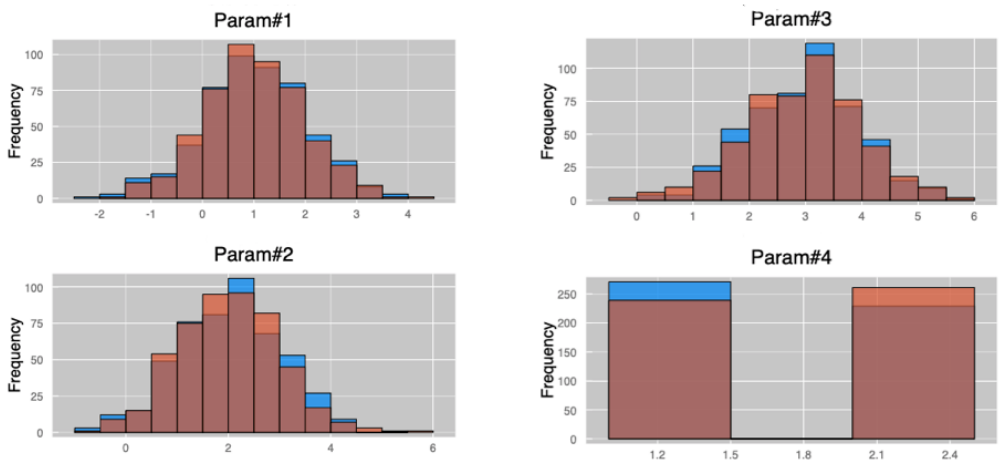
```
julia> gelmandiag(chn; alpha=0.05, mpsrf=false, transform=false)
           PSRF    97.5%
Param#1    1.000   1.001
Param#2    1.000   1.002
Param#3    1.001   1.001
Param#4    1.005   1.030
```

StatsPlots *understands* the Chain results structure and can display a variety of visualisations in addition to the standard one above - i.e. density distributions, histograms, auto-

correlations, mixed (continuous/discrete) plots etc.

The figure below is the histogram plot, for others the reader is referenced to the accompanying notebook.

```
julia> histogram(chn)
```




## Mamba

Mamba is a package designed general Bayesian model fitting via MCMC.

Like BUGS and JAGS, it supports a wide range of model and distributional specifications, incorporating a syntax for model specifications unlike those two, but similar to PyMC, Mamba provides a unified environment in which all interactions with the software are made through a single, interpreted language

It provides a framework for

1. specification of hierarchical models
2. block-updating of parameters
3. execution of sampling schemes
4. posterior inference.

The `package` is well documented and the following example, creating a Linear Regression Model, is explained in more detail there; based on one posed in the `BUGS 0.5 manual`.

It describes a regression relationship between observations $\mathbf{x}^T$ and $\boldsymbol{y}$ which can be expressed as $\mathbf{y} \sim \mathbf{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ where $\boldsymbol{\mu} \approx \mathbf{X} \otimes \boldsymbol{\beta}$.

$\boldsymbol{\beta} \Rightarrow [\boldsymbol{\beta}_1, \boldsymbol{\beta}_2]^T$ is normally distributed and $\boldsymbol{\sigma}^2$ has an inverse gamma distribution.

Terms in the model specification are referred to as nodes and are classified as :

- **Stochastic** nodes are terms that have likelihood or prior distributional specifications - in our example $\mathbf{y}$, $\boldsymbol{\beta}$ and $\boldsymbol{\sigma}^2$ are stochastic nodes.
- **Logical** nodes such as $\boldsymbol{\mu}$ are deterministic functions of other nodes.
- **Input** nodes are any remaining terms in the model, **X,** and are taken as fixed quantities.

In symbolic form the Mamba model may be expressed as:

```
julia> using Mamba, LinearAlgebra, Distributions
julia> model = Model(
    y = Stochastic(1, (μ,ν) ->  MvNormal(μ, sqrt(ν)), false),
    μ  = Logical(1,(xm, β) -> xm * β, false),
    β = Stochastic(1, () -> MvNormal(2, sqrt(1000))),
    ν = Stochastic(() -> InverseGamma(0.001, 0.001)));
```

Next we need to establish a sampling scheme.

The following are three such schemes: two based on NUTS (No-U-Turn) and two other using Gibbs sampling.

```
# Hybrid No-U-Turn (NUTS) Slice and Sampling Schemes
julia> sc1 = [NUTS(:β), Slice(:ν, 3.0)];

# No-U-Turn Sampling Scheme
julia> sc2 = [NUTS([:β, :ν])];

# User-Defined Samplers
julia> Gibbs_β = Sampler([:β],
      (β, ν, xm, y) ->  begin
      β_mean = mean(β.distr)
      β_invcov = invcov(β.distr)
      σ = inv(Symmetric(xm' * xm / ν + β_invcov))
      μ = σ * (xm' * y / ν  +  β_invcov * β_mean)
      rand(MvNormal(μ , σ))
   end);

julia> Gibbs_ν = Sampler([:ν], (μ, ν, y) -> begin
      a = length(y) / 2.0 + shape(ν.distr)
      b = sum(abs2, y - μ) / 2.0 + scale(ν.distr)
```

```
        rand(InverseGamma(a, b))
      end);

julia> sc3 = [Gibbs_β, Gibbs_ν];
```

[1] is one of the multivariate normal distributions available in the Distributions package, and the specification used in the example model implementation.

[2], a single univariate normal distribution is specified to imply independent priors of the same type for all elements of beta

[3] and [4] explicitly specify a univariate prior for each element of beta and allow for the possibility of differences among the priors.

As per the BUGS problem we define the X column vector as [1,2,3,4,5] and the y row vector as [1,3,3,3,5]; β has a normal distribution and ν a gamma

```
# Data
julia> line = Dict{Symbol, Any}(:x => [1, 2, 3, 4, 5],
                                :y => [1, 3, 3, 3, 5] )
julia> line[:xm] = [ones(5) line[:x]];

# Initial Values
julia> inits = [ Dict{Symbol, Any}(
                     :y => line[:y],
                     :β => rand(Normal(0, 1), 2),
                     :ν => rand(Gamma(1, 1))) for i in 1:3 ];
```

When a sampling scheme is formulated, it can be assigned to an existing model via to the `setsamplers!()` routine and the MCMC run against it, specifying the number of iterations and chains, a burnin period and a thinning factor (to reduce any effects of autocorrelation)

```
# Use scheme 1 as a sampling method
julia> setsamplers!(model, sc1)
julia> sim1 = mcmc(model, line, inits, 10000,
                   burnin=250, thin=2, chains=3)
MCMC Simulation of 10000 Iterations x 3 Chains...
Chain 1: 0% [0:00:43 of 0:00:43 remaining]
Chain 1: 10% [0:00:05 of 0:00:05 remaining]
Chain 1: 20% [0:00:04 of 0:00:04 remaining]
. . . . . .
. . . . . .
. . . . . .
Chain 3: 90% [0:00:01 of 0:00:06 remaining]
Chain 3: 100% [0:00:00 of 0:00:06 remaining]
```

```
Object of type "ModelChains"
Iterations = 252:10000
Thinning interval = 2
Chains = 1,2,3
Samples per chain = 4875
```

As with the previous MCMC-Chain example a number of diagnostic functions are available to assess the success of the iteration.

```
julia> gelmandiag(sim1,mpsrf=true,transform=true) |> showall

Gelman, Rubin, and Brooks Diagnostic:
                  PSRF     97.5%
        β[1]      1.000    1.000
        β[2]      1.003    1.003
           ν      1.003    1.011
Multivariate      1.006    NaN
```

Other diagnostics are computed in the notebook . . .

```
gewekediag(sim1)          # Geweke diagnostic
heideldiag(sim1)          # Heidel diagnostic
rafterydiag(sim1)         # Raffery diagnostic
```

. . . together with various statistics such as:

```
hpd(sim1)                 # Highest Posterior Density Intervals
autocor(sim1)             # Lag-Autocorrelations
changerate(sim1)          # Statespace Change Rate, per iteration
dic(sim )                 # Deviance Information Criterion
```

Once the simulation is run we can use describe() to summarise the results

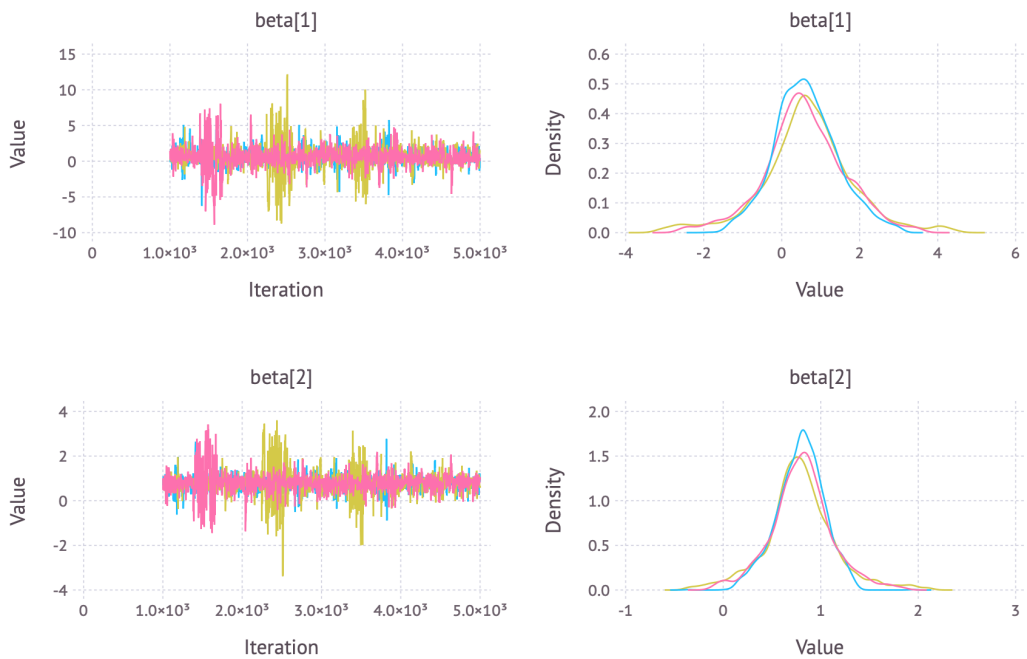
```
julia> describe(sim1)    # Summary Statistics
Iterations = 252:10000
Thinning interval = 2
Chains = 1,2,3
Samples per chain = 4875

Empirical Posterior Estimates:
        Mean   SD Naive   SE          MCSE        ESS
β[1] 0.6069200 1.07341065 0.008876014 0.016746056 4108.72190
β[2] 0.7984684 0.32376414 0.002677200 0.004784984 4578.21785
   ν 1.0911861 1.40365424 0.011606792 0.048496129 837.73397

Quantiles:
         2.5%    25.0%       50.0%      75.0%      97.5%
```

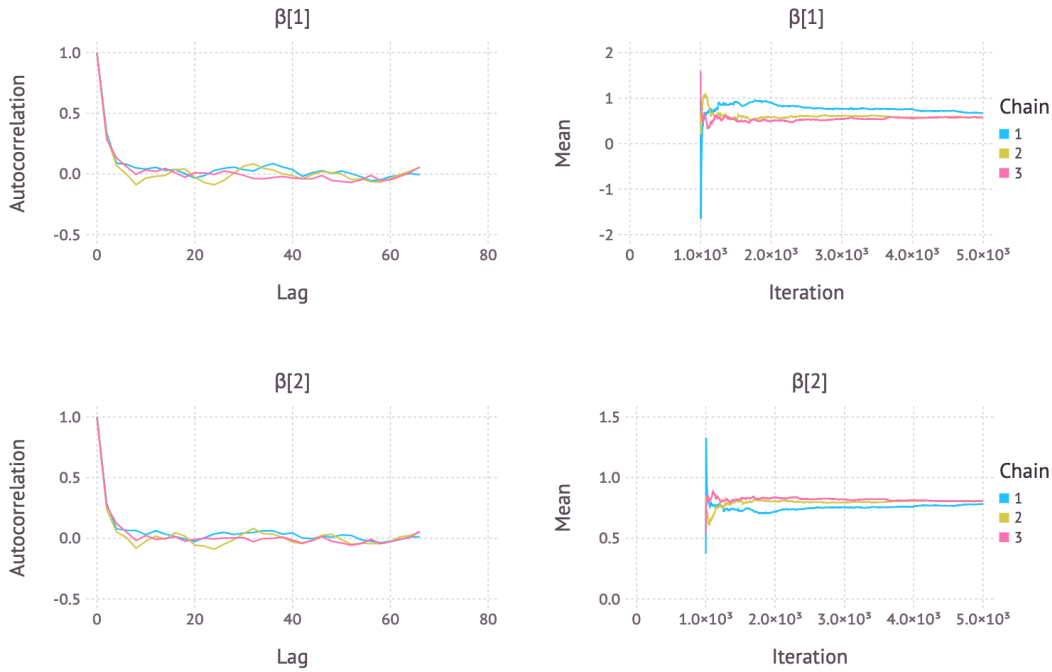
```
β[1] -1.54170371 0.028361366 0.60694753 1.18917177 2.7976852
β[2] 0.13584976  0.623769205 0.79784229 0.97294898 1.4417826
   ν 0.16788738  0.384784960 0.65347871 1.20108193 4.8562314
```

It is possible to use Gadfly to create a series of displays, which can be saved as SVG files and displayed in the browser.

```
julia> using Gadfly
julia> p = plot(sim1)
julia> draw(p, filename="summaryplot.svg")

# Autocorrelation and running mean plots
julia> p = plot(sim, [:autocor, :mean], legend=true)
julia> draw(p, nrow=3, ncol=2, filename="autocormeanplot.svg")

# Pairwise contour plots
julia> p = plot(sim, :contour)
julia> draw(p, nrow=2, ncol=2, filename="contourplot.svg")
```

The figure below shows the results for the beta for the 3 separate chains and the autocorrelation plots of the means.






# STAN

Stan is a system designed for performing statistical modelling computations rapidly which is popular within the Python and R communities, both providing wrapper modules to the Stan library.

However the Julia implementation utilises the command version `CMDSTAN` via a tasking interface. That is, Stan is run as a separate task and necessarily needs to be installed and the binary executable, *via the command path*. Because of this STAN is probably less well favoured than some of the frameworks discussed in the next section.

In Stan a set of log density functions are specified using a builtin probabilistic programming language (DSL).

From these it is then possible to obtain:

- Full Bayesian statistical inference with MCMC sampling (HMC / NUTS)
- Approximate Bayesian inference with variational inference (ADVI)
- Penalized maximum likelihood estimation (BFGS)

Stan has a loyal following and provides details and many examples which can be found on the `website` and also, similar to Julia, an active `discourse` forum.

For our purposes we are interested in the MCMC sampling.

Hamiltonian Monte Carlo (HMC) algorithm is a MCMC method for obtaining random samples sequences from a probability distribution when direct sampling is difficult. It is based on methods derived from dynamic systems, hence the epithet Hamiltonian, whereas Metropolis-Hastings (MCMC) methods are essentially random walk algorithms.

To use HMC, all sampling variables have to be unconstrained, that is the posterior density and its gradient must be well-defined for all real parameter values. It is usually more efficient than a Metropolitan Hastings approach but convergence can be a problem in cases where the model has 'local' minimums.

NUTS (*No U-Turn Sampler*) is a special case of HMC which voids the random walk part, instead performing a series of jumps, the length of each jump increasing as the algorithm runs. Convergence occurs if the trajectory reaches a point where it becomes possible to return to the starting point.

First we need to specify an environment where the command line version of Stan is installed.

```
julia> ENV["JULIA_CMDSTAN_HOME"] = "/Users/malcolm/cmdstan"
julia> using CmdStan
julia> const bernoullimodel = """
    data       { int<lower=0> N; int<lower=0, upper=1> y[N];}
    parameters { real<lower=0, upper=1> theta; }
    model      { theta ~ beta(1,1); y ~ bernoulli(theta); }
"""

julia> model1 = Stanmodel(name="bernoulli",
                          model=bernoullimodel);
julia> bernoullidata = Dict("N" => 10,
                    "y" => [0, 1, 0, 1, 0, 0, 0, 0, 0, 1])
julia> rc, sim1, cnames =
    stan(model1, bernoullidata, ".", CmdStanDir=CMDSTAN_HOME)

julia> monitor = ["theta", "lp__", "accept_stat__"]
julia> model1 = Stanmodel(name="bernoulli",
                          model=bernoullimodel,
                          monitors=monitor)

julia> size(sim1)
julia> sim = sim1[:, [8,1,2], :];
```

## Other Bayesian methods

### JAGS

JAGS.jl is an alternate package similar to STAN.jl and also utilises a command version of JAGS (*Just Another Gibbs Sampler*) similar to the earlier BUGS (B*ayesian inference Using Gibbs Sampling*) program, both sharing the same model specification syntax.

There are a number of JAGS binaries for common operating systems and the binary needs to be installed and on the executable path.

The environment variable JAGS_HOME must be set to point to the installation directory and JULIA_SVG_BROWSER to point to a web application to view SVG files, otherwise using the package will complain; JAGS.jl also requires the Mamba packages and Data Frames to be available

A sample run using JAGS.jl is included in the accompanying files to this chapter.

### Turing

Turing.jl is a Julia library for probabilistic programming.

Current features include:

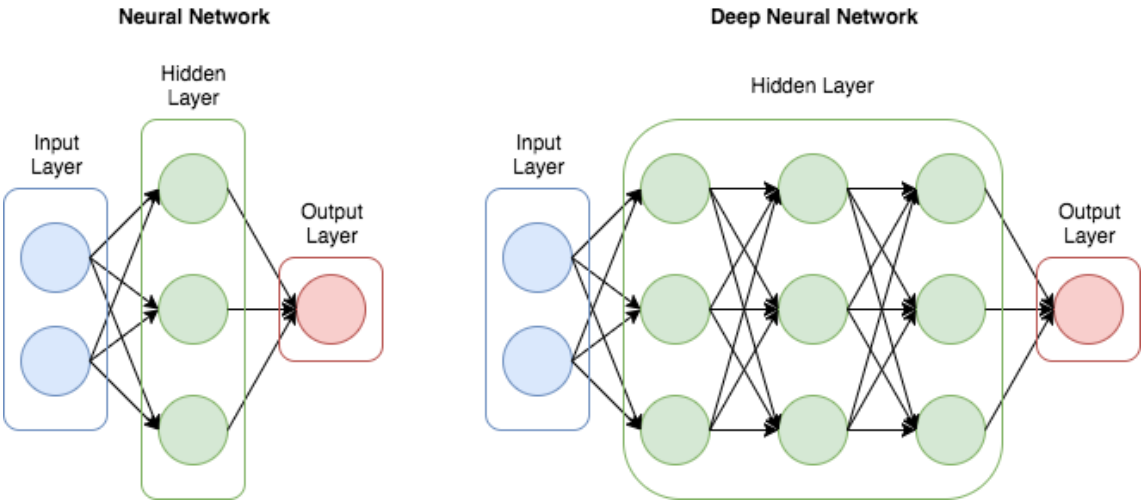
- Universal probabilistic programming with an intuitive modelling interface.
- Hamiltonian Monte Carlo (HMC) sampling for differentiable posterior distributions.
- Particle MCMC sampling for complex posterior distributions involving discrete variables and stochastic control flows.
- Gibbs sampling that combines particle MCMC, HMC and a variety of other MCMC algorithms.

There is an accompanying `website` which introduces the package and has a link to a set of tutorial examples.

# ML frameworks

The methods we have looked at so far have applied classical and/or Bayesian statistical methods to infer parameters which describe a model from target datasets. In this final section we will look at some packages which use artificial neural networks (ANNs) to tackle similar problems.

Neural networks are sometimes classified as convolutional (CNN) or deep (DNN). The difference is mainly on of degree, normally we would think of a network as being *deep* is it includes 3 or more *hidden* layers.




In the Julia world there are three very popular approaches which we will discuss later.

First a quick discourse on a package designed to help obtain that all important training (*test*) data.

# ML Datasets

The MLDatasets.jl modules associates a series of datasets with their own dedicated sub-modules and can be used to download training and test data for machine learning packages, consisting of:

- **CIFAR10** : 60000 32x32 colour images in 10 classes (50K train/10K test): airplanes automobile bird cat deer dog frog horse ship truck]
- **CIFAR100** : Like the CIFAR-10, except it has 100 classes containing 600 images each rouped into 20 superclasses
- **MNIST** : A training set of 60000 handwritten digits and a testset of 10,000 digits
- **FashionMNIST** : Similar to MNIST but of t-shirts, trousers, shoes etc.
- **SVHN2** : Another image dataset of (Google) Street View House Numbers, similar to MNIST

- **PTBLM** : Penn Tree Bank corpus which annotates syntactic or semantic sentence structure, used in textual analysis
- **UD_English** : Universal Dependencies Corpus for English of 254,830 words and 16,622 sentences, taken from weblogs, newsgroups, emails, reviews, etc.

For example you can load the training set and the test set of the MNIST database of handwritten digits using the following commands:

The package defines a common interface for interacting with the datasets, loading the necessary files from the cached directory ~/.julia/datadeps. If the data is not available in the cache a download from the online source is triggered.

For example to get the MNIST training and test data use the code:

```
julia> using MLDatasets
julia> train_x, train_y = MNIST.traindata();
julia> test_x, test_y = MNIST.testdata();
```

TIP

It is possible to overwrite the default cache folder by specifying: traindata(dir = <*directory*>) and testdata(dir = <*directory*>)

# MNIST

The MNIST database (*Modified National Institute of Standards and Technology database*) was mentioned above. It is a large dataset of handwritten digits which has been extensively used training image processing systems. To some extent it can be considered as the "*Hello World*" example of machine learning systems

Each digit in the training set is *tagged* with the number which the image represents..

The testset is then run to assess the extent to which the system has learned to distinguish between individual images. We saw in the previous section that other datasets, such as house numbers and articles of clothing are also available and can be acquired using MLDatasets.

The training set consists of 60,000 black and white images and a testset of 10,000 images. Each sample image has dimensions 28x28 and linearized as a bit-vector of size 784

# Flux

Flux is a library for machine learning from the prolific Mike Innes with many useful tools to simplify the specification, training of and prediction from neural networks applied to a wide set of machine learning problems primarily using methods to determine weighting based on gradient descent methods.

As such Flux as built-in methods for calculating gradients of simple (*linear*) functions.

```
julia> using Flux
julia> using Flux.Tracker
julia> f(x) = 3x^2 + 2x + 1;

# Compute the first derivative (for x => 3)
julia> df(x) = Tracker.gradient(f, x; nest = true)[1];    # => 6x + 2
julia> df(3)
```

```
20
# . . . and also the second derivative
julia> d2f(x) = Tracker.gradient(df, x; nest = true)[1];  # => 6
julia> d2f(3)
6
```

Normally we are dealing which set of linear equations of the form `W*x + b` corresponding to the interconnection of nodes, where `W` is the weighting matrix and `b` a bias vector. Because the dimensions of `W` and `b` may be quite large in complex ANNs, Flux let's parameters be specified prior to a function call rather than passed during its invocation.

```
julia> W = rand(2, 5)
julia> b = rand(2)
julia> predict(x) = W*x .+ b
julia> function loss(x, y)
    ŷ = predict(x)
    sum((y .- ŷ).^2)
end
julia> x, y = rand(5), rand(2);   # Create some dummy data
julia> loss(x, y)
0.1997736493419768
```

The `predict()` and `loss()` routines are defined to operate with matrices and vectors, so we can apply the same procedures here as well.

```
julia> W = param(W);
julia> b = param(b);
julia> gs = Tracker.gradient(() -> loss(x, y), params(W, b));
julia> gs[W]
Tracked 2×5 Array{Float64,2}:
 -0.0756615 -0.0282604 -0.245308 -0.0556396 -0.0927113
  0.0701236  0.0261919  0.227353  0.0515672  0.0859255
julia> gs[b]
Tracked 2-element Array{Float64,1}:
 -0.6556360697646515
  0.6076479603852478
```

Notice that both `W` and `b` are tracked arrays, which lets the weights to be updated (*on a per iteration basis*) and the loss recalculated.

```
julia> using Flux.Tracker: update!
julia> D = gs[W]
julia> update!(W, -0.1D)
Tracked 2×5 Array{Float64,2}:
 0.46683 0.0810045 0.829748 0.303616 0.778198
 0.25735 0.821648 0.905199 0.0288757 0.868002
```

```
julia> loss(x, y)
0.1854668672151869 (tracked)
```

Given methods to compute and upload weighting in the neural network we need now to define it. Flux provides a simple routine `Chain()` in which each parameter (excepting the last) corresponds to a layer, which are called in sequence on given inputs.

```
julia> model = Chain( Dense(768, 128, σ),
                      LSTM(128, 256),
                      LSTM(256, 128),
                      Dense(128, 10),
                      softmax) |> gpu
```

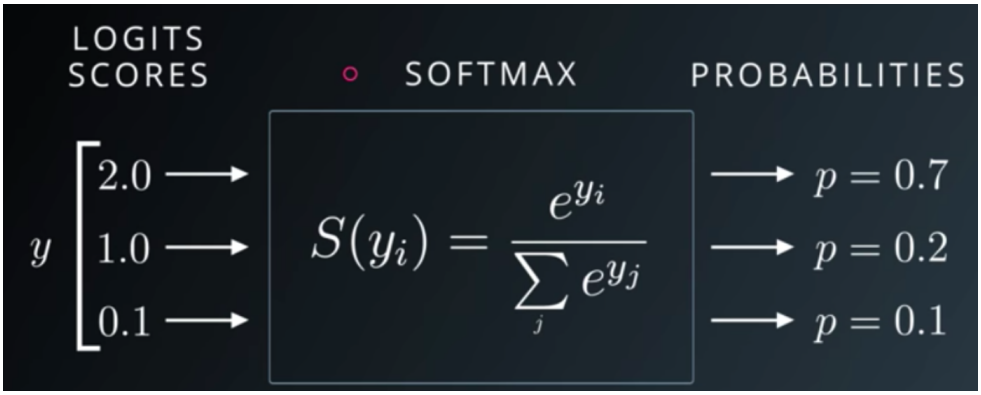
One feature of the call is that the routines such as Dense, LSTM etc., are merely Julia functions operating on their inputs as a pipeline, as so simple Julia functions can be also used in the chain

```
julia> m = Chain(x -> x^2,  x -> x + 4)
julia> m(5)   # i.e. (5*5) + 4  => 29
29
```

There are a variety of functions which can be used in the Chain() routine:

- **Dense**: Creates a traditional Dense layer with parameters W and b :
  y = σ.(W * x .+ b)
- **LSTM:** Long Short Term Memory recurrent layer which behaves like an RNN but generally exhibits a long(er) memory span over sequences.
- **RNN: A** basic recurrent layer; essentially acts as a dense layer but with the output *fed back* into the input each time step.

The final function in the example above is `softmax` . It is a generalisation of the sigmoid function with the feature that it that turns numbers *aka* logits into probabilities that sum to one and is popular as the final layer of a NN classifier.




The `|> gpu` statement will pipe the model to the GPU if it is available, otherwise it will be ignored, *so it can be added to the model regardless.*

To get GPU support for NVIDIA graphics cards, assuming your system has the appropriate hardware, you need to install have the CUDA toolkit installed together with the cuDNN, which currently is not bundled with the standard SDK but is available from the same `website`. Registration is required but use is free and as well as downloads there is a wealth of information, examples etc., with which the CUDA support can be tested.

Once CUDA is a available the packages: CuArrays, CUDAnative, CUDAapi should be installed; these can be found by using the `JuliaGPU group` github page.

## Model Zoo

This repository contains various demonstrations of the `Flux` machine learning library.

The models are broadly categorised into the folders

- **Audio:** speech recognition
- **Vision**: large convolutional neural networks (CNNs),
- **Text**: various recurrent neural networks (RNNs) and natural language processing (NLP) models
- **Other**: Miscellaneous and Reinforcement Learning examples .

Each model has its own project folder and it is possible to run model by opening that folder and using the package manager to `activate` and `instantiate` it.

```
(v1.0) pkg> activate
(v1.0) pkg> instantiate
```

```
# Alternatively it is possible to use the Pkg API as:
# using Pkg; Pkg.activate("."); Pkg.instantiate()
```

This effective creates a virtual environment for the model and fetches any dependences as lists in the package manifest. We will look at this further in the final chapter when discussing the development and deployment of our *own* module.

In the vision folder is the standard *hello-world* project MNIST, which we will look at next.

# MNIST

Flux has it's own download routines for MNIST, so we do not need to use the MLDatasets module to get the datasets.

```
julia> using Flux, Flux.Data.MNIST, Statistics
julia> using Flux: onehotbatch, onecold, crossentropy, throttle
julia> using Base.Iterators: repeated

# Uncomment the CuArrays statement if CUDA is availible
julia # using CuArrays
```

We also reference some of Flux's built-in routines which otherwise would need to implemented by the programmer; these are discussed below as they are called.

```
# Classify MNIST digits with a simple multi-layer-perceptron
julia> imgs = MNIST.images();

# Stack images into one large batch
julia> X = hcat(float.(reshape.(imgs, :))...) |> gpu
julia> labels = MNIST.labels() |> gpu

# One-hot-encode the labels
julia> Y = onehotbatch(labels, 0:9) |> gpu ;
```

`MNIST.images()` and `MNIST.labels()` load the dataset, fetching these from the online sources if not present on disk.

One-hot encoding corresponds to creating group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0). In the case of the MNIST labels, the value can only be one of 0 through 9; this is the action of `onehotbatch()`.

All these routines can be piped to the GPU is CUDA support is available. The pipeline acts as a NOOP if otherwise.

```
julia> m = Chain(Dense(28^2, 32, relu),
```

```
                    Dense(32, 10),
                    softmax) |> gpu
Chain(Dense(784, 32, NNlib.relu), Dense(32, 10), NNlib.softmax)
```

`relu` referes to a *rectifier logistic unit* is an activation function defined as the positive part of its argument, else zero.

At the end of the chain `softmax()` is applied to normalise the outputs.

```
# Accuracy is difference between the mean(X) and the (old) Y's
julia> accuracy(x, y) = mean(onecold(m(x)) .== onecold(y))
julia> dataset = repeated((X, Y), 200)

# Define an anonymous function to show the loss (cross-entropy)
julia> loss(x, y) = crossentropy(m(x), y)
julia> evalcb = () -> @show(loss(X, Y))
```

The loss function is defined in terms of evaluating the cross entropy between the layer inputs (*m(x)*) and the outputs (*y*). Cross entropy is a logistic classification method which measures measures the performance of a model whose output is a probability value between 0 and 1

```
julia> opt = ADAM()
ADAM(0.001, (0.9, 0.999), IdDict{Any,Any}())

julia> Flux.train!(loss, params(m), dataset, opt,
                         cb = throttle(evalcb, 10));
```

`ADAM()` , ADAptive Moment estimation, is a popular optimisation method which utilises a gradient descent algorithm.

```
# Test set accuracy
julia> tX =
     hcat(float.(reshape.(MNIST.images(:test), :))...) |> gpu;
julia> tY = onehotbatch(MNIST.labels(:test), 0:9) |> gpu;

julia> accuracy(tX, tY)
0.923
```

Running the model against the test set produces an accuracy of 92.3% after the model has been trained.

# Metalhead

Metalhead provides computer vision models that run on top of the Flux ML library.

Each model (like VGG19) is a Flux layer, and so any Flux procedure thing can be applied such as like moving it to the GPU, training or freezing components.

It includes support for working with several common object recognition datasets. The `datasets()` function tries to auto-detect any common dataset placed in the datasets folder.

Metalhead.download() can be used to download these datasets:

e.g. Metalhead.download(CIFAR10)

When loaded, training, validation, and test images can be performed by using the `trainimgs() valimgs()` and `testimgs()` routines.

Below is a an example using the VGG19 dataset.

VGG-19 is a convolutional neural network, which is trained on more than a million images from the ImageNet database, each image is of size 224 by 224. The network is 19 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

The code below references VGG19 and loads an image of an elephant, the left-hand image of the figure below, and apples the classify() routine to see what Metalhead makes of it.

```
julia> using Metalhead
julia> using Metalhead: classify
julia> using Images

# Switch to dataset images
julia> cd(homedir()*"/PacktPub/Datasets/Images")
julia> vgg = VGG19()
julia> img01 = load("elephant.jpg")
julia> classify(vgg, img01)
"African elephant, Loxodonta africana"
```

The image on the right-hand side is of my own cat, Harry, so I thought I would see what Metalhead makes of him.

```
julia> using Unicode
julia> img02 = load("harry.jpg")
julia> animal = uppercase(classify(vgg, img02))
julia> println("Meow, I don't think I am a $animal !!!")
Meow, I don't think I am a MINK !!!
```

We know that the animals sub-class consists of 1000 images, the first 5 are labelled as:

```
julia> animal_types = Metalhead.labels(vgg)
julia> animal_types[1:5]
5-element Array{String,1}:
 "tench, Tinca tinca"
 "goldfish, Carassius auratus"
 "great white shark, white shark, man-eater, man-eating shark, Carcharodon
carcharias"
 "tiger shark, Galeocerdo cuvieri"
 "hammerhead, hammerhead shark"
```

We can get the probabilities which are been applied when trying to classify an image.

For Harry these are listed below:

```
# How sure are we?
julia> probs = Metalhead.forward(vgg, img02)

# Sort the image probabilities: can easily pick off the largest
julia> sort(probs)[996:1000]
5-element Array{Float32,1}:
 0.00013183923
```

```
0.00014438889
0.0025139963
0.012086336
0.98500556
```

So to a 98.5% surety a single image has been identified and unsurprisingly this is the one corresponding to a mink. The high certainty value though is a surprise.

```
julia> perm = sortperm(probs)
julia> animal_types[perm[1000]]
"mink"
```

The only other animal identified as a possibility, and with a 1.2% surety, is :

```
julia> animal_types[perm[999]]
"schipperke"
```

A schipperke is a dog - *sorry Harry* - but at least it is black!

# Knet

Knet has been developed by Deniz Yuret and colleagues at Koç University (Istanbul). It is a deep learning framework implemented in Julia, purely in native code, similar in scope to Flux, preceding the latter and there is a degree of overlap in the examples furnished by the two frameworks.

Like Flux, Knet supports computation on the GPU operation when CUDA support is available. However Knet defines a Julia data type, KnetArray, which wraps GPU array pointers and does not require the CuArray module.

GPU memory allocation and transfer between the host and the GPU is (relatively) slow, Knet aims to minimise these bottlenecks and even when using native Julia code is competitive with popular C/C++ implementations.

| model | dataset | epochs | batch | Knet | Theano | Torch | Caffe | TFlow |
|---|---|---|---|---|---|---|---|---|
| LinReg | Housing | 10K | 506 | 2.84 | 1.88 | 2.66 | 2.35 | 5.92 |
| Softmax | MNIST | 10 | 100 | 2.35 | 1.40 | 2.88 | 2.45 | 5.57 |
| MLP | MNIST | 10 | 100 | 3.68 | 2.31 | 4.03 | 3.69 | 6.94 |
| LeNet | MNIST | 1 | 100 | 3.59 | 3.03 | 1.69 | 3.54 | 8.77 |
| CharLM | Hiawatha | 1 | 128 | 2.25 | 2.42 | 2.23 | 1.43 | 2.86 |

Knet incorporates AutoGrad(.jl), automatic differentiation package to enhance its performance. It can differentiate regular Julia code that includes loops, conditionals, helper

functions, closures etc. by keeping track of the primitive operations and using this execution trace to compute gradients and uses reverse mode differentiation (a.k.a. backpropagation) so as to efficiently handle functions with large array inputs and scalar outputs.

These in turn are used in coding a set of gradient descent optimization algorithms, an excellent review is given by `Sebastian Ruder`.

## Tutorial examples

Similar to Flux's Model-Zoo, Knet provides a large set of examples.

It has an excellent `documentation` website but rather than being on this they are part of the Knet.jl `github` source, being found in the examples and data folders.

The approach is different.

Each example has its own folders (*again*) but there is no attempt to classify these by type. In each folder there is a notebook and a script. The scripts can be run from the REPL or the (shell) command line. They will download the datasets - if they do not exist - using associated routines in the data folder.

Running the familiar MNIST example is shown below.

```
julia> using Knet, Images
julia> cd(Knet.dir("examples"))
julia> MLP.main("--help")
usage: <PROGRAM> [--seed SEED] [--batchsize BATCHSIZE]
  [--epochs EPOCHS] [--hidden [HIDDEN...]] [--lr LR]
  [--winit WINIT] [--fast] [--atype ATYPE] [--gcheck GCHECK]

mlp.jl (c) Deniz Yuret, 2016. Multi-layer perceptron model on the MNIST
handwritten digit recognition problem from
http://yann.lecun.com/exdb/mnist.

optional arguments:
  --seed SEED random number seed: use a nonnegative int for
             repeatable results (type: Int64, default: -1)
  --batchsize BATCHSIZE
              minibatch size (type: Int64, default: 100)
  --epochs EPOCHS number of epochs for training
           (type: Int64, default: 10)
  --hidden [HIDDEN...] sizes of hidden layers,
           e.g. --hidden 128 64
           for a net with two hidden layers (type: Int64)
```

```
  --lr LR learning rate (type: Float64, default: 0.5)
  --winit WINIT w initialized with winit*randn()
          (type: Float64, default: 0.1)
  --fast skip loss printing for faster run
  --atype ATYPE array type: Array for cpu, KnetArray for gpu
           (default: "Array{Float32}")
  --gcheck GCHECK check N random gradients per parameter
           (type: Int64, default: 0)
```

Each script has a `--help` switch which displays information on command options. There are a variety of options available and to manage the parsing of the command line ArgParse is used. This is quite a heavyweight module, i.e. takes a significant time to compile, possibly of the order of 1-2 sec.; this is fine for scripts which will have medium to long execution times but not for 'quickies'. Recall that with a script any additional packages will be recompiled at each execution.

We will be looking at the ArgParse package and alternatives when discussing Julia scripting in the final chapter of the book.

Running the 'main' routine again over 15 epochs, rather than the default 10 :

```
julia> MLP.main("--epochs 15")
opts=(:batchsize, 100)(:fast, false)(:atype, "Array{Float32}")
(:epochs, 15)(:gcheck, 0)(:winit, 0.1)(:lr, 0.5)
(:hidden, Int64[])(:seed, -1)

(:epoch, 0, :trn, 0.08636666666666666, :tst, 0.0875)
(:epoch, 1, :trn, 0.8992666666666667, :tst, 0.9025)
. . . . . .
. . . . . .
(:epoch, 14, :trn, 0.9211833333333334, :tst, 0.9162)
(:epoch, 15, :trn, 0.9216, :tst, 0.9166)
 13.80164 seconds (11.93M allocations: 6.979GiB, 6.39% gc time)
```

If we dissect the run, we can execute the data-loading routine. Because the data has already been download it will not be done so again.

We can map the `summary()` routine to the data arrays to show that the training data (*xtrn*) and the labels (*ytrn*) are of size 60000 and the test datasets (xtst/ytst) of size 10000.

Also the imagery is 32-bit (not 64) greyscale and the labels 8-bit.

```
julia> include("../data/mnist.jl")
julia> xtrn,ytrn,xtst,ytst = MLP.mnist()
julia> map(summary,(xtrn,ytrn,xtst,ytst))
```

```
("28×28×1×60000 Array{Float32,4}",
 "60000-element Array{UInt8,1}",
 "28×28×1×10000 Array{Float32,4}",
 "10000-element Array{UInt8,1}")
```

The Images package can be used to display images; the first 5 are shown in the figure below.

```
julia> knetview(x,i) =
     Images.colorview(Gray,permutedims(x[:,:,1,i],(2,1)))
julia> hcat([knetview(xtst,i) for i=1:5]...)
```

```
julia> ytst[1:5]'
1×5 LinearAlgebra.Adjoint{UInt8,Array{UInt8,1}}:
 0x07 0x02 0x01 0x0a 0x04
```

Here we can see that the fourth image in the test set could be a '0' or a '6'. If we look at the associated labels file we see that is a '0', *zero is encoded to be in category 10 (0x0a).*

A second example applies the `LeNet-5` model to the same, MNIST, dataset. This is a convolutional algorithm and was an early approach used by Yann *etal* in their original work. It is much slower than the more recent models.

```
bash-3.2$ julia lenet.jl --epochs 5
opts=(:batchsize, 128)(:fast, false)
     (:iters, 9223372036854775807)(:atype, "Array{Float32}")
     (:epochs, 5)(:gcheck, 0)(:lr, 0.1)(:seed, -1)
lenet.jl (c) Deniz Yuret, 2016. The LeNet model on the MNIST handwritten
digit recognition problem
CPU conv4 support is experimental and very slow.
. . . . . .
. . . . . .
45.074974 seconds (364.21 k allocations: 19.806 GiB, 4.61% gc time)
(:epoch, 5, :trn, 0.9878806089743589, :tst, 0.9845753205128205)

# Average around 45.08 sec per epoch
```

The machine I am using for this work is a MacPro with only a CPU. But I ran the above model on another (Windows) computer which has a GeoForce GTX 960 chipset.

The average execution time per epoch when using the CPU was 53.515 sec., whereas it

reduced to 5.161 sec. running on the GPU, an increase of around 10.4

When discussing the MLDatasets, we noted that there were also similar sets of street view house numbers and of articles of clothing.

The run below is of the fashion-mnist data. Again there are 60000 images, 50K used for training and 10K for testing , and being identified as being in one of 10 categories:

```
julia> include("fashion-mnist/fashion-mnist.jl")
julia> FashionMNIST.main("--epochs 20 --hidden 32 32")
opts=(:batchsize, 100)(:fast, false)(:atype, "Array{Float32}")
(:epochs, 20)(:gcheck, 0)(:winit, 0.1)
(:dropout, 0.5)(:lr, 0.15)
(:hidden, [32, 32])(:seed, -1)
(:epoch, 0, :trn, 0.10636666666666666, :tst, 0.1054)
(:epoch, 1, :trn, 0.8191166666666667, :tst, 0.8075)
. . . . . .
. . . . . .
(:epoch, 19, :trn, 0.8941, :tst, 0.867)
(:epoch, 20, :trn, 0.8947333333333334, :tst, 0.8673)
 18.23299 seconds (7.62M allocations: 12.181GiB, 8.78% gc time
```

This time the machine is less successful in distinguishing the various items but this can can be envisaged as a more difficult task than hand-written digits and that the value close to 90% is not too bad.

```
julia> hcat([knetview(xtst,i) for i=1:10]...)
```

The final example chosen is the VGG classification which we saw earlier in Flux's Minehead package. The target image here is a **cat** and this time (unlike Harry) the classification is much better, although it is unable to decide on the breed and a couple of foxes get in the too.-- however there is no sign of any minks, or dogs.

```
$> julia vgg.jl
vgg.jl (c) Deniz Yuret, İlker Kesen, 2016.
Classifying images with the VGG model from
http://www.robots.ox.ac.uk/~vgg/research/very_deep.
opts=(:atype, "Array{Float32}")(:top, 5)
(:image,"https://github.com/BVLC/caffe/raw/master/examples/images/cat.jpg")
(:model, "imagenet-vgg-verydeep-16")
[ Info: Loading imagenet-vgg-verydeep-16.mat ...
[ Info: Classifying
 2.301254 seconds (2.81M allocations: 606.690MiB, 5.83% gc time)
5×2 Array{Any,2}:
 0.273299 "tabby, tabby cat"
 0.253971 "Egyptian cat"
 0.247885 "tiger cat"
 0.0607373 "kit fox, Vulpes macrotis"
 0.0562257 "red fox, Vulpes vulpes"
```

# Tensor Flow

TensorFlow.jl is a package due to Jon Malmaud. It is a *wrapper* around Google's `TensorFlow` library used to implement deep-learning models.

Tensot Flow has proved popular in the Python world, since it provides reasonable execution speeds. With Julia this is less important since Julia's native code runs quickly and

packages such as `Flux` and `Knet` an interface to the GPU when present.

The Julia module implementation is very similar to the Python API.

Despite both the Julia and Python (or R) wrappers executing at the same rate on the GPU, Julia can be seen has having significant advantages when there is also significant computation on the host (which is often the case) since:

- Julia is quicker for input and munging data, especially when in uncommon formats; also have can use the Query Verse suite.
- Julia is much faster if post-processing of results on the CPU is needed, such as may be necessary for calculating statistics and graphics
- Use of multiple dispatch is efficient in creating models with native (Julia) code, since there is no overhead on routines which operate on complex structures when compared to simple functions

Most function names and arguments are semantically the same. However the Python API uses an object-oriented notation like `session.run(node)` whereas the Julia version is `run(session, node)`.

In the package documentation, an example on training the `MNIST` dataset is included

There may be difficulties in using `Tensorflow,jl` when a GPU is NOT present as the fallback to the CPU may result in error message:

*"Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA"*

# Summary

In our review of Julia's approaches to statistical modelling, Bayesian inference and neural networks analysis when applied to machine learning.

The topic is huge and the aim has been to outlined some of the important packages which are now version 1.0 compliant and to indicate via some of the simpler examples how these can be use.

The reader is encouraged to read further and a number of online references have been given and additional material and examples is provided in the files accompanying this

chapter.

# Index