Julia for Machine Learning

DG Wilson

- d9w.xyz
- d9w@pm.me

Outline

- What is Julia
- Data representation
- Statistics
- Machine Learning

What is Julia?

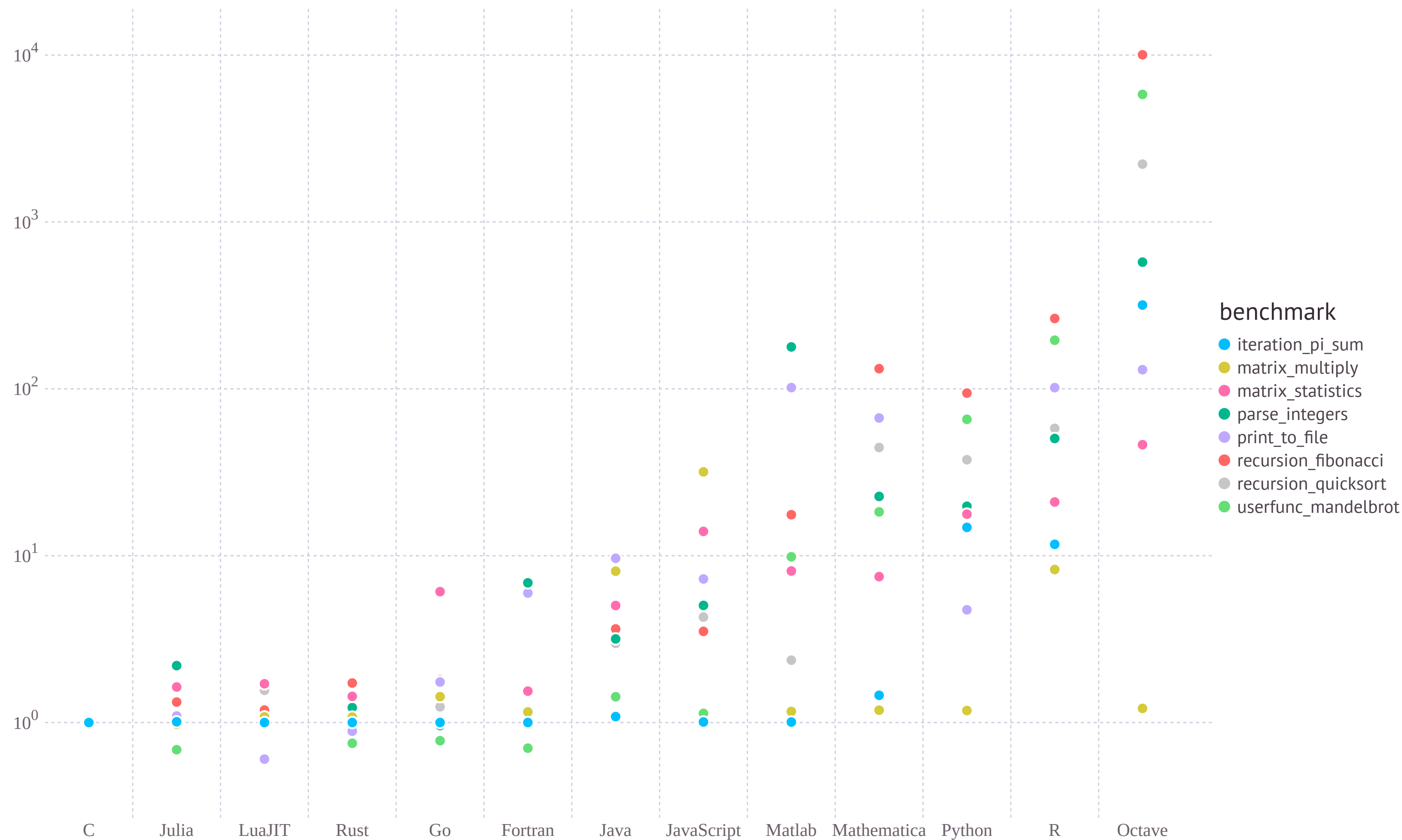
Julia is interactive

Julia can be run in the REPL in a terminal, in Jupyter (like this), or on a julia script (`script.jl`)

In [1]:
```
println("Hello World")
```

```
Hello World
```

Julia is fast



Julia is compiled

Julia uses Just-in-time (JIT) compilaton, implemented using LLVM. This means that code is compiled the first time that it is called. Here we define a Fibonacci function, but it is not yet compiled

In [8]:
```
fib(n) = n < 2 ? n : fib(n-1) + fib(n-2)
```

Out[8]:
```
fib (generic function with 1 method)
```

The first time that we run the function, it will take longer to compile. However, after compilation, the function will be much faster

In [9]:
```
@timev fib(10)
```

```
  0.003984 seconds (3.55 k allocations: 196.864 KiB)
elapsed time (ns): 3983782
bytes allocated:   201589
pool allocs:       3554
```

Out[9]:
```
55
```

In [10]:
```
@timev fib(10)
```

```
  0.000005 seconds (4 allocations: 160 bytes)
elapsed time (ns): 4638
bytes allocated:   160
pool allocs:       4
```

Out[10]:
```
55
```

We can also evaluate the machine code which the function compiles to

In [5]:
```
@code_lowered fib(10)
```

Out[5]:
```
CodeInfo(
│1 1 ─ %1 = n < 2
│  └──       goto #3 if not %1
│  2 ─       return n
│  3 ─ %4 = n - 1
│  │   %5 = (Main.fib)(%4)
│  │   %6 = n - 2
│  │   %7 = (Main.fib)(%6)
│  │   %8 = %5 + %7
│  └──       return %8
)
```

In [6]:
```
@code_native fib(10.2)
```

```
        .text
; Function fib {
; Location: In[2]:1
        pushq   %rbx
        subq    $16, %rsp
        vmovapd %xmm0, %xmm1
        movabsq $140100925897896, %rax  # imm = 0x7F6BC9EBBCA8
; Function <; {
; Location: float.jl:497
; Function <; {
; Location: float.jl:452
        vmovsd  (%rax), %xmm0           # xmm0 = mem[0],zero
        vucomisd        %xmm1, %xmm0
;}}
        ja      L99
        movabsq $140100925897904, %rax  # imm = 0x7F6BC9EBBCB0
; Location: In[2]:1
; Function -; {
; Location: promotion.jl:315
; Function -; {
; Location: float.jl:397
        vaddsd  (%rax), %xmm1, %xmm0
;}}
        movabsq $fib, %rbx
        vmovsd  %xmm1, (%rsp)
        callq   *%rbx
        vmovsd  %xmm0, 8(%rsp)
        movabsq $140100925897912, %rax  # imm = 0x7F6BC9EBBCB8
; Function -; {
; Location: promotion.jl:315
; Function -; {
; Location: float.jl:397
        vmovsd  (%rsp), %xmm0           # xmm0 = mem[0],zero
        vaddsd  (%rax), %xmm0, %xmm0
```

```
;}}
        callq   *%rbx
; Function +; {
; Location: float.jl:395
        vaddsd  8(%rsp), %xmm0, %xmm0
;}
        addq    $16, %rsp
        popq    %rbx
        retq
L99:
        vmovapd %xmm1, %xmm0
        addq    $16, %rsp
        popq    %rbx
        retq
        nopl    (%rax)
;}
```

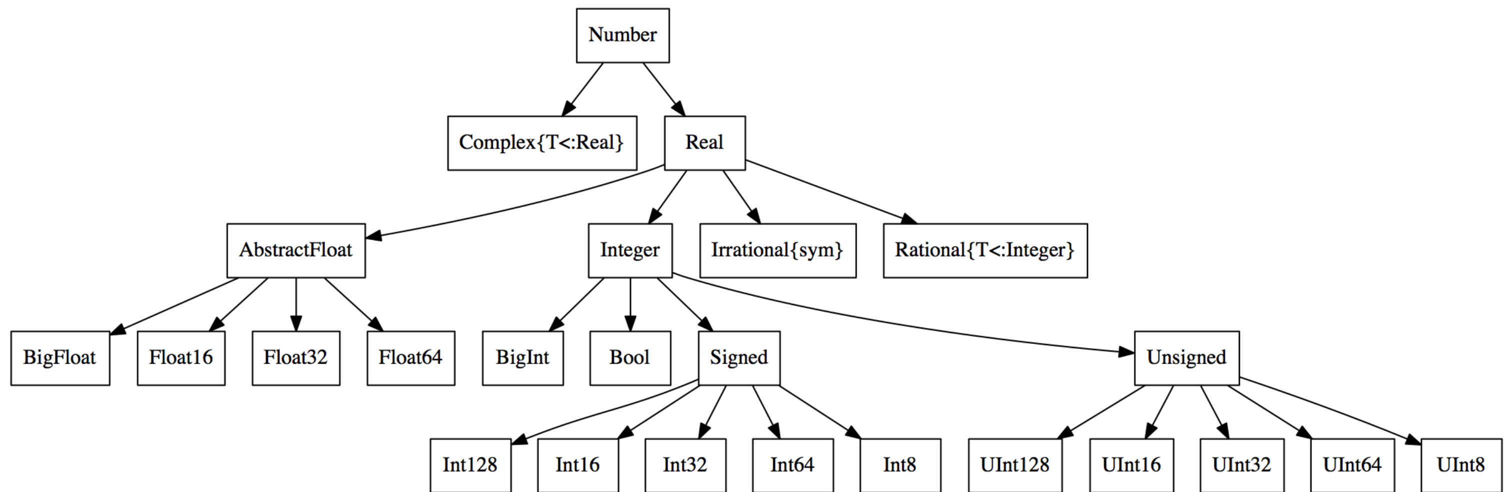
Julia is typed

Julia's type system is

- dynamic (types are checked at runtime)
- nominative (variables rely on explicity type declaration for compatibility)
- parametric (generic types can be parameterized)

Abstract types can be defined and subclassing, allowing for flexibility of not declaring a type. Every type in Julia is a subclass of the `Any` type.

Below is the type heirarchy of the Base Types (as of Julia 0.5, may not be current):



In [7]:
```
Integer <: Number
```
Out[7]:
```
true
```
In [8]:
```
Integer <: AbstractFloat
```
Out[8]:
```
false
```

We can define composite types, like a Class or object in different languages.

In [9]:
```
mutable struct Foo
    bar::Int
    baz
end
```

In Julia, functions are not part of the class definition, as they are in C++ or Python. Instead, only the values of the composite type are defined. We can define default constructors separately, or other functions:

In [10]:
```
Foo() = Foo(10, "Hello")

function Foo(x::Int)
    Foo(x, nothing)
end

function double!(x::Foo)
    x.bar *= 2
end
```

Out[10]:
```
double! (generic function with 1 method)
```

In [11]:
```
f = Foo()
g = Foo(10)
println("f: ",f)
println("g: ", g)
double!(f)
println("f: ", f)
```

```
f: Foo(10, "Hello")
g: Foo(10, nothing)
f: Foo(20, "Hello")
```

Julia is flexible

Multiple dispatch is at the base of Julia, allowing for both object oriented and functional programming methods

In [12]:

```
methods(+)
```

Out[12]:

174 methods for generic function **+**:

- +(x::**Bool**, z::**Complex{Bool}**) in Base at complex.jl:277 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(x::**Bool**, y::**Bool**) in Base at bool.jl:104 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(x::**Bool**) in Base at bool.jl:101 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +{T<:AbstractFloat}(x::**Bool**, y::**T**) in Base at bool.jl:112 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(x::**Bool**, z::**Complex**) in Base at complex.jl:284 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(a::**Float16**, b::**Float16**) in Base at float.jl:392 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(x::**Float32**, y::**Float32**) in Base at float.jl:394 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(x::**Float64**, y::**Float64**) in Base at float.jl:395 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(z::**Complex{Bool}**, x::**Bool**) in Base at complex.jl:278 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(z::**Complex{Bool}**, x::**Real**) in Base at complex.jl:292 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544
- +(::**Missing**, ::**Missing**) in Base at missing.jl:92 (https://github.com/JuliaLang/julia/tree/a4cb80f3edcf8cea00bd9660e3b65f544

In [13]:
```
f + g
```

```
MethodError: no method matching +(::Foo, ::Foo)
Closest candidates are:
  +(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:502

Stacktrace:
 [1] top-level scope at In[13]:1
```

In [14]:
```
import Base.+
+(a::Foo, b::Foo) = Foo(a.bar + b.bar)
```

Out[14]:
```
+ (generic function with 175 methods)
```

In [15]:
```
println("f: ", f)
println("g: ", g)
f + g
```

```
f: Foo(20, "Hello")
g: Foo(10, nothing)
```

Out[15]:
```
Foo(30, nothing)
```

Julia is compatible

Julia natively works with C and Fortran. Packages exist to interface with C++, Python, MATLAB, Java, and more.

In [21]:

```
using PyCall
```

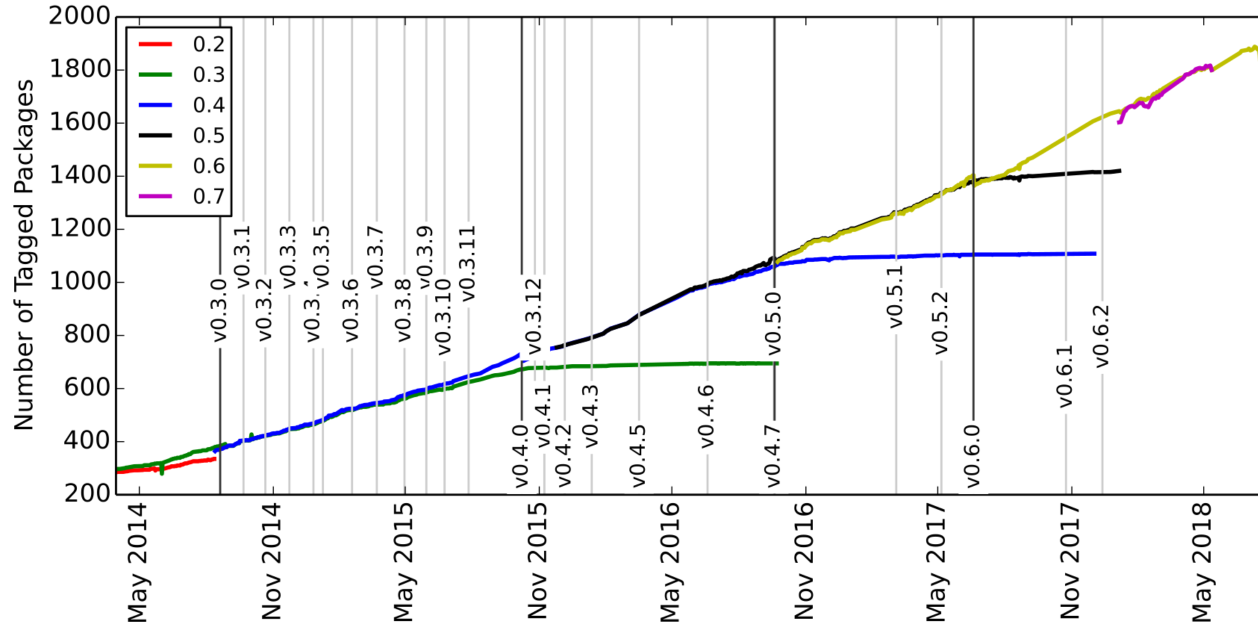
In [22]:

```
@pyimport scipy.optimize as so
so.newton(x -> cos(x) - x, 1)
```

Out[22]:

```
0.7390851332151607
```

Julia is growing



Data in Julia

Some (but not all!) popular packages for data representation, manipulation, and visualization

Database Interaction

C wrappers and full Julia implementations for many databases, such as

- SQLite.jl
- MySQL.jl
- Mongo.jl
- LibPQ.jl

In [30]:
```
using DataFrames
```

DataFrames.jl

Similar to pandas in Python, DataFrames is a library for data representation and manipulation

In [31]:
```julia
names = DataFrame(ID = [20, 40], Name = ["John Doe", "Jane Doe"])
```

Out[31]:

	ID	Name
	Int64	**String**
1	20	John Doe
2	40	Jane Doe

In [32]:
```julia
jobs = DataFrame(ID = [20, 40], Job = ["Lawyer", "Doctor"])
```

Out[32]:

	ID	Job
	Int64	**String**
1	20	Lawyer
2	40	Doctor

A DataFrame isn't a matrix, it operates more like a database. For example, you can do joins with DataFrames

In [33]:
```julia
join(names, jobs, on = :ID)
```

Out[33]:

	ID	Name	Job
	Int64	**String**	**String**
1	20	John Doe	Lawyer
2	40	Jane Doe	Doctor

In [34]:

```
using RDatasets
```

RDatasets.jl

Many sample datasets that are included in R and others that are popular in R

In [35]:
```
iris = dataset("datasets", "iris")
```

Out[35]:

	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
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
17	5.4	3.9	1.3	0.4	setosa
18	5.1	3.5	1.4	0.3	setosa
19	5.7	3.8	1.7	0.3	setosa
20	5.1	3.8	1.5	0.3	setosa
21	5.4	3.4	1.7	0.2	setosa
22	5.1	3.7	1.5	0.4	setosa
23	4.6	3.6	1.0	0.2	setosa
24	5.1	3.3	1.7	0.5	setosa
25	4.8	3.4	1.9	0.2	setosa
26	5.0	3.0	1.6	0.2	setosa
27	5.0	3.4	1.6	0.4	setosa
28	5.2	3.5	1.5	0.2	setosa
29	5.2	3.4	1.4	0.2	setosa
30	4.7	3.2	1.6	0.2	setosa
⋮	⋮	⋮	⋮	⋮	⋮

In [36]:

```
sort!(iris, :PetalLength)
ismall = head(iris, 4)
```

Out[36]:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64	**Float64**	**Float64**	**Float64**	**Categorical...**
1	4.6	3.6	1.0	0.2	setosa
2	4.3	3.0	1.1	0.1	setosa
3	5.8	4.0	1.2	0.2	setosa
4	5.0	3.2	1.2	0.2	setosa

In [37]:
```
using Query
```

Query.jl

Allows for querying many data structures, including DataFrames, to create new DataFrames or matrices

In [38]:
```
queried = @from i in ismall begin
    @where i.SepalWidth < 4.2 && i.SepalLength > 5.4
    @select {i.SepalWidth, i.SepalLength, i.Species}
    @collect DataFrame
end
```

Out[38]:

	SepalWidth	SepalLength	Species
	Float64	Float64	Categorical...
1	4.0	5.8	setosa

We can also do this with logical indexing, using the different columns as Arrays

In [39]:
```
x = ismall[:SepalWidth] .< 4.2
```

Out[39]:
```
4-element BitArray{1}:
 true
 true
 true
 true
```

In [40]:
```
y = ismall[:SepalLength] .> 5.4
```

Out[40]:
```
4-element BitArray{1}:
 false
 false
  true
 false
```

In [41]:
```
indices = x .* y
```

Out[41]:
```
4-element BitArray{1}:
 false
 false
  true
 false
```

In [42]:
```
queried
```

Out[42]:

	SepalWidth	SepalLength	Species
	Float64	Float64	Categorical...
1	4.0	5.8	setosa

In [43]:
```
ismall[:SepalLength][indices]
```

Out[43]:
```
1-element Array{Float64,1}:
 5.8
```

In [44]:

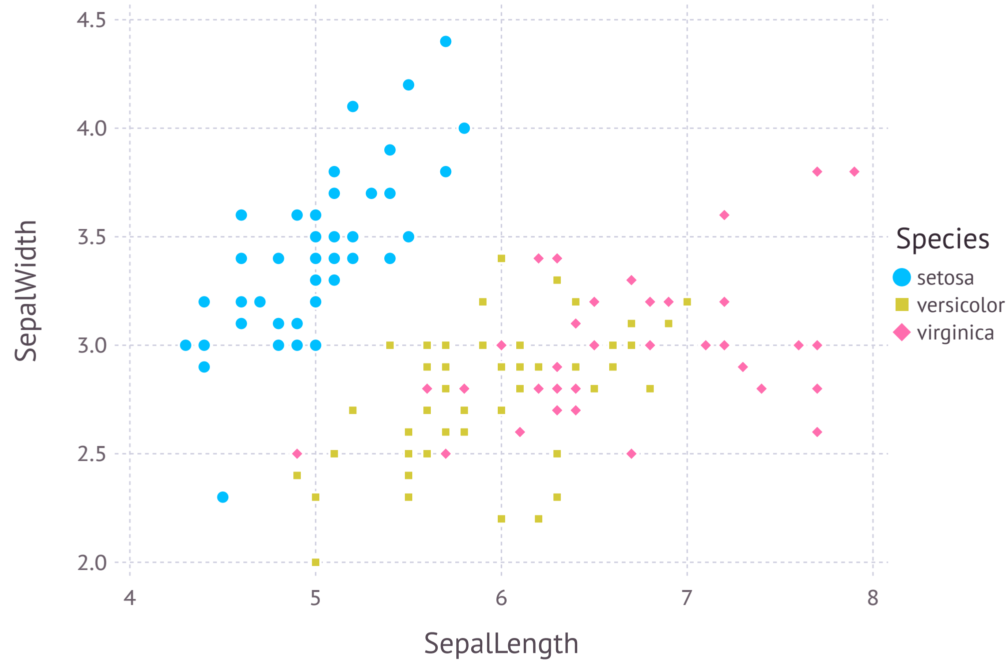
```
using Gadfly
```

Gadfly.jl

A popular pure-Julia data visualization package. Other options include PyPlot.jl (wrapper of matplotlib), GR.jl (wrapper of GR), and Plots.jl (meta-wrapper)

In [45]:
```
plot(iris, x="SepalLength", y="SepalWidth", color="Species", shape="Species", Ge
om.point)
```

Out[45]:



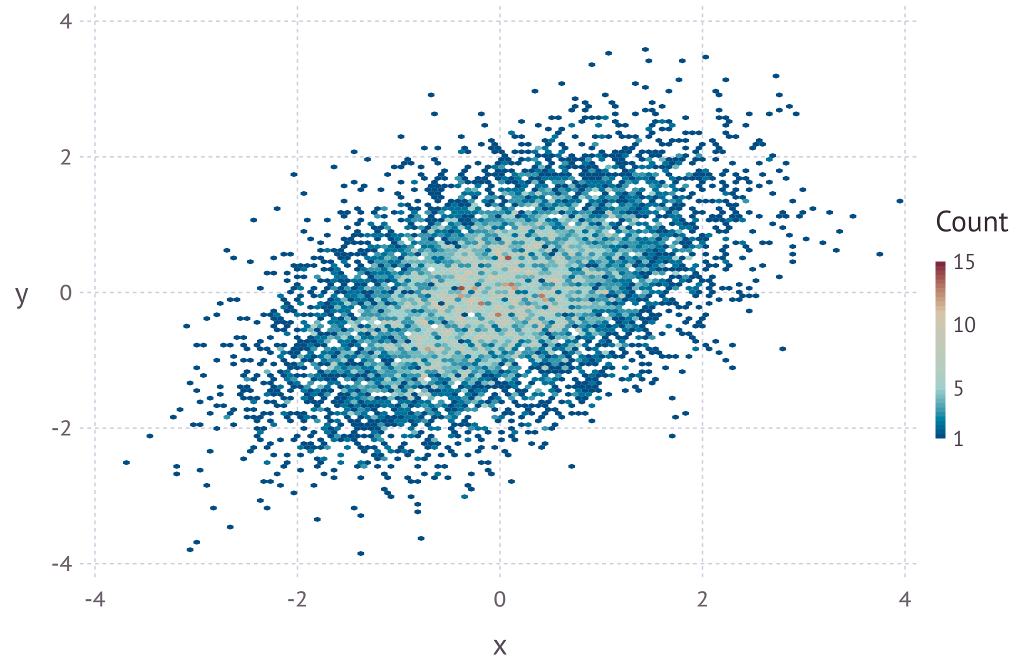
In [46]:
```
using Distributions
```

In [47]:
```
X = rand(MultivariateNormal([0.0, 0.0], [1.0 0.5; 0.5 1.0]), 10000);
println(X[1:10])
```

```
[1.28396, 0.786187, -0.0201795, 0.995887, 0.449988, -0.0982218, -0.833681, -0.
505184, 1.80533, 1.99911]
```

In [48]:
```
plot(x=X[1,:], y=X[2,:], Geom.hexbin)
```

Out[48]:



In [ ]:

Statistics in Julia

Julia has been used by mathematicians primarily over its history and therefore has a rich mathematic ecosystem

In [32]:
```
using Distributions
```

Distributions.jl

Used to generate data according to distributions (as seen in the previous section) or to fit distributions to data

In [33]:
```
using DataFrames, RDatasets
iris = dataset("datasets", "iris");
```

We can use Maximum Likelihood Estimation to fit a distribution

In [34]:
```
X = iris[:SepalLength]
d = fit_mle(Normal, X)
```

Out[34]:
```
Normal{Float64}(μ=5.843333333333335, σ=0.8253012917851409)
```

To compare the result, we can generate data from this distribution and calculate the mean squared error.

In [35]:
```
y = rand(d, length(X));
sum((X .- y).^2)
```

Out[35]:
```
180.43261776491047
```

In [36]:
```
using GLM
```

GLM.jl

Generalized Linear Models for linear regression. We'll look at ordinary least squares regression

In [37]:
```
species = unique(iris[:Species])
iris[:Sind] = [indexin([i], species)[1] for i in iris[:Species]]
head(iris)
```

Out[37]:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species	Sind
	Float64	Float64	Float64	Float64	Categorical...	Int64
1	5.1	3.5	1.4	0.2	setosa	1
2	4.9	3.0	1.4	0.2	setosa	1
3	4.7	3.2	1.3	0.2	setosa	1
4	4.6	3.1	1.5	0.2	setosa	1
5	5.0	3.6	1.4	0.2	setosa	1
6	5.4	3.9	1.7	0.4	setosa	1

In [38]:
```
ols = lm(@formula(Sind ~ PetalWidth), iris)
```

Out[38]:
```
StatsModels.DataFrameRegressionModel{LinearModel{LmResp{Array{Float64,1}},Dens
ePredChol{Float64,LinearAlgebra.Cholesky{Float64,Array{Float64,2}}}},Array{Flo
at64,2}}

Formula: Sind ~ 1 + PetalWidth

Coefficients:
             Estimate Std.Error t value Pr(>|t|)
(Intercept)  0.767001 0.0365708 20.9731   <1e-45
PetalWidth    1.02807 0.0257596 39.9102   <1e-80
```

In [39]:
```
stderror(ols)
```

Out[39]:
```
2-element Array{Float64,1}:
 0.036570760522212836
 0.02575959315195414
```

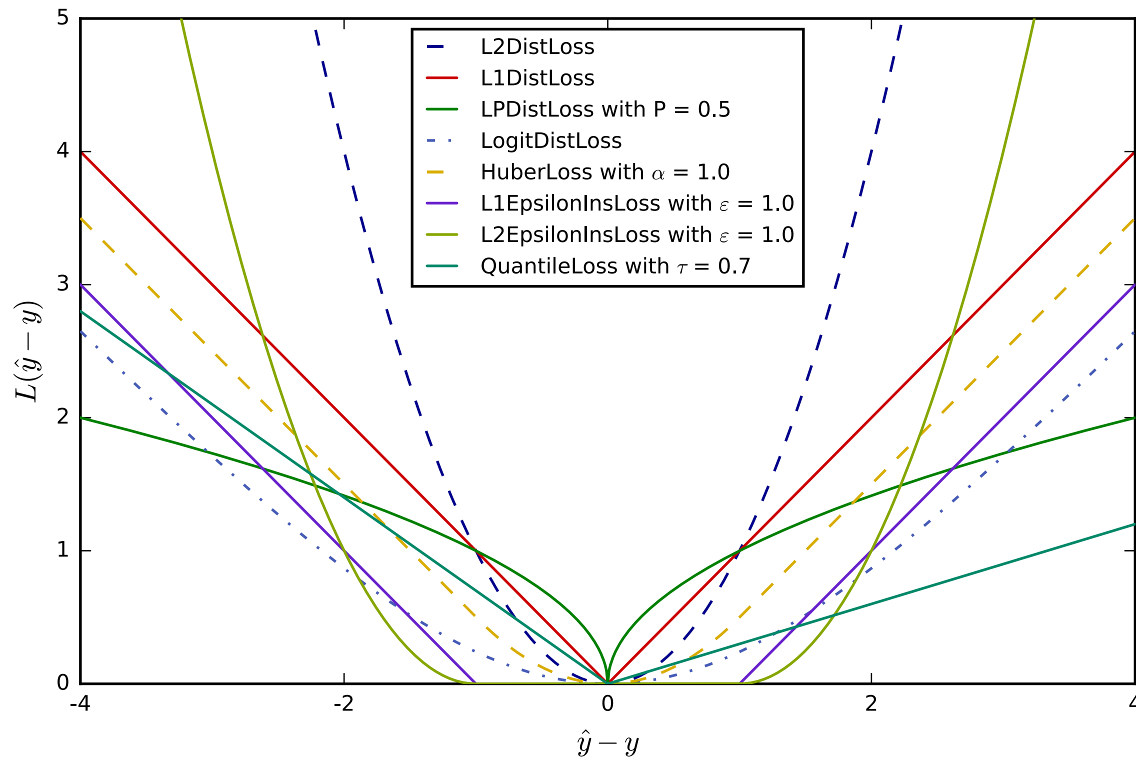
Machine Learning

As with the previous section, we'll look at some, but not all, popular machine learning packages in Julia. This part of the ecosystem is under heavy active development currently and could use more full-Julia options.

In [40]:
```julia
using LossFunctions
```

LossFunctions.jl

Provides a variety of loss functions for classification and regression tasks



In [41]:
```
h = predict(ols)
h[1:10]
```

Out[41]:
```
10-element Array{Float64,1}:
 0.97261481853977
 0.97261481853977
 0.97261481853977
 0.97261481853977
 0.97261481853977
 1.178228930906727
 1.0754218747232487
 0.97261481853977
 0.97261481853977
 0.8698077623562915
```

In [42]:
```
sum(value(LogitDistLoss(), iris[:Sind], h))
```

Out[42]:
```
2.107057372318687
```

In [43]:

```
using DecisionTree
```

DecisionTree.jl

Provides Decision Trees and Random Forests, with a scikit-learn based API

In [44]:
```
features = convert(Array, iris[1:4])
labels = string.(iris[:Species])
model = DecisionTreeClassifier(max_depth=2)
```

Out[44]:
```
DecisionTreeClassifier
max_depth:                2
min_samples_leaf:         1
min_samples_split:        2
min_purity_increase:      0.0
pruning_purity_threshold: 1.0
n_subfeatures:            0
classes:                  root:

nothing
nothing
```

In [45]:
```
DecisionTree.fit!(model, features, labels)
print_tree(model)
```

```
Feature 3, Threshold 2.45
L-> setosa : 50/50
R-> Feature 4, Threshold 1.75
    L-> versicolor : 49/54
    R-> virginica : 45/46
```

In [46]:

```
y = DecisionTree.predict(model, features);
sum(y .!= iris[:Species])
```

Out[46]:

```
6
```

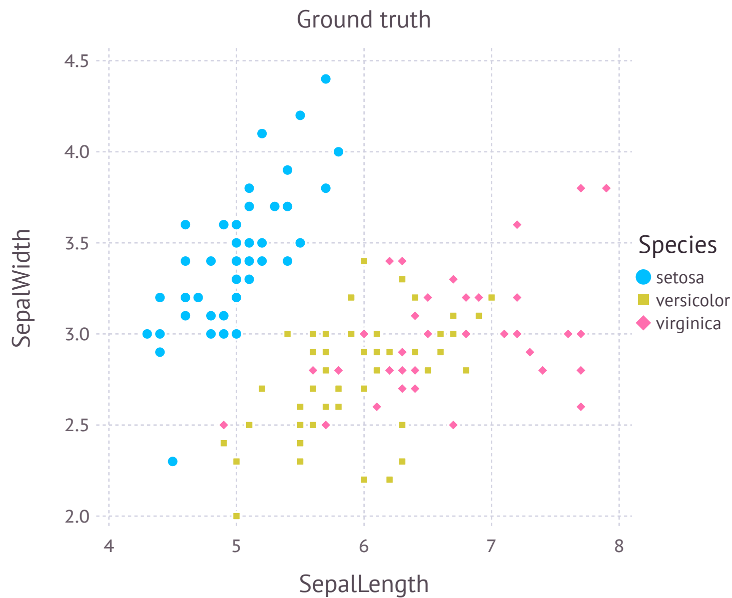
In [47]:

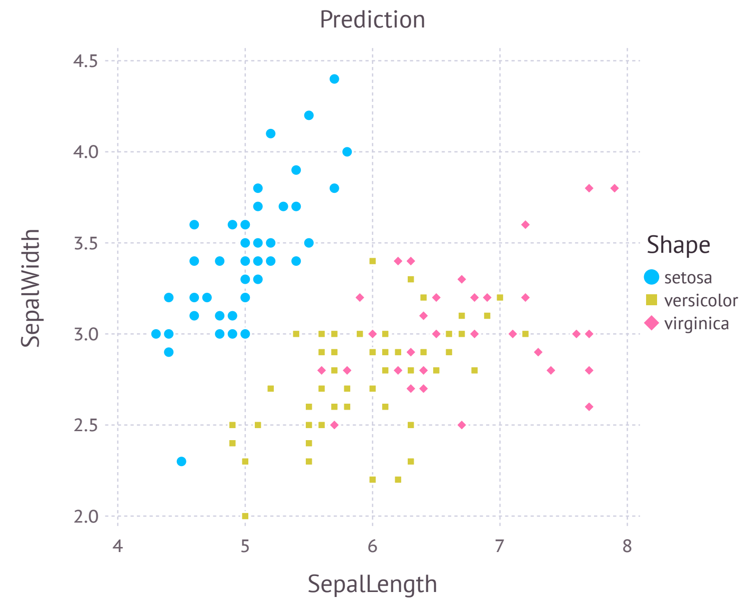
```
using Gadfly
```

In [48]:

```
p1 = plot(iris, x="SepalLength", y="SepalWidth", color="Species", shape="Species", Geom.point,
    Guide.title("Ground truth"));
p2 = plot(iris, x="SepalLength", y="SepalWidth", color=y, shape=y, Geom.point,
Guide.title("Prediction"));
set_default_plot_size(700pt, 300pt)
hstack(p1, p2)
```

Out[48]:





Deep Learning

Multiple new options being actively developed

- Mocha.jl
- Knet.jl
- Flux.jl
- Tensorflow.jl (wrapper of Python)

Native GPU support is available through CUDAnative.jl

In [ ]:

```
using Flux

model = Chain(
  Dense(768, 128, σ),
  LSTM(128, 256),
  LSTM(256, 128),
  Dense(128, 10),
  softmax)

loss(x, y) = crossentropy(model(x), y)

Flux.train!(loss, data, ADAM(...))
```

Thank you!

https://github.com/d9w/julia_presentation.git
(https://github.com/d9w/julia_presentation.git)