k-Nearest Neightbors

ref. from book **"Data Science from Scratch"**, Chap 12

```
begin
    using Test
    using Random
    using RDatasets
    using DataFrames
    using Plots
    # using Gadfly

    push!(LOAD_PATH, "./src")
    using YaLinearAlgebra
```

TOC

back to TOC

The model

Nearest makes no mathematical assumption and only requires two elements:

1. A distance
2. An assumption that points close to each other (given the distance) are similar.

The prediction for a given data point depends only on a few datapoints (k of them) "close" to it.

In this notebook the data points will be vector and the distance will be the Euclidean one.

Assuming we pick a number k, we then want to classify a new data point by finding the k nearest ones and let them vote on the new ouptut. To achieve this we need a function that counts the votes.

```
most_common (generic function with 1 method)
```

```
begin
    function raw_majority_vote(labels::AbstractVector{Symbol})::Symbol
        votes = count(labels)
        most_common(votes; k=1)[1]
```

```
    end

    function count(labels::AbstractVector{Symbol})::Dict{Symbol, Integer}
        hcnt = Dict{Symbol, Integer}()
        for label ∈ labels
            hcnt[label] = get(hcnt, label, 0) + 1
        end
        hcnt
    end

    function most_common(hcnt::Dict{Symbol, Integer};
            k::Integer=1)::Pair{Symbol,Integer}
        collect(hcnt) |> l -> sort(l, by=t -> t[2]; rev=true)[k]
    end
```

```
Test Passed
```

```
begin
    h = count([:b, :a, :b, :a, :c, :b, :d])
    @test h[:b] == 3
    @test h[:a] == 2
    @test h[:c] == h[:d] == 1

    @test sort(collect(h), by=t -> t[2]; rev=true)[1] == (:b => 3)

    @test raw_majority_vote([:b, :a, :b, :a, :c, :b, :d]) == :b
```

We need to take into account the possibility of ties. If this happens we will reduce k until we find a winner which will happen eventually. In the worse case we will go down until k equals 1.

```
majority_vote (generic function with 1 method)
```

```
function majority_vote(labels::AbstractVector{Symbol})::Symbol
    """
    Assuming labels is ordered from nearest to farthest
    """
    vote_counts = count(labels)
    winner, winner_cnt = most_common(vote_counts)
    num_winners = length([
        cnt for cnt ∈ values(vote_counts) if cnt == winner_cnt
    ])

    num_winners == 1 && (return winner)
    majority_vote(view(labels, 1:length(labels)-1))
```

```
Test Passed
```

```
begin
    @test majority_vote([:b, :a, :b, :a, :c, :b, :d]) == :b
    @test majority_vote([:b, :a, :b, :a, :c, :b, :d, :a]) == :b
    @test majority_vote([:b, :a, :b, :a, :c, :a, :d, :b]) == :a
```

Let us now create our classifer applying knn.

```
knn (generic function with 1 method)
```

```
begin
    const VF = AbstractVector{T} where T <: AbstractFloat

    struct LabeledPoint
        point::VF
        label::Symbol

        LabeledPoint(point::VF, sym::Symbol) = new(point, sym)
```

```
    end

    function knn(k::Integer, dataset::Vector{LabeledPoint}, qpoint::VF)::Symbol
        ## order from nearest to farthest using euclidean distance
        bydist = sort(dataset, by=lp -> distance(lp.point, qpoint), rev=false)

        ## find label from k closest
        k_near_labels = map(lp -> lp.label, view(bydist, 1:k))

        ## vote and return decision
        majority_vote(k_near_labels)
    end
```

Let us test this on an existing dataset.

back to TOC

Example with this Iris Dataset

We will use the Iris dataset which is kind of the "hello wolrd" of machine learning. It contains a set of measurements for 150 flowers representing three species of iris. For each flower we have its petal length, petal width, sepal length, and sepal width, as well as its species.

	variable	eltype	first
1	:SepalLength	Float64	5.1
2	:SepalWidth	Float64	3.5
3	:PetalLength	Float64	1.4
4	:PetalWidth	Float64	0.2
5	:Species	CategoricalValue{String,UInt8}	CategoricalValue{String,UInt8} "set

```
begin
    iris_ds = dataset("datasets", "iris");
    DataFrames.describe(iris_ds, :eltype, :first => first)
```

```
measurements_mx = 150×4 Array{Float64,2}:
 5.1  3.5  1.4  0.2
 4.9  3.0  1.4  0.2
 4.7  3.2  1.3  0.2
 4.6  3.1  1.5  0.2
 5.0  3.6  1.4  0.2
 5.4  3.9  1.7  0.4
 4.6  3.4  1.4  0.3
 ⋮
 6.7  3.3  5.7  2.5
 6.7  3.0  5.2  2.3
 6.3  2.5  5.0  1.9
 6.5  3.0  5.2  2.0
 6.2  3.4  5.4  2.3
 5.9  3.0  5.1  1.8
```

```
150
```

```
begin
    # [row[1:end] for row ∈ eachrow(measurements)] # copy => Vector{Vector}... slow
    # Vector{Float64}[row for row ∈ eachrow(measurements)] # view
    measurements = [row[1:end] for row ∈ eachrow(measurements_mx)]
    labels = Symbol.(iris_ds[!, :Species])
    iris_data = [
        LabeledPoint(p, l) for (p, l) ∈ zip(measurements, labels)
    ]
    N = length(iris_data);
```

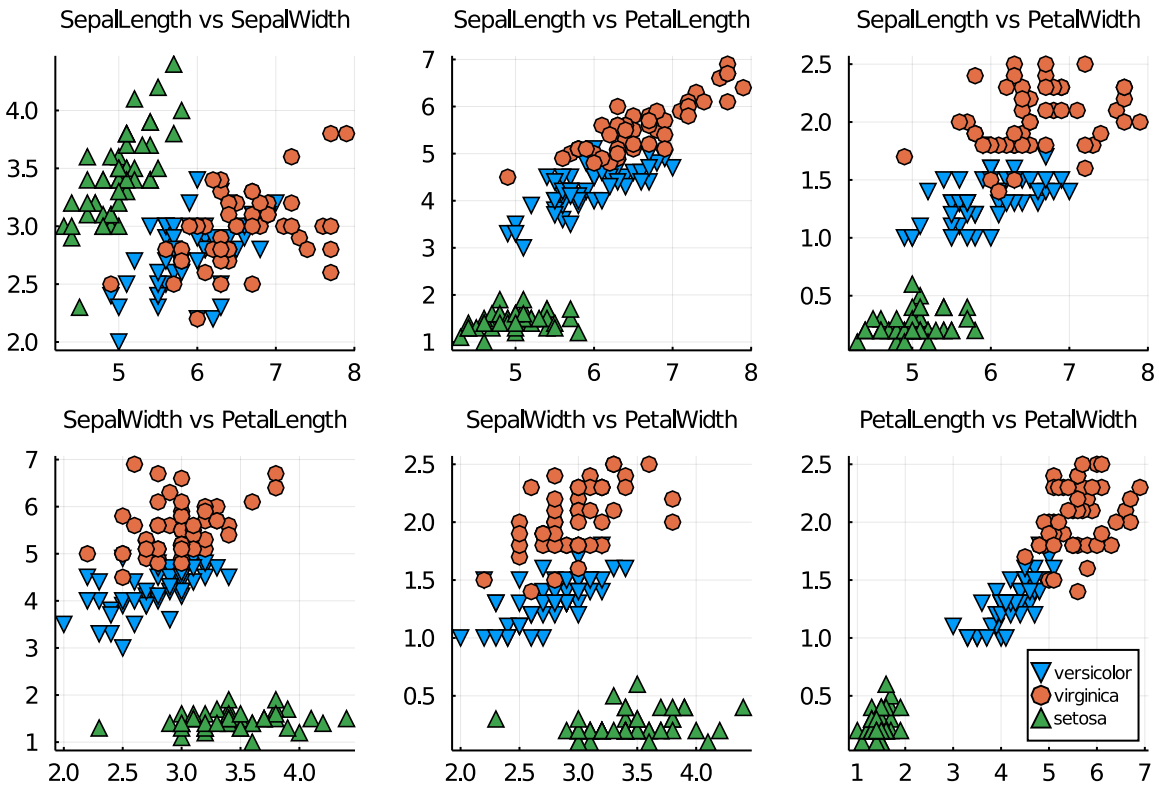
```
Test Passed
```

let us start with some visual exploration

```
Dict(:versicolor ⇒ AbstractArray{T,1} where T<:AbstractFloat[Float64[7.0,  3.2,  4.7
```

```
begin
    points_by_species = Dict{Symbol, Vector{VF}}()
    for iris ∈ iris_data
        ary = get(points_by_species, iris.label, VF[])
        push!(ary, iris.point)
        points_by_species[iris.label] = ary
    end
    points_by_species
```

```
begin
    metrics = names(iris_ds)[1:end-1]
    pairs = [(i, j) for i ∈ 1:4 for j ∈ (i+1):4]
    marks = [:v, :o, :^] # we have 3 classes, so 3 markers
    ps = []
    for row ∈ 1:2, col ∈ 1:3
        i, j = pairs[3 * (row - 1) + col]
        k = 0
        for (mark, (species, points)) ∈ zip(marks, points_by_species)
            xs = [p[i] for p ∈ points]
            ys = [p[j] for p ∈ points]
            if k % 3 == 0
                push!(ps, scatter(xs, ys, marker=mark,
                        label=string(species),
                        legend=false,
                        titlefontsize=8, title="$(metrics[i]) vs $(metrics[j])"))
            elseif row == 2 && col == 3
                ps[end] = scatter!(xs, ys, marker=mark,
                    label=string(species),
                    legendfontsize=6,
                    legend=:bottomright)
            else
                ps[end] = scatter!(xs, ys, marker=mark, label=string(species))
            end
            k += 1
        end
    end
```




```
begin
    lg = grid(2, 3, widths=[0.33, 0.33, 0.33], heights=[0.5, 0.5, 0.5])
    plot(ps..., layout=lg)
end
```

The measurement seem to really cluster by species. Looking at sepal length vs sepal width alone (top left graph) does not allow for a clean spearation between versiclor and virgina, but adding petal length and width help in the separation. Our nearest neighbors should be able to predict the species. Let us see...

```
String["SepalLength", "SepalWidth", "PetalLength", "PetalWidth", "Species"]
```

```
train_test_split (generic function with 1 method)
```

```
function train_test_split(ds::Vector{LabeledPoint};
        split=0.8, seed=42, shuffled=true)
    Random.seed!(seed)
    nr = length(ds)
    row_ixes = shuffled ? shuffle(1:nr) : collect(1:nr)
    nrp = round(Int, length(row_ixes) * split)
    (ds[row_ixes[1:nrp]], ds[row_ixes[nrp+1:nr]])
end
```

```
(Main.workspace82.LabeledPoint[LabeledPoint(Float64[5.6, 2.8, 4.9, 2.0], :virginic
```

```
Test Passed
```

```
begin
    @test length(iris_train) == round(Int, 0.7 * N)
    @test length(iris_test) == round(Int, (1 - 0.7) * N)
end
```

45

```
run_knn (generic function with 1 method)
```

```
function run_knn(iris_train, iris_test; k=5)
    num_correct = 0
    confusion_matrix = Dict{Tuple{Symbol, Symbol}, Integer}()
    for iris ∈ iris_test
        ŷ = knn(k, iris_train, iris.point)
        y = iris.label
        ŷ == y && (num_correct += 1)
        confusion_matrix[(ŷ, y)] = get(confusion_matrix, (ŷ, y), 0) + 1
    end
    perc_correct = num_correct / length(iris_test)
    (perc_correct, confusion_matrix)
end
```

```
(0.955556, Dict((:versicolor, :virginica) ⇒ 1, (:versicolor, :versicolor) ⇒ 16,
```

Not bad, we have two mis-matches: versicolor/virginica (and conversely), otherwise the rest looks good.

back to TOC

The Curse of Dimensionality

The k-nearest neighbors algorithm has problems in high dimensions due to a phenomena called the "curse of dimensionality". In a nutshell high-dimensional spaces are large (tautology) and points in such high-dimensional spaces tend to be far to one another.

One way to see this is by randomly generating pairs of points in a n-dimensional "unit cube" varying the dimensions and calculating the distances between them.

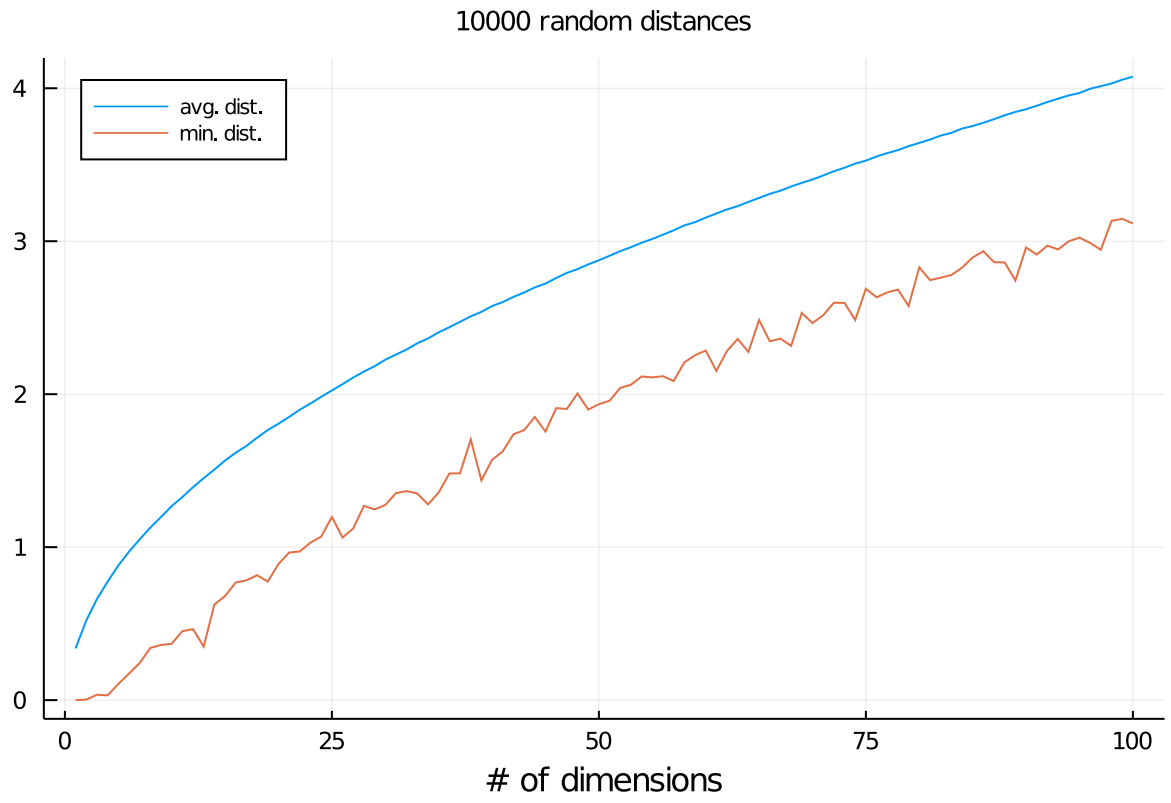
```
random_point (generic function with 1 method)
```

```
random_distances (generic function with 1 method)
```

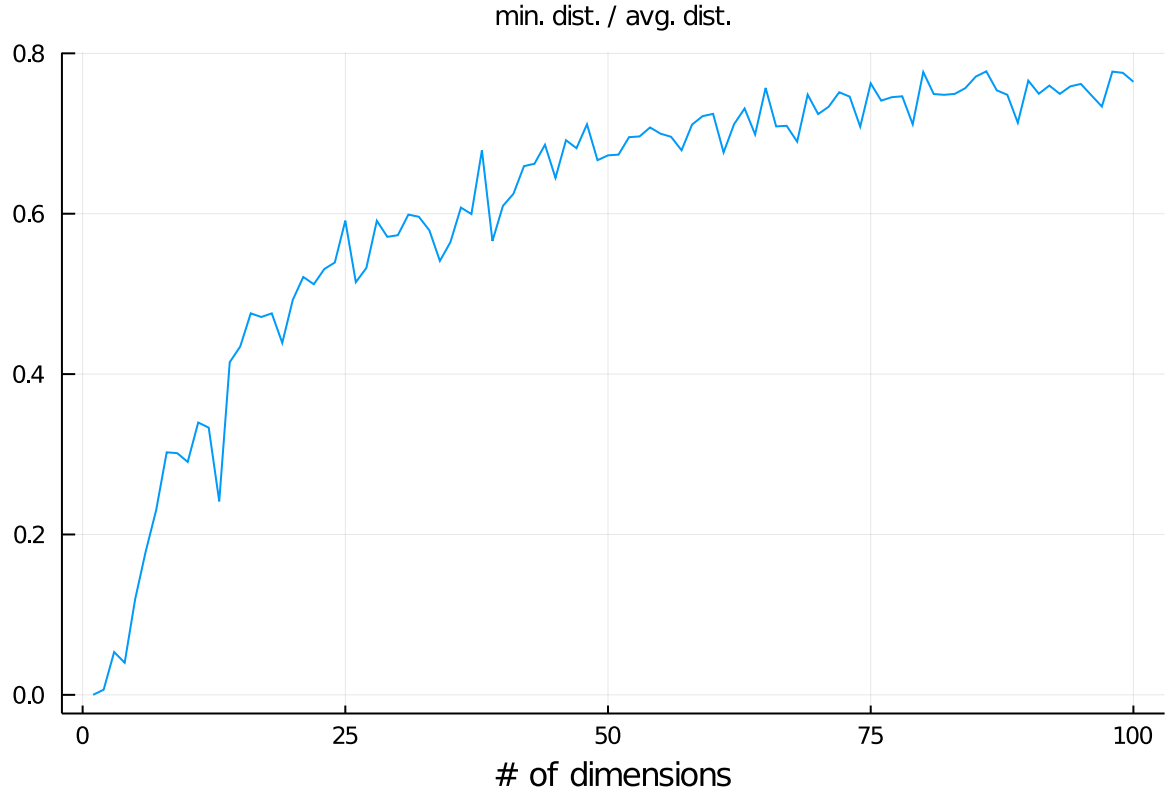
```
function random_distances(dim::Integer, num_pairs::Integer)
    [distance(random_point(dim), random_point(dim)) for _ ∈ 1:num_pairs]
end
```

```
begin
    const NS = 100
    const M = 10_000
    avg_dist, min_dist, min_avg_ratio = [], [], []
    Random.seed!(42)

    for dim ∈ 1:NS
        d = random_distances(dim, M)
        sum_d, min_d = sum(d) / M, minimum(d)
        push!(avg_dist, sum_d)
        push!(min_dist, min_d)
        push!(min_avg_ratio, min_d / sum_d);
    end
    # avg_dist, min_dist, min_avg_ratio;
end
```




```
begin
    plot(1:NS, avg_dist, legendfontsize=7, legend=:topleft,
        label="avg. dist.",
        title="$(M) random distances",
        titlefontsize=9,
        xlabel="# of dimensions"
    )
    plot!(1:NS, min_dist, label="min. dist.")
end
```



```
plot(1:NS, min_avg_ratio, legend=false, xlabel="# of dimensions",
    title="min. dist. / avg. dist.",
```

In low-dimensional datasets, the nearest points tend to be much closer than average.

The closeness of two points depends on how close thes points are in each dimension and note that every extra dimension is an opportunity for each point to be farther away from each other.

Thus when we have a lot of dimensions, the likelihood is that the closest points are not much closer than average (unless there is a lot of structure in the data)...