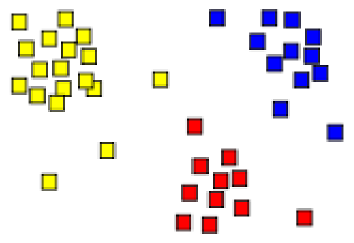
...a blog on programming, technology, philosophy

colindrake

- Home ·
- Blog ·
- Projects ·
- Resume ·
- Search ·
- Feed

Share |

Clustering in Ruby



Clustering algorithms play a very important role in the current ecosystem of web applications. If you're at all interested in automated data grouping/sorting you should at least be familiar with one or two types of these algorithms (in addition to more advanced Machine Learning topics, but this serves a good start). In this article, I'm going to go through the process of implementing a very simple Ruby program that can group a set of 2D coordinates into clusters, where each group is composed of a center point, and all of the data points closest to it.

The algorithm that we'll be using to accomplish this task is a simple one; k-means clustering will give us the behavior that we want with little fuss.

Overview

Our k-means clustering algorithm takes in, as input, a set of points in the 2-dimensional plane. As output, the points will be grouped into k clusters, where k is an integer specified by the user. Unfortunately, the algorithm can't decide how many groups there are by itself without more complication, so k must be given. Nonetheless, I'll describe the algorithm below:

1. Start by choosing k random points within the dataset's range as an initial guess for the positions of all the clusters. These points form the **centroid** point of all the clusters. All distances to other points will be measured from here.
2. For each point in the input data, assign it to the cluster that it is nearest to. After this step, each cluster will somehow be associated with a set of nearby points.
3. For each cluster, go through the set of associated datapoints and calculate the average among them. This will give a new centroid point that is directly in the center of all of the member points.
4. If the clusters didn't move from their previous locations after recentering, or if they all move less than a certain delta value, return the k clusters and their associated points. Otherwise, go back to **Step 2** after deassociating all of the associated points with their cluster. This lets the algorithm start fresh, but with more

accurate centroid points.

Implementation

To begin with, we'll need a class to store the points to be clustered by the algorithm. Essentially, we just need a Point class to hold x and y values. This is implemented below (I won't insult your intelligence trying to explain it).

```
class Point
  attr_accessor :x, :y

  # Constructor that takes in an x,y coordinate
  def initialize(x,y)
    @x = x
    @y = y
  end

  # Calculates the distance to Point p
  def dist_to(p)
    xs = (@x - p.x)**2
    ys = (@y - p.y)**2
    return Math::sqrt(xs + ys)
  end

  # Return a String representation of the object
  def to_s
    return "(#{@x}, #{@y})"
  end
end
```

This Gist brought to you by GitHub. gistfile1.rb view raw

Next, there has to be a class to hold clusters of data. As the algorithm described, clusters have groups of member points and a center point (not necessarily in the dataset) associated with them. This corresponds to two instance variables: `@points`, a list of Points, and `@center`, a single Point. Additionally, there needs to be a way for Clusters to update by averaging their member points. This is implemented in `recenter!`.

```ruby
class Cluster
  attr_accessor :center, :points

  # Constructor with a starting centerpoint
  def initialize(center)
    @center = center
    @points = []
  end

  # Recenters the centroid point and removes all of the associated
  def recenter!
    xa = ya = 0
    old_center = @center

    # Sum up all x/y coords
    @points.each do |point|
      xa += point.x
      ya += point.y
    end

    # Average out data
    xa /= points.length
    ya /= points.length

    # Reset center and return distance moved
    @center = Point.new(xa, ya)
    return old_center.dist_to(center)
  end
end
```

This Gist brought to you by GitHub. gistfile2.rb view raw

Finally, the algorithm itself needs to be implemented.

The parameters to the `kmeans` function are a dataset (list of Points), `data`, number of clusters to find, `k`, and an optional halting delta, `delta`. The algorithm will halt when all of the clusters are updated by a value less than `delta` on an iteration.

```ruby
def kmeans(data, k, delta=0.001)
```

This Gist brought to you by GitHub. gistfile1.rb view raw

Initially, the algorithm needs to choose the starting guesses for cluster centers. It does this by generating k Cluster objects, and assigning them a center from a randomly selected Point from `data`.

```ruby
  clusters = []

  # Assign intial values for all clusters
  (1..k).each do |point|
    index = (data.length * rand).to_i

    rand_point = data[index]
    c = Cluster.new(rand_point)

    clusters.push c
  end

  # ... code to follow below ...
```

This Gist brought to you by GitHub. gistfile2.rb view raw

Next is the main meat of the algorithm. The code loops indefinitely and assigns points to clusters by finding, for each point, which cluster center is the closest. This assignment will be updated, and become more accurate, each iteration of the loop while the clusters recenter.

```ruby
  # Loop
  while true
    # Assign points to clusters
    data.each do |point|
      min_dist = +INFINITY
      min_cluster = nil

      # Find the closest cluster
      clusters.each do |cluster|
        dist = point.dist_to(cluster.center)

        if dist < min_dist
          min_dist = dist
          min_cluster = cluster
        end
      end

      # Add to closest cluster
      min_cluster.points.push point
    end

    # ... code from below ...

  end  # end of while loop
```

This Gist brought to you by GitHub. gistfile3.rb view raw

Finally, in the code at the bottom of the while loop, we recalculate the centers of the clusters for the next iteration. This is done by calling `recenter!` on all of the Cluster objects. Additionally, we do some delta checking because we need to leave the loop eventually. By keeping track of the most that any Cluster was updated, we can compare it against `delta` to see if all of the Clusters were below the input delta. If the delta was hit, the algorithm terminates, returning a list of all of the Clusters found in the dataset.

```
  # Loop
  while true

    # ... code from above ...

    # Check deltas
    max_delta = -INFINITY

    clusters.each do |cluster|
      dist_moved = cluster.recenter!

      # Get largest delta
      if dist_moved > max_delta
        max_delta = dist_moved
      end
    end

    # Check exit condition
    if max_delta < delta
      return clusters
    end

    # Reset points for the next iteration
    clusters.each do |cluster|
      cluster.points = []
    end

  end  # end of while
end  # end of kmeans()
```

This Gist brought to you by GitHub. gistfile4.rb view raw

Overall, k-means clustering is a pretty simple algorithm, as you can see from above. The entire source file, along with glue/integration code, is available here to download and/or view. Next, let's see the program in action.

Example Run

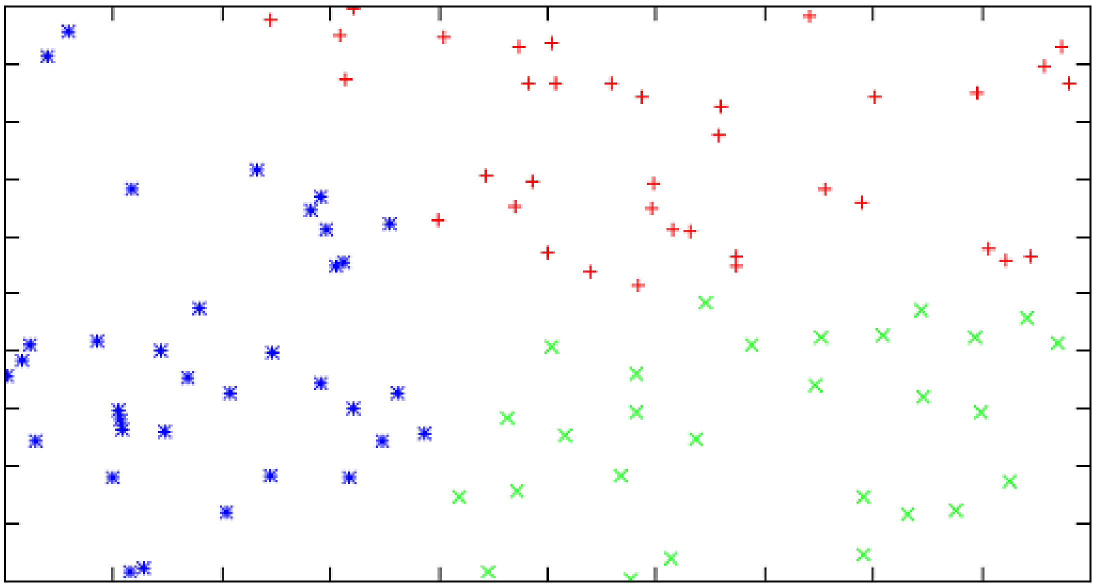
As you can see from the link above, I ended up writing some additional shell code to implement reading in data, getting `k`, and plotting the output of the algorithm. For the plotting, I used rgplot (`gem install gnuplot`) to pipe commands to a running instance of Gnuplot.

To run the full program, open up a terminal and execute:

```
$ ruby kmeans.rb CSVFILE
```

The program assumes `CSVFILE` has two comma-separated floating point numbers per line, specifying both the x and y coordinates of a single point.

To give you a feel for the output that k-means generates, I ran it on a random dataset. In the graph below, each set of points plotted with the same color indicates a Cluster object's member points. To run this yourself, you can grab my dataset here, although creating your own with more definite, pre-designed clusters may be more interesting to see.



Conclusion

While this is a very simple example, note that the x and y axis can be whatever you want them to be (latitude/longitude of households, baseball stats, etc). You could even (easily) extend the program to support 3 or more parameters (dimensions). Thus, k-means clustering can actually be a powerful tool for grouping real-world datasets, despite the apparent simplicity.

This entry was written by Colin Drake, posted on May 28, 2011, tagged with: clustering, ai, ruby, algorithm, and is available under a Creative Commons Attribution 3.0 Unported License.

1 person liked this.

Add New Comment

Login

Showing 0 comments

Sort by popular now

Subscribe by email RSS

Reactions

george_l via twitter

Clustering in Ruby - Colin Drake - colinfdrake.com http://t.co/MuKLCA6 via @AddToAny

7 hours ago

malihamariyam via twitter

RT @igrigorik: nice introduction (and a ruby implementation) of k-means clustering: http://bit.ly/iMKqnG

10 hours ago 4 more retweets from ______ ___ __________ _______

chrismarin via twitter

Clustering in Ruby http://zite.to/iX5CQb via @Ziteapp

1 day ago

LeanProdDev via twitter

RT @rgaidot: great post "Clustering in Ruby" #ruby http://t.co/EGeqQ37

1 day ago

l8stars_top_20 via twitter

Clustering in Ruby - Colin Drake - colinfdrake.com http://dlvr.it/TLtlL

1 day ago

ejstembler via twitter

Clustering in Ruby - Colin Drake - colinfdrake.com http://t.co/ALGVtnP via @AddToAny

1 day ago

Trackback URL http://disqus.com/forums/