Contextual Awareness In Sample Processing

Richard Pham

December 2021

1 Introduction

In applied computation, many problems involve data that may appear random to the solver. The randomness of this data is supported in large part by the unknown implications of the observed data and the sources that make this data appear random. In this paper, I present an online unsupervised learning algorithm called BallComp that relies on techniques from the field of fuzzy logic to process random data into "clusters" so that deductions and generalizations can be more easily formulated. It is easier to devise hypotheses on data that has already been pre-sorted when attempting to use more refined machine-learning techniques. This algorithm is similar to the k-means algorithm, but different constraints and methodologies have been injected that make BallComp more suitable for problems later mentioned in this paper.

Fuzzy logic is a field that was first introduced into mathematics by a Lotfi Zadeh, an engineering professor at U.C. Berkeley, in 1965 [San]. I used concepts in this field to base some of the estimation measures underlying BallComp. The first portion of this paper will focus on how I devised these estimation measures, and the later parts will rely on these measures to produce a description of BallComp.

2 Estimating 2-Intersection

I first focus my attention on devising an estimation formula for the area of intersection between two circles. There are actually already accurate and proven methodologies that use the area of a circular segment to calculate the area of intersection between two circles. The work in this paper is however devoted to the practice of fuzzy mathematics *in the form of rough estimation numbers.* I do not use probability distributions, fuzzy numbers, or other concepts found in fuzzy mathematics. *Suppose that I do not have knowledge of trigonometric functions and that I am incapable of deriving these functions, but I do have access to the scanning algorithm that will be detailed in the next section, so that my calculations on circle areas can only rely on that said algorithm.*

2.1 Devising The Scanning Algorithm

An algorithm is built using the Python programming language. This algorithm is run by a data structure called CircleScanner that I programmed. CircleScanner is first given an argument C, an $n \times 3$ matrix. There are n circles in consideration and the first two columns are the (x, y)-coordinates of the circle centers and the third column contains the circle radii. CircleScanner initially calculates the minimal rectangular region R that encloses all circles in C. The data structure can then be given an arbitrary argument x, in which x is a set of integers in the range $[0, n)$. Depending on the boolean function b that this data structure operates on, x could represent the requirement that an area lie in all circles of indices corresponding to x (b is intersection function) or the requirement that the same area lie only in one of the circles of index in x (b is disjunction function). For instance, if x is an empty set, then CircleScanner will estimate the area in R that none of the circles C enclose. If CircleScanner operates on disjunction and $x = \{1, 2, 3\}$, then the summation of the areas of each of the circles $1, 2, 3$ that do not intersect with one another is estimated.

The mechanism by which CircleScanner estimates area is through what I call "coexistence between linesets". A lineset is a set of 2×2 matrices, the first row is the minimum point of a line segment and the second row the maximum point, and will have identical values along one of the x or y axes. Given two linesets L_1 and L_2, an $n \times 2$ matrix is calculated. Each row in this matrix is the maximum range that a line segment in L_1 and another in L_2 share along the non-identical axis. After the calculation of this matrix, the length of each range is summed into a float value v. Multiply v by the absolute difference in the identical value of the identical axis between L_1 and L_2. CircleScanner will use this technique to scan across the axis of choice (the identical axis), and sum up the areas calculated from lineset pairs.

Due to computational constraints, all scanning values that I discuss in this paper will use a scanning increment of 0.01 (the increment along the identical axis).

2.2 Devising The Estimation Formula For Intersection Between Two Circles

I will describe a fuzzy line-of-reasoning used to concoct a preliminary formula.

Given two circles C_1 and C_2, represented in the 3-vector format described in the previous section, the below function R_i determines the "ratios of intersection" for each of C_1 and C_2, the two circles in question.

function $R_i(C_1,C_2)$
$d \leftarrow \left\|C_1[:2] - C_2[:2]\right\|$ ▷ euclidean distance b/t two circle centers
if $d \geq C_1[2] + C_2[2]$ **then return** $\varnothing$,$\varnothing$ ▷ no intersection
end if
if $d + C_2[2] \leq C_1[2]$ **then return** $0.0, 1.0$ ▷ $C_2 \subset C_1$
end if
if $d + C_1[2] \leq C_2[2]$ **then return** $1.0, 0.0$ ▷ $C_1 \subset C_2$
end if
$a \leftarrow \angle(C_1[:2], C_2[:2])$ ▷ angle between two centers
$a_2 \leftarrow (a + 180) \mod 360$ ▷ counter-angle
$p \leftarrow$ ▷ the point from C_1[:2] of length $C_1[2]$ and angle a
$p_2 \leftarrow$ ▷ the point from C_2[:2] of length $C_2[2]$ and angle a_2
$d_1 \leftarrow \frac{\|p-p_2\|}{C_1[2]\times 2}$
$d_2 \leftarrow \frac{\|p-p_2\|}{C_2[2]\times 2}$
return d_1, d_2
end function

R_i simply calculates the maximal length of the intersection area on the line between the two circle centers, and divides that length by the diameters of C_1 and C_2 respectively. If one of the circles is completely contained in the other, the pair $(0, 1)$ or $(1, 0)$ is output. Function R_i is then used to estimate the areas of the circle segments of C_1 and C_2 by their intersection ratios.

The below function is a pre-cursor function.

function $E(C,r)$ ▷ C is the circle representation, r the ratio of intersection
$b_a \leftarrow (C[2] \times 2)^2$ ▷ area of the minimal square enclosing C
$a \leftarrow b_a - (\pi \times C[2]^2)$ ▷ area of square not in C
if $r \leq 0.5$ **then return** $(b_a \times r) - (a \times r)$
end if
return $\pi \times C[2]^2 \times 0.5 + E(C, r - 0.5)$
end function

Remark. This function will require multiplication by a scalar.

My initial estimation measure for intersection between two circles is A_i.

function $A_i(C_1,C_2)$
  $r_1, r_2 \leftarrow R_i(C_1, C_2)$
  if $r_1 = \varnothing$ **then return** 0.0 ▷ case: no intersection
  end if
  if $r_1 = 1.0$ **then return** $\pi \times C_1[2]^2$
  end if
  if $r_2 = 1.0$ **then return** $\pi \times C_2[2]^2$
  end if
  return $\min([E(C_1, r_1), E(C_2, r_2)])$
end function

The following table of results, collected through a Python algorithm, demonstrates the use of A_i on two intersecting circles. A reference circle with a radius of 3.0 is used. The ratios in the columns "area ratio" and "intersection ratio" correspond to this circle.

area ratio	intersection ratio	area scanned	area est.	curvature est.
1/6	1/6	0.75040	0.78540	0.95544
1/6	1/4	0.72110	0.78540	0.91813
1/6	1/3	0.73160	0.78540	0.93150
1/6	1/2	0.76360	0.78540	0.97225
1/6	2/3	0.76360	0.78540	0.97225
1/6	5/6	0.76360	0.78540	0.97225
1/6	1	0.76370	0.78540	0.97237
1/4	1/6	1.13450	1.17810	0.96300
1/4	1/4	1.63690	1.76715	0.92630
1/4	1/3	1.58900	1.76715	0.89919
1/4	1/2	1.68520	1.76715	0.95363
1/4	2/3	1.68520	1.76715	0.95363
1/4	5/6	1.68520	1.76715	0.95363
1/4	1	1.68520	1.76715	0.95363
1/3	1/6	1.39580	1.57080	0.88860
1/3	1/4	2.25250	2.35619	0.95599
1/3	1/3	2.85940	3.14159	0.91018
1/3	1/2	2.96200	3.14159	0.94283
1/3	2/3	2.98020	3.14159	0.94863
1/3	5/6	2.98020	3.14159	0.94863
1/3	1	2.99680	3.14159	0.95391
1/2	1/6	1.71630	2.35619	0.72842
1/2	1/4	2.98060	3.53429	0.84334
1/2	1/3	4.26800	4.71239	0.90570
1/2	1/2	6.52200	7.06858	0.92267
1/2	2/3	6.59580	7.06858	0.93311
1/2	5/6	6.64800	7.06858	0.94050
1/2	1	6.56300	7.06858	0.92847
2/3	1/6	1.91720	3.14159	0.61026
2/3	1/4	3.42220	4.71239	0.72621
2/3	1/3	5.10150	6.28319	0.81193
2/3	1/2	8.68590	9.42478	0.92160
2/3	2/3	11.74180	12.56637	0.93438
2/3	5/6	11.78360	12.56637	0.93771
2/3	1	11.61580	12.56637	0.92436
5/6	1/6	2.06340	3.92700	0.52544
5/6	1/4	3.70340	5.89049	0.62871
5/6	1/3	5.60620	7.85398	0.71380
5/6	1/2	9.96480	11.78097	0.84584
5/6	2/3	14.47620	15.70796	0.92158
5/6	5/6	18.26960	19.63495	0.93046
5/6	1	18.00020	19.63495	0.91674
1	1/6	2.20500	4.71239	0.46792
1	1/4	3.97900	7.06858	0.56291
1	1/3	6.03040	9.42478	0.63985
1	1/2	10.78960	14.13717	0.76321
1	2/3	15.86900	18.84956	0.84188
1	5/6	20.87190	23.56194	0.88583
1	1	25.79290	28.27433	0.91224

Table 1: Values (rounded to 5 decimal places) for 2-intersection with reference circle radius of 3.0

An analysis of table 1 indicate two main issues. The first issue is that the area estimated and area scanned values are relatively different from one another. This fact can be seen in the column "curvature est.". For each of the values v in that column, $1 - v$ corresponds to the added arc area (the curvature) of the intersection. So function A_i will need to be modified. The second issue is observed in the green rows. The area estimated in these rows is actually more accurate that the values of area scanned (using the CircleScanner). This issue demonstrates the shortcomings of the scanning increment set to 0.01. *A relevant point to be made is that any estimate derived from this scanner set at the increment 0.01 is due to be "fuzzy".* Table 2 details this imprecision.

area ratio	area scanned	actual area	scanned/actual
1/6	0.7637	0.7854	0.97237
1/4	1.7004	1.76715	0.96223
1/3	2.9968	3.14159	0.95391
1/2	6.563	7.065	0.92895
2/3	11.6158	12.56637	0.92436
5/6	18.0002	19.63495	0.91674
1	25.7929	28.27433	0.91224

Table 2: Ratios in right-most column correspond to scanning inaccuracy.

Noting these issues, I attempt to compensate for the shortcomings by finding the scalar value (I mentioned in my last remark) to "fix" function E, so that the revised estimation measure will work fairly well for the purposes in the later parts of this paper. I want to stay on the theme of fuzz, and instead of using linear algebra, I decide to devise a mapping function. Fuzzy mathematics also has a subfield called "linear fuzzy systems", but the work in this paper will not go towards that direction. This mapping function will depend on the minimum and maximum curvature estimations for *each intersection ratio.* But recall the inaccuracy of the green rows, so only the minimum curvature estimations are of importance, and set the maximum to a rough 1.0. *The minimum curvature estimations are with area ratios of 1.0, so there is an inverse correlation between these two columns.*

intersection ratio	min. curvature est.
1	0.46792
5/6	0.52544
2/3	0.61026
1/2	0.72842
1/3	0.88860
1/4	0.96300
1/6	1.0

Table 3: Values in second column can be used to correct for over-estimation.

The values in the second column in Table 3 can be used to "fix" values obtained from function A_i. The focus is to find an effective mapping function S_f that determines scalar "fixes" using this table. This function will depend on the functions m_r, r_x, and c_r.

function $m_r(C_1, C_2)$ ▷ determines minimal area ratio
  return $min(\frac{C_1[2]}{C_2[2]}, \frac{C_2[2]}{C_1[2]})$
end function

function $r_x(C_1, C_2, r_1, r_2)$ ▷ determines reference intersection ratio
  ▷ r_1 is intersection ratio for C_1, r_2 is intersection ratio for C_2
  if $C_1[2] > C_2[2]$ **then return** r_1
  end if
  return r_2
end function

function $c_r(i_{ref})$
  ▷ maps reference ratio i_{ref} to closest estimation increment
  $t \leftarrow (\frac{1}{6}, \frac{1}{4}, \frac{1}{2}, \frac{1}{3}, \frac{2}{3}, \frac{5}{6}, 1)$ ▷ possible values for range point
  $i \leftarrow \arg\min_{x \in t} |x - i_{ref}|$
  return $t[i]$
end function

The function c_r outputs the enclosing ratio range for a reference ratio, calculated in functions r_x and m_r. Next, define a function T_{sf} that takes as input a value in the set $\{\frac{1}{6}, \frac{1}{4}, \frac{1}{2}, \frac{1}{3}, \frac{2}{3}, \frac{5}{6}, 1\}$ and outputs its corresponding minimal curvature value found in Table 3. For example, $T_{sf}(\frac{5}{6}) = 0.52544$. The necessary

components to succinctly define the wanted mapping function S_f have been laid out. Below is its pseudo-code description:

```
    function S_f(C_1, C_2)
2:                                  ▷ determines the scalar value applied to A_i
        a_r ← m_r(C_1, C_2)                                  ▷ the reference area ratio
4:      r_1, r_2 ← R_i(C_1, C_2)
        i_r ← r_x(C_1, C_2, r_1, r_2)                ▷ the reference intersection ratio
6:      return T_sf(a_r) + (1 − T_sf(a_r)) × i_r
    end function
```

The revised version of A_i is A_x, and is virtually identical except for the ending line.

```
    function A_x(C_1,C_2)
2:      r_1, r_2 ← R_i(C_1, C_2)
        if r_1 = ∅  then return 0.0                          ▷ case: no intersection
4:      end if
        if r_1 = 1.0 then return π × C_1[2]^2
6:      end if
        if r_2 = 1.0 then return π × C_2[2]^2
8:      end if
        return min([E(C_1, r_1), E(C_2, r_2)]) × S_f(C_1, C_2)
10: end function
```

Remark. The formulation is a little inefficient. Some values are calculated twice by calling S_f, but it works!

3 Scaling Up The Fuzz For N-Intersection

A formal definition for n-intersection between circles is necessary.

Definition 3.1. N-intersection between circles. The area shared by a set of n circles.

3.1 Recognizing Intersections From Intersections

The below lemma is important for proceeding to approximating n-intersections.

Lemma 3.1. *Given a set S of m-intersections (each element is a set containing m objects that intersect), there exists an n-intersection, $n > m$, if the following hold true:*
1) There are at least n unique objects.
2) The number of m-intersections is

$$\sum_{x=1}^{n-1} \binom{(n-x)}{(m-1)}.$$

3) Every unique object occurs in $\binom{(n-1)}{(m-1)}$ elements of S.

Remark. I call any set that satisfies the above lemma for an n-intersection an (n, m)-direct implication, because the elements directly imply the existence of an n-intersection.

Remark. This lemma can be used to deduce n-intersections from an initial set of 2-intersections, and the work in this section greatly relies upon it.

3.2 A Lazy Estimation Approach

At this point, for intersection areas that lie in more than two objects, I will not go about devising estimation measures that depend on measures collected from CircleScanner. Instead, I use the below measure:

---

function $A_y(A_m)$ ▷ A_m contains $(m+1)$ m-intersecting area values
▷ determines the $(m+1)$-intersecting area value
return $min(A_m) \times 0.8$
end function

Function A_y is restricted to estimating intersection areas that are one intersection $m+1$ above the areas of its argument A_m. An accompanying function to A_y is what I term "$(m-1)$-decomposing implication" and is denoted $D_i(S)$.

Definition 3.2. Decomposing Implication. For a set S of objects that intersect with one another, an alternative representation of S that would allow for direct implication back to S is S_d, a vector representing the combinations $\binom{|S|}{|S|-1}$ from S.

Remark. A $(m-1)$-decomposing implication is essentially the combinatorial $\binom{m}{m-1}$. The term "decomposing implication" is used in the context of object intersections.

4 Estimating Disjunction

Definition 4.1. Disjunction between n-intersecting circles. The cumulative area of the circles that lies in only one ball for a set of n circles that intersect.

For the purposes of the primary focus of this paper, the BallComp algorithm, estimation of disjunction is actually more important. In this section, a description of an estimation approach for area of disjunction that uses the concepts and data in the previous parts of this paper will be provided. Then an explanation of how this estimation approach is utilized by BallComp is briefly explained.

4.1 Estimating Disjunction Area

Suppose that there exists a set of n circles that intersect. This set is represented in vector form and each element is a 3-vector representation of a circle:

$$V_c =< c_1, c_2, ..., c_n > .$$

The below function estimates values using estimation measures presented in the previous parts of this paper.

$$F_{est}(C_s) = \begin{cases} \pi \times C_s[2]^2 & C_s \texttt{ is a circle} \\ A_x(C_s[0], C_s[1]) & C_s \texttt{ contains 2 circles} \\ A_y(C_s) & C_s \texttt{ is a vector of intersection area estimations} \end{cases}$$

Remark. The disjunction area estimation starts with the first case of F_{est}, then the second case, and will select the third case until completion. If the third case is selected, then the output value is an estimation of the area for a $|C_s|$-intersection.

Denote the subset of $\binom{n}{r}$ circles that intersect, $1 \leq r \leq n$, from V_c as V_{c_r} (vector form). For a vector of m-intersecting circles C_m, $m \leq n$, denote the estimated value of their intersection area as:

$$I_{est}(C_m).$$

When the estimation of disjunction is in progress, this value $I_{est}(*)$ will be kept in memory and can be readily accessed, and is different from F_{est}. Another pre-cursor function will be required.

function S_{c_r}
  if $r \leq 2$ **then**
    return V_{c_r}
  else
    $v \leftarrow \varnothing$
    for $x \in V_{c_r}$ **do**
      $v \leftarrow v \cup I_{est}(x)$
    end for
    return v
  end if
end function

The formula below is what will be referred to as the "additive" area, which the next formula will rely upon to estimate the total area of "disjunction" of these n circles.

$$F_{add}(r) = -1^{(r-1)} \times \sum_{i=0}^{S_{c_r}} (F_{est}(S_{c_r}[i]) \times q(r).$$

$$q(r) = \begin{cases} r & \texttt{if lower estimate} \\ q'(r) & \texttt{if upper estimate} \end{cases}$$

$$q'(r) = \begin{cases} r & \texttt{if } r \leq 2 \\ r-1 & \texttt{otherwise.} \end{cases}$$

The additive area is a summation of the area to add or subtract to the running estimation, the summation of the additive area before that time $r-1$.

Then the estimated area of disjunction for a set of n-intersecting circles is:

$$A_{dest}^{(n)}(C_n) = \sum_{j=1}^{n} (F_{add})(j)$$

4.2 A Short Proof On $A_{dest}^{(n)}$

Remark. This proof will not gauge the accuracy of the required estimation measures A_x and A_y.

We are given a circle set C_s.

If $|C_s| = 1$, then F_{est} will output the precise area value of $C_s[0]$ and we are done. If $|C_s| = 2$, then F_{add} will minus the area of intersection from each of the two circles' area. If $|C_s| \geq 3$, then during the progress of the estimation, F_{add} will alternately add (if odd) or subtract (if even) "additive" areas multiplied by their duplicative counts made during the previous calls to F_{add}. For an

upper-bound on the disjunction, F_{add} will add or subtract an extra copy of the additive area for each $i \in \texttt{range}(n)$.

Remark. Comparative test results between $A^{(n)}_{dest}$ and area scanned by CircleScanner are not provided in this paper.

4.3 A Review Of The Disjunction Estimation

I designed this disjunction estimation to be used in the BallComp algorithm. It is not designed to be maximally efficient as calculus solutions. Whereas calculus solutions involve integration over bounds and then subtraction of the output values of the two extremum to provide precise answers, *the estimator $A^{(n)}_{dest}$ involves calculations on the scale of $O(2^n)$, 2^n the number of possible sets chosen from the n circles.* So $A^{(n)}_{dest}$ does not run in polynomial time.

The estimation measures have focused on circles, $k = 2$ dimensions. To "extend" this disjunction measure to $k > 2$ dimensions, replace the formula $\pi \times r^2$ with more suitable volumetric measures. *This formula will be used on circular shapes greater than two dimensions in the next section and beyond.*

5 Description Of BallComp

BallComp is a data structure that takes the following arguments:
1) the number of balls,$n \in \mathbb{N}$,
2) maximum radius $r \in \mathbb{R}$ per ball,
3) the ball dimension, $k \in \mathbb{N}_{\geq 2}$.

BallComp's argument [1] denotes the number of k-dimensional balls, k being argument [3], that it can hold. Each of these balls have the maximum volume of argument [2]. These arguments denote upper-thresholds, values that cannot be exceeded. Additionally, these upper-thresholds fall into one of two classifications.

Definition 5.1. Soft Threshold. A threshold that can be violated but registers a cost value (usually negative) calculated by a cost function that takes the *degree of violation* as one of its primary arguments.

Definition 5.2. Hard Threshold. A threshold that cannot be violated by its respective infrastructure under any known conditions.

Argument [1] is a hard threshold, and arguments [2] and [3] are soft thresholds.

This data structure is equivalently denoted $B^{(C)}$. Argument [1] is referenced as $B^{(C)}_n$, argument [2] is $B^{(C)}_r$, and argument [3] is $B^{(C)}_k$.

5.1 The Ball Data Structure

The n-ball is a familiar data structure in geometry, such as in the fields of topology and calculus.

Definition 5.3. Ball. A data structure B in k dimensions, $k \geq 1$, with a radius of $B_r \in \mathbb{R}$ and center point $B_c \in \mathbb{R}^k$, that encloses the set of points $\{P\}$ that satisfy the following:

$$\|p - B_c\| \leq B_r.$$

Remark. This paper will refer to this data structure simply as "ball".

In the algorithm, BallComp, this data structure will not contain all possible points within the bounds specified by B_r. Instead, the Ball data structure will contain only the samples that BallComp receives. A term called the "radial reference" is crucial to accomodating this condition.

Definition 5.4. Radial Reference. The sample $\arg\max_{p \in B} \|p - B_c\|$ in a ball B, denoted B_{p_r}.

The radius of the Ball data structure is a non-static variable and is subject to change according to its present radial reference. Suppose a sample p is to be added to ball B. Then the delta of B_r is the following:

$$\Delta(B_r) = \begin{cases} 0 & \left\|B_{p_r} - B_c\right\| \geq \left\|B_{p_r} - p\right\| \\ \left\|B_{p_r} - p\right\| - \left\|B_{p_r} - B_c\right\| & otherwise. \end{cases}$$

If the addition of p to B yields in a $\Delta(B_r) > 0$, then its updated radial reference is $B_{p_r} = p$.

A neighbor $B^{(2)}$ of $B^{(1)}$ will satisfy the inequalities

$$\left\|B_c^{(2)} - B_c^{(1)}\right\| + B_r^{(2)} \leq B_r^{(1)},$$
$$\left\|B_c^{(2)} - B_c^{(1)}\right\| + B_r^{(1)} \leq B_r^{(2)}.$$

The operator + is a commutative pairwise operator between two balls $B^{(1)}$ and $B^{(2)}$ such that the resulting $B^{(3)}$ has the center point and radius calculated by the below function.

function $R^{(+)}(B^{(1)}, B^{(2)})$
  $r_1, r_2 \leftarrow R_i(B^{(1)}, B^{(2)})$
  ▷ ball 1 in ball 2
  if $r_1 = 1.0$ **then**
    return $B_c^{(2)}, B_r^{(2)}$
  end if
  ▷ ball 2 in ball 1
  if $r_2 = 1.0$ **then**
    return $B_c^{(1)}, B_r^{(1)}$
  end if
  $d_e \leftarrow \left\| B_c^{(1)} - B_c^{(2)} \right\|$
  ▷ farthest boundary point of ball 2 on line ($B_c^{(1)}$,$B_c^{(2)}$)
  $x_2 \leftarrow B_r^{(2)}/d_e$
  $e_2 \leftarrow B_c^{(2)} + x_2 \times (B_c^{(2)} - B_c^{(1)})$
  ▷ farthest boundary point of ball 1 on line ($B_c^{(1)}$,$B_c^{(2)}$)
  $x_1 \leftarrow B_r^{(1)}/d_e$
  $e_1 \leftarrow B_c^{(1)} + x_1 \times (B_c^{(1)} - B_c^{(2)})$
  ▷ calculate midpoint
  $m \leftarrow \frac{e_1+e_2}{2.0}$
  ▷ calculate radius
  $r \leftarrow \frac{\|e_1-e_2\|}{2.0}$
  return m, r
end function

5.2 Sample Input Into BallComp

Consider BallComp to be a structure that acts as a passive receiver of sample data. It is denoted as $B^{(c)}$ in formulas. Each sample is a k-dimensional vector. Upon receiving a sample p, BallComp takes one of two possible *pathways*,$P^{(1)}$ or $P^{(2)}$, to add p to its memory. These pathways use two functions DEC_1 and DEC_2, respectively. The arguments to DEC_1 are based on the variables below.

- The *unique volume*, v_u, is the volume of disjunction between a set of balls.
- The *total volume*,v_t, is the summation of a set of balls' volumes.
- The *maximum volume* per ball, calculated by $B_r^{(C)}$, is $B_V^{(C)}$.
- The *potential volume*,v_p, of a set of balls B_s is $B_V^{(C)} \times |B_s|$.

The arguments to DEC_2 are based on the variables below.

- The *present number of balls*, $|B^{(C)}|$.
- The *maximum number of balls*,$B_n^{(C)}$.

The first pathway P_1 calculates if p is to be added to a present ball. The second pathway P_2 calculates if a new ball is to be declared for p.

5.3 Pathway $P^{(1)}$

The ball $B^{(x)}$ in $B^{(C)}$ such that

$$\arg \min_{B \in B^{(C)}} \|p - B_c\|$$

is first selected. If $|B^{(C)}| = 0$, then ignore the rest of this section.

Add p to $B^{(x)}$ according to definition 5.4. The values v_u,v_t, and v_p will be determined according to the below methods.

---

function $V_U(B^{(x)}, n)$
▷ estimates the disjunction volume of a ball
▷ $B^{(x)}$ is the ball p is added to
▷ n is bool determining if its neighbors are included in the calculation.
  if n **then**
    $s \leftarrow \texttt{neighbors}(B^{(x)}) \cup B^{(x)}$
  else
    $s \leftarrow B^{(x)}$
  end if
  return $A^{(n)}_{dest}(s)$
end function

---

function $V_T(B^{(x)}, n)$
  ▷ $B^{(x)}$ is the ball p is added to ▷ n is bool determining if its neighbors are included in the calculation.
  if n **then**
    $s \leftarrow \texttt{neighbors}(B^{(x)}) \cup B^{(x)}$
  else
    $s \leftarrow B^{(x)}$
  end if
  ▷ sum each volume in s
  return $\sum_{b \in s}^{s} \texttt{VOLUME}(b)$
end function

---

function $V_P(B^{(x)}, n)$
▷ $B^{(x)}$ is the ball p is added to ▷ n is bool determining if its neighbors are included in the calculation.
  if n **then**
    $s \leftarrow \texttt{neighbors}(B^{(x)}) \cup B^{(x)}$
  else
    $s \leftarrow B^{(x)}$
  end if
                                ▷ calculate the maximum volume of one n-ball
  $v = VOLUME(B_r^{(C)})$
  return $v \times |s|$
end function

A cost function should be designed so that there are marked differences for balls with high intersection volume with its neighbors. There are many cost functions that would satisfy this requirement. A simple cost function could be the following:

function DEC_1(u,t,p)
  $x_1 = 1 - \frac{u}{t}$
  $x_2 = \frac{t}{p}$
  return $\frac{x_1 + x_2}{2}$
end function

Pathway $P^{(1)}$ of DEC_1 determines if a sample p is to be added to the ball $B^{(x)} \in BC$. Recall that addition of p to $B^{(x)}$ may result in $\Delta(B^{(x)}) > 0$.

$P^{(1)}$ will conduct two simulations of adding p to $B^{(x)}$. Both simulations will call the DEC_1 function, but under different conditions. The first simulation simply involves adding p to $B^{(x)}$ and then calling DEC_1, outputting a score s_1. Simulation 1 uses the below measures as input to DEC_1.

function $M_1(B^{(x)})$
  $v_u \leftarrow V_U(B^{(x)}, \texttt{TRUE})$
  $v_t \leftarrow V_T(B^{(x)}, \texttt{TRUE})$
  $v_p \leftarrow V_P(B^{(x)}, \texttt{TRUE})$
  return v_u, v_t, v_p
end function

Simulation 2 will add p to $B^{(x)}$ and then perform a merge operation, described below.

function $M_g(B^{(x)})$ ▷

$s \leftarrow \texttt{neighbors}(B^{(x)}) \cup B^{(x)}$

▷ calculate average center distances of balls

$d \leftarrow \sum_{b \in s}^{s} b_c / |s|$

▷ calculate the ordering of the balls based on center

▷ distance from average

$o \leftarrow \texttt{argsort}_{b \in s}^{s} \|b_c - d\|$

$B \leftarrow \texttt{pop}(o, 0)$

while $|o| > 0$ **do**

$B \leftarrow B + \texttt{pop}(o, 0)$

end while

return B

end function

function $M_2(B^{(x)})$

$B^{(y)} \leftarrow M_g(B^{(x)})$

$v_u \leftarrow V_U(B^{(y)}, \texttt{FALSE})$

$v_t \leftarrow V_T(B^{(y)}, \texttt{FALSE})$

$v_p \leftarrow V_P(B^{(y)}, \texttt{FALSE})$

return v_u, v_t, v_p

end function

Simulation 2 uses the measures from function M_2. After these two simulations are completed so scores s_1 and s_2 are retrieved, revert changes made to all involved balls in these two simulations. $B^{(x)}$ will no longer have the point p, its radial reference will be identical to that before the simulations took place, and $B^{(x)}$ is not merged with its neighbors. Then pathway P_1 will output the tuple (s_x, v_x), s_x is the simulation with the lower score and v_x is its score.

The aim of P_1 is to determine if adding p to a present ball in $B^{(C)}$ is feasible with respect to the unique volume, i.e. the disjunction volume.

5.4 Pathway $P^{(2)}$

$P^{(2)}$ calculates a score for a new ball $B^{(z)}$ centered at p with a minimal radius such as $B_r^{(C)} \times 10^{-3}$. The volume of $B^{(z)}$ will be relatively small, so $P^{(2)}$) will output a score from a DEC_2 that is not concerned with unique volume and bases its output value on the number of balls held by $B^{(C)}$ and $B_n^{(C)}$. An example of DEC_2 is below.

function $DEC_2(b_1, b_2)$
▷ b_1 is the number of balls held by $B^{(C)}$
▷ b_2 is $B_n^{(C)}$
return $\frac{b_1+1}{b_2}$
end function

5.5 BallComp's Decision-Making Pipeline

BallComp will take measures from the two pathways $P^{(1)}$ and $P^{(2)}$. The pathway with the lower score will be executed by BallComp. But there may be an issue after this decision. The issue is that one of the soft thresholds $B_r^{(C)}$ or $B_n^{(C)}$ may have been violated. The next section goes into detail on an auto-correction procedure.

5.5.1 BallComp's Auto-Correction

If $P^{(1)}$ is taken, then $B_r^{(C)}$ might have been violated. And if $P^{(2)}$ is taken, then $B_n^{(C)}$ might have been violated. If either of these thresholds are violated, then BallComp will require a methodology to handle the violation.

Denote this methodology as V_H (violation handler) that checks the decision conducted by BallComp for each new point it processes. V_H will check specific attributes of BallComp to determine whether BallComp should change its soft thresholds or undergo a delta termination.

Definition 5.5. Delta Termination. An alert declared by an instance of BallComp that will not permit it to add new balls to its solution and any future point processed by it will not result in any $\Delta(B_r^{(i)})$, $B^{(i)} \in B^{(C)}$. BallComp will obey this alert only after it *accepts* it.

V_H will have the capability to "split" balls if they violate $B_r^{(C)}$.

Definition 5.6. Ball-split. A set of balls $\{B'\}$ such that $\forall p \in B$, $\exists$ a ball $B_x \in \{B'\}$ that contains point p. The ballset $\{B'\}$ splits B.

Here is one possible design for V_H. BallComp will initially take one of these steps for violation of $B_r^{(C)}$:

1A Recommend a maximum radius $r \geq B_r^{(x)} \geq B_r^{(C)}$, B_x is the ball that the new point p is added to. If V_H will not permit such an r, then V_H declares a delta termination for this specific violation.

1B Split ball B_x into the set of balls $\{B_s\}$ according to a ball-splitting algorithm. If violation of $B_n^{(C)}$ occurs, V_H declares a delta termination. Otherwise, replace B_x from $B^{(C)}$ with $\{B_s\}$.

V_H will initially take one of these steps for violation of $B_n^{(C)}$:

2A Recommend a maximum number of balls $n \geq B_n^{(C)}$. If V_H will not permit such an n, V_H declares a delta termination.

2B Delete $B^{(x)}$, the new ball declared for p. Determine a ball $B^{(x_1)}$ by first calculating

$$\texttt{argmin}_{b \in B^{(C)}}(b+p)_r,$$

the ball with the least radius after adding point p. If $B_r^{(x_1)} \leq B_r^{(C)}$,then $B^{(x_1)}$ adds p. Otherwise, V_H declares a delta termination.

Now, a pipeline based on the steps described above, taken for the case of violation will be provided.

```
    function V_H(B^(C))
 2:     if VIOLATION(B^(C), B_r^(C)) then
            take step 1A
 4:         if delta termination then
                take step 1B
 6:             if delta termination then
                    take step 2A
 8:                 if delta termination then
                        accept delta termination
10:                 end if
                end if
12:         end if
        end if
14:     if VIOLATION(B^(C), B_n^(C)) then
            take step 2A
16:         if delta termination then
                take step 2B
18:             if delta termination then
                    take step 1A
20:                 if delta termination then
                        accept delta termination
22:                 end if
                end if
24:         end if
        end if
26: end function
```

5.6 Programmatically Improving The Solution

There a variety of sources on statistics, linear programming, and other topics in the field of machine-learning that can aid this task. I was not able to devise a linear program suitable for the variable requirements of BallComp.

The primary issue that BallComp aims to solve is to construct a context in the form of a set of balls containing all the points it has processed and will process. The algorithm by itself will not be able to deduce the bounds that these points will lie in.

Definition 5.7. Bound. An $n \times 2$ matrix, n denotes the dimension of the space, that represents the bounds for an n-rectangular space. Each row of this matrix will be a pair f_{i_1}, f_{i_2} in which $f_{i_1} \leq f_{i_2}$.

However, methodologies from statistical learning could certainly be included into BallComp's capabilities. For example, BallComp could be given a bounds calculated by a hypothesis and wants to determine the appropriate number of balls given a radius. Then the two lemmas below is applicable.

Lemma 5.1. *Suppose there is a bounds F and $D = \texttt{argsort}_{i=1}^{i \leq n}(F[i,1] - F[i,0])$ is the sorted lengths of each dimension in increasing order. Then the number of balls of maximum radius $r \geq D[-2]$ required to cover an n-dimensional bounds F is in linear space.*

Lemma 5.2. *The number of balls of maximum radius r required to cover an n-dimensional bounds F is*

$$\mathcal{O}((\max_{i=1}^{i \leq n}(F[i,1] - (F[i,0]))/(r \times 2))^n).$$

Or perhaps BallComp wants to retrain on a sequence of data that it has just finished processing. Then the definition below is useful.

Definition 5.8. D-Occupancy. For a bounds F representing an n-dimensional rectangular space and set $\{\mathrm{S}\}$ of points that lie in F, the minumum number of disjoint subsets $\{s_1, ..., s_r\} = Q$ that satisfy:
1)

$$\cup_{s \subset Q} s = S,$$

2) $\forall s \subset Q$, every pair of points $p_1, p_2 \in s$ satisfies $\|p_1 - p_2\| \leq D \times 2$.

So the minumum number of balls of maximum radius d required to fit all points in any n-rectangular bounds is equal to its d-occupancy.

Lemma 5.3. *Finding the minimal disjoint subsets Q for a n-dimensional bounds F is in linear time and linear space if maximum radius per ball r equals*

$$\max_{i=1}^{i \leq n}(F[i,1] - F[i,0]).$$

Below is a procedure that proves the above lemma:

```
    function Q_min(S, F, r)
2:                                          ▷ S is a subset of points in F
                                            ▷ F is a bound
4:                                          ▷ sort S
        S ← argsort_{s∈S} ‖s − F[:, 0]‖
6:      Q ← {}
        p ← pop(S, 0)
8:      q ← {p}
        while |S| > 0 do
10:         p_2 ← pop(S, 0)
            if ‖p − p_2‖ ≤ r then
12:             q ← q ∪ p_2
            else
14:             Q ← Q ∪ q
                q ← {p_2}
16:             p ← p_2
            end if
18:     end while
        Q ← Q ∪ q
20:     return Q
    end function
```

And here is an approximation procedure to calculate a minimal set for d-occupancy on any case.

```
function Q_approx(S, F, r)
                                          ▷ S is a subset of points in F
                                                          ▷ F is a bound

    function L(p_x)
        for q ∈ Q do
            if ||p_x − q[0]|| ≤ r then
                return q
            end if
            return ∅
        end for
    end function
                                                              ▷ sort S
    S ← argsort_{s∈S} ||s − F[:, 0]||
    p ← pop(S, 0)
    q ←< p >
    Q ← {q}
    while |S| > 0 do
        p_2 ← pop(S, 0)
        l ← L(p_2)
        if l ≠ ∅ then
            l.append(p_2)
        else
            q ←< p_2 >
            Q ← Q ∪ q
        end if
    end while
    return Q
end function
```

▷ The algorithm Q_{approx} may produce a solution with a ball/s that does not completely lie in the bounds F, and because of that, it is sub-optimal.

There is an approach that will fix the deficit produced by Q_{approx} that has a runtime equal to that of Q_{approx} multiplied by $\mathcal{O}(|S|)$, S the set of points. This approach is virtually identical to Q_{approx} except for two important changes by filter functions.

```
function F_1(S, F, r)
                                                    ▷ S is set of points
                                                    ▷ F is the bounds that S lies in
                                                    ▷ r is the maximum radius per ball
    S' ← {}
    for s ∈ S do
        if all_{b∈boundaries(F)}(‖s − b‖ ≥ r) then
            S' ← S' ∪ s
        end if
    end for
    return S'
end function
```

The function $\mathcal{F}_1$ filters out any point in S that lie at a distance less than maximum radius r from any boundary of F. There are $2^{|rows(F)|}$ boundaries.

```
function F_2(S, p, r)
                                                    ▷ S is set of points
                                                    ▷ p is point in question
                                                    ▷ r is the maximum radius per ball
    S' ← {}
    p' = ∅
    d = 0.0
    for s ∈ S do
        d' = ‖s − p‖
        if d' > r then
            continue
        end if
        if d' > d then
            p' ← s
            d ← d'
        end if
    end for
    return p'
end function
```

The function $\mathcal{F}_2$ determines the point p_2 in a set of points S that has a maximum distance to p and satisfies

$$\|p_2 - p\| \leq r.$$

This function will be used to calculate the point for the center of a new ball given an arbitrary point p that does not lie in any balls in the running solution for d-occupancy. The new approach that fixes the issue of Q_{approx} is $Q^{(2)}_{approx}$.

```
function Q^(2)_approx(S, F, r)
                                          ▷ S is a subset of points in F
                                                          ▷ F is a bound

    function L(p_x)
        for q ∈ Q do
            if ‖p_x − q[0]‖ ≤ r then
                return q
            end if
            return ∅
        end for
    end function
                                                              ▷ sort S
    S ← argsort_{s∈S} ‖s − F[:, 0]‖
    S' ← F_1(S, F, r)
    q ←<>
    Q ← {}
    while |S| > 0 do
        p ← pop(S, 0)
        l ← L(p)
        if l ≠ ∅ then
            l.append(p)
        else
            p_2 ← F_2(S', p, r)
            if p_2 ≠ ∅ then
                q ←< p_2, p >
            else
                q ←< p >
            end if
            Q ← Q ∪ q
            S' ← S' − {q}
        end if
    end while
    return Q
end function
```

5.7 Rationale Behind BallComp

An important fact about BallComp is that its *solution is strongly influenced by temporality.* The balls in its solution set are centered at points that are more likely to be found at the beginning of the algorithm's processing than near the end, when its solution is converging on the maximum ball radius and maximum number of balls. This is because *the algorithm sees each point it processes as relevant to the solution* of balls that it constructs and modifies. This principle can certainly be a weakness when the algorithm processes predominantly outlier

data in the beginning. But a strength of this algorithm is that it is able to hypothesize on a solution immediately after it starts processing samples, and it is equipped with the two soft threshold variables used to instantiate it, so *if there is outlier data at any point during BallComp's processing, the scores output from the algorithm's pathways P_1 and P_2 will allow it to recognize the subset of data as deviant.*

Another noteworthy attribute about BallComp is that its pathway P_1 penalizes for intersectional volume between balls. One of BallComp's purposes in its objective to construct a context for the data it processes is classification. *Intersectional volumes between balls are multi-labels*, and may introduce more ambiguity than post-analysis can conclude based on the number of intersectional volumes shared by a ball and its neighbors.

BallComp is designed for online processing of data that it can contextualize based on the data's inherent relational structure over time. The below two sections 5.7.1 and 5.7.2 briefly describe ideas for variants.

5.7.1 Variant 1: Equalizing BallComp

This variant, $B^{(C)}_{(E)}$, can be used during BallComp's processing of *new data*, data in which it does not have information for a strong hypothesis. It is essentially a procedure to adjust a ball by its center and shrink it to a minimal radius in which all points that were in the ball before remain in the ball afterwards.

During processing, BallComp will make note of balls with an "imbalance" of points, position-wise. This "imbalance" can be numerically described by the proportion of points of nearest distance to each axis of the ball.

For any ball $B^{(x)}$ that is properly recognized as imbalanced, BallComp will calculate a line segment from $B^{(x)}$ to the "heavy" part of $B^{(x)}$, and move $B^{(x)}_c$ to the midpoint p_m of the line segment. The new radius of $B^{(x)}$ is

$$\texttt{argmax}_{p \in B^{(x)}} \|p - p_m\| .$$

5.7.2 Variant 2: Ranker

This variant, $B^{(C)}_{(R)}$, retrains the points that it has previously processed by first ranking those points in descending order based on its relevance to a wanted solution. Ranking is a process that would differentiate this variant from $B^{(C)}$ by making it a semi-supervised or supervised learning procedure. After the ranking has taken place, $B^{(C)}_{(R)}$ proceeds in a similar manner to $B^{(C)}$.

6 Ending Comments On Algorithm

BallComp is a meta-heuristic algorithm that bases its calculations regarding volume on fuzzy logic. It is designed to be used with some presumption beforehand about the data. This fact is indicated by the arguments, maximum ball radius and maximum number of balls, that is used to initialize it. This

algorithm can be considered a temporal-based multi-label classifier. It is also an online learning system that uses a non-concrete classification methodology to structure its solution, a set of balls (each is a classification), around the points it receives.

The running solution of BallComp during its data reception is a set of balls centered at points based on factors that include the ordering of the points that it receives. *Points that BallComp receives earlier in its processing will have greater probability of forming new balls*, if all other factors such as remaining number of balls and unique volume are equal. BallComp thus operates by a pattern of convergence. This convergence is most strongly noted in cases in which BallComp continues to process points that do not veer too far out of a given n-rectangular bounds. BallComp will be less likely to assign these points to new balls.

BallComp can be thought of as a "coverage" system that uses meta-heuristics and programmable formulae to manage an inflow of data into a topological environment based on n-dimensional balls so that more specific calculations can be executed in the future without the additional work of data pre-analysis. This topological environment is the context and the fuzzy solution that BallComp seeks to produce.

A word of caution, this is a system that has not been fully tested. *There are certainly topological spaces that will make BallComp produce sub-optimal results.* For instance, spaces that are narrow and elongated will force BallComp to use a relatively small maximum radius and a relatively large maximum number of balls to accomodate the data samples.

Sources

[San] Arturo Sangalli. *fuzzy logic*. URL: https://www.britannica.com/science/fuzzy-logic. (accessed: 012.01.2021).