The Concave Hull

João Paulo Figueira Oct 26, 2018 · 12 min read

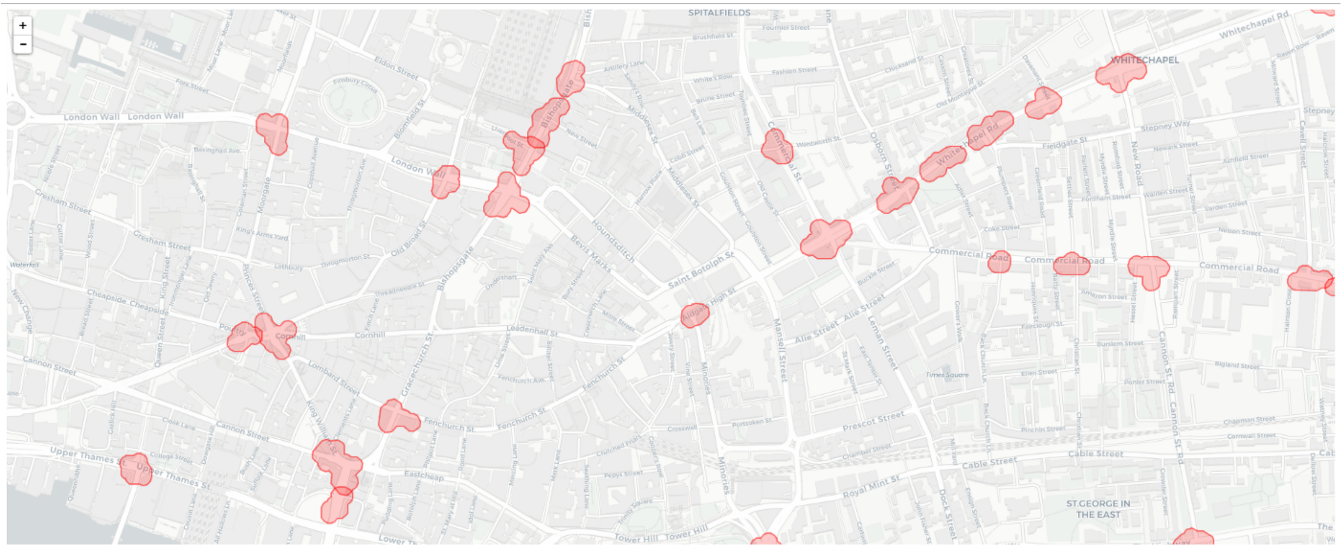
Creating a cluster border using a *K*-nearest neighbors approach

“white boat on green grass field under gray sky” by Daniel Ian on Unsplash the DBSCAN clustering algorithm on geographical data. In the article, I used geographical information published by the UK government on reported traffic accidents. My purpose was to run a density-based clustering process to find the areas where traffic accidents are reported most frequently. The end result was the creation of a set of geo-fences representing these accident hot spots.

By collecting all points in a given cluster, you can get an idea on how the cluster would look like on a map, but you will lack an important piece of information: the cluster's outer shape. In this case, we are talking about a closed polygon that can be represented on a map as a geo-fence. Any point inside the geo-fence can be assumed to belong to the cluster, which makes this shape an interesting piece of information: you can use it as a discriminator function. All newly sampled points that fall inside the polygon can be assumed to belong to the corresponding cluster. As I hinted in the article, you could use such polygons to assert your driving risk, by using them to classify your own sampled GPS location.

From Cloud to Polygon

The question now is how to create a meaningful polygon out of a cloud of points that make up a specific cluster. My approach in the first article was somewhat naïve and reflected a solution that I had already used in production code. This solution entailed placing a circle centered at each one of the cluster's points and then merging all circles together to form a cloud-shaped polygon. The result is not very nice, nor realistic. Also, by using circles as the base shape for building the final polygon, these will have more points than a more streamlined shape, thereby increasing the storage costs and making the inclusion detection algorithms slower to run.

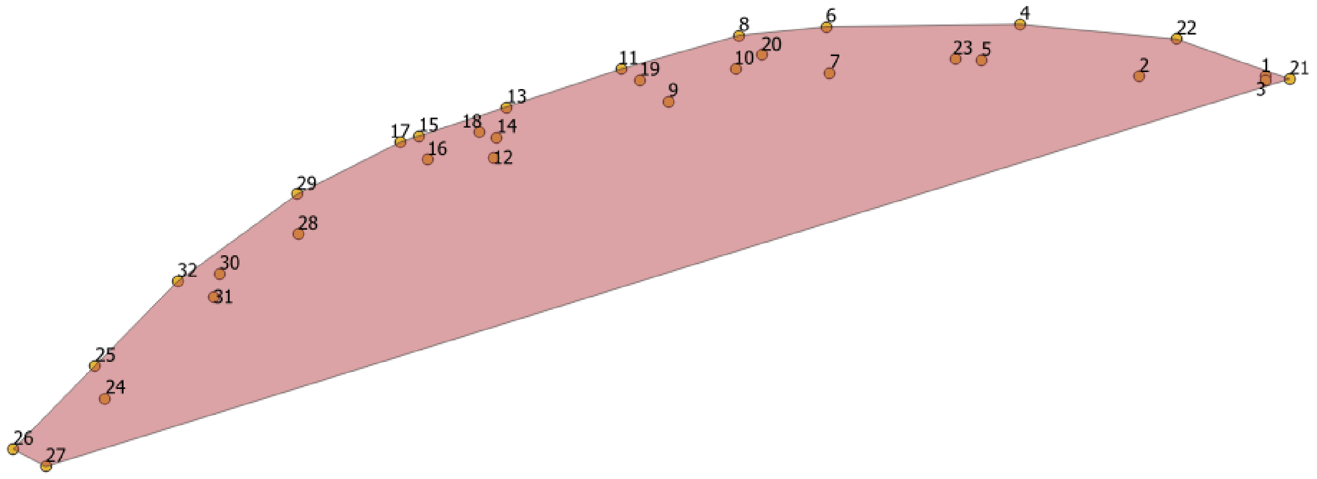


Cloud-shaped polygons

On the other hand, this approach has the advantage of being computationally simple (at least from the developer's perspective), as it uses Shapely's `cascaded_union` function to merge all circles together. Another advantage is that the polygon's shape is implicitly defined using all points in the cluster.

For more sophisticated approaches, we need to somehow identify the cluster's border points, the ones that seem to define the point cloud shape. Interestingly, with some DBSCAN implementations [1] you can actually recover the border points as a byproduct of the clustering process. Unfortunately, this information is (apparently) not available on SciKit Learn's [2] implementation, so we have to make do.

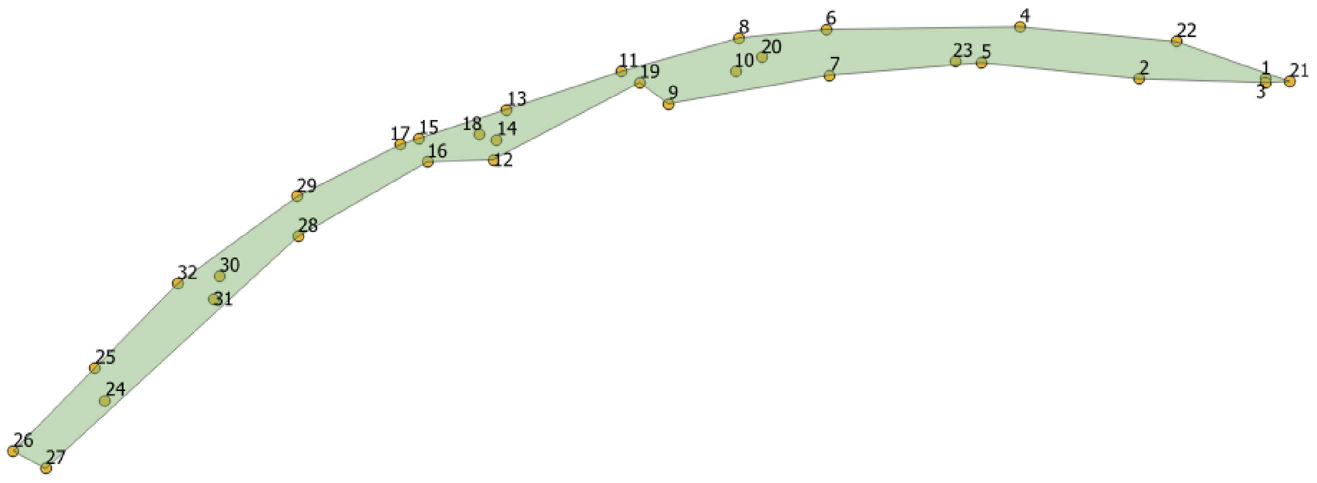
The first approach that sprang to mind was to calculate the convex hull of the set of points. This is a well-understood algorithm but suffers from the problem of not handling concave shapes, like this one:



The convex hull of a concave set of points

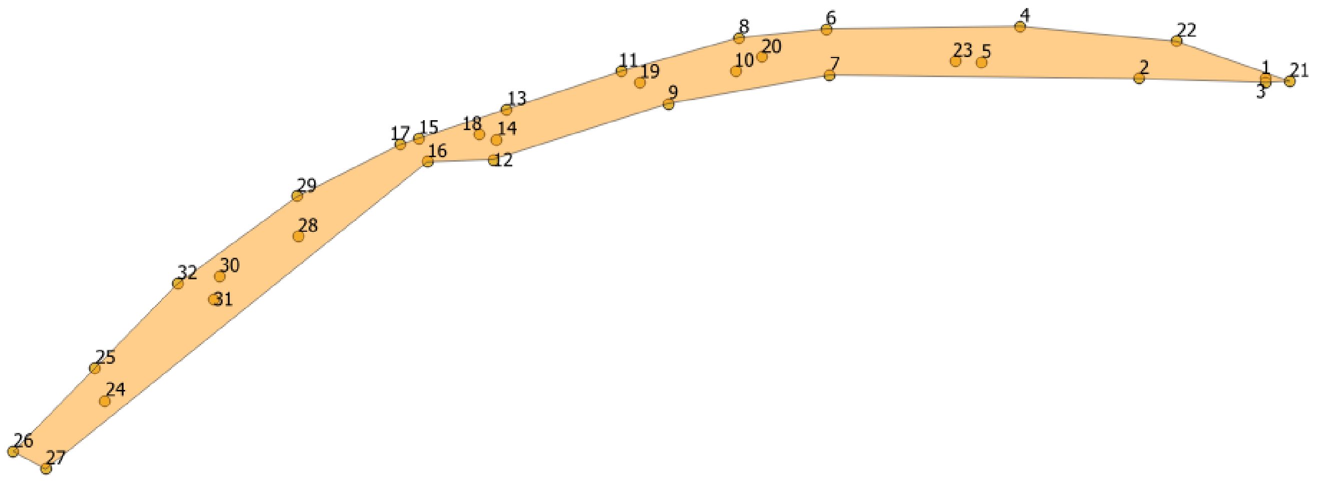
This shape does not correctly capture the essence of the underlying points. Were it used as a discriminator, some points would be incorrectly classified as being inside the cluster when they are not. We need another approach.

hull. Here’s what the concave hull looks like when applied to the same set of points as in the previous image:



Concave Hull

Or maybe this one:



A less concave hull

choice of how concave you want your hulls to be is made through a single parameter: *k* — the number of nearest neighbors considered during the hull calculation. Let's see how this works.

The Algorithm

The algorithm I'm presenting here was described over a decade ago by Adriano Moreira and Maribel Yasmina Santos from the University of Minho, Portugal [3]. From the abstract:

> *This paper describes an algorithm to compute the envelope of a set of points in a plane, which generates convex on non-convex hulls that represent the area occupied by the given points. The proposed algorithm is based on a k-nearest neighbors approach, where the value of k, the only algorithm parameter, is used to control the "smoothness" of the final solution. […]*

Because I will apply this algorithm to geographical information, some changes had to be made, namely when calculating angles and distances [4]. But these do not alter in any way the gist of the algorithm, that can be broadly described by the following steps:

1. Find the point with the lowest *y* (latitude) coordinate and make it the current one.
2. Find the *k*-nearest points to the current point.
3. From the *k*-nearest points, select the one which corresponds to the largest right-hand turn from the previous angle. Here we will use the concept of bearing and start with an angle of 270 degrees (due West).
4. Check if by adding the new point to the growing line string, it does not intersect itself. If it does, select another point from the *k*-nearest or restart with a larger value of *k*.
5. Make the new point the current point and remove it from the list.
6. After *k* iterations add the first point back to the list.
7. Loop to number 2.

The algorithm seems to be quite simple, but there are a number of details that must be attended to, especially because we are dealing with geographic coordinates. Distances

The Code

Here I am publishing is an adapted version of the previous article's code. You will still find the same clustering code and the same cloud-shaped cluster generator. The updated version now contains a package named `geomath.hulls` where you can find the `ConcaveHull` class. To create your concave hulls do as follows:

```python
# Create the concave hull object
concave_hull = ConcaveHull(points)

# Calculate the concave hull array
hull_array = concave_hull.calculate()
```

ShowHotSpots_1.py hosted with ♡ by GitHub view raw

In the code above, `points` is an array of dimensions (N, 2), where the rows contain the observed points and the columns contain the geographic coordinates (*longitude*, *latitude*). The resulting array has exactly the same structure, but contains only the points that belong in the polygonal shape of the cluster. A filter of sorts.

Because we will be handling arrays it's only natural to bring NumPy into the fray. All calculations were duly vectorized, whenever possible, and efforts were made to improve performance when adding and removing items from arrays (spoiler: they aren't moved at all). One of the missing improvements is code parallelization. But that can wait.

I organized the code around the algorithm as exposed in the paper, although some optimizations were made during translation. The algorithm is built around a number of subroutines that are clearly identified by the paper, so let's get those out of the way right now. For your reading comfort, I will use the same names as used in the paper.

CleanList[listOfPoints] —The cleaning the list of points is performed in the class constructor:

```python
    def __init__(self, points, prime_ix=0):
        if isinstance(points, np.core.ndarray):
            self.data_set = points
        elif isinstance(points, list):
            self.data_set = np.array(points)
        else:
```

```
9              # Clean up duplicates
10             self.data_set = np.unique(self.data_set, axis=0)
11
12             # Create the initial index
13             self.indices = np.ones(self.data_set.shape[0], dtype=bool)
14
15             self.prime_k = np.array([3, 5, 7, 11, 13, 17, 21, 23, 29, 31, 37, 41, 43,
16                                      47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97])
17             self.prime_ix = prime_ix
```

As you can see, the list of points is implemented as a NumPy array for performance reasons. The cleaning of the list is performed on line 10, where only the unique points are kept. The data set array is organized with observations in rows and geographic coordinates in the two columns. Note that I am also creating a Boolean array on line 13 that will be used to index into the main data set array, easing the chore of removing items and, once in a while, adding them back. I have seen this technique called a "mask" on the NumPy documentation, and it is very powerful. As for the prime numbers, I will discuss them later.

FindMinYPoint[listOfPoints] — This requires a small function:

```
    @staticmethod
    def get_lowest_latitude_index(points):
        indices = np.argsort(points[:, 1])
        return indices[0]
```

This function is called with the dataset array as the argument and returns the index of the point with the lowest latitude. Note that rows are encoded with longitude in the first column and latitude in the second.

RemovePoint[vector,e]
AddPoint[vector,e] — These are no-brainers, due to the use of the `indices` array. This array is used to store the active indices in the main data set array, so removing items from the data set is a snap.

```python
# Create a test hull to check if there are any self-intersections
next_point = np.reshape(self.data_set[knn[candidate]], (1,2))
test_hull = np.append(hull, next_point, axis=0)
```

ConvexHull_testhull.py hosted with ♡ by GitHub view raw

Later, the `test_hull` variable will be assigned back to `hull` when the line string is deemed as non-intersecting. But I'm getting ahead of the game here. Removing a point from the dataset array is as simple as:

```python
self.indices[current_point] = False
```

Adding it back is just a matter of flipping that array value at the same index back to true. But all this convenience comes with the price of having to keep our tabs on the indexes. More on this later.

NearestPoints[listOfPoints, point, k] — Things start to get interesting here because we are not dealing with planar coordinates, so out with Pythagoras and in with Haversine:

```python
    def haversine_distance(self, loc_ini, loc_end):
        lon1, lat1, lon2, lat2 = map(np.radians,
                                     [loc_ini[0], loc_ini[1],
                                      loc_end[:, 0], loc_end[:, 1]])

        delta_lon = lon2 - lon1
        delta_lat = lat2 - lat1

        a = np.square(np.sin(delta_lat / 2.0)) + \
            np.cos(lat1) * np.cos(lat2) * np.square(np.sin(delta_lon / 2.0))

        c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1.0 - a))
        meters = 6371000.0 * c
        return meters
```

ConcaveHull_haversine_distance.py hosted with ♡ by GitHub view raw array of distances in meters between the point in the second argument and the points in the third argument. Once we have these, we can get the *k*-nearest neighbors the easy way. But there is a specialized function for that and it deserves some explanations:

```python
    def get_k_nearest(self, ix, k):
        """
        Calculates the k nearest point indices to the point indexed by ix
        :param ix: Index of the starting point
        :param k: Number of neighbors to consider
        :return: Array of indices into the data set array
        """
        ixs = self.indices

        base_indices = np.arange(len(ixs))[ixs]
        distances = self.haversine_distance(self.data_set[ix, :], self.data_set[ixs, :])
        sorted_indices = np.argsort(distances)

        kk = min(k, len(sorted_indices))
        k_nearest = sorted_indices[range(kk)]
        return base_indices[k_nearest]
```

ConcaveHull_get_k_nearest.py hosted with ♡ by GitHub view raw

The function starts by creating an array with the base indices. These are the indices of the points that have not been removed from the data set array. For instance, if on a ten point cluster we started by removing the first point, the base indices array would be [1, 2, 3, 4, 5, 6, 7, 8, 9]. Next, we calculate the distances and sort the resulting array indices. The first *k* are extracted and then used as a mask to retrieve the base indices. It's kind of contorted but works. As you can see, the function does not return an array of coordinates, but an array of indices into the data set array.

SortByAngle[listOfPoints, point, angle] — There's more trouble here because we are not calculating simple angles, we are calculating bearings. These are measured as zero degrees due North, with angles increasing clockwise. Here's the core of the code that calculates the bearings:

```python
    def calculate_headings(self, ix, ixs, ref_heading=0.0):
        """
        Calculates the headings from a source point to a set of target points.
        :param ix: Index to the source point in the data set
        :param ixs: Indexes to the target points in the data set
        if ref_heading < 0 or ref_heading >= 360.0:
            raise ValueError('The reference heading must be in the range [0, 360)')

        r_ix = np.radians(self.data_set[ix, :])
        r_ixs = np.radians(self.data_set[ixs, :])

        delta_lons = r_ixs[:, 0] - r_ix[0]
        y = np.multiply(np.sin(delta_lons), np.cos(r_ixs[:, 1]))
        x = math.cos(r_ix[1]) * np.sin(r_ixs[:, 1]) - \
            math.sin(r_ix[1]) * np.multiply(np.cos(r_ixs[:, 1]), np.cos(delta_lons))
        bearings = (np.degrees(np.arctan2(y, x)) + 360.0) % 360.0 - ref_heading
        bearings[bearings < 0.0] += 360.0
        return bearings
```

The function returns an array of bearings measured from the point whose index is in the first argument, to the points whose indices are in the third argument. Sorting is simple:

```
            # Calculates the headings between first_point and the knn points
            # Returns angles in the same indexing sequence as in knn
            angles = self.calculate_headings(current_point, knn, prev_angle)

            # Calculate the candidate indexes (largest angles first)
            candidates = np.argsort(-angles)
```

ConcaveHull_sort_bearing.py hosted with ♡ by GitHub view raw

At this point, the candidates array contains the indexes of the *k*-nearest points sorted by descending order of bearing.

IntersectQ[lineSegment1, lineSegment2] — Instead of rolling my own line intersection functions, I turned to Shapely for help. In fact, while building the polygon, we are essentially handling a line string, appending segments that do not intersect with the previous ones. Testing for this is simple: we pick up the under construction hull array, convert it to a Shapely line string object, and test if it is simple (non self-intersecting) or not.

```
                # Create a test hull to check if there are any self-intersections
                next_point = np.reshape(self.data_set[knn[candidate]], (1,2))
```

```
            invalid_hull = not line.is_simple
```

ConcaveHull_check_self_intersect.py hosted with ♡ by GitHub view raw

In a nutshell, a Shapely line string becomes complex if it self-crosses, so the `is_simple` predicate becomes false. Easy.

PointInPolygon[point, listOfPoints] — This turned out to be the trickiest one to implement. Allow me to explain by looking at the code that performs the final hull polygon validation (checks if all the points of the cluster are included in the polygon):

```
        poly = asPolygon(hull)

        count = 0
        total = self.data_set.shape[0]
        for ix in range(total):
            pt = asPoint(self.data_set[ix, :])
            if poly.intersects(pt) or pt.within(poly):
                count += 1
            else:
                d = poly.distance(pt)
                if d < 1e-5:
                    count += 1

        if count == total:
            return hull
        else:
            return self.recurse_calculate()
```

ConcaveHull_test_polygon.py hosted with ♡ by GitHub view raw

Shapely's functions to test for intersection and inclusion should have been enough to check if the final hull polygon overlaps all the cluster's points, but they were not. Why? Shapely is coordinate-agnostic, so it will handle geographic coordinates expressed in latitudes and longitudes exactly the same way as coordinates on a Cartesian plane. But the world behaves differently when you live on a sphere, and angles (or bearings) are not constant along a geodesic. The example on reference [4] of a geodesic line connecting Baghdad to Osaka perfectly illustrates this. It so happens that under some circumstances, the algorithm can include a point based on the bearing criterion but

It took me a while to figure this out. My debugging help was QGIS, a great piece of free software. At every step of the suspicious calculations I would output the data in WKT format into a CSV file to be read in as a layer. A real lifesaver!

Finally, if the polygon fails to cover all the cluster's points, the only option is to increase *k* and try again. Here I added a bit of my own intuition.

Prime k

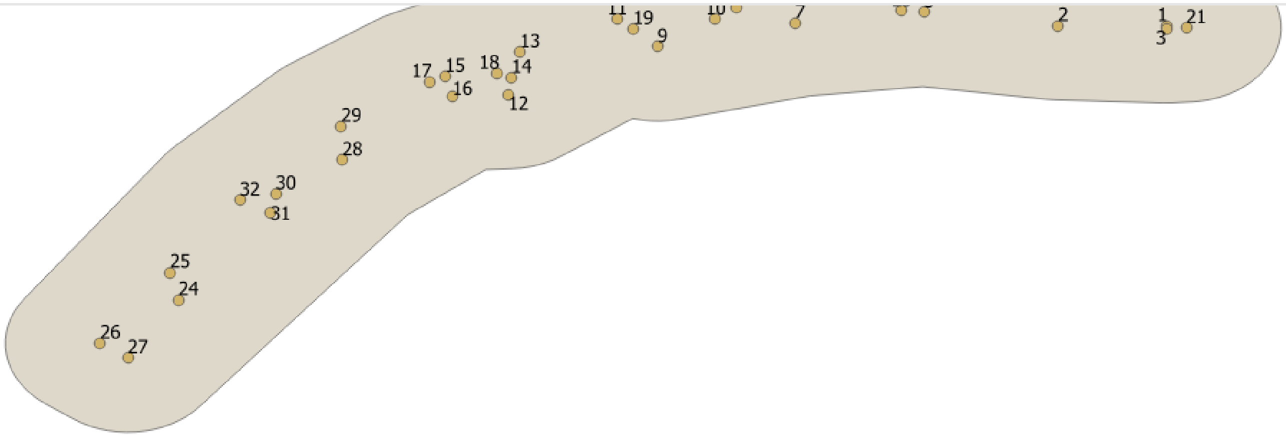
The article suggests that the value of *k* be increased by one and that the algorithm is executed again from scratch. My early tests with this option were not very satisfactory: run times would be slow on problematic clusters. This was due to the slow increase of *k*, so I decided to use another increase schedule: a table of prime numbers. The algorithm already starts off with *k*=3, so it was an easy extension to make it evolve on a list of prime numbers. This is what you see happening in the recursive call:

```python
    def recurse_calculate(self):
        """
        Calculates the concave hull using the next value for k while reusing the distances dict
        :return: Concave hull
        """
        recurse = ConcaveHull(self.data_set, self.prime_ix + 1)
        next_k = recurse.get_next_k()
        if next_k == -1:
            return None
        return recurse.calculate(next_k)
```

I have a thing for prime numbers, you know…

Blow Up

The concave hull polygons generated by this algorithm still need some further processing, because they will only discriminate points inside of the hull, but not close to it. The solution is to add some padding to these skinny clusters. Here I am using the exact same technique as used before, and here is what it looks like:



Buffered concave hull

Here, I used Shapely's `buffer` function to do the trick.

```python
    @staticmethod
    def buffer_in_meters(hull, meters):
        proj_meters = pyproj.Proj(init='epsg:3857')
        proj_latlng = pyproj.Proj(init='epsg:4326')

        project_to_meters = partial(pyproj.transform, proj_latlng, proj_meters)
        project_to_latlng = partial(pyproj.transform, proj_meters, proj_latlng)

        hull_meters = transform(project_to_meters, hull)

        buffer_meters = hull_meters.buffer(meters)
        buffer_latlng = transform(project_to_latlng, buffer_meters)
        return buffer_latlng
```

ConcaveHull_buffer_in_meters.py hosted with ♡ by GitHub view raw

The function accepts a Shapely Polygon and returns an inflated version of itself. The second parameter is the radius in meters of the added padding.

Running the Code

Start by pulling the code from the GitHub repository into your local machine. The file you want to execute is `ShowHotSpots.py` in the main directory. Upon first execution, the code will read in the UK traffic accident data from 2013 to 2016 and cluster it. The results are then cached as a CSV file for subsequent runs.

While the polygon generation code executes, you may see that a few failures are reported. To help understand why the algorithm fails to create a concave hull, the code writes the clusters to CSV files to the `data/out/failed/` directory. As usual, you can use QGIS to import these files as layers.

Essentially this algorithm fails when it does not find enough points to "go around" the shape without self-intersecting. This means that you must be ready to either discard these clusters, or to apply a different treatment to them (convex hull or coalesced bubbles).

Concaveness

It's a wrap. In this article I presented a method of post-processing DBSCAN generated geographical clusters into concave shapes. This method can provide a better-fitting outside polygon for the clusters as compared to other alternatives.

Thank you for reading and enjoy tinkering with the code!

References

[1] Kryszkiewicz M., Lasek P. (2010) TI-DBSCAN: Clustering with DBSCAN by Means of the Triangle Inequality. In: Szczuka M., Kryszkiewicz M., Ramanna S., Jensen R., Hu Q. (eds) Rough Sets and Current Trends in Computing. RSCTC 2010. Lecture Notes in Computer Science, vol 6086. Springer, Berlin, Heidelberg [Springer Link]

[2] Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825–2830, 2011

[3] Moreira, A. and Santos, M.Y., 2007, Concave Hull: A *K*-nearest neighbors approach for the computation of the region occupied by a set of points [PDF]

[4] Calculate distance, bearing and more between Latitude/Longitude points

[5] GitHub repository

Machine Learning K Nearest Neighbours Clustering GIS Data