Bisecting K-Means Analysis

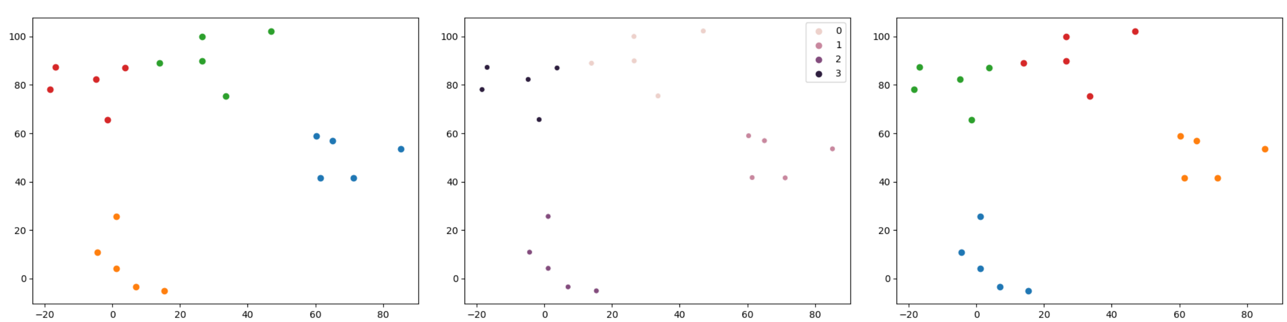
David Gillis

For this project, we are asked to implement the bisecting k-means algorithm, with two and four clusters. To test the algorithm, I've used the "make_blobs" method from "sklearn". For each case, I've generated a dataset with the appropriate number of clusters.

The configurations for both clustering, and generating the data are stored in a JSON configuration file. To spread the clusters out a bit, the standard deviation is set to 10. We can easily generate two or four clusters by changing the "n_samples" and "k" configurations. It is worth noting that these are independent for a reason - we can generate more clusters than our algorithm tries to find. While this is not explored in this paper, it is an interesting experiment. Finally, the seed is stored in the configuration so that our dataset is the same across multiple runs.

```json
{
  "seed": 42,
  "data":{
    "cluster_std": 10,
    "center_box": [-10, 100],
    "centers": 4,
    "n_samples": 20
  },
  "clustering": {
    "k": 4
  }
}
```

Because the data is generated in clusters, we are able to plot which clusters the data is "supposed to" belong to. For $k = 4$, we have:

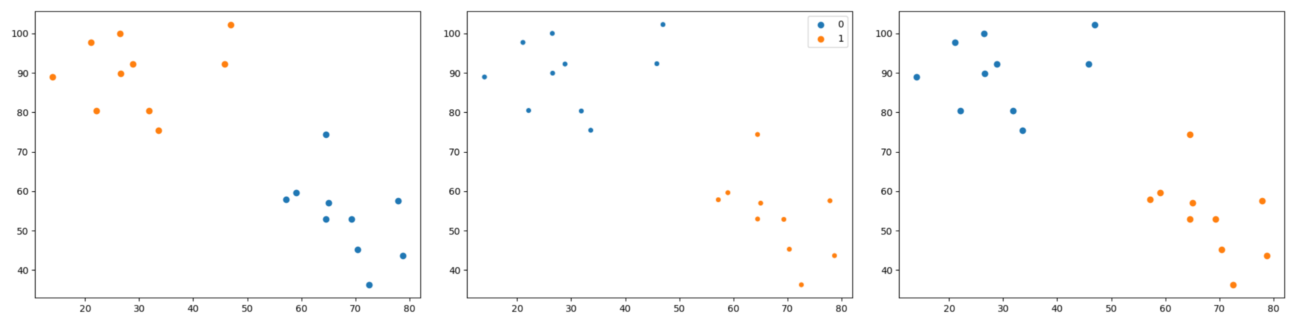


From left to right - euclidean clusters, original data, Manhattan clusters.

Finally, for $k = 4$, we have the following intra-cluster-distances:

```
MANHATTAN:
    Intra-Cluster Distances
    Max: 145.6351
    Min: 12.1390
    Mean: 90.3561
EUCLIDEAN:
    Intra-Cluster Distances
    Max: 112.9754
    Min: 10.3639
    Mean: 70.9461
```

Moving on to $k = 2$, we have:



From left to right - euclidean clusters, original data, Manhattan clusters.

```
MANHATTAN:
    Intra-Cluster Distances
    Max: 113.0022
    Min: 31.9538
    Mean: 74.4292
EUCLIDEAN:
    Intra-Cluster Distances
    Max: 80.2124
    Min: 25.8792
    Mean: 54.0818
```

```python
"""
NOTES:
    The epochs are implemented in the simple KMeans. With this strategy,
    we naively overcome the problems of poor initialization. Worth noting
    that this slows the algorithm down. For large datasets, this is obviously
    problematic. Furthermore, the initialization isn't enhanced between epochs.
"""
import numpy as np
from scipy.spatial.distance import cityblock, euclidean
import json
import os
import itertools


def configfile(filename='default') -> dict:
    filename = filename + '.json'
    filename = os.path.join(
        project_path(), filename
    )
    with open(filename, 'r') as c:
        cfg = json.loads(c.read())
    return cfg


def project_path() -> str:
    return os.path.dirname(
        os.path.abspath(__file__)
    )


METRICS = {
    'manhattan': cityblock,
    'euclidean': euclidean
}


AGGR = {
    'min': np.min,
    'max': np.max,
    'mean': np.mean
}


def make_2d_array(data: np.ndarray) -> np.ndarray:
    data = np.array(data)
    if len(data.shape) == 1:
        data = np.expand_dims(data, -1)
    return data


def sum_squared_error(points: np.ndarray) -> float:
    points = make_2d_array(points)
    centroid = np.mean(points, 0)
    errors = np.linalg.norm(points-centroid, ord=2, axis=1)
    return np.sum(errors)


class KMeans:
    def __init__(self, k=2):
        self.k = k

    def random_centroids(self, points: np.ndarray) -> np.ndarray:
        np.random.shuffle(points)
        return points[0:self.k]
```

```
    def fit(self):
        raise NotImplementedError

    def predict(self):
        raise NotImplementedError


class SimpleKMeans(KMeans):
    def __init__(self, k=2, epochs=10, max_iters=100, keep_training_history=True,
distance_metric='euclidean'):
        self.k = k
        self.epochs = epochs
        self.max_iters = max_iters
        self._clusters = [None]
        self.distance_metric = METRICS.get(distance_metric)
        if self.distance_metric is None:
            raise ValueError('Invalid distance metric')
        if keep_training_history:
            self._training_epoch_history = []
            self._training_iteration_history = []
            self._training_sse_history = []
        else:
            self._training_epoch_history = None

    @property
    def training_history(self):
        return self._training_epoch_history, self._training_iteration_history,
self._training_sse_history

    @property
    def clusters(self):
        return self._clusters

    def fit(self, predictors):
        """
        :param predictors:
        :return:
        """
        points = make_2d_array(np.copy(predictors))
        assert len(predictors) >= self.k

        best_sse = np.inf
        for epoch in range(self.epochs):
            centroids = self.random_centroids(points)

            last_sse = np.inf
            for iteration in range(self.max_iters):
                clusters = [None] * self.k
                for point in points:
                    index = np.argmin([
                        self.distance_metric(centroid, point) for centroid in
centroids
                    ])
                    if clusters[index] is None:
                        clusters[index] = np.expand_dims(point, 0)
                    else:
                        clusters[index] = np.vstack((clusters[index], point))

                centroids = [np.mean(cluster, 0) for cluster in clusters]

                sse = np.sum([sum_squared_error(cluster) for cluster in clusters])
                delta = last_sse - sse
                if sse < best_sse:
```

```
                    best_clusters, best_sse = clusters, sse

                if self._training_epoch_history is not None:
                    self._training_epoch_history.append(epoch)
                    self._training_iteration_history.append(iteration)
                    self._training_sse_history.append(sse)

                if np.isclose(delta, 0, atol=0.0001):
                    break
                last_sse = sse

        self._clusters = best_clusters

    def predict(self, predictors):
        pass


class BisectingKMeans(KMeans):
    def __init__(self, k=2, max_iters=10, distance_metric='euclidean'):
        self.k = k
        self.max_iters = max_iters
        self._clusters = [None]
        self.distance_metric = METRICS.get(distance_metric)
        if self.distance_metric is None:
            raise ValueError('Invalid distance metric')

    @property
    def clusters(self):
        return self._clusters

    def fit(self, predictors):
        predictors = make_2d_array(np.copy(predictors))
        self._clusters = [predictors]

        while len(self._clusters) < self.k:
            next_cluster_index = np.argmax([sum_squared_error(cluster) for cluster in
self._clusters])
            split_cluster = self._clusters.pop(next_cluster_index)
            c = SimpleKMeans(k=2, keep_training_history=False)
            c.fit(split_cluster)
            self._clusters.extend(c.clusters)

    def predict(self, predictors):
        pass

    def intra_cluster_metric(self, method='mean') -> float:
        aggr = AGGR.get(method)
        if aggr is None:
            raise ValueError('Invalid method')
        return aggr([
            self.distance_metric(p_1, p_2)
            for c_1, c_2 in itertools.combinations(self.clusters, 2)
            for p_1, p_2 in itertools.product(c_1, c_2)
        ])


def clustering_report(manhattan: BisectingKMeans, euclidean: BisectingKMeans) -> None:
    report_string = '''
    MANHATTAN:
        Intra-Cluster Distances
        Max: {:.4f}
        Min: {:.4f}
        Mean: {:.4f}
    EUCLIDEAN:
```

```
        Intra-Cluster Distances
        Max: {:.4f}
        Min: {:.4f}
        Mean: {:.4f}
    '''.format(
        manhattan.intra_cluster_metric('max'),
        manhattan.intra_cluster_metric('min'),
        manhattan.intra_cluster_metric('mean'),
        euclidean.intra_cluster_metric('max'),
        euclidean.intra_cluster_metric('min'),
        euclidean.intra_cluster_metric('mean')
    )
    print(report_string)


if __name__ == '__main__':
    from sklearn.datasets import make_blobs
    import matplotlib.pyplot as plt
    import seaborn as sns

    config = configfile()
    config['data']['center_box'] = tuple(config['data']['center_box'])

    np.random.seed(config['seed'])

    x, y = make_blobs(
        **config['data']
    )
    euclid_clusterer = BisectingKMeans(**config['clustering'])
    euclid_clusterer.fit(predictors=x)

    for cluster in euclid_clusterer.clusters:
        points = make_2d_array(cluster)
        if points.shape[1] < 2:
            points = np.hstack([points, np.zeros_like(points)])
        plt.plot(points[:, 0], points[:, 1], 'o')
    plt.title = 'Euclidean'
    plt.show()
    sns.scatterplot(x[:, 0], x[:, 1], hue=y)
    plt.title = 'Original Data'
    plt.show()

    manhattan_clusterer = BisectingKMeans(**config['clustering'],
distance_metric='manhattan')
    manhattan_clusterer.fit(predictors=x)
    for cluster in manhattan_clusterer.clusters:
        points = make_2d_array(cluster)
        if points.shape[1] < 2:
            points = np.hstack([points, np.zeros_like(points)])
        plt.plot(points[:, 0], points[:, 1], 'o')
    plt.title = 'Manhattan'
    plt.show()

    clustering_report(manhattan_clusterer, euclid_clusterer)
```