Project Writeup: KMeans Clustering and Basketball

Colab: <https://colab.research.google.com/drive/18B_ZCNgcboQ9T5ZvTou2GR3SU-TQN6p-?usp=sharing>

When I first began work on this project, I wanted to create a predictive algorithm that could determine a player’s likelihood of winning one of two awards given out annually in the National Basketball Association: Sixth Man of the Year and Defensive Player of the Year. Most attention around NBA award season focuses on Most Valuable Player and Rookie of the Year, and rightfully so; players in contention for those awards are the present and future of the league. However, I chose to analyze the lesser-known awards, hoping to find what statistics are most indicative of likelihood to win one of the awards. I should say upfront that this project is no longer a predictive algorithm, but instead research into KMeans. After processing the data I pulled for the NBA API, I had data frames with game logs for each game in the year that a player wins one of the awards mentioned above. I will return to the NBA data later in this project to gain insight via KMeans clustering. First, I have to understand KMeans at a much more comprehensive level.

I decided to pivot to a research-based project on KMeans after I experimented with the clustering algorithm on the game log data. I was initially planning to use KMeans as an exploratory tool just to find a trend before doing dimensionality reduction. After reviewing the class notes, my interest in KMeans clustering piqued. What exists in the early testing section now is a tool where, given two statistics and a number of clusters, a graph is output showing the KMeans cluster centroids for the statistics. The logic behind this is to create means of the statistics by season and find the KMeans centers of these points using the code from Lab 14. While the output graph can be interesting, it is not particularly insightful on its own. With a greater comprehension of KMeans, I can return to this tool to refine it and hopefully reveal underlying trends in the data.

**Getting an Intuitive Understanding for KMeans**

At the basic level, KMeans clustering is an unsupervised machine learning method to divide data into groups or clusters. Given a set of randomly generated points in two dimensions, separate these points into clusters by minimizing the squared Euclidean distance between the points and the centroid, or center point representative of the cluster’s mean. This concept of within-cluster variance is called inertia, and this concept proves crucially important in selecting the number of clusters as explained later. The code section titled “KMeans at the base level” uses the generate\_blobs function from Lab 14 to create a set of points, though there are now 15 points per blob and the seed has been changed to one with (somewhat) less clear cluster centers. I will be using this set of points and centers repeatedly to show the concepts described below. First, I need to explain KMeans clustering in more depth.

In the standard KMeans algorithm, k observations are chosen from the list to be the cluster centers. For now, these centroids are chosen at random, commonly by the Forgy or random partition methods. The Forgy method randomly selects k datapoints to be the cluster center, whereas random partition randomly assigns each point to cluster k and calculates the centroid as the mean of the randomly assigned points. In the Forgy method, you then compare each point with the centroids, and each point is assigned to its nearest cluster (lowest Euclidean distance). From there, you compute the average of each cluster, and compare the distances of the points to the cluster centers again, reassigning if necessary. This process ends when the clusters do not change (the centroids are fixed, and the points stop changing). Random partition follows the same process, only differing in the initial assignment process.

This process repeats a number of times specified by the n\_init parameter of sklearn’s KMeans. These iterations represent how many times KMeans runs, and thus how many different ways that initial clusters are chosen. More iterations mean a more accurate cluster with less intra-cluster variance, albeit with the tradeoff of higher computational load.

**Problem: Where should the initial clusters be?**

In the “KMeans at the base level” example, the code will repeatedly come to the same result, as the data is fairly uncomplicated, and the computer can run enough iterations to find the local optimum. However, constraining the second while maintaining simplicity reveals a problem with KMeans: suboptimal clustering! The “Problem with KMeans: Initial cluster location” example starts with a rectangle of 4 points with width 20 units and height 2 units. To the naked eye, the optimal KMeans clustering with k = 2 would be to have a centroid at (0,1) and (20,1), as each point is only 1 unit away from its centroid. However, running KMeans on these 4 points 15 times with n\_init = 1 should reveal multiple graphs with suboptimal clusters at (10, 0) and (10, 2). This simple example helps illustrate how, although it is a powerful and useful tool especially given its simplicity, KMeans (especially the standard algorithm) is not always guaranteed to find the optimal solution. With significantly larger and higher dimensional data, it may not be viable to use KMeans, calculate distances continually between many points, and reach a suboptimal clustering.

**Solution: k-means++**

Thankfully, there exists a better method for initial cluster determination that is already built into sklearn’s KMeans algorithm by default. K-means++ works by having the computer first select one of the points at random to be the centroid. From there, find the distance between the existing centroid and the other points, and randomly choose another centroid where the new centroid’s probability of selection is proportional to the distance; this will likely be the furthest point from the initial centroid. Now that you have two centroids, compare all unchosen points to both centroids and randomly select another centroid again (likely to be the maximum distance between an existing centroid and unchosen point). Repeat this process until you have k centroids, and thus k clusters.

Fitting KMeans on the set of simple 4 points another 15 with the default initialization value of ‘k-means++’ will output the correct centroids all 15 times. This is because picking one point in the rectangle leads to a high likelihood that the next centroid will be the point diagonally across, ensuring that the two clusters will be between the closer points once the clusters average the points again. K-Means++ is said to be quicker than standard K-Means, even though the cluster initialization takes longer; this occurs because the initial clusters are now much closer to their local optimum. This updated method is fallible; K-Means++ has to take k passes across the data to find all the centroids, and this can be too computationally taxing in a large data set. In addition, there are other problems with KMeans clustering that need to be addressed.

**Problem: Balancing the number of clusters against inertia**

How many clusters should the data have? Too few clusters means that the inertia, or within-cluster variance, could be too high. In this case, you may not accurately discern between groups, more easily visualized in my code example. Bringing back the blobs from earlier, only using 1 or 2 clusters leaves a variance that is far too high and there are no clearly distinguishable patterns in the cluster(s). Conversely, electing to use too many clusters risks overfitting/redundancy as well as a computational complexity nightmare. In my code example, choosing 15 clusters makes about 11 clusters redundant when they could seemingly be easily described in 4 clusters. However, estimating an easy example in two dimensions is not enough; I need to explore the existing methods for choosing an accurate number of clusters.

**Solution 1: Elbow method**

In this code section, we compare the k number of clusters to the inertia, or within-cluster sum of squares. The concept of the elbow method is to choose the point, or elbow, where the benefit of using more clusters provides diminishing return on reducing inertia. In our case, using somewhere in the range of 3 to 5 clusters seems to balance lowering inertia and reducing overfitting. However, the lack of clarity on exactly how many clusters to choose is the elbow method’s shortcoming here: it is subjective, relying on the user to correctly choose an optimal number of clusters.

**Solution 2: Silhouette**

Using the silhouette method can provide a more objective way of determining the number of clusters. This method works by measuring how close a point is to its own cluster in comparison to the other clusters. Each point is assigned a value between [-1, 1] based on the formula (b-a)/max(a,b) where a represents the average distance of a point to other points in the same cluster and b represents the average distance to the nearest other cluster. Usually, this translates to S = 1-(a/b). What is returned in the output graph is the average silhouette score for each point given a certain number of clusters. The example data has the highest silhouette score with 4 clusters, though the score is not much lower for 3 clusters.

**Solution 3: The Two Indices (Davies-Bouldin and Calinski-Harabasz)**

There are other existing methods to determine an optimal number of clusters. The first of a few different methods is the Davies-Bouldin index. This index measures separation between clusters, and lower values indicate better separation. This methodology for this index is to calculate the average distance between each point in a cluster and its centroid, calculate the distance between the centroid and the nearest cluster, and make a ratio of the first quantity divided by the second. Repeat this process for each cluster, averaging those values to find the Davies-Bouldin index. For example, a small number in the numerator (closely clustered) and a large denominator (far from the nearest cluster) would mean a small DB index. The Calinski-Harabasz index is similar; however, instead of measuring the average similarity of each cluster to its nearest neighbor, the CH index creates a ratio of separation to cohesion, or distance between clusters over intra-cluster variance. Both of these indices are included in sklearn’s metrics package, so I included a graph testing both indices against the data points from earlier. Using 3 clusters minimizes the DB index, whereas using 5 clusters maximizes the CH index. The use of 4 clusters seems to balance both indices, much like the results of the elbow method and silhouette. These methods for cluster number determination have provided evidence for the use of 4 clusters on our data, much more objective than eyeballing the data.

**Problem and Solution: Sensitivity to outliers and k-medoids**

Another of KMeans’ shortcomings is that the algorithm is that means are sensitive to outliers. Like a failing grade on an exam significantly lowers an otherwise satisfactory score in a class, an observation (or multiple) that is too far away from other observations makes clustering less effective. This can be visualized more easily using the arbitrary “bad” points that I chose; fitting this data using KMeans reveals two uneven and uninformative clusters. Checking the inertia of the KMeans fit reveals within-cluster variance of 422. However, there exists a method to reduce inertia in the presence of outliers: k-medoids.

Like a median filters out outliers better than a mean, k-medoids is better at filtering outliers than KMeans. A medoid is the point in the cluster where the dissimilarity with all other cluster points is minimized. To start this clustering method, pick k random points to be medoids and assign other points to a cluster by its closest medoid (Euclidean distance, for example). From there, you take each non-medoid and swap it with a medoid. Recalculating the distance of each data point to the new medoid, either the cost (total distance) has increased (in which case you swap the points back) or decreased (maintain the swap). This method yields a far lower inertia for the example data using the same number of clusters.

**Returning to the NBA data**

Looking back at the original NBA awards data, I want to expand on two suspicions I have about NBA trends. The first of these is that Defensive Player of the Year Winners will cluster into distinct groups when comparing steals and blocks per game. I theorize that the two should be negatively correlated, as steals tend to be higher for guards (shorter and faster) whereas blocks will be higher for taller forwards and centers. The elbow plot points towards 2 to 5 clusters, though 4 clusters scores poorly on the silhouette test. Using 3 or 5 clusters minimizes the DB between 2 and 5, and 5 clusters is far better than 3 for maximizing CH index. Applying KMeans with 5 clusters seems (by eye) to fit the data well, grouping the award winners into five groups: high blocks and low steals, high steals and low blocks, middle on both statistics, low on both, and high on both. The first three can be explained by positional differences: the taller the player, the more blocks play a role in defensive output. The clusters of interest are when a player is high or low on both. Having a comparatively high number of both blocks and steals could indicate an all-time great defensive season, whereas having low blocks and steals may indicate that other factors (team performance, player reputation, offensive contribution) were at play when voters decided on the award winner. I also applied k-medoids to the data, though the inertia was worse than for KMeans given the lack of outliers.

For the sixth man data, I considered the minutes per game against plus-minus. The latter statistic is a measure of how well the team plays when the player is on the floor. For example, a plus-minus of +10 for a game indicates that the player’s team outscored the other team by 10 points while the player was on the court. The sixth man for a basketball team acts as a fringe starter, often seeing significant playing time; because of this, I wanted to discern if more playing time for a sixth man means better or worse team performance. Applying the same tests as before reveals clustering that is not as strong though two clusters is the best of mediocre bunch. This groups the winners into comparatively high and low play time, though there is more variance in the more playing time group. Also, k-medoids does not produce a significantly lower inertia, again due to the lack of outliers.

**Conclusion**

KMeans is a simple yet powerful and effective tool in the machine learning arsenal. I now feel far more confident in solving KMeans shortcomings, including choosing an accurate number of clusters, avoiding the impact of outliers, and knowing why initial cluster selection is crucially important. Moving forward, I hope to gain a greater understanding of KMeans’ prominent role in real-world data, such as in market segmentation with respect to product and service marketing.

References:

1. NBA API (general): <https://github.com/swar/nba_api>
2. NBA API documentation for accessing player info: <https://github.com/swar/nba_api/blob/master/docs/nba_api/stats/examples.md>
3. Python Lab 14: I used a modified version of Lab 14’s generate\_blobs function to produce example data to show the different properties of KMeans clustering. I also used the code for fitting to KMeans repeatedly.
4. <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html> . I used the documentation for KMeans to help understand the outputs of the function: cluster\_centers\_, labels\_, inertia\_, and n\_iter\_. No code taken from here, though this helped with elbow method.
5. Silhouette: <https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html>
6. DB Index: <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html>
7. CH Index: <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.calinski_harabasz_score.html>
8. K-medoids: <https://scikit-learn-extra.readthedocs.io/en/stable/generated/sklearn_extra.cluster.KMedoids.html>
9. Np.hstack: <https://numpy.org/doc/stable/reference/generated/numpy.hstack.html>

The links above inspired the code and the writing, though there are no chunks fully pulled from another site. Most of the code optimizes sklearn’s built-ins found in the documentation. The links below are the other resources I used to better understand KMeans and its related concepts.

1. <https://www.geeksforgeeks.org/k-means-clustering-introduction/>
2. <https://medium.com/analytics-vidhya/comparison-of-initialization-strategies-for-k-means-d5ddd8b0350e>
3. <https://www.geeksforgeeks.org/ml-k-means-algorithm/>
4. <https://en.wikipedia.org/wiki/K-means%2B%2B>
5. <https://www.geeksforgeeks.org/elbow-method-for-optimal-value-of-k-in-kmeans/>
6. <https://www.analyticsvidhya.com/blog/2021/05/k-mean-getting-the-optimal-number-of-clusters/#:~:text=Silhouette%20Analysis,scikit%2Dlearn%2Fsklearn%20library>.
7. <https://www.geeksforgeeks.org/davies-bouldin-index/>
8. <https://www.geeksforgeeks.org/calinski-harabasz-index-cluster-validity-indices-set-3/>
9. <https://www.geeksforgeeks.org/ml-k-medoids-clustering-with-example/>