산업컴퓨팅개론 Assignment #5 2010-12086 황영석

<Subject> K-means Algorithm

< Steps >

1. Initial Centroids : K-means 알고리즘의 가장 중요한 단계 중의 하나는 초기 Centroids 값을 정하는 것이다. K++ 알고리즘은 K-means의 초기값을 구하는 우수한 방법 중의 하나이다. 214개의 데이터에서 임의로 7개의 구심점을 구하는 대신, K++ 알고리즘을 이용하면 좋은 Initial Centroids를 구할 수 있다. Initial Centroids를 정하는 것은 알고리즘이 Local Optimum에 빠지지 않도록 하는 가장 안전한 방법이자 선행되어야 하는 과제이다.

2. Distance Measure : Center와 하나의 데이터 간의 거리는 다양한 방법으로 정의할 수 있다. 1-norm, 2-norm, Mahalanobis distance, Cosine distance, Standardized distance, Correlation coefficient, Matching coefficient 등 매우 다양하다.

많은 방법을 시도해 보았지만 일반적으로 1-norm이 가장 좋은 결과를 나타내었다. 프로그램을 조금씩 고칠 때마다 다른 방법이 더 좋아지게 되었을 가능성도 배제할 순 없다. 그러나 1-norm은 크게 다른 것에 영향을 받지 않고 꽤 좋은 결과를 보장하였다.

Glass 데이터 자체를 python에서 출력해 보니, -1~1 값으로 선형변환이 되어 있었다. 따라서 특별히 정규화한다고 성능이 크게 증가하는 경향은 없었다. 다만 분산에 따라 차이를 좀더 크게 보기 위해 weight가 가미된 거리를 사용하였을 때 더 나은 결과를 보장했었다.

3. Terminate Cond: 종료 조건은 크게 2가지로 구성할 수 있다.

1) Converge : iteration 을 돌고 난 후에 구심점의 변화 폭이 없거나, ε 이하이면 수렴한다고 생각하여 알고리즘을 종료시킬 수 있다.
2) Max iteration : 수렴이 되지 않았다면 무한히 K-means 알고리즘을 돌릴순 없으므로 Max iteration 값을 설정한다. Max iteration 값을 증가시켰을 때 경우에 따라 퍼포먼스가 증가하기도하였으나 기준치 이상만 되면 크게 유의미한 결과를 보장하진 않았다.

4. Ensemble : 클러스터링 알고리즘의 성능을 위해서 여러 가지 기법들을 혼용해서 사용하는 것을 '앙상블' 이라 부른다고 한다. 좋은 결과값을 얻기 위해 파일에 여러 가지 방법이 혼용되서 사용되었다. 특히 추가로 Voting Scheme (투표)을 사용해 보았다.

일반적인 K-means 알고리즘은 단순히 1회 돌았을 때의 Cluster 값을 데이터에 배정한다. 따라서 데이터의 분산이 매우 크며, 결과 Score값의 분산 역시 최대 0.1까지도 차이가 났다. (0.20~0.30)

그런데 Voting Scheme을 사용하면 알고리즘을 여러번 돈 후에 데이터가 가장 많이 소속되었던 Cluster의 번호를 그 데이터의 label로 설정한다. Voting Scheme은 기존에 알고리즘의 성능이 일정 수준 이상될수록 계속 더 좋은 결과를 보장한다. 물론 소수의 특정 데이터가 다른 클러스터에 계속적으로 배치될 수도 있다. 그러나 전반적으로 대부분의 데이터는 자신이 속할 확률이 가장 높은 클러스터에 안정적으로 배치되었다. 따라서 알고리즘 결과 값의 분산을 굉장히 감소시켜 주었다.

< Results & Limitations >

Initial Centroids를 정하는 것만 구현하여도 Score값이 0.2 점대로 상승하는 것을 관찰할 수 있었다. K++ Algorithm은 좋은 Initial Centroids를 약간의 확률(random) 개념이 사용되어 구현되기 때문에 실험을 반복할 때마다 Score값에 변동이 존재했었다.

K++ Algorithm의 구현 이후 distance measure와 같은 몇 가지 새로운 시도를 해보았더니 Score가 0.20~0.30까지의 값을 가지며 평균 0.24~0.25로 수렴하였다. 그런데 가끔 가다가 10번의 Sample이 한 쪽으로 치우치면, 프로그램을 실행시킬 때마다 평균 Score 값도 흔들렸었다.

Voting Scheme과 distance measure에 분산에서 도출된 weight를 추가하니 성능도 평균 0.28~0.30 으로 크게 증가하였고, 무엇보다 실행에 따른 분산이 크게 감소하였다.

아쉬웠던 점은 많은 시도를 해보기에 충분한 시간이 없어서 K-means 및 Machine-Learning에 대한 궁금증을 다 풀 수 없었다는 것이다. Glass 데이터의 경우, 일반적으로 구현이 완성된 K-means Algorithm을 사용하면 Error percentage가 0.44정도 된다고 한다. 그에 다다르기에는 merge 방법 등을 사용할 수 있겠으나 python 사용 숙련도를 더욱 키운 후에 실행해보기로 하였다.

< CODE >

[mySGT_k_pp.py] : **K++ Algorithm** 을 사용하여 Initial Centroids 를 구해서 전달한다.

[mySGT.py]

def kmeans_prediction(X) **def** kmeans_prediction_for_votes(X) : Voting Scheme 을 구현하기 위한 K-means Algorithim.
def init_centers(X, k) : mySGT_k_pp.py(K++ Algorithm)을 실행하여 **Initial Centroids** 를 구한
def votes(labels) **def** countvote(X, init_votes, max_iter)
def find_max_vote(totalvotes)
: **Voting Scheme** 을 적용한 알고리즘