Data Mining: Learning From Large Data Sets

Fall Semester 2015

{usamuel, adavidso, kurmannn}@student.ethz.ch

December 10, 2015

Extracting Representative Elements

Problem formulation. The goal of this task is to learn a policy that explores and exploits among available choices in order to learn user preferences and recommend news articles to users. For this task we will use real-world log data shared by Yahoo!. The data was collected over a 10 day period and consists of 5 million log lines that capture the interaction between user visits and 271 news articles, one of which was randomly displayed for every user visit. In each round, you are given a user context and a list of available articles for recommendation. Your task is to then select one article from this pool. If the article you selected matches the one displayed to the user (in the log file), then your policy is evaluated for that log line. Otherwise, the line is discarded. Since the articles were displayed uniformly at random during the data collection phase, there is approximately 1 in 20 chance that any given line will be evaluated.

Given a set of features of images that belong to different (unknown) clusters, the goal was to find representative elements for each cluster. The quality of our selection is quantified by the sum of quadratic distances that each representative has to the elements of its cluster. This problem statement is equivalent to the solution of k-means, but we were able to use a map-reduce framework to speed up this process.

Approach and Results. To solve this problem, we used SciKit Learn's integrated implementation of kmeans, which internally uses kmeans++ to generate good starting centers that help the algorithm converge faster.

In our initial approach, we ran kmeans on the mappers to compute k=200 centers each, which were then all passed to the reducer, who in turn ran kmeans to reduce the number of centers to 100, which we then proceeded to output as our result. Increasing the number of centers computed by the mappers to 1000 gave us the surprisingly high score of 9.17742.

This was without taking into the consideration the weights of the clusters/coresets passed by the mappers. We tried to override SciPy's kmeans update centers method, but we failed because it is defined in the binary file `_k_means.x86_64-linux-gnu.so`. Another approach was to adapt the mapper to output each center multiple times proportionally to its weight. However, this approach didn't improve our score.

In our third approach, we kept the mappers mostly the same, but we tried to take into account the

weights of the centers outputted by the mappers. For this, we stopped performing kmeans in the reduce step, but instead opted for merging the points manually. For this, the reducer selected a random center and merged it with the closest other center by taking a weighted average. This approach netted us with a score of 9.02585.

Workload distribution. Alexander implemented the initial approach. At first there were some technical problems that caused his code to crash on the cluster. This issue was resolved by Samuel. The entire group then tried to optimize the parameters of the first approach and discussed other approaches. Nico tried to overwrite SciKits internal kmeans implementation. Samuel adapted the initial solution to output centers multiple times. Alexander implemented the final approach where the reducer merges the centers. Nico wrote the final report.