Data Mining: Learning from Large Data Sets - Spring Semester 2014

Bojana Dimceva (dbojana@student.ethz.ch)
Diego Ballesteros Villamizar (diegob@student.ethz.ch)

June 10, 2014

1 Approximate near-duplicate search using Locality Sensitive Hashing

The goal of this task is to detect duplicates from a dataset of videos. Two videos are defined as duplicates if they are at least 85% percent similar. We work with a dataset of already shingled elements and each video is represented with a collection of integers.

1.1 Algorithm

To solve this task, we use the Min-Hash algorithm in the mapper, while in the reducer we check the similarity of all pairs of videos that belong in the same bucket and discard the pairs that have similarity less then 85%.

As suggested in class and in [?] we simulate permuting the rows for each element to avoid actually permuting them, which is expensive. We draw our hash functions from a locality sensitive family

$$\mathcal{H} : h(x) = \min_i h(x_i)$$

Where $h(x_i) = (a \cdot x_i + b) mod c$. Where we generate a and b randomly, while c is the number of possible elements. We append k such hash functions to get our hash key that will be emitted from the mapper together with the shingles of the hashed video. The shingles are still needed to calculate the similarities in the reducer.

In the reducer we process each bucket by forming all possible pairs of video ids that belong to it. Once we have formed all possible pairs we calculate their similarity and output the pairs that have similarity at least 85%, while discarding the rest.

1.2 Parameter Tuning

We achieved F-Score of 1 by setting the k parameter to 1, i.e. by using only one hash function for the key of each video. This corresponds to doing a 256-way OR of the hash functions.

2 Large-Scale Image Classification

2.1 Algorithm

For this classification problem, the approach used was parallel stochastic gradient descent as described in the class and in [?].

In each of the mappers, a linear SVM was trained using the given subset of data and then the output coefficient vector was passed to the reducer where it was averaged to produce the final linear model for evaluation. This was possible because it was known a priori the number of mappers and there was a single reducer.

The specific details of the implementation of the SVM were left to the SGDClassifier class from the scikit-learn library [?] which implements a batch stochastic gradient descent algorithm.

Another approach that was implemented for this task was the use of stochastic sampling in order to approximate kernel SVMs, this was done by transforming the input vectors in each mapper using the RBFSampler class from scikit-learn. This is the method of using random Fourier features to approximate kernels as described in [?].

2.2 Parameter tuning

For the linear SVM method, the parameters to tune were:

- The loss function, this could be a hinge or logistic loss which implement a soft-margin SVM or logisitic regression respectively.
- The penalty function, this could be L1 or L2 which determines the sparsity of the solution.
- The regularization parameter, which helps reduce overfitting due to increased model complexity.

For the SVM with random features, two extra parameters were added:

- The gamma parameter for the RBF sampler which controls the kernel function.
- The number of random features to generate, more features imply a better approximation but it also increases the complexity of the operation.

These parameters were tuned using 10-fold cross validation on the given test data with full grid search on the parameters.

2.3 Results

The best score was attained using a linear SVM because any attempt to use the random features with a significant number of features resulted in timeouts in the mappers. However, the attained scored was halfway between the easy and hard baseline. It is possible that some values of the parameters were not explored during the cross validation or rather the approach was too simplistic to solve the problem and instead an approach like PEGASOS should have been tried.

3 Extracting Representative Elements From Large Datasets

This task consists of applying a clustering method to a large image dataset and extracting representative elements of each cluster. The task description imposes an implementation for many mappers and one reducer, so every mapper gets as input partition of the data while the reduces gets as input the outputs from all of the mappers.

Our approach for this task has the K-Means algorithm in its core. First we will present the algorithms used in the mapper, i.e. the initialization algorithm for the K-Means algorithm and our implementation of the K-Means algorithm, then we will present the aggregation algorithm used in the reducer.

3.1 Mapper:Initialization of centroids

We initialize the cluster centroids by picking from the existing data points and setting them as centroids. We want the chosen centroids to be as far away from each other as possible as suggested in [**?**]. For this purpose, we pick the first centroid at random, while every consecutive centroid is chosen as the data point whose minimal distance from all the already chosen centroids is maximal. More formally:

$$\mu_m = \max_{i\in[1,N]} \min_{j\in[1,m-1]} ||\mu_j - x_i||_2^2$$

Where $x_i \in \mathcal{R}^d, i \in [1, N]$ are the data points from the dataset, N is the cardinality of the dataset. A point that satisfies the previous equation has to be found for each centroid $\mu_m, m \in [1, K]$, where K is the number of clusters, which is given for this task. The algorithm follows.

Algorithm 1 Centroid initialization algorithm

```
Require: $nrst\mu$ Array to keep the distance to the nearest centroid for each point
 1: $\mu_1 = x_{rand\_int(1,N)}$                                    ▷ Pick first centroid randomly
 2: for $m \leftarrow 2, K$ do
 3:     for $i \leftarrow 1, N$ do
 4:         if $||\mu_{m-1} - x_i||_2^2 < nrst\mu[i]$ then
                        $$nrst\mu[i] = ||\mu_{m-1} - x_i||_2^2$$
 5:         end if          ▷ Update the minimum distance to centroid for the current point if needed
 6:         if $new\mu.min\_distance < nrst\mu[i]$ then
                        $$new\mu = x_i$$
 7:         end if          ▷ The new centroid holds the maximal minimal distance, update if needed
 8:     end for
 9:     $\mu_m = new\mu$
10: end for
```

3.2 Mapper: K-Means Algorithm

For the K-Means Algorithm we implemented our own straight forward implementation of the Lloyd's algorithm. The only difference from the standard algorithm is that we use slightly modified convergence criteria. For the algorithm to stop two criteria have to be satisfied:

1. There is no change in the maximal centroid-point distance, where the point belongs to the cluster of that particular centroid.
2. The number of iterations is not larger than 1000.

The first criteria, stopping when there is no change in the maximal centroid-point distance, is not the same as stopping when the cluster assignments no longer change. The maximal centroid-point distance is not guaranteed to change when the cluster assignments change. However, since we do aggregation in the reducer and we needed to speed up the processing in the mapper, we decided that we can afford to use this convergence criteria - stop earlier than typically and not guarantee reaching local minimum.

After we have learned the centroids, we have to emit the learned configuration for the reducer. It is important to emit the number of points per cluster as well as the coordinates of its centroid. The number of points per cluster will be useful in the reducer.

3.3 Reducer: Weighted Average of Centroids

In the reducer we merge the centroids greedily and apply a weighted averaging algorithm to the sets of merged centroids. As weight we use the information about number of points per cluster. Before we can average sets of merged centroids we have to decide which centroids are going to be merged together. For this we use a greedy approach where we keep a collection of already merged centroids and for each centroid in this collection we find its closest centroid from the current input and merge them together. The pseudo code for the merging is presented as Algorithm **??**.

Algorithm 2 Centroid merging algorithm

```
1: $merged\mu = parse_i nput()$        ▷ Initialize collection of merged centroids from the first input
2: for $l \leftarrow 1, L$ do        ▷ Assuming L mappers
3:     $current\mu = parse\_input()$
4:     for $k \leftarrow 1, K$ do
5:         $merge(merged\mu_k, arg \min_{i \in [1,L]} ||merged\mu_k - current\mu_i||_2^2)$
6:     end for
7: end for
```

After we have determined which centroids to merge we merge the centroids as suggested in [**?**], by applying:

$$\mu^* = \frac{\sum_{i=1}^{L} no\_points_i * \mu_i}{\sum_{i=1}^{L} no\_points_i}$$

4 Explore-Exploit Tradeoffs in Recommender Systems

4.1 Algorithm

For this problem, the algorithm used was linear upper confidence bound (linUCB) as described in the class' lectures and in [**?**]. For this task, only the user features provided at each evaluation step were used, so the algorithm can be classified more accurately as the linUCB with disjoint linear models (Algorithm 1 of [**?**]).

In summary, the implemented algorithm works as follows:

1. When a recommendation is requested, the Upper Confidence Bound is calculated for each article in the given subset. The UCB is computed using the closed solution for the ridge regression problem, i.e. equation **??**, presented in [**?**] where A_a is the inverse of the covariance matrix for article a, b_a is the response vector from previous feedback for article a and x_t is the vector of user features for the current recommendation. Finally, the recommended article is the one with maximum calculated UCB.

$$UCB = (A_a^{-1} b_a)^T x_t + \alpha \sqrt{x_t^T A_a^{-1} x_t} \tag{1}$$

2. After a recommendation is made and the feedback is received then there are two cases to consider:
   - If the feedback is -1, then the article recommended was not recommended by the random policy that generated the logs and therefore no information can be retrieved from that log line. The algorithm simply continues to the next recommendation step.
   - If the feedback is 0 or $+1$, then the article recommended matched the one in the log and an update can be made on the covariance matrix and the mean vector for the recommended article, this update is made using the equations given in [**?**].

4.2 Parameter tuning

In this algorithm, there is a single parameter that can be used to control the trade-off between exploitation and exploration. From equation **??** it can be observed that this value regulates the effect of the estimated covariance in the UCB, large values of α may lead to too much exploration that wastes exploitation opportunities but values to small may fall short of exploring potentially good options. In order to increase the click through rate (CTR) several submissions were made with different values of α using as a reference the results presented in [**?**] for the learning bucket, in the end it was found that the optimal value is 0.2 which is the same value that produces a peak in the results from [**?**].

The result with the chosen algorithm and parameter value were enough to surpass the hard baseline.