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

dagrunh@student.ethz.ch
amarco@student.ethz.ch
bhatlav@student.ethz.ch

June 9, 2014

Approximate near-duplicate search using Locality Sensitive Hashing

In this project we were asked to use a MapReduce model to identify files that were duplicates or near duplicates (i.e. files that were at least 85% similar), using the Jaccard distance to calculate the similarity among them. The implementation was carried out in Python.

We were given a text file as input, with each line representing the features of one video file. We were supposed to output what videos were near-duplicates of each other. In the final output file, each line represented a pair of videos we had found to be near duplicates of each other. The videos were represented by an integer, which was the line number corresponding to the video in the input file.

Mapper

We created two lists of random integers called a and b. The lists contained 256 integers each. These integers were used to create hash functions later on. We wanted all of the videos to use the same hash functions, so they used the same integers (a and b), to create hash functions.

We also created a list with 16 integers (number of hash functions divided by the number of bands) called aBand. This list was used for hashing the bands of the signature list. We also created an integer called bBand, to use for this hashing.

The mapper iterated over the videos.

For each video we created a list of signature values with 256 elements. For each of our 256 hash functions i, we iterated over the shingles for the video, applied our hash function (`a[i]*(shingle value)+b[i] mod 10000`), and stored the smallest value in the end. In the end this gave us 256 min values for each video.

After creating this list of signature values we partitioned this list into several smaller lists called bands. We had 16 of these bands. Each band contained 16 values from the signature list. Each band was hashed to a single value. The hash function we used for that was `aBand.dot(band) + bBand`.

The output from our mappers was a tuple with the ID of the band with the hash value as the key, and a tuple with the ID of the video this band correspond to, and a list with all the shingles.

Reducer

In the reducers we found and printed the duplicates.

The reducers went through the input line by line. For each line, they stored the value of the key, and compared it to the key value from the previous line. If they were the same, the video ID for this line was appended to a list of possible duplicates. If the key value was not the same, it went through the list of possible duplicates, to check whether they actually were duplicates. The duplicates that were considered true duplicates then got printed. After going through the list of possible duplicates, it then removed everything from the list, and appended the video ID for this line to the list of possible duplicates.

To check whether potential duplicates actually were duplicates, we compared the shingles of the potential duplicates, to check the actual similarity measure. Doing this, we were sure that all the false positives from the final result were removed.

The final score that we have got in this project using this strategy is 0.989821882951654, above the Hard Baseline.

Large-Scale Image Classification

This task required us to classify a set of images into one of two categories ("Nature" or "People") based on their visual content. This visual information was presented to us in the form of a vector of features describing each image, hence eliminating the need for us to perform feature extraction ourselves. The tool we employed to perform this classification was a Support Vector Machine (SVM).

Mapper

The first step undertaken by our mapper is reading in the training data and processing it to organize it into a matrix format that can be easily utilized by the rest of the solution.

Since each line corresponds to a different image and contains its description in the form of a feature vector, our procedure for the processing is to read the file in line-by-line (hence isolating each image) and then stacking its features into a matrix. Doing this for the whole dataset gives us a Nx400 feature matrix where each row contains the features of the corresponding image in the dataset. The N images make up the N rows, and the 400 columns correspond to the 400 features available for each image.

In addition, the classification data for each image (i.e. +1 or -1 depending on whether the image corresponds to the class Nature or People) is recorded in a separate vector. This is done simply by extracting the first element as we read in each line and storing it incrementally in an array that ultimately contains the classification data of all images in the training dataset.

The key component of this stage of our solution is the use of a linear-kernel SVM to fit a classification model to the training data. We were able to accomplish this by using the Linear SVC (Support Vector Classifier) class from the scikit-learn.svm library. We arrived at this choice after rejecting the regular SVC and SGDC (Stochastic Gradient Descent classifier) as they gave us inadequate results.

The first step was to create a classifier using some parameters for the SVC class. The ones we focused on in particular were C (the penalty term), loss (the loss function used), penalty (the penalization norm) and dual a Boolean input to check whether we want to solve for the dual optimization problem. Inputting the feature matrix and our classification data array into our classifier, we were able to build a classification model for the data. Tuning the parameters to give us better results, we ultimately arrived at the optimal values.

The mapper finishes off by emitting the results of the classifier.

Reducer

In this task there was only one reducer, and its purpose was to collect all the results generated by the mappers and add them up in an array. Finally, every element of the array was divided by the number of added-up arrays (i.e. the number of mappers). In other words, the final result is the average of the results produced by the individual mappers.

Our final score of using this method was 0.789967, which is above the Easy Baseline, but below the Hard (0.8).

Extracting Representative Elements From Large Datasets

The goal of this project was to extract the 200 most representative elements from a large image dataset. In other words, what the problem requires is solving K-means on a large dataset, but even using smart initialization techniques (like K-means++) K-means is not fast enough for large datasets. We tried several approaches for solving this problem.

First approach

This method was one we tried, but did not get a good score with.

Mapper

In this approach we did k-means in the mapper using the `MiniBatchKMeans` method from `sklearn`. After having done the k-means with the `.fit()` method, we stored all the 200 centers we found in a matrix. We put an extra element, containing a value of 1, at the end of each center. This acts as the weight of the center.

After getting all the centers, we used the `fit_predict()` method to calculate the closest center to each point. We looped over this list of closest centers, and each time we incremented the weight for the corresponding center element in our matrix with the list of centers.

The mappers final output is the centers with their weights.

Reducer

In the reducer, we did k-means from all the centers we got from the mappers. But in this k-means method we did not use a library method, as we wanted to take the weights into account.

We did the k-means method by having random points as our initial centers. At the end of each point we added one element for total weight. We then went through all of our points and found which center was the closest one. For the center that was closest we added the values from the point multiplied by the points weight to the centers dimensions. For the last element (the total weights) we simply added the weight of the point. After having gone through all of the points like this we divided the values of the centers by the total weight they had. The result from one iteration of the algorithm was these centers (without the element for total weight).

We did not do k-means until it converged to a result, but had a variable that we could change to decide how many times the k-means function would iterate.

Second approach

In this approach we run `MiniBatchKMeans` in the mappers for getting up to 250 clusters in each mapper, then in the reducer we tried two different approaches: in one of them we run `MiniBatchKMeans` again with the clusters we got from the mappers, and in the other approach we selected 200 random clusters from the ones generated by all the mappers.

This approach got us a better result than the previous approach, but was not the one we got our best result with.

Third approach

In this approach we implemented coresets. This was the approach that got us our best result in the submission system.

Mapper

The mapper was the one actually implementing the corsets algorithm. We implemented the method outlined in the lecture slides. In the mapper, our algorithm sent the selected points in the space (the coresets) to the reducer. The number of corsets a mapper created depended on how many points it was creating coresets for. So, with a higher number of points as input the mapper was emitting more coresets to the reducer.

Reducer

We tried a few different approaches in the reducer.

We tried doing k-means in the reducer. The k-means reducer was implemented using the `MiniBatchKMeans` method from the `sklearn` library. It was simply doing `.fit()` on the centers emitted by the mappers. We tried using different parameters and values when creating the `MiniBatchKMeans` solver. Although tweaking the different parameters had an impact on our score, the improvement was minor.

We also tried a reducer that was randomly selecting some of the coresets. To our surprise the corset mapper with the random reducer was actually the one that got us our best score. This implies that the k-means approach for the reducer was not a good choice (as it was beaten by a random one), but even after a lot of effort we unfortunately were unable to improve it.

The final score we got for this problem is 748.623211531, which is above the Easy Baseline but below the Hard.

Explore-Exploit Tradeoffs in Recommender Systems

In this project we were asked to make a policy that explores and exploits among available choices in order to learn user preferences and recommend news articles to users.

We first implemented the UCB1 algorithm as described in the lecture slides. The run time of this algorithm was low, however the results we obtained with it were not very good (i.e. below the Easy Baseline).

We then implemented a LinUCB (disjoint) approach as described in the lecture slides. Although it was slow initially and it was timing out, we managed to make it faster by trying a few different things. In the beginning, we were recalculating the inverses every time we needed them. To speed up our implementation we tried calculating all the inverses in the beginning and storing them in a dictionary, while updating only the inverse of the one matrix that was changed, instead of the inverse of every matrix. Finally, we used the ShermanMorrison formula to calculate the inverses faster.

To get the best result we tried a lot of different alpha values ranging from 0.001 to 0.3. With an alpha value of 0.2 we got our best result, which was 0.059632 and above the Hard Baseline.