Data Mining: Learning from Large Data Sets - Fall Semester 2015

fgmehlin@student.ethz.ch
matteopo@student.ethz.ch
piusv@student.ethz.ch

December 13, 2015

1 Approximate near-duplicate search using Locality Sensitive Hashing

In this project we applied Locality Sensitive Hashing (LSH) to select pair of near-duplicates from a set of videos. The input consists of a long list of lines, where every line contains the video ID and a set of shingles for that video.

1.1 Mapper

In order to implement LSH we need to compute a signature for each video. The calculation of the signature was made using a set of 100 hash functions. Every hash function is of the form:

$$h_i(r) = a_i r + b_i \tag{1}$$

where a_i and b_i are random numbers. Therefore before starting to read the input we computed these two random values for every hash function, i.e. a random matrix of size 100×2.

```
hash_functions = np.random.randint(MAX_INT, size=(HASH_FUNC_NUM, 2))
```

A signature for video v is then computed using the following algorithm:

```
for i in 1:100 do
    signature[i] ← ∞
end
for r in shingle do
    for j in 1:100 do
        hash ← h(r) % MAX_INT
        signature[i] ← min(signature[i], hash)
    end
end
```

MAX_INT is the maximum 16 bits integer. Each signature will be split into 10 segments; therefore every segment is a vector of length 10. Every segment will be then mapped into a bucket; so the number of

buckets is also 10. Hence before reading the input we need to initialize the hash function that maps a segment into a bucket. These hash functions have the following form:

$$h(\mathbf{s}) = \sum_{i=1}^{10} c_i s_i + b \tag{2}$$

The hash functions are initialized using the following command:

```
bucket_hash_fn = np.random.randint(MAX_INT, size=(BUCKET_SIZE + 1))
```

Finally the mapper will output for each line of input 10 key value pairs, where the keys are the output of the above 10 hash functions and the original input line as value.

1.2 Reducer

The reducer computes the real Jaccard Similarity for every pair of videos that hashed to the same value in some bucket and outputs the pair if the similarity is ≥ 0.9, in order to eliminate false positives.

1.3 Participation

We started by approaching the problem individually, so each of us could deeply understand the project and try to solve it. As all the solutions were competent in the sense of the results (score), we decided to merge all the versions into one. Therefore, we all participated at all levels of the project.

2 Large Scale Image Classification

This project consists of training a model for classification of two images, namely : Nature and People. From a set of extracted feature, we applied made use of a Stochastic Gradient Descent (SGD) classifier.

2.1 Mapper

In order to make a justified choice of the several parameters we could tune, we used an estimation of the out-of-sample accuracy of the classification algorithm. To obtain this estimation, every time we run the classifier we randomly split the provided data set into a training set (80% of the dataset) and a smaller test set (20%). This split is made on-line as we stream the data. We then train the classifier on the training set and evaluate it on the test set, so that we obtain an estimate of the out-of-sample accuracy.

The mapper is divided into two subsequent parts :

1. Feature transformation
2. Classification with (SGD)

1.) The method transforms applies the following transformation on the sample features :

- $\widetilde{x_1} = \sqrt{|x|}$
- $\widetilde{x_2} = cosh((\frac{\pi}{2}) * x) - 1$
- $\widetilde{x_3} = sin((\frac{\pi}{2}) * x)$

The output feature is given by the following concatenation: $x_{out} = [1, \widetilde{x_1} + \widetilde{x_2} + \widetilde{x_3}]$. The leading 1 is used as intercept.

NOTE: We also tried to implement Random Fourier Feature transformation but they gave worse results that the aforementioned transformations.

2.) We use a Stochastic Gradient Descent Classifier algorithm with l1 regularizer provided by the sklearn library. We use *hinge* as loss-function which is the default linear SVM.

Finally the mapper will output the tuple (1, feature_weights) for the learned dataset.

2.2 Reducer

The reducer aggregates the weights from the different mappers to produce the final feature weights.

2.3 Participation

We started by approaching the problem individually, so each of us could deeply understand the project and try to solve it. After some time, we defined one of the models as the most competent one and tried to improve it individually.

3 Explore-Exploit tradeoffs in Recommender Systems

In this project we built a recommender system that suggests news articles to users given the history of their clicks. The model we used is Linear UCB and our script exposes two main functions: `recommend` and `update`.
The input consists of a log file where every line contains, among other information, a vector of user features and a list of available articles, where each article is identified by an ID. For every line of this file the script calls the function recommend and, in case a user feedback is available, the model is updated.

3.1 Model

The `LinearModel` class of our script contains a dictionary `stat_dict` where for every article x we store a `LinStat` object. The linear model also has some parameters, such as the dimension `dim` of the user features vectors. The `LinStat` object is simply a wrapper that contains four elements

- `cov`: a matrix M_x of size `dim` $\times$ `dim`
- `inv_cov`: the inverse of M_x, which is stored for cashing purposes
- `b`: a vector b_x of size `dim` also used in the model
- `mean`: a cashed vector $w_x = M_x^{-1} b_x$

3.2 Recommend

The recommendation phase is performed for every line in the log file, which provides the user features z and a set of articles $\mathcal{A}$ which could be recommended. This function calls the method `predict` in the `LinUCB` object. For each of the articles in $\mathcal{A}$ we compute the UCB_x value:

$$\text{UCB}_x = w_x z + \alpha \sqrt{z M_x^{-1} z} \qquad \alpha = 1.0 \tag{3}$$

and then the recommended article is computed as $\arg\max \text{UCB}_x$. The coefficients w_x and M_x^{-1} are retrieved from the `LinearModel` object. Here the use of the cashed version of the inverse of M_x entails a sensible speed-up. In case these items are not in the dictionary already, M_x is initialized to the identity matrix and both b_x and w_x to the zero vector.

3.3 Update

If the recommended item matches the one actually suggested to the user, then the `update` function is called. A variable y is set to 1 in case of a positive feedback (i.e. in case the user clicked on the link) and to -1 otherwise. The objects stored in the `stat_dict` for the article x are updated in the following manner:

- `cov`: $M_x \leftarrow M_x + zz^t$, where z is the user features vector
- `inv_cov`: M_x^{-1}

- `b`: $b \leftarrow b + y * z$
- `mean`: $w_x \leftarrow M_x^{-1} b_x$

Participation

We started by approaching the problem individually, so each of us could deeply understand the project and try to solve it. We could then help each other to face the problems that we encountered individually; the final version merges everyone's effort.