Supervised Papers Classification on Large-Scale High-Dimensional Data with Apache Spark

Leonidas Akritidis, Panayiotis Bozanis, Athanasios Fevgas

Department of Electrical and Computer Engineering

Data Structuring and Engineering Lab

University of Thessaly

The Fourth IEEE International Conference on Big Data Intelligence and Computing (DataCom 2018)

12-15 August 2018, Athens, Greece

Supervised papers classification

- Classify a set of unlabeled articles X into one research field y of a given taxonomy structure Y.
- Supervised learning problem: The algorithm will exploit a given set of articles with known labels.
- Important problem for academic search engines & digital libraries - a robust solution allows:
  - Results refinement by category.
  - Browsing of articles by category.
  - Recommendations of Similar articles.

Large-Scale Dataset

- This work makes use of the *Open Academic Graph* https://www.openacademic.ai/oag
- It contains 167 million publications.
- The articles are classified in 19 primary research areas and thousands of children categories.
- It occupies 300GB in uncompressed form.
- Each paper and its characteristics are represented by a record in JSON format.

Features

- *Keywords*: Special words which are selected by the authors to highlight the contents of their articles. Present in some papers.
- *Title words*: treated as normal keywords after removing the stop words. Present in all papers.
- *Authors history*: The research fields of the each author and the respective frequency for each field.
- *Co-authorship information:* We record the research fields of the each (author, coauthor) pair.
- *Journal history:* The research fields of the each journal and the respective frequency for each field.

The original classification algorithm

- L. Akritidis, P. Bozanis. “A supervised machine learning classification algorithm for research articles”, ACM SAC, pp. 115-120, 2013.
- Given a set of labeled articles, we build a model which correlates the keywords, the authors and the journals with one or more research fields.
- The model contains a dictionary data structure M with the features F of the dataset.

Original algorithm: Training phase

- For each feature f we store:
  - A global frequency value $|X_f|$ (occurrences of f in dataset)
  - A relevance desc. vector (RDV) with components $(y, |X_{f,y}|)$.
  - $|X_{f,y}|$: number of f and y co-occurrences (i.e. how many times y has been correlated with f).

Algorithm 1: Model Training

```
1  initialize dictionary M;
2  for each sample x ∈ T with known label y do
3      extract all features F_x of x;
4      for each feature f ∈ F_x do
5          if M.search(f) == false then
6              M.insert(f);
7              set |X_f| ← 1;
8              M.insertRDV(f,y);
9              set |X_{f,y}| ← 1;
10         else
11             set |X_f| ← |X_f| + 1;
12             if M.searchRDV(f,y) == false then
13                 M.insertRDV(f,y);
14                 set |X_{f,y}| ← 1;
15             else
16                 set |X_{f,y}| ← |X_{f,y}| + 1;
17             end
18         end
19     end
20 end
```

Original algorithm: Test phase

- For each unlabeled article x we extract the features F_x and $\forall f \in F_x$ we search M.
- We retrieve its RDV and we compute a score S_y for each label y in the RDV:

$$S_y = \sum_{f \in F_x} w_f \frac{|X_{f,y}|}{|X_f|}$$

- w_f: The weight of the feature.
- Good accuracy: w_k=0.3, w_a=0.2, w_j=0.5

Algorithm 2: Research Articles Classification

```
1  initialize list of candidate labels Y;
2  for each unlabeled article x ∈ R do
3      extract all features F_x of x;
4      for each feature f ∈ F_x do
5          if M.search(f) == true then
6              for each label y in the RDV of f do
7                  set S_y ← S_y + w_f|X_{f,y}|/|X_f|;
8                  Y.insert(y, S_y);
9              end
10         end
11     end
12     Y.sortByScore();
13     set label of x ← Y[0];
14 end
```

Apache Spark

- Spark is a fault-tolerant parallelization framework.
- In contrast to MapReduce, it has been designed to allow storage of intermediate data in the main memory of the cluster nodes.
- In contrast to MapReduce which forces a linear dataflow, it is based on a *DAG Scheduler* which enhances the job execution performance.
- The core element is its *Resilient Distributed Datasets* (*RDDs*), i.e. fault-tolerant & read-only abstractions for data handling with various persistence levels.

Apache MLlib

- Spark powers a set of libraries including Spark SQL, GraphX, Spark Streaming and MLlib.
- MLlib is Spark's scalable machine learning library.
- It implements a series of classification and regression algorithms.
- We are interested in comparing our model with the multi-class classification algorithms of MLlib:
  - i.e., Logistic Regression, Decision Trees and Random Forests.

The LIBSVM Format

- MLlib algorithms accept their input data in LIBSVM format.
- Each row is represented by a `LabeledPoint`, an abstraction which contains:
  - The label of the data sample (double).
  - A sparse vector of feature-weight pairs (int, double).
- We converted the dataset in LIBSVM format with the aim of comparing our method with the adversary classifiers of MLlib.

Dataset Preprocessing

- Spark offers a powerful SQL-like filtration mechanism.
  - We discard all the unlabeled samples and all the samples with irrelevant labels.
  - We discard all the non-English articles.
  - We convert the dataset to the LIBSVM format by applying the hashing trick.

Algorithm 5: Dataset filtration with SparkSQL (steps 1–3) and conversion to a collection of LIBSVM-formatted labeled points (steps 4–26)

```
1  RDD<Row> OAG ← readJSON(location);
2  RDD<Row> OAG1 ← filter(OAG, FOS ∈ Y);
3  RDD<Row> FilteredOAG ← filter(OAG1, lang="en");
4  RDD<String> ConvertedOAG ← FilteredOAG.map<Row>
5      for each row x ∈ FilteredOAG do
6          initialize a new string container line;
7          y_x ← extract label of x;
8          line.append(y_x);
9          K_x ← extract all keywords and title words from x;
10         for each keyword k ∈ K_x do
11             line.append(hash(k):w_k);
12         end
13         A_x ← extract all authors from x;
14         for each author a ∈ A_x do
15             line.append(hash(a):w_a);
16             for each author a' ∈ A_x after a do
17                 line.append(hash(a+a'):w_a);
18             end
19         end
20         j ← extract publishing journal of x;
21         line.append(hash(j):w_j);
22         return line;
23     end
24 end
25 ConvertedOAG.repartition(8);
26 ConvertedOAG.save(location);
```

Dimensionality Reduction

- After preprocessing, our dataset consisted of about 75 million articles and 83 million features.
- Our method executes normally on this huge feature space. However, MLlib algorithms do not:
  - Out of memory errors
- Therefore, we applied Sparse Random Projection to reduce the dimensionality of the feature space.
  - The built-in dimensionality reduction algorithms of MLlib, SVD and PCA, failed to complete the task.
  - The final projected space included only 4181 features.

Algorithm parallelization on Spark (1)

- The Driver program controls the execution flow of the job.
- The dataset has been converted to LIBSVM format.

Algorithm 3: Driver program

```
1  Function main()
2      Configure and Initialize Spark;
3      set RDD<LabeledPoint> D ← readLIBSVM(location);
4      set RDD<LabeledPoint> [] W ← D.split(N, 1 − N);
5      set RDD<LabeledPoint> T ← W[0];
6      set RDD<LabeledPoint> R ← W[1];
7      Initialize Model M;
8      M.train(T);
9      broadcast M;
10     M.classify(R);
11     Shutdown Spark;
12 end
```

- The `split` method automatically shuffles the dataset and splits it in the training and test sets, based on the parameter N (we set N=0.6).
- After the model M has been trained, it is transmitted to all cluster nodes via a special `broadcast` call.

Algorithm parallelization on Spark (2)

- Our parallel model implementation includes:
  - The features dictionary M.
  - The training (fitting) function.
  - The classification function.
- The training phase is a flatMap function which operates in two phases:

Algorithm 4: Model implementation in Spark

```
1  Model
2      Features Dictionary M
3      Function train(RDD<LabeledPoint> T)
4          List<Tuple(f, y, w_f)> l ← T.flatMap(LabeledPoint)
5              for each row ∈ T do
6                  initialize list λ;
7                  set y ← row.label;
8                  set V ← row.featureVector;
9                  for each entry v ∈ V do
10                     set featureID f ← v.f;
11                     set weight w_f ← v.w_f;
12                     λ.insert(f, y, w_f);
13                 end
14                 return λ;
15             end
16         collect
17         for each tuple (f, y, w_f) ∈ l do
18             if M.search(f) == false then
19                 M.insert(f);
20                 set |X_f| ← 1;
21                 M.insertRDV(f,y);
22                 set |X_f,y| ← 1;
23             else
24                 set |X_f| ← |X_f| + 1;
25                 if M.searchRDV(f,y) == false then
26                     M.insertRDV(f,y);
27                     set |X_f,y| ← 1;
28                 else
29                     set |X_f,y| ← |X_f,y| + 1;
30                 end
31             end
32         end
33     end
```

Algorithm parallelization on Spark (3)

- Stage 1: For each LabeledPoint of the input return a local list λ of (f,y,w_f) tuples.
- Collect all local lists λ and merge them into a list l.
- Stage 2: Traverse the list l and build the model M (insert the features, compute frequencies, build the RDVs).

Algorithm 4: Model implementation in Spark

```
1  Model
2      Features Dictionary M
3      Function train(RDD<LabeledPoint> T)
4          List<Tuple(f, y, w_f)> l ← T.flatMap(LabeledPoint)
5              for each row ∈ T do
6                  initialize list λ;
7                  set y ← row.label;
8                  set V ← row.featureVector;
9                  for each entry v ∈ V do
10                     set featureID f ← v.f;
11                     set weight w_f ← v.w_f;
12                     λ.insert(f, y, w_f);
13                 end
14                 return λ;
15             end
16         collect
17         for each tuple (f, y, w_f) ∈ l do
18             if M.search(f) == false then
19                 M.insert(f);
20                 set |X_f| ← 1;
21                 M.insertRDV(f,y);
22                 set |X_f,y| ← 1;
23             else
24                 set |X_f| ← |X_f| + 1;
25                 if M.searchRDV(f,y) == false then
26                     M.insertRDV(f,y);
27                     set |X_f,y| ← 1;
28                 else
29                     set |X_f,y| ← |X_f,y| + 1;
30                 end
31             end
32         end
33     end
```

Algorithm parallelization on Spark (4)

- Test Phase: For each unlabeled article, establish a list of candidate labels Y.
- For each feature, search M.
- In case it is found, retrieve its RDV and for each entry in the RDV, compute the scores and update Y.
- Sort Y in decreasing score order.
- Assign the highest-scoring label to the article.

```
34  Function classify(RDD<LabeledPoint> R)
35      List<Tuple(y', y)> L ← R.map(LabeledPoint)
36          for each row ∈ R do
37              initialize candidates list C;
38              set y ← row.label;
39              set V ← row.featureVector;
40              for each entry v ∈ V do
41                  set featureID f ← v.f;
42                  set weight w_f ← v.w_f;
43                  if M.search(f) == true then
44                      for each label c in the RDV of f do
45                          set S_c ← S_c + w_f|X_{f,c}|/|X_f|;
46                          C.insert(c, S_c);
47                      end
48                  end
49              end
50              C.sortByScore();
51              set y' ← C[0];
52              return Tuple(y', y);
53          end
54      collect
55  end
56  end
```

Experiments

- We used the cluster of our department: 8 nodes with 16 CPUs and 64GB of RAM each.
- Java 1.8, Hadoop 2.9.0, HDFS, YARN, Spark 2.3.0.
- 15 executors (2 executors per node) with 28 GB of available RAM each.
  - One node runs only 1 executor plus the Application Master.
- Our method: Paper Classifier (PC)
- MLlib adversaries: Logistic Regression, Decision Trees, Random Forests.

Accuracy Measurements

Features	Algorithm	Accuracy
$83.3 \cdot 10^6$	Paper Classifier	79.1
	Logistic Regression	–
	Decision Trees	–
	Random Forests	–
4181	Paper Classifier	52.1
	Logistic Regression	46.9
	Decision Trees	18.6
	Random Forests	24.9

TABLE I
COMPARISON OF THE PAPERS CLASSIFICATION ACCURACY FOR VARIOUS ALGORITHMS

- In the original feature space (83,3M features) only our method managed to complete, achieving accuracy of 79.1%.
- In the reduced feature space, our method lost a portion of its accuracy (52.1%), however, it outperformed all the classifiers of MLlib.

Efficiency Measurements

Algorithm/Method	Duration (min)
Paper Classifier ($83.3 \cdot 10^6$ features)	92
Paper Classifier	17
Logistic Regression	182
Decision Trees	239
Random Forests	263
Dataset preprocessing (Algorithm 5)	51
Singular Value Decomposition	–
Principal Component Analysis	–
Sparse Random Projection	44

TABLE II
COMPARISON OF THE ALGORITHMS EXECUTION TIMES (IN MINUTES)

- Our method was much faster than the algorithms of MLlib. Even in the original feature space (83M features), it was twice as fast as Logistic Regression in the reduced feature space. In the reduced feature space, it was 11 times faster.
- The dashes symbolize the failure of SVD and PCA in reducing the feature space.

Conclusions

- We presented a parallel supervised learning algorithm for classifying research articles on Apache Spark.
- Our method takes into consideration multiple features including keywords, title words, authors, and publishing journals of the articles.
- Our method operates effectively and efficiently on large, high-dimensional datasets.
- It outperforms the built-in Spark MLlib classifiers by a significant margin.