Vehicle Incident Hot Spots Identification: An Approach for Big Data

Isaac Triguero*‡, Grazziela P. Figueredo*†, Mohammad Mesgarpour §, Jonathan M. Garibaldi*† and Robert I. John*‡
* School of Computer Science, The University of Nottingham, NG8 1BB
† The Advanced Data Analysis Centre, The University of Nottingham, NG8 1BB
‡ The Automated Scheduling Optimisation and Planning Research Group, The University of Nottingham, NG8 1BB
§ Microlise, Farrington Way, Eastwood, Nottingham NG16 3AG, UK
Emails: Isaac.Triguero@nottingham.ac.uk, gzf@cs.nott.ac.uk, Mohammad.Mesgarpour@microlise.com, jmg,rij@cs.nott.ac.uk

***Abstract*—In this work we introduce a fast big data approach for road incident hot spot identification using Apache Spark. We implement an existing immuno-inspired mechanism, namely SeleSup, as a series of MapReduce-like operations. SeleSup is composed of a number of iterations that remove data redundancies and result in the detection of areas of high likelihood of vehicles incidents. It has been successfully applied to large datasets, however, as the size of the data increases to millions of instances, its performance drops significantly. Our objective therefore is to re-conceptualise the method for big data. In this paper we present the new implementation, the challenges faced when converting the method for the Apache Spark platform as well as the outcomes obtained. For our experiments we employ a large dataset containing hundreds of thousands of Heavy Good Vehicles incidents, collected via telematics. Results show a significant improvement in performance with no detriment to the accuracy of the method.**

I. Introduction

Transportation research mostly aims at establishing the means for improving driving performance, economy and safety [1]. Logistics complexity coupled with large transport networks has required the widespread use of sensors, tracking devices, and mobile communication equipment in order to enable such developments [2]. These devices constantly gather information of vehicles and their journeys. This includes, for instance, safety hazards, vehicle diagnostics and driving behaviour [3], [4]. Given the velocity by which large volumes of data are produced, the challenge is to establish effective tools for fast processing and analysis so that the information can be employed by transport stake holders in a timely manner. As data availability increases, opportunities and challenges to extract useful information arise. Our interest lies in addressing the problem of Hot Spot Identification (HSID) for traffic incidents and accidents for very large datasets.

Current literature does not effectively address the hot spot problem for big data. Identified gaps include (i) the number of instances investigated is very limited; (ii) experiments are mostly conducted within a small number of routes and journeys and/or considers simulated data; and (iii) current methods show a significant decrease in performance as the dataset size increases. In an attempt to overcome these limitations, Figueredo *et al.* [5] introduces an immuno-inspired algorithm for HSID in roads for larger datasets. The datasets employed include thousands of data points collected across the UK via telematics. Although the solution is effectively implemented for the case studies, it is observed that the approach's processing time increases significantly once millions of data points are considered.

Fortunately, advances in cloud-based technologies provide us with distributed environments to alleviate the computational complexity and the management issues with big datasets. Hadoop MapReduce [6] [7] tackles data-intensive applications and employs a distributed file system, which allows for the parallelisation of multiple jobs across a cluster of computers, in fault-tolerant environment [8]. Its limitations however occur when there is the need to share data across multiple steps. This has therefore become an important restriction to iterative algorithms or interactive queries. To address these limitations, Apache Spark [9] has been introduced and consolidated as a platform to cache data into main memory and query it repeatedly.

In this work, we devise a fast big data approach for HSID implemented using Apache Spark. We formulate the mechanism proposed in [5] as a series of MapReduce-like operations in such a way that our design provides exactly the same behaviour as the original algorithm. Our experiments are conducted with the same datasets as in [5] and results show a considerable improvement in performance, which allows for far more data to be processed. In the next section we introduce the details of the problem, challenges and describe the previous work. Subsequently, we present the basic concepts of the big data technologies employed in this work. In Section IV we introduce the implementation details of our solution. Then, Section V presents experimental evaluation and discusses the results obtained. Finally, we draw the conclusions and opportunities for future research.

II. Problem Description

The problem regards the determination of incident hot spots from large telematics data. Hot spots can be defined as those areas where there is a high likelihood of incident occurrence. The determination of road hot spots assists stake holders to effectively manage risks of danger to drivers and vehicles. Hot spots are generally determined by road experts or via analysis

of historical data. However, as very large datasets are considered, several challenges to establish effective determination of hot spots take place. For illustration purposes, let us consider the scenarios of Figure 1. In the figure, the red circles represent incident instances. The hot spots should indicate the location of groups of points within a distance range that share similar properties (such as location, direction of the road, angle of the road, etc.). Therefore groupings such as those established by the green circles in the figure do not produce valid hot spots. The blue circles, instead are more likely to represent acceptable solutions. The pink circle is unlikely to be a hot spot, as it encompasses only one incident.

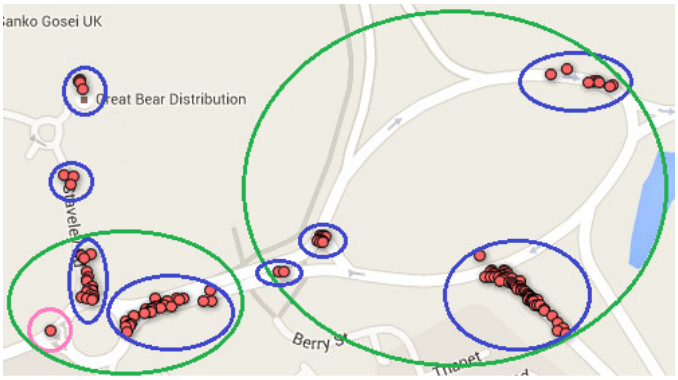


Fig. 1. Examples of possible hot spot clusters. Clusters indicated in blue represent good candidate solutions. Clusters inside pink and green lines represent invalid solutions. Adapted from Figueredo *et al.* [5].

From Figure 1, several factors to establish an effective hot spot detector should be considered. Once a distance range is defined for the hot spot clusters, the solution is required to be accurate and complete; this means that all existing hot spots should be determined. Furthermore, the hot spot cluster centres should indicate with a certain degree of precision those areas where incidents take place frequently. In addition, it is required that the solution is general and adaptable to different types of incidents. In our case studies, for instance, we consider three different types of incidents (speed, harsh braking and harsh corners). These incidents have different characteristics and are likely to occur in distinct parts of the road (e.g., harsh braking and harsh corners are likely to happen in roundabouts or when there is a bend in the road, while speeding incidents occur with higher frequency in motorways). As more incident data is gathered, the method should be robust enough to modify the hot spot topology to a more current scenario, based on new information. The approach should also be noise-tolerant. Preferably, due to the large volumes of data considered and the industrial application, it is desirable that the solution is relatively low in complexity and fast to run.

For our case study, we investigate a large dataset containing 1,000,612 incidents collected for three HGV companies over a three-month period in 2015, across the UK. The data is distributed between incidents, as follows: 773,323 speeding incidents (3139 speed and 770,184 contextual speed); 213,697 harsh braking incidents; 13,592 harsh cornering incidents.

A. Related work on Hot Spot Identification

HSID is mostly addressed by statistical methods assisted by historical data. Government and public [10] also contribute to the knowledge, however it also results in inaccuracies [11]. Two main reviews compare HSID methods. Montella [12] investigates crash frequency, equivalent property damage only crash frequency, crash rate, proportion method, Empirical Bayes (EB) [13], EB estimate of severe-crash frequency (EBs) and the potential for improvement (PFI) method for HSID. In crash rate, locations are sorted by accident frequencies divided by the length of road segments. EDPO ranks accidents according to damage, costs and injuries. Crash rate normalises crash frequency per traffic volume. The proportion method prioritises sites depending on their crash probabilities. In EB, the estimation of the safety of a spot is obtained using the history of crashes and the expected crashes from safety performance functions for similar sites. PFI is the difference between the EB expected accident frequency and a crash prediction model. Montella's case study employs geometric, traffic and 2245 crash records from 2001-2005 for a motorway in Italy. 646 segments are considered. The author's conclude that the EB method is more suitable to detect priority locations. Similarly, Cheng and Washington [14] employ simulation data to evaluate ranking, confidence interval and EB, and also conclude that EB performs better.

Anderson [15] introduces a kernel density estimation (KDE) for HSID with clustering to determine classes of hot spots and their causal indicators. The author assumes that road accidents are influenced by their density in a specific area. The KDE is therefore employed to establish those areas of high risk of incidents and their spread, which are further categorised after clustering. London traffic accident data (1999-2003) is employed. The main limitation of the KDE, as pointed out by the author is that it *treats discrete events as a continuous surface.* This generalisation might incur inaccuracies, as not all adjacent events necessarily belong to the same hot spots. Bíl *et al.* improves the KDE cluster detection to overcome the lack of confidence in the accuracy of the results. The data used consists of 7121 traffic incidents, collected via GPS, in eastern Czech Republic. The analysis is performed on primary roads separated into 713 sections, without intersections. Monte Carlo is incorporated to create repeated random simulations with variations in the incident locations. The work is limited, however, as areas in the map are excluded and the roads are segmented. This technique requires further investigation to assess its applicability to big data.

Effati *et al.* [16] introduces a geospatial neuro-fuzzy approach for HSID. Historical data along with roadway information is used to calibrate and validate the model. Their methodology employs roadway geometry and environmental factors, which are processed through an adaptive neuro-fuzzy inference system. Their case study considers layers of data regarding a highway in the North of Teheran. The correlation between calculated hazardous zones and hot spots obtained is verified; further, additional hazardous zones are spotted.

Although successfully applied to the case study, to the best of our knowledge the method has not yet been exploited for larger datasets. In addition, it requires several layers of information provided by different stakeholders, which makes the generalisation of the method far more laborious.

El-Basyouny and Sayed [17] proposes a depth-based multivariate method using a full Bayes approach. They employ 236 signalized intersections from Vancouver with collision and traffic data (2001-2008). Data is split in 2001-2005 for ranking and 2006-2008 for evaluation. Markov Chains and Monte Carlo are employed to obtain a set of full Bayes posterior estimators on each multivariate Poisson log-normal model. Dangerous intersections are detected after applying a depth threshold, which value depends on funding available for safety improvement. The model is compared to analogous methods based on depths of accident frequency (AF). Results suggest better results. This work is limited however to one dataset and it requires further research in order to assess its applicability to HSID.

From the literature it is possible to identify there is very little research regarding generic methods for HSID with big data. An attempt to tackle large data sets is found in the work introduced by Figueredo *et al.* [5], as further discussed in (Section IV) .

III. Big Data Technologies

MapReduce [7] was originally designed by Google in 2003 as a scalable data processing tool to run on commodity hardware. It has become popular in its open-source implementation, called Apache Hadoop [18]. This platform transparently processes data in a distributed cluster, relieving the user from technical details, such as data partitioning, fault-tolerance and job communication. Hadoop relies on its distributed file system (HDFS) to tackle data-intensive application based on the principle of data locality, which means that the computation is placed near the data. There are tasks however for which Hadoop MapReduce is not the most appropriate solution due to the additional costs required for reusing data. This is the case of interactive queries and online or iterative computing.

Among other platforms such as Haloop or Twister, Apache Spark overcomes the drawbacks of Hadoop when dealing with these tasks. Spark is a data processing framework layer on top of the Hadoop ecosystem; it relies on the HDFS to preserve the data locality principle. A Spark cluster is usually composed of a driver/master node and a number of worker nodes, in which the HDFS systems is setup. The key advantage of Spark comes from a set of in-memory primitives that allows reuse of data multiple times. The most well-known abstraction of Spark is called Resilient Distributed Datasets (RDDs). This is a distributed data structure that allows for computations in parallel in a transparent way. RDDs can be cached in main memory (of multiple nodes involved in the processing), so that it makes easier and more efficient to reuse data. In addition, Spark employs lazy evaluation, which allows the engine to optimise consecutive data transformations without requiring any action from the user. The partitioning of RDDs can also be managed to optimise data placement. Spark is in continuous development, as more efficient APIS, such as DataFrame and Datasets are being designed to further accelerate data processing.

IV. The Hot Spot Identification Approach

SeleSup is inspired by the Immune System self-regulation mechanism, where only the fittest immune cells remain as part of the body defense. It was introduced and further exploited in [19]–[22] for instance selection and data classification. In Figueredo *et al.* [5] the method is adapted to tackle HSID for large datasets. It works by establishing a set of points (suppressor set), which is meant to have the most significant information in the data (in our case it is the set of hot spots). This set size and its data points are initially defined randomly; however, the self-adjustable, self-adapting character of the method allows for the establishment of an optimal number of hot spots, even when new data is acquired over time. The remaining incidents not contained in the suppressor set constitute the set to be reduced, as they represent redundant information. The method is well-suited to the hot spot problem, as for the establishment of the hot spots there is no strong need for exact precision in terms of location. As hot spots represent a region of high likelihood of incidents, rather than a specific location in the road (exact latitude and longitude), the indication of the region is enough to solve the problem. In addition, the simplicity of the method allows for fast processing and quick deliver of results.

SeleSup pseudo-code applied to the hot spot problem is shown in Algorithm 1. Initially, the centres of the hot spot clusters are chosen randomly, according to a predefined size. The candidate centres are part of the suppressor set. In the first loop of the algorithm, candidate centres that are redundant, that is, those belonging to the same hot spot region are removed. Centres belong to the same region when their distance is smaller than the predefined hot spot distance range. In addition, depending on the objectives of the problem, other constraints such as address, direction and angle of the road, day of the week, time of the day can also be incorporated. The second loop associates the remaining incidents (set to be reduced) to the cluster centres, also based on distance. Instances successfully assigned to a hot spot centre are removed, as they are redundant in the system. Those incidents that remain after the second loop in the set to be reduced are then potential hot spots, as they are not close to any existing hot spot centre. The final loop of the algorithm addresses those incident points that were not in the range of a cluster centre and removes possible redundancies within this set. At each elimination stage, the remaining hot spots are associated with a score (fitness value). This value indicates the number of incidents that occurred in a certain hot spot cluster. This allows for hot spot ranking and comparison. In addition the the hot spot stores the date in which the last incident took place for future hot spot topology updating. A more detailed explanation of the algorithm is found in [5].

Algorithm 1: The SeleSup for Hot Spot Identification

inputs : Incident data, suppressor set size, mileage range for hot spot

forall *SuppressorCells* **do** *fitness* $= 0$;

STAGE 1

foreach *Suppressor cell* s_i *from SuppressorCells* **do**

  SetOfRedundantSupressors $\leftarrow$ suppressor cells within the similarity range of s_i;

  SuppressorCells $\leftarrow$ *SuppressorCells - SetOfRedundantSuppressors*;

  s_i's fitness $\leftarrow$ size(*SetOfRedundantSuppressors*);

STAGE 2

foreach r_j *from* $CellsToBeEliminated$ **do**

  Nearest suppressor $s_k \leftarrow$ Find the *SuppressorCell* within the similarity range of r_j;

  if *NearestSuppressor is not empty* **then**

    $CellsToBeEliminated \leftarrow CellsToBeEliminated$ - r_j;

    increase s_K fitness;

STAGE 3

forall *CellsToBeEliminated* **do** *fitness* $= 0$;

if $CellsToBeEliminated$ *is not empty* **then**

  foreach r_l *from* $CellsToBeEliminated$ **do**

    RedundantSet $\leftarrow$ other cells from $CellsToBeEliminated$ within the mileage (and constraints) of r_l);

    CellsToBeEliminated $\leftarrow$ *CellsToBeEliminated - RedundantSet*;

    r_l's fitness $\leftarrow$ size(*RedundantSet*);

  $SuppressorCells \leftarrow SuppressorCells +$ $CellsToBeEliminated$;

Eliminate those *SuppressorCells* with *fitness* $= 0$;

Output the set of *surviving SuppressorCells* as the reduced set containing the hot spots locations;

Although the method provided in Algorithm 1 is satisfactory in terms of processing time for hundreds of thousands of data points, as the data sizes approach millions of points, it lacks scalability. The two main problems identified when dealing with large sets of data points are (i) **runtime**: The complexity of SeleSup to identify Hot Spots in a set with N data points is quadratic (O(N^2)), where the Haversine distance has the highest computational cost. When dealing with very big datasets, the performance of the method therefore decays significantly; and **memory consumption**: For a rapid computation of the distances between data points, the SeleSup model requires that all data to be loaded and stored in memory. For big data this means that the available RAM memory limits are easily exceeded. In addition, due to the processing time and memory restrictions, it is harder to conduct a thorough study regarding the stability and consistency of the method. Furthermore, the differences in performance for different sizes of suppressor sets have not yet been assessed. To overcome these limitations, we therefore formulate the SeleSup HSID method as Spark operations. The reasoning behind adopting Spark as big data technology comes from the multiple iterations that SeleSup needs to perform through the data, which as we stated before fits better with Spark rather than Hadoop MapReduce. The details of the implementation, as well as the results obtained are discussed next.

A. The Spark-based Approach Adaptation

SeleSup has three stages for redundancy removal and determination of the hot spots. From a data perspective, stages 1 and 3 (Algorithm 1) remove redundancies within the same data subsets; Stage 2 eliminates redundancies of a larger set by comparing it against a set of Suppressors. Algorithm 2 shows the pseudo-code for the SeleSup Spark version. It focuses on the required Spark operations for the three stages. We describe the most significant instructions, which are enumerated from 1 to 11. The source code of this algorithm is available at GitHub[1].

Algorithm 2: SeleSup Spark-based Hot Spot Identification

Require: IncidentData; #Maps; #SuppressorSetSize; #MileageRange

1: incidentsRDD $\leftarrow$ textFile(IncidentData, #Maps).zipWithIndex().cache()

2: *SuppressorsCells* $\leftarrow$ incidentsRDD.takeSample(false, SuppressorSetSize)

3: CellsToBeEliminated $\leftarrow$ incidentsRDD.filterNot(line $\rightarrow$ SuppressorsCells.Indexes.contains(line.Index))

  forall *SuppressorCells* **do** $fitness_i = 0$;

  STAGE 1

4: RemoveRedundancies(SuppressorsCells) {Updating fitness accordingly}

  STAGE 2

5: SuppressorsCells_BC $\leftarrow$ sc.broadcast(SuppressorsCells)

6: $< CellsToBeEliminated, fitnessStage2 >\leftarrow$ CellsToBeEliminated.mapPartitions(dataset $\Rightarrow$ EliminateAndFitness(dataset,SuppressorsCells_BC, MileageRange))

7: fitness = fitness + fitnessStage2.aggregate()

  STAGE 3a

8: $< CellsToBeEliminated, fitnessStage3a >=$ CellsToBeEliminated.mapPartitions(dataset $\Rightarrow$ RemoveRedundacies(dataset))

9: fitnessCellsToBeEliminated =fitnessStage3a.aggregate() **STAGE 3b**

10: **for** $i = 0$ **to** $\#Maps$ **do**

11: $BC - Map_i \leftarrow$ broadcast($CellsToBeEliminated$.getSplit(i))

12: CellsToBeEliminated $\leftarrow$ $CellsToBeEliminated$.mapPartition(dataset $\rightarrow$ CleaningFromPartion1toN(dataset, $BC - Map_i$, MileageRange,i))

13: **end for**

In the algorithm above, let $IncidentData$ be the dataset of incidents stored in the HDFS as a single file. This file is composed of h HDFS blocks that can be examined from any computing node. The algorithm starts off reading the entire *IncidentData* set from HDFS as an RDD, denoted as *incidentsRDD*, splitting the dataset into an user-defined number of *#Map* disjoint partitions (Instruction 1). This operation spreads the data across the computing nodes, caching the different subsets (*Map*$_1$,*Map*$_2$,...,*Map*$_m$) into main memory. For simplicity, *incidentsRDD* is zipped with indexes (using *zipWithIndex()* operation. Subsequently, we need to extract a random subset from $IncidentData$ to create the initial *SuppressorCells* set. This is done according to a user-defined parameter *#SuppressorSetSize*. We assume that this is a small number, so that the SuppressorCells set can be stored in the driver node without exceeding memory contritions. As we show in our experiments (Section V), there is no need to establish large values for this parameter, as the outcome of the algorithm remains fairly similar.

The Spark action '*takeSample*' randomly subsamples the data and sends it to the driver node (Instruction 2). Once the initial set of *SuppressorCells* is defined, we need to remove those selected data points from the original *incidentsRDD* to obtain a set of *CellsToBeEliminated*. To do so, we use a filter transformation for removal according to their indexes (Instruction 3). This a MapReduce-like operation that is run in parallel and does not move any information back to the driver node.

[1]https://github.com/triguero/Immune-HotSpot

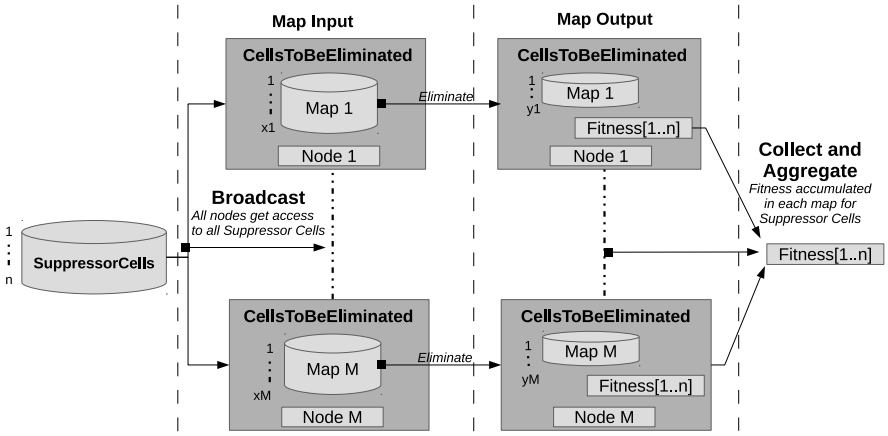


Fig. 2. Stage 2: Eliminating redundant points in *CellsToBeEliminated* w.r.t. the *SuppressorCells* and compute the resulting fitness

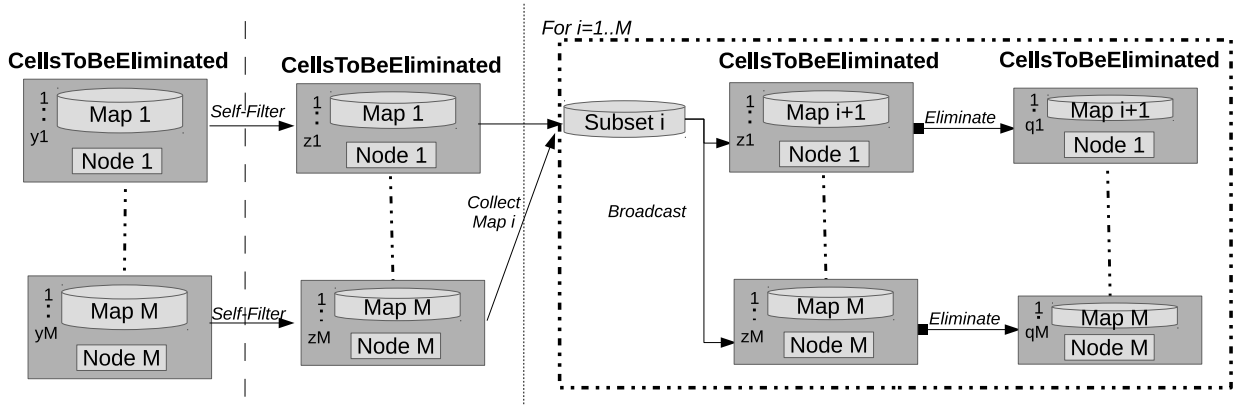


Fig. 3. Stage 3: An iterative Spark approach to remove redundancies from the same dataset: *CellsToBeEliminated*. Note that in the plot $y > z > q$.

As detailed in the previous section, the first stage of this algorithm consists of removing those candidate centres that are redundant. In our implementation, we run this operation in a sequential manner in the driver node rather than in parallel. This set should be sufficiently small to be processed quickly in the driver, without the need for parallelisation. This operation however is the same as that performed in Stage 3 (implemented in parallel), and therefore could be parallelised if necessary.

Algorithm 3: Map function to clean CellsToBeRemoved and update fitness of SuppressorCells

Require: $CellsToBeEliminated_i$; #Miles; *SuppressorCells* (Broadcast)
  forall *SuppressorCells* **do** *fitness* $= 0$;
  foreach r_j *from* $CellsToBeEliminated_i$ **do**
    Nearest suppressor $s_k \leftarrow$ Find the *SuppressorCells* within the similarity range of r_j;
    if *NearestSuppressor is not empty* **then**
      $CellsToBeEliminated_i \leftarrow CellsToBeEliminated_i$ - r_j;
      increase s_K fitness;
  return $< CellsToBeEliminated_i, fitness >$

Stage 2 begins when we have both sets *SuppressorCells* (in the driver node) and *CellsToBeEliminated* (spread across the cluster of nodes as an RDD) ready; we have to find those data points in *CellsToBeEliminated* that are suppressed by the elements in *SuppressorCells*. In order to accomplish this task, the SuppressorCells set needs to be available in all the nodes involved in this computation. For this reason, we broadcast the *SuppressorCells* set to all the computing nodes. The *broadcast* function of Spark allows us to maintain a read-only variable cached on the main memory of each machine rather than copying this within each task. Note that this set of data points is stored only once in each node independently of the number of tasks executed in it. After that, a map phase starts over the #Map partitions of *CellsToBeEliminated* to filter each of these partitions out. The mapPartitions() transformation runs the function defined in Algorithm 3 on each block of the *CellsToBeEliminated* RDD, concurrently. This operation does not only filter redundant data points from *CellsToBeEliminated* but it also computes the fitness of *SuppressorCells* (i.e. number of data points that a given cell from SuppressorCells has suppressed) in each map partition. This means that the fitness obtained in each map needs to be collected in the driver and aggregated appropriately with the fitness achieved in the first stage (See Instruction 7). It is important to point out that at this stage, the *CellsToBeEliminated* set continues to be distributed in the different nodes. Figure 2 shows a flowchart with the parallelisation steps of this stage.

In the last stage the remaining elements of *CellsToBeEliminated* need cleaning. We explore two alternatives in our experiments. For the first alternative, assuming that the number of remaining elements in *CellsToBeEliminated* could be considerably small at this stage, we collect the data from the worker nodes to the driver node and apply the removal of redundant elements in a sequential fashion (as performed in stage 1). Secondly, as an option for those cases in which at this stage the set of *CellsToBeEliminated* is still large, we consider the following two parallel steps (Instructions 8-13). First, taking advantage that *CellsToBeEliminated* is parallelised, we clean each single partition individually (Instruction 8). This is not the same process originally defined in Algorithm 1; however, it has the exact same behaviour. We carry out this process by means of a map phase that filters those data points that are redundant within the same map partition. This step also carries the fitness associated to the remaining *CellsToBeEliminated*. For the rest of elements in *CellsToBeEliminated*, we compare the redundancy between the data of the different partitions. That implies moving data around and we want to minimise the impact of this operation. We propose therefore to iteratively collect back to the driver a partition of *CellsToBeEliminated* (from Map_1 to Map_m) and to broadcast it. Then we compare such partition (Map_i) against the remaining partitions ($\forall j, j > i$) (instructions 10-13). We do not need to compare all partitions against each other; instead, we compare the current partition i to the next partitions (respecting the order). The point here is that after every iteration of this loop, the entire *CellsToBeEliminated* become smaller. To perform this stage properly, we have to consider the fitness of the elements of *CellsToBeEliminated*. Figure 3 summarises the main parallel operations applied to *CellsToBeEliminated*. Next we investigate the performance of both options.

V. Experiments and Results

We employ our method to four real-world datasets of speeding, harsh cornering, harsh braking and contextual speeding incidents, ranging from three thousand incidents to more than seven hundred thousand data points. The data refers to three months of incidents collected via telematics. All datasets contain the same attributes (latitude, longitude, course, address). The mileage limit ranges for the hot spots clusters definition are set to 0.5, 2 and 5 miles for speeding and contextual speeding incidents; and 0.1, 0.2 and 0.5 mileage limit for harsh braking and harsh cornering incidents. We have ran the algorithm considering mileage limit, course and address as constraints for the hot spot definition.

The experiments have been carried out in a single node with an Intel(R) Xeon(R) CPU E5-1650 v4 processor (12 cores) at 3.60GHz, and 64 GB of RAM. In terms of software, we have used the Cloudera's open-source Apache Hadoop distribution (Hadoop 2.6.0-cdh5.4.2) and Spark 1.6.2. In our experiments, we have set a total number of 8 concurrent tasks.

First, we compare the SeleSup implemented in [5] against our Spark-based solution (including parallelisation of Stage 3). Note that our aim is not to establish an exhaustive runtime comparison between the implementation from [5] (in Matlab) and our model (in Scala and Spark). Instead, we want to show the significant difference in time performance achieved with the proposed scheme. The datasets, number of hot spots (reduction set size) obtained and runtime are shown in Table I. In the table we also show the number of hot spots with fitness higher than zero, and those with fitness higher than 10. Figure 4 shows the hot spots identified by both SeleSup (Figure 4(a)) and SeleSup Spark (Figure 4(b)) for harsh braking with half a mile range around Cambridgeshire (UK).

As we have developed an exact parallelisation of the original SeleSup algorithm, the results of the parallel and sequential versions are expected to be very similar. Nevertheless, as the

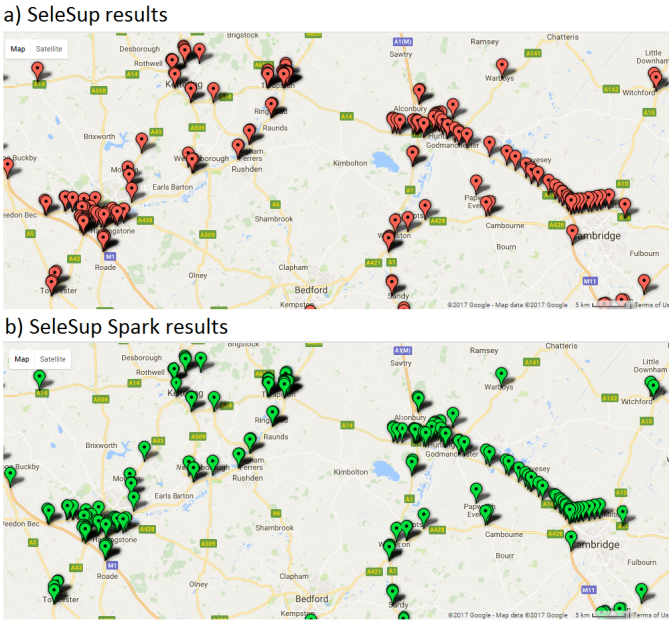


Fig. 4. Results comparison for SeleSup SeleSup Spark implementations. Figure (a) shows the results for SeleSup (red markers) and Figure (b) shows the results for SeleSup Spark (green markers).

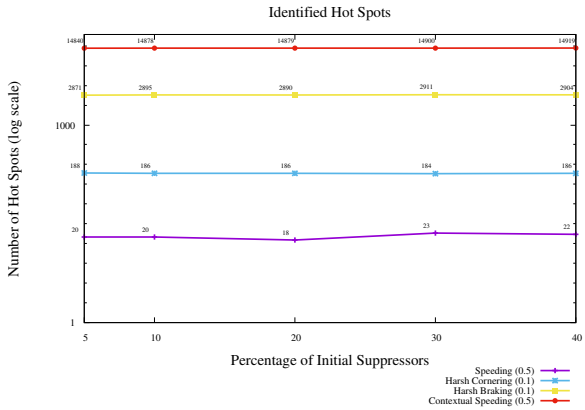
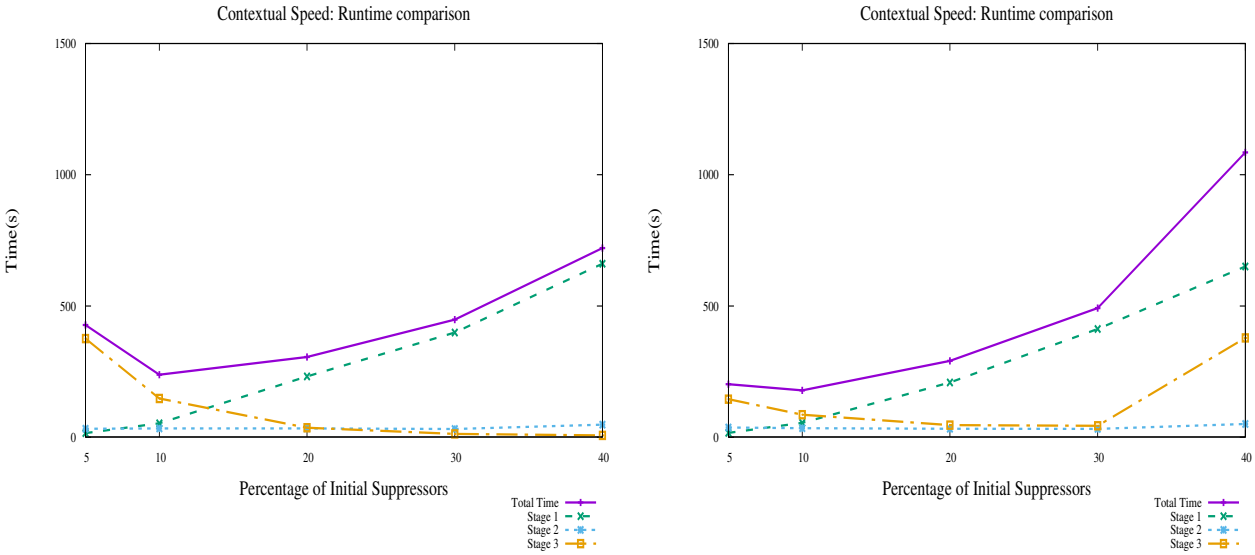


Fig. 5. Number of Hot Spots identified (Fitness>10) w.r.t the initial percentage of randomly selected suppressors. For each dataset, mileage limit is indicated between brackets.

TABLE I
COMPARATIVE RESULTS BETWEEN THE ORIGINAL IMPLEMENTATION IN MATLAB AND THE FULLY PARALLEL VERSION DESIGNED IN SPARK. NOTE THAT TIME IS EXPRESSED AS HH:MM:SS

Data set	**Incidents**	**Mileage**	**Methods' performance**					
			SeleSup HSID			**SeleSup Spark HSID**		
			Fitness > 0	**Fitness >10**	**Time**	**Fitness > 0**	**Fitness >10**	**Time**
speeding	3139	0.5	615	18	00:03:32	620	20	00:00:03
		2	513	32	00:01:44	517	34	00:00:03
		5	385	62	00:01:04	391	60	00:00:03
harsh cornering	13568	0.1	1939	191	00:38:10	1943	186	00:00:04
		0.2	1937	202	00:33:34	1940	203	00:00:04
		0.5	1903	211	00:31:13	1897	213	00:00:04
harsh braking	213697	0.1	30516	2897	08:34:45	30497	2895	00:02:30
		0.2	29417	3244	07:02:44	29463	3228	00:01:47
		0.5	26402	3704	05:06:24	26509	3655	00:01:03
contextual speeding	770184	2	23209	9998	20:38:47	23323	10002	00:01:07
		5	16718	7550	13:45:51	16739	7556	00:00:55



(a) Version 1: Stage 3 in Driver node

(b) Version 2: Stages 2 and 3 in parallel

Fig. 6. Contextual Speed dataset: Runtime comparison across different initial SuppressorCells sizes

initial selection of the suppressor set is random, the number of hot spots and their location can vary slightly. As the mileage range increases, the number of hot spots reduces as each hot spot represents a bigger region in the same road, address and direction.

In terms of runtime, we can observe that the required time of the sequential version with more than tens of thousands of data points becomes prohibitive. Our new Spark proposed solution is able to cope with even the largest dataset within a reasonable amount of time.

In addition, the parallel design allows us to carry out a more thorough study of the behaviour of the algorithm in these datasets. We therefore asses the SeleSup HSID in terms of stability and consistency. The only parameter associated to SeleSup is the initial number of randomly selected suppressors (*#SuppressorSetSize*). In Figure 5, we plot the number of identified hot spots (Fitness>10) according to different percentages of initial suppressors (5, 10, 20, 30 and 40%). We can observe that the algorithm is quite stable in the number of resulting identified hot spots, independently of the initial number of selected data points.

We also explore the performance of the different stages that are considered within SeleSup. To achieve this, we focus on the largest dataset (Contextual speeding), which is the most time consuming. Figure 6 depicts a comparison between the two different versions of our algorithm, including parallelisation of stage 3 (denoted as fully parallel, Figure 6b) and sequential stage 3 (named as partially parallel, Figure 6a). In the figures, we present the runtime required for the three main stages of the algorithm and the total runtime. From the plots we conclude that:

- As we decided to keep a sequential version of Stage 1, a very high number of initial suppressors clearly affects the performance, becoming the most time consuming

operation. However, as noticed before, the outcome of the algorithm is independent of the number of initial suppressors. Our recommendation is to keep this stage sequential with a low percentage of initial suppressors.

- Despite being one of the most important and time-consuming stages of SeleSup, the parallelisation of Stage 2 has turned out to be very efficient in comparison to the other stages.
- For this large dataset in particular, the parallelisation of Stage 3 has resulted in a good reduction of the required time (under 180 seconds). However, as the number of initial suppresors increases the number of elements in the last stage is so small that the parallelisation is slower than the sequential version.

VI. Conclusions

In this paper we have developed a big data approach for identifying vehicle incident hot spots on roads. We have designed a fast immune-inspired mechanism to detect road incident hot spots, using MapReduce as a programming paradigm and Spark as big data technology. The results have proven the effectiveness, stability and efficiency of the proposed algorithm. As future work, we consider the evaluation of even larger datasets, and the development of an appropriate approach to tackle data streams.

References

[1] J. D. Davey, J. E. Freeman, D. E. Wishart, and B. D. Rowland, "Developing and implementing fleet safety interventions to reduce harm: Where to from here?" in *International Symposium on Safety Science and Technology*, Beijing, China, 2008.

[2] M. Mesgarpour, D. Landa-Silva, and I. Dickinson, "Overview of telematics-based prognostics and health management systems for commercial vehicles," *Activities of Transport Telematics*, vol. 395, pp. 123–130, 2013.

[3] J. R. Edwards, J. Davey, and K. A. Armstrong, "Profiling contextual factors which influence safety in heavy vehicle industries," *Accident Analysis & Prevention*, vol. 73, pp. 340–350, 2014.

[4] G. P. Figueredo, P. R. Quinlan, M. Mesgarpour, J. M. Garibaldi, and R. I. John, "A data analysis framework to rank hgv drivers," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE, 2015, pp. 2001–2006.

[5] G. P. Figueredo, M. Mesgarpour, A. M. Guerra, J. M. Garibaldi, and R. I. John, "Detecting danger in roads: An immune-inspired technique to identify heavy good vehicles incident hot spots," *Submitted to: IEEE Transactions on Emmerging Topics in Computational Intelligence*, vol. tbc, no. tbc, p. tbc, tbc.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03, 2003, pp. 29–43.

[7] J. Dean and S. Ghemawat, "Map reduce: A flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[8] A. Fernández, S. Río, V. López, A. Bawakid, M. del Jesus, J. Benítez, and F. Herrera, "Big data with cloud computing: An insight on the computing environment, mapreduce and programming frameworks," *WIREs Data Mining and Knowledge Discovery*, vol. 4, no. 5, pp. 380–409, 2014.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 1–14.

[10] W. Kowtanapanich, Y. Tanaboriboon, and W. Chadbunchachai, "Applying publich participation approach to black spot identification processs: a case study in thailand," {*IATSS*} *Research*, vol. 30, no. 1, pp. 73 – 85, 2006.

[11] B. P. Loo, "Validating crash locations for quantitative spatial analysis: A GIS-based approach," *Accident Analysis & Prevention*, vol. 38, no. 5, pp. 879 – 886, 2006.

[12] A. Montella, "A comparative analysis of hotspot identification methods," *Accident Analysis & Prevention*, vol. 42, no. 2, pp. 571 – 581, 2010.

[13] H. Robbins, "An empirical bayes approach to statistics," in *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. Berkeley, Calif.: University of California Press, 1956, pp. 157–163.

[14] W. Cheng and S. P. Washington, "Experimental evaluation of hotspot identification methods," *Accident Analysis & Prevention*, vol. 37, no. 5, pp. 870 – 881, 2005.

[15] T. K. Anderson, "Kernel density estimation and k-means clustering to profile road accident hotspots," *Accident Analysis & Prevention*, vol. 41, no. 3, pp. 359 – 364, 2009.

[16] M. Effati, M. A. Rajabi, F. Samadzadegan, and S. Shabani, "A geospatial based neuro-fuzzy modeling for regional transportation corridors hazardous zones identification," *International Journal of Civil Engineering*, vol. 12, 2014.

[17] K. El-Basyouny and T. Sayed, "Depth-based hotspot identification and multivariate ranking using the full bayes approach," *Accident Analysis & Prevention*, vol. 50, pp. 1082 – 1089, 2013.

[18] A. H. Project, "Apache hadoop," 2017. [Online]. Available: http://hadoop.apache.org/

[19] G. P. Figueredo, N. F. F. Ebecken, and H. J. C. Barbosa, "The supraic algorithm: A suppression immune based mechanism to find a representative training set in data classification tasks," in *ICARIS*, ser. Lecture Notes in Computer Science, vol. 4628. Springer, 2007, pp. 59–70.

[20] ——, "An immune-inspired sampling technique for data selection," in *The XXX Iberian Latin American Congress on Computational Methods in Engineering (CILAMCE 2009)*, 2009.

[21] G. P. Figueredo, N. F. F. Ebecken, D. A. Augusto, and H. J. C. Barbosa, "An immune-inspired instance selection mechanism for supervised classification," *Memetic Computing*, vol. 4, pp. 135–147, 2012.

[22] M. L. C. Passini, K. B. Estbanez, G. P. Figueredo, and N. F. F. Ebecken, "A strategy for training set selection in text classification problems," *International Journal of Advanced Computer Science and Applications*, vol. 4, no. 6, pp. 54–60, 2013.