Unsupervised Anomaly Detection using *H2O.ai*

Peter Schrott
Berlin Institute of Technology
peter.schrott@campus.tu-berlin.de

Julian Voelkel
Berlin Institute of Technology
voelkel@campus.tu-berlin.de

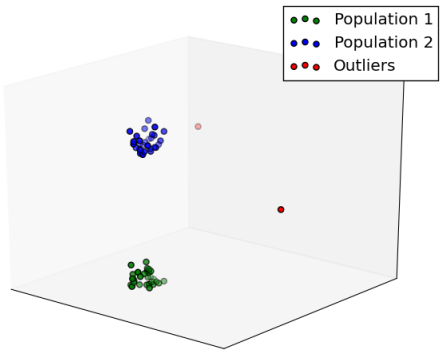


Figure 1: In this simple example, we can see two outliers within one dataset with two populations.

1. INTRODUCTION

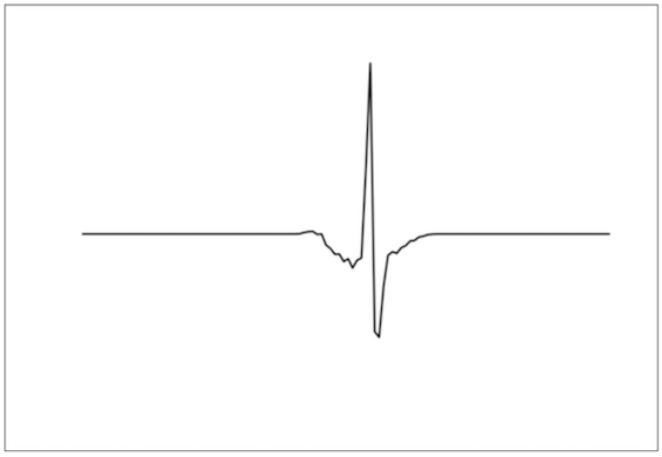
Anomaly detection (commonly referred to as *outlier detection*) is one of a few very common tasks in the field of Machine Learning. The goal of algorithms designed for the purpose of anomaly detection are concerned with finding data in a dataset that does not conform to a pattern. That means, the goal is to identify data points, that are special in regards to their behavior, compared to the data points in the dataset, that are considered "normal". [**?**] These data points are called outliers, because they show different behavior than one would expect. Figure 1 shows a basic plot containing data points, with two different populations, along with two outliers, that do not seem to fit either pattern.

The value of identifying outliers in a dataset lies in the action one can take after detecting them. Common applications of anomaly detection algorithms include among others health care and fraud detection. In the former, those algorithms can for instance help identifying sick patients, by identifying anomalous vital signs compared in a group of similar patients. In the latter application, those algorithms can account for fast, actionable information in case of credit card fraud, which can be identified by anomalous purchases, given the owners purchase pattern in form of historical purchases.

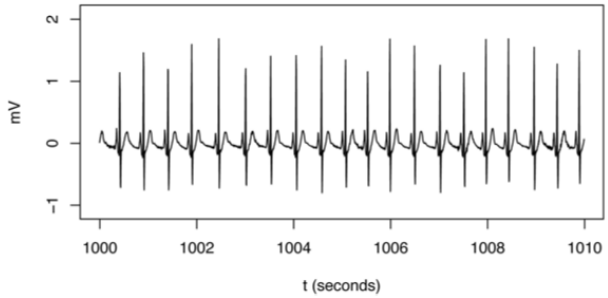
Anomaly detection can happen in a supervised, semi-supervised, as well as in an unsupervised fashion. The credit card fraud detection would be a semi-supervised learning task, since we can assume, that a new credit card will not be the subject of fraud for at least the first couple of purchases. Hence, we obtain a training set of "normal" purchases (i.e. data point) and can for each newly generated data point decide, whether it conforms to the pattern or does not. Since our project is focused exclusively on the unsupervised case where we do not know which data points are considered normal, but rather have to find a structure or pattern in the data first, in order to then be able to identify data points not conforming to the pattern, the next section will focus on unsupervised anomaly detection and the challenges we face in its context.

1.1 Challenges of Unsupervised Anomaly Detection

One difficulty that arises in unsupervised anomaly detection is, since we do not have any labels for training data, we do not even know what we are looking for. That is, we do not know what a "normal" data point would look like, let alone what an anomalous point would look like. To put it in Ted Dunning's and Ellen Friedman's words: "Anomaly detection is about finding what you don't know to look for." [**?**] Since there is no labeled training data in the most widely applicable case of unsupervised anomaly detection, the approach of finding outliers is a different one compared to training a model and then predicting to which class an unseen data point belongs to (much like binary classification). Instead, in case of unsupervised anomaly detection, we generally assume that the number of "normal" data points exceeds the number of anomalous data points by far.[**?**] This assumption is fundamental to unsupervised outlier detection, since we would not be able to learn what is normal otherwise, as a relatively large number of anomalous points would change the skew the structure of the data in a way, that would make determining what is "normal" impossible . Not being able to determine what "normal" is, means there is no way of finding what is anomalous. Since the goal of anomaly detection is finding what is anomalous, unsupervised anomaly detection usually starts with figuring out what "normal" is.



(a) Obvious outlier in small sample size...



(b) ...do not have to be outliers in the context of the whole dataset

Figure 2: Contextual anomaly

After achieving this (which, oftentimes, is much harder than it sounds), we can determine the deviation of a data point to what is "normal" using some similarity measure. There are, however, different algorithms dealing with the problem of unsupervised anomaly detection for different fields and problem domains. The main reason why there is no single approach applicable to each problem is, that there are tremendous differences in what is considered normal and what is considered anomalous, depending on the application domain we are looking at. Considering an example for this circumstance given in [**?**], one can easily imagine why that is the case: "The exact notion of an anomaly is different for different application domains. For example, in the medical domain a small deviation from normal (e.g., fluctuations in body temperature) might be an anomaly, while similar deviation in the stock market domain (e.g., fluctuations in the value of a stock) might be considered as normal. Thus applying a technique developed in one domain to another is not straightforward." Besides the domain specificity of anomalies, there is also the context specificity of anomalies to keep in mind. This concept is illustrated in Figure 2, taken from [**?**]

Common techniques and algorithms used to perform unsupervised anomaly detection include clustering based methods, statistical techniques, information theoretic methods as well as spectral techniques. Note, however, that the choice of algorithm can depend heavily on the problem's domain.

1.2 Deep Learning Auto-Encoder

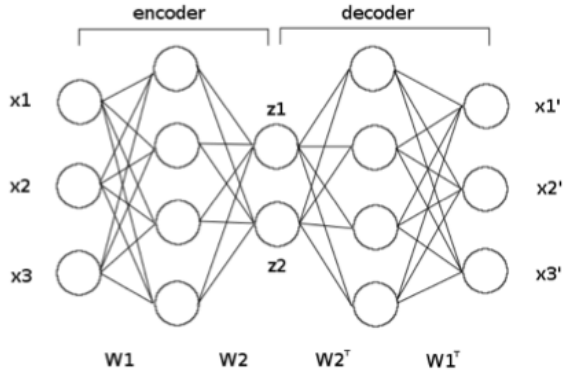


Figure 3: Schematic diagram of a basic auto-encoder with three input features

An auto-encoder, autoassociator or Diablo network is a specific type of artificial neural network. The goal of a deep learning auto-encoder is to learn a compressed encoding of a dataset. Due to that purpose, the auto-encoder consists of one input layer, one or more hidden layers and an output layer with equally as many neurons (i.e. features) as the input layer. In order to achieve the goal of representing a dataset in a compressed manner, the auto-encoder is given the original dataset as input, while the target output is the input itself. [**?**] The loss function is some type of dissimilarity function (typically a squared error function) between the input and the output of the auto-encoder. This way, the auto-encoder is forced to learn a nonlinear (or linear), compressed representation of the original dataset. This, of course, makes the auto-encoder a useful tool for dimensionality reduction. For the special case where there is only one linear hidden layer with k neurons and the mean squared error criterion is used to train the auto-encoder, the hidden layer consisting of the k neurons learns to represent the dataset in the dimension of its first k principal components. [**?**] This is much like Principal Component Analysis (PCA). If, however, the hidden layer is of nonlinear nature, then the auto-encoder behaves very different compared to PCA. [**?**]. Due to its ability to learn a compressed version of the dataset, the main application of the deep learning auto-encoder is obviously dimensionality reduction. In our case, though, we want to use the deep learning auto-encoder in order to perform unsupervised anomaly detection.
A schematic diagram of an auto-encoder taken from [**?**] is given in Figure 3.

2. PROBLEM STATEMENT

As already mentioned in **1. Introduction**, anomaly detection refers to the task of identifying observations, that do not match the general pattern of the data set they arise in. Oftentimes anomaly detection happens in an unsupervised context, which means that the dataset being operated on is unlabeled, and the goal is to identify exactly those samples, that fit the pattern of the dataset the least. This is also the case we want to investigate regarding the usability of a certain algorithm originally designed for a different purpose. Within the scope of this project, we analyze the performance of a particular algorithm more commonly used

in a field different to the one of unsupervised anomaly detection. Specifically, with this project, we aim at providing an answer or at least hints to the answer of the question: **Is a deep learning auto-encoder** (see section 1.2) **well suited for anomaly detection in an unlabeled dataset?**

2.1 Target

As stated above, target of this project is to evaluate the quality of a deep learning auto-encoder model for the task of identifying anomalies in an unsupervised context. Experiments comparing the performance of the deep learning auto-encoder with the performance of other algorithms in the same context shall indicate whether it is a good idea to use the auto-encoder in the context of unsupervised anomaly detection or not.

2.2 Scope

This project consists of various different steps in order to obtain an answer to the problem specified above. These steps can be outlined as follows:

1. Study an existing implementation of the deep learning auto-encoder model
2. Apply this implementation to a given unlabeled dataset
3. Compare the outcome to already existing outcomes of other algorithms
4. Draw conclusions about the general suitability of the algorithm based on the results produced by the application of its implementation compared to those of other algorithms

3. METHODOLOGY

Within the scope of this project, we use H2O.ai's (see section 3.1) implementation of the deep learning auto-encoder model through its Sparkling Water API on top of Apache Spark. The dataset we use in order to be able to compare our results to those of our peers using different algorithms is the AXA Driver Telematics Analysis dataset (see 3.2), which contains multiple trips by multiple drivers.

3.1 H2O.ai and H2O Deep Learning

H2O by H2O.ai is an open source software project primarily used for fast scalable in-memory machine learning. The product strongly aims for data scientists who work in a distributed manner. The software offers a predictive analytics platform, combining high performance parallel processing with an extensive machine learning library. [**?**] H2O was built on top of Apache Hadoop as well as Apache Spark. As of February 2015, the software has more than 12,000 users and is deployed by more than 2,000 companies, including PayPal, Nielsen and Cisco. [**?**]

H2O is shipped with its own distirbuted in-memory data store which is integrated as an key-value store on top of non-blocking hash maps. The basic type for data sets in H2O is the H2OFrame. An H2OFrame consits of vectors, where each vector represents one row, respectivle one feature. As opposed to the H2OFrame, the vectors are immutable. The access to the data is granted by an R evaluation layer. This layer also builds the bridge to the REST interface. For the computation part the base is build by basic operations as fork, join and map reduce tasks. These are used by the H2O prediction engine. All algorithm, either provided by H2O or custom implementations, use that engine to build their individual model. The data storage and data analysis takes palace in the so called H2O cloud which runs on one or more nodes. For each node a Java Virtual Machine is instanciated and perform the above described operations. The H2O context is accessable through network by interfaces for Python, R, Java, Scala, Tabelau/Excel or the H2O WebUI.

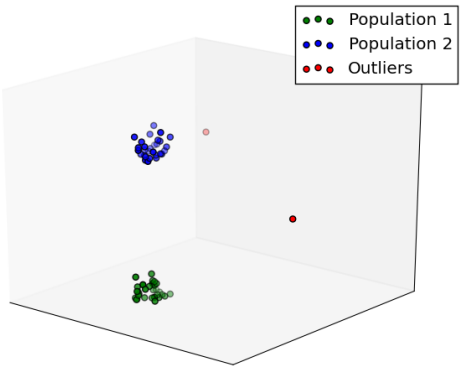
Besides Distributed Random Forests, K-means, Generalized Linear Model and Gradient Boosting Machine, H2O also offers readily available deep learning algorithms, which includes the auto-encoder.
As mentioned in Section 1.2, the deep learning auto-encoder is an algorithm that is used primarily for dimensionality reduction. The way we want to use the auto-encoder to detect outliers in an unsupervised manner is shown in figure 4. As it can be seen there, the algorithm is forced to learn the identity through a non-linear, reduced representation of original data. This is done by first reducing the data's dimensionality and then reconstructing it from that reduced representation. Since one assumption in unsupervised anomaly detection is that the number of normal data points exceeds the number of anomalous ones by far (see section 1.1), that learned model will be mostly influenced more by what is normal in the data than by what is anomalous. Thus, attempting to reconstruct a data point from its reduced representation will have a greater error than anomalous data points that it will have for normal data points.

3.2 The AXA Driver Telematics Analysis Dataset

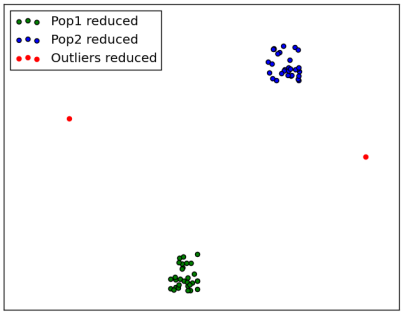
The AXA Driver Telematics dataset is a dataset that was being released to the public in form of a Kaggle challenge [1]
The dataset is a directory based dataset. This means meta data is implicitly contained in the directory and file structure of the dataset. In total there are logs for 2736 drivers. There is one designated folder for each driver, each of which contains 200 different trips in form of CSV files. In the raw data, a single trip is given by a single CSV file consisting of two columns and a varying number of rows. For every single drive, there is one column containing x coordinates one column containing y coordinates. Each row then represents the driver's position one second after the previous row. Every trip has been anonymized, such that each trip starts at position $(x, y) = (0, 0)$ and all the following coordinates have been randomly rotated.
For instance, the first few rows in the CSV file representing driver one's first trip are shown in Table 1

The catch with this dataset is that while there is a folder for each driver with a number of his or her respective trip, there is always a varying and unknown number of trips that were being generated by other drivers (otherwise not represented in the dataset) in that particular folder as well. These unlabeled outliers is what we hope end up identifying using the deep learning auto-encoder.
Figure 5 (taken from kaggle.com) shows what the whole dataset might look like, if we just plotted each driving trip as a line connecting its consecutive (x, y) points.

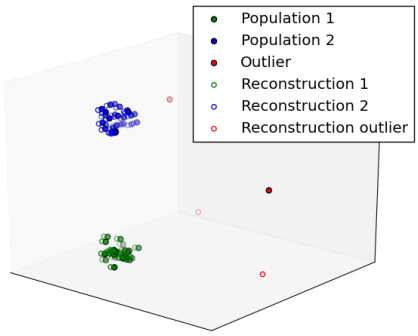
[1]Find the challenge with the dataset here: `https://www.kaggle.com/c/axa-driver-telematics-analysis/data`



(a) The given unlabeled dataset...



(b) ...is reduced in dimensionality...



(c) ...and reconstructed, in order to find outliers

Figure 4: Identifying outliers by reconstruction error

x	y
0	0
18.6	-11.1
36.1	-21.9
53.7	-32.6
.	.
.	.
.	.

Table 1: The first few rows of the driver one's first drive

Figure 5: A visualization of the raw dataset. Each line is generated by connecting the (x, y) coordinates of one particular trip and is thus a visualization of that one particular trip

With a size of 1.44 GB compressed and 5.92 GB in extracted state, this dataset can be considered reasonably large and thus, processing the dataset on a parallel system is justified.
The fact, that AXA, a major car insurance provider (amongst other things), releases a dataset of this kind with the goal of identifying anomalies in driving patterns to the public, hints at a real world application for anomaly detection algorithms that generates value of some kind. In this case, the companies goal might have been to identify or count the times a person not insured for a particular car still drove said car. Identifying those instances might for instance allow challenging of fraudulent insurance claims. Another way AXA might profit from identifying anomalies in drives is, that having an estimate of the numbers of uninsured drives allows for adjustment of internal calculations of revenue, deductions and the like, as well as adjustment of insurance rates. If not one of the above, there has to be some kind of incentive for AXA to be able to identify anomalies.
In our case, however, the fact that more than 1,500 teams already submitted their solutions, allows for some benchmarking of the deep learning auto-encoder.

3.3 Feature Extraction

In order to be able to find the anomalous driving trips for each driver contained in our dataset by applying the H2O deep learning auto-encoder, we have to extract features from our trips first. Ideally, those feature would be meaningful, with large variance in those driving trips that were generated by other drivers. We use Apache Flink in order to extract our features, as illustrated in Figure 6. Since every trip is initially given as a CSV file containing only (x, y) coordinates of that trip (see section 3.2), we implement a feature extraction engine that calculates different features for each driving trip.
Hence the dataset is not provided in the usual, file-only format contributions to the Apache Flink project are made. This was necessary to maintain the information given by the folder / file structure. The name of the folder is considered as the driver ID, the filename as trip ID. Furthermore a custom implementation of an Apache Flink conform *InputFormat* has to be implemented. Simply reading the CSV files would destroy the context of the data. The result of the input pipeline are trips labeled by driver ID and trip ID containing a sequence of (x, y) coordinates. Also important is to preserve the sequence of the (x, y) tuples as the chronological order characterizes the route.
The mined features include for instance driven distance, time of the trip, speed, acceleration, changes of heading. For speed and acceleration the mean, median, deviation of the mean of the drivers mean, standard deviation and maximum are added. The change of heading is used to determine the number of significant turns. This are turns bigger than 35°, 75°and 160°within a window of 10 seconds. Additionally a left turn is considered as a negative value, a right turn as a positive. The intention behind the counts is to characterise the drivers routes. Utilized by the delta speed of each entry in a trip, stops can be computed. This is where the speed is 0. Similar to the changes of heading the length of the stops are provided as further features. Stops longer than 1, 3, 10 and 120 seconds are counted. The shorter stops describe stop-and-go traffic. Traffic light stops are between 10 and 120 seconds. Longer stops can be considered as breaks.

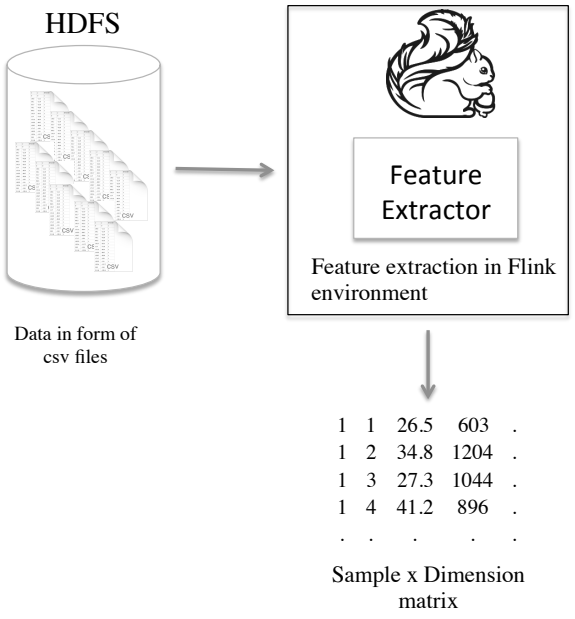


Figure 6: Schematic visualization of our feature extraction

After extracting all the features, we obtain one feature vector for each trip of each driver. The outcome is materialized as CSV file where the rows represent a trip and the columns a feature. The feature vectors are ultimately pass to the auto-encoder as input vectors, are of fundamental meaning for the quality of the anomaly detection. Depending on their significance for a driver's fingerprint, the algorithm might perform well or it might produce unusable results in case the features do not bear a large significance for a driver's pattern of driving.

3.4 Exploratory Data Analysis

In order to get an overview of the data we are dealing with, we perform exploratory data analysis, mainly in the form of visualizations.

Figure 7 gives a overview of the dataset by displaying the duration and distance of every single trip for a few drivers. What we can see here already, is a rather large variance in trip duration among different drivers. Also, there seem to be cases where the driver did barely move, as well as very short trips (short in the sense of time passed during the trip).

Figure 8 provides some insights about driver 18. Since the plot contains all 200 recorded driving trips of that driver, including the drives not originating from driver 18, we can attempt to see some anomalies here. In this case, we can clearly see, only by looking at the driven distance, that a handful drives do not seem to fit the pattern of this particular driver, which seems to be to drive rather short distances. Using the above mentioned features we extracted, exploratory data analysis yields some other insights about variation in driving pattern amongst drivers, as well as some anomalous behavior among driving trips for single drivers as well.

The box plot in Figure 9 for instance shows not only the variation of driving distance among drivers, but also between

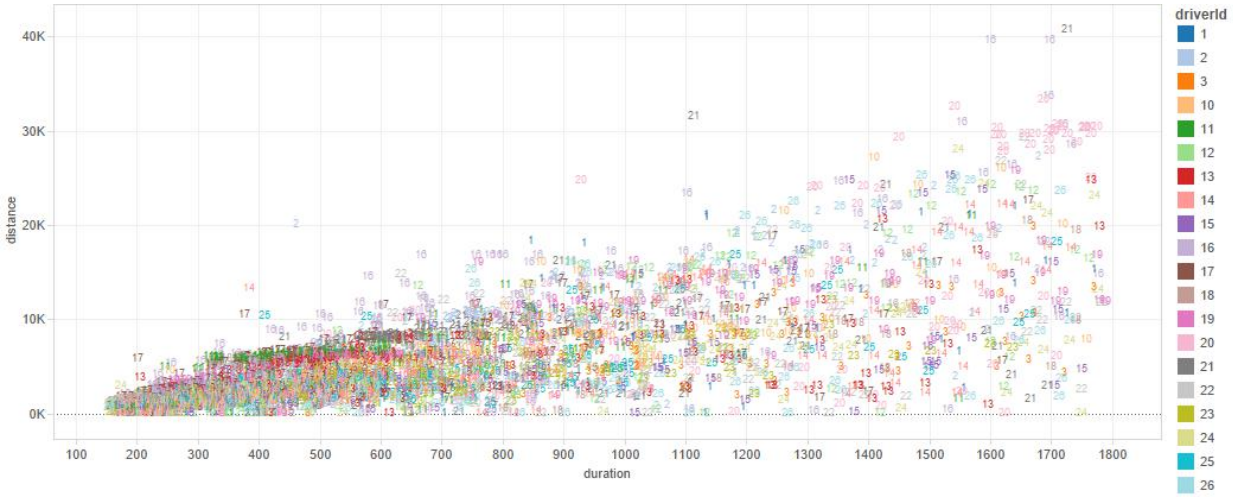


Figure 7: Plot of duration versis distance for each trip by each driver

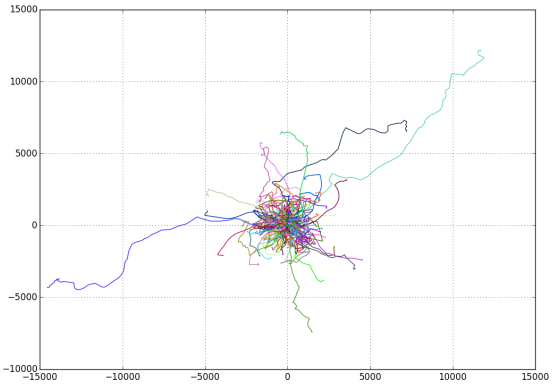


Figure 8: All trips of driver 18 visualized

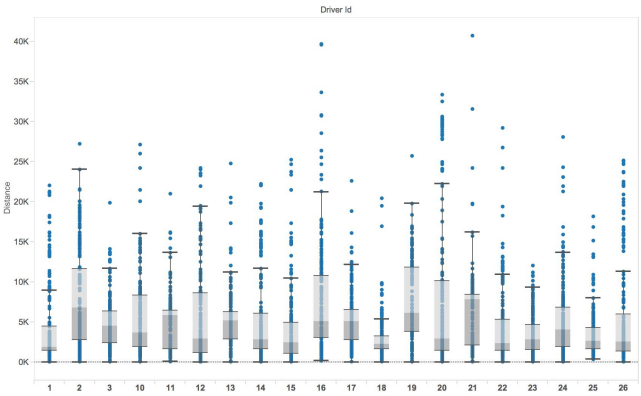


Figure 9: Boxplot of distances for 20 driver

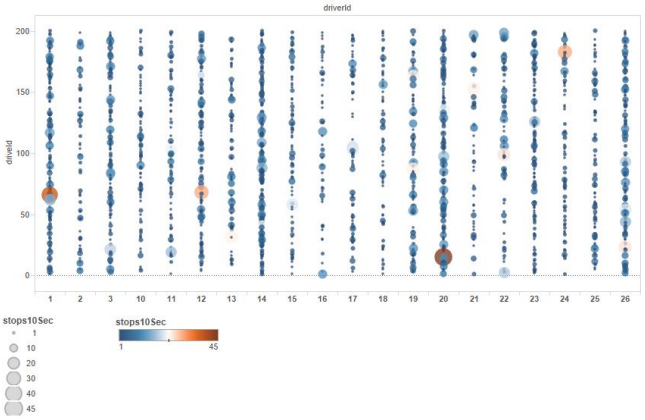


Figure 10: Number of stops with 10 - 120 seconds

different trips for one driver. In particular, this box plot seems to confirm the assumptions made about driver 18's driving pattern after having seen the plot of all that driver's trip in Figure 8. That is, because we can clearly see in the box plot , that driver 18 has a very low variation when it comes to trip length.

Considering the number of stops per driver and their respective driving trip is also helpful in detecting driving patterns. In Figure 10, size and color of the circle visualize the number of stops between 10 seconds and 120 seconds. The insights we gain from this plot can be the revealing of the area the respective driver commonly drives in. A constant high number of short breaks might indicate a driver mainly driving in urban areas. In this case, trips with more infrequent, shorter stops could be considered anomalous because the might originate from a rural driver.

4. EXPERIMENTS

4.1 Experimental Approaches

4.2 Sparkling Implementation

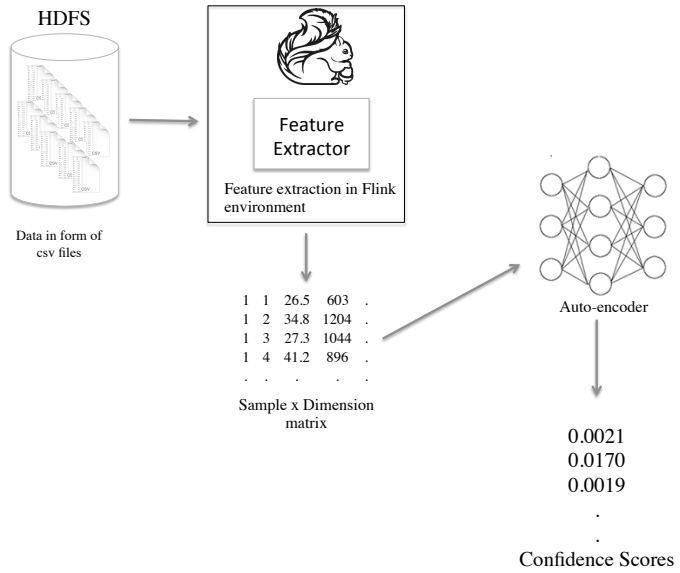


Figure 11: Schematic visualization of our experiment(s)

Parameter	Value
training data	all available trips (2736 x 200)
response column	any non static column
autoencoder	true
activation function	tanh
hidden layers	{ 15 }
epochs	3
L2-normalization	0.01

Table 2: Parameter set for deep learning out-encoder in experiment 1

Parameter	Value
training data	all trips of current driver (200)
response column	any non static column
autoencoder	true
activation function	tanh
hidden layers	{ 15 }
epochs	3
L2-normalization	0.2

Table 3: Parameter set for deep learning out-encoder in experiment 2s

As stated in section 3.1 the scalable machine learning library of H2O.ai builds the core of the experimental implementation. H2O.ai provides a comprehensive API named Sparkling Water for interfacing the H2O cloud. This API is written in Java and Scala and is usually embedded in a data flow of an Apache Spark application.
In the following the implemented data flow and data transformation are explained. The first step is to load the input data from the HDFS on the cluster. A regular Apache Spark RDD is used as container. The first transformation is to convert the line-wise input string to representative Plain Old Java Objects (POJOs). The POJOs are stored in the Apache Spark context on the nodes of the cluster. This is a very common step within scalable data analysis.
Within the next step towards the outlier-detection the RDD containing the input data is transformed to an Apache Spark DataFrame. A DataFrame is the representation of a rational database table in Spark SQL and stored in its SQL-context. Apache Spark aims for structured data processing and acts as distributed SQL-engine. To transform the data, the schema of the table is to be set and alongside the DataFrame registered in the SQL-context. Once this is done, a SQL query can be used to obtain the data in the format of an H2OFrame. A H2OFrame is, analogue to the DataFrame, the representation of data sets within the H2O-context. For the experiments all features for every trip were fetched as training data.
Once the training data are in the right format the model for the deep learning auto-encoder is set up and parameterized. This is done by initiating an instance of the **DeepLearning** class of the Sparkling Water API. The parameters for experiment 1 and experiment 2 are shown in table 2 and table 3 respectively.

For the prediction the model provides an interface called *scoreAutoEncoder(..)*, which takes the test data as parameter. As stated in section 3.1 in our case the test dataset is equivalent to the training dataset. The result of the prediction is a vector containing the reconstruction error of the features for every trip. As the vectors of H2OFrames are immutable the resulting reconstruction errors can be joined to the test dataset row by row. This is simply done by adding the error as additional column to the input data set. Furthermore the columns of the features are deleted as they are not relevant for further evaluations.
In the final step the H2OFrame is transformed back into the Spark RDD representation. Back in the Spark context, a simple map operation is used to normalize the reconstruction error by using formula (1).

$$\hat{x_i} = \frac{x_i - min(X)}{max(X) - min(X)} \tag{1}$$

After normalizing, the Spark API is used to print the the output to a text file. The resulting text file is in the Kaggle format, consisting of 547200 rows containing the probabilities for each driver and trip. In addition to the output for Kaggle a second file is created containing the raw reconstruction error instead of the probability.

4.3 Evaluation of the Result

In order to be able to obtain indications as to whether the deep learning auto-encoder is suitable for anomaly detection, we have to somehow evaluate its findings. Since our dataset is of unlabeled nature, we do not have any labels to test and evaluate the deep learning model on. As stated in section 3.2, the dataset was the subject of a Kaggle.com challenge. Our prediction of outliers will automatically be graded after uploading it to this site. The website will give us the accuracy of our predication as value of the area under the ROC curve. The Kaggle submissions format is outlined in Table 4. Every row represents a trip, labeled by driver ID and trip ID, along with a probability whether said trip actually belongs to the driver or not. Value 1 indicates the highest probability, 0 the lowest.

5. RESULTS

driver_trip,	prob
1_1,	1
1_2,	1
1_3,	1
1_4,	0
...	...
...	...

Table 4: Submission format for Kaggle.com

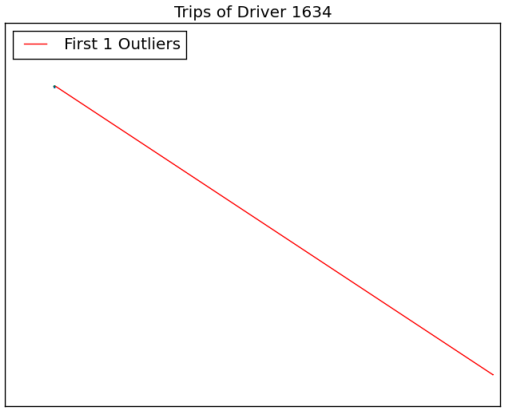


Figure 12: All drives of driver 1634 including junk drive

5.1 Kaggle Results

[Write something about obtained Kaggle results here]

5.2 Qualitative Evaluation

6. CONCLUSION

As outlined in...
I

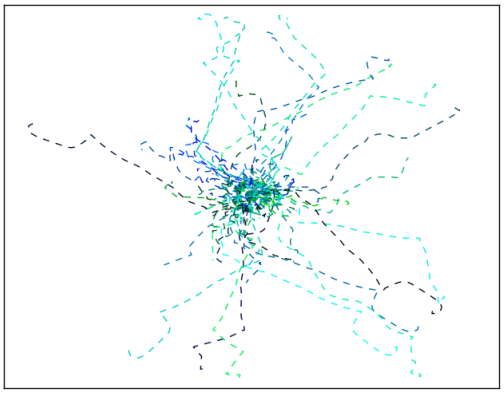


Figure 13: Plot including junk drive

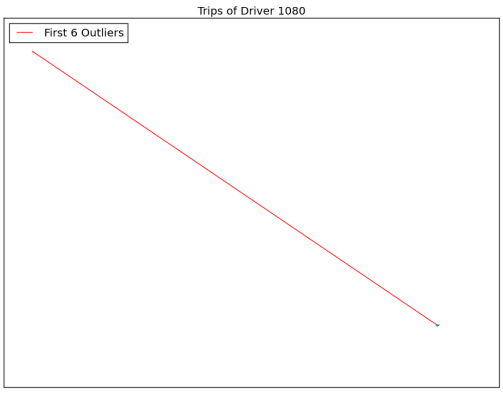


Figure 14: Plot including junk drive

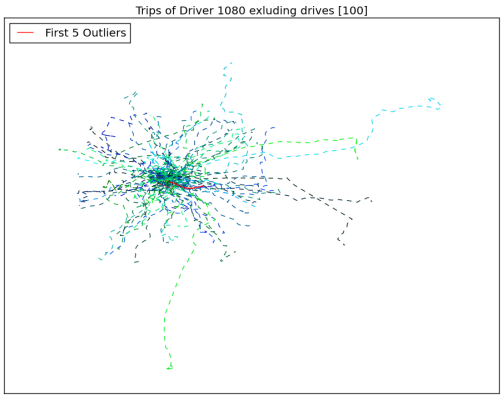


Figure 15: Plot including junk drive

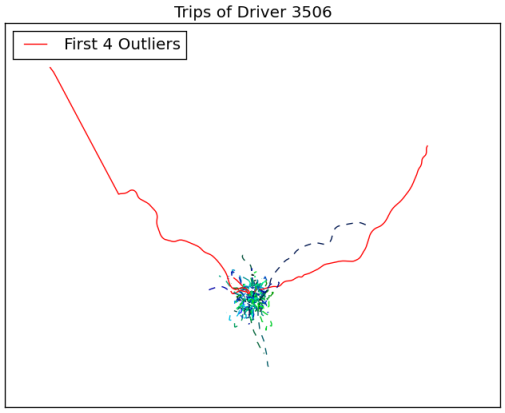


Figure 16: All drives of driver 3506. Drives with the highest reconstruction error are marked in red