Unsupervised Anomaly Detection using *H2O.ai*

AIM-3 - Scalable Data Analysis and Data Mining

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

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

ABSTRACT

In the era of Big Data and scalable data analytics, we observe a rapidly developing space of both supervised and unsupervised algorithms for various different purposes within the domain of machine learning. Incremental progress in some spaces and rebirth along with signficant improvements for whole families of algorithms like neural networks mark the last few years of research in this area. Those constant improvements of algorithms together with the ubiquity of digital sensors and thus data, allow for application of algorithms in the real world yielding valuable insights for the respective user of those algorithms. Remarkable progress has been made in particular, in the family of deep learning algorithms for different kinds of applications, primarily though, in image recognition as well as natural language processing.

In this work, we investigate the performance of a specific algorithm from the family of deep learning, originally designed for a different purpose, in the domain of unsupervised anomaly detection. Throughout this investigation, we try to provide insights into the usability as well as the suitabiliy of the algorithm for this problem domain, by applying an existing implementation oft the algorithm to a publicly available dataset.

1. INTRODUCTION

Anomaly detection (commonly referred to as *outlier detection*) is one of a few hot topics 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". [7] 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.

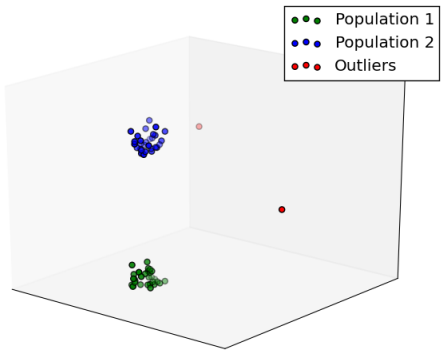
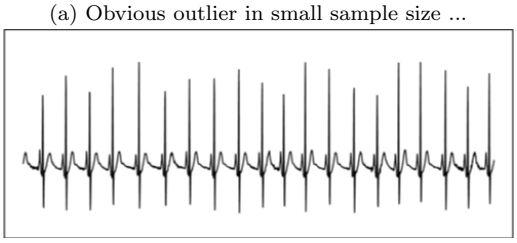


Figure 1: Sample of two populations a two outliers.

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.

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



(b) ... are not necessarily outliers in the context of the whole dataset

Figure 2: Contextual anomaly

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." [6] 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.[7] 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. 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 [7], 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 [6]
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.

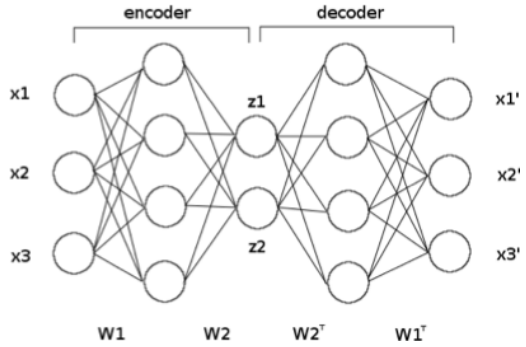


Figure 3: Schematic diagram of a basic auto-encoder with three input features

1.2 Deep Learning Auto-Encoder

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. [1] 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. [1] 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. [4]. 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 [2] 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 dataset 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:

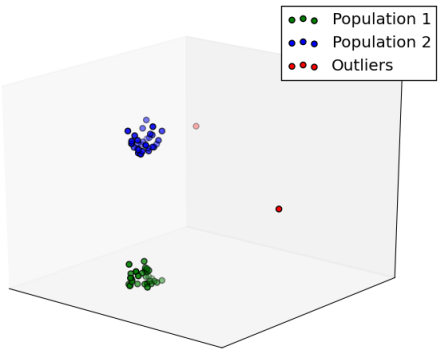
1. Study an existing implementation of the deep learning auto-encoder model
2. Apply this implementation to a given unlabeled dataset
3. Compare the outcome to already existing outcomes of other algorithms
4. Draw conclusions about the general suitability of the algorithm based on the results produced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

3. METHODOLOGY

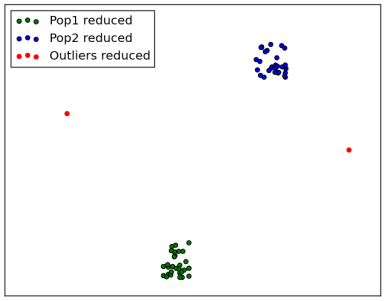
Within the scope of this project, we use H2O.ai's (see section 3.1) implementation of the deep learning auto-encoder model through its Sparkling Water API on top of Apache Spark. The dataset we use in order to be able to compare our results to those of our peers using different algorithms is the AXA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

3.1 H2O.ai and H2O Deep Learning

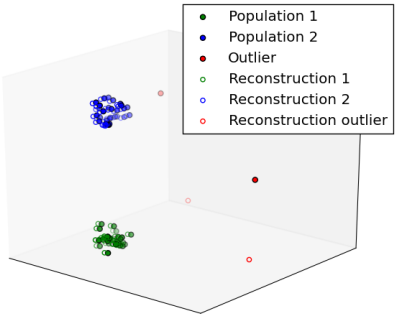
H2O by H2O.ai is an open source software project primarily used for fast scalable in-memory machine learning. The product strongly aims for data scientists who work in a distributed manner. The software offers a predictive analytics platform, combining high performance parallel processing with an extensive machine learning library. [3] 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



(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

and is deployed by more than 2,000 companies, including PayPal, Nielsen and Cisco. [5]

H2O is shipped with its own distributed in-memory data store which is integrated as an key-value store on top of non-blocking hash maps. The basic type for datasets in H2O is the H2OFrame. An H2OFrame consists of vectors, where each vector represents one column, respectively one feature. As opposed to the H2OFrame, the vectors are immutable. The access to the data is granted by an evaluation layer build with R. This layer also builds the bridge to the REST interface. For the computation part the base is build by basic operations as fork, join, map and reduce. These are used by the H2O prediction engine. All algorithm, either provided by H2O or custom implemented, 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 run the above described operations. The H2O cloud is accessible 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.
The H2O version of its deep learning auto-encoder is based on simple map reduce tasks as of that is a fast an memory efficient implementation. It profits from an multi-threaded implementation that allows a distributed parallel computation for deployment on a multi-node cluster. The algorithm uses stochastic gradient descent for the problem of minimizing the loss function. It provides regularisation including L1, L2, dropout and Hogwild!. Also the activation and loss function can be parametrized by the user.

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

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

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

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 trip

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.

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 jus-

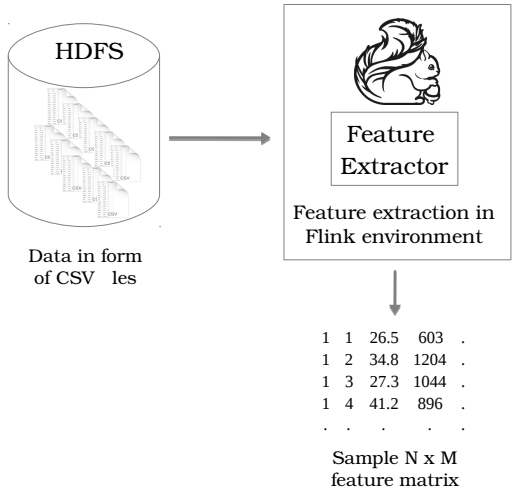


Figure 6: Schematic visualization of our feature extraction

tified.
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 format contributions to the Apache Flink core project are made to create the necessary API for preserving 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 programmed. Simply reading the CSV files would destroy the context of the data. The trick is, to not split the files while reading, as it is usually done on a distributed file system like HDFS. 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 (see formula (1)), time of the trip, speed (see formula (2)), acceleration (see formula (3)), changes of heading (see formula (4)). 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.
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.

$$\delta d_i = \sqrt{(x_{i-1} - x_i)^2 + (y_{i-1} - y_i)^2} \tag{1}$$

$$\delta v_i = \frac{\delta d}{\delta t} \tag{2}$$

$$\delta a_i = \frac{\delta v_{i-1} - \delta v_i}{\delta t} \tag{3}$$

$$\cos \alpha_i = \frac{\vec{v_1} \cdot \vec{v_2}}{|\vec{v_1}|_2 * |\vec{v_2}|_2} \tag{4}$$

where:

$$v_1 = p_{n-1} - p_n$$
$$v_2 = p_n - p_{n+1}$$

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

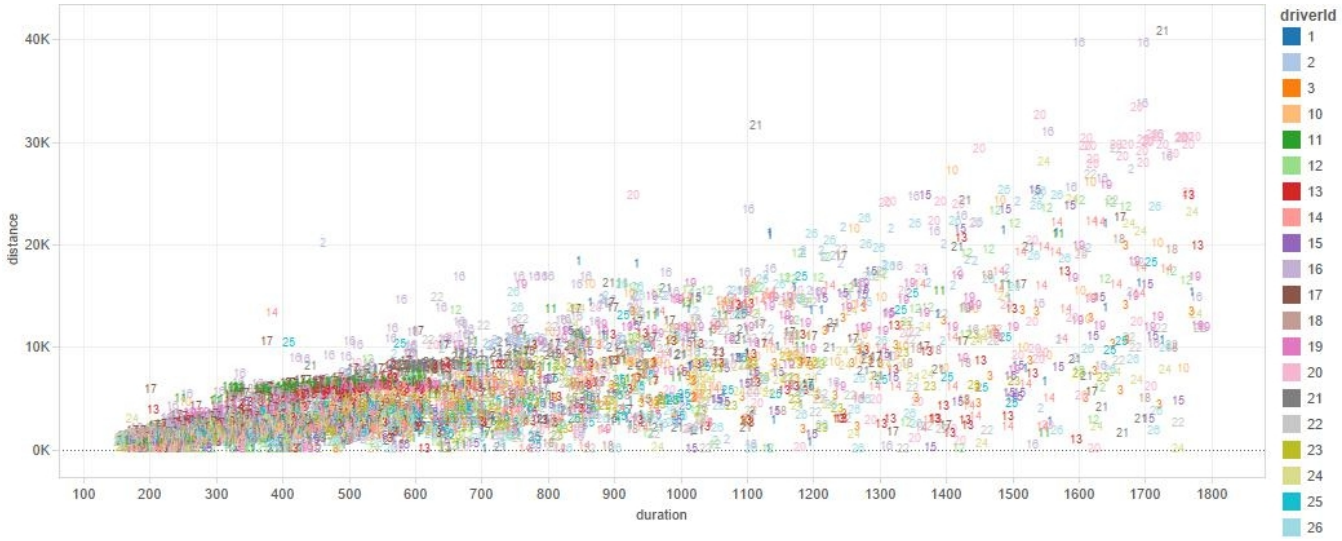


Figure 7: Plot of duration versis distance for each trip by each driver

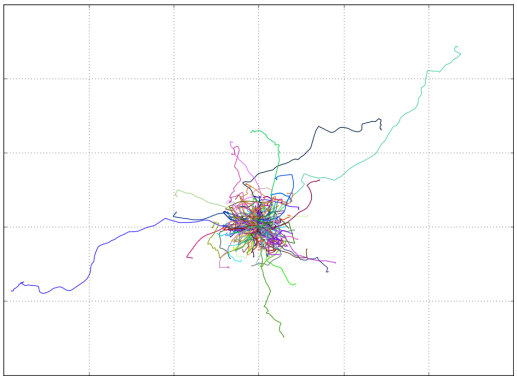


Figure 8: All trips of driver 18 visualized

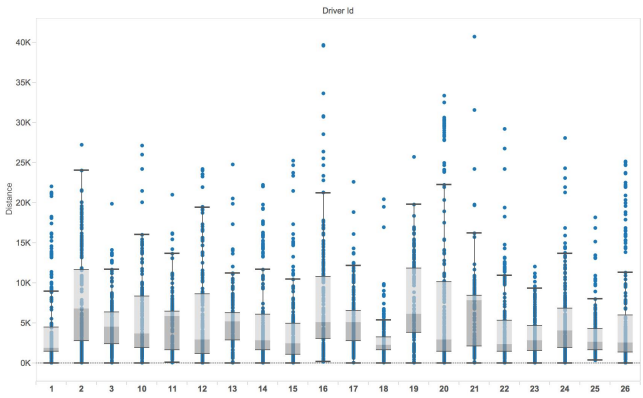


Figure 9: Boxplot of distances for 20 driver

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 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

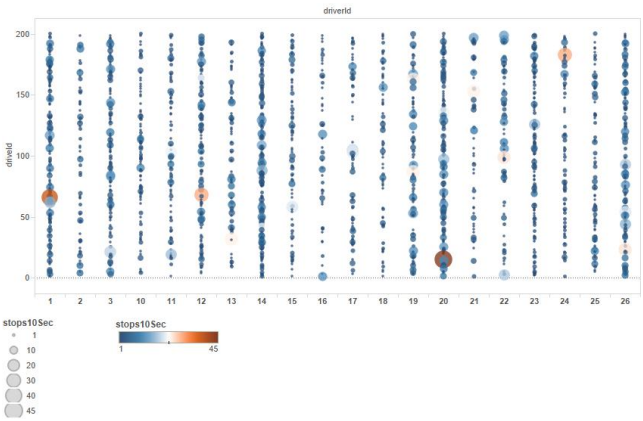


Figure 10: Number of stops with 10 - 120 seconds

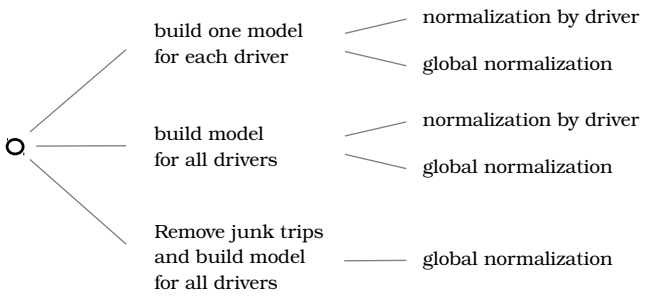


Figure 11: Experiment plan

the might originate from a rural driver.

4. EXPERIMENTS

In order to ensure a structured approach for finding the optimal application of the deep learning auto-encoder on the AXA Driver Telematics datast and the corresponding hyper-parameters, we set up a plan to run the experiments. A simplification of the execution plan can be seen in figure 11. To cover the question, whether the anomalies can only be found among the trips of one driver or even in a more general global context, we build the model in two different ways. First the model is fitted on the entire trainings dataset. In the second and third approach one model for each driver is build. In the third run a recursive implementation is chosen.

4.1 Experiment 1

In experiment 1 a generalized model is build. The idea behind this approach is, that a wide model will cause a certain reconstruction error for each feature vector. Hence the generalization this error will not be minimal, but bigger for anomalies. Our assumption is that the resulting vector of reconstruction errors will represent the probability of an outlying data point. The normalization of the reconstruction error is, as it can be seen in Figure 11. The determination of the probability is considered in a local, as well as a global context. A more detailed description of the computations can be found in section 4.4.

4.2 Experiment 2

The second approach aims at overcoming this generalization by assembling trips of one driver to build the trainings set for the deep learner auto-encoder. In a first step a unique set of driver IDs is created. After that, iterative model fitting and scoring is done driver by driver. To avoid overfitting a higher L2-regularisation is applied. The figures can be seen in Table 3. The final result is build as a union of the individual result sets. The normalization is performed as explained in experiment 1.

4.3 Experiment 3

The idea of experiment 3 lays in the result of the initial two experiments (see section 5.2 and 5.2) and the qualitative evaluation as described in section 5.3. Low results on Kaggle.com leave open questions about the quantity of the model. However plots show, that the model is capable of finding outliers at least outliers in a visual respect. For experiment 3 the assumption is made, that extreme outliers influence the model fitting in a negative manner. To prove

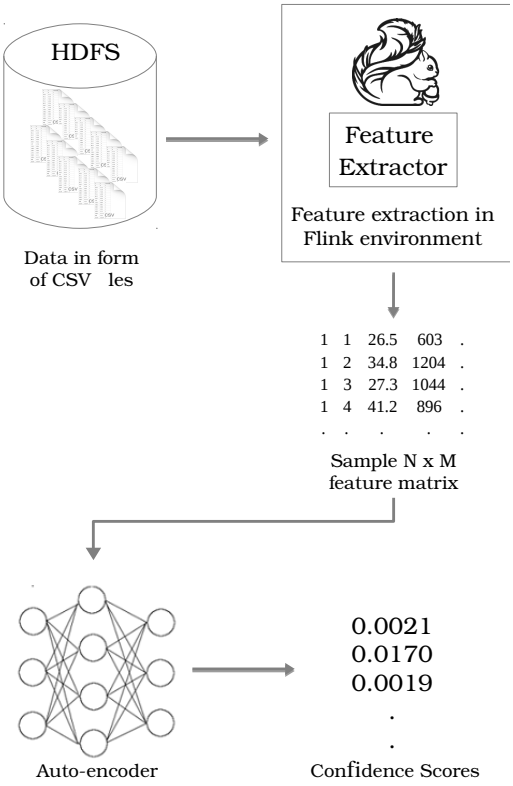


Figure 12: Schematic visualization of experiments

the truth of this assumption we use a recursive model fitting approach. After fitting a model for each driver on the entire dataset the two trips with the highest reconstruction error are removed from the dataset. The filtered dataset is fed back into the model building process. This method helps to prevent a strongly biased model fitting on outlying data points. The parameter set is left unchanged to experiment 2. Hence the trips with the biggest reconstruction error are removed they also will not influence the final computation of the probability as described in formula (5).

4.4 Sparkling Implementation

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 transformations are explained. See figure 12 for an overview. The first step is to load the input data from the HDFS on to 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 at structured data process-

Parameter	Value
training data	all available trips (2736 x 200)
response column	any non static column
autoencoder	true
activation function	tanh
hidden layers	1
Neurons per hidden layer	{ 8 }
epochs	20
L2-normalization	0.001

Table 2: Parameter set for deep learning auto-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	1
Neurons per hidden layer	{ 8 }
epochs	20
L2-normalization	0.2

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

ing 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 datasets within the H2O-context. For the experiments, we fetch all features for every trip as training data.
Once the training data is in the right format the model for the deep learning auto-encoder is set up and parametrized. 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. This parameter set is chosen based on the evaluation results returned by Kaggle.com. As the model fitting takes place in a unsupervised environment, the response column will be ignored and for that reason set to any non static column. Tangens hyperbolicus provides a bounded activation function for the output forwarding to the next neuron. It is a simple activation function and the symmetry around 0 makes the algorithm converge faster. The number of hidden layers is set to 1 layer with 8 neuron. The number of neurons on the hidden layer is determined by applying Principal Component Analysis (PCA) on the dataset and getting an estimate of the number of principal components that cover most of the variance in the dataset.

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 an H2OFrame 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 dataset. 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 (5).

$$\hat{x_i} = \frac{x_i - min(X)}{max(X) - min(X)} \tag{5}$$

After normalizing, the Spark API is used to print 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.

5. RESULTS

5.1 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. The common approach in data analysis and machine learning for classification is to do k-fold cross-validation method to find the optimal hyper-parameter. This parameter set minimizes the error of the prediction. For this the dataset is at least divided into to random splits, usually 70/30, for training and testing respectively. After fitting the model with the training dataset, predications on the test dataset are made. By comparing the predicted result and the true labels the error is calculated. This error represents the quality of the model. In a unsupervised learning environment such a optimization strategy is not possible, due to the absence of labels for the dataset. One measurement is the reconstruction error for anomaly detection by dimensionality reduction. As stated in section 3.2, the dataset was the subject of a Kaggle.com challenge.For the scope of our problem statement we luckily can use the evaluation of the prediction on the website. Our prediction of outliers will automatically be graded after uploading it. 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.
As a second method towards quality evaluation a visual approach was chosen. The trips with the n biggest reconstruction error are filtered from the original dataset. In a second step the m biggest outliers of each driver found in step one is collected. This method provides a decent amount of result for visual evaluation. Utilized by python and matplot-lib the trips and outliers are plotted. For our experiments we filter 15 trips with the biggest reconstruction error and highlight the first most significant outliers for each unique driver among the filtered trips.

5.2 Kaggle Results

As stated in Section 5.1, we upload the predictions obtained from our implementation to Kaggle.com, where they are being evaluated.

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

Table 4: Submission format for Kaggle.com

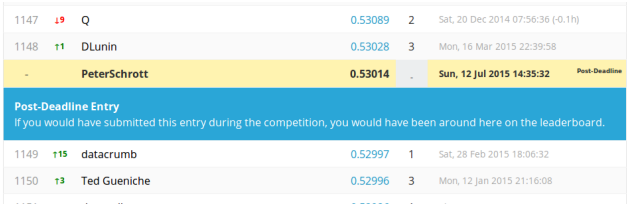


Figure 13: Scoring on Kaggle.com's leaderboard for experiment 1

Result of Experiment 1

The score on Kaggle.com for experiment 1 was quite low. We achieved a ROC score of 0.53014 as can be seen in Figure 13. This score is only slightly better than randomly guessing or labeling every trip with the highest probability. What this experiment also showed is, that the result is invariant towards changes in normalization, activation function, loss function as well as the methodology of building the deep learning auto-encoder model. Also, the different approaches for computation of the probability does not significantly change the score.

Explanations for this result are among others:

- Too high degree of generalization of the model
- Wide variances between individual drivers
- Lack of cross-validation for optimizing the hyper-parameter

Result of Experiment 2

Driven by the sobering low results of experiment 1 and the resulting change of model fitting strategy increased the score on Kaggle.com by almost 20%. As shown in Figure 14 the score reached 61.897% of a prediction accuracy. This result confirms the assumption drawn in experiment 2. We claimed a less general fitting approach will result in a better detection of anomalies. As the result shows, the variance among the drivers is overcome and the individual outliers are found more reliably.

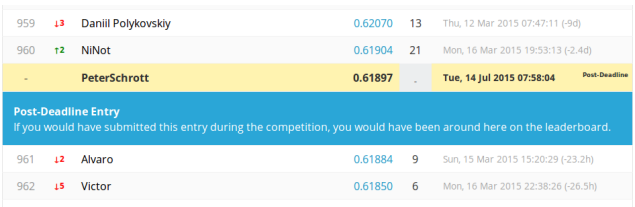


Figure 14: Scoring on Kaggle.com's leaderboard for experiment 2

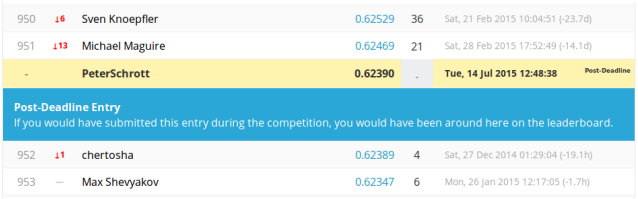


Figure 15: Scoring on Kaggle.com's leaderboard for experiment 3

The mid-range quality of this result can be explained by the following reasons:

- Noisy data can influence the fitting of the model
- Lack of cross-validation for optimizing the hyper-parameter

Result of Experiment 3

Experiment 3 for a more stable model towards *learning what's normal* once again resulted in an improvement of accuracy, thus yielding a higher score on Kaggle.com. Removing the two trips with the highest reconstruction error improves the prediction result as can be seen on the rating of an area under the ROC curve value of 0.6239. The result is shown in figure 15.

5.3 Qualitative Evaluation

For the qualitative evaluation, the drivers having trips with especially high reconstruction errors are filtered from the original dataset. For each unique driver out of the filtered data, all trips are plotted. Those trips with the by far highest reconstruction error are then marked red.
Manually evaluating those chosen trips (without true labels of course) shows some interesting insights, giving more hints as to how the poor accuracy of predictions came to be, as opposed to the value of the area under the ROC curve given by Kaggle.com, which does not allow for more assumptions.

In Subfigure 17a it can clearly be seen, that the driving trip with the highest reconstruction error is a false recording or junk drive, as all the other trips can barely be seen (small fuzz in the top left corner). This becomes even more obvious after removing said junk drive from the plot with the goal of being able to see the structure of the other drives, as illustrated in Subfigure 17b. This figure reveals the true structure of the nonjunk drives of driver 1634. The same pattern can be seen in Figure 18. The unintuitively edgy and straight looking lines can clearly be classified as data noise from an objective point of view. Those specific trips are clearly not more than junk data resulting from poor GPS recordings. Apparently, only jumps in locations are recorded, even outside of the boundary of a usual trip distance as Subfigure 17a shows in a clear manner. For other drivers, not containing obvious junk drives, the model seems to successfully identify (judging by eye) anomalous driving trips. This can be seen in Figure 16
The conclusions that can be drawn from the qualitative evaluation is, that the model was successfully trained to find outliers. Unfortunately, our feature extraction does not account for all data noise and junk drives, which might cause problems in two ways. First of all, the model (or what is

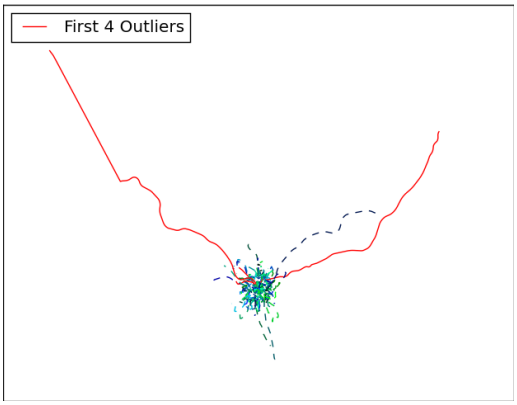


Figure 16: All trips of driver 3506. Trips with the highest reconstruction error are marked in red

considered normal) will additionally to the influence of the outliers be influenced by the data noise. Secondly, resulting confidence scores that a data point is anomalous are of course higher for junk trips, because those are - in a way - anomalous. The computation of the outlier probabilities, based on the reconstruction error, possibly causes the suboptimal result on Kaggle.com. Observing formula (5) shows, that if the maximum of the reconstruction error is far off from the other values, the range of the probabilities is distorted. That is, a junk drive that becomes a extraordinarily high confidence score of being anomalous, causes the other drives to have a in relation to that score really low score. Computing the outlier probabilities then results in having a really high number of trips with a probability of being normal close to 99 %, while only the few junk drives have a low probability of being considered normal. This can also be seen by analyzing the files, uploaded to Kaggle.com.

6. CONCLUSION

The aim of our project was to provide evidence to the question as to whether the deep learning auto-encoder is suitable for anomaly detection in an unsupervised environment. Applying an existing parallel implementation of the algorithm from a software house with a particularly strong reputation - namely H2O.ai - should enable us to provide evidence to the answer to the before mentioned question to at least some extent. Since evaluating unsupervised learning algorithms is a particularly tough task, in particular if the dataset's size has to be big enough to be considered "Big Data", working on a dataset from Kaggle.com, should allow us to let their system take care of the model evaluation part. Additionally, the availability of scores from more than 1,500 other teams using other algorithms would allow us to compare the auto-encoder's performance directly to those of other algorithms.

As we describe in Section 5, our third and last experiment would have finished 952th out of 1,529 teams by obtaining an area under the ROC curve value of 0.6239. This result is far from being optimal for the use case of anomaly detection. However, it is also significantly better than the random classifier benchmark of 0.5. That being said, taking our inexperience with neural networks and unsupervised anomaly detection into account, one might argue, that an expert in the area or at least a more experienced team might achieve better results using the same algorithm, but in a more experienced design of experiments. Also, paying more attention to the noise contained in the dataset, along with smarter feature extraction might lead to further improvements. Conceptionally, however, we show, that the deep learning auto-encoder is applicable to real world unsupervised anomaly detection, although it is particularly tough to find the right hyper-parameters.

For future work we propose applying the algorithm to an already cleaned dataset, with - if possible - already existing features. The fact that unsupervised anomaly detection becomes even harder in the presence of noisy data lets us reach this conclusion. In particular, we propose applying the deep learning auto-encoder to a dataset consisting of digital images, in order to be able to draw more conclusions in regards to its suitability for this domain, without spoiling the results with a poor choice of features or a lot of noise in the data.

7. REFERENCES

[1] Y. Bengio. Learning deep architectures for ai. Technical report, University of Montreal, 2009.

[2] P. G. et. al. Skynet: An efficient and robust neural network training tool for machine learning in astronomy. *Monthly Notices of the Royal Astronomical Society*, 2013.

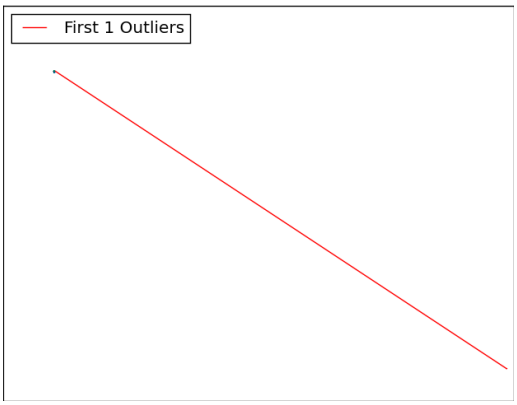
[3] H2O.ai. http://h2o.ai/product/.

[4] N. Japkowicz. Nonlinear autoassociation is not equivalent to pca. *Neural Computation*, 2000.

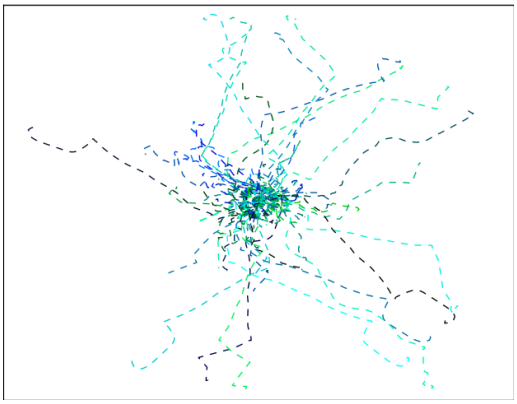
[5] A. C. . V. Parmar. Deep learning with h2o.

[6] E. F. Ted Dunning. *Practical Machine Learning: A New Look At Anomaly Detection*. O'Reilly, 2014.

[7] V. K. Varun Chandola, Arindam Banerjee. Anomaly detection : A survey. *ACM Computing Surveys*, September 2009.

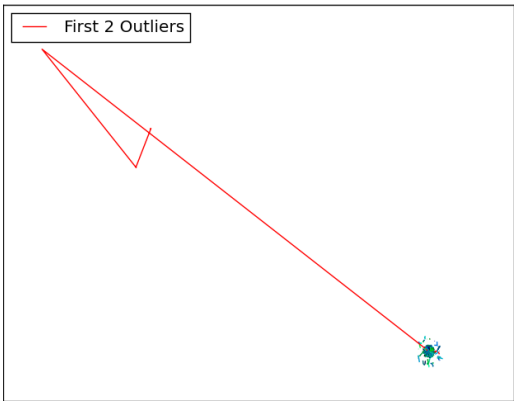


(a) All trips including junk

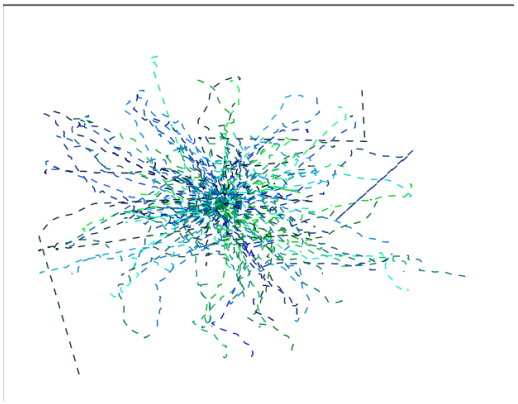


(b) Trips without junk drive

Figure 17: All trips of driver 1634



(a) All trips including junk



(b) Trips without junk drive

Figure 18: Trips of driver 1635