NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS

Faculty of Computer Science
Bachelor's Programme "Data Science and Business Analytics"

Software Team Project Report on the Topic:

Development of a System for Analyzing Telemetry And Identifying Anomalies in the Operation of Cloud Infrastructure

Submitted by the Students:

group #БПАД222, 2nd year of study	Fatianov Vladimir Andreevich
group #БПАД222, 2nd year of study	Fokina Viktoriia Konstantinovna
group #БПИ216, 3rd year of study	Ryzhkov Viacheslav Andreevich

Approved by the Project Supervisor:

Bulankin Danil Andreevich
TeamLead
1C

Moscow 2024

Contents

Annotation

The aim of this project was to create different machine learning models, which would be able to detect anomalous data in time series. Methods such as iForest (Isolation Forest) and LSTM (Long Short Term Memory) have been used. We have implemented version of the iForest algorithm in the accordance with the original paper. In addition to that, LSTM autoencoder, with the usage of TensorFlow Keras library, has been written.

Аннотация

Целью данного проекта было написание различных моделей машинного обучения, способных обнаруживать аномальные данные во временных рядах. Были использованы такие методы, как iForest (Isolation Forest) и LSTM (Long Short Term Memory). Мы реализовали версию алгоритма iForest в соответствии с оригинальной статьей. Кроме того, был написан автоэнкодер LSTM с использованием библиотеки TensorFlow Keras.

Keywords

Machine Learning, Time Series, Anomaly Detection, Data processing, iForest, LSTM

1 Introduction

1.1 General Idea

It is crucial for every company to proactively detect anomalies in their data to prevent any malfunctions, such as a service crash or data loss, which results not only in financial losses but also undermines trust of users and partners. Monitoring systems in cloud infrastructure play a key role in ensuring the stability and reliability of services, because they make it possible to detect anomalies early, which allows to find errors, either before any serious consequences occur or as soon as possible after their occurrence and fix them in time. This project is highly relevant as there is currently a need in the market for domestic monitoring systems that have the ability to detect anomalies. Thus, it will be possible to satisfy the problem of partial import substitution of already existing services for searching for anomalies in time series.

The main objective of this project was to develop and implementing machine learning models and neural networks that are be able to search for anomalies in time series - dynamic

data collected and recorded at consecutive time intervals. The idea behind the project was to integrate various anomaly search methods in a certain environment, which is going to retrieve data from the Prometheus system. Prometheus is designed to record metrics in real time, the data will be transmitted to the Prometheus monitoring system using a push mechanism. Then, through API calls, the data would be processed using various methods. This process aims to extract responses, predictions and other metrics, providing a deeper analysis of the time series and identifying potential anomalies in the data.

Our part of the project requires writing models to find anomalies. As models we chose were iForest and an LSTM neural network. The choice of iForest was justified by the fact that it is a relatively new algorithm, plus it runs in linear time and produces fairly accurate results with minimal computational load. LSTM, on the other hand, was chosen because of its ability to predict the data, providing early warning of anomalies. In addition to this, LSTM is a recurrent neural network capable of memorizing information over long periods of time and thus working more efficiently with current data by linking them together. Both models are functioning independently of each other. iForest and LSTM have different characteristics and ways of processing data, so they perform differently on various amounts of data and handle problems such as data swamping differently. Based on all of these factors, the user can choose the best model based on specific results and their own requirements.

With all said, iForest and LSTM models had been written by both Vladimir and Viktoriia, while organisation of the infrastructure, which would handle data and models,has been developed exclusively by the Viacheslav Ryzhkov.

1.2 Overview of existing solutions

While the global market offers a variety of solutions for monitoring and searching for anomalies in data, the Russian market is currently facing a shortage of open and affordable systems capable of meeting the needs of domestic companies. Most of the time series database management systems are developed not in Russia. Nowadays many companies, such as VictoriaMetrics, have left the Russian market. It has become impossible to pay for other foreign analogs, such as DataDog. Therefore, there is a critical need to create our own solutions that will be able to become a replacement. The problems with the existing services available in the Russian market are that the alternatives either do not specialize in anomaly detection or are too expensive and do not offer free solutions.

2 Literature review. Description and characterisation of relevant works

To detect anomalies, we have identified two methods: Isolation Forest and LSTM.

- There are different ways to implement the isolation forest algorithm. For example, the very first version [12], proposed in 2008, has drawbacks, such as inability to adapt to data streams. Nevertheless, some of this issues have been addressed in later versions. For instance, SCiForest [13], a modernisation, which was proposed in 2010, is able to separate different clusters. Another version, called EIF [1] (Extended Isolation Forest), fixes the problems with assigning anomaly scores to given data points arising in iForest due to the vertical or horizontal separation of nodes. However, only four modifications are identified as capable of handling continuous data streams: HS Forest [7], iForest ASD [2], RS-Forest [5] and PCB-iforest [3]. In addition to the original version of the algorithm, we decided to consider iForest ASD, thus to write both iForest and iForest-ASD, compare them and use the most efficient one.

- LSTM [4] for anomaly detection has been applied to time series in different ways and for different purposes. Some of the methods of using LSTM for anomaly detection are using supervised and unsupervised [9] learning, combining LSTM with additional methods including clustering [10], and autoencoders are some common approaches. One of the early works in LSTM autoencoders are by Malhotra et al [11], [8] where stacked LSTM networks and an LSTM based Encoder-Decoder anomaly detection model for multivariate time series data were proposed. Their method performed better than traditional time series analysis methods in detecting anomalies such as k-means clustering and ARIMA models. It has drawbacks, because creators used a multivariate Gaussian distribution assumption for error vectors, which could not hold true in real life. Nguyen et al [6] in their paper described a method to avoid this problem using a quantile kernal estimator to define a threshold which is needed for detecting anomalies. Our code includes stacked LSTM layers for both encoding and decoding, which is a part of Malhotra et al.'s methodology. However, our code does not rely explicitly on Gaussian distribution, it uses other approach for optimal threshold. We used precision recall curve function from sklearn library to calculate it.

3 Main part

3.1 Isolation forest

3.1.1 Introduction

This part of the work is going to be dedicated to stating the problem, identifying principle of iForest and work, which has been done.

3.1.2 Problem description

First and foremost, idea of iForest should be discussed. iForest algorithm [12] proposed by Fei Tony Liu, Kai Ming Ting and Zhi-Hua Zhou in 2008 is based on the principles of isolation, meaning that each point in a dataset could be isolated from others with n number of splittings. Since in real data anomalies are quite rare occurrence, points stand out from regular ones, thus it would require only few splittings to isolate them from other points.

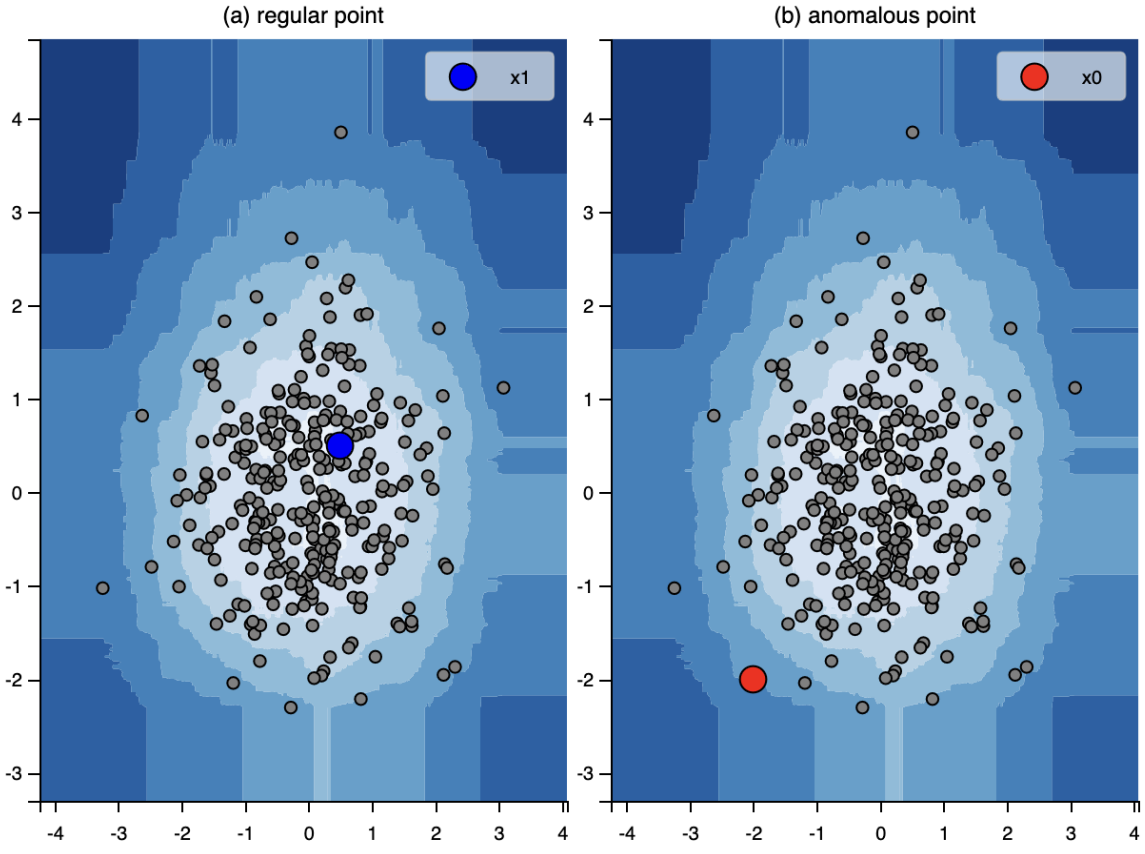


Figure 3.1

As seen in Figure 3.1, the anomalous data point is located in a less dense area, while the non-anomalous one is situated in the region with high density. Thus, isolation of non-anomalous point from other points would require many splits.

The points are isolated from each other by constructing a random forest that contains many random trees. A random tree is a tree that is built by randomly selecting features and partitioning them into subsets. The maximum depth of the tree is then set, the maximum depth is given by the formula for unsuccessful search in a binary search tree $c(n) = 2H(n-1) - (2(n-1)/n)$, where H is defined as the harmonic number and it can be estimated by $ln(i) + 0.5772156649$ (Euler's constant). Thus the tree will be built until it isolates all points belonging to it or until it reaches the height limit. After that, the length of the path to each point in each tree of the same forest is calculated, based on it anomaly score for each point is received from this formula $s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$, if score is greater than the given threshold, then such a point is considered as anomalous.

3.1.3 Deeper dive in iForest

General idea of iForest has been mentioned earlier, however it is important to research other aspects of it. Before training a model, its parameters must be set. The original implementation implies that two parameters can be passed at this stage: the sample size and the number of trees. The sample size means how many points (rows from the dataset) will be randomly taken to build each tree, the number of trees means how many trees will be built by the model. After that, the process of training the model on a non-normalized protaiyet takes place. Then the model is used to predict the anomaly, in addition to the dataset itself in which to find anomalies, a threshold is passed, which will determine whether the point is considered anomalous or not. Thus, iForest offers quite effective anomaly detection abilities with relatively low load and high speed. Nevertheless, iForest is not ideal for streaming data, since it is only able to adapt to natural changes in data, hence there are other algorithms which are able to do so.

3.1.4 iForest based algorithm

The simplest algorithm which theoretically could be useful on streaming data, iForest-ASD [2], is based on the sliding window principle. The main problem faced by the authors (and everyone else, including us) is the choice of the sliding window dimension. If the dimension is too large, the model will not be able to correctly determine drift, and if it is too small, there will not be enough data to build the right model. The algorithm assumes that when the behavior of the data changes, the entire model will be rebuilt based on the current window. Unfortunately, the authors of this algorithm used a fixed window because it is extremely difficult to derive a unified algorithm that can automatically take into account the dimension. Moreover, when rebuilding

the model, historical data is forgotten, which can lead to negative results in cases where previous values are important, for example, when considering seasonality.

iForest-ASD represents a rather raw idea since the modifications proposed by the authors (dynamically changing window dimensions; more efficient calculation of the average anomaly level, depending on which the model will automatically rebuild; selective, not complete removal of data history during model reconstruction) have not yet been implemented and more than 11 years have passed since iForest-ASD release. We have tried to implement iForest-ASD according to authors papers, but our implementation either gave a good AUC score and very poor performance or vice-versa. In addition to that iForest-ASD requires window to be filled, which also worsens response time. Hence, it has been decided, that iForest is going to be our first model, since it showed very good results in synthetic tests.

3.1.5 iForest implementation

As mentioned earlier, we opted to refrain from using iForest from libraries as SkLearn, since such implementations are very far from the original one. Moreover, it would be possible to implement other version of models, which are based on the original implementation.

First algorithm, which was described in the paper and could be seen on Figure 3.2, is called IForest, it is ensemble of all iTrees. It consists of 4 functions: fit, path_length, anomaly_score and predict (however pseudo code from original paper describes only fit function). Fit is used for model training, purpose of other functions has been described before.

Algorithm 1 : $iForest(X, t, \psi)$

Inputs: X - input data, t - number of trees, ψ - sub-sampling size
Output: a set of t *iTrees*

1: **Initialize** $Forest$
2: set height limit $l = ceiling(\log_2 \psi)$
3: **for** $i = 1$ to t **do**
4: $\quad X' \leftarrow sample(X, \psi)$
5: $\quad Forest \leftarrow Forest \cup iTree(X', 0, l)$
6: **end for**
7: **return** $Forest$

Figure 3.2: from [12]

```
class IForest:
    def __init__(self, sample_size=256, tree_num=100):
        self.sample_size = sample_size
        self.tree_num = tree_num
```

```
        self.forest = []
        self.height_limit = np.ceil(np.log2(sample_size))

    def fit(self, X: np.ndarray):
        for i in range(self.tree_num):
            x_prime_indices = np.random.choice(X.shape[0], self.sample_size)
            x_prime = X[x_prime_indices]
            tree = ITree(self.height_limit)
            tree.fit(x_prime, 0)
            self.forest.append(tree)
        return self

    def path_length(self, X: np.ndarray):
        allPL = []
        for x in X:
            pathLength = []
            for tree in self.forest:
                pathLength.append(PathLength(x, tree.root, 0))
            allPL.append(pathLength)
        paths = np.array(allPL)
        return np.mean(paths, axis=1)


    def anomaly_score(self, X: np.ndarray):
        avg_path_lengths = self.path_length(X)
        scores = []
        for h in avg_path_lengths:
            score = 2 ** (-h / c(self.sample_size))
            scores.append(score)
        scores = np.array(scores)
        return scores

    def predict(self, X: np.ndarray, threshold: float):
        anomaly_scores = self.anomaly_score(X)
```

```
        return np.where(anomaly_scores >= threshold, 1, 0)
```

Algorithm 2 : $iTree(X, e, l)$

Inputs: X - input data, e - current tree height, l - height limit

Output: an iTree

1: **if** $e \geq l$ or $|X| \leq 1$ **then**
2: return *exNode*$\{Size \leftarrow |X|\}$
3: **else**
4: let Q be a list of attributes in X
5: randomly select an attribute $q \in Q$
6: randomly select a split point p from max and min values of attribute q in X
7: $X_l \leftarrow filter(X, q < p)$
8: $X_r \leftarrow filter(X, q \geq p)$
9: return *inNode*$\{Left \leftarrow iTree(X_l, e+1, l)$,
10: $Right \leftarrow iTree(X_r, e+1, l)$,
11: $SplitAtt \leftarrow q$,
12: $SplitValue \leftarrow p\}$
13: **end if**

Figure 3.3: from [12]

Algorithm 2 (Figure 3.3) describes process of building iTress. In order for iTree operate correctly, we have created 2 additional classes, ExNode (External Node) and InNode (Internal Node).

On the other hand InNode stores SplitAtt - a random feature index q from the range from 0 to (Num of features - 1), which would be used to split the data in the node. SplitVal - random value p from the interval between the min and max values of the feature q. This random value p determines the threshold at which the data will be divided into two subtrees. Points with a feature value lower than p are sent to left subtree and points with a feature value greater than or equal to p are sent to the right subtree. All of this values are assigned inside of the ITree class from below

```
class ExNode:
    def __init__(self, size):
        self.size = size


class InNode:
    def __init__(self, left, right, splitAtt, splitVal):
        self.left = left
```

```
        self.right = right
        self.splitAtt = splitAtt
        self.splitVal = splitVal



class ITree:
    def __init__(self, lim_height):
        self.lim_height = lim_height
        self.root = None

    def fit(self, X: np.ndarray, curr_height=0):
        if curr_height >= self.lim_height or len(X) <= 1:
            self.root = ExNode(len(X))
            return self.root

        Q = X.shape[1]
        q = np.random.choice(Q)
        p = np.random.uniform(X[:, q].min(), X[:, q].max())

        x_l = X[X[:, q] < p]
        x_r = X[X[:, q] >= p]

        left = ITree(self.lim_height)
        right = ITree(self.lim_height)
        left.fit(x_l, curr_height + 1)
        right.fit(x_r, curr_height + 1)

        self.root = InNode(left.root, right.root, splitAtt=q, splitVal=p)
        return self.root
```

Last described algorithm (Figure 3.4) is responsible for calculation of path length.

Algorithm 3 : $PathLength(x, T, e)$

Inputs : x - an instance, T - an iTree, e - current path length; to be initialized to zero when first called
Output: path length of x

1: **if** T is an external node **then**
2: return $e + c(T.size)$ {$c(.)$ is defined in Equation 1}
3: **end if**
4: $a \leftarrow T.splitAtt$
5: **if** $x_a < T.splitValue$ **then**
6: return $PathLength(x, T.left, e + 1)$
7: **else** $\{x_a \geq T.splitValue\}$
8: return $PathLength(x, T.right, e + 1)$
9: **end if**

Figure 3.4: from [12]

In order to calculate path from root to leaf, direction of movement is required to be known. When x[a] < T.splitVal we move to the left subtree, then recursive PathLength function for the left subtree is called, otherwise we move right and do same thing for right subtree.

After that, path length is increased by 1 (because we are moving one level lower) and current path length e is increased by 1 because we are moving closer to the leaf. Upon reaching external node, it is only required to return (e + depth of the exact iTree).

```
def PathLength(x, T, e):
    if isinstance(T, ExNode):
        return e + c(T.size)

    a = T.splitAtt
    if x[a] < T.splitVal:
        return PathLength(x, T.left, e + 1)
    else:
        return PathLength(x, T.right, e + 1)
```

3.1.6 Conclusion on iForest

To sum up, we were able to write a version of the forest that is as close to the original idea as possible. As a result of the tests, it was found that although iForest cannot dynamically adapt to seasonal data changes, it performed extremely well in synthetic tests on KDD Cup 99 HTTP and SMTP data. Algorithms like iForest-ASD attempt to address these issues but come with their own challenges, such as determining the optimal window size for the sliding window

technique. Our implementation and testing of iForest-ASD highlighted these challenges, as well as well as the need for further modifications to enhance its performance. iForest-ASD showed its efficiency (AUC = 0.83) only when a large sliding window size (from 1000) was specified, and the model had to be retrained every window, which extremely degraded performance and took from 10 minutes of real time to process the dataset.

3.2 LSTM

3.2.1 Introduction

LSTM itself stands for Long Short-Term Memory. It is type of a recurrent neural network that can be used to make predictions based on the given data and detect anomalies. To begin with, RNNs are neural networks which have the ability of "remembering" the previous data which is especially good for sequential data such as time series in this case. This happens because they have connections between nodes which can turn into cycles and thus output from some nodes affects input into other nodes. However, RNNs have a drawback - their memory is quite limited and at some point the context is lost, which is especially bad for prediction of data in time series. This problem is fixed in LSTM, which is going to be explored and implemented in this project for anomaly detection.

3.2.2 LSTM and LSTM-Autoencoder

LSTM's major advantage is that it is able to create long-term dependencies between data by connecting previous sequences of data with current ones. LSTM network is of a chain form, where each repeating module connected by layers has three control gates. These gates control the information by erasing redundant data in the "forget gate", memorising in "input gate", and expose current information in the "output gate" [6]. Thus, LSTM uses a set of cells to find out which previous time steps are useful for predicting the current time step. However, LSTM by itself does not detect anomalies, it only predicts the data based on the previous information it has.

An Autoencoder is a neural network that can encode the input into lower-dimension, thus compressing it, and decode it back into output layer. They are used primarily for reduction of data dimension, extraction of features, and denoising of data. We implemented an LSTM-Autoencoder for detecting anomalies in time series. It is a neural network that combines architecture of LSTM units and that of an autoencoder. It handles sequences of data or time series data by using LSTM layers in the autoencoder framework. It outputs a reconstructed version of the input data rather than a prediction of future values. An LSTM-Autoencoder is thus an effective way to detect

anomalies in normal data. We also calculated the reconstruction error, which is informally the difference between real data and data reconstructed by an autoencoder, and compare it to some threshold value determined priorly. The main point of an autoencoder is that it compresses the data while keeping the general features of the data structure. Thus, it is not sensitive to slight fluctuations which are often presented in real data, but it detects significant deviations quite well.

3.2.3 Implementation

We implemented LSTM-Autoencoder using mainly Keras and Tensorflow library to build and train neural network and NumPy for numerical computation, Pandas for data manipulation, and Matplotlib for data visualization. We first load and preprocess the dataset from a csv file using NumPy and pandas, dividing it into features and labels. Labels are outputs which the model tries to predict. We normalize features because it is important for speeding the convergence of neural network while training. The data from features is the divided into sequences of length which a user inputs as timesteps, corresponding label is taken from labels(outcomes). Then we split the data of sequences and labels into training dataset to train our model and validation dataset to evaluate performance of model. The most important part is encoder and decoder. First we define the input layer for the model which is shaped by timesteps and number of features. We need to compress the data into lower-dimensional representation, and the first layer of LSTM is the first step towards it. We use relu activation, because it helps the model to reduce vanishing gradient and learn better. Return-sequences is used to return full output sequence, and l2 regularization to reduce overfitting problem. We use BatchNormalization to improve speed of training. Then another layer of LSTM is used to reduce the sequence even more to a single vector. Then to prepare for decoding the layer repeats timesteps times. The first decoder layer mirrors the layer of encoder, but starting to reconstruct the sequence. The next layer reconstructs it to original dimensions. Layers are not fixed and can be modified including number of neurons, for example for more complex datasets more deep layers is better. Here is an example of hpw we created a model using 4 LSTM layers and the described characteristics.

```
inputs = Input(shape=(timesteps, features_num))
encoded = LSTM(16, activation='relu', return_sequences=True,
               kernel_regularizer=l2(0.01))(inputs)
encoded = BatchNormalization()(encoded)
encoded = LSTM(4, activation='relu', return_sequences=False,
               kernel_regularizer=l2(0.01))(encoded)
```

```
encoded = BatchNormalization()(encoded)

decoded = RepeatVector(timesteps)(encoded)
decoded = LSTM(4, activation='relu', return_sequences=True,
               kernel_regularizer=l2(0.01))(decoded)
decoded = BatchNormalization()(decoded)
decoded = LSTM(16, activation='relu', return_sequences=True,
               kernel_regularizer=l2(0.01))(decoded)
decoded = BatchNormalization()(decoded)

autoencoder = Model(inputs, decoded)
```

In the next step LSTM-Autoencoder is being compile - we create a model using Keras library. It is compiled with Adam optimizer and MSE as loss function. They were chosen because they are most common and effective. To prevent overfitting EarlyStopping is included to stop training if validation loss doesn't improve in a chosen amount of consecutive epochs. Then the fit method trains the model, train data is passed as input and output because autoencoder has to reconstruct the input. Parameters such as batch size, epochs are also not fixed and can be modified. For example, increasing epochs may allow model to have more opportunities to learn from data, but it can cause overfitting. Validation data as parameter is used to evaluate performance on new data. It monitors validation loss. Next we are plotting training and validation loss during epochs. We visualize it using matplotlib. One of the most important points is to establish a correct threshold for the reconstruction error, which is difference between original and reconstructed data. We implemented a function for that purpose. We use precision recall curve function from scikit-learn library and pass labels and predicted scores. It returns precision(true positives/total positives), recall(true positives/actual positives), and thresholds derived from reconstruction errors. Then f1 score is calculated for each threshold. The function finds a threshold that maximizes f1 score out of all.

$$F1score = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

In the function plot anomalies autoencoder recreates data and reconstruction error is calculated and drawn. Threshold is shown on the graph as a red line and all anomalies are represented as red dots. AUC score is calculated to understand accuracy of the model. ROC curve is plotted.

Vladimir worked on loading, processing, and normalizing data. He created sequences, corresponding labels, split the data into training and validation sets and trained the model. Viktoriia

built encoder and decoder parts of the model, found reconstruction error and optimal threshold, plotted anomalies on the graph and ROC curve.

3.2.4 Conclusion on LSTM

As already mentioned, LSTM is a part of Tensorflow Keras libraries, however as to our knowledge, LSTM-Autoencoder can not be imported directly. We implemented it using methods from already existing libraries, and used precision recall curve function from sklearn library to calculate optimal threshold, which we didn't see in other works. The number of memory cells, layers, hidden layer size, and other hyperparameters of LSTM networks can all be adjusted to maximize performance. It depends on the complexity of task and the size of input data. The LSTM model has to be properly configured in order for it to be able to identify long-term dependencies in the data, which we tried to do. With some adjustments LSTM-Autoencoder can be integrated into the infrastructure.

3.3 Infrastructure creation (Viacheslav Ryzhkov, SE)

- Infrastructure provisioning: selecting a technology stack for developing services, exchanging information between them, storing data, building and delivering services, organising the environment to work in
- Metrics analysis orchestration: developing a service for scheduling analysers' work (model training or prediction based on the trained model), its distribution among analysers, and supplying data from sources for analysis
- Metrics management: organisation of storage of tracked metrics settings, as well as CRUD API for editing these settings.
- Organisation of output data storage: uploading analysed data to a separate DBMS

4 Conclusion

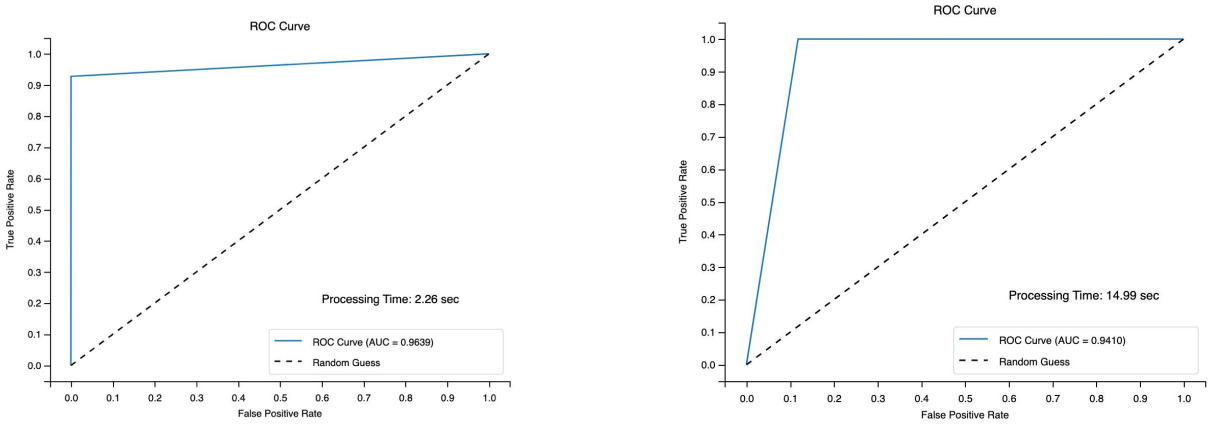
	HTTP				SMTP			
	AUC score	F1 score	Train time, seconds	Test time, seconds	AUC score	F1 score	Train time	Test time
sample size = 64	**0.93**	0.58	0.02	9.64	**0.85**	0.82	0.01	1.50
sample size = 256	**0.93**	0.41	0.04	12.46	**0.94**	0.94	0.04	2.26
sample size = 1024	**0.94**	0.42	0.11	14.99	**0.96**	0.95	0.09	2.26
LSTM autoencoder	**0.98**	0.73	2.82	32.02	**0.98**	0.70	0.79	8.27

Table 4.1: Comparison of AUC and Time for Different Algorithms

In order to estimate effectiveness of models, we have used AUC and F1 scores. AUC (Area Under Curve) is area under ROC (Receiver Operating Characteristic, shows the relationship between True Positive Rate and False Positive Rate for different classification thresholds), it helps to estimate effectiveness of the model in prediction/determination. The higher AUC score, the higher the AUC, the better. That is, if AUC < 0.5, it means that the model does not perform better than random selection, while values between 0.9 and 1 indicate that the model performs almost perfectly. F1 score indicates how well the model performs on the classification task, considering both accuracy and completeness. The F1 score ranges from 0 to 1, where 1 indicates a perfect model.

For synthetic tests we used KDD Cup 1999 Dataset, specifically a 10% dataset. From this dataset we took away all data that was related to HTTP and SMTP metrics, since they are relevant to our topic. For both models 10% of data has been used to train models and rest to test on. iForest model used threshold = 0.5 and tree_num = 100. LSTM used automatically counted threshold. When we tested our models on SMTP dataset, same number of features has been used for both models, but in case of the HTTP dataset, iForest used only 3 features: duration, src-bytes and dst-bytes, while LSTM utilised most of the them. As could be seen from Table 4.1, both iForest and LSTM autoencoder are able to detect anomalies with decent score. Best ROC curves of iForest could be seen in Figure 4.1 and for LSTM in figure 4.2. However, LSTM autoencoder test time is significantly higher, than iForest. On the other hand, LSTM autoencoder gives higher score, especially in case of the F1 score, when model was tested on the HTTP dataset.

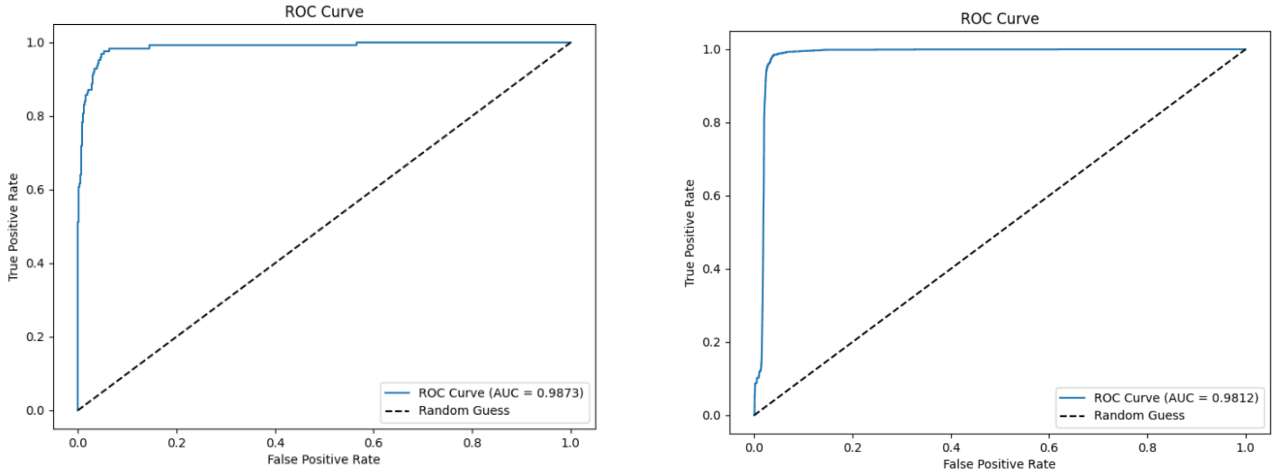
Comparison of ROC : iForest



(a) smtp (b) http

Figure 4.1: Comparison of ROC curves

Comparison of ROC : LSTM



(a) smtp (b) http

Figure 4.2: Comparison of ROC curves

To sum up, we can say that we have successfully made and tested 2 models on synthetic data, which have shown good results in tests and with minor modifications could be integrated into an anomaly detection infrastructure in the future. The main feature is that both models use completely different approaches for anomaly detection and are suitable for different conditions. For example, when there are no computational power constraints and there is a pronounced data drift, LSTM can be used. When you need to minimize the load on the system and at the same time data drift is minimal, iForest would be the ideal model.

References

[1] Sahand Hariri; Matias Carrasco Kind; Robert J Brunner. "Extended isolation forest". In: *arXiv preprint arXiv:1811.02141* (2018).

[2] Zhiguo Ding; Minrui Fei. "An anomaly detection approach based on isolation forest algorithm for streaming data using sliding window". In: *IFAC Proceedings Volumes, 46(20):12–17, 2013* (2013).

[3] Michael Heigl; Kumar Ashutosh Anand; Andreas Urmann; Dalibor Fiala; Martin Schramm; Robert Hable. "On the improvement of the isolation forest algorithm for outlier detection with streaming data". In: *Electronics, 10(13):1534, 2021* (2021).

[4] Hochreiter Sepp; Schmidhuber Jürgen. "Long Short-Term Memory". In: *Neural Computation* (1997).

[5] Kun Zhang; Wei Fani; Andrea Edwards; S Yu Philip Ke Wu. "Rs-forest: A rapid density estimator for streaming anomaly detection". In: *In 2014 IEEE International Conference on Data Mining, pages 600–609. IEEE, 2014* (2014).

[6] Sébastien Thomassey Kim Phuc Tran Hu Du Nguyen. "Anomaly detection using Long Short Term Memory Networks and its applications in Supply Chain Management". In: *Manufacturing Modelling, Management and Control - 9th MIM 2019* (2019).

[7] Swee Chuan Tan; Kai Ming Ting; Tony Fei Liu. "Fastanomalydetec- tion for streaming data". In: *In Twenty-Second International Joint Conference on Artificial Intelligence, 2011* (2011).

[8] Malhotra Pankaj; Vig Lovekesh; Shroff Gautam; Agarwal Puneet. "Long Short Term Memory Networks for Anomaly Detection in Time Series". In: *Proceedings, volume 89* (2015).

[9] Ergen Tolga; Kozat Suleyman Serdar. "Unsupervised Anomaly Detection With LSTM Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[10] Shukla Raj; Sengupta Shamik. "Scalable and Robust Outlier Detector using Hierarchical Clustering and Long Short-Term Memory (LSTM) Neural Network for the Internet of Things". In: *Internet of Things* (2020).

[11] Pankaj Malhotra; Anusha Ramakrishnan; Gaurangi Anand; Lovekesh Vig; Puneet Agarwal; Gautam Shroff. "LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection". In: *ArXiv* (2016).

[12] Fei Tony Liu; Kai Ming Ting. "Isolation-based Anomaly Detection". In: *In 2008 Eighth IEEE International Conference on Data Mining, pages 413–422. IEEE, 2008.* (2008).

[13] Fei Tony Liu; Kai Ming Ting; Zhi-Hua Zhou. "On Detecting Clustered Anomalies using SCi-Forest". In: *In Joint European Conference on Ma- chine Learning and Knowledge Discovery in Databases, pages 274–290.* (2010).