university of groningen / faculty of science and engineering

Stochastic token-generator models for simulating industrial message logs

Bachelor's Project Thesis

Stevan Wilts, s2771535, j.w.wilts@student.rug.nl,
Supervisor: Dr. L.R.B. Schomaker

Abstract: Many machine learning implementations rely on synthetic data, but there are few, if any, general solutions for generating data. This study explores a new method for stochastic token-generation in the context of industrial message logs, using finite-state machines. For each of three different data distributions 50.000 finite-state machines are created. These machines can generate token sequences of arbitrary lengths and are randomly generated within a bounded parameter space. The distributions of the generated token sequences are compared to the target distributions and the machine that generates the sequence with the smallest mean squared error to the target distribution is selected. To test the predictability of the selected datasets, four different neural networks are trained on each set. Each of these models performed equally with 80 – 92% accuracy, depending on the dataset, and only three percent better than they would perform by only repeating the last message code. This shows that there is not enough internal dependency in the data to make reliable predictions about the future. This technique can however be used to generate sequences that can be used in the field of machine-learning to learn to solve these specific sequential problems.

1 Introduction

Within the field of Artificial Intelligence, data generation is an increasingly popular subject. With the development of many machine learning algorithms a lack of adequate training data has become apparent. To solve this, engineers have developed methods to generate synthetic data. Synthetic data is information that is artificially manufactured rather than generated by real-world events. Synthetic data is created algorithmically, and it is used as a stand-in for test datasets of production or operational data, to validate mathematical models and, increasingly, to train machine learning models. (Laskowski, 2019 as cited in Popić et al., 2019).

There already are many methods of data generation, most of which use some machine learning algorithm. For instance, one study used a Long Short Term Memory network (or LSTM), together with a Mixture Density Network (Alzantot et al., 2017). Another study used nested Hidden Markov Model's (or HMM's) (Dahmen and Cook, 2019). Unfortunately, it appears that every problem needs the building of its own data generator, which is time-consuming and error-prone (Popić et al., 2019).

This study does not aim to solve this issue. Instead it explores another method of generating synthetic data by simulating the underlying process that generated the original data. This is done by generating a finite-state machine that generates data of which the distribution is close to the original distribution.

1.1 The context

This is done in the context of industrial machine message logs. Many, if not all, industrial machines produce message codes when certain events occur. The total range of these codes can be seen as a language with a relatively small lexicon. This study assumes a lexicon size of 1000-2000 words. Furthermore, in industrial machines, events can trigger other events. For instance, a shortage in oil can cause a motor overheat, or a faulty sensor can cause an arm to go out of position. This also means that there will be a causality between certain mes-

sage codes. This whole system can be modelled in a finite-state machine, where events trigger other events, and certain events trigger the creation of a message code. This is why this study uses a finite-state machine to generate message codes. For each industrial machine there exists a finite-state machine that completely describes the processes of the real machine. The properties of this finite-state machine are unknown, but a finite-state machine that produces message codes with a similar distribution might come close to the original model. Because this finite-state machine comes close to describing the underlying processes of the original machine, it might be a better data generator than more abstract machine learning algorithms.

1.2 The chosen distribution

In real industrial applications, the proper distribution of the data can be derived from the history of message codes. However, such data is not available for this study, so some distribution has to be chosen. Because the data describes a lexicon, the choice falls on a Zipf distribution. George Kingsley Zipf stated that, for the datasets he studied, the frequency of word usage approximated the following equation

$$r * f = C \tag{1.1}$$

in which r refers to the words rank and f to its frequency of occurence. (Zipf, 1949) Zipf called the formula rewritten as

$$f = C/r \tag{1.2}$$

a rank/frequency plot. It is effectively a power-law distribution, given by the equation

$$P(x) = C * x^{-\alpha} \tag{1.3}$$

(Newman, 2005). In Zipf's original paper, α was set to 1, giving the equation above. In this study three different values for α are used, which are 2, 3 and 4. The constant C is given by the following equation:

$$C = \frac{1}{\sum_{n=1}^{N}(n^{-\alpha})} \tag{1.4}$$

Where N is the number of elements in the dataset. This results in the total of all frequencies adding up to 1. Combining these parts gives the formula:

$$f(x, \alpha, N) = \frac{x^{-\alpha}}{\sum_{n=1}^{N}(n^{-\alpha})} \tag{1.5}$$

The Zipf distribution has shown to be a good fit for larger lexicons, such as the English language (Moreno-Sánchez et al., 2016). Therefore it might also be a proper distribution for the lexicon of message codes.

There is one aspect of the Zipf distribution that gives some problems with prediction. Because of the Zipf distribution of the data, the most common message code takes up a large proportion of the data, ranging from 60%, for the data with exponent $\alpha = 2$, to 92%, for the data with exponent $\alpha = 4$. This makes sense in real world applications: the most common message codes indicate that the machine is working correctly, or are minor warnings, which happens most of the time. However, in predicting, the more severe errors are the most interesting and useful to predict. Because the most severe errors generally occur the least often, the least common message codes are the most important. The problem with these distributions is that a model can get up to 92% accuracy by always returning the most common code, and possibly even higher by repeating the previous code. This last accuracy score is called the repetition rate of the dataset. This is not desirable, because this accuracy does not say anything about the quality of the model or the predictability of errors in industrial machines. To solve this problem, some models in this study are forced to change the representation of the data, so that the previous code cannot be repeated directly. This is done using an encoder-decoder model. The details of this approach will be explained in section 2.2.2. Section 3.3 will also focus on the recall score of the predictions. This is because the recall score shows how good the model is in prediction the less frequent message codes.

1.3 This research

This all leads to the research question:

Using finite-state machines generated through a stochastic process, is it possible to generate message logs that have a distribution that comes close to a Zipf distribution and that have enough internal dependency to make accurate predictions about the future?

Both the closeness and the accuracy of predictions are hard to quantify, because the closeness

must be compared to the target distribution and the accuracy must be compared to the repetition rate of the dataset. Therefore the assumption is made that by creating enough finite-state machines, the sequence created by the finite-state machine that is closest to the target machine can get close enough to the target distribution. The focus is therefore on the accuracy of the models on these sequences. To decide which finite-state machine comes closest to the actual machine, the mean squared error of the generated sequence to the target distribution is calculated. The Mean Squared Errors are presented in section 3.1. To assess the accuracy of predictions, four models are compared on each dataset. The hypothesis is that there is enough internal dependency in the data so that at least one of the models will have a significantly higher accuracy than the repetition rate of the dataset.

This study consists of two parts. The first part is about generating synthetic data using a finite-state machine. The method of generating this finite-state machine will be explained in detail and the characteristics of the resulting finite-state machine will also be shown. The second part is about training neural networks on data sequences generated by the finite-state machine. The details of the used neural networks will be shown, as well as the results. It should be noted that the model is trained on self-generated data. This data is not directly based on real events and the characteristics of the synthetic data are therefore very important. This study aims to show that it is possible to train a model for predicting machine message codes while only needing the underlying distribution of the codes.

2 Methods

This section will explain how the experiments are conducted. As the study consists of two parts, this section is also split up into two parts. Section 2.1 will explain how the finite-state machines and the message code sequences are generated. Section 2.2 will explain how these finite-state machines are evaluated through prediction of the message code sequences.

2.1 Generating Synthetic data

The first part of this study is about generating synthetic data using a finite-state machine. To narrow down the hypothesis space of possible finite-state machines, there are a number of limitations. Firstly, all finite-state machines have to be unordered trees. Secondly, all finite-state machines need to have between 1000 and 2000 leaves, because this study assume a lexicon size of 1000 to 2000 codes. Thirdly, there are limitations on the following parameters:

- **Maximum fanout:** The maximum number of children each node can have.
- **Maximum depth:** The maximum distance to the root node.
- **Lambda(λ):** The exponent of the exponential distribution.

The values of these parameters are chosen randomly for each tree, as will be explained later. However, the maximum fanout is always between 1 and 9, the maximum depth is always between 1 and 11 and λ is always between 0 and 50. Besides these limitations, there are two system characteristics that are modelled. The first one is the auto-recurrence of codes. The message codes can be seen as updates about the system state at a regular interval. Because of this, the same code would be given for as long as the system remains in that state. This should be modelled in the finite-state machine. The second characteristic is the possibility of complete system break-down. In real machines, very grave system errors can lead to a complete system break-down. This should also be modelled in the finite-state machine.

2.1.1 The trees

As stated in section 2.1, each tree has parameters for the maximum fanout and maximum depth. These parameters are used to bound the scale of the trees, because one of the limitations is the number of leaves. The third parameter λ is used for the probability distribution over the children of the nodes. When creating the tree, a random fanout between 1 and the maximum fanout is chosen for the root node, using the Java Random class, which uses the system time to seed a random generator.

(Hahn et al., 2003) This process is repeated recursively for all child nodes until there are no more child nodes left. The expansion of a branch also stops when the distance to the root node is equal to the maximum depth. The leaves of the tree are then numbered in a random order. Each leaf also gets a random auto-recurrence probability between 0 and 1. This is randomly chosen using the Java Random class. To model the possibility of complete system breakdown, an inverse relationship between occurrence of the message codes and gravity of the error is used. In other words, the least common message codes have the most serious consequences. Because there is no direct measure of the occurrence of message codes in the model itself, the auto-recurrence probability of the leaves is used. Each leaf with an auto-recurrence probability of less than 0.1 has a system-failure probability of 1 in 100.000 codes.

2.1.2 Generating codes

To generate a message code from a tree, the tree is traversed from the root node until one of the leaves is reached. Each node has a probability distribution over its child nodes. This distribution only uses λ as a hyperparameter, which is a parameter of the whole tree, and therefore the probability distribution is the same for each node. The probability distribution is an exponential distribution, given by the probability density function:

$$f(x, \lambda) = \lambda e^{-\lambda x} \tag{2.1}$$

over the interval $[0, 1)$. At each node a random draw using this distribution is done and the result is multiplied by the number of children of the node. The result is rounded down to the nearest integer and used as the index of the child to go to. This process is repeated until a leaf is reached. The number of the leaf is returned as the message code. At the leaf, the message code is repeated, using a random draw with the auto-recurrence probability of the leaf.

2.1.3 Experimental setup

To generate trees, a Monte Carlo simulation is used. For each tree, a random maximum fanout f, maximum depth d and exponential parameter λ are drawn such that

$$f, d \in \mathbb{N}, \quad f \in [1, 9], \quad d \in [1, 11]$$

$$\lambda \in \mathbb{R}, \quad \lambda \in (0, 50)$$

With these parameters the tree is generated. If the tree doesn't have between 1000 and 2000 leaves, it is discarded and a new tree is generated. With each tree, a dataset of 100.000 message codes is created. A histogram of this dataset is compared to the target distribution and the Mean Squared Error between the two distributions is calculated. The tree that creates the data with the lowest Mean Squared Error is considered closest to the actual machine and is therefore selected for the second part of the study.

2.2 Predicting message codes

The second part of this study is about predicting message codes using neural networks. Four different models are trained on three sets of synthetic data, generated through the process explained in section 2.1. As explained in section 1.3, one problem with the datasets is that a model can get an accuracy of approximately 90% by repeating the previous code. The proposed solution for this is using an encoder-decoder model. The encoder part of the model squashes the information from the input into a multidimensional vector. The decoder part predicts the output from this vector. To force the model to change the representation of the input, the encoded vector has fewer dimensions than the input. This is called a bottleneck. Because of this, the model cannot repeat the previous code directly. To test whether models with a bottleneck layer perform better than models without a bottleneck layer, two of the four models use a bottleneck, while the other two don't. The different models will be described in section 2.2.2. The preprocessing of the data will be described in section 2.2.1.

2.2.1 Preprocessing the data

From the first part of the study three datasets are selected. These three datasets have the lowest Mean Squared Error to the three target distributions. Because they are all modelled to a different target distribution, they have different characteristics. Dataset 1 is modelled to the least steep power function and therefore contains the most different message codes. In the set of 100.000 codes that is used, it has a total of 76 unique codes. Dataset 2

has a total of 18 unique codes. Dataset 3 is modelled to the steepest power function and has a total of 13 unique codes.

Although our data consists of integer values within a specific range, there is no mathematical relation between the numbers. Therefore, the data is categorical. To feed the categorical data to the models, the integer values are converted to a one-hot encoding, an array of zeros where only the index corresponding to the integer value contains a one. We use two different forms of input-output pairs. For the one-to-one mdels, all possible pairs of two subsequent codes are used as pairs of input and output. The model must predict the following code based on the current code. For the sequence-to-sequence models, we use sequences of five codes as input and output. The starting index of the output sequence is the starting of the input sequence plus one and therefore the output is the input shifted one place. The dataset is split in a train and test set by using the even pairs as train set and the odd pairs as test set. This is done because there might be higher order relations within the data, just as a real machine can have different life stages. In this way these relations are present in both sets.

2.2.2 The models

This study compares four different models for predicting message codes. All models use multiple layers of LSTM cells. The LSTM is a variation on the Recurrent Neural Network (or RNN). RNN cells feed their output both to the next layer and to themselves. Because of this, they can incorporate information from previous timesteps. RNN cells do however suffer from a vanishing gradient problem because of the recurrent link and it is therefore hard for RNN's to incorporate information from many timesteps ago. LSTM cells solve this problem by keeping an internal cell-state. This cell-state is updated with new input and can keep information from many timesteps before. This is very useful when dealing with sequential data, especially when information from far before the current input is still relevant, which is the case with our data.

The last layer of each model is a fully connected layer with a softmax activation function. This layer maps the prediction onto a one-hot encoded output.

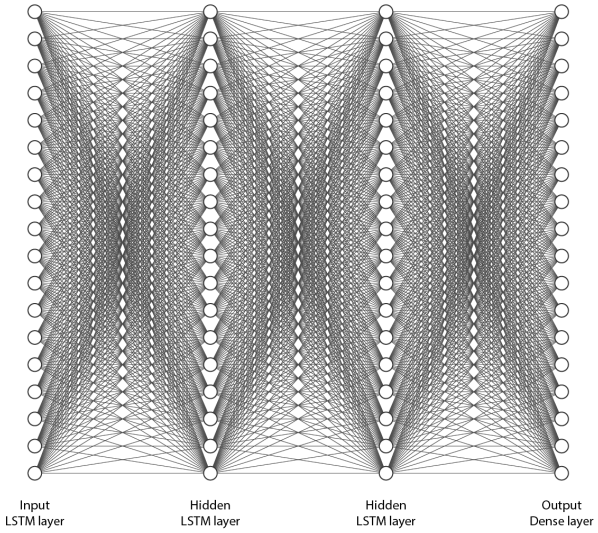

Figure 2.1: Visualization of model 1

Model 1 is a stacked LSTM model. It is a one-to-one model, containing three LSTM layers. Each layer has the same amount of cells as the dimensions of the input, and therefore there is no dimensionality reduction or expansion. A visualization of the higher model architecture can be seen in figure 2.1

Model 2 is a one-to-one encoder-decoder model. It is also a one-to-one model, but it contains five layers of LSTM cells. The encoder consists of three layers with a decreasing number of cells. A different number of cells is chosen for each dataset, because the input dimensionality of the dataset differs. The model for the first dataset has an input dimensionality and first layer of 76 cells and contains 30 and 10 cells in the second and third layer. The model for the second dataset has 18, 15 and 10 cells in the first three layers and the model for the third dataset has 13, 12 and 8 cells. The decoder consists of two cells with increasing dimensionality. The fourth layer has the same number of cells as the second layer and the fifth layer has as many cells as the output dimensionality (which is the same as the input dimensionality). Altogether, the model forms a diabolo structure, as can be seen in figure 2.2

Model 3 is a sequence-to-sequence encoder-decoder model with dimensionality reduction. The encoder consists of one LSTM layer with as many cells as the input dimensionality. The decoder con-

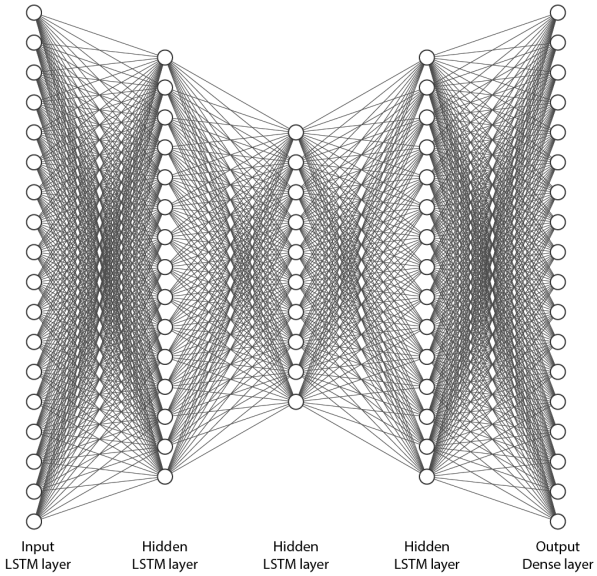

Figure 2.2: Visualization of model 2

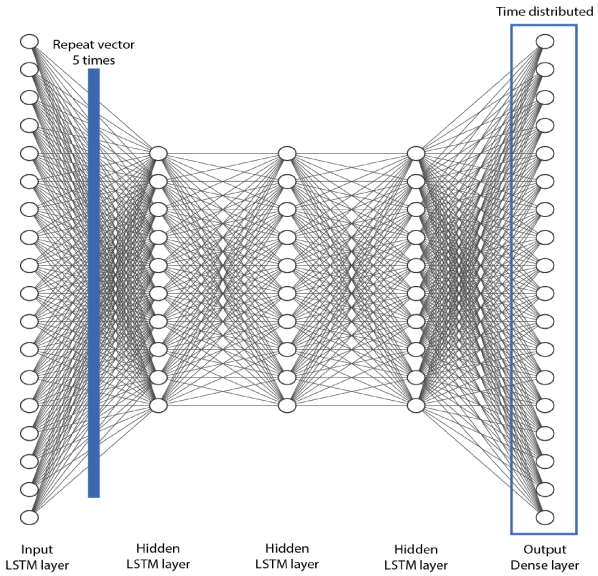

Figure 2.3: Visualization of model 3

sists of three LSTM layers with each 10 cells. The encoder and decoder are connected by a repeat-vector layer. This layer presents the encoded input five times to the decoder. The last fully connected layer for this model is time distributed, meaning that the softmax activation function is applied to each of the five vectors individually. A visualization of the higher model architecture can be seen in figure 2.3

Model 4 has the same structure as model 3, but it uses dimensionality expansion instead of reduction. The decoder of this model consists of three LSTM layers with each 80 cells.

2.2.3 Experimental setup

Each model is trained on all three datasets for 100 epochs. As the dataset is split into a train set and test set of equal size, both sets contain 50.000 codes. This makes it possible to generate 49.999 input output pairs. Of these pairs the last pair is dropped and the remaining 49.998 pairs are divided in batches of 26 pairs. The train set is then split into a train set of 45058 samples and a validation set of 4940 samples. Because models 3 and 4 use an input and output of 5 codes, only 49.995 pairs can be created. These pairs were all divided into batches of 33 pairs. The train set is split into a train set of 45045 samples and a validation set of 4950 samples.

All models use the Adam optimizer to update the weights. The Adam optimizer is aimed towards machine learning problems with large data sets and/or high-dimensional parameter spaces. It combines the advantages of the AdaGrad and RMSProp optimizers. (Kingma and Ba, 2014) The Adam optimizer is used in combination with the categorical cross-entropy loss, which is given by the formula

$$H(p,q) = -\sum_{c=1}^{C} p_i, c * log(q_i, c) \qquad (2.2)$$

where H(p,q) is the cross-entropy between p and q, C is the set of all categories c, and p_i,c and q_i,c are the binary values of p and q for class c.

After training, each model is tested on the full test set, which results in the test accuracy and loss. The full training pipeline can be seen in figure 2.4

3 Results

This section will present the results of the experiments. Like the previous section, this section is also split into two parts. Section 3.1 will present the generated finite-state machines and message code sequences. Section 3.2 will present the results of the evaluation of the finite-state machines.

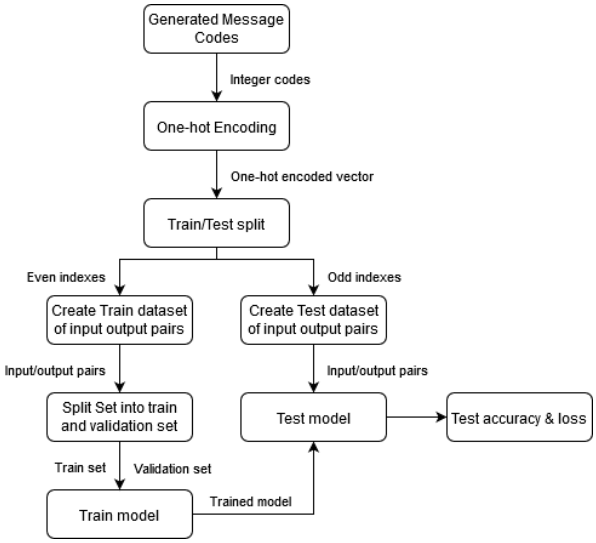

Figure 2.4: Training pipeline

3.1 The datasets

For each target distribution 50.000 finite-state machines are produced. This section will show the characteristics of the finite-state machines that come closest to each target distribution. For the first target distribution, the Zipf distribution with $\alpha = 2$, the closest dataset has a mean squared error to the target distribution of 16.0, and 76 unique codes in the set of 100.000 codes that is used. The finite-state machine that produced this dataset has a maximum depth of 6, a maximum fanout of 9 and a λ of 13.7. For the Zipf distribution with $\alpha = 3$, the closest dataset has a mean squared error to the target distribution of 1.27, and 18 unique codes in the set of 100.000 codes that is used. The finite-state machine that produced this dataset has a maximum depth of 6, a maximum fanout of 8 and a λ of 19.8. For the last target distribution, the Zipf distribution with $\alpha = 4$, the closest dataset has a mean squared error to the target distribution of 0.08, and 13 unique codes in the set of 100.000 codes that is used. The finite-state machine that produced this dataset has a maximum depth of 7, a maximum fanout of 8 and a λ of 23.2.

3.2 Performance of the models

The accuracy of each model on the test set of each dataset can be seen in table 3.1. As can be seen from the table, there are no significant differences in performance between the models. There are significant differences between the three datasets.

Table 3.1: Accuracy for each model on the test set of each dataset (in percentage).

Model	Dataset 1	Dataset 2	Dataset 3
1	82.3	87.4	91.9
2	82.6	87.4	91.9
3	81.1	87.0	91.9
4	80.5	86.3	91.7

However, when adjusted for the repetition rate, which is the baseline performance a model would get by just repeating the last code, the differences in score between the datasets are insignificant. The adjusted scores can be seen in table 3.2. The models score approximately 2.5% better than they would by just repeating the last code.

Table 3.2: The model accuracy adjusted for the repetition rate for each model on each dataset (in percentage)

Model	Dataset 1	Dataset 2	Dataset 3
1	102.8	102.7	103.0
2	103.6	102.7	102.9
3	101.7	102.1	102.9
4	101.0	101.5	102.7

3.3 Performance on dataset 2

This section will show more detailed results on the performance on dataset 2. This dataset is representative of the results on the other datasets. For each model both the unweighted and weighted precision, recall and F1 score are calculated. The F1 score is the harmonic mean of the precision and recall and is calculated by the formula:

$$F1(x) = 2 * \frac{precision(x) * recall(x)}{precision(x) + recall(x)} \tag{3.1}$$

Table 3.3 shows the unweighted metrics.

For the unweighted metrics, the metrics are calculated for each class (i.e. each possible message code.) separately and then averaged to get the results in table 3.3. This is called macro averaging. As can be seen from the table, all models score

very similarly on all metrics. It can also be seen that the models have a high precision and low recall. This means that most of the false negatives can be found in the classes with few instances in the test set, while most of the false positives can be found in the classes with many instances. This is because the number of false positives is equal to the number of false negatives (each false positive in one class must be a false negative in another), and the impact of one error is bigger in classes with fewer instances. In other words, the classes with many instances in the test set get predicted too often, while the classes with few instances get predicted not often enough.

Table 3.3: Average unweighted precision, recall and F1 score over all classes, for each model with dataset 2 as training set.

Model	Precision	Recall	F1 Score
1	0.75	0.25	0.26
2	0.90	0.22	0.23
3	0.65	0.21	0.22
4	0.51	0.29	0.31

The weighted metrics are shown in table 3.4. The calculations for the weighted metrics are very similar to the calculations of the unweighted metrics, the difference being that with the weighted metrics the weight of each class is determined by the number of occurrences in the dataset, whereas with the unweighted metrics all weights are equal. It should be noted that the weighted recall is actually the same as the accuracy. This is because to calculate the recall the number of true positives is divided by the number of occurrences of the class in the dataset. To calculate the weighted recall this is then multiplied again by the number of occurrences, leaving only the number of true positives. All true positives (i.e. the recall scores) are then added up and divided by the total number of samples, which is exactly the same as the accuracy. As can be seen, the weighted precision and recall are approximately the same.

Table 3.4: Average weighted precision, recall and F1 score over all categories, for each model with dataset 2 as training set.

Model	Precision	Recall	F1 Score
1	0.85	0.87	0.86
2	0.87	0.87	0.86
3	0.84	0.87	0.85
4	0.83	0.86	0.85

4 Discussion

The results in the previous section show no significant difference between the four models and only a slight improvement over a model that always repeats the last message code. Furthermore, section 3.3 shows that the recall of the models is very low, which means that the classes with many instances get predicted too often. Taking into account the fact that one class takes up more than 80% of the data, it is very likely that the model either predicts the previous code or the most common code.

Given the context, this is absolutely not desirable. The prediction of the error codes, especially more severe errors, is more important than prediction of message codes that indicate a normal state. Therefore it is desirable that the least common classes, which indicate the least common and therefore probably the most severe errors, have as few false negatives as possible. It is better to have a model that predicts the severe errors too often than a model that predicts them not often enough. As stated in section 1.2 the most important metric in this context is the recall and therefore the conclusion is that none of the models are good at predicting the message logs based on the synthetic data.

4.1 Evaluating the data

The main goal of this study is to create data that can be used to train a model so it can predict message codes accurately, using only the underlying distribution. In this prediction task, the model is completely dependent on the internal dependencies in the data. A model can only predict a message code consistently when there is some dependency involved that it can observe. Therefore, the evaluation of the model is also partially, if not almost entirely, an evaluation of the data. Looking at the model evaluation, no model is able to distinguish itself from the rest, even though the models are very different. This leads to the conclusion that there is not enough ground for the models to distinguish

themselves on, because there is no dependency in the data to learn.

Looking back at the process of creating the data, this is a very logical result. By using the tree structure, no dependency between the codes is put into the data, except for the chance of repetition. After the generation of each code the process is started all over again and no information from the past is used. The only dependency the model can get from this is the chance of repetition and the number of samples of each class in the data.

Therefore the answer to the research question is that the datasets created by the finite-state machines come close enough to the target distribution, but none of them contain enough internal dependency to make accurate predictions about the future. Each model is only slightly better than the repetition rate and none of them are able to distinguish themselves from the rest.

4.2 Future research

There are some things that can be done differently in future research. The main problem with this method of creating finite-state machines is that the tree structure does not bring any internal dependency in the data. After each code, the process is started over again and no information from the past is used. This can be solved by using a different architecture. The architecture that comes closest to the actual finite-state machine is that of a cyclic directed graph. This is however a much more complex architecture and therefore many more finite-state machines need to be generated to find one that fits the target distribution well enough. This would take up more resources, but it can also give better results.

There should also be more appropriate quantitative descriptors for the graphs. In this study only the parameters of the graph and the resulting distribution are used as quantitative descriptors. To make a better a priori assessment of the quality of the graph and of the complexity of the resulting dataset, these quantitative descriptors are needed. One specific part of the process in which this is useful is the comparison to the actual machine. In this study, only the data distributions are compared, but this might not be enough. An easy improvement would be to compare the transition matrix of the sequences in addition to the distribution. There are however many other quantitative measures possible.

Lastly, it might be more appropriate to test the sequence-to-sequence model using a convolutional neural network. Because the input consists of multiple time steps of one-hot encoded message codes, the input can also be represented by a matrix with the time steps as the columns and the classes as the rows. A convolutional neural network is more appropriate for this type of input and might lead to better results.

4.3 Impact in the field

The results of this study are not good enough to say that this technique for synthetic data generation can be used for prediction of real data from industrial machines. However, more finite-state machines and corresponding message code sequences can be generated with this technique. These sequences could be categorised in different levels of difficulty and used in the field of machine learning to try to solve these specific sequential problems. It might very well be possible to find better models to predict these sequences.

Furthermore, this technique still offers unique features, that can become really valuable when the method is improved enough to be useful in real predictions. Although most finite-state machines that are generated this way are not as good as one that would be directly modelled by a human, they are very easy and cheap to make and many different finite-state machines can be created and assessed in a small time span. This technique is also easy to adapt to different sequence prediction problems and requires very little knowledge about the actual machine that needs to be approximated and especially does not require any real data. This is a big deal, because generative, easy to implement data generation models are hard to find. Whether this technique offers enough value to be used in practice is for further research to decide.

References

M. Alzantot, S. Chakraborty, and M. Srivastava. Sensegen: A deep learning architecture for synthetic sensor data generation. In *2017 IEEE International Conference on Pervasive Comput-*

ing and Communications Workshops (PerCom Workshops), pages 188–193, March 2017. doi: 10.1109/PERCOMW.2017.7917555.

Jessamyn Dahmen and Diane Cook. Synsys: A synthetic data generation system for healthcare applications. *Sensors (Basel)*, 5, 2019.

Brian D Hahn, A Hahn, and Katherine M Malan. *Essential Java for Scientists and Engineers*. Elsevier, Burlington, MA, 2003. URL https://cds.cern.ch/record/1320455.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL http://arxiv.org/abs/1412.6980.

Nicole Laskowski. Synthetic data. 2019. URL https://searchcio.techtarget.com/definition/synthetic-data.

Isabel Moreno-Sánchez, Francesc Font-Clos, and Álvaro Corral. Large-scale analysis of zipf's law in english texts. *PLOS ONE*, 11(1):1–19, 01 2016. doi: 10.1371/journal.pone.0147073. URL https://doi.org/10.1371/journal.pone.0147073.

MEJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):323–351, 2005. doi: 10.1080/00107510500052444. URL http://www.tandfonline.com/doi/abs/10.1080/00107510500052444.

Srđan Popić, Ivan Velikic, Nikola Teslic, and Bogdan Pavkovic. Data generators: a short survey of techniques and use cases with focus on testing. 08 2019.

George Kingsley Zipf. *Human Behavior And The Principle Of Least Effort*. 1949.