FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Fault Analysis in Reactive Systems Using Log Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Fabian Exenberger, BSc.
Matrikelnummer 00951919

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Asst.-Prof. Dr. Ezio Bartocci
Mitwirkung: DI Dr. Cristinel Mateis
Dr. Dejan Ničković

Wien, 15. August 2019

Fabian Exenberger	Ezio Bartocci

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Fault Analysis in Reactive Systems Using Log Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Fabian Exenberger, BSc.
Registration Number 00951919

to the Faculty of Informatics

at the TU Wien

Advisor: Asst.-Prof. Dr. Ezio Bartocci
Assistance: DI Dr. Cristinel Mateis
Dr. Dejan Ničković

Vienna, 15th August, 2019

Fabian Exenberger | Ezio Bartocci

Erklärung zur Verfassung der Arbeit

Fabian Exenberger, BSc.
Wiedner Hauptstraße 54/8, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. August 2019

Fabian Exenberger

Danksagung

Ich möchte meine Dankbarkeit ausdrücken gegenüber Prof. Dr. Ezio Bartocci, dem Betreuer meiner Masterarbeit. Er hat sich immer wieder Zeit genommen, um mir wichtiges Feedback und Ratschläge zu geben, insbesondere zur Erhöhung der Wissenschaftlichkeit meiner Arbeit. Weiters hat er wichtige Ideen beigetragen. Ich möchte Dr. Cristinel Mateis und Dr. Dejan Ničković am Austrian Institute of Technology danken für wertvolle Ratschläge, Feedback und Unterstützung. Sie haben sich immer viel Zeit genommen um mir zu helfen, und auch sie haben wichtige Ideen zum Projekt beigetragen. Auch dem AIT selbst bin ich dankbar. Es hat mich während der Masterarbeit finanziell unterstützt, und – was noch wichtiger ist – hat mir einen Arbeitsplatz in unmittelbarer Nähe zu Dr. Mateis und Dr. Ničković zur Verfügung gestellt, sodass ich umso mehr von ihrem Rat profitieren konnte. Ebenfalls danken möchte ich dem Industriepartner des AITs für dieses Projekt und insbesondere unserer Kontaktperson in diesem Unternehmen, welche Daten und Feedback im Verlauf des Projekts geliefert haben. Zu guter Letzt möchte ich meinem Arbeitgeber willhaben danken, der mir erlaubt hat auf Bildungskarenz zu gehen, um mich ganz auf die Masterarbeit zu konzentrieren.

Acknowledgements

I would like to express my gratitude towards my thesis advisor Prof. Dr. Ezio Bartocci. He took time out of his schedule to give important feedback and advice, especially for increasing the scientific rigor of the thesis. And he contributed important ideas to this project. I would like to thank Dr. Cristinel Mateis and Dr. Dejan Ničković at the Austrian Institute of Technology for their invaluable advice, feedback and support. They were always generous with their time, and they too contributed important ideas to this project. My gratitude also goes to the Austrian Institute of Technology itself, which supported me financially while I was writing the thesis, and more importantly, provided me with a workspace close to Dr. Mateis and Dr. Ničković so that I could benefit from their advice even more. I would also like to thank AIT's industry partner on this project and in particular our contact person there for providing data and feedback over the course of the project. Furthermore I would like to thank my employer, willhaben, which allowed me to go on educational leave to concentrate on this thesis.

Kurzfassung

Die Identifizierung der Ursachen von Systemausfällen ist ein wichtiger Bestandteil der Entwicklung und Wartung von reaktiven Systemen. Aufgrund der Komplexität solcher Systeme kann dies eine sehr anspruchsvolle Aufgabe sein. Logdaten können bei der Analyse von Fehlern helfen, aber die Unstrukturiertheit dieser Daten sowie ihre schiere Menge machen eine manuelle Analyse oft unmöglich. Automatisierte Verfahren zur Analyse von Logdaten, um die Ursachen von Systemausfällen zu finden, sind daher notwendig. Ich entwickle vier verschiedene solche Methoden, die Hidden Markov Models, Supervised Learning, Automata Learning und eine Visualisierung verwenden, die ich einen "Weighted Sequence Tree"nenne. Ich bewerte die Leistung dieser Methoden anhand von drei verschiedenen Datensätzen, von denen zwei synthetische Datensätze mit unterschiedlichen Eigenschaften sind und einer reale Daten von einem Industriepartner umfasst.

Abstract

Identifying the causes of system failures is an important part of the development and maintenance of reactive systems. Due to the complexity of such systems, this can be a very challenging task. Log data can help analyze failures, but the unstructured nature of this data, as well as its sheer volume, often makes manual analysis infeasible. Automated methods for analyzing log data to find the causes of system failure are therefore necessary. I develop four different such methods, which respectively employ hidden Markov models, supervised learning, automata learning, and a visualization I call a "weighted sequence tree". I evaluate the performance of these methods on three different datasets, of which two are synthetic datasets with different properties, and one is real-life data from an industry partner.

Contents

CHAPTER 1

Introduction

In the development and maintenance of most complex reactive systems (e.g. interactive software), finding and removing errors takes up a large part of engineers' time. Finding errors is especially difficult when those tasked with doing so are not familiar with the internal structure of the system, for example in the case of legacy software. In such a situation, it is very helpful to have a method for narrowing down the potential causes of the error even without any knowledge about the program's internals, so that the engineer can focus on understanding only a smaller fragment of the program. In theory, log data, which is commonly used in software systems, can provide such information, and it is indeed often used for this purpose. But just reading the log output may not be enough to gain insight into the causes of an error, because of the following issues:

Cryptic error messages Sometimes the same lack of system specific knowledge that makes the program difficult to debug also makes the error messages difficult to interpret.

Proximate and ultimate causes The error message usually only tells us the proximate, but not the ultimate cause of the error message. For example, it may tells us that a program crashed in a particular location because of a division by zero (proximate cause), but the bug that should be fixed is that the divisor was set to zero in a completely different part of the program beforehand (ultimate cause). Even if the log contains information about the events that ultimately led to the error, it may not be possible for a human to see the connection to the error message in the mass of data.

Lack of information Finally, there may just not be enough information in the log data to find out much about an error even in principle.

Computer algorithms might be fruitfully applied to this problem roughly in the following way: We can find out which kinds of event often occur in the program before a certain

error happens, and given some assumptions or supplementary domain knowledge we can make statements about causal relations within the program, e.g. an error is caused by a certain action or event described by a preceding log statement, or by a combination of different actions or events occurring in a certain order. This can be useful for dealing with some, but not all, of the problems with a purely human analysis of logs I described above:

Cryptic error messages The log lines that have been found to often precede the error line could be less cryptic than the error message, and may shed some light on the nature of the error.

Proximate and ultimate causes By getting information about what lines (or combinations thereof) precede the error lines, we might gain insight into what combinations of circumstances cause the error, i.e. the ultimate causes of the error.

Lack of information This problem cannot be solved with any data analysis technique by itself. If the log data has this problem, the methods described in this thesis will fail.

There are many possible ways to implement this basic idea or variations thereof. But there is currently not a body of publicly available best practices engineers could draw from regarding which algorithms are fit for this purpose and in what way they should be applied. The goal of this thesis is to lay the foundation for such a body of knowledge. To this end, it describes four different methods for performing fault analysis using log data and evaluates their performance on three different kinds of log data – two types of synthetic data with different properties, and a dataset from industry.

CHAPTER 2

Background

The higher the complexity of a system, the more difficult it is to avoid errors in the construction of the system, and the more difficult it is to identify the cause of a failure. We can distinguish two types of complex system [HP85]:

Transformative systems generally take inputs and transform them to produce outputs. Sometimes they may ask for additional inputs or produce some outputs during the transformation.

Reactive systems "are repeatedly prompted by the outside world, and their role is to continuously respond to external inputs" [HP85].

Compared to transformative systems, developing reactive systems poses a variety of additional challenges. Of particular interest for this thesis is that of finding the cause of a system failure. In a transformative system, when we encounter such a failure, we can simply re-run the transformation with the same inputs, and observe how the error occurs. But in a reactive system, there are many different ways for a system to interact with its environment, so it is hard to identify the exact circumstances under which the failure occurred, and therefore hard to reproduce it. One way to mitigate this problem, commonly used in software systems, is to use log files. These are text files to which the system writes information about various inputs, outputs, and internal state information on an ongoing basis, as well as "warnings" about possible problems during executions and messages chronicling errors. Typically, each entry occupies a line in the log file, and contains a time stamp indicating when it was created as well as a human-readable message about an event that occurred in the system. Often it also contains a "severity", which indicates how problematic the logged event is. For example the scale might have the levels "info" (meaning the event is not problematic), "warning" (something odd has happened, and it may or may not be a problem) and "error" (something has definitely

```
03/03/2019 08:21:55 INFO    system started
03/03/2019 08:21:55 INFO    establishing network connection...
03/03/2019 08:21:56 INFO    connection established
03/03/2019 08:21:56 INFO    Requesting info from server...
03/03/2019 08:22:01 WARNING Request timed out, retrying...
03/03/2019 08:21:06 ERROR   Server could not be reached
...
```

Figure 2.1: *Example excerpt of a log file. Each line refers to a different event, and contains a timestamp, a severity level and a human-readable log message.*

gone wrong). Depending on the system, there can be additional fields. For an example of a log file, see Fig. 2.1.

In case of a failure, one can search the log entries preceding the error for clues about what may have caused it. And monitoring the log may alert operators to errors they may not otherwise have noticed or may have noticed only much later. However, the system only logs information it has been explicitly instructed to log by the system's programmers. Providing such instructions is time-consuming, and so there is typically a large portion of events in a complex system that is not reflected in the accompanying log, or that is reflected only in the form of cryptic messages that do not allow the reader to understand what is happening in the system. However, if all events are logged, this might also not be helpful, since the events relevant to a particular error would be hard to identify among the myriad of irrelevant ones, like needles in a haystack. Thus, while analyzing log data is often the only feasible way to find the causes of a reactive system failure, doing so "by hand" is often more art than science, and can be very time-consuming. Automating parts of this analysis could be quite helpful. Since often the same kind of error will occur many times, we can use statistical methods to find patterns in the logged events and state of the system leading up to occurrences of the error we are investigating. The data in the log may not be detailed enough to identify the root cause of an error definitively even with the best possible method of analysis, but even then we can use the identified patterns as a point of departure for further analysis – e.g. we can cause a "suspected" pattern to occur in the system and observe if the error also occurs. If so, we can observe the system in this configuration in detail (e.g. with a debugger) to find the exact mechanism behind the error. The events that are part of the pattern before an error can serve as clues about what parts of the system we should investigate to find the root cause of the error.

CHAPTER 3

Literature Review

While there are many papers on using machine learning to extract information from log data, there is little existing literature on using log data for fault analysis in particular.

Fronza et al. [FSS+13] describe an approach to predict failures based on log data using Support Vector Machines.

Sipos et al. [SFMW14] also use SVMs to predict system failures. Their data consists of log data, which features are extracted from, and service notifications (i.e. customer complaints) which is used as the source of labels. They use Multi-Instance Learning [DLLP97] to model the problem, so that the labeled instances are intervals of log statements (e.g. all logs from one week) that are labeled negative if there were no failures in the relevant interval according to the service notifications. They perform feature selection by training multiple L1 regularized classifiers on subsamples of the data, summing the weight of features across models, and using the features with the highest total weights in the final model.

Zhang et al. [ZXM+16] also use log data to predict system failures. They first cluster log statements with similar format and content. Then they divide the observations into epochs of fixed length. The frequency of each pattern in an epoch (specifically the "term frequency - inverse document frequency" [SB88], with the patterns used as terms and the epochs as documents) is then used as a feature of that epoch. The resulting data is fed to a Long Short-Term Memory neural network to predict failures. Some of the authors of this paper later filed a patent application [XZCN19] that appears to describe largely the same method, but extended with explanations for individual error predictions by showing "the reason for the prediction with feature weights via a local-faithful surrogate model for every selected time bin". Though the application document does not go into detail, this appears to refer to explainable machine learning methods such as SHAP [LL17].

Du et al. [DLZS17] use log data and an LSTM neural network for anomaly detection. In contrast to [ZXM+16] their system is designed to also detect unknown types of anomaly.

Fu et al. [FLWL09] use unstructured log data for anomaly detection. They attempt to cluster the free-form log messages by the log statement in the source code they originated from. Using training log sequences from normal operations, they first perform the aforementioned clustering and use the output to train a Finite State Automaton of the system. Additionally they use the timing data from the logs to model the performance characteristics of the system under normal operation. Using these models, they can detect anomalies in new log data.

Brown et al. [BTHN18] also perform anomaly detection using log data. They use a Recurrent Neural Network, which they augment with attention [BCB15] to aid interpretability.

Pitakrat et al. [POvHG18] attempt to enhance real-time failure predictions of software systems by leveraging knowledge of the system's architecture. They use information about the dependencies between subsystems to propagate predictions of failures of one subsystems to the dependent subsystems. The relationship between the subsystems and the propagation of failures is modeled via a Bayesian network. The necessary architectural knowledge is extracted automatically from monitoring data and execution traces of user requests. The described approach is agnostic about the way the initial predictions of failure for individual subsystems are performed; it only specifies how they are to be propagated through the system.

Nagaraj et al. [NKN12] present a system for diagnosing performance problems using log data. Log data from normal and performance-impaired periods or nodes are analyzed separately, and a Dependency Network [HCM+00] is learned for each. The divergences between the two models are analyzed and the most salient ones are presented to the user for further analysis.

There are a number of papers on the related topic of finding software bugs using not log data but more fine-grained information such as execution traces and/or the code of the program itself: Befrouei et al. [TWW16] use sequential pattern mining [ME10] to analyze concurrency bugs from execution traces. In [JM11], Jose et al. describe BugAssist, a tool that takes C-programs annotated with assertions and, using model checking, locates errors in the program leading to assertion violations. Ermis et al. [ESW12] introduce the concept of an "error invariant", and use it to localize faults in programs by minimizing error traces.

Wong et al. [WGL+16] survey the literature on software fault localization. The sections on machine learning and data mining-based techniques are of particular interest for the thesis.

Some companies such as Loom Systems [Sys] offer commercial services for fault analysis using log data, but they do not publish their methodologies. Many others (e.g. [Spl19, Sum19]) offer products in the related but distinct field of anomaly detection or predictive

analytics using machine learning on log data. None of them seem to have published their methodologies in detail. Nevertheless, the extent of commercial activity in this area indicates that it is of considerable practical relevance.

CHAPTER 4

Methods

I evaluate four different methods in this thesis. Log files typically record a sequence of events generated by the system. There is often a causality relation between events recorded in the log file – if an event e_1 causes another event e_2 in the system, e_1 will precede e_2 in the log file. I sometimes abuse the vocabulary and say that the log recording of e_1 causes the log recording of e_2.

4.1 Preprocessing

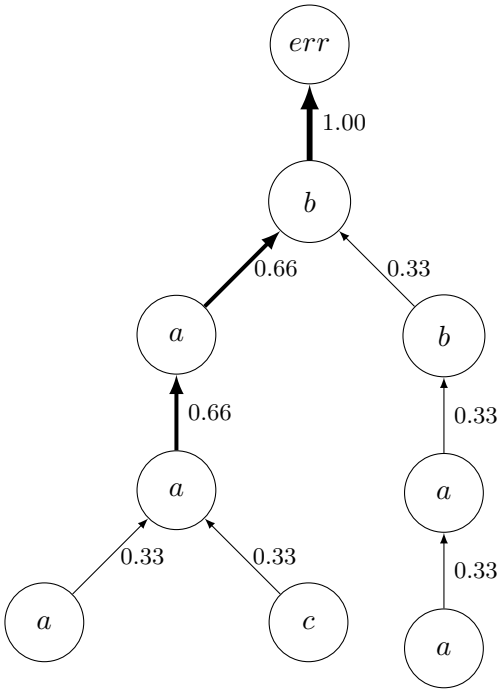
The methods expect as their input a set of one or more sequences of symbols, $L = \{\mathbf{T_1}, \mathbf{T_2}, ..., \mathbf{T_z}\}$, where each sequence $\mathbf{T_i} = (s_1, s_2, ..., s_r)$ represents the temporally ordered log output of an independent run of the underlying system. In practice, the raw log data might not be temporally ordered, e.g. because there are multiple concurrent threads in the system which write to the same log file, flushing write buffers at different times. Since a log line normally contains a timestamp indicating when it was produced, the lines can be brought into a temporal order. Then, each line is mapped to a symbol of some finite alphabet that can represent that line.

4.2 Weighted Sequence Tree

The goal of this method is to find out if there are sequences of events that frequently occur immediately before an error. From the preprocessed log data in L (c.f. Section 4.1)), for each sequence element with an error of interest errorX, we create a sequence consisting of that element together with k previous elements (or, if there are fewer than k previous elements, as many as there are). We now have a set of n sequences, each of them ending with errorX, which we turn into a weighted sequence tree. Fig 4.1 shows an example of such a tree being constructed from a set of sequences. The weighted sequence tree is defined as follows:

a, a, a, b, err
a, a, b, b, err
c, a, a, b, err

(a) *example sequences*



(b) *tree created from example sequences*

Figure 4.1: *weighted sequence tree example*

Definition 4.2.1. weighted sequence tree. For an alphabet Σ, and a set of sequences $L' = \{\mathbf{T_1}, \mathbf{T_2}, ..., \mathbf{T_n}\}$, where each sequence $\mathbf{T_i} \subset \Sigma^k$, and each ends with the same symbol s_e, the weighted sequence tree is the tree that fulfills the following conditions:

1. Each node is labeled with a symbol from Σ
2. If we follow the path from any node v to the root and form a sequence of symbols from the labels of the nodes we traverse, that sequence is a suffix of at least one of the sequences in L'. We say that this sequence is "the suffix starting at v".
3. The first edge on the path from any node v to the root has a weight that equals the proportion of sequences in L' with the suffix starting at v.

Once constructed, a weighted sequence tree can easily be visualized. By visually emphasizing frequent suffixes, we can see at a glance what sequences typically occur just before an error. There are some disadvantages of this method:

- It cannot distinguish between sequences that are only frequent before errors and ones that are always frequent. This is because the method considers only data in the neighborhood of errors and makes no attempt to compare and contrast it with data from other parts of the log.

- It cannot detect patterns that are interspersed with irrelevant messages. If, for example, a certain warning always occurs 2 lines before an error, but the statement in between is completely random (e.g. because a different thread is writing to the same log file at the same time), instances of this phenomenon with different in-between symbols will be treated as completely separate.

Nevertheless, it is a simple and computationally inexpensive way to explore the data and see if something "jumps out".

4.3 Supervised Learning

Supervised learning [JWHT13] is an area of machine learning which deals with the following problem: We are given a set of example observations drawn from some probability distribution P(the "training set"), each consisting of one or more "feature" variables $x_1, x_2, ..., x_n$ and one "label" y. Our task is to learn a function $f(x_1, x_2, ..., x_n)$ that maps the features to a predicted label $\hat{y}$. This function should then perform as well as possible on observations that are drawn from the same distribution P as the training set, but not part of the training set. Performance is measured in different ways depending on the problem at hand, but it is generally related to minimizing the difference between actual values of labels y and predicted values $\hat{y}$. Supervised learning can be divided into two main areas:

- Regression deals with data where the labels have quantitative (cardinally scaled) values. An example of such a label would be a person's age.
- Classification is concerned with data where each label falls into one of a finite number of categories, e.g. the label might refer to a person's country of residence.

Our labels will indicate whether or not a certain error occurred on a given line, and therefore ours is a classification problem. More specifically, it is a so-called binary classification problem, meaning that there are only two categories – the error either occurred or it did not.

In practice, supervised learning algorithms generally take the feature and label data as two separate inputs: The feature data is given in the so-called feature matrix, denoted by convention as $\mathbf{X}$, in which each row corresponds to an observation in the sample, and each column corresponds to a different feature. The second is the label vector, denoted as $\vec{y}$, which contains the label for a different observation on each row. For a binary classification problem such as ours, $\vec{y}$ is a binary vector – for example in our case, an observation has label $True$ if it corresponds to a line where $errorX$ occurred, and label $False$ otherwise.

The basic idea behind using supervised learning to explain errors is to learn a classifier model that, for each line in the log file, "predicts" whether or not the line contains a

given type of error based on the contents of some number of preceding lines. In contrast with many other applications of supervised learning, here we are not primarily interested in the prediction itself (we already know the true value, since it is part of the data), but in the parameters of the learned model, which give us information about the effect of different features (i.e. different kinds of preceding lines). To learn the causal effect of each feature, I estimate several different models following the literature on causal inference and the back-door criterion (see Subsection 4.3.3).

4.3.1 Combining Lines Into the Feature Matrix

How then do we construct our feature matrix and label vector from the preprocessed log data described in Section 4.1? As mentioned, the preprocessed log data consists of a set of sequences $L = \{\mathbf{T_1}, \mathbf{T_2}, ..., \mathbf{T_z}\}$. I will first describe how to generate $\mathbf{X}$ and $\vec{y}$ from a single sequence of symbols $\mathbf{T} = (T_1, T_2, ..., T_n)$ of length n, before describing how to handle multiple sequences. Each sequence element T_i indicates what type of event was logged at the corresponding ith line of the log sequence. To create the feature matrix $\mathbf{X}$, for each of its rows we need to combine multiple elements of T, since we want to estimate the effects of preceding lines at different distances from the line in question. When we are discussing how the probability of an error on each line is affected by the line that comes some given number of lines before it, I will call that distance between the lines its "shift". For example, if an error of interest errorX occurs at line k then the line at shift i w.r.t. errorX is the line $k - i$. If we want to include the effects of lines with shift j to lines up to and including shift $j + k$ in the model (where $j > 0, k \geq 0, j + k < n$), we have to transform the sequence T as follows:

$$\mathbf{X'} = \begin{pmatrix} T_1 & T_2 & ... & T_{k+1} \\ T_2 & T_3 & ... & T_{k+2} \\ ... & ... & ... & ... \\ T_{n-j-k} & T_{n-j-k+1} & ... & T_{n-j} \end{pmatrix}$$

$\mathbf{X'}$ has $k + 1$ columns. Each of them corresponds to a different shift value - the ith column maps to a shift of $j + k + 1 - i$. Each row corresponds to a different observation. Nevertheless, the matrix does not have n elements as the symbol sequence T it was generated from, since for the first j+k elements of T there are not enough previous values to generate a matrix row from. Hence $\mathbf{X'}$ only has $n - j - k$ rows, and its ith row maps to T_{i+j+k}. Since it can be expected that $j + k << |\mathbf{T}|$, we only lose a minuscule fraction of the data to this issue. The resulting feature matrix consists entirely of categorical variables (i.e. variables that can take on one of a finite number of values), with the values for each column / variable showing the event at a different "shift" from the line indicated by the row index. Most classifiers cannot use categorical features with more than two values directly. Instead, each such feature needs to be converted into one or more binary "dummy" features in the following way: For a feature x_f which can take on c different categories, c binary features are created. The ith of these binary features then indicates if the original feature x_f took on the value of the ith category. For many classifiers, it is

important that no feature be a linear combination of any other features. With a feature for each category, this condition is violated, since each is $True$ if and only if all the other binary features are $False$. To fulfill the condition, we can drop one of the binary variables from the feature matrix. However, for the classifier we will be using (regularized logistic regression) this is not necessary [ÖA00]. The entire procedure is generally known as dummy encoding or one-hot encoding. Performing dummy encoding on $\mathbf{X}'$ results in our final feature matrix $\mathbf{X}$. Constructing the label vector $\vec{y}$ is considerably simpler: We check if the error of interest errorX occurred at each index, shifted so that the indices of $\vec{y}$ fit together with the row indices of $\mathbf{X}$:

$$\vec{y} = \begin{pmatrix} T_{1+j+k} == errorX \\ T_{2+j+k} == errorX \\ ... \\ T_n == errorX \end{pmatrix}$$

In Fig. 4.2, you can see an example of the transformation procedure described above.

So far we have discussed how to transform one sequence into feature matrix and label vector. For multiple sequences, the overall feature matrix is simply the concatenation of the feature matrices of the individual sequences, and analogously the overall label vector is the concatenation of the individual label vectors. The order in which they are concatenated is arbitrary, as long as the order for the matrices is the same as the one for the vectors, so that the ith row of the feature matrix still corresponds to the same observation as element y_i of the label vector.

4.3.2 Choosing a Classifier

In principle, many different classification algorithms could be used. Ideally, the chosen algorithm would make highly accurate predictions, giving us confidence in the learned coefficients, while also allowing an easy interpretation of said coefficients. In the supervised learning part of this thesis, I focus on logistic regression. This and other types of linear classifier yield easily interpretable coefficients. However, they cannot completely capture the causal structure in situations where an error is caused by a combination of different events – e.g. an error becomes very likely if event A occurs directly after event B, but the probability of an error does not change if only one of these events occurs. Depending on the data, in most cases there will still be a correlation between each of the events and the error (since each of them occurring is a necessary condition for both of them occurring), so the learned coefficients will suggest a causal connection between A and the error as well as B and the error, but the specific interaction causing the error will not be captured. New features could be generated as polynomial combinations of existing features. However, there might be hundreds of different kinds of log statements in a single dataset (as in our industry data), which together with some number of previous lines means we already have thousands of features. Generating features for all interactions of two preexisting features would therefore lead to millions of features, and even more for combinations of more different features. This would take up a lot

Message Code
info_1
info_2
warning_1
error_of_interest
info_1
...

(a) *Preprocessed log data*

$\mathbf{X}'$			$\vec{y}$
Message Code shift 3	Message Code shift 2	Message Code shift 1	
info_1	info_2	warning_1	True
info_2	warning_1	error_of_interest	False
...	...	...	...

(b) *matrix* $\mathbf{X}'$ *and label vector* $\vec{y}$ *generated from (a)*

err._of_i. shift 3	...	info_1 shift 2	err._of_i. shift 1	warning_1 shift 1	info_2 shift 1	info_1 shift 1
False	...	False	False	True	False	False
False	...	False	True	False	False	False
...	...	...	...	...	...	...

(c) *Final feature matrix* $\mathbf{X}$ *created by dummy-encoding* $\mathbf{X}'$ *from (b)*

Figure 4.2: *Example for transformation of preprocessed log data into feature matrix and label vector*

of computational resources. Nonlinear models such as gradient boosted trees or neural networks are better suited to capturing interaction effects. However, such models are notoriously hard to interpret [LWLZ17]. They can nevertheless be useful as benchmarks for more easily interpretable models by comparing against their predictive performances. And they can be used in combination with tools for explainable machine learning such as SHAP [LL17] to provide explanations for individual occurrences of an error.

4.3.3 Inferring Causal Effects

I am using the coefficients of logistic regression models to gain information about the causal effects of certain events. However, without additional assumptions, a causal interpretation is not necessarily correct. I use Pearl's Structural Causal Model (SCM) [Pea09] to identify the right logistic regression equations for estimating causal effects.

Why is it that we cannot use logistic regression alone to estimate causal effects? To review, the general form of a logistic regression is:

$$\ell = \beta_0 + \beta_1 * X_1 + ... + \beta_m * X_m \tag{4.1}$$

where $X_1, ..., X_m$ are the values of the different features, β_i is the coefficient for feature i, and ℓ are the log odds for the label being positive given the feature values. The odds o, which we can of course easily get from the log odds, can further be converted to a probability by using the formula

$$p = \frac{o}{o+1} \tag{4.2}$$

Intuitively, we can express the odds as a pair of numbers and say that something has odds of e.g. 5:1 or "a million to one". We can also compare the likelihood of different events using the ratio of their odds o_1/o_2.

To interpret the influence of different features, we must understand the meaning of their respective coefficient values. These values state how much the log odds of a positive label change when we change the corresponding feature by 1. Equivalently, a coefficient value β_i is the log of the odds ratios for a positive label when we change X_i by 1. A large magnitude of coefficient β_i therefore means a strong association between X_i and the label. However it does not necessarily mean that X_i has any causal effect on the label. And conversely, a near-zero $|\beta_i|$ does not necessarily imply the absence of a causal effect between the two variables. There are several phenomena we need to account for before we can make such causal claims.

Reverse Causation Instead of X_1 causing Y, Y might be causing X_1. However, in our case, the features are events from log statements preceding Y, and since effects cannot precede causes, we can rule out reverse causation.

Confounding Bias There might be a third variable, Z, causing both X_1 and Y, while there is no direct causal relationship between X_1 and Y. If this variable is omitted from the model, $|\beta_1|$ will be increased and might lead us to mistakenly assume a

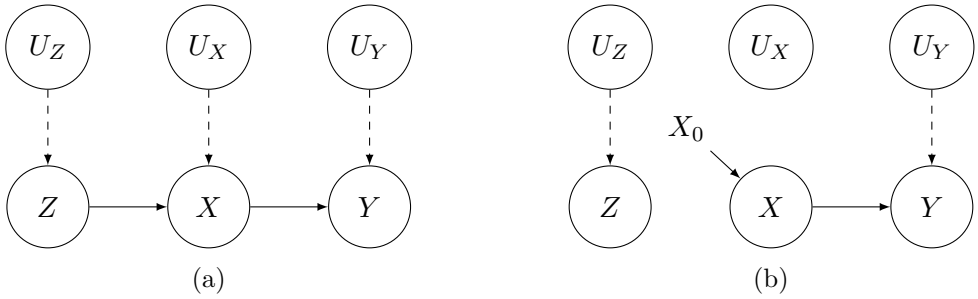


Figure 4.3: *DAGs depicting an intervention (adapted from [Pea09]). (a): The DAG corresponding to the structural model of Eq.* (4.3). *(b): The DAG corresponding to a modified model representing the intervention* $do(X = x_0)$

causal relationship between X_1 and Y. To mitigate this problem, we must include potential common causes in the regression. In our case, these are events preceding both X_1 and Y. However, there might be important events in the system that are not reflected in any log statements, and these unobserved variables could still bias the results.

Mediation There might be a third variable M, where X_i causes M and M causes Y. If M is also included in the model, and there is no direct causal effect of X_i on M, the expected value of β_i is 0, even though there is a causal effect. This is because β_i indicates the association between X_i and Y, all else – including M – being equal. But the causal effect is defined as what happens if we intervene and change X_i, and in that case all else is not equal, since this affects M. To account for this phenomenon, we should exclude potential mediators from the regression equation. In general, this might conflict with the prescription to include potential common causes, but luckily in our case the two sets are disjoint - while potential common causes are events that occur before X_i and Y, potential mediators happen in between the two.

We can use the Structural Causal Model [Pea09] as a framework to show more rigorously what controls we must include in the regression to correctly estimate causal effects. In the SCM, we start with a directed acyclic graph (DAG) depicting the possible causal relationships between different variables – for example, in Fig. 4.3(a), Z might influence X which might influence Y. The errors only indicate that a causal relationship is possible, not that there definitely is such a relationship. The absence of an arrow, on the other hand, means that there definitely is no causal relationship. We can state the relationships in the model by using equations:

$$\begin{aligned} z &= f_Z(u_Z) \\ x &= f_X(z, u_X) \\ y &= f_Y(x, u_y) \end{aligned} \tag{4.3}$$

A causal effect is defined via the so-called do-operator $do(x)$, which "simulates physical interventions by deleting certain functions from the model, replacing them by a constant $X = x$". For example, to simulate an intervention which sets $X = x_0$, we replace the equation for x in Eq. (4.3) with $x = x_0$. The graphical representation of this model can be seen in Fig. 4.3(b). The causal effect can then be obtained from the controlled distribution function

$$P(Y = y|do(x)) = \sum_z P(z, y|do(x))$$

In our case, both the label and the features are binary, so we need only compare the probabilities of $Y = True$ for setting $X = True$ vs. setting $X = False$ But can we estimate such causal relationships from observational data? It depends on our causal assumptions as depicted in our DAG. If we can find a set of covariates that fulfills the so-called back-door criterion with respect to X and Y, comparing treated and untreated data points having the same values for the covariates gives us the correct treatment effect. We call such a set of covariates an admissible set. The back-door criterion is defined as follows [Pea09]:

Definition 4.3.1. The back-door criterion. A set S is admissible for adjustment if two conditions hold:

1. No element of S is a descendant of X
2. The elements of S "block" all "back-door" paths from X to Y, namely all paths that end with an arrow pointing to X

Luckily, such an admissible set is available to us. To see this, we must look at the DAG depicting the possible relationships between our variables (Fig. 4.4). Recall that it is the absence of an arrow that makes a definitive claim, namely that there is no causal effect between two variables. I have excluded arrows that would imply effects preceding causes. In a sense, variables of the same shift value also affect each other, since if one of them is True, all others must be False – after all, if a log statement is of one type, it cannot be of another. Since this is always trivially true and does not change the choice of regression model, I have omitted these arrows as well. Given the resulting DAG, for each variable X_i, an admissible set is the set of all variables that come before X_i, i.e. all variables with a higher shift value than X_i. Note that the DAG does not show any unobserved variables. By definition, I cannot account for such variables, and if there are any that significantly and directly affect both a feature and the label, this will bias the estimate of the causal influence for the feature upwards. The reliability of the estimates therefore depends on important events consistently being logged.

Given all these considerations, we can estimate the causal effects of different events at different lines the following way: Say for $line_i$ at position i, we want to estimate the effects of various possible events at $line_{i-j}$. Then we learn a model that includes events at $line_{i-j-k}$ to $line_{i-j}$. The learned coefficients for $line_{i-k}$ are then a good causal

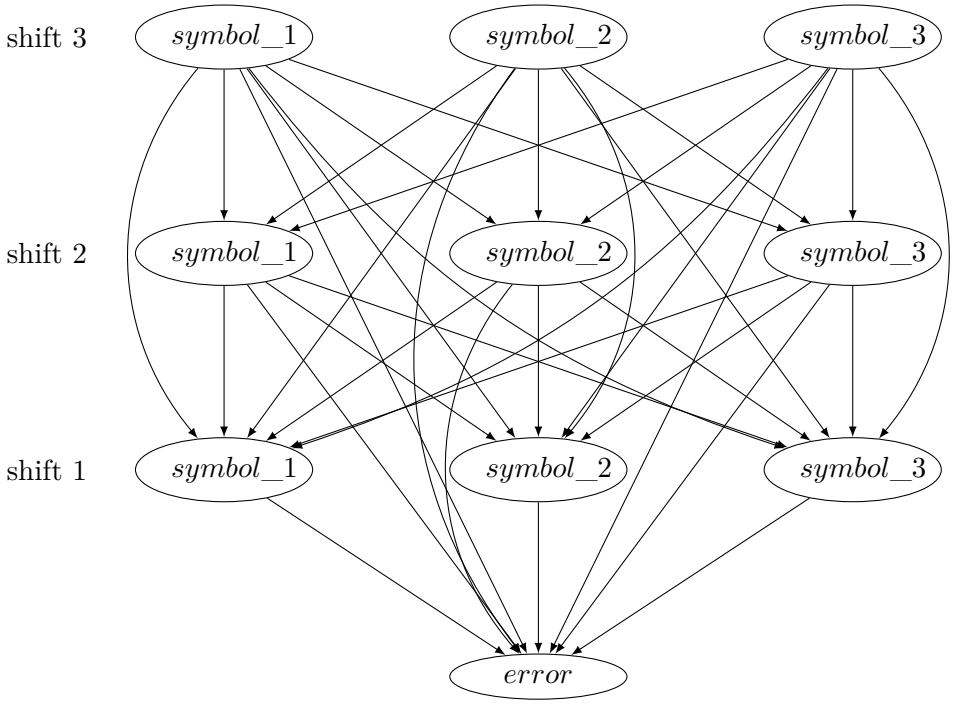


Figure 4.4: *DAG showing the possible causal relationships between features and label for a hypothetical dataset with three different message codes and only looking at variables with shift values 1-3. A variable can potentially affect all variables with a lower shift value, but not ones with a higher shift value.*

estimate – they do not include the potential mediators in lines $i-j$ to i, and they include potential confounders from the previous lines. The classifier we are using learns good values of some parameters for us, but since k is already set when we start the learning process, we cannot find the optimal value of k the same way. In machine learning, parameters that need to be set before training, as opposed to be learned during training, are called hyperparameters. Finding the optimal values of hyperparameters is called "hyperparameter tuning". During tuning, many models with different values for the hyperparameters are learned and compared using some numeric optimization criterion. To tune k, we therefore need such an optimization criterion. Since the ratio of positive to negative examples is very small, accuracy is a misleading metric of model performance – we could get a very high accuracy by simply predicting a negative label every time. Instead I use a metric called "average precision" (AP), which does not suffer from this issue. AP is defined in terms of two other metrics, precision and recall, which in turn are defined as follows:

$$precision = \frac{true\_positives}{true\_positives + false\_positives} \tag{4.4}$$

$$recall = \frac{true\_positives}{true\_positives + false\_negatives} \tag{4.5}$$

The classifier itself only outputs a probability for each sample being positive. We need a threshold probability above which a prediction is classified as positive to get values for precision and recall of a classifier on some sample. There is a trade-off between precision and recall – by increasing the threshold, we can increase precision, but lower recall. We can plot precision $p(r)$ as a function of recall r. By taking the integral of this function over the interval from $r = 0$ to $r = 1$, we obtain the average precision value:

$$AP = \int_0^1 p(r)dr \tag{4.6}$$

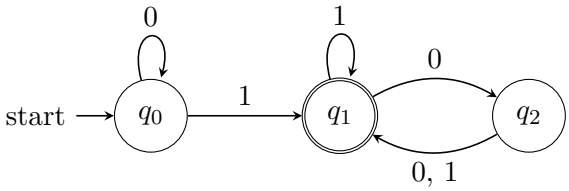
The maximum value of AP is 1, while the expected value for a dummy classifier predicting the same probability for all observations or ranking them randomly would have an AP close to the proportion of positive observations in the sample [Bes15]. An important problem in supervised learning is overfitting, the tendency of models with many parameters to learn idiosyncratic characteristics of the training data which do not generalize to additional data. A standard remedy, known as regularization, is to penalize large parameter magnitudes during learning. The penalty is commonly either linear ("lasso regression") or quadratic ("ridge regression") in the size of each parameter. In both cases, there is a coefficient that adjusts the strength of regularization. I employ regularization to combat overfitting, and I treat both the choice between lasso and ridge regularization and the size of the coefficient as a hyperparameter. To estimate the causal effects of n previous lines, I simply estimate n separate models of the type described above, with j varying from 1 to n. To make the coefficients from different lines comparable, I need to use the same hyperparameters for the different models. I therefore tune the hyperparameters for all these models together, and try to optimize the average average precision (sic, meaning the average of the different average precision values) over the different models. This leaves the question of how to choose n. I use the raw odds ratios between error occurrence and events at different shifts as determined by a Fisher exact test to decide. These odds ratios are upper bounds on the sizes of the causal effects, since they are not adjusted for either mediators or confounders. So if at some distance from $line_i$, the coefficients for all event types are either insignificant or just too small to matter in practice, we can assume that events at this distance have no or negligible causal effect on Y. Since calculating the odds ratio is computationally much cheaper than estimating full logistic regression models, we can do this test for a reasonably large number of previous lines. If we do this for e.g. 30 lines and conclude that there are potential causal effects in lines at shift 1 to 20, but not from 21 to 30, we can decide to estimate logistic regression models only for lines 1 to 20. However, this rests on the assumption that the strength of the causal connection between lines decreases somewhat monotonically with their distance to each other. In theory, there might be no causal effect of lines $i - k$ to $i - k - 1000$, but a strong causal effect of line $i - k - 1001$.

$$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

$\delta =$

	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

(a)



(b)

Figure 4.5: *Example of a deterministic finite automaton. (a): Definition. (b): Visualization.*

4.4 Automata Learning

Deterministic finite automata (DFAs) are mathematical constructs that have different states which they transition between depending on the inputs which they receive. There are clear parallels between DFAs and real-world reactive systems, and so in the approach described in this section, I use the former to model the latter. I interpret the logged events as inputs, while the states are inferred using an automata learning algorithm based on the minimum description length principle.

4.4.1 Deterministic Finite Automata

Definition 4.4.1. Deterministic Finite Automaton. A Deterministic Finite Automaton is a 5-tuple, $(Q, \Sigma, \delta, q_0, A)$ where

1. Q is a finite set of states
2. Σ is the alphabet, a finite set of input symbols
3. $\delta : Q \times \Sigma \to Q$ is the transition function
4. $q_0 \in Q$ is the initial state
5. $A \subseteq Q$ is the set of accept states

The automaton starts in state q_0 and reads from a sequence of input symbols $\in \Sigma$. When it consumes a symbol $a \in \Sigma$ while in state $q \in Q$, it changes to state $\delta(q, a)$. It then proceeds with the next symbol in the sequence. When the complete input sequence has been consumed, if the current (and last) state is an element of the set of accept states A, the automaton is said to accept the input sequence. See Fig 4.5 for an example of a DFA.

4.4.2 Generating Automata from Data

Now that we have established what a DFA is, how can we use our log data to model the underlying system as one? The basic idea is that we have a black box system (the "system under learning" / SUL), for which we only know some sequences of symbols we model as inputs to the automaton). Our goal is to learn an automaton as a model for the internal structure of the black box, so that we can find out what sequences of symbols can occur in the system, and which lead to an error. In the learned automaton, sequences that occur are modeled as ones that are accepted, and an error is simply a state transition that has the error of interest as the input symbol.

The task of generating automata from data is known as automata learning [SHM11]. There are different subfields: In active learning, we have access to the system under learning during the learning process, and can therefore test hypotheses about the system – we form a hypothesis in the form of an automaton, test if the real system behaves like the automaton, and if it does not, we refine the hypothesis and test again. Passive learning, in contrast, concerns the case where we have data from the system under learning, but cannot test hypotheses on the system under learning for additional information. Within passive learning, another distinction is whether we have positive and negative examples or only positive examples. A positive example in this context is a sequence of symbols that can occur in our SUL, and hence one that should be accepted by our automaton, while negative sequence cannot occur and should be rejected. We must use passive learning with positive examples only, which poses some additional challenges compared to other kinds of automata learning. Mainly, there are many different automata that accept all our examples, and not all of them are useful. For example, an automaton with one state which is both start state and accept state and has self loops for all symbols will accept any possible symbol sequence from the alphabet, and thus will accept all our examples. Since this is always the case, clearly this tells us nothing about the SUL. So we want to find an automaton that not only accepts all positive examples and symbol sequences that the SUL could produce but that are not part of the example set, but also rejects sequences that cannot be produced by the SUL. We cannot fulfill the second two conditions with 100% accuracy (this would be essentially equivalent to solving the problem of induction), but we can try to do as good a job as possible. One way to do this is to use the principle of minimum description length.

4.4.3 Minimum Description Length

Minimum description length (MDL) [Ris83, GMP05] is a general principle for model selection, that is for deciding among competing explanations for data given limited observations. It starts from the insight that any regularity in the data can be used to compress it. It then equates finding regularity with learning, so that the more we can compress some data, the more we have learned about it. Compressing data means to describe it in a shortened way that enables us to restore the original data. The result therefore also must include a description of how to accomplish this decompression step. Thus, to compress data D with length $L(D)$, we first encode a hypothesis H in the set

of considered hypotheses $\mathcal{H}$. We then encode the data with the help of H. The overall length is the sum of the two encodings: $L(D) = L(H) + L(D|H)$. And, as the name "minimum description length" suggests, the best hypothesis out of $\mathcal{H}$ is the one with which the data can be described with the shortest length, i.e. the one which minimizes $L(H)+L(D|H)$. MDL is closely related to Occam's razor – when two models fit the data equally well, MDL will choose the simpler one. This is also a built-in protection against overfitting. For a description of how to use MDL in the context of passive automata learning, see [dlH10].

4.4.4 Implementation

To accomplish the learning task laid out above, I use the automata learning software Learnlib [IHS15]. Specifically, I use the included BlueFringeMDLDFA algorithm, which utilizes the MDL principle for passively learning DFAs from positive examples. Learnlib's implementation is based on [dlH10]. The set of sequences L we obtain after preprocessing (c.f. Section 4.1) can be used without any further transformations.

4.5 Hidden Markov Model

Another method I explore is using the log data to learn a hidden Markov model. This gives us insight into patterns occurring before an error. In contrast with automata learning, it includes information about probabilities, allowing us to focus on the most prominent patterns.

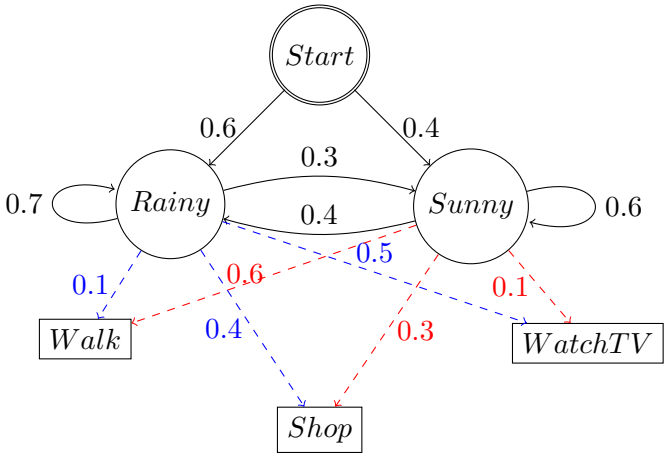
Definition 4.5.1. Hidden Markov Model. A hidden Markov model is a 5-tuple consisting of the following components:

1. a set of states $Q = q_1, q_2, ..., q_N$
2. a transition probability matrix A_{ij}, where element a_{ij} represents the probability of moving from state q_i to state q_j
3. a set of observation sequences, with each observation drawn from alphabet $\Sigma = s_1, s_2, ..., s_s$
4. an emission probability matrix E_{ij}, where element e_{ij} represents the probability of emitting symbol s_j while in state q_i
5. an initial probability distribution over the hidden states, $\pi = \pi_1, \pi_2, ..., \pi_N$, where π_i represents the probability that the process will start in state i

The parameter learning task for HMMs is to find the optimal transition and emission probability matrices as well as the initial probability distribution, given a set of observations, an alphabet Σ, and the desired number of hidden states N. Maximum likelihood is usually used as the optimization criterion, meaning that we attempt to find

$$Q = \{Rainy, Sunny\}$$
$$\mathbf{A} = \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix}$$
$$\Sigma = \{Walk, Shop, WatchTV\}$$
$$\mathbf{E} = \begin{pmatrix} 0.1 & 0.4 & 0.5 \\ 0.6 & 0.3 & 0.1 \end{pmatrix}$$
$$\pi = [0.6, 0.4]$$

(a) *Formal specification*



(b) *Visualization*

Figure 4.6: *Example of a hidden Markov model*
structure of the example taken from Wikimedia Commons:
https://commons.wikimedia.org/wiki/File:HMMGraph.svg

the parameters under which the model is most likely to generate the observed data. The Baum-Welch algorithm [BPSW70] is used for this purpose. The number of hidden states needs to be specified in advance, so ideally we would have domain knowledge about what number of states makes sense for our problem. Otherwise, we can perform hyperparameter tuning. When tuning the number of states, we should not just optimize for maximum likelihood, since this tends to lead to overfitting. We need a criterion that penalizes a model for having a high number of states. For this purpose, I use the Bayesian Information Criterion (BIC) [Sch78], which incorporates both factors. If we have some knowledge about what the learned model should look like, we can also set some parameters in advance. Fig. 4.6 shows an hidden Markov model.

As described in Section 4.1, our input after preprocessing consists of a set L of sequences of symbols representing independent runs of the underlying system. For the purposes of this method, we assume that the system resets after an error of interest errorX. Therefore, what happens after such an error is independent of what came before, and should be in a different sequence. Hence all the occurrences of errorX occur at the ends of their sequences. We now take only those sequences that end with an errorX. We first reverse the order of these sequences, so that they start with the error. Since we know that the first emission will always be errorX, we can encode this in the model by picking a state q_e as the error state, setting its probability to emit errorX to 1 (and its other emission probabilities to 0), and setting the initial probability distribution π so that

$$\pi_i = \begin{cases} 1 & i = e \\ 0 & else \end{cases}$$

Fitting the remaining parameters results in an HMM where we go to the error state first,

and can then see what states we are most likely to have come from, and what emissions are typical for these other states.

CHAPTER 5

Results

To evaluate the methods described in Chapter 4, I use three different types of data – two synthetic ones and one from industry. The following sections describe these types of data and how the various methods perform on them.

5.1 Synthetic Data Type 1

The first type of synthetic data is produced by the probabilistic deterministic finite automaton (PDFA) in Fig. 5.1. A PDFA is a DFA that, for each state, has a probability distribution over the outgoing transitions. The script that implements the automaton prints the label of every transition it goes through. Test data of size n is generated by simply running this script (i.e. going through the automaton) over and over again until it has produced n lines. Some important features of this type of data are:

- There are few different symbols (4 including the error symbol)

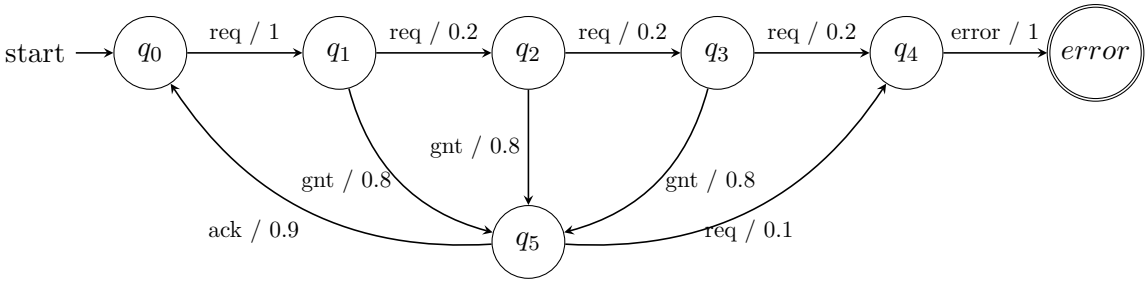


Figure 5.1: *Probabilistic deterministic finite state machine used to generate synthetic data type 1*

- Interactions between the symbols matter. "req" by itself never creates an error, but if it is preceded by "gnt" or three other "req"s, it always does.

- Errors are deterministic. There is enough information in the data to in principle always predict with certainty whether the next line will be an error. This is a stronger condition than the occurrence of errors being deterministic in the underlying system, since it additionally requires that all the relevant information be logged.

Two specific sequences of events lead to an error: four "req"s in a row, and a "req" directly after a "gnt". The performance of a method for fault analysis should be judged by its ability to find these patterns.

The analysis of the generated data was conducted in a blinded fashion – my advisors came up with the data generation process, and generated the data. I then analyzed this data, with my only additional piece of information being that the underlying system resets after each error. This enabled us to better simulate real-world conditions where the underlying causal structure of the data would also not be known in advance. The data I analyzed initially had only 10,000 lines. However, for consistency with the other synthetic data, the results I present are for another set of data I generated with 1,000,000 lines. There is no substantial difference in results between the two.

5.1.1 Supervised Learning

Fig 5.2 shows the results of supervised learning for synthetic data type 1. The odds ratios for some symbols are highly significant even for shifts up to 25. However, the ratios themselves are very close to 1 at shifts above 10, and the high significance level seems to be caused only by the large sample size (since there are few different symbols, each one occurs much more often than in the other types of data). I therefore chose 10 as the maximum shift for the logistic regression models. The average precisions of the models go from perfect precision at shift 1 down to 0.049, which is only a slightly better than the expected average precision of a dummy classifier at $1/n_positive_instances = 0.034$. The log odds for the different combinations of symbols and shifts – the core result of the method – does not really give insight into the causes of the error. For example, we can see that req at shift 1 makes an error much more likely; that's true (in fact, it is a necessary condition for an error), but it does not help us figure out why the errors occur. If we took it as an indication that there is something wrong with the implementation of req, we would be mistaken, and would probably waste time searching for the error in the wrong place. All this is despite the fact that for shift 1 the model makes perfectly precise predictions. All in all the supervised learning method does not perform all that well on this data compared to other methods. Intuitively, this is because the method tells us which symbols are relevant at which shifts. But in this data, all symbols are relevant, and it is their interactions that are the key to understanding the errors.

shift	avg. precision
1	1.000
2	0.100
3	0.097
4	0.074
5	0.074
6	0.063
7	0.059
8	0.058
9	0.051
10	0.049

(a)

symbol	shift	log odds
ack	1	-9.08
req	1	7.41
gnt	4	-3.37
req	3	3.35
gnt	3	-3.23
req	5	-2.84
ack	2	-2.07
gnt	5	1.99
gnt	7	-1.72
ack	3	-1.68
req	8	-1.45
ack	4	1.20
gnt	1	-1.12
gnt	10	-1.12
ack	6	-1.08
req	4	1.06

(b)

Figure 5.2: *Results of supervised learning for synthetic data type 1*

5.1.2 Automata Learning

Using automata learning on synthetic data type 1 produces the DFA in Fig. 5.3(a). It gets the underlying structure mostly, but not completely, correct. It correctly shows one of the sources of errors, "gnt"–>"req". It is less successful in identifying four "req"s in a row as the other error source – it only shows that after 0 or more repetitions of "req" from the start state there can be an error – which is not wrong, but imprecise. The automaton also does not show that every run must start with a "req". However, with this automaton together with manually looking at the data, I was able to reconstruct the correct automaton, as shown in Fig. 5.3(b) (which, as I described above, I did not know in advance). This raises the question of why the chosen automata learning algorithm was unable to do what I did. The principle of minimum description length should, I think, make the true automaton optimal for a large enough sample size. It could be that the sample size was too small, however the learned automaton did not substantially change between samples of size 10,000 and 10,000,000. There might also be a limitation in the specific implementation of the algorithm in Learnlib. All in all, however, the automata learning approach still performed well for data of this type.

5.1.3 Hidden Markov Model

Fig. 5.4 shows the hidden markov model for synthetic data type 1. The number of states – 14 – was chosen because it gave the best BIC during hyperparameter optimization.

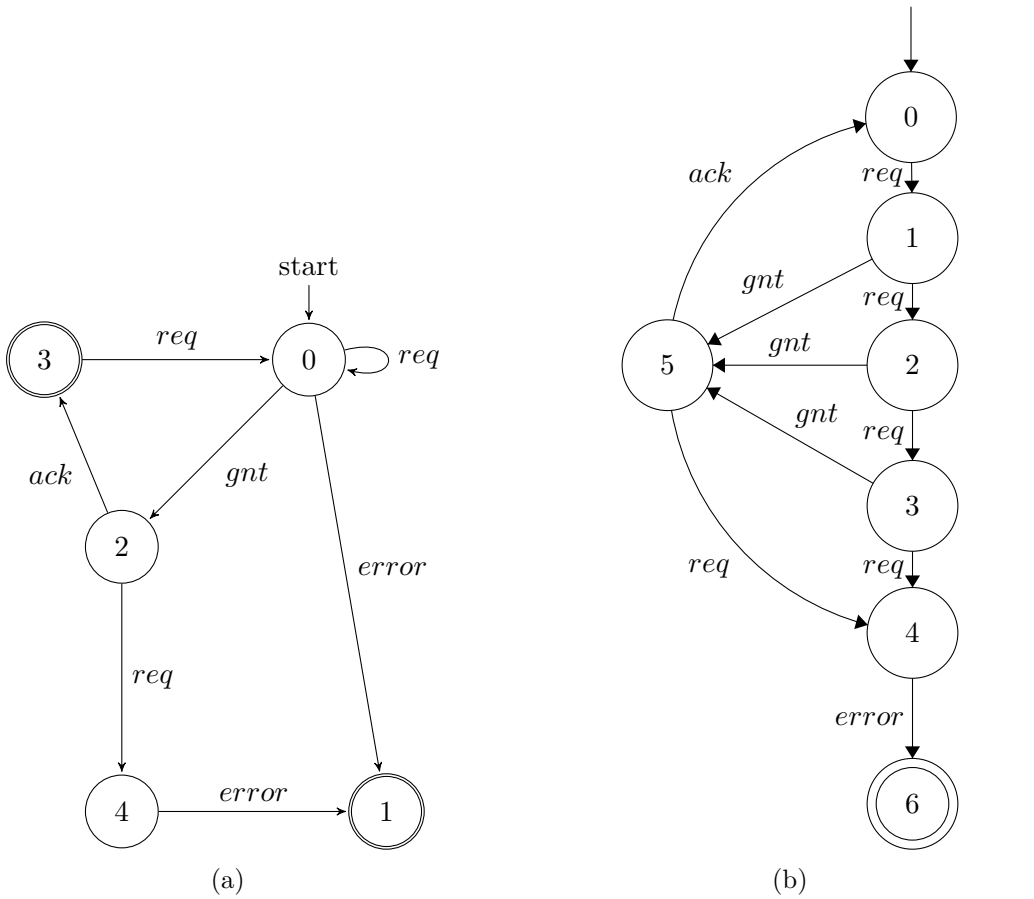


Figure 5.3: *Automata learning result for synthetic data type 2. (a): Raw automata learning output. State 3 is an end state because "ack" was the last printed symbol when the requested sample size was reached during data generation, so the last run happens to end with "ack". (b): Enhanced by manually looking for additional patterns in the data. It is equivalent to the generating PDFA in Fig. 5.1, except that it does not give any probabilities on the transitions*

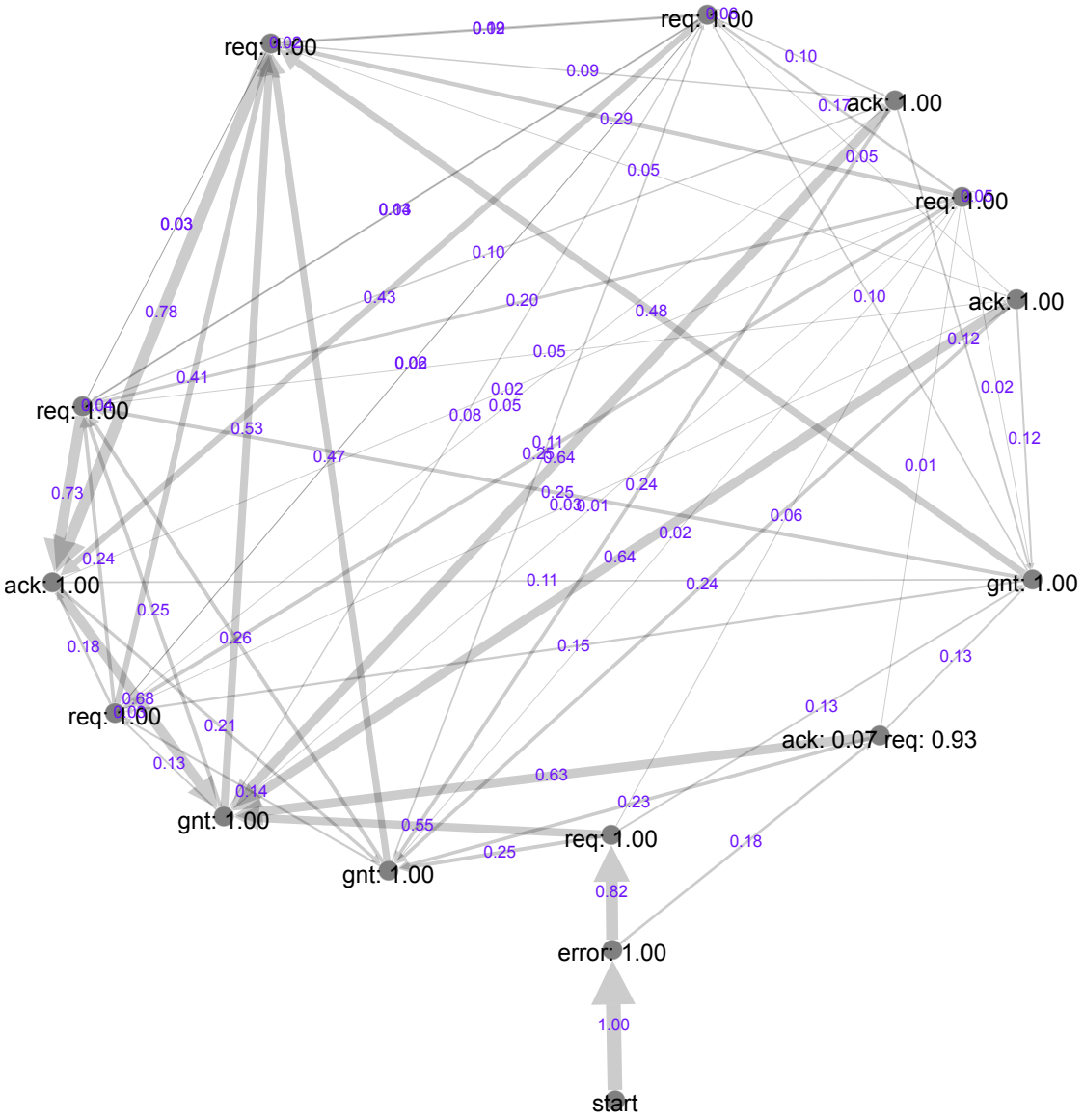


Figure 5.4: *Hidden markov model for synthetic data type 1 with 14 states.*

However given what we know about how the data is generated, it seems that the number of states is too high, and that the BIC is not penalizing additional states enough to prevent overfitting in this instance. Having this many states also makes it hard to intuitively grasp what is happening by looking at the visualization. Still, one can figure out that "gnt" –> "req" leads to an error. On the other hand, it seems very difficult without prior knowledge to figure out that repeated "req"s lead to an error as well.

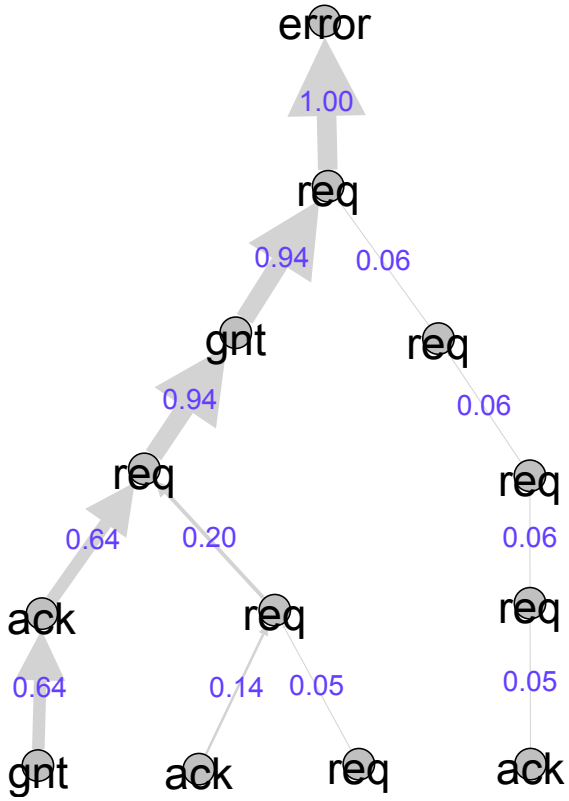


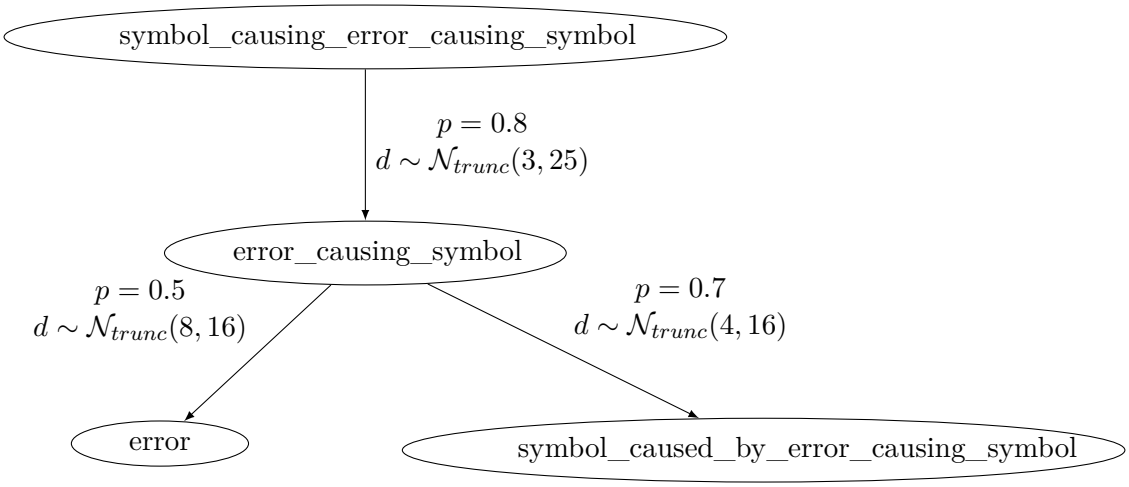
Figure 5.5: *Weighted sequence tree for synthetic data type 1. Only sequences above 1% frequency and up to 5 symbols long are shown.*

5.1.4 Weighted Sequence Tree

Fig. 5.5 shows the weighted sequence tree for synthetic data type 1. It shows both patterns that lead to the error – "gnt" –> "req" as well as four times "req". And it shows that the former is much more frequent before an error than the latter. However as always with this method, we cannot be sure whether these patterns are frequent only before the error or in general, let alone whether or not the patterns are causal.

5.2 Synthetic Data Type 2

The second type of data is generated in the following way: There is a base probability for each line being one of 100 different "irrelevant" symbols. They are irrelevant in the sense that they do not change the probability distribution over events in any subsequent lines. There are also some special events that do affect this probability distribution. They each have their own base probability, and one or more other symbol that they cause to occur with some probability on a subsequent line. If this effect triggers, the distance in lines from cause to effect is also random. More specifically, it follows a truncated normal



(a)

symbol	base probability
error	0.001
error_causing_symbol	0.001
symbol_causing_error_causing_symbol	0.001
symbol_caused_by_error_causing_symbol	0.001
irrelevant_symbol_n (1-100)	each 0.01004

(b)

Figure 5.6: *Synthetic data type 2. (a) An arrow from symbol A to symbol B means that when symbol A occurs on line n, with probability p it will cause symbol B to occur on line $n + d$, where d has a normal distribution truncated from below at* 1*. The symbols on any lines not affected by such effects are instead determined according to the base probability distribution given in (b).*

distribution (modeling a system where many different independent factors determine this delay). See Fig 5.6 for details, and see Appendix A for the code.

This second type of synthetic data has some important differences to the first:

- It has a much larger number of event types (about 100), most of which are "irrelevant" events that do not cause any other events and, for the purposes of fault analysis, are mostly noise.
- All causal effects are strictly additive – there are no interactions causing a larger or smaller total effect than the sum of the individual effects.
- Errors are probabilistic. There is not enough information in the data to predict errors with certainty. While in our case the underlying data generation process is (pseudo-) random as well, data like this could also be generated by a deterministic system that just does not log the necessary data to predict errors with certainty.

As for the first type of synthetic data, the system resets when an error is encountered. 1,000,000 lines were generated to evaluate the methods against.

5.2.1 Supervised Learning

Fig. 5.7 shows the results of supervised learning for the synthetic data type 2. The range of shift values was chosen by taking all the shift values for which there were symbols with positive odds ratios for a positive label at $p < 0.05$ (Bonferroni-corrected). The average precision values are quite low even for low shift values, and for the highest shift values they diminish to about the value expected by pure chance (which is approximately the frequency of the positive label in the sample, 0.002). Thus it seems that the models with the very highest shift values should not be taken at face value. And indeed, the coefficients from these models do not appear among the symbol-shift interactions with the highest impact in Table 5.7(b). This table shows exactly what it should: "error_causing_symbol" has the highest impact on average, with a peak close to the real peak of 8. It is followed by "symbol_causing_error_causing_symbol". "Symbol_caused_by_error_causing_symbol" meanwhile is not shown as having any unusual causal impact – it appears among the "irrelevant" symbols so far down that it is not displayed in the table, which only shows the lines with the largest coefficients. This clearly shows that the approach was able to distinguish between causation and mere correlation.

5.2.2 Automata Learning

Figure 5.8 shows the learned DFA for synthetic data type 2. It does not contain much insight – only that the sequences end with an error, errors never occur before the end, and that there are no further constraints on what can happen. All of that is true, but not very helpful for finding the causes of errors. Since the automaton only shows what is possible, not what is probable, it is not a good fit for this type of data, where almost anything is possible.

5.2.3 Hidden Markov Model

One might think that a hidden markov model would be a good fit for synthetic data type 2. After all, the process by which it is generated is quite similar to a hidden markov model – there is a probability distribution over emissions, and this distribution changes when certain events occur. The only property that does not seem to fit are the normally distributed distances from cause to effect, but even that could be modeled by an HMM, albeit with a lot of states. In practice however, the HMM approach did not perform well. Hyperparameter optimization with BIC yielded an HMM with only two states (see Fig. 5.9), which gives the same trivial information as the automata learning approach, with the only (also unhelpful) addition being the overall frequencies of the non-error symbols in the sample. Manually adding more states also did not help: I tried 8 states, but did not gain any additional insights.

shift	avg. precision
1	0.024
2	0.025
3	0.027
4	0.026
5	0.025
6	0.027
7	0.025
8	0.020
9	0.017
10	0.012
11	0.010
12	0.011
13	0.008
14	0.007
15	0.004
16	0.004
17	0.003
18	0.003
19	0.003
20	0.003
21	0.002
22	0.002

(a)

symbol	shift	log odds ratio
error_causing_symbol	6	3.57
error_causing_symbol	7	3.28
symbol_causing_error_causing_s.	15	3.26
error_causing_symbol	5	3.14
error_causing_symbol	8	3.06
symbol_causing_error_causing_s.	10	3.03
error_causing_symbol	9	2.97
error_causing_symbol	4	2.97
symbol_causing_error_causing_s.	12	2.90
symbol_causing_error_causing_s.	11	2.90
symbol_causing_error_causing_s.	14	2.87
symbol_causing_error_causing_s.	16	2.74
error_causing_symbol	11	2.70
symbol_causing_error_causing_s.	13	2.69
symbol_causing_error_causing_s.	18	2.68
symbol_causing_error_causing_s.	17	2.67
error_causing_symbol	3	2.66
error_causing_symbol	12	2.56
symbol_causing_error_causing_s.	9	2.55
error_causing_symbol	10	2.49
symbol_causing_error_causing_s.	20	2.36
error_causing_symbol	2	2.29
symbol_causing_error_causing_s.	8	2.28
symbol_causing_error_causing_s.	19	2.27
error_causing_symbol	1	2.15

(b)

Figure 5.7: *Supervised learning results for synthetic data 2. (a) shows the average precisions for the models estimating the causal effects at the given shifts. (b) shows the causal effects of the different symbols at different shifts. Only the symbol-shift combinations with the strongest impact are depicted, down to an (arbitrary) log-odds cutoff of 2.*

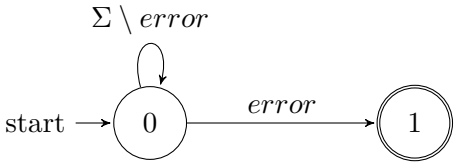


Figure 5.8: *Learned automaton for synthetic data type 2. The transition label* $\Sigma \setminus error$ *signifies that this transition is valid for all symbols of the alphabet* Σ *except error.*

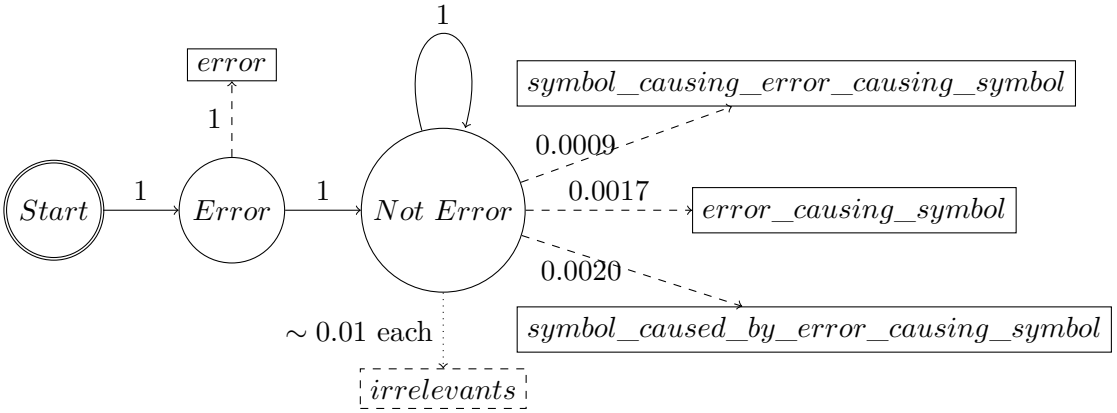


Figure 5.9: *2-state hidden markov model for synthetic data type 2. The emissions for the 100 "irrelevant" symbols are not shown separately, but consolidated for visual clarity.*

5.2.4 Weighted Sequence Tree

The weighted sequence tree method is not a good fit for synthetic data type 2. Between a symbol with causal significance and its effect, there are usually several irrelevant, completely randomly selected symbols. For example, when an "error_causing_symbol" causes an error, the mean and modal distance to the error is 8, with a standard deviation of 4. For the modal case of distance 8, there are 7 irrelevant symbols in-between. As there are 100 different irrelevant symbols with the same probability and with an overall probability of one of them occurring at any given line of almost 1, there are $100^7 = 100$ trillion different paths with the same probability, so it is quite unlikely that one of them will occur multiple times in a sample, let alone enough to jump out by clearing some weight threshold necessary to keep the visible paths to a manageable level. For smaller distances the chances are better, but these smaller distances are themselves less likely. Fig 5.10 shows the weighted sequence tree results for the synthetic data type 2 dataset. All the sequences above the 1% frequency level have a length of only one element. All but three of them are irrelevant symbols at 1% frequency. The first of the three is another irrelevant symbol that reached 2% by chance. The second is the error_causing_symbol, which is the one that should be featured most prominently in a causal analysis, but which does not quite escape the noise floor at 2%. And the third is the symbol_caused_by_error_causing_symbol, which is more prominent at 4% and reflects a real correlation between that symbol and errors, but as we know does not cause the errors. All in all, as theoretical considerations predicted, the weighted sequence tree method does perform very well on this data.

5.3 Industry Data

The third type of data is real-life data from an industry source. The system that generated the log data is a test automation system for the automotive industry. The dataset

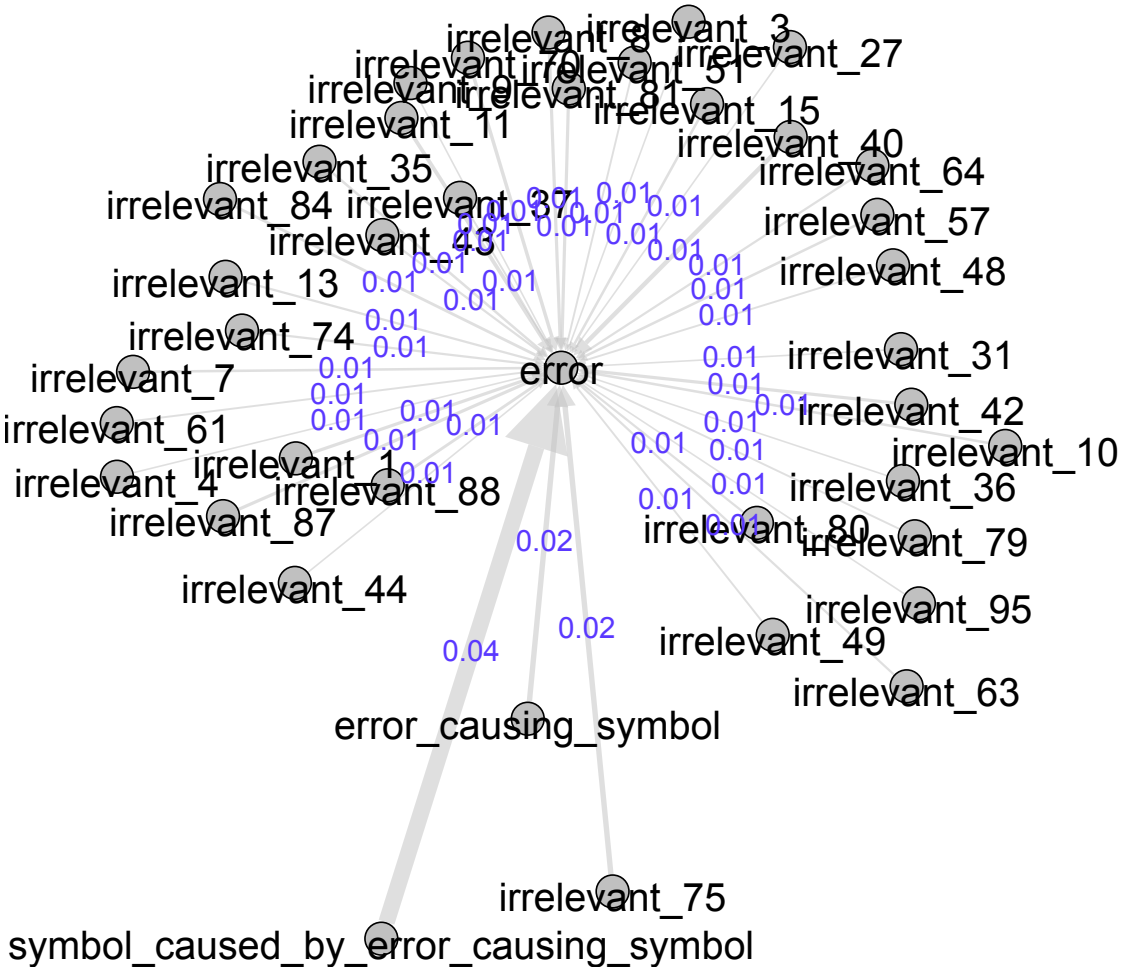


Figure 5.10: *Weighted sequence tree for synthetic data type 2. Only sequences above 1% frequency are shown.*

used for this thesis contains the log output from a single customer using the system over about six months, from November 2017 to May 2018. It comprises around 9.2 million individual log statements. Each log statement consists of the following data points:

Timestamp given down to the millisecond level.

Severity in ascending order: Info, Warning, Error, Alarm

Process The use of different processes does not necessarily indicate parallelism. "Processes" in this context are better understood as functional components of the software, akin to modules.

Message Code There is a limited number of message codes, and each log line must use one of them.

Message Text Contains additional, specific information about the error in natural language.

I use the "Message Code" for my analysis. I also experimented with using the "Process" field in addition, but I judged that the marginally better results did not justify the additional complexity. I elected not to use the "Message Text", which would need to be parsed in some way to be useful for machine learning (but see [FLWL09] for one way to do this). In the original logfiles, the lines are mostly, but not always, arranged in the order indicated by the timestamp. I suspect that such discrepancies occur when parallel processes aggregate log data in separate buffers and flush them to the log file at different times. I used the timestamp to order the log statements, which should arrange the log statements in the order in which the underlying events actually occurred. Statements with the same timestamp are left in the order in which they occurred in the logfiles. The most severe errors, and therefore the most interesting ones from a fault explanation perspective, are the ones with severity "Alarm". There are a number of different error types with this severity, but many of them do not appear often enough to make a statistical analysis feasible. I concentrate on the single most frequent error of this category. I also filter out all statements of severity "Info" during preprocessing, on the advice of a domain expert from our industry partner. From the same source, we learned that after an alarm the system has to be reset, and we also learned which log lines indicate this reset. Since further alarms between an alarm and the next reset are probably just followup errors caused by the same cause as the first alarm, and behavior after the reset not affected by what has happened before, all the lines between an alarm and the next reset are considered irrelevant and are also filtered out. This leaves about 2.6 million lines, of which 2.3 million are warnings, 0.3 million are errors, and 2181 are alarms. Of the alarms, 931 are instances of errorX. There are 1580 different message codes, of which 207 appear in warnings, 1368 in errors, and 9 in alarms (four of the codes appear in two categories). As mentioned, there are certain lines in the log that indicate a reset of the system, which enables me to split the log data into 13506 independent sequences, of which 931 end with an instance of errorX and 1251 end with another alarm.

Judging the accuracy of fault analysis results for this dataset is somewhat challenging, since in contrast to the synthetic datasets, nobody, including my industry partner, knows the exact underlying causal structure. Nevertheless the industry partner is in a position to compare our results to their domain knowledge and give feedback on whether the results are expected or unexpected, and in the latter case investigate further to confirm or refute the findings. As it turns out, all my results were judged by the industry partner to be consistent with their knowledge of the system, but not significantly add to it. I will further discuss the implications of this outcome in Chapter 6

At the request of our industry partner, I have anonymized sensitive parts of the data shown in this paper: All the message code names, which normally contain some human-readable information about the type of event that occurred, have been replaced with generic names with the pattern "messageCode<number>". Everything else is unchanged.

5.3.1 Supervised Learning

Fig. 5.11 shows the results of supervised learning on industry data. I estimated 20 different logistic regression models to estimate the causal effects of symbols at shifts 1 through 20. As shown in Table 5.11(a), the average precision monotonically decreases with increasing shifts, but always remains clearly above the level expected by chance, which, as laid out in Section 4.3, is approximately equal to the proportion of positive observations, $931/2,600,000 = 0.0004$. I elected to treat the data as one long sequence, instead of splitting it at the "reset" log statements, since with supervised learning it is not necessary to have multiple sequences, and I could not be sure that the reset was total. And in fact there is some indication in the results that the reset really is not total, or that there is some cause of errors that is external to the system: One of the symbols listed as causing errorX is errorX, which should not be possible if after each error the system completely resets.

5.3.2 Automata Learning

Fig 5.12 shows the learned DFA for industry data. Since we are interested in how errorX occurs, rather than the complete structure of the system, I only used sequences ending with errorX as input. I also only use the last few observations of each sequence, both because when using the full sequences the learning algorithm does not terminate in an acceptable amount of time, and to keep the result legible. In fact, even when using only the 10 last observations for each sequence, the result is already quite large and hard to interpret. For a large majority of symbols, including errorX, there is also a self-loop at the start state, so the automaton does not constrain possibilities by much. If there is any usable information for finding the cause of the error in this result, I cannot see it.

5.3.3 Hidden Markov Model

Fig. 5.13 shows the learned hidden Markov model for industry data. Due to the large number of states, the model is hard to grasp as a whole, but it is encouraging that there is quite a bit of structure: Most states have only one or two emissions with probabilities above 10%, and there are three transitions out of the error state that clearly dominate. However, the model appears to either not show any symbol interactions that cause errors or to be too complex for them to be clearly visible, so using it to investigate root causes of errors might boil down to investigating individual message codes that are frequently emitted by states frequently transitioned to from the error state in the model. It seems to me that the supervised learning approach is better suited to this task, since it can distinguish causality from mere association. However there may be some value in seeing which message codes are closely related, such as when they both occur with high probability in the same state, and that state has a high probability of transitioning to itself.

shift	avg. precision
1	0.360
2	0.181
3	0.040
4	0.035
5	0.027
6	0.014
7	0.014
8	0.007
9	0.007
10	0.008
11	0.007
12	0.007
13	0.007
14	0.006
15	0.005
16	0.005
17	0.005
18	0.005
19	0.005
20	0.004

(a)

symbol	shift	log odds ratio	occurrences
msg_code_1411	2.0	6.67	33
msg_code_1414	2.0	6.44	497
msg_code_1222	1.0	6.27	120
msg_code_1409	1.0	5.58	1291
msg_code_1145	2.0	4.71	114
msg_code_1414	1.0	4.68	497
msg_code_1222	5.0	4.49	120
msg_code_1650	5.0	4.41	29
msg_code_1145	1.0	4.25	114
msg_code_1414	3.0	3.93	497
msg_code_1601	2.0	3.65	199
msg_code_1145	3.0	3.54	114
msg_code_1222	7.0	3.04	120
msg_code_1414	4.0	2.97	497
msg_code_1213	2.0	2.96	959
msg_code_1637	4.0	2.92	806
msg_code_1753	7.0	2.80	25
msg_code_1294	6.0	2.75	164
msg_code_1409	2.0	2.60	1291
msg_code_1676	4.0	2.53	29
msg_code_1654	18.0	2.53	114
msg_code_1222	8.0	2.46	120
msg_code_1417	1.0	2.40	5
msg_code_1665	7.0	2.40	29
msg_code_1669	3.0	2.38	4398
msg_code_1654	20.0	2.29	114
errorX	2.0	2.26	931
msg_code_1624	3.0	2.17	215
msg_code_1616	4.0	2.15	48
msg_code_1666	3.0	2.14	112
msg_code_1463	13.0	2.13	138
msg_code_1411	1.0	2.11	33
msg_code_1654	19.0	2.08	114

(b)

Figure 5.11: *Results of supervised learning for industry data. Table (a) shows the average precisions for the models estimating the causal effects at the given shifts. Table (b) shows the causal effects of different symbols at different shifts. Only the symbol-shift combinations with the strongest impact are depicted, down to an (arbitrary) log-odds cutoff of 2. The "occurrences" column shows how often each symbol occurs in the sample, to indicate how relevant each effect might be in practice.*

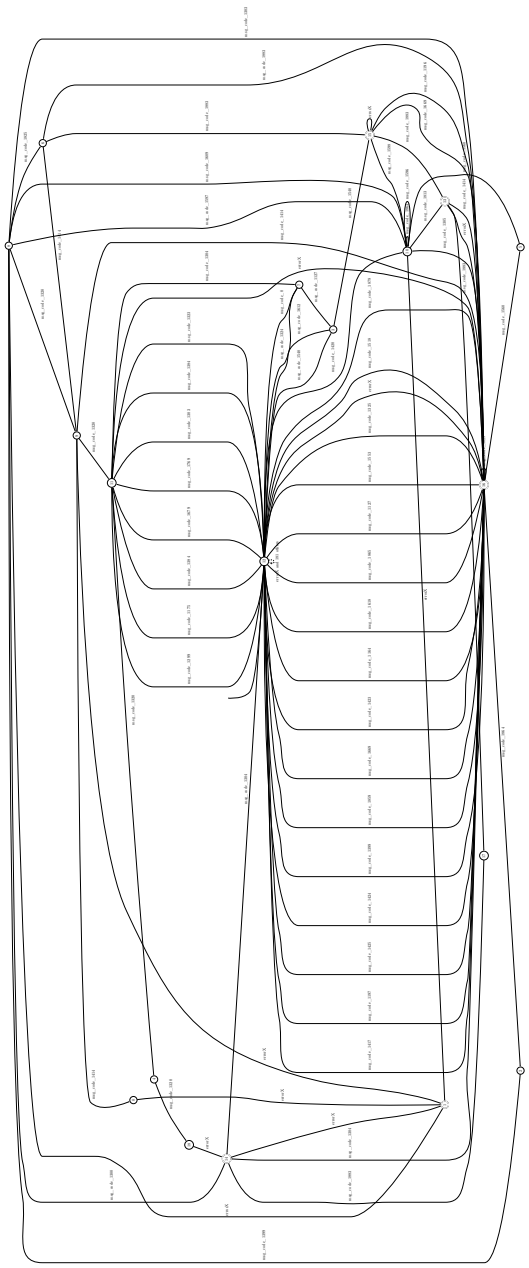


Figure 5.12: *Learned automaton for industry data. Only sequences that end with errorX were used. Furthermore only the last 10 observations of each sequence were used, both for performance reasons and for legibility. The start state (S12, in the middle) has self loops for 182 out of 209 symbols in* Σ*. I do not show each of them in the figure explicitly, again for legibility.*

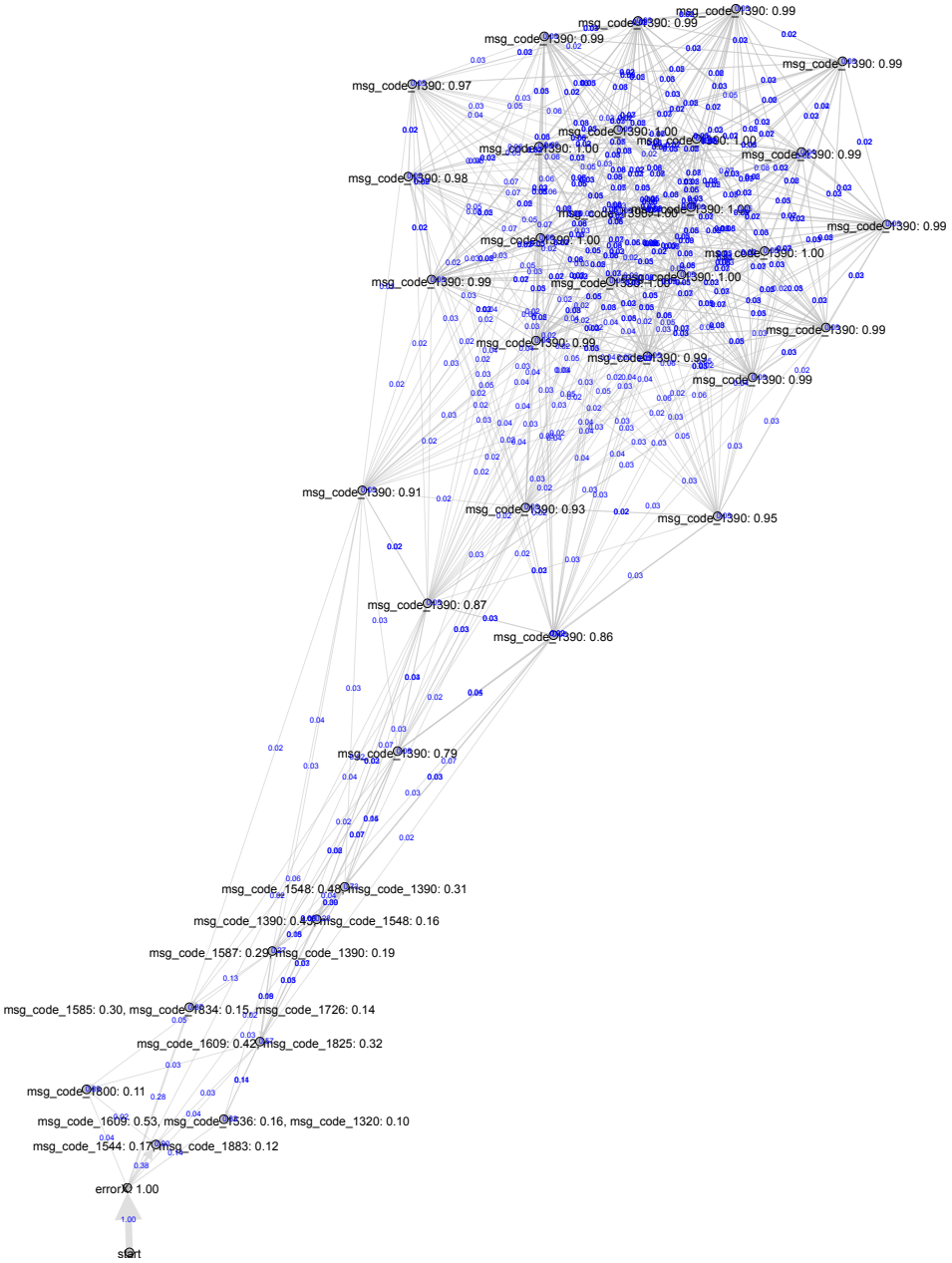


Figure 5.13: *The learned hidden Markov model for industry data. It has 37 states, as suggested by hyperparameter tuning. For the sake of visual clarity, transitions with a probability below 2% are not shown, as are emissions with a probability below 10%.*

5.3.4 Weighted Sequence Tree

Fig. 5.14 shows the result of the weighted sequence tree method described in Section 4.2 applied to our industry data set. Given the large number of different message codes, it is striking that there are some long subsequences - of up to seven message codes - that appear before more than 1% of the errors of interest. A caveat is that in some of the subsequences, the same message is repeated many times. It is possible that in such cases we are not seeing a sequence of multiple events leading to an error, but one event being reported multiple times. We must look at the distribution of message codes in the overall log files to make sure that the message codes and sequences in the tree are frequent specifically before errorX rather than simply frequent in general. msg_code_1390 does occur very frequently – 1.7 million times, which is actually the majority of lines in the data. And when it occurs, it seems to often occur in many repetitions. In light of this fact, its long branch in the weighted sequence tree probably does not hint at a causal connection to the error. The other message codes all occur orders of magnitude less frequently. While it is still possible that for some of them the association is spurious, we can take the sequences in the tree as a point of departure for investigating within the underlying system.

5.4 Summary of Results

There is no single measure on which to compare the quality of the different results or methods. Measures such as predictive accuracy are not well suited to the problem because only supervised learning makes discrete predictions that could be measured this way, and because we are not actually interested in predicting anything. Rather, a result must fulfill multiple, somewhat qualitative criteria to be useful:

- It must be interpretable, i.e. by studying it one must be able to learn something about the system and why it produces a particular error without being overwhelmed by the complexity of the result.
- It should discover patterns that are relevant.
- It should not misleadingly show patterns that are not relevant.

Some results, namely those produced using hidden Markov models and automata learning on industry data, failed on the criterion of interpretability. Admittedly, the question of how much complexity and detail is too much is somewhat subjective, and there are edge cases. The result of using the hidden Markov model on synthetic data type 1 might be such an edge case. I could see one pattern that causes errors, but not the other. But it is possible that someone else might see more or less.

Some results clearly failed to discover relevant patterns – mainly when analyzing synthetic data type 2. The ones using hidden Markov model and automata learning did not really produce any results at all, having collapsed into small models that only showed

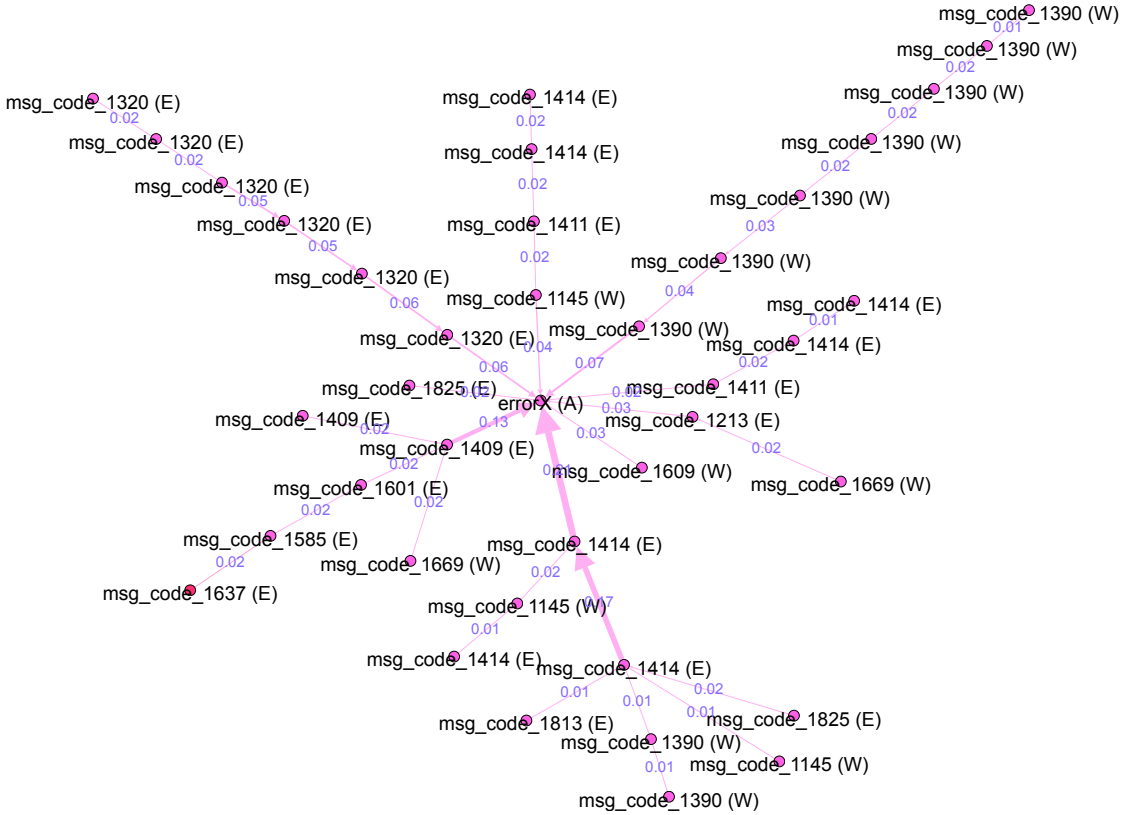


Figure 5.14: *The weighted sequence tree for industry data. Edges with a weight below 0.01 (corresponding to suffixes occurring in less than 1% of sequences) have been filtered out for the sake of legibility. Next to each message code, its severity is shown – "(W)" for warning, "(E)" for error, and "(A)" for alarm*

truisms. The weighted sequence tree result found one relevant pattern, but it was barely more prominent than random irrelevant patterns, and less prominent than a real, but non-causal pattern.

It seems like most results that were interpretable preformed well on the last criterion (not misleadingly showing irrelevant patterns), with the exception of the weighted sequence tree on synthetic data 2, which shows many random irrelevant patterns as mentioned.

On synthetic data type 1, all the methods did somewhat well. The weakest here was supervised learning, which cannot show the interactions that are important for this data type. Automata learning did best here. For the other two data types, supervised learning performed best, while the other methods mostly failed. There is however a caveat for any claims of good performance on industry data: I do not know what actually causes the error, and can only say that the results are plausible and consistent with my industry partner's domain knowledge. Table 5.1 shows a summary of the results for different combinations of results and data types.

	synthetic data type 1	synthetic data type 2	industry data
weighted sequence tree	good	poor (error-causing symbol did not rise above noise floor, could not distinguish between causal and non-causal associations)	ok[1]
hidden Markov model	good (could see one pattern that leads to the error, but not the other, less frequent one)	poor (collapsed into a 2-state model with no useful information)	poor (too complex to see patterns)
automata learning	very good (got automaton mostly correct, could be corrected with supplementary manual analysis)	poor (collapsed into a single state with no useful information)	poor (too complex to see patterns)
supervised learning	ok (could predict errors very accurately, but gives no information about interactions)	very good (correctly identified causal effects, not tricked by non-causal associations)	good[1]

Table 5.1: *Summary of the results for different combinations of methods and data types. I have each result a grade on a scale of poor-ok-good-very good. The grading is somewhat subjective, but since the performance differences are mostly very clear, I think it is far from arbitrary. The most salient characteristics of each result are also given.*

[1]The results could not be fully verified

CHAPTER 6

Conclusion

In this thesis, I present several methods for fault analysis using log data. In contrast with previous works using machine learning methods on log data, which largely make predictions of failures and in some cases give explanations for individual errors, I attempt to give unified explanations for whole classes of errors. Each of the four methods I present is evaluated using three different types of data, revealing the strengths and weaknesses of each method. There are several possibilities for future work:

- This work uses data that already comes "pre-clustered" – the industry data and synthetic data have a limited set of message codes and symbols respectively. However, log data often lacks such a preexisting clustering, and even the industry data used here additionally contains unstructured data that would potentially allow for more fine-grained clustering, akin to the clustering described in some of the papers I cite in the literature chapter. By adopting such a clustering approach, the presented methods could be extended to unstructured log data, and their performance could be improved for log data which is only partially structured.
- The strengths and weaknesses of the different methods are such that it might be useful to combine some of them. For example, the supervised learning method works well for finding some relevant types of events among a large number of irrelevant ones (the "needles in the haystack", e.g. synthetic data type 2), while hidden Markov models or automata learning is works well at showing the interactions of a few event types that are all more or less relevant (e.g. synthetic data type 1). For data where there are both many irrelevant messages and the interactions between the relevant ones are important, it might be useful to first filter out the irrelevant ones with supervised learning and then analyze the interactions of the relevant ones with one of the other approaches.
- The synthetic data used in this thesis is relatively simple, so it is not clear whether the good results some of the methods achieved on it would extend to real-world

datasets. As for the industry data, I lacked sufficient information about the true causes of errors to make definitive claims about the quality of the results of the analyses. And I also did not have access to the system that generated the data to confirm or refute my results. It would be useful to test the methods developed in this thesis on more complex synthetic data and / or real-world data where the causes of errors are known in advance or where the results of analyses can be checked directly in the system that generated the log data.

- As mentioned in Section 5.4, the measures employed to compare results are somewhat subjective. It would be good to find or develop an objective metric for such comparisons.

Bibliography

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations*, 2015.

[Bes15] Yves Bestgen. Exact expected average precision of the random baseline for system evaluation. *The Prague Bulletin of Mathematical Linguistics*, 103(1):131–138, 2015.

[BPSW70] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *Ann. Math. Statist.*, 41(1):164–171, 02 1970.

[BTHN18] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *Proceedings of the First Workshop on Machine Learning for Computing Systems*, MLCS'18, pages 1:1–1:8, New York, NY, USA, 2018. ACM.

[dlH10] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[DLLP97] Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1):31 – 71, 1997.

[DLZS17] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1285–1298, New York, NY, USA, 2017. ACM.

[ESW12] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 187–201, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[FLWL09] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009.

[FSS+13] Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, Mikko Terho, and Jelena Vlasenko. Failure prediction based on log files using random indexing and support vector machines. *Journal of Systems and Software*, 86(1):2–11, 2013.

[GMP05] Peter D Grünwald, In Jae Myung, and Mark A Pitt. *Advances in minimum description length: Theory and applications*. MIT press, 2005.

[HCM+00] David Heckerman, David Maxwell Chickering, Christopher Meek, Robert Rounthwaite, and Carl Kadie. Dependency networks for inference, collaborative filtering, and data visualization. *Journal of Machine Learning Research*, 1(Oct):49–75, 2000.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1985.

[IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 487–495, Cham, 2015. Springer International Publishing.

[JM11] Manu Jose and Rupak Majumdar. Bug-assist: Assisting fault localization in ANSI-C programs. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 504–509, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*. Springer, 2013.

[LL17] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[LWLZ17] Shixia Liu, Xiting Wang, Mengchen Liu, and Jun Zhu. Towards better analysis of machine learning models: A visual analytics perspective. *Visual Informatics*, 1(1):48 – 56, 2017.

[ME10] Nizar R Mabroukeh and Christie I Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):3, 2010.

[NKN12] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and*

Implementation, NSDI'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[ÖA00] Fikri Öztürk and Fikri Akdeniz. Ill-conditioning and multicollinearity. *Linear Algebra and its Applications*, 321(1):295 – 305, 2000. Eighth Special Issue on Linear Algebra and Statistics.

[Pea09] Judea Pearl. Causal inference in statistics: An overview. *Statist. Surv.*, 3:96–146, 2009.

[POvHG18] Teerat Pitakrat, Dušan Okanović, André van Hoorn, and Lars Grunske. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software*, 137:669 – 685, 2018.

[Ris83] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *The Annals of Statistics*, 11(2):416–431, 1983.

[SB88] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[Sch78] Gideon Schwarz. Estimating the dimension of a model. *Ann. Statist.*, 6(2):461–464, 03 1978.

[SFMW14] Ruben Sipos, Dmitriy Fradkin, Fabian Moerchen, and Zhuang Wang. Log-based predictive maintenance. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1867–1876, New York, NY, USA, 2014. ACM.

[SHM11] Bernhard Steffen, Falk Howar, and Maik Merten. *Introduction to Active Automata Learning from a Practical Perspective*, pages 256–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Spl19] Splunk Inc. Modernize your IT monitoring with predictive analytics. Brochure, 2019.

[Sum19] SumoLogic. Devops unleashed. Company website, `https://www.sumologic.com/solutions/operations-analytics/`, 2019.

[Sys] Loom Systems. Sophie's AI - engine overview. Brochure.

[TWW16] Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. *Formal Methods in System Design*, 49(1):1–32, Oct 2016.

[WGL+16] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[XZCN19] Jianwu Xu, Hui Zhang, Haifeng Cheng, and Bin Nie. Log-based system management and maintenance. Patent Application US 2019 / 0095313 A1, 2019.

[ZXM+16] Ke Zhang, Jianwu Xu, Martin Renquiang Min, Guofei Jiang, Konstantinos Pelechrinis, and Hui Zhang. Automated it system failure prediction: A deep learning approach. In *2016 IEEE International Conference on Big Data*, pages 1291–1300, Dec 2016.

Appendices

APPENDIX A

Code for generating synthetic data

A.1 Synthetic data type 1

```
1  import java.io.FileNotFoundException;
2  import java.io.PrintWriter;
3  import java.io.UnsupportedEncodingException;
4  import java.util.Random;
5
6  import static java.nio.charset.StandardCharsets.UTF_8;
7
8  public class ErrorTraceGenerator {
9      public static void main(String[] args) throws FileNotFoundException,
           ↪ UnsupportedEncodingException {
10         //Random number generator initialized with a different seed
11         Random rand = new Random(System.currentTimeMillis());
12         int tracelen = 1000000;
13         PrintWriter writer = new PrintWriter(
14                 "data/synthetic_traces/trace" + String.valueOf(tracelen) +
15                         "_" + String.valueOf(System.currentTimeMillis()) + ".txt",
16                 UTF_8
17         );
18         int state = 0;
19
20         while (tracelen != 0) {
21             tracelen--;
22             switch (state) {
23                 case 0:
24                     // You do a request with probability 1
25                     writer.println("req");
26                     state = 1;
27                     break;
28                 case 1:
29                     // Request with probability 0.2 (state 2)
30                     // Grant with probability   0.8 (state 3)
31                     if (rand.nextDouble() <= 0.2) {
32                         writer.println("req");
```

```
33                          state = 2;
34                      }
35                      else {
36                          writer.println("gnt");
37                          state = 3;
38                      }
39                      break;
40                  case 2:
41                      // Request with probability 0.2 (state 4)
42                      // Grant with probability   0.8 (state 3)
43                      if (rand.nextDouble() <= 0.2) {
44                          writer.println("req");
45                          state = 4;
46                      }
47                      else {
48                          writer.println("gnt");
49                          state = 3;
50                      }
51                      break;
52                  case 3:
53                      // Ack with probability 0.9 (state 0)
54                      // Req with probability 0.1 (state 5)
55                      if (rand.nextDouble() <= 0.9) {
56                          writer.println("ack");
57                          state = 0;
58                      }
59                      else {
60                          writer.println("req");
61                          state = 5;
62                      }
63                      break;
64                  case 4:
65                      // Request with probability 0.2 (state 5)
66                      // Grant with probability   0.8 (state 3)
67                      if (rand.nextDouble() <= 0.2) {
68                          writer.println("req");
69                          state = 5;
70                      }
71                      else {
72                          writer.println("gnt");
73                          state = 3;
74                      }
75                      break;
76                  case 5:
77                      writer.println("error");
78                      state = 0;
79                      break;
80              }
81          }
82          writer.close();
83      }
84  }
85  import java.io.IOException;
86  import java.io.PrintWriter;
87  import java.util.Random;
88
89  import static java.nio.charset.StandardCharsets.UTF_8;
90
91  public class ErrorTraceGenerator {
92      public static void main(String[] args) throws IOException {
93          //Random number generator initialized with a different seed
94          Random rand = new Random(System.currentTimeMillis());
```

```
95          int tracelen = 1000000;
96          PrintWriter writer = new PrintWriter(
97                  "data/synthetic_traces/trace" + String.valueOf(tracelen) +
98                          "_" + String.valueOf(System.currentTimeMillis()) + ".txt",
99                  UTF_8
100         );
101         int state = 0;
102
103         while (tracelen != 0) {
104             tracelen--;
105             switch (state) {
106                 case 0:
107                     // You do a request with probability 1
108                     writer.println("req");
109                     state = 1;
110                     break;
111                 case 1:
112                     // Request with probability 0.2 (state 2)
113                     // Grant with probability   0.8 (state 3)
114                     if (rand.nextDouble() <= 0.2) {
115                         writer.println("req");
116                         state = 2;
117                     }
118                     else {
119                         writer.println("gnt");
120                         state = 3;
121                     }
122                     break;
123                 case 2:
124                     // Request with probability 0.2 (state 4)
125                     // Grant with probability   0.8 (state 3)
126                     if (rand.nextDouble() <= 0.2) {
127                         writer.println("req");
128                         state = 4;
129                     }
130                     else {
131                         writer.println("gnt");
132                         state = 3;
133                     }
134                     break;
135                 case 3:
136                     // Ack with probability 0.9 (state 0)
137                     // Req with probability 0.1 (state 5)
138                     if (rand.nextDouble() <= 0.9) {
139                         writer.println("ack");
140                         state = 0;
141                     }
142                     else {
143                         writer.println("req");
144                         state = 5;
145                     }
146                     break;
147                 case 4:
148                     // Request with probability 0.2 (state 5)
149                     // Grant with probability   0.8 (state 3)
150                     if (rand.nextDouble() <= 0.2) {
151                         writer.println("req");
152                         state = 5;
153                     }
154                     else {
155                         writer.println("gnt");
156                         state = 3;
```

```
                    }
                    break;
                case 5:
                    writer.println("error");
                    state = 0;
                    break;
            }
        }
        writer.close();
    }
}
```

A.2 Synthetic data type 2

```
import os
import random
import string
from typing import Dict

import numpy as np
import pandas as pd
from scipy import stats

from src.constants import DATA_DIR


class SymbolInfluence:
    def __init__(self, origin: "SpecialSymbol",
                 target: "SpecialSymbol",
                 probability_of_causation: float,
                 mean_distance_in_case_of_causation: int,
                 std_dev_in_case_of_causation: int
                 ):
        self.origin = origin
        self.target = target
        self.probability_of_causation = probability_of_causation
        self.mean_distance_in_case_of_causation = mean_distance_in_case_of_causation
        self.std_dev_in_case_of_causation = std_dev_in_case_of_causation


class Symbol:
    def __init__(self, name: str):
        self.name = name


class SpecialSymbol(Symbol):
    def __init__(self, name: str,
                 base_probability: float = 0.001,
                 influenced_symbols: Dict["SpecialSymbol", SymbolInfluence] = None
                 ):
        super().__init__(name)
        self.base_probability = base_probability
        self.influenced_symbols = influenced_symbols if influenced_symbols is not None
            ↪ else {}

    def __hash__(self):
        return hash(self.name)
```

```
43
44      def __eq__(self, other):
45          return isinstance(self, type(other)) and self.name == other.name
46
47
48  ERROR = SpecialSymbol("error", base_probability=0.001)
49
50  ERROR_CAUSING_SYMBOL = SpecialSymbol("error_causing_symbol", base_probability=0.001)
51  ERROR_CAUSING_SYMBOL.influenced_symbols[ERROR] = SymbolInfluence(ERROR_CAUSING_SYMBOL,
        ↪ ERROR, 0.5, 8, 4)
52
53  SYMBOL_CAUSED_BY_ERROR_CAUSING_SYMBOL = SpecialSymbol("
        ↪ symbol_caused_by_error_causing_symbol", base_probability=0.001)
54  ERROR_CAUSING_SYMBOL.influenced_symbols[SYMBOL_CAUSED_BY_ERROR_CAUSING_SYMBOL] = \
55      SymbolInfluence(ERROR_CAUSING_SYMBOL, SYMBOL_CAUSED_BY_ERROR_CAUSING_SYMBOL, 0.7,
            ↪ 4, 4)
56
57  SYMBOL_CAUSING_ERROR_CAUSING_SYMBOL = SpecialSymbol("
        ↪ symbol_causing_error_causing_symbol", base_probability=0.001)
58  SYMBOL_CAUSING_ERROR_CAUSING_SYMBOL.influenced_symbols[ERROR_CAUSING_SYMBOL] = \
59      SymbolInfluence(SYMBOL_CAUSING_ERROR_CAUSING_SYMBOL, ERROR_CAUSING_SYMBOL, 0.8, 3,
            ↪ 5)
60
61
62  def generate(n_lines, n_irrelevant_symbols):
63      irrelevant_symbols = [Symbol("irrelevant_" + str(i)) for i in range(
            ↪ n_irrelevant_symbols)]
64      special_symbols = \
65          [ERROR, ERROR_CAUSING_SYMBOL, SYMBOL_CAUSED_BY_ERROR_CAUSING_SYMBOL,
                ↪ SYMBOL_CAUSING_ERROR_CAUSING_SYMBOL]
66      p_special_symbol = sum(special_symbol.base_probability for special_symbol in
            ↪ special_symbols)
67      p_irrelevant_symbol = (1 / n_irrelevant_symbols) * (1 - p_special_symbol)
68      symbols = irrelevant_symbols + special_symbols
69      base_probabilities = \
70          n_irrelevant_symbols * [p_irrelevant_symbol] + [symbol.base_probability for
                ↪ symbol in special_symbols]
71      file_path = os.path.join(DATA_DIR, "synthetic_traces", "many_symbols", "
            ↪ many_symbols_trace_" + str(n_lines)
72                               + "_lines_" + ''.join(random.choices(string.
                                     ↪ ascii_uppercase + string.digits, k=5)) + ".txt"
73                               )
74      with open(file_path, "w") as file:
75          special_symbol_to_distance_to_next_occurrence = {symbol: None for symbol in
                ↪ special_symbols}
76          for _ in range(n_lines):
77              for symbol in special_symbols:
78                  if special_symbol_to_distance_to_next_occurrence[symbol] is not None:
79                      special_symbol_to_distance_to_next_occurrence[symbol] -= 1
80
81              imminent_symbols = \
82                  [symbol for symbol, distance in
                        ↪ special_symbol_to_distance_to_next_occurrence.items() if
                        ↪ distance == 0]
83              if len(imminent_symbols) != 0:
84                  current_symbol = np.random.choice(imminent_symbols)
85                  special_symbol_to_distance_to_next_occurrence[current_symbol] = None
86                  imminent_symbols.remove(current_symbol)
87                  for symbol in imminent_symbols:
88                      special_symbol_to_distance_to_next_occurrence[symbol] = 1
89              else:
90                  current_symbol = np.random.choice(symbols, p=base_probabilities)
```

```
91
92              print(current_symbol.name, file=file)
93
94              # on error, all probabilities reset (simulating the system going back to
                    ↪ the initial state)
95              if current_symbol == ERROR:
96                  special_symbol_to_distance_to_next_occurrence = {symbol: None for
                        ↪ symbol in special_symbols}
97              elif current_symbol in special_symbols and len(current_symbol.
                    ↪ influenced_symbols) > 0:
98                  for symbol_influence in current_symbol.influenced_symbols.values():
99                      if random.random() < symbol_influence.probability_of_causation:
100                         mean = symbol_influence.mean_distance_in_case_of_causation
101                         std_dev = symbol_influence.std_dev_in_case_of_causation
102                         distance_to_next_occurrence = \
103                             stats.truncnorm((0 - mean) / std_dev, np.inf, loc=mean,
                                    ↪ scale=std_dev)\
104                             .rvs().round().astype(int)
105                         old_distance = special_symbol_to_distance_to_next_occurrence[
                                ↪ symbol_influence.target]
106                         if old_distance is not None:
107                             distance_to_next_occurrence = min(old_distance,
                                    ↪ distance_to_next_occurrence)
108                         special_symbol_to_distance_to_next_occurrence[symbol_influence.
                                ↪ target] = \
109                             distance_to_next_occurrence
110
111     return file_path
112
113
114 def read(file_path):
115     with open(file_path, "r") as file:
116         lines = [line.rstrip('\n') for line in file]
117
118     df = pd.Series(lines, name="symbol").to_frame()
119     df["is_error"] = df["symbol"] == "error"
120     return df
```