SERVICE VIRTUALIZATION USING RECORDED INTERACTIONS

by

Hasan Ferit Enişer

B.S., Computer Engineering, Boğaziçi University, 2015

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University

2017

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor Assoc. Prof. Alper Şen for guiding me with his experience and skill, and fellow jury members Assoc. Prof. Arzucan Özgür and Assoc. Prof. Feza Buzluca for their invaluable feedback. I am grateful to my wife Ezgi Enişer, my parents Cumhur and Sema Enişer and my brothers Taha and Tayyip Enişer, for their unlimited support. I would not complete this thesis without them. I also thank my fellow lab partners, for discussing new ideas. Last, I thank my old friends Erenay Dayanık and Umutcan Ayasandır for their all creative ideas.

ABSTRACT

SERVICE VIRTUALIZATION USING RECORDED INTERACTIONS

Today's enterprise software systems are much more complicated than the past. Increasing number of dependent applications and heterogeneous technologies makes testing of such systems challenging due to multiple reasons including unavailability of components, high cost of using services and conflicting schedules of different development teams. Therefore in such software systems, it may be more convenient to use virtual components instead of the real ones. Service virtualization is a technique to mimic the behavior of a real component. Services are classified into two groups namely; stateful and stateless services. In this thesis, we introduce techniques for creating virtual copies of both stateful and stateless services. To the best of our knowledge, this is the first work to create virtual services for stateful services. We employ bioinformatics and machine learning algorithms in developing our solutions. We demonstrate the validity of our approaches on data sets collected from real life services and obtain promising results.

ÖZET

KAYITLI ETKİLEŞİMLERİ KULLANARAK SERVİS SANALLAŞTIRMA

Günümüzde yazılım sistemleri geçmişe göre çok daha karmaşık bir hale gelmiştir. Birbirine bağımlı uygulamaların, birbirinden farklı teknolojilerin aynı anda kullanıldığı yazılım sistemlerinin test edilmesi bir çok farklı nedenden dolayı zordur. Bu nedenler, bağımlı bileşenlerin ulaşılamaz olması, üçünü parti servisleri kullanmanın yüksek maliyeti ve farklı takımların takvimlerinin çakışması gibi sıralanabilir. Bu sebeplerden dolayı, bu tür yazılım sistemlerinde gerçek bileşen yerine sanal bir kopyayı kullanmak kolaylık sağlayabilmektedir. Servis sanallaştırma verilen bir bileşenin davranışlarını taklit etmeye yarayan bir tekniktir. Servisler durumsal ve durumsal olmayan olmak üzere iki sınıfa ayrılır. Bu tezde, hem durumsal hem de durumsal olmayan servislerin sanallaştırılması için yeni teknikler önerilmiştir. Bizim bilgimize göre bu çalışma durumsal servislerin sanallaştırılmasını konu alan ilk çalışmadır. Bu çalışmada bioinformatik ve makine öğrenmesi algoritmaları kullanılmıştır ve çalışmamızın geçerliliği gerçek servislerden toplanan verilerle test edilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

C	The set of all classes to be predicted
D	A dictionary whose keys are requests or responses and values are cluster labels.
E_{ij}	Expected value of ith test and jth output
K	K parameter in modified k Nearest Neighbour.
P_{ij}	Predicted value of ith test and jth output
RP	Response contents to be predicted
Y_{ij}^{c}	Indicator of jth output of ith test is equal to c in expected outputs.
Z_{ij}^{c}	Indicator of jth output of ith test is equal to c in predicted outputs.
τ	Cluster separation threshold.
$\mathbb{1}$	Indicator function
$\Re$	A list of requests or responses.

LIST OF ACRONYMS/ABBREVIATIONS

ANN	Artificial Neural Networks
ATS	Airline Ticketing Service
AUT	Application Under Test
CBV	Classification Based Virtualization
EFSM	Extended Finite State Machine
EMR	Exact Matching Ratio
IFR	Identical Field Replacement
kNN	k Nearest Neighbour
LSTM	Long Short Term Memory
MSA	Multiple Sequence Alignment
NW	Needlman-Wunsch
OSV	Opaque Service Virtualization
RGE	Response Generation Engine
RNN	Recurrent Neural Networks
RS	Real Service
SMR	Subset Matching Ratio
SOA	Service Oriented Architectures
SSBV	Sequence-to-Sequence Based Virtualization
VS	Virtual Service
WSDL	Web Service Definition Language

1. INTRODUCTION

Today's enterprise software systems have higher number of interconnected components, interdependent teams and heterogeneous technologies than the past due to increasing number of composite applications in software systems. Multi-layered and interdependent architectures of such applications like Service Oriented Architectures (SOA) provide a lot of benefits but at the same time they increase the complexity and introduce new constraints. In such complicated software systems, developers would spend considerable amount of time to access a component e.g mainframe or doing data set up instead of development or testing because of the conditions below:

- Still evolving or uncompleted services.
- Limited capacity or availability of services at inconvenient times.
- Services that are controlled by a third-party that grants restricted or costly access.
- Services that are needed simultaneously by different test teams with various set up and requirements.

Therefore software development sometimes requires *test doubles* that allows developers to decouple the application from the dependencies when testing the application under test (AUT).

The most common test doubles used in practice are stubs and mocks. Stubs return hard-coded responses that are tightly coupled to the test suites. They are suitable for personal use or sharing with testers. Wider sharing can cause issues related with software platform and deployment infrastructure dependencies. Mocks are most useful when you have a large test suite where stubs will not be adequate since each test case requires a different data set up. Mock objects are also shareable between testers but wider sharing is limited. Therefore, stubs and mocks do not provide the re-usability to move projects forward. An alternative to these options is *service virtualization (emulation). Service Virtualization* is a practice to create virtual copies of a dependent component. Virtual services are suitable for sharing within a team and across teams.

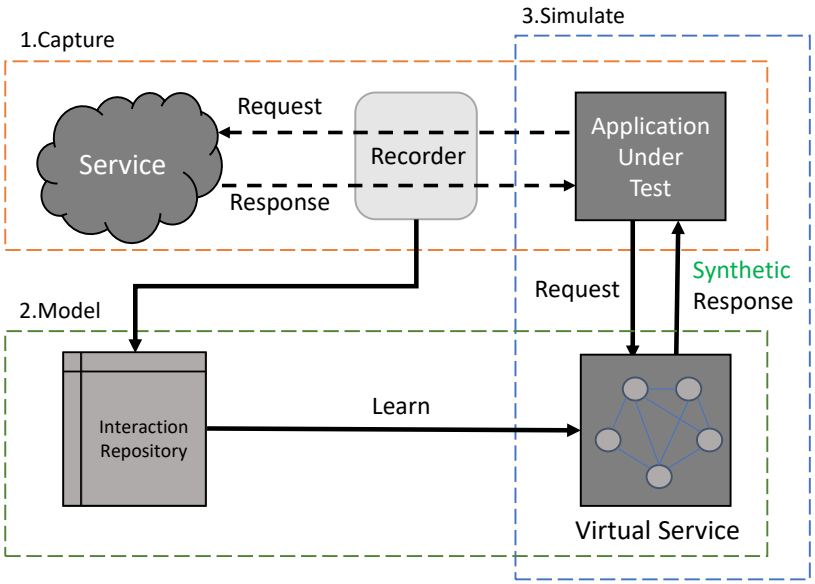

Figure 1.1. Overview of service virtualization.

They can simulate performance and data characteristics of the real component. Also, service virtualization is suitable for complex and very large legacy software that has many dependencies. [1, 2]

A good service virtualization solution creates virtual services that [3]:

- Simulate the behavior of the real component
- Synthesize responses with realistic data
- Synthesize responses with configurable throughput

The fundamental process of service virtualization practice can be abstracted into three phases; capture, model and simulate. First, the required information to virtualize a service is captured and a model is constructed using the captured data. This model corresponds to the core of a *virtual service*. Last, the virtual service is deployed to development environment and employed instead of the real one.

There are mainly two automated methods to create virtual services [3]. The first method uses detailed knowledge of the target service's protocol and message structure such as Web Service Definition (WSDL). This method guarantees to generate a valid response for an incoming request. However, this method is not suitable when specifications are not available, e.g third party services. The second method, which we also employ, observes the system in action and records request and response messages between the system to be tested and the service to be virtualized. A general outline can be seen in Figure 1.1. Recorded messages are kept in a transaction repository (capture) to train a model to simulate the response behavior of the real component (model). Application under test sends requests to the virtual service when the real service is not available for some reason (simulate). This is a more comprehensive approach.

Services can be examined in two groups: the ones that keep state information (stateful services) and the ones that do not (stateless services). An example stateful service can be a service which a client can create, read, update or delete calendar events. The service keeps track of the state information to return the true response when a calendar event is requested. An example of stateless service can be a service which returns the capital city of the given country in the request. In this case, the state of the service does not change with the requests. Current service virtualization solutions have limited accuracy and performance and they are not applicable to services that have a stateful behavior. Virtualization of stateful services is a harder task since state behavior of the service has to be simulated.

Our main contributions in this thesis are as follows:

- We propose automated service virtualization approaches that can handle services with stateful and stateless behavior.
- Our solution can handle messages with arbitrary message format for each of the stateful and stateless services.
- We employ bioinformatics and machine learning techniques to obtain a virtual service from recording of request response pairs.
- We implement our techniques in a tool and validate our approach on real services.

In Chapter 2 we describe the related work and in Chapter 3 we provide the necessary background. Chapter 4 and Chapter 5 explain and evaluate the techniques introduced for virtualizing stateless and stateful services in order. Last, in Chapter 6 we conclude our research and discuss future work.

2. LITERATURE SURVEY

In this chapter, we introduce studies that are related to the topic of this work. Service virtualization is a relatively new practice. Thus, there are very few studies that brings a solution to the challenges encountered while virtualizing services.

In [3, 4], the authors describes the basics of service virtualization. They also explain the benefits, capabilities and best practices of service virtualization. Nizamic *et al.* [5] presents a case study of using virtual services by creating a simulation environment for a Dutch law for supporting people that have a chronic disease or disability.

The latest studies that are related to service virtualization are in [6–10]. Those works employ some bioinformatics algorithms to virtualize a service. The latest one of these studies, namely [10] improves all previous results and presents a technique called Opaque Service Virtualization (OSV). OSV consists of two phases: analysis and runtime. In analysis phase, they record transactions between the client and the service. Then recorded requests are clustered and a prototype for each cluster is obtained where each prototype refers to a request type. In runtime phase, a given request is matched to the closest prototype and a synthetic response is generated. OSV performs best if the messages have a fixed length or the length info is encoded in the message [11]. In addition to that OSV technique can not virtualize stateful services. In this work, we address these limitations.

Newly emerged containerization tools like Docker [12] are orthogonal to service virtualization. For example, a virtual service can be ran in a container.

Leading software companies such as IBM, HP, CA, SmartBear, and Parasoft are also provide various commercial service virtualization tools. These tools are compared and evaluated in reports [13, 14]. A recent survey of service virtualization vendors conducted on real users can be found in [15].

One of the goals of this work is bringing a solution to stateful service virtualization problem. In other words, we want to simulate the state behavior of the service to be virtualized. Therefore, we surveyed studies in the field of black-box model inference [16–33] which builds models by observing executions. There are numerous works that infer model by relying on event ordering in the execution and/or data attached to them.

K-tails [16] is the most basic and well-known algorithm for black-box model inference. It serves as a basis for many model inference algorithms [17–22]. These algorithms (1) extend k-tail to achieve higher precision or recall [17,19–22], (2) enhance the models with information about event probabilities [18].

Lorenzoli *et al.* [22] propose GK-tails which is also an extension of K-tails. It enhances K-tails by combining the program state and method parameters information together with event types (method invocations) to state machine construction. GK-tails produces much more comprehensive models. However, it relies on a good range of data for each event type to construct correct models [23].

Walkinshaw *et al.* [23] also combine the information of event ordering and data values of events. They use data values to predict the next event by employing a classification algorithm.

Dallmeir *et al.* [24] introduce ADABU to infer correct program behavior. They observe actual program executions to construct state machines, called *object behavior models*. In ADABU program states are abstracted using predicates. However, these predicates are predetermined (e.g., ADABU abstracts integers only as negative, zero, or positive).

Synoptic [26], CSight [27], and Perfume [28] use the CEGAR [34] approach to create a coarse initial model, and then refine it using counterexamples that falsify temporal invariants.

Krka *et al.* [29], divide dynamic model inference strategies into four classes namely, *traces only* [16], *invariants only* [25], *invariant-enhanced-traces* [22] and *trace-enhanced-invariants* [30]. They introduce one technique for each strategy and compare these techniques. They show that their algorithm in *trace-enhanced-invariants* class works best for several popular Java libraries.

None of these black-box model inference techniques are dedicated to predict the outcome (response) of a specific event (request). Their motivation to construct a state model includes making debugging and documentation easier [28, 29], automatic test case generation [22] and reverse engineering [23].

State machine extraction is also studied in neural networks field. Especially recurrent neural networks are used in the literature for model extraction [31–33]. However, state models extracted in these works do not focus on predicting outcomes of specific events.

3. BACKGROUND

This section provides the background necessary for the rest of this thesis. First, we make necessary definitions used throughout the thesis, then we present Needlman-Wunsch and ClustalW algorithms. Last, we introduce a modified version of k-nearest neighbour algorithm.

3.1. Definitions

Let $Req, Res, T_{req}, T_{res}, C_{Req}, C_{Res}$ be a finite set of requests, responses, request types, response types, request contents, response contents, respectively.

A request, $req \in Req$ is a 2-tuple $(type, content)$, where $type \in T_{req}$ and $content \in C_{req}$. A response, $res \in Res$ is also a 2-tuple $(type, content)$, where $type \in T_{res}$ and $content \in C_{res}$.

An interaction is defined as a request response pair: (req, res) with $req \in Req$ and $res \in Res$. We define an *interaction trace*, $it \in IT$ as a finite sequence of interactions observed during the execution of the service; (req_1, res_1), (req_2, res_2), $\ldots$, (req_n, res_n). A Interaction Repository (IR) keeps all recorded interaction traces.

Given an interaction trace it, we define the history, h, of a request req_i, as the following: $h_{req_i} = (req_1, res_1)$, $\ldots$, (req_{i-1}, res_{i-1}), (req_i), that is, the trace ends with req_i and the corresponding response is absent. Similarly, the *k history of a request* req_i is shown as: $h^k_{req_i} = (req_{i-k}, res_{i-k})$, $(req_{i-k+1}, res_{i-k+1})$, $\ldots$, (req_i). The set of all histories for all requests in all interaction traces is denoted by H, whereas the set of all k histories of all requests in all interaction traces is denoted by H_k.

We divide the services into two classes; *stateless* and *stateful services*. A *stateful service*, $StatefulS : H \rightarrow Res$, is a function from the set of histories to the set of responses. Hence, given the history of a req_i we can determine its response $resp_i$.

A stateless service, $StatelessS : Req \rightarrow Res$, on the other hand, is a function from the set of requests to the set of responses. The response of a request can solely be determine from the request itself. Informally, in a stateless service, a request's response is not dependent on the request's history. We describe sample stateless and stateful services in later chapters.

A *virtual service*, $VS : H_k \rightarrow Res_{syn}$ is a function where H_k is k histories of all requests, Res_{syn} is a finite set of synthesized responses, where k equals to 1 for a stateless service. On the other hand, k is specified by the user for a stateful service. A synthesized response is an artificial response which is the same with the actual response in the perfect case.

3.2. Needleman-Wunsch (NW) Algorithm

Needleman-Wunsch (NW) algorithm [35] is a dynamic programming algorithm for aligning two sequences that is originally developed in bioinformatics. The algorithm finds the globally optimal pairwise alignment defined by the chosen scoring function in *O(m.n)* time, where m and n are the lengths of the sequences. The scoring function gives a score for every possible alignment between two sequences by calculating matches, mismatches, and gaps in each alignment and chooses the alignment with the best score.

Below, there is an example of pairwise alignment. Sequences are taken from Table 4.1 which is an interaction repository that consists of recorded transactions. Consider requests with id 2 and 5 when wrapped with necessary content names (i.e. type, username) and other other characters to form JSON [36] format:

```
{type:searchuser, username:---the-popularuser}
{type:searchuser, username:anotherpopularone-}
```

Figure 3.1. Pairwise alignment of two requests.

"-" characters show the gaps inserted in favor of matches during the alignment to maximize the score. We use NW algorithm in both off-line and on-line phases of our method to calculate the distances between messages. We do not use the exact score produced by the algorithm to measure the distance. Instead we normalize them by dividing the number of matching characters to the length of the alignment and subtract the normalized value from 1.

3.3. ClustalW Algorithm

ClustalW [37] is a *multiple sequence alignment* (MSA) algorithm which is shown to be an *NP-complete* problem [38]. ClustalW is a heuristic algorithm that is originally used for aligning multiple genes and it produces successful results for most cases. ClustalW consists of three steps:

- *Pairwise alignment:* A distance matrix is constructed by aligning each pair. We use NW pairwise alignment algorithm in our implementation.
- *Guide Tree Construction:* The trees used to guide the final multiple alignment process are calculated from the distance matrix of step 1 using the Neighbor-Joining method [39].
- *Progressive Alignment:* The basic procedure at this stage is to use a series of pairwise alignments to align larger and larger groups of sequences, following the branching order in the guide tree.

Below there is an example of multiple sequence alignment. Consider the requests with id 2, 3, and 5 in Table 4.1 when wrapped with necessary content names (i.e. type, username) and other other characters to form JSON format. According to ClustalW, we first make a pairwise alignment between id 2 and 5 as shown above. Then we align this pair with id 3 using several heuristics.

```
{type:searchuser, username:thenonexisting-user}
{type:searchuser, username:----the-popularuser}
{type:searchuser, username:an-otherpopularone-}
```

Figure 3.2. An example multiple sequence alignment of three requests.

3.4. Modified K-Nearest Neighbour (kNN) for Clustering

Typically clustering algorithms require the number of clusters that will be generated as an input. In our case, we want to automate this step and not require this number as an input parameter. For this purpose, we modify the well-known k-Nearest Neighbour (kNN) classification algorithm [40] and employ it to cluster the recorded data. Figure- 3.3 details our technique.

Require $\Re$: A list of requests or responses.
τ : Cluster separation threshold.
K : K parameter.
Ensure D : A dictionary whose keys are requests or responses and values are cluster labels.
clusterLabel $\leftarrow 0$
$C[\Re[0]] \leftarrow$ clusterLabel {The first element gets its label.}
clusteredMessages.*append*($\Re[0]$)
for each r in $\Re$ **do**
  $distances, labels \leftarrow$ findKNearestNeighbour(K,r,$clusteredMessages$)
  if min($distances$)$> \tau$ **then**
    $clusterLabel = clusterLabel + 1$ {Create a new cluster.}
    $D[r] \leftarrow clusterLabel$
  else
    $D[r] \leftarrow$ getMostOccurringLabel(labels) {Most occurring *label*.}
  end if
  clusteredMessages.*append*(r)
end for

Figure 3.3. Modified k-Nearest Neighbour

We assign requests or responses to clusters one by one. In Lines 3,4 and 5 we start with only one cluster with one element in it. For each request and response, we find k closest neighbours in the already clustered messages (Line 7). We choose k=*1* in this work. We measure closeness by the distance calculation mentioned in NW. Then we check whether the minimum of the distances for the request or response is larger than the cluster separation threshold τ, given as an input. If yes (Line 8), then this data point is located far from the other points. Therefore we put it into a new cluster (Line 9). If no (Line 11), then we just assign the most occurring cluster label of all nearest neighbours (Line 12). In Line 14 we append the current message to the *clusteredMessages* list.

3.5. Artificial Neural Networks, Recurrent Neural Networks and Long-Short Term Memories

Artificial neural networks (ANN) that are inspired by human brain are introduced to simulate the incredible abilities of human brain in applications such as vision, speech recognition and learning [41]. An artificial neural network consists of layers of perceptrons and edges connecting them. However, traditional neural networks have a major shortcoming. Humans build their thinking on top of previous information that they have, rather than start their thinking from scratch every moment. ANNs do not consider historical data when producing outputs as shown in Figure 3.4.

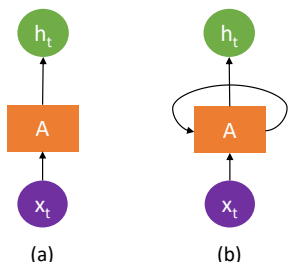

Figure 3.4. Given, A is a chunk of neural network, X_t is input and h_t is output, (a) shows the architecture of a traditional neural network and (b) shows the outline of RNNs.

Recurrent neural networks (RNN) can address this issue with a recurrent link in the network as shown in Figure 3.4, allowing them to carry an information from one step to the next one. However, sometimes we need to carry an information for further steps. For example, ten requests before the current request can affect the expected response of the current request. Traditional RNNs have limited ability to carry an information for further steps. A special form of RNNs, namely, *Long-Short Term Memory* (LSTM) networks [42] are used for this task. The LSTM's strength is learning on data with long range temporal dependencies [43]. LSTM networks are more complex than traditional neural networks and RNNs.

4. STATELESS SERVICE VIRTUALIZATION

In this chapter we introduce a technique named FancyMock to learn the function VS defined earlier for virtulization of services that do no keep state information. Figure 5.1 demonstrates the general overview of FancyMock technique. It consists of two main phases: Off-line analysis and Response Generation Engine (RGE). In the first phase recorded interactions are mined and the information that is necessary for producing responses is learned. This information is used in formation of RGE which is the second phase. RGE simulates the behavior of the real service when it is not available. We now present an example stateless service, then describe the details of Off-line Analysis as well as Response Generation Engine shown in Figure 5.1. Last, we evaluate FancyMock.

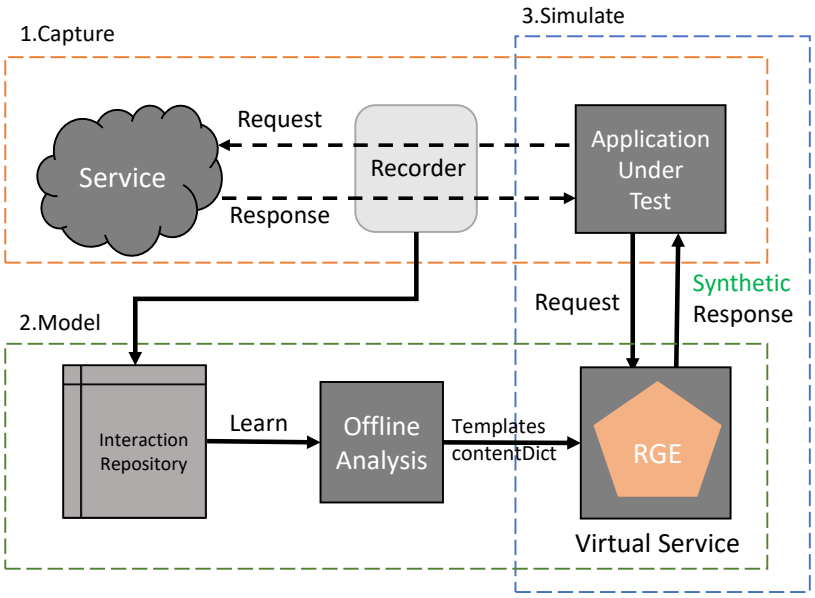

Figure 4.1. Overview of stateless service virtualization.

4.1. A Sample Stateless Service

In this section, we demonstrate a sample stateless service where a user can search for users or songs by providing the necessary content. A response includes corresponding information about the song or the user searched.

Consider the interactions demonstrated in Table 4.1. For the first request, the client searches a song with name *somesong* by *somesinger*. The corresponding response includes the name of the song *somesong* and also the url *someurl*. Assume that message exchange is done in JSON [36] format in this service. Thus, when wrapped with the necessary field names the first request and the first response are shown in figure below:

```
{type:searchsong, title:somesong, singer:somesinger}
{type:searchsongresponse, title:somesong, streamurl:someurl}
```

Figure 4.2. The first request and the first response in JSON format.

For the second request, the client searches for a user by providing the username *thepopularuser*. The response includes the first name and the last name of the user searched. It is shown in figure below in JSON message format:

```
{type:searchuser, username:thepopularuser}
{type:searchuserresponse, fname:John, lname:Doe}
```

Figure 4.3. The second request and the second response in JSON format.

The third and the fourth requests are replied with error messages. The fifth and the last ones are similar to first two ones.

4.2. Off-line Analysis

In this phase, we generate templates of request and response messages using the recorded interactions. The response generation phase will make use of templates to generate a valid response to a request that does not occur in the recording. In case the request occurs in the set of recorded requests the response is simply the response of the recorded request. Also we make a content analysis that is used in generating more diverse responses.

Table 4.1. An artificial transaction repository consisting of two different request types. Requests and responses with the same IDs correspond to each other.

	Requests		**Responses**	
	Type	**Content**	**Type**	**Content**
1	searchsong	somesong, somesinger	searchsongresponse	somesong, someurl
2	searchuser	thepopularuser	searchuserresponse	John, Doe
3	searchuser	thenonexistinguser	searchuserresponse	Not found
4	searchuser	anothernonexistinguser	searchuserresponse	Not found
5	searchuser	anotherpopularone	searchuserresponse	Jane, Roe
6	searchsong	anothersong, somesinger	searchsongresponse	anothersong, anotherurl

We start with clustering of recorded messages that is necessary step for extraction of templates. Then we explain the details of template extraction procedure. Last, we present content analysis step.

4.2.1. Clustering of Messages

A naive way to generate a response for an incoming request that does not exist in the interaction repository is to check the distance between the incoming request and all the requests in the repository then finding the closest request's response. However, performing this check for a large repository can be time consuming. Instead, we first put requests/responses within a distance threshold in the same cluster. This threshold value is set to 0.8 based on experiments. In the perfect case each cluster refers to a request or a response type. For example, we expect that all *searchsong* requests are collected into the same cluster. Then we generate a template for each cluster. When an incoming request that does not exist in the interaction repository arrives we simply calculate the distance between this request and all templates. This allows us to quickly return a response. We generate two sets of clusters, one for requests and one for responses. A possible set of clusters for the interaction repository in Table 1 with two request clusters and three response clusters is as follows. Each cluster contains ids of requests or responses.

Request Clusters: $\{\mu_1$:{1, 6}, μ_2:{2, 3, 4, 5}$\}$
Response Clusters: $\{\nu_1$:{1, 6}, ν_2:{2, 5}, ν_3:{3, 4}$\}$

After clustering, we need to find correspondences between request and response clusters. This is because responses of messages in the same request cluster can be in different response clusters. For example, request messages 2 and 3 are in request cluster μ_2, whereas their responses are in response clusters ν_2 and ν_3. We find which request types are replied with which response types in the interaction repository. The response generation process takes these found correspondences into account to synthesize more realistic responses. We refer to this process as *cluster mapping*.

Although clustering helps improve response generation time and needs to be done only once, even generating the clusters can be costly because during clustering, the distance between all pairs of requests/responses is calculated. The resulting set of clusters contains all requests/responses in the interaction repository. In order to improve performance, we use a simple solution where during clustering we put a bound on the number of elements in each cluster. In this case although we have the same number of clusters as before, we now reduce the number of alignments because of cluster bound, hence we obtain a speed up as will be shown in experiments. The number of alignments decreases from quadratic to linear. We call this approach *pre-sampling*. For example, if we apply pre-sampling with cluster bound three, μ_2 above could be as follows: μ_2:{2, 3, 4}.

4.2.2. Template Extraction

We use *ClustalW*, a *multiple sequence alignment* (MSA) algorithm to generate a template for every cluster, where a template is a representative of that cluster. We apply this process for both request and response clusters. After MSA is constructed for each cluster we check for consistency of the alignment. If there is no *consistency* between the characters at the same index of alignment, we put a wild-card character to that index at the template.

We define *consistency* as a consensus higher than a threshold. In our model 80% of the characters need to be the same to form a consensus. If there is a consensus on a character we put it to the corresponding index of the template. According to this definition, the template of the request cluster μ_2 is is shown in figure below:

```
{type:searchuser, username:#######}
```

Figure 4.4. Template of the request cluster μ_2.

Note that, in μ_2's template *searchuser* field is not replaced by wild-card characters. This is because all characters at the same index throughout the cluster alignment are the same. This approach preserves essential parts of traces, while filling contents with wild-card characters '#'.

We choose the wildcard characters outside of the character set of the interaction repository. Therefore there can be no match, for this reason we basically ignore mismatches with wildcard characters while calculating distances with templates. Our teamplate extraction technique is based on the prototype extraction in [10].

4.2.3. Content Analysis

Previous works have a limited ability to generate realistic responses [7,8,10]. We aim to generate diverse and thus realistic synthetic responses. In this step we find each content of each recorded response. To find the contents of a response message, we align the message with the template of the cluster that it belongs to. The characters of the response message that are matched with wildcard characters of the template are assumed to be contents. These fields are then kept in a dictionary. An example of how the contents are extracted is shown in figure below:

```
{type:searchsongresponse, title:somesong, streamurl:someurl}
{type:searchsongresponse, title:########, streamurl:#######}
```

Figure 4.5. Alignment for content extraction.

By considering the alignment between the response (the first line) and its template (the second line) shown above, we see that *somesong* and *someurl* are contents of this response. We apply this procedure to all response messages in all response clusters and store them in a dictionary called *contentDict*. An example *contentDict* is shown in Table 4.2.

Table 4.2. Keys and values of *contentDict* for Table 1 are shown. For example (2,1) stands for the first content of the responses in the second response cluster.

Key	Value
(1,1)	[somesong, anothersong, ...]
(1,2)	[someurl, anotherurl, ...]
(2,1)	[John, Jane, ...]
(2,2)	[Doe, Roe, ...]

4.3. Response Generation Engine (RGE)

Response Generation Engine (RGE) is the part that is responsible for generating valid, logical and diverse responses by using the information gathered in off-line phase. At the end of the off-line phase we have templates for each cluster and a *contentDict*. The process of RGE consists of 3 main steps:

(i) Select *base* messages (Base Message Selection):
- Find the nearest request cluster template.
- Find and select one of the possible response clusters.
- Pick a *base response* from the selected response cluster. Find the corresponding *base request*.

(ii) Apply Identical Field Replacement (IFR) procedure:
- Find indices of the identical contents between *base request* and *base response*.
- Apply the identical field replacement procedure for the *incoming request* and the *base response* by replacing its corresponding fields.

(iii) Replace the rest of the contents using *contentDict* (Diversification).

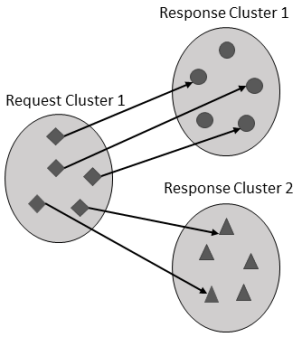

Figure 4.6. Three of the requests are replied with responses from Response Cluster 1 (Type 1) and two of them are replied with responses from Response Cluster 2 (Type 2). In this case, response type of an incoming request which is close to Request Cluster 1 will be in Type 1 with $\frac{3}{5}$ probability and in Type 2 with $\frac{2}{5}$ probability.

4.3.1. Base Message Selection

First, we align the incoming request with each request cluster template and find the closest one. Since the same type of requests may be replied with different types of responses, we can select one of the possible response clusters that we find in off-line phase. Figure 4.6 shows a case of cluster mapping and the probabilities of selecting response clusters. After selecting a response cluster, we randomly pick one of its elements as the *base response* and pick the corresponding request as the *base request*. The generated response will be the modified version of *base response*. Modifications are done in IFR and Diversification steps.

4.3.2. Identical Field Replacement (IFR)

For this step, we find the identical contents between the *base response* and *base request*. An example is shown below. In this example *somesong* is an identical content between the base request and the base response. However, *searchsong* is not considered as an identical content because this field is a fixed field of the template (it is common to all messages in the cluster shown earlier).

Base request:
`{type:searchsong, title:somesong, singer:somesinger}`
Base response:
`{type:searchsongresponse, title:somesong, streamurl:someurl}`

This example shows us that the first content of base request and the first content of base response are identical. Next, we replace the first content of base response with the first content of incoming request. Remember that incoming request and base request belong to the same cluster and the base response is selected form one of the possible clusters. Unlike [7], we consider the identicality between contents instead of each character. Also, notice that if the inserted content is longer than the one in *base response* then we extend the field, if shorter we shrink it. If there is no identical content, this step will be skipped.

4.3.3. Diversification

The rest of the content of base response are chosen randomly from the relevant key value pair of *contentDict*. In this way, we diversify the generated responses while preserving identical fields between request and response. From a tester's perspective more diverse responses lead to higher coverage. An example for this step is shown in figure below. Synthesized response's *streamurl* field is chosen from *contentDict*.

`{type:searchsong, title:asong, singer:coolsinger}`
`{type:searchsongresponse, title:asong, streamurl:randomurl1}`

Figure 4.7. Incoming request and synthesized response (modified base response)

4.4. Evaluation

In this section, we present experiments conducted to evaluate our approach which we implemented in FancyMock tool. We, first describe the subject services that are virtualized and then explain our evaluation setup and last, demonstrate the results.

Table 4.3. Datasets

Name	Format	#Request Type	#Traces
RIQS	XML	4	1200
SoundCloud API	JSON	3	1200
WeatherUgrnd API	JSON	1	1200

4.4.1. Subject Services

We tested our approach on three different datasets collected from live systems for realistic results. The first dataset was collected from a Resident Information Query System (RIQS). In this system, one can query a person with an ID or a full name. The expected response consists of detailed information including the date of birth, place of birth, and address. Interactions collected from RIQS correspond to previous queries sent from a dependant application. No policy followed while selecting persons for query. The second dataset was collected from SoundCloud API for songs. SoundCloud provides a *RESTful* API and message exchange is done in JSON format. We implemented a Python script that can connect to the API, then send and receive messages. We sent requests to the SoundCloud API with totally random parameters. Those parameters include track id, user id etc. Finally, we collected historical weather data from WeatherUnderground API, again a *RESTful* service where messages are in JSON format. Again, we implemented a Python script for collecting data from the API. We chose ten cities around the world and specified various dates throughout the year for the parameters of the requests that we sent to WeatherUnderground API. Table 4.3 shows the number of request types and traces for each dataset.

4.4.2. Evaluation Setup

4.4.2.1. Metrics. To evaluate our approach, we measured four metrics in experiments. The validity of response messages, average distance between generated responses (diversity), off-line phase running time, and average response generation time. Also, we conducted experiments to show the effect of pre-sampling on template extraction.

A message can be *invalid* in two ways: either the produced response does not satisfy the message format (*JSON, XML, etc.*) or the produced response does not belong to the expected response cluster, for example a *searchuser* response is produced while *searchsong* is expected.

Diversity is defined as the dissimilarity between generated valid responses. We aim to generate diverse responses in favor of coverage. We calculate diversity by finding the distance between each pair of same type of responses e.g two *searchuser* responses. Then, we take the average of the distances calculated for each response type.

Off-line phase running time refers to the time passed during off-line analysis phase. The average response generation time refers to the average time passed between the arrival of the request and the generation of the response. Ideally, the real service's average response time should be an upper bound on this time. Nevertheless, this is not always possible. We consider an average response generation time *acceptable*, if it is less than the maximum measured response time of the real service. One can also put a lower bound for response generation upon specific needs. However this is a trivial task that can be achieved by adding delay.

4.4.2.2. Experimental Design. We compared our approach with a baseline approach denoted by Baseline. The Baseline approach is our implementation of the algorithm in [10] which does not include pre-sampling, content analysis and diversification, it includes a simpler version of IFR and employs a different clustering algorithm as described in [10]. We shuffled each dataset and recorded 60% of each in the interaction repository for training and the rest are set for testing, in the experiments. Experiments were run on a server with 32 GB memory and Intel Xeon E5520 2.27GHz CPU.

Table 4.4. Validity rates of generated responses.

Name	Baseline	FancyMock
RIQS	48.6%	94.8%
SoundCloud API	18.3%	76.1%
WeathrUndrgrnd API	21.2%	100%

Table 4.5. Average distances between pairs of generated responses.

Name	Baseline	FancyMock
RIQS	0.21	0.71
SoundCloud API	0.09	0.66
WeathrUndrgrnd API	0.11	0.43

4.4.3. Evaluation Results

4.4.3.1. Validity Evaluation. Table 4.4 shows our validity evaluation results. This table shows that FancyMock can generate valid responses in a protocol-independent fashion and preserves the message format. Most of the responses generated by FancyMock are valid while that is not the case for Baseline.

4.4.3.2. Diversity Evaluation. Table 4.5 shows that responses generated by Baseline have limited ability to generate diverse responses. FancyMock, on the other hand, synthesizes responses which are different from each other.

4.4.3.3. Performance Evaluation of Off-line Analysis. In this case, we measured the time passed during off-line analysis. We compare FancyMock with Baseline, which does not use pre-sampling and instead calculates the distance for each pair of elements in the interaction repository. We conducted this experiment with datasets of different sizes, including 400, 800, and 1200 traces so that we can see the performance change with the increase in interaction repository size. Table 4.6 shows the results. We can see that our pre-sampling and clustering techniques are helpful when dataset is large (800 or 1200) which is typically the case.

Table 4.6. Off-line Analysis Phase Running Times (hh:mm:ss)

	Baseline			FancyMock		
	400	800	1200	400	800	1200
RIQS	01:53:03	08:47:17	20:11:08	01:22:07	02:37:08	03:54:12
SoundCloud API	00:08:47	00:16:03	00:41:06	00:18:23	00:35:12	00:52:34
WeathrUndrgrnd API	00:10:30	00:39:43	01:24:34	00:16:04	00:32:58	00:52:12

Table 4.7. Average Response Time (seconds)

	Baseline			FancyMock			Real Service		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
RIQS	1.42	1.91	1.68	2.50	3.92	3.09	0.02	8.94	0.61
SoundCloud API	0.47	0.91	0.72	0.33	1.22	0.86	0.10	0.54	0.45
WeathrUndrgrnd API	0.69	0.84	0.74	0.57	1.31	0.66	0.05	0.96	0.63

Another factor that affects running time is the average message length. For SoundCloud case our approach did not outperform the Baseline technique since the requests are very short for this dataset. FancyMock makes a difference when the message length gets longer.

4.4.3.4. Average Response Generation Time. Table 4.7 presents our average response generation time results. For this experiment we also show results for the real service. From the table we observe that synthesizing responses takes longer time with our approach in general. This is due to the extra steps that we include in RGE in order to generate a diverse and logical response (*Diversification and IFR*). There is an exception in WeatherUnderground API. In this case FancyMock generates responses faster than Baseline on the average. This is because there are no identical contents between the requests and the responses. Therefore FancyMock goes directly to Diversification step by skipping IFR. However, Baseline takes identicality of characters into account. Therefore for this case FancyMock skips IFR step but Baseline makes replacements.

Table 4.8. Effect of pre-sampling on correct template extraction.

Cluster Bound	Correctness Rate (%)
5	82.5
10	92.5
20	95.0
25	95.0
Whole Cluster	95.0

4.4.3.5. Effect of Presampling on Correct Template Extraction. Finally, we present how pre-sampling affects the correctness of the template extracted. In a correct template, we expect that essential parts are preserved and all contents are filled with wild-card characters unless those fields are common throughout the cluster. An example of correct template is shown in figure below:

```
{type:searchsong, title:########, streamurl:#######}
```

Figure 4.8. An example of correct template.

The two templates shown below are examples of incorrect templates. The first one's title field does not consist of purely wild-card characters. The problem with the second one is that an essential part is replaced with wild-cards.

```
{type:searchsong, title:####s###, streamurl:#######}
{type:searchsong, t##le:########, streamurl:#######}
```

Figure 4.9. Examples of incorrect templates.

This measurement is independent of the dataset and whether it is a request or response. Therefore we used only requests of RIQS. We compared cluster bounds including 5, 10, 20, 25 and the whole cluster. Number of elements in the whole cluster changes for each request type in RIQS. We conducted the measurement 10 times for each bound and took the average of correct templates.

Table 4.8 shows our results. From the table, we see that correctness rate increases with the number of elements in the cluster. However, it reaches a saturation at bound 20. Thus if we keep the cluster bound large enough, we gain speed-up while not sacrificing from accuracy since the number of distance calculations decreases.

4.4.4. Limitations

We note that the clustering algorithm that we applied in this work relies on choosing a proper threshold value. One can adapt more advanced clustering algorithms like DBSCAN [44] or OPTICS [45] from machine learning domain to create more robust virtual services. Our tool is not be suitable for services requiring very short response generation time such as less than a second. Because, experiments showed that our response generation time exceeds one second for given datasets. Also, currently we do not handle messages that are encoded but a potential solution is to use a decoder in case it is available.

5. STATEFUL SERVICE VIRTUALIZATION

In this section we introduce two different approaches to learn the function VS defined earlier for virtualization of stateful services. In the first technique named Classification Based Virtualization (CBV), we turn the response generation problem into a classification problem. In the second technique named Sequence-to-Sequence Based Virtualization (SSBV), we employ sequence-to-sequence models which is a deep learning algorithm used in transformation of sequences from one form to another form.

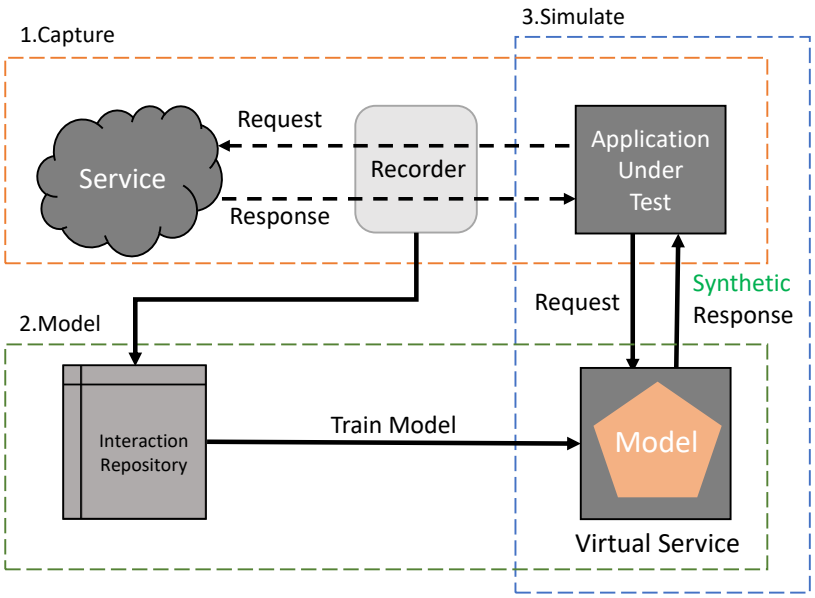

Figure 5.1. Overview of stateful service virtualization.

We now present an example stateful service, then describe the details of CBV as well as SSBV. Last, we evaluate the techniques introduced.

5.1. A Sample Stateful Service

We provide a sample stateful service where a client can create events, update events, and get the latest information on events.

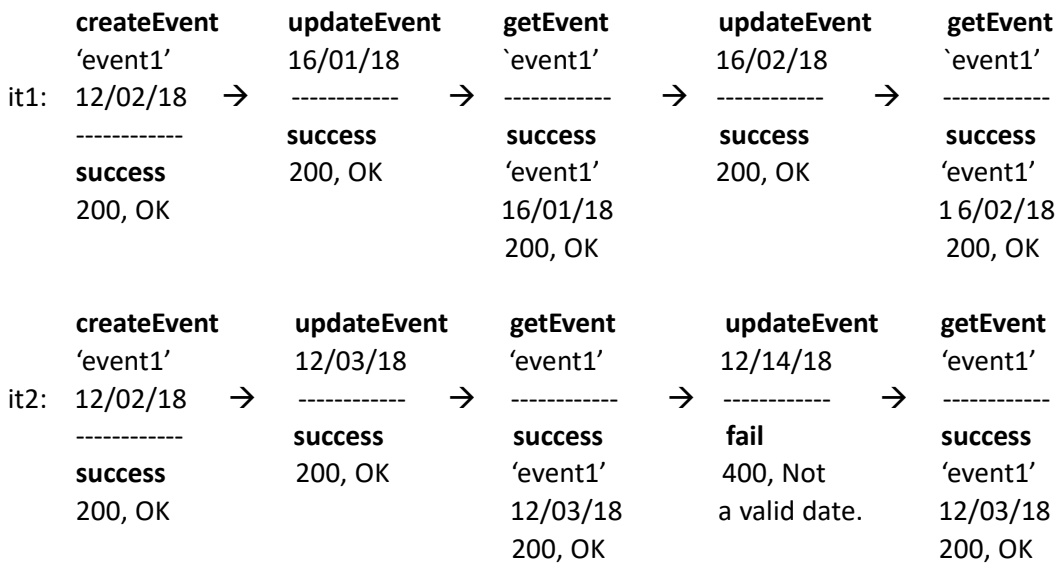

Figure 5.2. Two sample interaction traces are shown in figure. Request types (bold) and request contents are shown above dashed line. Response types (bold) and response contents are shown below dashed line.

Consider sample interaction traces shown in Figure 5.2. For these sample traces, requests are shown above the dashed lines with request types denoted in bold characters (*createEvent, updateEvent* and *getEvent*) and the contents following the types. Similarly, responses are shown below the dashed lines with request types denoted in bold characters (*success, fail*) and the response contents following the types.

These traces show that, for a stateful service, the history of a request has to be considered instead of only the current request to predict the correct response, since the current request's response can be affected by one of the previous interactions.

5.2. Classification Based Virtualization (CBV)

Classification is a supervised learning method in pattern recognition where the task is to learn the mapping from the input to the output [41]. An example classification problem can be assigning of customers to two classes: low-risk and high-risk. The information about a customer such as the income and savings form the input to the classifier whose task is to map the input to one of the two classes.

In a stateful service, a request's response is affected by previous interactions in the history. Therefore, in our case we have to train a model that learns the mapping between h_{req_i} and the response res_i for each req_i. Specifically we use the k history of requests rather than the full history. Both h_{req_i} and res_i show characteristics of categorical data thus we employ one-hot encoding in our approach. Figure 5.3 shows the inputs and the outputs of an example trace. Note that although we use one-hot encoding for our input data, however the prediction of the classifier, which is the response of the request can be enumerated. In classification, outputs are only class labels, therefore ordinality is not the case. Hence, during training we keep a set of all contents to be predicted seen so far. This set contains, *event1, 12/02/18, 16/01/18, 16/02/18, 12/03/18, 12/14/18, 200 OK* and *400, Not a valid date* for the example traces. Outputs are the enumerated values of the elements in this set.

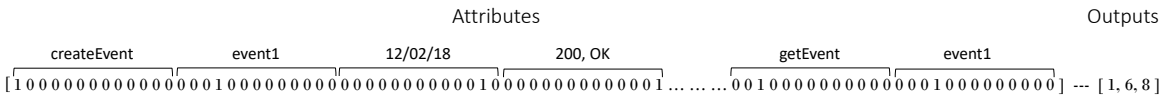

Figure 5.3. The figure shows an example datapoint that will provided for training of a classifier.

If there are more than one class is to be predicted, and those classes can possibly be assigned to more than two types of labels, this is called *multioutput-multiclass classification* [46]. For example, for the first data point in Figure 5.3 we want to predict three fields: the event's label, the event's date and result type. Each field has various number of possible values to take.

We employ Repeated Incremental Pruning to Produce Error Reduction (RIPPER) which is a pure rule based classification algorithm proposed in [47]. RIPPER produces a set of IF-THEN rules for separation. Note that, response prediction can be handled by defining appropriate decisions. An example decision would be, *if the response of updateEvent request is not 200, OK then ignore it*, assuming a failed request does not imply changes in responses in the future. Therefore, a rule based classification technique perfectly fit in response prediction in this case. Other advantages of using RIPPER include interpretability of the resulting rules and shorter training times.

After training the classifier it is ready to predict responses for a given request. When a request arrives, we encode its history and give the encoding as the input to the classifier. The classifier predicts the corresponding response. If the incoming request contains a feature that is not seen in training data, it is encoded in a way that is different from all other features in the training data. Also note that, this technique requires parsing the interactions to find request types, parameters and the response to be encoded.

5.3. Sequence-to-Sequence Based Virtualization (SSBV)

In this section, we describe another approach for virtualization of stateful services. We will train a *sequence-to-sequence* model to learn the function VS for this approach. This approach can be used for services using custom message formats for performance and security purposes. Whereas, the classification based approach assumes a well-defined format such as JSON or XML.

Sequence-to-sequence models were previously employed in problems requiring to consider whole history of input such as language translation [43] or automatic chat-bot creation [48, 49] and demonstrated successful results. *Sequence-to-sequence* models use a special form of *Recurrent neural networks* (RNNs), namely, *Long Short Term Memory* (LSTM) [42]. LSTMs allow the usage of historical data in several steps in the future. This is crucial because stateful services use historical data. The basic architecture of a sequence-to-sequence model is depicted in Figure 5.4. The architecture consists of two LSTM networks [50]; an *Encoder* (with embedding phase) that processes the input sequence and a *Decoder* that produces the output sequence, which are both LSTM networks. We use [51] for sequence-to-sequence learning.

We start with creating a *vocabulary* from the corpus, IR in our case. Previous works [43, 48] add each unique word in the corpus to the vocabulary since their aim is to find the correspondences between the words and the sentences.

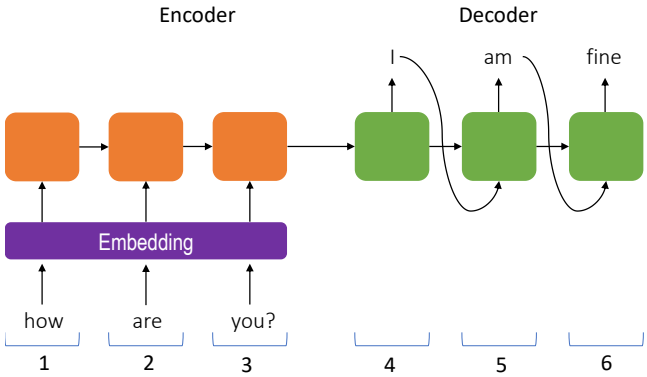

Figure 5.4. The general outline of sequence-to-sequence models consisting of an encoder and a decoder. The sequence *how are you?* is transformed to sequence *I am fine* in the figure.

Table 5.1. The inputs and the outputs corresponding to interactions of an example trace used in training. Spaces are put for clarity here, for actual training we do not put spaces if it is not included in data itself.

Input	**Output**
createEvent event1 12/02/18	200, OK
createEvent event1 12/02/18 200, OK updateEvent 12/03/18	200, OK
createEvent event1 12/02/18 200, OK updateEvent 12/03/18 200, OK getEvent event1	event1 12/03/18 200, OK

In our case we assume that we can not parse the interactions (they come from a custom format). Therefore, we create a vocabulary with the letters and characters in IR. Then the elements in the vocabulary are enumerated. In the embedding phase the input sequence is transformed to a list with enumeration IDs of the letters in the input. The transformed input is given to an encoder network. The encoder learns to encode an input sequence into a vector and the decoder learns to decode this vector back to the output sequence.

In training our sequence-to-sequence model, we use prefixes of interaction traces, hence the model is learned iteratively. For example, Table 5.1 represents the inputs and the outputs for an example interaction trace. Our experience showed that this way of training makes the model learn the mappings faster.

5.4. Evaluation

In this section we demonstrate the experiments to evaluate our approaches. We used our approaches to virtualize three real services. Also, we compared the accuracy, micro-average and macro-average F1-scores of our approaches with the technique introduced in [23]. First, we describe the services used in evaluation. Then we explain the setup and goals of our evaluation and present the evaluation results.

5.4.1. Subject Services

We used two services in evaluation, namely, a proprietary Airline Ticketing Service (ATS) and an open source Google Calendar API (Calendar). Calendar uses the JSON format, whereas ATS uses a custom format for messaging.

Interactions collected from ATS corresponds to previous ticketing operations for testing purposes. For Calendar, we implemented a script in Python that sends random requests to the API and saves the responses. We collected 400 traces each with 10 interactions for these particular services. Our experiences showed that 400 traces are adequate for training of both CBV and SSBV methods. We also note that ATS data includes 26 different request types, whereas Calendar API data includes 5 request types.

5.4.2. Evaluation Setup

5.4.2.1. Metrics. We evaluated the techniques proposed in this work in terms of performance and correctness. Performance refers to the training time of the models. For correctness of CBV, we calculate two correctness scores, namely, exact matching ratio (accuracy) [52] and subset matching ratio. For correctness of SSBV, we use accuracy. The difference is because CBV predicts response classes, whereas SSBV predicts the responses themselves.

We now define the metrics used in correctness. Let E be the expected outputs, P be the predicted outputs, n the number of tests in validation phase, and p the number outputs predicted for each test, (which is three in the example datapoints shown in Figure 5.3 (e.g. 1, 5 and 8)). The set $C = \{c_1, c_2, \ldots, c_n\}$ is the set of all classes to be predicted. There are eight different classes to be predicted in the example shown in Figure 5.3.

$\mathbb{1}$ denotes a slightly modified version of indicator function. $\mathbb{1}$ of a set A and set X is a function

$$\mathbb{1}_A : X \to \{0, 1\}$$

defined as

$$\mathbb{1}_A(x) := \begin{cases} 1, & \text{if } x = A \\ 0, & \text{if } x \neq A \end{cases}$$

$$ExactMatchRatio = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{E_i}(P_i)$$

Clearly, a disadvantage of this measure is that it doesn't distinguish between complete incorrect and partially correct predictions which can be considered as harsh for some cases. Therefore we define subset matching ratio:

$$SubsetMatchRatio = \frac{1}{np} \sum_{i=1}^{n} \sum_{j=1}^{p} \mathbb{1}_{E_{ij}}(P_{ij})$$

The higher SubsetMatchRatio values are the better. We expect that we will have more optimistic results with subset matching ratio measurement.

We also calculate macro and micro averaged f-scores for evaluation of CBV method. In macro average, metrics are calculated for each label, and their unweighted mean is found. In micro average, metrics are calculated globally. F-score is originally defined for binary classification, however, an extension is proposed for multi-label classification. We modify macro and micro averaged f-score calculations proposed for multi-label classification [52] as follows:

$$F^{c}_{macro} = \frac{2\sum_{i=1}^{n}\sum_{j=1}^{p} Y^{c}_{ij} Z^{c}_{ij}}{\sum_{i=1}^{n} Z^{c}_{i} + \sum_{i=1}^{n} Y^{c}_{i}} \quad F_{macro} = \frac{1}{|C|}\sum_{k=1}^{|C|} F^{c_k}_{macro}$$

$$F_{micro} = \frac{2\sum_{k=1}^{|C|}\sum_{j=1}^{p}\sum_{i=1}^{n} Y^{c_k}_{ij} Z^{c_k}_{ij}}{\sum_{k=1}^{|C|}\sum_{j=1}^{p}\sum_{i=1}^{n} Z^{c_k}_{ij} + \sum_{k=1}^{|C|}\sum_{j=1}^{p}\sum_{i=1}^{n} Y^{c_k}_{ij}}$$

where

$$Y^{c_k}_{ij} = \begin{cases} 1, & \text{if } c_k \text{ is actually at } Y_{ij} \\ 0, & \text{otherwise} \end{cases}$$

$$Z^{c_k}_{ij} = \begin{cases} 1, & \text{if } c_k \text{ is correctly predicted at } Z_{ij} \\ 0, & \text{otherwise} \end{cases}$$

In SSBV we use *accuracy* for evaluation. Accuracy in SSBV is defined as follows. Let the real service to be virtualized be RS and the virtual service be VS. If, for a given request history, the outputs of the real service to be virtualized and its virtual service are the same, then the response predicted by the virtual service is an accurate response.

5.4.2.2. Research Questions. We describe the research problems that we evaluated with the results of our experiments. We look for the answers of the research questions given below:

RQ1: How successful are the techniques introduced in this work in replacing real services in terms correctness?

RQ2: How effective are the techniques introduced in this work in terms of performance?

5.4.2.3. Experimental Design. Parameters selected in experiments are depicted in Table 5.2. Our primary concern in choosing those parameters is maximizing the correctness of the models.

For each of technique we created models with history sizes of $k \in 1, 5, 10$. For $k = 1$, we use the last interactions. For $k = 5$ ($k = 10$) we use the last 5 (10) interactions recorded including the incoming request.

For CBV method, we use Weka [53] implementation of RIPPER. Weka is a library of machine learning algorithms for data mining tasks. For SSBV method, we created models using Tensorflow [51] implementation of sequence-to-sequence model with history sizes $k \in 1, 5, 10$. There are other implementations of sequence-to-sequence models are available in several deep learning frameworks.

Table 5.2. Parameters selected in experiments.

Method	**Parameters**
CBV (RIPPER)	*minNo = 1*
SSBV (Tensorflow)	*hidden_size = 25* *batch_size = 128* *layers = 2* *epochs = 1* *iteration = 1000*
EFSM Tool (J48)	*default*

A more comprehensive comparison between deep learning frameworks can be found in [54]. We employed Keras [55] for running Tensorflow framework which is a high-level neural-network API running on top of Tensorflow. EFSM tool presented in [23] can be found in [56]. We used J48 option [53] for classification. This option generates C4.5 [57] decision trees.

In the experiments, for each service, we performed three steps: (1) collect interactions from real services, (2) learn models using the collected interactions, and (3) measure the correctness and performance. We employed 5-fold cross validation for CBV and EFSM Tool where we use 80% of the data for training and 20% of the data for validation for each fold. Experiments were run on a server with 16 GB memory and Intel Xeon E5-2630L v2 2.40GHz CPU.

5.4.3. Evaluation Results

We now present and discuss the results of the experiments. First, we evaluate correctness and then we evaluate performance of the methods mentioned in this paper. We present the results of the experiments in Table 5.3 and Table 5.4. The results demonstrated in these tables are achieved with parameters shown in Table 5.2.

5.4.3.1. RQ1: Correctness Results. We achieve the highest accuracy, exact matching ratio and subset matching ratio with a history size of 10 for both ATS and Calendar API. We observed that matching ratios obtained from CBV method are slightly higher than those for EFSM Tool, since both of them are classification based techniques. However EFSM Tool does not consider the history while predicting a response. Therefore we obtain higher values with CBV for most of the experiments. Also F_{micro} and F_{macro} scores support results achieved in exact and subset match ratios. The highest F_{micro} and F_{macro} are close to 0.9 which is satisfying for.

We also employed statistical tests to learn if the difference in correctness between CBV and EFSM Tool are significant or not. First, we applied Shapiro-Wilk test to learn if our samples are distributed normally. Our samples consist of EMR and SMR results and f-score ratios of each fold in our 5-fold cross validation process. We consider only k=10 history of correctness results which are the highest among all k values. According to the results of Shapiro-Wilk test our samples are distributed normally for EMR, SMR and f-score metrics for CBV and EFSM Tool. Since normality assumption is achieved, we performed one sided two sample t-test to check whether the difference in results between CBV and EFSM Tool are statistically significant or not. Results showed that true difference in correctness is not equal to 0 with 95% confidence interval for EMR, SMR and f-scores. This means that EMR, SMR and f-score results are higher for CBV compared to EFSM Tool with 95% confidence. Note that these statistical tests are performed for both of ATS and Calendar API data separately.

When we look at the traces in detail, we observed that there can be responses which are not possible to predict using automated methods like CBV or SSBV. An example of such a response can be a response including today's date which is not possible to predict using CBV or SSBV. However, these kind of interactions can be handled with a small manual effort in practice. Also we observe that, we achieve higher match ratios for ATS when compared to Calendar API. This is because ATS has limited type of request and response contents, however in Calendar there are extensive types of request and response contents.

Table 5.3. Correctness results of CBV, SSBV and EFSM Tool. EMR stands for Exact Matching Ratio, SMR stands for Subset Matching Ratio.

Service	**k**	**CBV**				**EFSM Tool**				**SSBV**
		EMR(%)	SMR(%)	F_{macro}	F_{micro}	EMR(%)	SMR(%)	F_{macro}	F_{micro}	Accuracy (%)
ATS	1	76.6	79.6	0.781	0.767	78.1	80.6	0.803	0.783	92.1
Calendar	1	70.3	78.5	0.757	0.741	70.7	76.0	0.721	0.715	93.2
ATS	5	82.7	84.3	0.813	0.798	80.0	83.7	0.806	0.786	96.5
Calendar	5	81.1	84.1	0.843	0.825	71.5	77.1	0.753	0.741	97.3
ATS	10	82.7	84.3	0.813	0.798	80.0	83.7	0.806	0.786	96.5
Calendar	10	82.0	88.5	0.871	0.866	72.1	78.0	0.767	0.751	99.3

In summary, we find that virtual services created using SSBV technique are accurate enough to replace the real services when 90% or more accuracy is needed. Virtual services created using CBV technique can replace the real services when an exact match is not required and a high subset match is enough.

5.4.3.2. RQ2: Performance Results. In CBV method, virtual services are created much faster than the SSBV technique. The training time of RIPPER in CBV method is in minutes, however, the training time of SSBV takes hours. Note that we conducted our experiments on a CPU and deep learning algorithms can run faster with GPUs.

The training time of the EFSM tool is shorter than SSBV method, however, longer than CBV method. If time is not important SSBV method can be used to virtualize a service since SSBV is the most successful method for generating correct responses. If time is limited it would be logical to use CBV method instead of EFSM Tool since again correctness in CBV is higher than EFSM Tool and also CBV runs faster. One advantage of EFSM Tool over CBV and SSBV is the model inferred. EFSM tool is originally designed for reverse engineering [23] and can extract behavioral models of the systems as EFSMs.

Table 5.4. Performance results of CBV, SSBV and EFSM Tool. Training time in format (hh:mm).

Service	**k**	**SSBV**	**CBV**	**EFSM Tool**
ATS	1	01:42	00:01	00:06
Calendar	1	02:21	00:01	00:06
ATS	5	08:51	00:03	00:11
Calendar	5	09.54	00:03	00:14
ATS	10	14:42	00:04	00:18
Calendar	10	16:19	00:04	00:19

5.4.4. Threats To Validity

In this section, we discuss threats to our evaluation's validity.

5.4.4.1. Internal validity. The selected services in this work do not imply any bias. They are taken from different business areas and also their message format are different from each other.

Interactions from Calendar API are, in part, manually collected such that the parameters to requests are manually specified. This may bias the evaluation results. To mitigate this threat, we created a set for each parameter field and randomly chose one element from each set for each request. This makes a huge number of combinations of parameters for each request and prevent any bias for the results. Interactions from ATS are collected during testing of a dependent component, therefore we do not expect ATS data bias the results.

Some of the requests in ATS have only one type of response. Thus all the techniques mentioned in this work can predict correct responses for such requests which makes overall correctness higher. However, ATS is in use in real life. This demonstrates us that there can be such cases in the industry. Therefore we do not expect that this threats our validity.

The algorithms used in this work can produce different results according to the parameters given. We use the best achieving parameters for all tools to prevent any bias in evaluation.

5.4.4.2. External Validity. Subject services in this work are selected from real life services. Additionally, the custom message format of ATS and the standard JSON message format of Calendar API make a good combination in terms of evaluation of techniques. Also, the techniques proposed in this work can easily be implemented. Thus, we say that our results are generalizable. However, these approaches would not work if interactions are encrypted.

5.4.4.3. Construct Validity. The metrics specified in this work are performance and correctness which are the main concerns in service virtualization. We measure the correctness of SSBV with accuracy. On the other hand, we define exact matching and subset matching ratios for CBV to evaluate its success in detail. Also, we measure micro and macro averaged f-scores, since sometimes raw results of classification do not tell the whole story.

5.4.5. Limitations

Results show that SSBV takes hours of training time when the dataset is large. This would be limitation when an urgent use is needed. On the other hand, CBV is a pretty fast technique. When performance is the concern CBV can be employed. However, we note that CBV requires that the messages to be parsed. This is a drawback when the services use a custom message format in requests and responses.

6. CONCLUSIONS

In multi-layered and service oriented architecture based software systems testing or development can be painful because of interdependent nature of such systems. This thesis studied how to overcome the dependency issue by creating *virtual services*. Recent studies [8–10] proposed techniques to create test doubles namely virtual services. However, there is still room for improvement in this field. Current works suffer from limited accuracy and performance. Also they can not simulate the state behavior of real services. In this work, we presented bioinformatics and machine learning based novel methods to create virtual services namely FancyMock, CBV and SSBV. FancyMock is specialized for virtualization of stateless services. Stateless services do not require to keep record of interaction histories. FancyMock makes use of pairwise and multiple sequence alignment algorithms to produce a valid response. CBV and SSBV are specialized for virtualization of services keeping state. CBV transforms response prediction problem into a classification problem and in SSBV we employ sequence-to-sequence models. CBV and SSBV are designed to reflect the state behavior of the service to be virtualized.

Our evaluations demonstrate that the techniques introduced in this thesis are successful in terms of the defined metrics. In stateless service virtualization FancyMock outperforms the baseline technique in terms of validity and training time. Also FancyMock generates more diverse responses. However, it takes longer time to generate response in FancyMock. In stateful service virtualization, virtual services trained by CBV technique significantly outperform the other methods in terms of training time. On the other hand, virtual services trained by SSBV model produce the most accurate responses. Our results highlight the importance of the trace size in training where longer traces result in more accurate responses. In addition, our research shows that methods proposed in this work are better in predicting responses than the technique presented in [23].

In future, we will work on generating responses faster in FancyMock. Also, for stateful service virtualization, we plan to reduce training time of models while preserving correctness as high as SSBV's correctness. This can be achieved using faster neural networks or GPUs. Last, we plan to infer the state model of a service using recorded requests and responses. State models can also be used in debugging and documentation.

REFERENCES

1. InfoQ, "Stub, Mock and Service Virtualization", `https://goo.gl/vNv29f`, accessed at September 2017.

2. SmartBear, "SmartBear Service Virtualization", `https://goo.gl/1Cu5me`, accessed at May 2017.

3. Hurwitz, K., *Service Virtualization for dummies*, John Wiley & Sons, 2013.

4. Michelsen, J. and J. English, *What is service virtualization?*, Springer, 2012.

5. Nizamic, F., R. Groenboom and A. Lazovik, "Testing for highly distributed service-oriented systems using virtual environments", *Proceedings of 17th Dutch Testing Day*, 2011.

6. Schneider, J.-G., P. Mandile and S. Versteeg, "Generalized Suffix Tree based Multiple Sequence Alignment for Service Virtualization", *Software Engineering Conference (ASWEC), 2015 24th Australasian*, pp. 48–57, IEEE, 2015.

7. Du, M., J.-G. Schneider, C. Hine, J. Grundy and S. Versteeg, "Generating service models by trace subsequence substitution", *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, pp. 123–132, ACM, 2013.

8. Du, M., S. Versteeg, J.-G. Schneider, J. Han and J. Grundy, "Interaction traces mining for efficient system responses generation", *ACM SIGSOFT Software Engineering Notes*, Vol. 40, No. 1, pp. 1–8, 2015.

9. Versteeg, S., M. Du, J. Bird, J.-G. Schneider, J. Grundy and J. Han, "Enhanced playback of automated service emulation models using entropy analysis", *Continuous Software Evolution and Delivery (CSED), IEEE/ACM International Workshop*

on, pp. 49–55, IEEE, 2016.

10. Versteeg, S., M. Du, J.-G. Schneider, J. Grundy, J. Han and M. Goyal, "Opaque service virtualisation: a practical tool for emulating endpoint systems", *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 202–211, ACM, 2016.

11. CA, "ODP Manual Page", `https://goo.gl/Yu5zir`, accessed at August 2017.

12. Boettiger, C., "An introduction to Docker for reproducible research", *ACM SIGOPS Operating Systems Review*, Vol. 49, No. 1, pp. 71–79, 2015.

13. Giudice, D. L., "Service Virtualization And Testing Solutions", *Forrester Wave*, 2014.

14. Murphy, T. E. and N. Wilson, "Magic Quadrant for Integrated Software Quality Suites", *Gartner Research*, 2013.

15. IT-Central-Station, "Service Virtualization A Peek Into What Real Users Think", `https://goo.gl/oN23qH`, accessed at December 2017.

16. Biermann, A. W. and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior", *IEEE transactions on Computers*, Vol. 100, No. 6, pp. 592–597, 1972.

17. Cook, J. E. and A. L. Wolf, "Discovering models of software processes from event-based data", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 7, No. 3, pp. 215–249, 1998.

18. Lo, D. and S.-C. Khoo, "SMArTIC: towards building an accurate, robust and scalable specification miner", *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 265–275, ACM, 2006.

19. Lo, D., L. Mariani and M. Pezzè, "Automatic steering of behavioral model infer-

ence", *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 345–354, ACM, 2009.

20. Walkinshaw, N. and K. Bogdanov, "Inferring finite-state models with temporal constraints", *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 248–257, IEEE Computer Society, 2008.

21. Reiss, S. P. and M. Renieris, "Encoding program executions", *proceedings of the 23rd International Conference on Software Engineering*, pp. 221–230, IEEE Computer Society, 2001.

22. Lorenzoli, D., L. Mariani and M. Pezzè, "Automatic generation of software behavioral models", *Proceedings of the 30th international conference on Software engineering*, pp. 501–510, ACM, 2008.

23. Walkinshaw, N., R. Taylor and J. Derrick, "Inferring extended finite state machine models from software executions", *Empirical Software Engineering*, Vol. 21, No. 3, pp. 811–853, 2016.

24. Dallmeier, V., C. Lindig, A. Wasylkowski and A. Zeller, "Mining object behavior with ADABU", *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pp. 17–24, ACM, 2006.

25. de Caso, G., V. Braberman, D. Garbervetsky and S. Uchitel, "Automated abstractions for contract validation", *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 141–162, 2012.

26. Beschastnikh, I., Y. Brun, S. Schneider, M. Sloan and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models", *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 267–277, ACM, 2011.

27. Beschastnikh, I., Y. Brun, M. D. Ernst and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight", *Proceedings of the 36th International Conference on Software Engineering*, pp. 468–479, ACM, 2014.

28. Ohmann, T., M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh and Y. Brun, "Behavioral resource-aware model inference", *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 19–30, ACM, 2014.

29. Krka, I., Y. Brun and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants", *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 178–189, ACM, 2014.

30. Krka, I., Y. Brun, D. Popescu, J. Garcia and N. Medvidovic, "Using dynamic execution traces and program invariants to enhance behavioral model inference", *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pp. 179–182, ACM, 2010.

31. Giles, C. L., C. B. Miller, D. Chen, H.-H. Chen, G.-Z. Sun and Y.-C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks", *Learning*, Vol. 4, No. 3, 2008.

32. Casey, M., "The dynamics of discrete-time computation, with application to recurrent neural networks and finite state machine extraction", *Dynamics*, Vol. 8, No. 6, 2008.

33. Cleeremans, A., D. Servan-Schreiber and J. L. McClelland, "Finite state automata and simple recurrent networks", *Neural computation*, Vol. 1, No. 3, pp. 372–381, 1989.

34. Clarke, E., O. Grumberg, S. Jha, Y. Lu and H. Veith, "Counterexample-guided abstraction refinement", *Computer aided verification*, pp. 154–169, Springer, 2000.

35. Needleman, S. B. and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", *Journal of molecular biology*, Vol. 48, No. 3, pp. 443–453, 1970.

36. JSON, "JSON Message Format", `https://www.json.org/`, accessed at December 2017.

37. Thompson, J. D., D. G. Higgins and T. J. Gibson, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice", *Nucleic acids research*, Vol. 22, No. 22, pp. 4673–4680, 1994.

38. Wang, L. and T. Jiang, "On the complexity of multiple sequence alignment", *Journal of computational biology*, Vol. 1, No. 4, pp. 337–348, 1994.

39. Saitou, N. and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees.", *Molecular biology and evolution*, Vol. 4, No. 4, pp. 406–425, 1987.

40. Roussopoulos, N., S. Kelley and F. Vincent, "Nearest neighbor queries", *ACM Sigmod record*, Vol. 24, pp. 71–79, ACM, 1995.

41. Alpaydin, E., *Introduction to machine learning*, MIT press, 2014.

42. Hochreiter, S. and J. Schmidhuber, "Long short-term memory", *Neural computation*, Vol. 9, No. 8, pp. 1735–1780, 1997.

43. Sutskever, I., O. Vinyals and Q. V. Le, "Sequence to sequence learning with neural networks", *Advances in neural information processing systems*, pp. 3104–3112, 2014.

44. Ester, M., H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.", *Proceedings of the Sec-*

ond International Conference on Knowledge Discovery and Data Mining, Vol. 96, pp. 226–231, 1996.

45. Ankerst, M., M. M. Breunig, H.-P. Kriegel and J. Sander, "OPTICS: ordering points to identify the clustering structure", *ACM Sigmod record*, Vol. 28, pp. 49–60, ACM, 1999.

46. Buitinck, L., G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project", *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

47. Cohen, W. W., "Fast Effective Rule Induction", *Twelfth International Conference on Machine Learning*, pp. 115–123, Morgan Kaufmann, 1995.

48. Xu, A., Z. Liu, Y. Guo, V. Sinha and R. Akkiraju, "A New Chatbot for Customer Service on Social Media", *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 3506–3510, ACM, 2017.

49. Zhou, X., B. Hu, Q. Chen, B. Tang and X. Wang, "Answer sequence learning with neural networks for answer selection in community question answering", *arXiv preprint arXiv:1506.06490*, 2015.

50. Cho, K., B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation", *arXiv preprint arXiv:1406.1078*, 2014.

51. Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems", *arXiv preprint arXiv:1603.04467*, 2016.

52. Sorower, M. S., "A literature survey on algorithms for multi-label learning", .

53. Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I. H. Witten, "The WEKA data mining software: an update", *SIGKDD Explorations*, Vol. 11, No. 1, pp. 10–18, 2009.

54. Bahrampour, S., N. Ramakrishnan, L. Schott and M. Shah, "Comparative study of deep learning software frameworks", *arXiv preprint arXiv:1511.06435*, 2015.

55. Chollet, F. *et al.*, "Keras", `https://github.com/fchollet/keras`, accessed at November 2017.

56. Walkinshaw, N., "EFSM Inference Tool", `https://bitbucket.org/nwalkinshaw/efsminferencetool/`, accessed at December2017.

57. Quinlan, R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.