CODI
A Conversation Disentanglement Microservice

Edoardo Riggio

June 2022

Supervised by
Prof. Dr. Michele Lanza

Co-Supervised by
Marco Raglianti

Bachelor Project
Abstract

Instant Messaging Applications (IMAs) – such as Discord, Gitter, and Slack – are becoming popular means of communication. However, IMAs lack features to disambiguate and identify conversations. A conversation is a sequence of time-stamped messages representing an interactive discussion between multiple people. A message is said to be part of a conversation if it answers a previously posed question or discusses a similar topic as other messages. We implemented an accessible and user-friendly REST microservice that can automate the disambiguation of a set of messages to form conversations by leveraging machine learning algorithms.
# Contents

Abstract

1 Introduction
   1.1 Contributions ................................................. 2
   1.2 Document Structure ........................................ 3

2 State of the Art
   2.1 Instant Messaging ........................................... 4
      2.1.1 Brief History ........................................... 4
   2.2 Conversation Disentanglement ......................... 6
   2.3 Summary and Outlook .................................. 7

3 CODI Implementation
   3.1 Architecture .............................................. 8
   3.2 Backend .................................................. 9
      3.2.1 REST API ............................................. 11
      3.2.2 Disentangler ......................................... 11
      Feature Extraction ........................................ 13
      Max-Entropy Classifier .................................. 14
      Correlation Clustering .................................. 14
   3.3 Frontend ................................................ 15
      3.3.1 Homepage ............................................ 15
      3.3.2 Validation Statistics ............................... 16
      3.3.3 Prediction Statistics ............................... 17
      3.3.4 Dataset Annotation ................................ 18

4 Result Analysis
   4.1 Evaluation Methods ....................................... 19
      4.1.1 Max Entropy ......................................... 19
      4.1.2 Correlation Clustering .............................. 19
   4.2 Disentanglement Performance ............................ 20
      4.2.1 Original Datasets .................................. 20
      4.2.2 Custom Dataset ..................................... 20
   4.3 Disentanglement Quality ................................ 21
   4.4 Time Performance ......................................... 22
   4.5 Summary and Outlook ................................ 23

5 Conclusion and Future Work
   5.1 Future Work ............................................... 24
   5.2 Concluding Remarks ....................................... 24
Chapter 1

Introduction

Instant Messaging Applications have evolved into one of the most revolutionary means of communication, gradually supplanting conventional asynchronous media (e.g., email). People thousands of kilometers apart can communicate in real-time thanks to these services. Thus, some instant messaging apps, such as Slack and Discord, might have extremely high throughput. This means that hundreds of messages could be transmitted every second by people all around the world. High throughput in large chats may cause numerous conversations to overlap and be separated by interleaving messages.

Researchers have found algorithms that could mimic what human beings do to separate conversations [1–6]. Of these algorithms, we decided to implement the one proposed by Elsner and Charniak [1,3] and later extended by Chatterjee et al. [2]. This is a two-step algorithm in which pairs of messages and their features are first fed to a max-entropy classifier – which determines the relatedness of messages. The output of this classifier is given as input, together with the list of all the messages, to the second step, the correlation clustering algorithm which will group messages based on the previously found relations into conversations.

With Codi, we attempted to achieve ease of use. We aimed to make it easy for researchers to annotate and perform conversation disentanglement on given datasets. Codi is a simple web application with a client module that allows users to drag and drop datasets to perform training, validation, or prediction operations, as well as graphically pick which feature-sets to employ in the disentanglement process. Users can also directly access the REST API endpoints, which can be invoked using scripts, for example. Moreover, we have created a view for dataset annotation. This was done so that even researchers with limited technical knowledge could readily manually label the dataset that would be given to the disentangler.

1.1 Contributions

There are several contributions to this project. Below we will identify all of them.

- **Replication Contribution**
  This consisted in reproducing the algorithm developed by Elsner and Charniak [1,3]. This two-step disentangling approach, which includes both the max-entropy and correlation clustering algorithms, is used in Codi’s backend.

- **Engineering Contribution**
  This is the development of Codi itself. Codi can accept input files containing raw or structured dumps of Discord and Slack chats, with or without conversation annotation. The input dataset is pre-processed before being sent to the two-step disentanglement algorithm. Once the algorithm has generated the conversations, they are displayed to the user along with statistics on the disentanglement’s effectiveness, such as accuracy, precision, recall, and F1 score for the max-entropy algorithm; and micro-averaged F1 score for the correlation clustering algorithm.
• **Dataset Contribution**
  In addition to the datasets made public by Elsner and Charniak [1, 3] and by Chatterjee et. al [2], we have annotated our own dataset. This dataset comprises 294 messages taken from a Pharo Discord server. Two people manually annotated the messages, which were then merged to form the final annotated dataset. We have used this dataset to evaluate our implementation of the disentanglement algorithm on a new ground truth.

• **Validation Contribution**
  In the final phase of the implementation of CODI, we used some annotated datasets to verify our implementation’s correctness and understand its limitations. In addition to two datasets offered by Elsner and Charniak and Chatterjee et al., we also use our custom dataset in the evaluation. The two original datasets were mainly used to verify the correctness of the implementation. The results obtained were compared with the ones obtained by the original authors. The third dataset, the custom one, was used to understand the limitations of such an algorithm.

1.2 **Document Structure**

This document has been divided into several different chapters. Each chapter describes one or more of the steps that allowed us to complete this project. More specifically, we have four chapters.

• **State of the Art**
  In this chapter, we talk about the research done in this field prior to our project. We have also inserted a section dedicated to the history of instant messaging and its applications.

• **CODI Implementation**
  This chapter explains in detail how CODI works. It talks about how we structured CODI and what a user can do to disentangle their datasets.

• **Result Analysis**
  In this chapter, we analyze the experiments we performed on CODI with several different datasets and report the results of such experiments.

• **Conclusion and Future Work**
  In this last chapter, we have a brief concluding summary of the project and suggestions on how to further improve CODI.
Chapter 2

State of the Art

2.1 Instant Messaging

Instant messaging is a set of protocols for communication between two or more people over the Internet. This technology differs from others – such as email – because conversations happen in real-time rather than asynchronously, hence the word *instant*.

2.1.1 Brief History

The first instant messaging platform dates back to 1973. Its name was *Talkomatic* (Figure 2.1a), and it allowed five people to have each a section of the screen – each user could write only five lines of text and chat in real-time. Differently from modern-day instant messaging applications, *Talkomatic* was a real-time text service. This means the messages were broadcasted letter-by-letter as the users typed them in. This type of technology characterized most of the early real-time messaging services.

In the early 1990s, however, a company named *Quantum Link* – which later became *America On-Line* – developed and released an online service for the Commodore 64 in which concurrently connected users could exchange messages in real-time. This concept was further refined and integrated by *America Online* in one of the first and most important mass-distributed instant messaging platforms, *AOL Instant Messenger* (AIM) ¹ (Figure 2.1c). This platform was also one of the first with a *Graphic User Interface* (GUI).

Today, instant messaging has become one of the most prevalent communication mediums. Now we have platforms such as *WhatsApp*, *Discord* (Figure 2.1d), *Slack*, *Skype*, *Telegram*... These platforms have billions of monthly active users ² ³ and have reshaped how we communicate. *Gitter*, *Slack*, and *Discord* are currently some of the most used instant messaging platforms and have also been studied as possible data mining sources for software-related information [7–13] among software developers.

IM platforms allow users to create communities with several different channels where people can discuss specific subtopics. In many cases, such communities and channels are related to software development. These channels often follow a Q&A approach. This means that developers ask questions related to the subtopic of the channel and receive answers from other fellow developers. Users may participate in such conversations by sending messages. Code blocks, mentions to other users or channels, multimedia, or attachments can sometimes accompany such messages. These additional elements can help the disentanglement process, particularly in software development environments.

¹https://en.wikipedia.org/wiki/Instant_messaging
²https://www.businessofapps.com/data/slack-statistics/
³https://www.businessofapps.com/data/discord-statistics/
2.1. Instant Messaging

FIGURE 2.1: Interfaces of instant message platforms through the years
2.2 Conversation Disentanglement

Conversation disentanglement is the task of clustering messages into a set of conversations. Even for a human reader, it is challenging to reconstruct the conversation flow in real-time. But, for a machine, it represents a highly complex task. IM platforms can reach throughputs of several hundred messages per hour during active conversations. This means that multiple messages can arrive at almost the same time, resulting in interleaving messages about different simultaneous conversations. As we can see in Figure 2.2 (taken from an annotated dataset in the source code of [2]), the conversation between Chauncey and Gale is interrupted by Nestor, who is following up to a possible coding question he had sent previously. The first conversation is resumed as soon as Nestor finishes.

![Example of fragmented conversation](https://www.protocol.com/media-library/discord-chat.png?id=24629797)

**Figure 2.2: Example of fragmented conversation**

Another interesting thing to notice is that on Instant Messaging platforms, people tend to write many short messages one after the other rather than sending one long message. This can also be seen in Figure 2.2, where Nestor, instead of sending one long message, decides to send five messages, each less than ten words long. This can make the task of creating conversations more difficult. For example, in Figure 2.3, the messages sent by Chauncey – which are part of the same conversation – are interrupted by Laura asking a completely unrelated question. A possible algorithm should have a way of referring to previous conversations to decide which conversation to assign that last message from Chauncey. Aoki et al. [14] demonstrated the complexity of conversation disentanglement by analyzing voice conversations between 10 people. Elsner and Charniak [1, 3] investigated IRC logs disentanglement and proposed the core of the algorithm implemented in this project. Further improvements were introduced by Chatterjee et al. [2], adapting the algorithm to Slack conversations of software developer communities.

According to Liu et al. [5] conversation disentanglement algorithms can be divided into two main categories. The first is composed of two-step algorithms, where the relatedness of message pairs is computed, and then messages are clustered based on this metric. The other category is end-to-end algorithms, where

---

[5]https://c9x.me/irc/irc.c.png
global properties of the conversation flow are captured at once [6]. Elsner and Charniak were among the first researchers to propose a two-step solution, which they used to disentangle IRC chats [3]. Later, this algorithm was adapted by Chatterjee et al. to support Slack chat disentanglement [2] and, just recently, Subash et al. applied to the original algorithm to disentangle Discord chats [6]. An example end-to-end approach has been proposed by Jiang et al. [4] who used a Siamese Hierarchical Convolutional Neural Network to identify threads of conversations. Although of great interest and achieving comparable or even better performances, end-to-end approaches usually offer a black-box tool. We decided to start by implementing a two-step algorithm for its explainability and extensibility, particularly with new features.

2.3 Summary and Outlook

We have seen in this chapter how challenging it is for humans and machines to disentangle conversations. Researchers have found several algorithms using supervised and unsupervised machine learning strategies to ease this task. We decided to implement a two-step approach proposed by Elsner and Charniak [1,3]. In the next chapter, we will present the implementation of CODI. We will explain how the service works and how we implemented the original two-step algorithm.
Chapter 3

CODI Implementation

The following chapter will provide an overview of how we developed CODI. We present the backend implementing the core services. In the sections dedicated to the frontend, we introduce a web interface to interact and experiment with the backend. The frontend also provides information to evaluate the results of the disentanglement and the performance of the process.

3.1 Architecture

CODI is a web server composed of both frontend and backend and divided into several modules (Figure 3.1). The frontend is composed of the client module. The backend includes both the disentangling and REST API modules. On the one hand, the REST API module is responsible for connecting the frontend with the disentangling module. On the other hand, the disentangling module provides all the logic and algorithms for pre- and post-processing the messages and clustering them into conversations.

CODI receives as input a JSON file containing the dump of either a Slack or Discord community. The structure of the JSON file can be seen in Figure 3.2. Each file must correspond to one community, which must have an ID, a platform name, a name, a feature set, a list of members, and a list of channels. In the case of the feature set, it is represented as an array. The first element of the array indicates if chat features should be included when extracting features from messages. This is a bit, and it can be set to 1 to include this feature group or 0 otherwise. The other two elements of the array are for the discourse features and the content related features.
The list of members should contain all authors – i.e., members who wrote at least one message on one of the channels – and non-authors. Furthermore, CODI will partially initialize a member not in the list only if another member mentions it in one of the community messages. This new member will either only have an ID – since a Discord member mention is of the form <@123456> – or an ID and a name – since a Slack member mention is of the form <!@123456|username>.

The list of channels must contain channel objects. Each channel must have an ID, a name, a path, and a list of messages. Optionally, they can also contain a list of topics – which needs to have a list of keywords, and a description. These messages must each have an ID, an author ID (which represents the author’s ID of the message), a content, a conversation ID (which is explained in the following paragraph), and a timestamp (which has to be in a standard time format). Optionally, messages can also have a list of attachments – which have an url field of the multimedia content attached to the message.

The dataset format to be sent to CODI is slightly different based on the operation that the user needs to carry out. These operations are the following:

- **Training**
  This operation is used to train the disentangler’s classification model.

- **Validation**
  This operation creates the conversation clusters and validates them with respect to a gold dataset.

- **Prediction**
  This operation creates the conversation clusters.

In the case of training or validation, the dataset must contain a "conversation": "T1" field for each message. The format of the "conversation" content must be T + number. This field is used in training and validation operations to provide a ground truth for the disentangled conversations.

Given the input dataset, the client will call the respective REST API endpoint, which will call the methods necessary for the disentanglement of the conversations. As soon as the disentanglement algorithm finishes, the conversations will be visible on the website.

### 3.2 Backend

The backend of this project was developed entirely in Python 3.10, with the aid of Django\(^1\) – a Python web framework. We decided to use Django because it is a de-facto standard framework for Python web development. In particular, we used the package Django REST Framework\(^2\) to create all the endpoints for our application.

The implementation of the backend follows an Object-Oriented approach. We modeled the domain of instant messaging conversations and translated this model into Python classes supporting the corresponding data structures and operations needed (Figure 3.3). The format of the input JSON file described earlier (Figure 3.2) emerges from this modeling and closely matches the essential aspects of the domain.

\(^1\)Django: The web framework for perfectionists with deadlines https://www.djangoproject.com/

\(^2\)Django REST Framework https://www.django-rest-framework.org/
FIGURE 3.2: Input JSON dataset

```json
{
  "platform": "slack",
  "features": [1, 1, 1],
  "id": "b4138f14-af37-4c23-9bda-289bfc36a7fb",
  "name": "training-set",
  "members": [
    {
      "id": "d1ff9c5b-f1fc-47c4-a49d-e2691d103b57",
      "name": "Jermaine"
    },
    ...
  ],
  "channels": [
    {
      "id": "c65d238f-d987-49fe-844b-eef3cfceee4c",
      "name": "discussion",
      "path": "agile/discussion",
      "topics": [
        {
          "keywords": [
            "Agile",
            ...
          ],
          "description": "Agile Visualization to its greatest!"
        },
        ...
      ],
      "messages": [
        {
          "id": "34ce13f1-6577-4710-a007-da4e12d523ef",
          "authorId": "d1ff9c5b-f1fc-47c4-a49d-e2691d103b57",
          "content": "But $(date) is in large format",
          "conversation": "T35",
          "timestamp": "2022-02-06T19:24:23.777+00:00",
          "attachments": [
            {
              "url": "https://cdn.discordapp.com/..."
            },
            ...
          ],
          ...
        },
        ...
      ]
    }
  ]
}
```
3.2. Backend

![Class Diagram](image-url)

**Figure 3.3: Class Diagram**

### 3.2.1 REST API

The REST API is built with *Django REST Framework*[^DjangoREST]. This package allows for the creation of endpoints. When a new API endpoint is added, the package will automatically create a frontend view for that endpoint which will only accept the specified HTTP operations.

The API is used to connect the frontend with the disentangling module. The frontend makes API calls every time the user navigates to one of the pages or performs an action. In addition, the API can also be called separately with *cURL* or any other API client for REST (such as *Postman* or *Insomnia*). After an exploratory phase, this feature could be leveraged by researchers or users to integrate the results provided by the application into a custom pipeline for processing conversations.

Table 3.1 illustrates the different endpoints of the microservice. In addition, Figure 3.4 shows the flowchart of the three operations that can be carried out by *CODI*.

### 3.2.2 Disentangler

The disentangler is the core module of the microservice. Specifically, it is used to group messages into conversations. Conversation grouping is a two-step process consisting of a classifier and a clustering algorithm. For the classification, we used a *max-entropy classifier*. The second step consists of a *correlation clustering* algorithm. It provides as output the groups of messages divided into conversations.
### Table 3.1: REST API endpoints

<table>
<thead>
<tr>
<th>Path</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>api/convert</td>
<td>POST</td>
<td>Converts the annotated ANNOT dataset to an annotated JSON dataset</td>
</tr>
<tr>
<td>api/train</td>
<td>POST</td>
<td>Sends the input annotated JSON dataset to the disentangler. The latter will use this dataset for training</td>
</tr>
<tr>
<td>api/validate</td>
<td>POST</td>
<td>Sends the input annotated JSON dataset to the disentangler. The trained disentangler will predict the conversations and compare them to those given by the annotation – i.e., the gold set</td>
</tr>
<tr>
<td>api/predict</td>
<td>POST</td>
<td>Sends the input non-annotated JSON dataset to the trained disentangler. The latter will predict the conversations</td>
</tr>
<tr>
<td>api/statistics</td>
<td>GET</td>
<td>Retrieves the disentangled conversations and – when validating – statistics about the disentangled conversations</td>
</tr>
</tbody>
</table>

### Figure 3.4: Training, validation, and prediction flowcharts
3.2. Backend

Feature Extraction

Before feeding the messages to the classifier, we need a preprocessing phase. Two steps must be performed. In the first step, we extract all pairs of messages given some constraints. For the pair of messages to be examined, they need to be within a time window because “for utterances further apart than this, the classifier has no significant advantage over the majority baseline” [1]. Moreover, as suggested by Chatterjee et al. [3], we also consider message pairs outside the previously defined time window. The four messages preceding the current one are always analyzed independently of their arrival time for relatedness.

In the second step of the preprocessing phase, we extract the features from the pairs of messages previously generated. As done in [1–3], we extract the following features:

- **Chat-Specific Features**
  - **Time**
    The time difference between the first and second messages. This difference is discretized in logarithmically sized bins – which are one-hot encoded.
  - **Speaker**
    If the two messages have the same author or not.
  - **Mention**
    If the message of author one mentions author two, or vice versa. The one-hot encoding of this feature is made up of two bits. The first bit indicates if the first message mentions the author of message two, while the second bit indicates if the second message mentions the author of message one.
  - **Mention Same**
    If both messages mention the same member of the community.
  - **Mention Other**
    If either message one or message two mentions another member of the community who is not the author of either message one or message two. The one-hot encoding of this feature is made up of two bits. The first bit indicates if the first message mentions another community member, while the second bit indicates if the second message mentions another community member.

- **Discourse Features**
  - **Cue Words**
    Either message one or message two uses a greeting (e.g. *hello*), a one-word answer (e.g. *yes, no, thanks*), or a thanking answer (e.g. *makes sense, got it*). This feature is made up of six bits. Two bits are used for each group of cue words – i.e., *greetings, one-word answers*, and *thanking answers*. Of these two bits, the first one indicates if the first message contains one of the cue words of that group, while the second bit indicates if the second message contains one of the cue words of that group.
  - **Question**
    Either message one or message two contains a question – which is explicitly marked by a ?, or it contains a question word (e.g. *what, who*). This feature is composed of two bits. The first indicates if the first message contains a question mark or question word, while the second indicates if the second message contains a question mark or question word.
  - **Long**
    Either message one or message two are longer than ten words. This feature is composed of two bits. The first indicates if the first message is longer than ten words, while the second indicates if the second message is longer than ten words.
• Content Features
  
  – Repeat
  After building a unigram probability dictionary – of which the first 50 most frequent words are removed, and retrieving all the common words between message one and message two, bin the common words logarithmically based on their probability. Finally, the bins are one-hot encoded into ten bits.
  
  – Tech
  If either message contains technical jargon. This feature is composed of two bits. The first indicates if the first message has technical jargon, while the second indicates if the second message contains technical jargon.
  
  – Code
  If either message one or message two contains a code snippet. This feature is composed of two bits. The first bit indicates if the first message has a code snippet. While the second bit indicates if the second message includes a code snippet.
  
  – Link
  If either message one or message two contains a URL. This feature is composed of two bits. The first bit indicates if the first message contains a URL while the second bit indicates if the second message contains a URL.

Given these features, their one-hot encodings are condensed in matrix form to be used as input for the classifier. Each row of the matrix corresponds to a pair of messages, and each column corresponds to a bit of the one-hot encoding of a feature. This matrix can now be used as input for the max-entropy classifier.

Max-Entropy Classifier

In the first step of the actual disentangling algorithm, we use a binary classification regression model to determine whether the pair of messages are related to one another or not. A logistic regressor carries out this classification. Before starting to use CODI to disentangle conversations, the model must be trained. The user can use either the frontend or the API to send the regressor an annotated dataset through the api/train endpoint. Since this is a supervised machine-learning algorithm, the dataset must be annotated. When the model has finished training, the user can now perform validations and predictions on datasets. To perform validations, an annotated dataset must be sent to the api/validate endpoint. In contrast, in the case of a prediction, a non-annotated dataset must be sent to the api/predict endpoint of CODI.

Correlation Clustering

The final step of the disentangling algorithm consists in clustering the messages. Such a clustering algorithm aims to reconstruct groups of related messages based on the output of the relatedness classifier. This algorithm uses the previously found relations to create clusters of messages most closely related. Such a result can be obtained by implementing a correlation clustering algorithm. Being such an algorithm NP-complete, our reference paper [1] opted for a heuristic approach to the problem. In particular, the algorithm they used is a greedy voting algorithm [15].

The greedy voting algorithm (Algorithm 1) loops on all of the messages in a channel of a community (line 3). We loop on all clusters in C for each of these messages. For each cluster, we compute the sum of the weights – i.e., \( \omega_{ij} = p_{ij} - 0.5 \), where \( p_{ij} \) is the probability that messages \( i \) and \( j \) are related – and append it to array \( Q \) (line 6). After exhausting all clusters, we must find the clustered index with the highest quality value (line 8). Finally, if the value of \( C[c^*] \) is greater than 0, then we can append the current message \( i \) to cluster \( c^* \) (line 10); otherwise, we create a new cluster and append message \( i \) to this new one (line 12).
Algorithm 1 Greedy voting algorithm

1: \( C \leftarrow \text{array}[\cdot] \) \hspace{1cm} \( \triangleright \) The array of clusters
2: \( k \leftarrow 0 \) \hspace{1cm} \( \triangleright \) The number of clusters
3: \( \text{for } i = 1 \ldots n \text{ do} \)
4: \( Q \leftarrow \text{array}[\cdot] \) \hspace{1cm} \( \triangleright \) The array of qualities of each cluster
5: \( \text{for } c = 1 \ldots k \text{ do} \)
6: \( Q[c] \leftarrow \{\sum_{j \in C[c]} w_{ij}\} \)
7: \( \text{end for} \)
8: \( c^* \leftarrow \arg \max_{1 \leq c \leq k} Q[c] \)
9: \( \text{if } Q[c^*] > 0 \text{ then} \)
10: \( C[c^*] \leftarrow C[c^*] \cup \{i\} \) \hspace{1cm} \( \triangleright \) Add message \( i \) to cluster \( C[c^*] \)
11: \( \text{else} \)
12: \( C[k++] \leftarrow \{i\} \) \hspace{1cm} \( \triangleright \) Create a new cluster and add message \( i \) to it
13: \( \text{end if} \)
14: \( \text{end for} \)

3.3 Frontend

In the frontend module of CODI, we have created several pages for users to interact with the backend quickly (Table 3.2). Users can visualize the generated conversations and valuable statistics on the validation process. This module allows the user to select only a subset of the features and perform training, validation, or prediction operations again to evaluate the impact of the selected feature groups on the disentanglement process.

<table>
<thead>
<tr>
<th>Path</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Home</td>
<td>Here the users can upload a JSON file containing the messages of the community.</td>
</tr>
<tr>
<td>statistics/validation</td>
<td>Validation Statistics</td>
<td>Here the user can check the performance and statistics of the most recent validation operation performed.</td>
</tr>
<tr>
<td>statistics/prediction</td>
<td>Prediction Statistics</td>
<td>Here the user can check the conversations of the most recent prediction operation performed.</td>
</tr>
<tr>
<td>annotation</td>
<td>Dataset Annotation</td>
<td>Here the user can manually annotate a dataset given as input.</td>
</tr>
</tbody>
</table>

Table 3.2: Frontend pages

3.3.1 Homepage

In the homepage of CODI (Figure 3.5), on the left, we have the section of the page where the user can drop the JSON file of the community. Instead, we can find the section dedicated to some file settings on the right. The user can specify the type of operation to perform, which can be training, validation or prediction. It can also specify which platform the message was taken from, which can be auto – this means that the platform is inferred from the platform field of the JSON file, Discord, or Slack. Finally, we have three checkboxes representing the group of features we want to extract from the messages. These groups are chat-specific features, discourse features, and content features.
3.3.2 Validation Statistics

In the validation statistics section (Figure 3.7), the user can view the statistics of the most recently performed validation. The statistics include the time performance of the algorithm, the micro-averaged F1 score of the disentanglement, and the accuracy, precision, recall, and F1 score of the single feature groups – as well as the combined one – of the max-entropy classifier. Finally, the messages are shown with the resulting conversation assignment (Figure 3.6). On the left side, the output of CODI can be compared with the ground truth on the right side. Conversation groups are color coded to help identify them better than by simply looking at their conversation id.

![Figure 3.5: Homepage view](image)

![Figure 3.6: Validation statistics collapsed view](image)
3.3. Frontend

3.3.3 Prediction Statistics

In the prediction statistics page (Figure 3.8), we have a similar layout as the one seen on the previous page. Unlike the other page, here, the two tables both represent the messages coming from the disentangler. The table on the left represents messages in chronological order, while on the right, we have the same messages but grouped in their corresponding conversation. Moreover, if a user clicks on one of the messages on the left, it will be guided to the same message as assigned to a conversation on the right – and vice-versa.
3.3.4 Dataset Annotation

The user can upload a JSON file containing a Discord or Slack community dump to the dataset annotation page. When CODI has finished analyzing the data, it will present a view similar to Figure 3.9. On this tab, the user can manually annotate the conversations for each of the community’s messages.

Let’s assume that the user wishes to complete annotating the data later. In that instance, the partially annotated dataset can be saved by clicking the "Finish Annotating" button at the top or bottom of the message list. When saving the conversation numbers, a None will be added wherever a label has not been yet assigned. Otherwise, the user-specified conversation number will be saved. When the user decides to upload the partially annotated file again, CODI will automatically recognize whether a message has already been assigned to a conversation or not and display it accordingly – as shown in Figure 3.9.

![Figure 3.9: Dataset annotation](image-url)
Chapter 4

Result Analysis

In this section, we will describe the experiments we carried out to assess the correctness and performance of our version of the two-step method implemented by Elsner and Charniak [1,3], and Chatterjee et al. [2]. The first experiments were carried out utilizing annotated datasets provided by the original authors, which can be obtained from their GitHub repository. More tests were subsequently carried out, this time using the Discord dataset we annotated to determine any potential limitation of such an implementation.

4.1 Evaluation Methods

We use several types of methods to evaluate the solution produced by CODI. We use accuracy, precision, recall and F1 score for the binary classification – i.e., the max entropy. We use microaveraged F1 score for the evaluation of the clustering part of the algorithm – i.e., correlation clustering.

These measurements were useful to compare our implementation with the original one. Since we re-implemented from scratch the message pairs creation, the feature extraction, the max-entropy classification and the correlation clustering, we needed to make sure that everything was working correctly and that the performance of CODI was on par with the performance of the original implementation by Elsner and Charniak [3] – and build up from there.

4.1.1 Max Entropy

The accuracy measures how many times the model guessed correctly. Precision measures how precise the model is by considering all of the predicted positives – true positives and false positives. Recall, on the other hand, measures how precise the model is by considering all of the actual positives – false negatives and true positives. Finally, the F1 score is the harmonic mean between precision and recall. This measurement is used to determine which classifier produces better results.

As done by Elsner and Charniak [1] and Shen et al. [16], let’s assume we have a gold conversation \(i\) with size \(n_i\), a predicted conversation \(j\) with size \(n_j\), and an overlap between the two of size \(n_{ij}\). Then we can write the formulas for precision, recall and F1 score as follows:

\[
P = \frac{n_{ij}}{n_j} \quad R = \frac{n_{ij}}{n_i} \quad F1(i,j) = 2 \cdot \frac{P \cdot R}{P + R}
\]  

(4.1)

4.1.2 Correlation Clustering

For the evaluation of the clustering algorithm, we cannot use the F1 score described above. After clustering the messages, we are not dealing with two classes anymore. The number of classes is now equal to the number of conversations created by the clustering algorithm. Moreover, the conversations generated might not be perfectly lined up with the "gold" conversations. If we were to use the F1 score, then a slight
misalignment could cause the score to drop and be inaccurate.

Thus, we need to use another evaluation measure that will compute a global average F1 score. This evaluation measure is called micro-averaged F1 score (Formula 4.2).

\[
F1 = \sum_{i} \frac{n_i}{n} \max F1(i, j)
\]

(4.2)

### 4.2 Disentanglement Performance

#### 4.2.1 Original Datasets

To verify the correctness of our implementation with respect to the state-of-the-art [1–3], we’ve tested CODI with extracts from both the Python and Clojure datasets present in the literature [6]. We’ve found that the two implementations give comparable results by running such tests. In Table 4.1 we have the performances of the combined features for both the max-entropy classifier – accuracy, precision, recall, and F1 score; and the performance of the correlation clustering algorithm – micro-averaged F1 score.

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1 Score</th>
<th>Micro-Averaged F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>python</td>
<td>60</td>
<td>86</td>
<td>62</td>
<td>72</td>
<td>78</td>
</tr>
<tr>
<td>clojure</td>
<td>67</td>
<td>94</td>
<td>68</td>
<td>79</td>
<td>88</td>
</tr>
</tbody>
</table>

Table 4.1: CODI results from original datasets

#### 4.2.2 Custom Dataset

To further test our implementation’s robustness and accuracy, we chose to annotate 294 messages from a Discord server about Roussal development. We’ve purposefully chosen these messages because many conversations are interleaving one another. After training the model with the training_slack.annot annotated dataset – which has been used by the original authors [1–3] as well to train their models; by performing a validation operation on the pharo.annot dataset, we obtained the results displayed in Table 4.2.

<table>
<thead>
<tr>
<th></th>
<th>Chat Features</th>
<th>Discourse Features</th>
<th>Content Features</th>
<th>Combined Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>68</td>
<td>64</td>
<td>69</td>
<td>68</td>
</tr>
<tr>
<td>Precision</td>
<td>82</td>
<td>80</td>
<td>78</td>
<td>68</td>
</tr>
<tr>
<td>Recall</td>
<td>76</td>
<td>72</td>
<td>83</td>
<td>77</td>
</tr>
<tr>
<td>F1 Score</td>
<td>79</td>
<td>76</td>
<td>81</td>
<td>79</td>
</tr>
<tr>
<td>Micro-Averaged F1</td>
<td>66</td>
<td>53</td>
<td>47</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 4.2: CODI results from the Roussal dataset

After training both CODI and Chatterjee et al.’s implementations with the original training_slack.annot, we fed pharo.annot to both the modified version of the original algorithm [2], and to our algorithm. By doing so, we obtained a micro-averaged F1 score of 61 for Chatterjee et al.’s implementation and a micro-averaged F1 score of 63 for CODI. This result suggests that the algorithm proposed by Chatterjee et al. can
sometimes find it challenging to disambiguate certain conversations.

The inability to properly disambiguate conversations may be caused by the fact that this algorithm does not work well with long one-argument conversations, where two or more people talk about the same topic for an extended period of time. We have noticed that, in some cases, it seems like the algorithm forces the separation of messages, which in reality, are part of the same conversation. For example, we can see in Figure 4.1 that both CODI and Chatterjee et al.’s algorithm perform similarly. In the red boxes of the two images, we can see how the two messages are being forced into a new conversation – even though they are part of the previous conversation.

![Figure 4.1: Conversation disentanglement comparison – on the left CODI, on the right Chatterjee et al.](image)

### 4.3 Disentanglement Quality

In addition to performance analysis of CODI, we also perform a qualitative analysis of the generated conversations. This qualitative analysis was performed on the conversations generated by a model trained with training_slack.annot and predicted on pharo.annot. Figure 4.2 shows a conversation prediction made by CODI. In this case, we can see that the algorithm correctly separates conversations 2 and 3. Conversation 2 is resumed with the last message (3). As we can see also on the right of the image, the interleaving conversation is separated from the original conversation (2), and the last message is appended to conversation 2. The second message (1) defines another of the feature extraction problems performed by CODI. The problem is that Stepharo tries tagging user Alexandre Bergel but uses the wrong name. This causes Discord to not recognize it as a valid mention, and the exported dataset contains @alex instead of @Alexandre."

Figure 4.3 shows another section of the disentangled conversation. This dataset is the same as the one used in 4.2. In this case, we have a correct disentanglement again. Here we can see that conversation 3 is interrupted by Ellis Harris. Correctly, CODI considers this a separate conversation and gives it an ID of 9.
This conversation is then interrupted by Alexandre Bergel – who is answering to chicoary. Finally, Alexandre Bergel answers to Ellis Harris, making the last message part of conversation 9.

### 4.4 Time Performance

In addition to computing the disentangling performance, CODI also computes the time each step of the algorithm takes. Table 4.3 summarizes the times of the three datasets we’ve tested CODI on (the results are in seconds).

<table>
<thead>
<tr>
<th></th>
<th>Training</th>
<th>Max-Entropy</th>
<th>Correlation Clustering</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>python</td>
<td>103.6</td>
<td>8.5</td>
<td>0.1</td>
<td>166.3</td>
</tr>
<tr>
<td>clojure</td>
<td>104.1</td>
<td>3.5</td>
<td>0.06</td>
<td>122.5</td>
</tr>
<tr>
<td>pharo</td>
<td>110</td>
<td>3</td>
<td>0.02</td>
<td>120.5</td>
</tr>
</tbody>
</table>

**Table 4.3: CODI times for all datasets**
The *training* column indicates how much time the training of the model took in total. All models were trained with the `training_slack.annot` dataset. The *max-entropy* column indicates the time CODI takes to extract all the features from the pairs of messages and make a prediction using the trained max-entropy classifier. The *correlation clustering* column represents the time the algorithm uses to perform the greedy voting algorithm that creates the clusters of conversations. Finally, the *total* column represents the time CODI used to train the max-entropy model, extract the features from the message pairs, run the max-entropy classifier on the extracted pairs, and finally cluster the messages into conversations.

## 4.5 Summary and Outlook

In this section, we’ve presented comparisons with the reference implementation to assess the correctness of CODI. We’ve also seen limitations of the algorithm. In some cases, it performs very well, as in the case of the `python` and `clojure` datasets. However, the custom dataset we annotated highlighted its limitations. This is mainly because the messages in the dataset have some problematic sections to disentangle. In the next chapter, we will give some concluding remarks on the projects and suggest some enhancements that can be done to CODI.
Chapter 5

Conclusion and Future Work

With this project, we implemented a REST API capable of disentangling conversations using state-of-the-art algorithms. The service, in addition to an API, has a graphical client, which makes the tasks of training the model, validating, and predicting simple and fast. CODI also offers – in the case of a validation operation – the possibility to receive statistics regarding the performance of the algorithm and the quality of the conversations. Its ease of use makes it the best tool for even non-technical researchers to choose in the exploratory phases of their research. In addition to being able to disentangle conversations, CODI also offers the possibility to upload a non-annotated dataset. This can be manually annotated by the user and exported directly as a JSON file.

5.1 Future Work

Many aspects of CODI can be further expanded upon. The following are some possible enhancements:

- **Customizable First Step**
  The first step of the disentangling algorithm can be made customizable. This means that we could offer several algorithms and models to choose from, and the user – from a specific view – can decide which of the algorithms to use.

- **Selectable Hyperparameters**
  Another possible enhancement is to make the hyperparameters selectable. This means that from a particular view, as in the case of the pluggable algorithm discussed earlier, the user can select the values of all the hyperparameters used by CODI.

- **API Keys**
  Finally, another possible enhancement would be implementing access based on API keys. This would mean that each user can train and predict disentanglements with CODI as if it had many isolated compartments assigned to a key – i.e., a user. By doing so, the models trained and used by each user would not be able to influence any other model trained by any other user of the service.

5.2 Concluding Remarks

As shown throughout this report, our implementation of the two-step algorithm works very well with some datasets and not that well with others. This means that this algorithm is not an all-around good and changes significantly based on the dataset we use to validate the model. Finally, some conversations are challenging – if not impossible – to disentangle even by a human being. This encourages extending CODI with new state-of-the-art models, leveraging its architecture to simplify comparisons and extensive experimentation on different datasets.
Bibliography


gitter platform,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 1, 

"party" in "multi-party"? Analyzing the structure of small-group sociable talk,” in Proceedings of the 

in Proceedings of the Workshop on Integer Linear Programming for Natural Language Processing, pp. 19–27, 
2009.

[16] D. Shen, Q. Yang, J.-T. Sun, and Z. Chen, “Thread detection in dynamic text message streams,” in Pro-
cedings of the 29th annual international ACM SIGIR conference on Research and development in information 
retrieval, pp. 35–42, 2006.