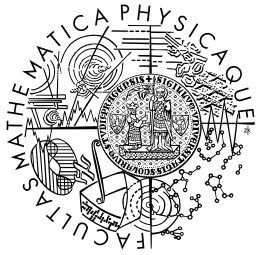

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Viktória Brezinová

Automatic Extraction of the Main Characters from Books and Their Interactions

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: RNDr. David Mareček, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank RNDr. David Mareček, Ph.D., the supervisor of this thesis, for his help, answering all my questions and finding time to consult the work almost any time.

I would also like to thank my family and my boyfriend for their support even in the hardest times.

Title: Automatic Extraction of the Main Characters from Books and Their Interactions

Author: Viktória Brezinová

Institute: Institute of Formal and Applied Linguistics

Supervisor: RNDr. David Mareček, Ph.D., Institute of Formal and Applied Linguistics

Abstract: The goal of this work is to automatically find named characters in the books, detect all occurrences of these characters and determine places in the text where two or more characters interact together. One of the outputs of this work is the tool for displaying interactive graphs that show us the occurrences and interactions of the characters throughout the book. We can search and analyze the places of occurrences and interactions using this tool, since the graphs are connected to the text of the book. We also evaluated our methods on the unseen texts, analyzed errors, and proposed improvements that could be explored in future work.

Keywords: extraction of characters named entities fiction

Contents

Introduction

Fiction books are an important part of many people's lives and a great source of entertainment. But there are more books than one person can read in his lifetime. Many recent works in the field of natural language processing focus on the automatic processing of texts and extracting relevant information from them. This information might help us to quickly find out which books we might want to read or just give us some information without reading the whole text. The goal of this work is to try to automatically detect interactions between characters. We firstly needed to find characters and their mentions in the text to work with this information when trying to find interactions. The tool for creating interactive graphs with occurrences and interactions of the characters from the text is part of our work. We can use it to analyze the text, see which characters occurred in which parts of the text, or see which characters interacted together the most.

The first chapter describes the task of finding the character names in the text and merging different names of the same character. The main problem with creating a list of all characters from the book automatically is that one character can be called different names in different parts of the book.

In the second chapter, we describe the problem of finding all the occurrences in the text for each of the characters. We want to find all character mentions in the text, even the mentions by a pronoun or a nominal. We use a trained model for coreference resolution for this task and further process its output.

The third chapter deals with the most interesting part of this work. We try to detect places in the text where two or more characters interact together. We firstly needed to define what interaction is, as it is not an easy question. Then we analyzed texts from various books and tried to find out if some rules could help us detect interactions. We focus on dialogues, as they are the clear sign of interaction. We also developed a set of rules that can help us detect sentences in the text that describe the interaction between characters.

In the fourth chapter, we describe the final output of our work – the interactive graphs which display occurrences and interactions of the characters throughout the book. We can quickly find out which characters were probably important for the plot or which characters interacted together the most just by looking at the graph.

The fifth chapter focuses on the evaluation of the methods implemented on the unseen data.

In the sixth chapter, we describe the implementation of our work in `Python`.

The seventh chapter describes all the technical details and steps we must do before using our tool and also different options for using it.

1. Finding Characters in the Text

Characters are the essence of all literary texts. When we want to automatically process a book and find information about what is happening in it, we usually need to start by finding the characters in the book.

The first thing we need to do is to define what a character is. The definition of the character differs in different existing works. Labatut and Bost [2019] mentions that a character can appear in the text in three forms: as a proper noun, a nominal or a pronoun. All existing works detect proper nouns but not necessarily two other forms. In some works, the authors also detect characters that do not have any proper name in the text, but are only referred to by pronouns or nominals (for example, "the gardener"). Other authors only detect characters that were referred to by a proper noun at least once in the text.

1.1 Our Definition of a Character

We define a *character* as an animate being that has at least one proper name and was referred to by this proper name at least once in the text.

This definition can overlook some characters that do not have a proper name and are referred to only by nominals and pronouns. We think that these characters usually are only minor characters and so this won't be a big problem for most of the books.

1.2 Problems with Detecting Characters

Most characters in the books are referred to by a personal title (also called honorific), first name, last name, nickname, or diminutive of the first name, or some combination of this (for example, a personal title with the last name or a full name). The main problem is that one character can be called differently throughout the book. For example, a character named "Margareth March" can be called "Margaret", "Meg" or "Mrs. March". It depends, for example, on who is talking to her. Sometimes, it is difficult to tell whether "Margareth" and "Meg" are the same person or not without reading the full text.

Another problem is that two or more characters can have the same first name or the same last name, and we must decide which character it refers to from the context.

1.3 Background

We now describe what tokenization and name entity recognition is, because we will use these terms in the rest of this chapter.

1.3.1 Tokenization

Tokenization is the task of dividing a raw text into smaller meaningful chunks, called *tokens*. There exist different types of tokenization, for example word to-

kenization or subword tokenization. We use word tokenization in our work.

Word tokenization divides text into tokens that are usually individual words and punctuation marks. But the output tokens can differ depending on a tokenizer. We use the `Tokenizer` for English from the `SpaCy` library [1] in our work. This tokenizer firstly splits text on whitespace characters and then applies a set of rules that are specific for each language. These rules split off punctuation marks to separate tokens or further divide some words into more tokens. For example, "don't" is divided into two tokens – "do" and "n't". We can show an example of tokenization for a simple sentence:

The sentence: *"Mrs. March doesn't live in the U.K. but in the USA."*
Tokens: *"Mrs.", "March", "does", "n't", "live", "in", "the", "U.K.", "but", "in", "the", "USA", "."*

1.3.2 Named Entity Recognition

Named entity recognition (NER) is the task of identifying expressions that refer to the named entities in a text and tagging them with their corresponding type. A *named entity* is an object with a proper name, like a location, a person or an organization. According to Mansouri et al. [2008], there are different approaches to NER used nowadays. For example, Rule-base Ner, Machine Learning-base NER or Hybrid NER.

Named entity recognition models usually also assign inside-outside-beginning tags to the input tokens. The beginning tag means that the token is the first token of the name of the entity. The inside tag means the token is a part of the name of the entity, but not the first one. The outside tag means that the token is not a named entity. We use pretrained models for NER in our work.

We can show an example of using the NER tool:

The input sentence: *"Mark Zuckerberg is one of the founders of Facebook, a company from the United States."*
The output:

- "Mark Zuckerberg" is tagged as Person, "Mark" has the beginning tag, "Zuckerberg" has the inside tag.
- "Facebook" is tagged as Company with the beginning tag.
- "United States" is tagged as Location, "United" has the beginning tag, "States" has the inside tag.

1.4 Existing Approaches

Many approaches to character identification were explored in the survey by Labatut and Bost [2019]. Some authors use a predefined list of characters either from Wikipedia or manually created and only try to match names found in the text to the characters from the list. Other authors don't use any additional information, like the list of characters, but try to find the characters automatically,

[1] `https://spacy.io/api/tokenizer`

usually using some tool for Named Entity Recognition and retaining only entities tagged as Person.

We describe a few approaches that inspired our approach in some way.

1.4.1 Multi-stage Clustering Approach

Elsner [2012] create a list of all possible character references detected by the NER parser and then discard the references that occur less than 5 times. Their system uses the multi-stage clustering approach for determining which references in the list refer to the same character. First, they merge all identical mentions that are at least two words long. Then, they assign each of these mentions a gender using a list of female and male names and gendered titles. Next, they merge mentions in which genders do not clash, first and last names are consistent, and the mentions are at least two words long. The last step is merging one-word mentions to matching multiword mentions if the mentions appear in the same paragraph. When a one-word mention matches some multiword mentions but none of them is in the same paragraph, then this mention is merged with the matching multiword mention that occurs in the most paragraphs.

1.4.2 Structure-based Clustering Approach

Coll Ardanuy and Sporleder [2014] introduce an interesting idea, that some names, for example "Leicester", might be tagged as a person in one paragraph and as a location in the other paragraph. They assume that in a novel one proper name is likely to refer to the same entity throughout the whole novel. Because of this, they take into account only the most frequent tag for each name. So, the name is considered to be a character name if it was tagged as a person more times than as a location (or some other tag).

They do the task of resolving which names refer to the same character in three steps:

1. Parse the name into components like title, first name and last name.

2. Assign gender to each name using lists of female and male personal titles and female and male first names and looking at pronouns in an immediate context.

3. A matching algorithm is performed. It groups different names of the same character together. They process the names in the given order depending on the parts it consists of. They start with the three word names that consist of title, first name and last name, continue with names that consist of first and last name, then names with title and first name, then names with title and last name and they match the one word names in the last step.

   In the matching algorithm, it is considered that a first name can appear as a nickname or an initial and that the genders must agree when merging two names together. If it is ambiguous, because we have, for example, two characters with the same last name, then it is assumed that name refers to the most relevant character from the candidates (the most frequent one).

1.4.3 Eight-stage Pipeline Approach

Vala et al. [2015] proposed a novel idea that uses an eight-stage pipeline to detect characters. In this pipeline, a graph is being built. Nodes of the graph are names of the characters and edges connect the names that refer to the same character.

1. Firstly, the nodes are initialized with names from the output of the NER model.

2. In the second stage, the coreference resolution is run on the text, and the edges are added between the names that are in the same coreference cluster.

3. Thirdly, some name variation rules are applied to add edges between nodes that may refer to the same character (for example, the name is the same as the other name after removing a personal title).

4. The list of hypocorisms is used to connect names like Tim and Timmy.

5. In the fifth stage, there are three rules that prohibit the merging of two names. Two nodes are not merged if the genders of the names are different, or the names have the same last name, but different first names or if the honorifics of the names are different. This results in removing some of the edges created before.

6. The sixth stage prohibits the merging of two names if they appear in the text connected by a conjunction, or one name is the speaker mentioning the other name in the direct speech, or both names appear together in one quote.

7. Characters that do not have a proper name and thus are not recognized by NER are trying to be identified. The noun is identified as a character if it appears in a dependency relationship with a verb that is typical for people (for example, say or eat). Generic nouns from the list, like a man, are not added.

8. The eight stage removes nodes that are disconnected from the rest of the graph and contain a name that is a part of some other name in another node. These nodes are removed because they typically represent ambiguous names of some characters that are represented by the other nodes, so they are not needed.

The remaining nodes are merged according to edges to create sets of names that are associated with different characters.

1.4.4 Matching Names to Predefined List of Characters

Lajewska and Wr'oblewska [2021] work with a list of full names of characters obtained from Wikipedia. They try to match each entity tagged as a person by the NER model to one of the character's in the list. Their matching algorithm firstly tries to find the most similar name in the list using partial string matching measured by Levenshtein distance.

If the closest match is not similar enough to the found name, then the list of diminutives is used. The list of diminutives contains different name variations for common English names. For example, we can find in it that Margareth can be also called Meg.

The last special case that is discussed is when a named entity is referred to only by a last name. In this case, they distinguish whether it refers to the single character or the whole family by analyzing the word before a last name. If the word before is a personal title, then it is matched to the one character with the correct gender. In all the other cases, the name is treated as the whole family name and not a single character.

1.5 Our Approach

We assume that our input is a tokenized text and we want our output to be a list of all named characters from the text. We want to know a name for each character (it can have more parts like a title and last name) and also a set of other names that occurred in the text and we think that they may refer to this character, for example because this other name is a diminutive of the first name of this character. We also determine the gender of each character if possible. The goal of this part of work is to find all named characters that occurred in the text. We are not trying to find occurrences of these characters in the text (that is discussed in the chapter 2). So if the name in the text is ambiguous (can refer to more characters), we don't care about what character it refers to, it is enough that we know that the given name does probably refer to some of the already found characters.

Firstly, we need to find candidates for the names of the characters. We can find all names that refer to a named entity by using a trained tool for Named Entity Recognition, and then work only with names tagged as Person. We use the `NameTag 2` and the NER tool from the `SpaCy` library for this task. We wanted to find out if there would be a difference between the outputs of these two tools, so we tried to use both.

When we get a name from the NER tool, we look at the token before the name and if it is a personal title, we add it to the name. We do this because the NER tool does not always find a name with the title together. We also applied a rule described in Coll Ardanuy and Sporleder [2014]: we keep the name only if it was tagged "Person" more times than some other tag. That means, for example, that if a name "Florence" was tagged many times as a "Location" and only once as a "Person" in the text, then we don't keep Florence in our list of character names. We assume that it probably was a tagger mistake.

1.5.1 NameTag 2

`NameTag 2` [2] is an open-source tool for Named Entity Recognition that achieved state-of-the-art for English in 2019. According to Straková et al. [2019], this tool also recognizes nested entities of arbitrary depth, entities that overlap and also supports labelling entities with more than one label. It uses a neural model.

[2] `https://ufal.mff.cuni.cz/nametag/2`

The tool with the `English CoNLL Model` uses 4 different labels for different types of named entities – PER, ORG, LOC and MISC. In our work, we work only with entities labelled as PER, which means person.

1.5.2 SpaCy NER

The `SpaCy` library includes the statistical entity recognition system [3], which we use in our work. The system only identifies non-overlapping labelled spans of tokens. This tool uses 18 different labels for different types of named entities, but we work only with entities tagged as PERSON.

1.5.3 Processing the Output of the NER Tool

When we get the set of names as the output of the NER tool, we must further process it to get a list of all characters in the text. We want to merge different names that refer to the same character. Then, we want to create a list of all characters, and we also try to determine the gender for every character.

From our analysis of the approaches described in section 1.4, we came to the conclusion that we want to divide the names to some meaningful parts like personal title, first name and last name. Then we want to gradually merge names starting with the multiword names using some rules, a list of diminutives and gender information. This was mainly described in Elsner [2012] and Coll Ardanuy and Sporleder [2014]. But the role of genders in merging was also described in Vala et al. [2015]. In contrast with the described works, we don't have to resolve what to do with ambiguous names, as we only want to get the list of characters without their occurrences. So we can simplify some of the rules. Our approach is the most similar to the Multi-stage Clustering Approach and Structure-based Clustering Approach, but we added using a list of diminutives and detecting family names from Lajewska and Wr'oblewska [2021].

We decided this part will work also without using the coreference resolution. Therefore, our basic algorithm for detecting characters doesn't use the coreference resolution in any way. It uses simple deterministic rules, a list of diminutives[4], lists of female, male, and androgynous honorifics[5] and lists of female and male first names[6].

We manually created the list of female, male, and androgynous honorifics with the help of a list from Wikipedia. We also added names of some family members like mother, father, uncle, aunt to these lists because we observed that our NER models also output these words as parts of names sometimes (for example, "Uncle Vernon") and we want to treat these words in the same way as personal titles.

We divide the names of the characters into three main parts as in Coll Ardanuy and Sporleder [2014]: personal title, first name, and last name. Each character can also have some nicknames or diminutives assigned.

[3] `https://spacy.io/usage/linguistic-features#named-entities`

[4] `https://github.com/carltonnorthern/nickname-and-diminutive-names-lookup` used in Lajewska and Wr'oblewska [2021].

[5] Manually created by us.

[6] Source: `http://www.cs.cmu.edu/Groups/AI/areas/nlp/corpora/names/`, used in Coll Ardanuy and Sporleder [2014]

The token is considered to be a title if it is the first token in a name and it is in our list of honorifics.

The token is considered to be a first name if it is in our list of female names, male names, or diminutives, or it is the first token of multiword name (and it is not a title), or when the first token is a title and this token is the second token of at least three tokens long name.

The token is considered to be a nickname if it is in the list of diminutives, and we have a character that can be called by this diminutive (because his first name can refer to the same name as this diminutive).

The token is considered to be the last name if it is not a title, first name, or diminutive.

If the name consists of more tokens considered to be the last name, then these tokens are jointly considered a last name (so the last name can contain more tokens).

This division has some problems when dealing with characters that also have a middle name, but we decided to ignore this case, as we think it doesn't occur often in the books.

Every character has exactly one of these values assigned as a gender: `female`, `male`, `plural` or `unknown`. We use gender `plural` for the family names that refer to more people, like "Weasleys". The idea to treat family names separately and have a different gender label for them came from the work of Lajewska and Wr'oblewska [2021].

We guess the gender of a character by the following rules (in the given order):

1. If the character has a personal title that is in the list of female or male titles, then we assign the corresponding gender to the character.

2. If the character's first name is in the list of female or male names, then we assign the corresponding gender to the character.

3. If the character has a first name, then we use a gender guesser[7] to guess the gender. This gender guesser takes a name as an input and outputs one of these labels: `unknown` (name not found), `andy` (androgynous), `male`, `female`, `mostly_male`, or `mostly_female`. If the output is male or mostly male, we assign a male gender. If the output is female or mostly female, we assign a female gender.

4. If the last letter of the name is "s" we assign it a plural gender.

5. If none of the above assigned gender, the gender remains unknown.

This assignment is not a hundred percent correct. Assignment based on the title should be correct, but when we just guess based on the first name, we can be wrong as some names can be used for both females and males. And when the last letter of name is "s" it doesn't always mean that it is a family name.

In our algorithm, we first divide all found names by NER into three groups: full names that consist of a title, first name and last name, two word names that

[7] `https://pypi.org/project/gender-guesser/`, mentioned in Lajewska and Wr'oblewska [2021]

consist of title and name, or first name and last name, and one word names that consist only of one token.

Now, we describe the steps for merging different names that refer to one character.

1. All the different full names (consisting of all three parts) are considered to be different characters. We create characters with corresponding names and assign genders to them.

2. We look at names that consist of two parts. In contrast with Coll Ardanuy and Sporleder [2014], we don't divide these names into three groups based on which two parts the name consists of. We do this in one step, as we think the order of merging doesn't make a difference in these cases.

3. If it is a first name and a last name and we already have a character with the corresponding names, then these names refer to the same character. This merges for example "John Walker" to "Mr. John Walker".

   If we have a character with the same last name and the first names are different but one can be the diminutive of the other, then we assume these names refer to the same character. This merges for example "Meg March" to "Margareth March", as in our list of diminutives we have that "Meg" is diminutive of "Margareth".

   If the first name is an abbreviation (for example, "H.Potter") and we have a character with the same last name and this abbreviation can be an abbreviation of his first name, these names refer to the same character. This merges, for example, "H. Potter" to "Harry Potter".

4. If it is a title and a name and we already have a character with the corresponding title and name, then it refers to the same character. This merges "Mr. Brown" to "Mr. John Brown".

   If we have a character with the same name (first or last) and this character doesn't have any title yet and the detected gender of the title and a character do not clash, then we assume that it is the same character and assign him this title.

   The fact that genders do not clash means that both names have the same detected gender or that one or both names have assigned an unknown gender.

   This merges "Mr. Brown" to "John Brown" as we detect that both are of male gender. It can also merge "Dr. Brown" to "John Brown", as the gender of "Dr.Brown" is unknown (because "Dr." is in an androgynous title).

5. If none of the above holds, we create a new character with the corresponding names and assign him a gender.

6. Lastly, we look at one word names. If this name is a first or last name of some character, then we assume that it refers to the same character. This matches "John" to "John Brown" or "Brown" to "John Brown".

If this name can be used as a diminutive of some existing character's first name (or vice versa), then we assume that it refers to the same character. This matches "Meg" to "Margareth March".

If none of the above holds, we create a new character with this name and assign a gender.

If we have some diminutive that can refer to more characters, then we add this diminutive to all of these characters as an other name, because we are not trying to decide right now which character it refers to.

After this procedure, we have what we think is a set of characters with all parts of their names that we found and also a set of other names that could possibly refer to this character. We will use this output in the next chapter.

2. Detecting occurrences

The goal of this part is to detect all occurrences of all characters in the text. We have a tokenized text and a list of all characters with all their possible names in the text as an input and we want to have a list of all occurrences for each character as the output.

We have two options for finding occurrences – with or without using the model for coreference resolution. We also tried to process the output of the coreference resolution model in two distinct ways.

2.1 Coreference Resolution

Coreference resolution is the task of finding all linguistic expressions that refer to the same entity in the text. The output of the coreference resolution model are usually the coreference clusters.

A *coreference cluster* is simply a collection of all tokens that refer to one real-world entity in the text (this cluster usually consists of numbers that are indices of corresponding tokens in the tokenized text).

A *mention* is a token or some number of subsequent tokens that refer to one character. It can be a name, a pronoun, a nominal or some description, like "a tall boy". We can represent the mention by two numbers: a pair of indices of first and last token for this mention in the tokenized text. The coreference cluster is then a list of all mentions of one character in the whole text.

The coreference resolution is a crucial part of all systems that deal with natural language understanding.

We can show an example of coreference resolution clusters:

The input sentences: *"Jane Young went to the concert. She met David there."*
Output clusters:

- In the first cluster, there will be a "Jane Young" and "She".
- In the second cluster, there will be a "concert" and "there".
- There can be a third cluster, in which only the token "David" is. Some coreference resolution models output also clusters of length 1 as this one. Other models do not output clusters with only one mention.

The coreference resolution helps us find pronouns and nominals that refer to one of the characters. This is really important when we want to find out more about the characters and their occurrences in the text as normally the character is referred to by his name only a few times in the section of the text and then is referred to by pronouns like he or she many times.

The coreference clusters can also help us decide which ambiguous names refer to which character.

2.2 Our Approach

Besides the input in the form of character's different names, we also create dictionaries with all the first names and last names (we include all nicknames or diminutives in the first names). The dictionary of the first names contains all the first names as keys and for each first name we have a list of characters with the given first name as its value. Similarly, the dictionary of the last names contains all the last names as keys and for each last name we have a list of all characters with the given last name as its value.

Imagine there were three characters in the text: "John Brown", "Amy Brown" and "John Small". Then in the first names' dictionary, we would have keys "John" and "Amy". John would have a list of characters John Brown and John Small as its value. Amy would have a list containing only a character Amy Brown as its value. It would be similar for the last names' dictionary with two keys: "Brown" and "Small".

This will be useful for quickly finding characters with a given name and deciding whether the name is ambiguous or not. When the name has only one character in the list, we know it is unambiguous as no other character has this name. In the above example, Amy is an unambiguous first name and Small is an unambiguous last name.

If the list contains more characters, that name is ambiguous without further context, as the names John and Brown in the above example.

For example, if we have this sentence in the text:

"John was sitting and reading a book.",

then we don't know for sure if John is John Brown or John Small without further context.

2.2.1 Finding Unambiguous Character Occurrences

Now we describe our algorithm.

The first step of our algorithm is to find occurrences of the characters that are unambiguous. Those are the places where a character is referred to by his proper name and this name belongs only to that one character.

First we go through all tokens in the text and look if this token is a first name or nickname for some of the characters in our list. If it is, then we decide if this name is unique for that character with the help of preceding word (if it is a title) or following word (if it is a last name) and our first names' dictionary. If this name uniquely identifies one character (no other character has the same name, or the same combination of title and first name, or first name and last name), then this mention is assigned to that character. Similarly if the token is a last name for some of the characters, we decide if this last name is unique for one of the characters with the help of the preceding word (if it is a title) and our last names' dictionary. If it is unique, we assign this mention to that character. We store the ambiguous names that were not assigned for the later.

The second step differs depending on whether we want to use a model for coreference resolution or not. We first describe our solution using a coreference resolution model.

2.2.2 Detecting All Occurrences

In the second stage we want to detect all occurences of the characters. That means also mentions that refer to the character by pronoun or nominal. And ideally we also want to resolve ambiguous mentions that were not resolved in the first stage. The coreference resolution helps us with this task.

Coreference Resolution Model

We use a pretrained model for coreference resolution. This model outputs all types of coreference clusters, not just clusters for our characters. Some clusters contain for example mentions that refer to the same thing like table or to the same location like house. Therefore, we need to filter the output of the model and also assign clusters to individual characters.

We use a `fast-coref`[1] model for coreference resolution in our work [Toshniwal et al., 2020] [Toshniwal et al., 2021]. We've decided to use this model, because it is optimized for long document coreference resolution.

Toshniwal et al. [2020] point out that long document coreference resolution is challenging because of the large memory and runtime requirements as other recent works require to keep all entities in memory at the same time. Toshniwal et al. [2020] propose a memory-augmented neural network that keeps in memory only a small number of entities at the same time. It guarantees a linear runtime in length of document and it outperforms rule-based systems.

We used the model trained on the OntoNotes dataset [2]. This model is described in Toshniwal et al. [2021], which is an extension of the work Toshniwal et al. [2020].

This model uses the `SpaCy` small English language model for tokenization, the same one as we use in the other parts of the work. Therefore we know that the tokens on the output of the coreference resolution model corresponds to our tokens as the same tokenizer was used.

Processing the Coreference Clusters

We tried two different approaches for assigning coreference clusters to individual characters. We describe both of them.

1. **Assigning to the most prominent character**

   For each mention in the coreference cluster we determine if this mention refers to some character by proper name, which we know from the first stage. We count how many mentions each character has in the cluster. If at least one character has at least one mention in this cluster, we assume this cluster corresponds to a person. Then we assign all not-yet-assigned mentions in this cluster to the character who has the most mentions by proper name in this cluster. If it happens that two characters have the same number of mentions, then it is decided arbitrarily (assigned to one of them non-deterministically).

[1] `https://github.com/shtoshni/fast-coref`
[2] `https://catalog.ldc.upenn.edu/LDC2013T19`

The mentions that were already assigned to some character are not changed. We are looking for assigning only not yet assigned mentions. This is because in the first stage we assigned only mentions that could be assigned uniquely only to one character. Therefore, we assume that if in the cluster there are mentions of different characters, then it is probably an error of the coreference resolution model.

We can show an example:

A coreference cluster contains these mentions: *John, he, John, his, he, Jody*

We know that *he, his, he* are not-yet-assigned mentions, both *John* mentions are assigned to the character named John and *Jody* is assigned to the character named Jody

Using this approach we assign *he, his, he* to the character John, because he had two mentions in this cluster and Jody had only one.

In the ideal case, all of the coreference clusters would contain mentions assigned only to one or zero characters.

This is the most simple approach - just assign to the character that was mentioned the most times in this cluster.

2. **Assigning according to gender**

   We decided to try another approach that considers the gender of the character. Looking at the output of the coreference resolution model we observed that sometimes it happens that tokens that clearly refer to the characters with different genders, like "he" and "she" end up inside the same cluster. Therefore, we decided to propose a method that would try to correct this type of error in the output of the model and see if it made any difference.

   We know genders for some of the characters from the first part of our system. We also know the genders of some basic pronouns like he or she.

   The approach is similar to the one above, but it takes into account the gender, if known. First, we determine if the mention is referring to some character by proper name for each mention in the cluster. We count how many mentions each character has in a cluster, but we divide characters according to gender to 4 categories - female, male, plural and unknown. We also count how many pronouns for the female, male and plural gender are in the cluster.

   If there were no mentions of known characters, then we don't assign mentions in this cluster to any character.

   If there was a mention of some characters, we determine the most probable gender for this cluster by counts of gendered pronouns and mentions of known characters with known gender. The most probable gender is the one that was detected in the most mentions from this cluster.

   Then, we try to determine for each not yet assigned mention to which character it refers to. We determine a gender of given mention either by

gendered pronoun or gendered first name that occurred in a mention (otherwise it is unknown). We detect the gender for the first name in the same way as in the previous chapter. We use a list of pronouns that refer to one of the genders.

We assign *best female character* found in the cluster to a female mention, *best male character* found in the cluster to a male mention, *best plural character* found in the cluster to a plural mention.

When the gender of mention is unknown, but there was a name of a character in the mention we assign it the *best unknown character* found in the cluster.

Otherwise we assign it *best gender character* found in the cluster.

We defined *best overall, female, male, plural, unknown* and *gender character* like this:

- **Best overall character**: It is the character that has the most mentions from the cluster assigned. It is the same as the most prominent character in the previous approach.
- **Best unknown character**: It is the character that has the most mentions from the cluster assigned from the characters with the unknown gender. If none of the characters that is mentioned in this cluster has an unknown gender, then it is the best overall character.
- **Best female character**: It is the character that has the most mentions from the cluster assigned from the characters with the female gender. If none of the characters that is mentioned in this cluster has female gender, then it is the best unknown character.
- **Best male character**: It is the character that has the most mentions from the cluster assigned from the characters with the male gender. If none of the characters that is mentioned in this cluster has male gender, then it is the best unknown character.
- **Best plural character**: It is the character that has the most mentions from the cluster assigned from the characters with the plural gender. If none of the characters that is mentioned in this cluster has plural gender, then it is the best unknown character.
- **Best gender character**: When the most probable gender is female, male or plural then the best female, male or plural character is accordingly assigned to this character. If the most probable gender is none of these (because there were no mentions that would tell us anything about the gender), then it is the best unknown character.

Why did we define it like this?

We decided to define it like this in the hope that if there are some gendered pronouns, then they would end up assigned to the most prominent character with the same gender. Best female/male/plural character should be the character with the given gender that occurred in the most mentions in the cluster. But when no character with the given gender was mentioned,

then we assign it the best unknown character. The best unknown character should be the character with unknown gender that occurred the most times, but when no character with unknown gender was mentioned, then it is the best overall character (character mentioned most times from all the characters). So when we find some gendered pronoun like she, we assign it to the best female character. That will be (in given order) either the character that has a female gender and occurred the most times from the characters with the female gender, or the character with the unknown gender that occurred the most times from the characters with the unknown gender or when no character with female or unknown gender was mentioned, than it is just the most probable character from all the characters. We think that this makes sense in a sense that we try to assign it according to gender, so the gender would not clash and prefer known gender before unknown, but in the worst case we assign it to the most prominent gender as a fallback, so all mentions would be assigned to some character. We think that it should not perform worse than the previous approach if we would have the correct genders assigned to the characters.

For the mentions with unknown gender, we look if the character name is in the mention and in that case assign it to the best character with unknown gender. But if the character name is not in the mention we assign it the best gender character as we assume that it is the most probable.

The third stage is the same for the coreference and without the coreference approach (in non-coreference approach, the second stage is omitted).

In the third stage, we assign a character to all yet not assigned mentions by proper name, that we stored for later in the first stage.

We do this by following rules in following order:

- If the character with that name is mentioned in the X tokens before this mention, we assign a mention to the closest character with this name mentioned before.
- If the character with that name is mentioned in the X tokens after this mention, we assign a mention to the closest character with this name mentioned after.
- We assign it to the character with the most occurrences from characters with a given name.

The number X is a parameter, we use 200 as default value, but it can be changed.

The first two rules follow the idea that if the character with the given name was mentioned close to this unresolved mention, it is likely that this ambiguous mention belongs to the same character. For example, the character is mentioned by his full name in the one sentence and in the following sentence, he is mentioned only by his first name because it is clear from the context that it talks about the same character. Even if his first name can be ambiguous in the terms of the whole text (another character with the same first name occurs in another part of the text).

In the work of Elsner [2012] the same idea was used, but they did not look at the X tokens before and after the mention, but at tokens inside the same paragraph. We decided to change it, because in our work we do not work with dividing text into paragraphs.

The third rule was also used in the work of Coll Ardanuy and Sporleder [2014]. The ambiguous name is matched to the most relevant character (with the most occurrences) from the characters with the given name. This results from the idea that it is the most probable that it refers to the character mentioned the most times. We think that usually in the book there are not two main characters with the same name, as it would be confusing. Usually when two characters have the same name, only one of them is the main character and the other is some minor character. So the assumption that ambiguous mention most likely refers to the more relevant character is probably valid.

In this stage all mentions by a proper name should have a character assigned.

The fourth stage is done only when coreference is used. It consists of assigning mentions in clusters that weren't assigned in the second stage because no character for that cluster was known at that time. But can be assigned after the third stage because they contained some mention that was assigned in the third stage. This stage is the same as the second, but it runs only on the not yet resolved clusters that were stored in the second stage for later.

After all these stages we have a list of occurrences for each of the characters.

3. Interactions of Characters

The goal of this part is to detect places in the text where two or more characters clearly interact together. But what is an interaction? That is a hard question and depends on our point of view. We will probably all agree that when characters are speaking together, then there is an interaction between them. But dialogues are not the only places that describe some sort of interaction. When characters are doing something together at the same place at the same time, we can also consider it interaction. For example, when they are dancing together, playing a game, or just walking together somewhere.

Sometimes it is really hard to decide what is a boundary between an interaction and not an interaction. For example, when one character is thinking about what he was doing yesterday with some other character, does that sentence describe an interaction or not? And what about a sentence that describes that two characters are at the same place, but each of them is doing something else?

We can show an example of this type of sentence:

"Mary was reading a book and John was writing a letter in the living room, when Adam walked in."

3.1 Definition of Interaction in Other Works

First, we describe how an interaction is defined in the other works.

The exhaustive survey about the extraction of the characters and the interactions between them, and creating character networks was made by Labatut and Bost [2019]. It describes different approaches that were used in previous works for all of these problems.

According to Labatut and Bost [2019], there are five distinct approaches for detecting interaction depending on how we define the interaction. We briefly mention each of them.

1. **Co-occurrence**

   The interaction between characters is detected when they occur in the same narrative unit. The narrative unit can be a certain number of sentences, paragraphs, pages, chapters, or something more sophisticated, like a portion of the text that is happening at the same location.

   The first step of this approach is to define what a narrative unit is and divide the text to these narrative units. Then the interaction between two characters is detected when they are both mentioned in the same narrative unit.

   This approach has clear downsides. As we mentioned earlier, when two characters are mentioned in the same sentence, it does not always mean that they are interacting. It can be that one thinks about the other.

2. **Conversations**

   In this approach, the interaction between characters is detected when one character explicitly talks to the other. It focuses only on dialogues, and for

each utterance of the dialogue, it determines which character is talking to which character. The downside of this approach is that it does not look in any way at parts of the book without dialogue. So it would perform really badly on books where there is little to no dialogue and interactions are mainly described by the narrator.

3. **Mentions**

   This approach is similar to the one above in the sense that it only looks at dialogues. But in this case, the interaction is detected only when one character talks about the other character or explicitly mentions the other character in his utterance.

4. **Direct Actions**

   This approach looks at all types of action that occur between characters, not just verbal interactions. The interaction is detected when two characters perform some action together or when one character performs some action on the other. The main problem is that we need to define what an action is, and we must be able to detect these kinds of actions in text. Some authors focus only on certain semantic classes of actions.

   This approach works a lot better than the Conversations approach for the texts with a small number of dialogues.

5. **Affiliations**

   Actions and verbal interactions are not important for this approach. The interaction between characters is detected when there is some sort of a relationship between them described in the text. For example, they are family, friends, or belong to the same social group.

3.2 Our Definition

We will use a combination of Conversations and Direct Actions approaches. We will try to detect which characters were speaking together for each dialogue. Then, we will try to detect direct actions between characters in the sentences outside of the dialogues. However, we define direct action in our own way.

Direct action between two or more characters is a situation in which these characters are doing something together or are in the same place at the same time, and at least one character knows about the other (e.g. sees him or hears him) and this "knowing" is described. This action is supposed to be happening in real time from the perspective of the story. That means that it is not just a description of something that happened before the main storyline or it's not just a memory of the past of some character.

We define the *interaction* between two or more characters as a situation in which these characters have a conversation or there is some other direct action happening between them.

We know that this definition can sound a little ambiguous. There probably are sentences about which one person would think there is an interaction based on our definition and the other would think there is not. But we think it is really

hard to define it unambiguously without listing all specific situations we consider an interaction in the definition.

Why did we decide to define it like this? We think dialogues are a clear sign of an interaction but not the only one. Sometimes there is a long section without dialogues in the book, but there is clearly an interaction happening between characters. For example, the author describes characters having some sort of physical contact together. We decided to restrict only to situations happening in the main storyline. We think that descriptions of what happened in the past or what some character is thinking are not what we want to consider when we look at how characters interact throughout the story.

We think that it is a challenging task to try to find interactions in the text based on our definition that wasn't really examined in the previous works.

3.3 Background

We describe what the Part-of-speech tags and dependency parsing is as it is used in the following part of this chapter.

3.3.1 Part-of-speech Tags

A *part-of-speech tag* (or POS tag) is a label assigned to each token in the text that indicates its' part of speech. A part of speech is a category of words that have similar grammatical properties. The examples of the POS tags are: noun, pronoun, adjective, verb or adverb.

We use the POS tags that are given to the tokens by the `SpaCy` model[1]. The POS tags in the `SpaCy` are the Universal POS tags[2].

In our work, we only look at whether the POS tag is a verb.

3.3.2 Dependency Parsing

Dependency parsing is the process of analyzing the grammatical structure of a sentence and examining the dependencies between words in it. It is based on the dependency grammar.

As described in Nivre [2010], the dependency structure of a sentence represents head-dependent relations between tokens. That means that each token in a sentence has exactly one of the other tokens from the sentence assigned as his head (besides the one token that is the root). Therefore the syntactic relations of a sentence form a tree structure. This tree is called a *dependency tree*. Each token has also a label assigned that describes his dependency on the head word. These labels can be, for example, an object or a subject.

In our work, we use a dependency parser from the `SpaCy` library[3]. The `SpaCy` library offers us a simple way to navigate the dependency tree. We can access the head token of the token, the children of the token (the tokens which are the immediate syntactic dependents of the token) or all tokens in the subtree below the token (the children of the token and children of the children and so on).

[1] `https://spacy.io/usage/linguistic-features#pos-tagging`

[2] `https://universaldependencies.org/u/pos/`

[3] `https://spacy.io/usage/linguistic-features#dependency-parse`

The labels for the dependency relations that are mentioned in this chapter are *nsubj*, which is a nominal subject and *obj*, which is an object. The list of all basic dependency relations for English with the description can be found in Universal Dependencies[4].

3.4 Previous Works

3.4.1 Quote Attribution

Quote attribution (assigning speakers to quotes in the text) was examined in many previous works. The approaches differ a lot, from rule-based systems to the ones using machine learning and neural networks. We describe some of the simpler approaches that inspired our work.

A Two-stage Sieve Approach

Muzny et al. [2017] proposed a deterministic sieve-based system. They divide this task into two stages: the first one assigns quotes to mentions in the text and the second one assigns mentions to characters. We only describe the first stage, as we dealt with linking mentions to characters in the previous chapter.

The linking of quotes to mentions is done by a series of these deterministic sieves:

- Trigram matching on these patterns: Quote-Mention-Verb, Quote-Verb-Mention, Mention-Verb-Quote, and Verb-Mention-Quote, where the mention is either a character name or a pronoun.
- Extraction of all verbs and their dependent nsubj nodes from the sentences surrounding the target quote. If the verb is a common speech verb from the list and nsubj node contains a character name, pronoun or an animate noun from the list, then the target quote is assigned to the mention in nsubj node.
- If there is only one mention in the paragraph of the quote in the text outside the quote, then link the quote to that mention.
- If there is a vocative in a preceding quote, then link the target quote to that vocative. They created a list of vocative patterns they use.
- If the target quote is at the end of the paragraph, then it is linked to the final mention that occurred in the preceding sentence.
- The last two sieves assign non-assigned quotes according to a conversational pattern – if the speaker of a quote in paragraph number n is known, then this speaker is assigned to the quote in a paragraph number $n + 2$. This is done because it is assumed that in a conversation two characters alternate in speaking in paragraphs if none character was mentioned in this paragraph explicitly.

[4] `https://universaldependencies.org/en/dep/`

These sieves are executed in the following order and when one sieve assigns a mention to some quote, then this is not changed by the other sieves. The sieves are applied only on not yet assigned quotes.

A Salience-based Technique

Glass and Bangay [2007] proposed a salience-based technique for quote attribution. This method does not use any machine learning or knowledge-based techniques. It consists of three phases: locating the speech verb for the quote, locating the mention that is associated with that speech verb and selecting the speaker from the list of characters that are participating in that scene. The fact that this method is salience-based means that we create a list of candidates in each phase (for example, a list of potential speech verbs) and a salience value is assigned to each of the candidates. Then we choose the candidate with the greatest salience.

We look at the first and second phase and not at the third phase as we already have mentions linked to characters.

In the first phase, the list of all candidates for speech verbs is created from the verbs in sentences surrounding the quote. A number of features, each with its own salience value, are used to decide which verb is the speech verb. The features are:

- Verb is marked as the main verb (root) in a sentence by a dependency parser.
- Verb has one of the verbs: “communicate“, “verbalise“ or “breathe“ as an ancestor in a hierarchical lexical tree. This information is provided by Wordnet [Fellbaum, 1998].
- Verb is within the adjacent sentence.
- Verbs are given a salience based on the proximity to the quote(verbs nearer quote have larger salience).

All features except the last have a salience score of 1 point, the last one is a value between 0 and 1 based on the distance.

The verb with the highest score is chosen as the speech verb for the given quote.

In the second phase, the Actor (mention that is believed to be an actor for that speech verb found in the first phase) is identified. The method is based on the idea that there is a dependency identified by the dependency parser between the speech verb and the Actor. But the authors point out that sometimes there are more verbs in a sentence that create a verb chain (number of verbs linked to the one main verb).

We can look at an example of the verb chain that Glass and Bangay [2007] mention. The following sentence from the Phantom of the Opera by Gaston LeRoux is used as an example:

“She shuddered when she heard little James speak of the ghost, called her a ”silly little fool” and then, as she was the first to believe in ghosts in general, and the Opera ghost in particular, at once asked for details: ”Have you seen him?” “

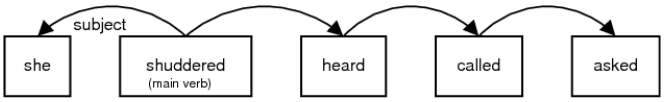

Figure 3.1: Visualisation of verb chain produced by the dependency parser from the work of Glass and Bangay [2007]

We can see the dependencies between verbs and a subject in the given sentence in fig. 3.1. The verb "asked" was considered to be the speech verb and the goal was to detect the word "she" as an actor. But we can see that the word "she" is dependent only on the main verb (the head of the verb chain) and not on the speech verb.

The actor is mostly the subject or object of the main verb that is not always the speech verb. Sometimes, the speech verb is somewhere in the verb chain and the actor is not dependent on that speech verb but only on the main verb.

Therefore, the algorithm traverses up the verb chain until the main verb is found. Then the children of the main verb are candidates for the Actor. It looks at whether the children contain subject or object relation and are not inside the quote.

But the errors in dependency parsing of complex sentences are frequent, so a salience-based technique is used to identify the Actor. All tokens that can be reached by traversing in both directions from the speech verb are candidates for the Actor. The features used are:

- The token is the subject or the object of the main verb.
- The token is a noun that is a descendant of the word person in WordNet [Fellbaum, 1998] or is not a recognized English noun, which means that it can be a name, or the token is a pronoun.
- The token is capitalized and it is not the first token in a sentence. If it is the first token in a sentence, then it must not be a recognized English noun. Tokens marked as prepositional-complements are excluded and not given a salience.
- Abbreviations and personal titles, as "Mr.", are not awarded a salience.
- Salience is awarded based on the distance to the speech verb (the closer is better).

All features except the last one are given 1 point. The last one increments the score of the candidate by multiplying the existing score by 0.1% for each word between the speech verb and the candidate token.

The token with the highest score is chosen for the Actor.

3.5 Data

We used the LitBank dataset[5] for creating rules in this part of the work. It consists of a small portion of text (approximately 2000 words) of 100 different books

[5] `https://github.com/dbamman/litbank`

drawn from the public domain texts on the Project Gutenberg[6]. It contains books of different genres and styles. The total dataset contains 210,532 tokens. This dataset is annotated – it has entity annotations, event annotations, coreference annotations and quotation annotations.

This dataset and the entity annotations were described in Bamman et al. [2019b] and the coreference annotations were described in Bamman et al. [2019a].

3.6 Our Approach

We suppose that we have a tokenized text and the coreference clusters of all mentions (even anaphoric mentions like pronouns) for all characters in the text. This is the output of the previous part of our program, where we tried to find all characters with all their mentions. We take it as an input for this part. In this part we only work with this input and we are not trying to correct it in any way. This part of the system can be also used alone, when we have a gold data list of the characters and the coreference clusters for these characters.

The output of this part should be the list of the places (indices of sentences) where two or more characters interacted together with the characters that interacted together in the corresponding sentences.

3.6.1 Dialogues

To begin with, we describe how we define dialogue in our work.

We assume that all direct speech in the text is within the quotation marks. We call an *utterance* a sentence that contains a direct speech (contains some text in quotation marks). For simplicity we ignore the fact that sometimes there is a text inside the quotation marks, even if it is not a direct speech, as it does not happen very often.

We decided that a sentence is part of a *dialogue* if it is an utterance or it is a sentence right before or right after an utterance (her distance from the closest utterance is 1). The sentences that are part of the same dialogue are considered to be a dialogue. So the dialogue begins with one sentence before the first utterance, then it contains some number of utterances, and between two utterances is zero or one sentence that is not an utterance. The dialogue ends with one sentence after the last utterance.

We can find dialogues easily with this definition; we firstly find all utterances and then combine them into dialogues.

When we find all the dialogues, we must detect which characters are talking together in each dialogue. We don't really need to know exactly which utterance belongs to which character. It is enough to know which characters were part of that dialogue.

He et al. [2013] states that the speaker alternation pattern is often used in dialogues between two characters. The speakers are usually identified explicitly (by their name or pronoun referring to them) in the beginning of the dialogue and then they take turns in following utterances. That means that odd-numbered utterances belong to the first character and the even-numbered utterances belong

[6] `https://www.gutenberg.org/`

to the second character. The reader knows who speaks with whom from the context. If this pattern is violated (e.g. one of the speakers misses their turn or a new speaker is introduced), a clue is provided in the text (e.g. explicit mention of the new character).

In our work, we don't need to care about who said which utterance in a dialogue. We just try to find explicit mentions of the characters in the utterances, sentences close to the utterances or as vocatives inside the direct speech. Then we assume that these characters were present in the whole dialogue and that no other unmentioned speakers were there. This assumption is not a hundred percent true, but it works in most of the cases. It is unusual that the character is part of the dialogue, but is not mentioned as a speaker in at least one of the utterances or as a vocative in a direct speech. The previous works described before also work with this assumption, but the whole algorithm was more complicated because they tried to assign all the utterances to one of the characters.

We assign a whole dialogue to the found set of characters mentioned in that dialogue and we ignore the fact that sometimes one speaker leaves in the middle of the dialogue or joins in the middle of the dialogue. We assume that usually in these cases there are at least two sentences without direct speech between them, and so the dialogue is divided into two dialogues in our definition of the dialogue.

Now, we describe our algorithm for detecting characters that were part of a dialogue.

We created a list of speech verbs manually starting with the lists defined in existing works [Muzny et al., 2017] and adding new words that were synonyms or that we found in our data.

The speech verbs are the following: *say, cry, reply, add, think, observe, call, answer, encourage, announce, tell, warn, ask, talk, continue, gasp, sigh, mumble, shout, thank, explain, report, mutter, interrupt, yell, scream, wail, hiss, whisper, whimper, rant, stutter, ramble, groan, grumble, moan, murmur, stammer, beg.*

We detect that character is a part of a dialogue by following rules:

1. If a sentence contains a direct speech and there is a verb outside the direct speech in this sentence and a mention of a character is in the subtree under the main verb of the verb chain for this verb, and the character mention does not have the word "about" in the syntactic descendants, then this character is considered to be a part of the dialogue.

2. If a sentence does not contain a direct speech, but it contains a speech verb from our list and a mention of a character is in the subtree under the main verb of the verb chain for this verb, and the character mention does not have the word "about" in the syntactic descendants, then this character is considered to be a part of the dialogue.

3. If a mention of a character is in a place of vocative in the direct speech, we consider this character to be a part of the dialogue. We use 9 vocative patterns for finding the vocatives that are used in the work of Muzny et al. [2017]. [7]

[7] We added one new vocative pattern to 9 patterns from Muzny et al. [2017] that we observed in our data: a mention between " and !

The first two rules were inspired by the work of Glass and Bangay [2007] described above. We used the idea that when we want to find a speaker we look at all the descendants of the speech verb or when the speech verb is in the verb chain, then of the main verb of this chain.

However, we simplified it a little bit, as we do not only want to find a speaker but also the listeners and all the people that were part of the dialogue. For example, in the sentence: *"John said to Mary: "Hi!"*", we ideally want to detect both John and Mary. And in this case both John and Mary will be the syntactic descendants of the verb say.

Because of this, we look at all descendants of the verb and don't look at their dependency tag.

All characters that were detected by these three rules are considered to have interaction in the places of the detected dialogue.

We also decided to treat all verbs that are in an utterance but not inside quotes as a speech verb as it should work like trigram matching in the work of Muzny et al. [2017].

In the sentences without direct speech, we only work with verbs that are in our list of speech verbs, as the sentences surrounding the utterances don't have to always describe something that is connected only with the characters that were part of the dialogue.

We added the rule, that the mention of a character can't have the word "about" in the syntactic descendant to be considered part of the dialogue. This was added because we think that when there is in a sentence the word about before the mention, then the character does not need to be in the same place.

For example: *"John was speaking about Mary."*

It oftens describes that someone was talking about someone else, but the person that they were talking about is often not part of the dialogue.

3.6.2 Direct Actions

The second part consists of finding direct actions between characters outside of the dialogues. It is a hard task to do automatically because sentences without the context are often ambiguous. We decided to only look at sentences, in which two or more characters are mentioned. We do not try to detect an action that is described in more sentences and none of the sentences mentions more than one character because that would be an even more complex task. When we find a sentence that mentions more characters, we try to decide whether that sentence indicates direct action between these characters or not. We can show some examples of simple sentences, where two people are interacting:

"John walked with Mary to the cinema."
"John and Mary played the game together."

And when they are not interacting:

"John was thinking about Mary."
"John and Mary are doctors."

We can see that the fact that two characters are mentioned in the same sentence doesn't mean they are interacting. And sometimes we need to know further

context to decide. In our work, we tried to find places where it is clear that people are interacting.

We put together a list of "action" verbs. Action verbs are verbs that describe some direct action and when two character's mentions are syntactically dependent on that verb, then there can be an interaction happening between these characters.

Examples of these verbs are: *go, meet, touch, climb, walk, look, warn, gaze, sit, run, hit, play, dance, hold, follow, approach, laugh, kick, listen, dine, hug, kiss, eat, drink.*

Whole list of these verbs is in the appendix A.1.

We created this list by analyzing sentences in which two or more characters were mentioned. Firstly, we generated all sentences with mentions of more characters from our dataset (described in section 3.5), which has annotated mentions of characters. Then we looked at these sentences and we tried to decide whether there was an interaction between them for each of the sentences. We tried to compare the sentences that described the interaction and that not from the point of a syntactic structure and words they contained. We tried to observe some rules that could help us decide. We also tried to find out if there are verbs that repeat in the sentences with interaction and don't appear in a sentences without interaction or when they appear in a sentence without interaction, there is a simple rule that could rule out this sentence from the others with interaction (of course, just in our sample of sentences).

We found out that there are verbs that repeat a lot in the sentences with interaction, so we put them in the list of action verbs also with some of their synonyms.

However, we also noticed that all of these action verbs can be used in a sentence that does not describe direct action, as we defined it. Sometimes the sentence describes something from the past, something from the future or represents just something the character dreams of. Sometimes, the sentence describes something that is happening often, but not right now. We show some examples of these types of sentences:

- *John and Mary often watched TV together.*
- *It was not possible for Jane to meet Mr. Brown.*
- *How much more he told her as to his occupation it was impossible for Winnie's mother to discover.*
- *We will go there and meet David.*
- *She couldn't even imagine what she would do if she saw him.*

Therefore, we come to the conclusion that we must also look at how the sentence looks, what grammar tense is used, or what other words are in the sentence.

From our analysis of many sentences from our data, in which these verbs appeared, we created the following rules for deciding whether the sentence describes interaction or not.

Rules

1. If a sentence contains an action verb from our list and two or more mentions of characters are in the subtree under the main verb of the verb chain for this verb (we don't count character mentions that have the word "about" in the syntactic descendants to these mentions), then this sentence is a candidate for interaction. But this sentence can be ruled out by some rule below.

2. If the word before the action verb or the head token of the action verb is one of the following:

   "had", "'d", "will", "'ll", "would", "not", "n't", "could", "might", "should",

   then we rule out this sentence. It is because these words in most cases indicate that something happened in the past, will or can happen in the future or does not happen.

3. If the two words before the action verb are "going to", then we rule out the sentence, because it probably describes something that is just going to happen in the future.

4. If the subtree under the main verb of the verb chain for this action verb contains the word or subsequent words from this list:

   "once", "always", "never", "usually", "often", "seldom", "now and then", "any time", "every time", "sometimes", "ever", "could", "might", "if", "should", "since", "until",

   then we rule out the sentence. It is because these words probably indicate that the action is not happening right now, but happened in the past or is happening repeatedly (but not right now), or does not happen, or can happen in the future.

The sentences that were not ruled out, are detected as describing the interaction between the mentioned characters (besides the character with the word about in his syntactic descendants).

The examples of sentences from our data, that were detected by these rules as an interaction:

- *Words passed between Clayton and the captain, the former making it plain that he was disgusted with the brutality displayed toward the crew, nor would he countenance anything further of the kind while he and Lady Greystoke remained passengers.*

- *She glanced toward Jock in the hallway, then lowered her voice.*

- *While Tom was eating his supper, and stealing sugar as opportunity offered, Aunt Polly asked him questions that were full of guile, and very deep – for she wanted to trap him into damaging revealments.*

- *But just that instant the officer turned to leave Lord and Lady Greystoke, and, as he did so, tripped against the sailor and sprawled headlong upon the deck, overturning the water-pail so that he was drenched in its dirty contents.*

- *We know only that on a bright May morning in 1888, John, Lord Greystoke, and Lady Alice sailed from Dover on their way to Africa.*

It is clear that this rule-based system will not work in every situation. There are many exceptions that we don't cover and probably all of the rules are not right in every situation. These rules are also hard to evaluate, because there is no annotated data for this task. Also, some sentences are ambiguous without further context (and sometimes even within context) so that we cannot really tell if the sentence should be detected as an interaction. Therefore, we evaluated our system during training (creating and testing rules) and during testing (trying these rules on unseen texts without further improving these rules) only by hand.

4. Graphs

We created interactive graphs for visualizing the results of our work. We use two types of graphs. One for the occurrences of characters (an output of the chapter 2) and one for the interactions between characters (an output of the chapter 3).

4.1 Occurrences Graph

The first type of graph shows the occurrences of individual characters throughout the book. On the x axis, there are the numbers that correspond to the individual tokens. For example, if the input text has 2 000 tokens, then the x axis represents the numbers from 0 to 1 999, which are the indices of the corresponding tokens. On the y axis, there are individual characters. Every character has a different position on the y axis. We can choose what is the minimum number of occurrences for the character to be shown in the graph, because we probably don't want too many characters shown (mainly the ones that occurred only a few times).

There is a point on the graph in the (x, y) coordinates, when the character in the y position has an occurrence detected for the token number x. We represent the occurrences of the character as a list of indices of tokens that correspond to the mention of the given character. Therefore, it is easy to create this graph when we have this information.

When you click on the point on the graph, the sentence with the corresponding token is displayed and the token that corresponds to the x coordinate is highlighted in the sentence. More about this is in the section 7.3.2.

4.2 Interactions Graph

The second type of graph shows interactions between characters throughout the book. On the x axis, there are numbers that correspond to the sentences in the text. In this graph, we work with sentences instead of tokens because we detect interactions for the whole sentences. For example, if the input text has 500 sentences, then the x axis represents the numbers from 0 to 499, which are the indices of the corresponding sentences. On the y axis, there are individual characters the same way as in the occurrences graph. We can also choose the minimum number of occurrences in interactions (number of sentences in which it is detected that the given character interacted with another character) for the character to be shown in the graph.

There is a point in the coordinates (x, y) if the character in the position y has an interaction detected in the sentence number x. We represent the interactions for each character as the list of indices of sentences which were assigned to the given character as an interaction. The interaction for the given sentence is always assigned to at least two characters. So when we want to find which characters interacted together at a certain time in the book, we look at the vertical slice of the graph. If some of the characters have the point on the same coordinate x, then they interact in the sentence number x.

When you click on the point on the graph, the sentence with the corresponding number is displayed. More about this is given in the section 7.3.2.

4.3 Examples of the Graphs

We show an example output of our program for the short samples of three novels. Both types of graphs are created on the same sample for each novel. These samples are in the electronic attachment to this thesis. Position in the text is measured by number of tokens in the occurrences graph and by number of sentences in the interactions graph. That is why the numbers of positions are different for the occurrences and interactions graph for the same sample.

In fig. 4.1 and fig. 4.2, we can see the occurrences and the interactions graph for the sample from Wuthering Heights.

In fig. 4.3 and fig. 4.4, we can see the occurrences and the interactions graph for the sample from The Little Women. On the occurrences graph, there are only characters that occurred at least 10 times in the sample, because there were too many characters.

In fig. 4.5 and fig. 4.6, we can see the occurrences and the interactions graph for the sample from The Scarlett Letter. On the occurrences graph, there are only characters that occurred at least 10 times in the sample.

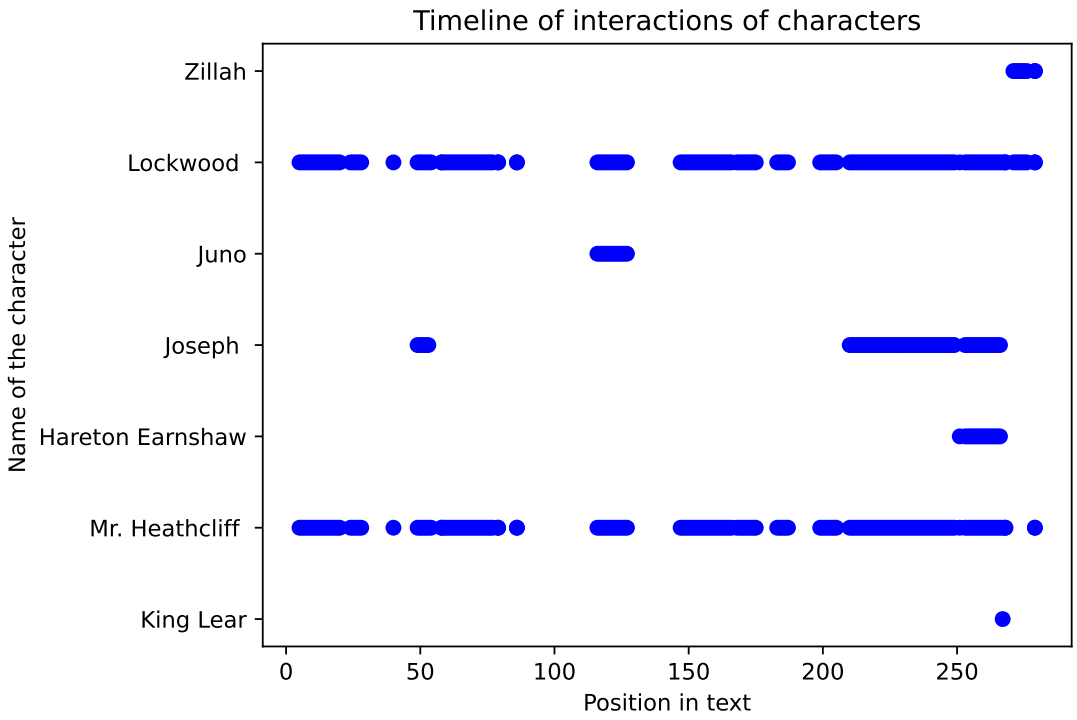

Figure 4.2: The interactions graph for the sample from Wuthering Heights.

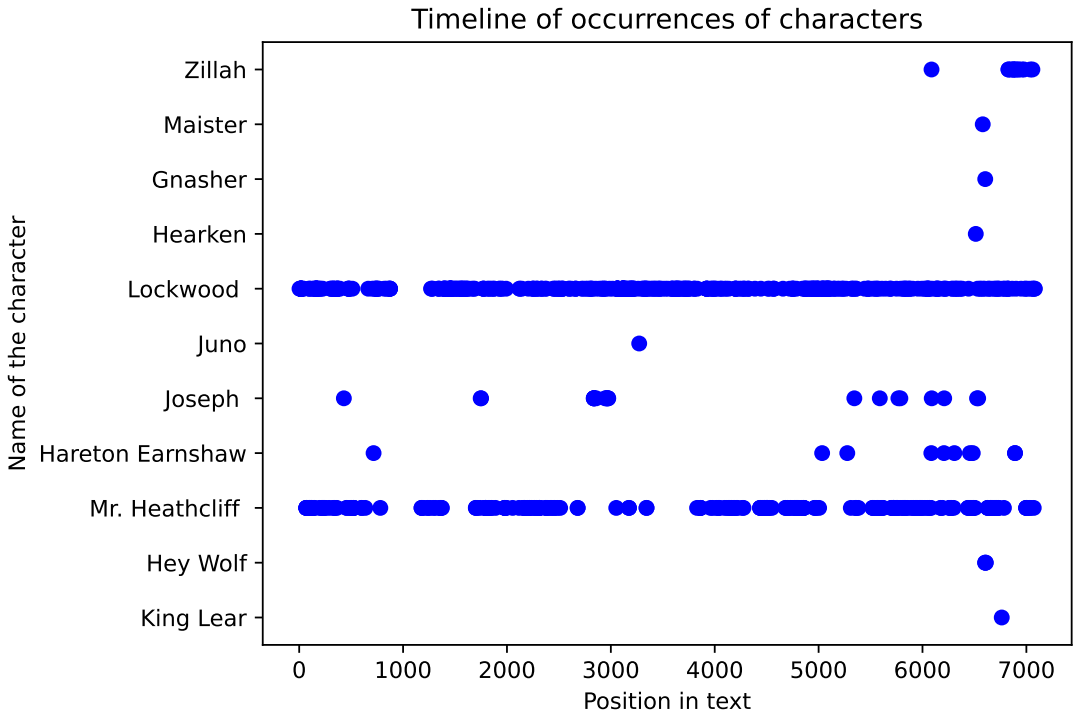

Figure 4.1: The occurrences graph for the sample from Wuthering Heights.

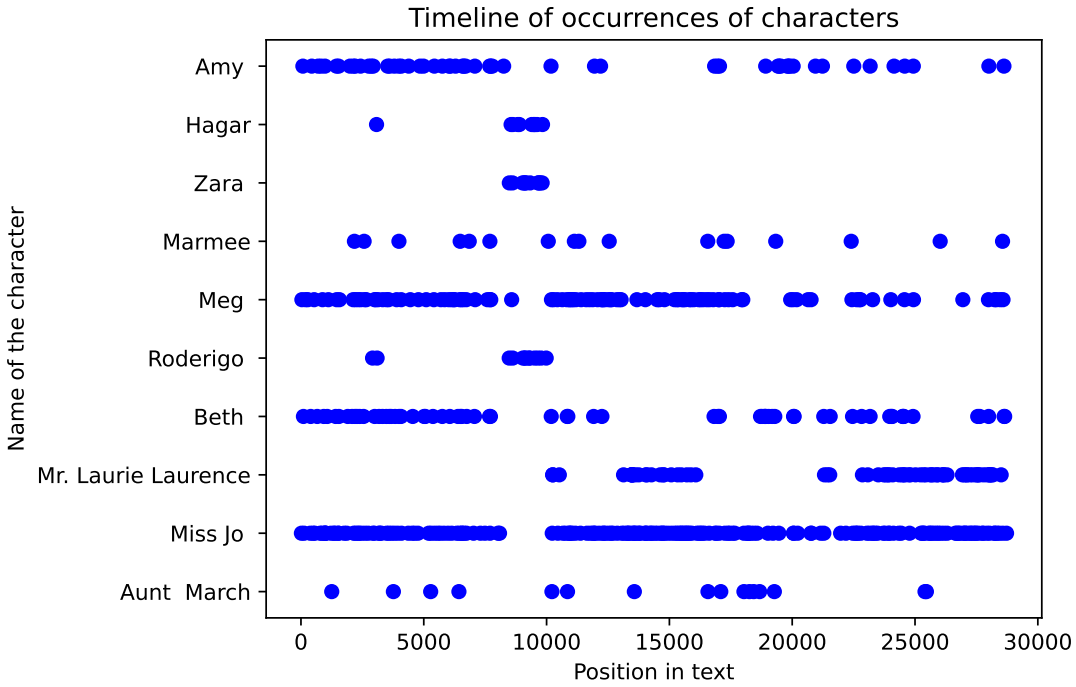

Figure 4.3: The occurrences graph for the sample from The Little Women.

Figure 4.4: The interactions graph for the sample from The Little Women.

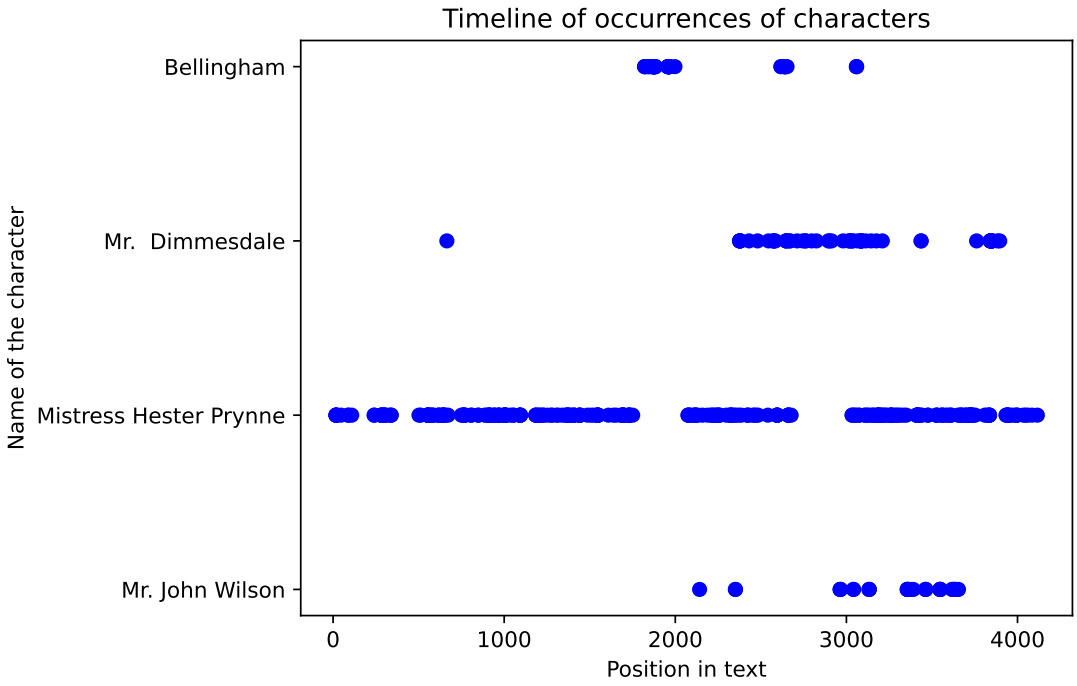

Figure 4.5: The occurrences graph for the sample from The Scarlett letter.

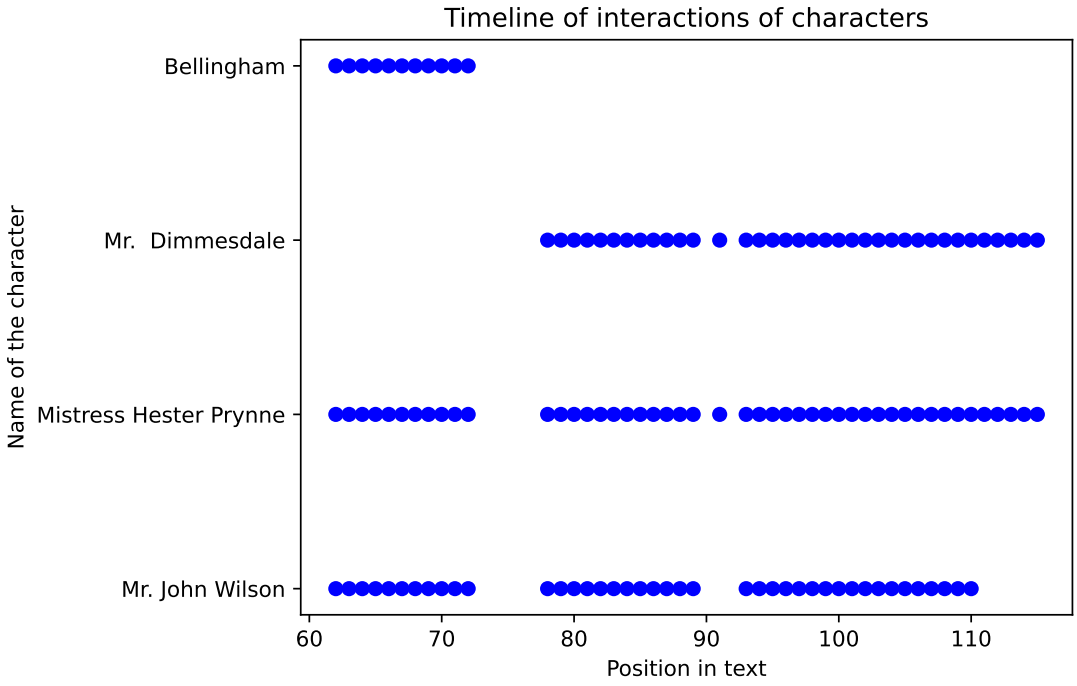

Figure 4.6: The interactions graph for the sample from The Scarlett letter.

5. Evaluation

5.1 List of Characters

5.1.1 Dataset

We evaluate detecting the characters on the dataset that consists of 30 novels listed on Sparknotes[1] with the corresponding lists of characters. These character lists contain mostly main or important characters. This dataset was used as a second dataset for evaluation in the work by Vala et al. [2015]. It was mentioned that it omits many minor characters, but it serves as a baseline of those characters that any method should be able to detect. We think that our work does not concentrate on finding the minor characters that aren't important for the plot and don't interact much with the other characters. We also do not try to detect unnamed characters in contrast with Vala et al. [2015]. Therefore, we think that it is sufficient to evaluate on this dataset for our purposes. We also didn't look at the novels in this dataset and didn't use them in any way during the implementation.

The 30 novels in this dataset are the public domain texts published on the Project Gutenberg[2]. The list of the names of these novels is in the appendix A.2.

We use the character lists that are in the attachment to the work of Vala et al. [2015] as our gold list of characters. We excluded "the Narrator", which was included in some of these lists, from the gold lists as we don't try to detect the narrator as a character in our work.

5.1.2 Getting Data for Evaluation

We run the first module of our work for finding names of the characters, merging different namings for one character and getting a list of character names as a result, as was described in chapter 1, on these novels. We run it two times; once using the `NameTag 2` and once using the `SpaCy` library as a NER tool. We save the full names of the characters found by our system for evaluation.

Then we assign all character names found by our system to zero or one character from the gold list of characters. First, we try to do it automatically.

We describe the assignment of one character generated by our system. We assign a name of the character to the character in the gold list with exactly the same name. If none of the characters in the gold list has the same full name, we split the name on whitespaces to more words, and we count how many words in the given character's name are the same as a word in a name of the character in the gold list for every character in the gold list. We assign the character to the gold character with the highest number of the same words in their names. If none of the characters from the gold list contain any of the words in the given character's name, we don't assign the character to any of the gold characters as it probably means that it is the name of a character that is missing in the gold data or the name is not even a character name, but some error. If two or more gold

[1] `sparknotes.com`

[2] `https://www.gutenberg.org/`

character names contain the same number of words from the given character's name, then we assign the character to one of them arbitrarily. It does not matter much, because we will look at the data after this automatic assignment.

After assigning all the characters, we looked at this automatic assignment and corrected the errors by hand. We decided for each of the names that were assigned to the gold character if it really is a name of the character or if this name cannot refer to some other character. For example, if an automatic assignment assigned "John Smith" to the gold data character named "John Moore", because they are both called John, we erased it, because we know this can't be the same character. But if the name was ambiguous, we assigned it to all the characters in the gold data that it may refer to. For example, if the character name was only "Smith" and in the gold data there were two characters with the last name "Smith", we assign this character to both of them, because we cannot decide which one it refers to.

Therefore, after this step, some of the character names can be assigned to more than one character, if they are ambiguous in the context of the gold data list. We also find character names that occurred in the novel with different spellings as in the gold data list. This happened for the books that were originally written in Russian, as there are probably different versions of the English translation of these names. We did this phase only by hand, so it is possible that we overlooked some errors.

After this phase, we have a list of character names generated by our system for every character in the gold list that may refer to this gold character. If the length of this list is zero, it means that our system did not find this character at all. If the length of this list is one, it means that our system found this character and only one of the found characters has a name that can refer to this character, so our system probably succeeded in merging different namings for this character into one (or the character was referred to only by one name throughout the whole novel). If the list has length bigger than one, it means that our system found the character, but more characters have the name that could potentially refer to this character, so probably some of the different namings for this character weren't merged together. It could also happen that some of these names are ambiguous and looking only on the gold list it seems that the name can refer to more characters, but in reality, there is another character with the given name that is not present in the gold list and the name refers to him.

These different namings can be merged or it is decided to which of the characters the ambiguous names refer to in the second part of our work – the module for finding occurrences chapter 2. It could happen that all of the occurrences of the names we assigned to more characters would be resolved to refer to only one of the characters, and some of the characters would have zero occurrences as a result. We can tell that the character with zero occurrences isn't a character at all, because he was merged with some other character during the second part of our program. But now we only focus on evaluating the first part of our program and don't consider this.

5.1.3 Evaluation

We decided to evaluate the accuracy of two different metrics.

The first one is how many of the characters from the gold data were found by our system percentage wise. That means how many of the gold characters have at least one character assigned divided by the number of gold data characters. We call this a *Found characters* metric.

The second one is how many of the characters from the gold data were found only once by our system percentage-wise. In this case, we only count how many of the gold characters have exactly one of the characters assigned divided by the number of the gold data characters. The accuracy of this metric will be lower or equal to the accuracy of the first metric. We call this a *One-to-one characters* metric.

We think that the first metric is more important because when some character was not found at all by our system, then it means he will not be found later, as we only work with the characters found by the first module. The low value for the second metric does not necessarily mean the system would work bad in the end, as some of the characters may be merged into one in the second phase by using the coreference resolution model as we mentioned above.

We evaluated it for all the novels in the dataset individually and then made an arithmetic average of the results. The results are shown in 5.1.

Metric	NameTag 2	SpaCy
Found characters	0.799	0.710
One-to-one characters	0.569	0.489

Table 5.1: Accuracy of character detection on SparkNotes dataset.

We see that `NameTag 2` results are slightly better than the `SpaCy` one in both metrics. We also observed when looking at the data that `NameTag 2` could sometimes find character names that `SpaCy` didn't find, but it can be just on this data.

The characters that weren't found at all were sometimes the unnamed characters or characters that were only described by a nominal, like "Aunt", that we do not detect. Some of the names were not detected by the NER and they were often unusual names like "Kiche" or "Mit-Sah" and that was probably why the NER did not find it. The accuracy measured by the first metric was pretty high for most of the books (higher than 0.75) and was equal to 1 for 7 books using the `NameTag 2`. But there were a few books with low scores that decreased the average. These books often had only a few characters so even not detecting 1 or 2 of them has made a big difference percentage wise.

The main problems in merging different names together, which we observed in the data, were the following:

- Our lists of honorifics do not contain some of the honorifics used in this dataset like *King* or *Queen*. It resulted in treating these words as first names and not the honorifics. Adding these words to the lists of honorifics would probably improve the result of the second metric a bit.

- Some of the characters are called in the text by 2 different honorifics in different parts of the book. For example, *Mr. Smith* and *Master Smith* refer to the same character. In our work, we only consider that a character can

have 0 or 1 honorific, and this results in creating two different characters, one called Mr. Smith and the other Master Smith.

- The NER tool sometimes tags as part of a character's name even the one word before the name, and if it is capitalized (because it is the beginning of the sentence), we treat it as a name in our work. The examples from the data are *Dear Katherine*, *Ah Sarah* or *Barefooted Friar*. This results in treating Dear as a first name and Katherine as the last name of one character. And if Katherine is called by her first and last name together, it is not merged with the mention *Dear Katherine*. It is treated as another character, even if it is the same person.

- Sometimes the NER tool tags more words that contain mentions of two characters as one character, for example: *Mr. Michelangelesque!"—little Bilham* or *Charles and Hal*. We do not correct this in any way and treat it as a new character.

- There are characters with a middle name in the dataset, mostly in the books that are originally written in Russian. For example, *Pyotr Petrovich Luzhin*. This decreased the accuracy of the second metric for these books, because the character was sometimes called Petrovitch Luzhin and other times as Pyotr Petrovitch. Our system didn't merge these two names because we treat Petrovitch as a first name in the first case and as a last name in the second case as we don't use a concept of middle names in our work.

- Guessing the genders of characters based on the first name does not work on the books in which the names of the characters aren't of English origin as our lists of gendered names contain only English names.

We decided not to evaluate how many of the characters found by our system were in the gold list, as many of the found characters were correctly found, but are not in the gold list, because they are only minor characters. But we observed in our lists of characters that the NER tool also tags some of the single capitalized words as a name, even if it is not a name. Examples from the data are: *"Ah", "Aha", "Shall", "Thou", "Hush", "Hah", "Ay", "Go".*

In the electronic attachment, there are the files we created during the evaluation. In the `characters` subdirectory of the `results` directory, there are the gold lists of characters (in `gold` subdirectory) and the `csv` files we generated and than corrected by hand (in `key_nametag` and `key_spacy` subdirectories). The `csv` file contains the name from the gold list in the first column and then the names assigned to the given gold character in the same row in other columns. In the last column, there is a number of assigned characters for the given gold character. In the last row, there is accuracy of the first and second metric for this novel.

5.2 Character's Occurrences

5.2.1 Dataset

We evaluate the second part of our system on the first fifteen books from the dataset of novels from Project Gutenberg as the first part (section 5.1.1). But we used only the sample consisting of the first few pages of each book (usually the first chapter or the first few chapters, if chapters were short) and not the whole novels. These texts are attached to the thesis in the data directory.

5.2.2 Evaluation

We want to compare the coreference clusters we obtain by using the first and the second methods to assign the clusters to the characters described in the chapter 2. The first method assigns a cluster to the character with the most mentions in the cluster. The second one also considers genders of characters and pronouns.

We run our tool on the samples from the first fifteen novels from the SparkNotes dataset two times; once using the first method and once using the second method for assigning mentions from coreference clusters to the characters. We save the output clusters and also mentions that were assigned to characters so we could look at them.

We found that for 10 of the 15 texts, the clusters were exactly the same. They differed slightly for 5 out of 15 texts, but the differences were not significant; it was just a few mentions assigned to different characters. In the sample from the book "The Scarlett letter", we observed that the character named Hester Prynne was assigned both the female and male pronoun mentions using the first method, even when the character is female. Some of the male pronoun mentions were assigned to another character using the second method.

There were also slight changes in the assignment of plural pronoun mentions, like "they", in these 5 texts.

We couldn't evaluate it more precisely as the texts aren't annotated for this task. But judging from only the small number of changes in only 5 of 15 samples, we think that the differences between using these two methods will be small. Therefore, using the second method is probably not worth it as it is more complicated and runs a little longer. But it would be useful to evaluate it on the longer annotated texts that are sampled from more books.

In the electronic attachment, there are the files we created during the evaluation. In the `occurrences` subdirectory of the `results` directory, there are the `json` files we generated and than compared (in `gendered` and `most_prominent` subdirectories for the two approaches). The `json` file contains tokenized text and the coreference clusters.

5.3 Interactions

5.3.1 Dataset

We evaluate the third part of our system on the same dataset of novels from Project Gutenberg as the first part (section 5.1.1). But we used only the sample

consisting of the first few pages of each book (usually the first chapter or the first few chapters, if chapters were short) and not the whole novels. The samples for the first 15 novels are the same as in the section 5.2. These texts are attached to the thesis in the data directory.

5.3.2 Interaction Sentences

We run our whole program on these novels. We save the sentences found with our system from the text that were marked as interaction sentences (besides dialogues) with the list of characters that were detected to have an interaction in the given sentence. We looked at these sentences, and for each of the sentences we decided if the sentence really describes the interaction as defined by us between some of the characters. We look only at sentences detected by our system. We don't look at the original texts and don't search for all of the interaction sentences in them. We decided not to evaluate whether the interaction is happening between the detected characters or between other characters in the sentence because it would penalize bad mention assignments to the character. For example, if there is the word "she" in the sentence, we don't care about what character it refers to. We just evaluate if the sentence describes an interaction between the mention "she" and mention of some other character.

Then we computed the precision: how many of the retrieved sentences really describe the interaction between at least two character mentions. This isn't the best way to evaluate the system, but without preannotated data for this task, we couldn't evaluate it much better. Our decision whether the sentence describes interaction or not might not be a hundred percent correct as we only did it by hand. When we could not decide just based on the given sentence, we also looked at the surrounding sentences in the original text.

We excluded the book "Ulysses" from the dataset as the dialogues in this book are inside the quotation marks, and so the system outputted the text inside dialogues as interaction sentences. At least one interaction sentence was detected in 19 from 29 remaining samples.

Our system detected 77 sentences as interaction sentences in these samples. We evaluated that 47 out of these sentences really describe the interaction. This gives us the precision of 61%.

Our observations during doing this evaluation are the following:

- The number of sentences detected as interaction is really low for most of the texts. It might be different if we looked at the whole book or if we used a book that is better suited for this task in the way that it contains less dialogues and more sentences that describe interaction. In the samples used, there were many dialogues. Another problem can be that an interaction is not described by only one sentence, but by more sentences together. We look only at sentences that contain the mentions of more characters.

- The sentences that were detected and don't describe an interaction sometimes described something the character is only thinking about, or something that could happen in the future, or happened in the past. Our rules did not rule it out because the sentence did not follow the basic pattern we implemented.

- Some sentences were detected as an interaction because of the token in the sentence that was treated as a character mention by our system, but it was not. The examples are names of cities or places. These errors would not happen, if we had the gold data for the mentions of characters and used our system with them.
- The results could probably be better by creating new rules or changing some of the rules. Action and speech verb lists could also be extended or some semantic relations between words could be explored, for example, by using WordNet [Fellbaum, 1998].

Examples of the interaction sentences detected in the data that really describe an interaction:

- *Stepan Arkadyevitch saw Matvey wanted to make a joke and attract attention to himself.* (from the book "Anna Karenina")
- *He looked at her, and the fury expressed in her face alarmed and amazed him.* (from the book "Anna Karenina")
- *As soon as it was moonlight, and that poor thing began to crawl and shake the pattern, I got up and ran to help her.* (from the book "The Yellow Wallpaper")
- *She laughed and said she wouldn't mind doing it herself, but I must not get tired.* (from the book "The Yellow Wallpaper")
- *Mr. Heathcliff and his man climbed the cellar steps with vexatious phlegm: I don't think they moved one second faster than usual, though the hearth was an absolute tempest of worrying and yelping.* (from the book "Wuthering Heights")
- *I uttered an expression of disgust, and pushed past him into the yard, running against Earnshaw in my haste.* (from the book "Wuthering Heights")
- *At last he came up to Morio.* (from the book "War and Peace")
- *On his way to the aunt he bowed to the little princess with a pleased smile, as to an intimate acquaintance.* (from the book "War and Peace")
- *He now took the stick from my hands and examined it for a few minutes with his naked eyes.* (from the book "The Hound of the Baskervilles")

Examples of the detected sentences in the data that do not describe an interaction:

- *This spectacle drove me back immediately; I took my hat, and, after a four - miles' walk, arrived at Heathcliff's garden-gate just in time to escape the first feathery flakes of a snow shower.* (from the book "Wuthering Heights")

  The "Heathcliff" was detected as a character mention, but it refers to the garden-gate, not to the character.

- *She conceived him as rich, but as fearfully extravagant – saw him all in a glow of high fashion, of good looks, of expensive habits, of charming ways with women.* (from the book "The Turn of the Screw")

  This sentence just talks about what "she" thinks about "him".

- *Had he gone to his aunt's, he would have been sure to meet Lord Goodbody there, and the whole conversation would have been about the housing of the poor, and the necessity for model lodging-houses.* (from the book "The Picture of Dorian Gray")

  This sentence describes what would happen if he had gone to his aunt. It is not happening now.

- *Every day at eight in the morning he was brought his breakfast by Mrs. Grubach's cook – Mrs. Grubach was his landlady – but today she didn't come.* (from the book "The Trial")

  This sentence describes what is happening every day, but not now.

- *Now, her bedside table had been pulled into the middle of the room to be used as a desk for these proceedings, and the supervisor sat behind it.* (from the book "The Trial")

  The token "her" does not mean that the character it refers to is at that place. It talks about her bedside table.

In the electronic attachment, there are the files we created during the evaluation. In the `interactions` subdirectory of the `results` directory, there are the `txt` files we generated and evaluated by hand whether they describe interaction or not(in `interaction_sentences` subdirectory). The `txt` file contains sentences detected by our system with the characters mentioned in them and our comment whether it is an interaction or not.

5.3.3 Dialogues

We evaluate the assignment of characters to the dialogues. We used 10 samples from the dataset for this purpose. The names of the used samples are in the table 5.2.

For each dialogue, we save the characters that were assigned to be part of the dialogue. We then read the dialogues and decided which characters were really part of the dialogue. We created our gold data this way, so that we could compare with the output of the system. We look only at the dialogues that were detected by our system as an interaction between two or more characters.

We compute precision and recall for each of the individual dialogues. The precision is computed as the proportion of how many of the detected characters were in the gold data. So we count how many detected characters are in the gold data and divide it by the number of detected characters. The recall is computed as the proportion of how many of the gold data characters were detected by our system. So we count how many detected characters are in the gold data and divide it by the number of the gold data characters.

We then compute the arithmetic average for all of the dialogues in one sample and finally the arithmetic average over these 10 samples.

In the table 5.2, we can see the precision and recall for the individual samples from novels and in the last row, there is an average precision and recall for these 10 samples (all numbers in the table are rounded to three decimal places).

Name	Precision	Recall
The Ambassadors	0.929	0.929
Wuthering Heights	0.976	0.880
The Call of the Wild	1	0.583
A Tale of Two Cities	1	1
Ivanhoe	1	0.75
The Turn of the Screw	0.5	0.333
Northanger Abbey	1	0.75
The Time Machine	0.833	0.329
The Scarlett Letter	1	0.833
Kidnapped	1	1
Average	0.924	0.739

Table 5.2: Precision and recall for the assignment of characters to the dialogues.

Our observations during doing this evaluation are the following:

- In most of the texts, there were a lot of sentences that were part of the dialogue. Usually, more than half of the sentences from the text contained direct speech or were close to the sentence with the direct speech and talked about the characters that were a part of the dialogue. It means that at least for these texts, it is useful to focus on the dialogues when we talk about finding interaction between characters.

- We think that our approach for finding characters in dialogues as vocatives and speakers is successful for most of the dialogues. In most cases, all of the detected characters were really part of the dialogue. This results in pretty high precision. The errors of the type that character was detected as a part of dialogue even if he was not there were mainly caused by the mention that was assigned to him even if it belonged to some other character. This is the error of the coreference resolution step. These mistakes wouldn't happen if we had the gold mentions for each of the characters.

- The low recall value is caused mainly by not detecting characters that are not even in our list of characters. Those are mainly unnamed characters or characters with unusual names or the narrator, like we discussed in section 5.1. These characters couldn't be assigned to the dialogue by our system. If we used a gold list of characters and mentions corresponding to them as an input, we think that our system would have higher recall.

- We observed that some of the direct speeches weren't detected as an interaction because there were at least two sentences without the direct speech between the sentences with the direct speech. That means that these utterances were considered to be separate dialogues consisting only of one utterance (and surrounding sentences without direct speech) by our system.

Because of that, to each of these dialogues with only one utterance was assigned only one character. We should consider whether our limit of maximum of one sentence between the utterances is not too strict.

In the electronic attachment, there are the files we created during the evaluation. In the in `dialogues` subdirectory of the `interactions` subdirectory of the `results` directory, there are the `txt` files we generated and evaluated by hand whether. The `txt` file contains sentences detected by our system as a dialogue. After the last sentence in a dialogue (dialogue consists of more sentences), there are the characters our system detected as being a part of the dialogue. On the next line, there are the gold data, we created by reading the dialogue. Below that is computed precision and recall for the dialogue. One file contains all of the dialogues for one sample from the book.

6. Implementation

Our work is divided into four main modules - Character detection, Occurrences detection, Interaction detection and Graph plotting. All of these modules can be used independently, if we use them with correct parameters. So we can, for example, use the Interaction detection module for finding interactions on the gold data list of characters and their mentions.

We have a `main.py` file that binds and uses the first three modules together. The module for the graphs is used separately. Now, we describe these modules. The installation, directory structure and usage are described in the next chapter chapter 7.

6.1 Character Detection

We describe the main classes in this module, which is in a `find_characters.py` file.

We have the class `Character` representing one character – all of the parts of his name, his id, guessed gender, a list of his occurrences in the text, number of his occurrences in the text and list of his occurrences in interactions with other characters. The list of occurrences contains the numbers corresponding to the tokens that are mentions of the character in the tokenized text. The list of interaction occurrences contains the numbers corresponding to the sentences in which the interaction for the character was detected.

The goal of this class is to store all relevant information for the character in the same place. The instance of a `Character` is first created without any information besides the id. Then the information is step by step added to the character as all of the modules are executed. In the end, we can store the `Character` instance (by using a `pickle`) for further use. When we want to further evaluate the results or just look at the graphs, we can just load these characters and don't have to process the text all over again. It is useful because processing of the text can take some time.

We have the class `SpacyTokenizer`, which has a method for processing and tokenizing the input text and returns a `SpaCy Span object` of `Tokens` and also a list of tokens in the `string` form. It uses just the basic small English model for tokenization[1]. It also contains a function for dividing the tokens to lists according to sentence boundaries detected by `SpaCy`. It takes a `SpaCy Span object` and returns the list of lists of tokens, where one sentence is in one list. It also returns a list of the cumulative lengths of sentences. That is the sum of the lengths of the sentences before the given sentence.

The main class of this module is `CharactersFinder`. It has methods that do all the work described in chapter 1. When we create the instance of this class, we specify a `spacy` parameter that decides whether `SpaCy` or `NameTag 2` is used as a NER tool. We also specify two other parameters: `doc`, that is a `SpaCy Span object` and a list of tokens. We can obtain these parameters using the class `SpacyTokenizer` first.

[1] `https://spacy.io/models/en#en_core_web_sm`

Then we can find all characters in the tokenized text by calling the method `find_all_characters`. This method takes one parameter, which specifies the minimum number of occurrences for the name in the text to be considered a character. The default value for this parameter is zero. It returns the list of `Character` objects, that have all known parts of name and gender assigned, the list `id_to_character` that is a dictionary that maps the ids to the `characters`, `first_names` and `last_names`, which are dictionaries that contain a list of all `characters` with the given first/last name

6.2 Occurrences Detection

We describe the module, which is in a `find_occurences.py` file.

The class in this module is called `OccurrencesFinder`. It has different methods that do all the work described in chapter 2. When we create the instance of this class, we specify these parameters:

- `use_coref` that tells whether to use the coreference resolution model or not
- `gendered_coref` that tells whether to use the gendered version of assigning the clusters to characters or not
- `text` that is a raw text
- `tokenized_text` that is a list of tokens,
- `characters` that is a list of `Character` objects (the output of the previous module)
- `id_to_character` that is a dictionary that maps the ids to the `characters` (the output of the previous module)

We can find all occurrences of all characters in the list by calling the method `find_all_occurences` with parameters `first_names` and `last_names`. These parameters are dictionaries that contain a list of all `characters` with the given first/last name. We have these dictionaries from the output of the previous module or can create them with the method `create_first_last_names_dicts`. The method `find_all_occurences` assigns found occurrences to the `Character` objects in the `characters` list. It returns the list `character_occurences` which has the length equal to the number of tokens and stores the `id` of a character that has a mention in the given token. If the token is not a character mention, then it stores 0 (`ids` of characters start from 1).

6.3 Interaction Detection

We describe the module, which is in a `interactions.py` file.

The class in this module is called `InteractionsDetector`. It has different methods that do all the work described in chapter 3. When we create the instance of this class, we specify these parameters:

- `characters` that is a list of `Character` objects (with the occurrences filled in),
- `doc`, that is `SpaCy Span object`,
- `tokenized_sentences` which is a list of lists of tokens, where lists correspond to sentences (we can obtain this using `SpacyTokenizer`),
- `cummulative_length` which is a list of the cumulative lengths of sentences (we can obtain this using `SpacyTokenizer`),
- `id_to_character` that is a dictionary that maps the ids to the `characters`, can be empty and created later
- `character_occurences` which is the list of `ids` of characters mentioned for every token (output of the previous module), can be empty and created later.

This class has a method for creating the dictionary `id_to_character` and the list `character_occurences` from `characters`.

We can find all the interactions by calling the method `find_interactions`. It firstly detects utterances and dialogues, then assigns characters to dialogues and finds interaction sentences besides dialogues at last. The method assigns all the found interaction sentences to the `Character` objects in the `characters` list as an indices of corresponding sentences.

6.4 Graphs

We describe the module, which is in a `graphs.py` file.

This module has methods for creating the interactive graphs that are described in the chapter 4. It uses the Tkinter library for creating the graphs.

It can be used as an entry point, its usage is described in section 7.3.2. In the `main` function, the input text and `characters` from the binary file are loaded. The `characters` must have occurrences and interactions filled in, to be shown on a graph. Then the text is tokenized using `SpacyTokenizer`. The graphs are created by calling the method `occurrences_graph` or `interactions_graph`. These methods create a Tkinter window, in the upper part is the graph that is created using the Matplotlib library and inserted in the window. In the lower part, there are two text fields, one with the whole text and the text box is scrollable, the second one serves for displaying the sentence that the user clicked on.

6.5 Main File

We describe the `main.py` file.

It serves as an entry point, its usage is described in chapter 7. In the `main` function, the input text is read, then tokenized using `SpacyTokenizer`. Then the characters are found using `CharactersFinder`, occurrences are found using `OccurrencesFinder` and interactions are detected using `InteractionsDetector`.

The output (list of `Character` objects with all the information) is stored to a binary file (using `pickle`) for later usage or for displaying the graphs by using this binary file as an input for the `graphs.py`.

This file also contains some methods for printing/saving the characters or information about them.

7. User Guide

7.1 Installation

The code attached to this thesis was written and tested on a Linux operating system. It is written in the Python programming language[1] and was developed and tested using the version 3.8.8. Firstly, you need to install Python, ideally of version 3.8.8 (the higher versions should probably work fine). You also need to install Python packages that are specified in the requirements.txt file and can be installed all at once by running:

```
pip install -r requirements.txt
```

The interactive graphs are written using the Tkinter library [2]. The Tkinter module should normally be part of the basic Python installation, but if you face some problems running it, then you must install it separately, if you want to see the graphs.

For Fedora-based Linux, it can be installed by running [3]:

```
sudo dnf install python3-tkinter
```

If the program still doesn't work after this installation, try to install the debug extension of the package, called python3-tk-dbg.

Before running the software, you will also need to have the SpaCy small language model downloaded:

```
python -m spacy download en_core_web_sm
```

When you want to run the software with the coreference resolution model, you will need to clone the code for using the model from Github[4] and download the trained model[5] in the root of the project directory. It can be done by running:

```
git clone https://github.com/vikibrezinova/fast-coref.git
gdown 1CQxUq2zvCHc1mJUEZ_Zy6WSJQqFz76Pw
```

All of these installation steps (besides installing Python and Tkinter library, which must be done before) also with creating a virtual environment called venv and installing the libraries inside this environment are packed in the install.sh file. This file works on the Linux system. For running it on the other operating system, you will probably need to change some of the commands, at least for the creation of a virtual environment. On the Linux system, you can just run:

```
./install.sh
```

[1] https://www.python.org/

[2] https://docs.python.org/3/library/tkinter.html

[3] For installation on other Linux distributions see: https://www.geeksforgeeks.org/how-to-install-tkinter-on-linux/

[4] https://github.com/vikibrezinova/fast-coref

[5] https://drive.google.com/drive/folders/1y6rE81o49g5X9ZaweLMCt3HqRNNsWgMz

when you are in the root of the project directory.

If you don't want to use the coreference resolution model and so you don't want to clone the code and download the model and install libraries that are needed only for using the model, you can run:

```
./install_without_coref.sh
```

It is the same as `install.sh`, but without the last two steps and it uses the file `requirements_without_coref.txt` for installation of only the subset of packages. The packages that are needed only for the coreference resolution model to run are not installed.

If you just want to see graphs for already processed input or on the demo input, you don't need to install all of the libraries in the `requirements.txt` file. It is sufficient to install libraries in the `requirements_graphs.txt` file.

On `Linux`, you can run:

```
./install_graphs.sh
```

It creates the virtual environment called `venv` and installs the requirements for the graphs and downloads the `SpaCy` small language model.

7.2 Project Structure

In the root directory, we have the installation scripts we can use for the installation of all packages: `install.sh`, `install_without_coref.sh`, `install_graphs.sh` and the files with requirements: `requirements.txt`, `requirements_graphs.txt`, `requirements_without_coref.txt`.

There is also a script for running the program without coreference resolution model: `run_without_coref.sh`, a script for running with coreference resolution model: `run_coref.sh` and a script for only displaying graphs: `run_graphs.sh`.

These demo scripts activate the virtual environment called `venv`, which should be created when you run the corresponding installation script. The program then runs on the sample input provided with the project. Running it with the coreference resolution model has higher memory and CPU requirements, so it may crash on the ordinary computer (we used the `UFAL AIC` cluster for running it). Using the program without the coreference resolution should work fine, but it can take a few minutes on long texts. Displaying the graphs for the already processed inputs shouldn't take more than a few seconds.

There are three main subdirectories in the project: `src`, `data`, and `results`.

In the `src` directory, there is all of the code written in Python. There is also another subdirectory called `constants` inside `src`. There are the lists of words we use, like the lists of gendered honorifics or names and the lists of speech and action verbs.

The main files inside `src` are: `find_characters.py`, `find_occurrences.py`, `interactions.py` and `graphs.py`, they can be used as a separate `Python` module. We also have a file `main.py` that connects the first three parts together and can be used as an entry point with different types of parameters. It also saves the processed output. The `graphs.py` file can be used as an entry point if we want to look at graphs for already processed texts.

In the `data` directory, there is a `sparknotes_samples` subdirectory which contains the samples from the 30 novels from the dataset described in section 5.1.1. We used these samples for evaluation, as described in chapter 5. There is a file `Little_Women.txt` that we use as a sample input. In the `binary_outputs` subdirectory, there are binary files with the output of our program in a form of pickled characters' list for two samples from the `sparknotes_samples` and for `Little_Women.txt`. These files were created without using the coreference resolution model, so there are not all of the occurrences that can be find when using the model. We use them as a sample input for creating graphs. The sample from The Little Women serves great as a sample input, because the characters are called by their names a lot in that book.

There are three subdirectories: `characters`, `interactions`, `occurrences` in the `results` directory. They contain data we generated and used for evaluation. More about the evaluation is in chapter 5.

In addition, we have an `output` directory, where we can store binary outputs and `graphs` directory, where there are six graphs that are also included in the text of this thesis.

7.3 Modes of Execution

7.3.1 Main File

When we want to run the program on the new text, we can do it by running the `main.py` file with the right command-line arguments. The input text should be in a format of plain text with `UTF-8` encoding and with a `txt` file extension. The input text should use `"\n"` or `"\r\n"` as a newline character. The input text should not be longer than 1 000 000 characters. If it is longer, only the first 1 000 000 characters of the text are processed. This limitation is due to the maximum length of the input for `SpaCy` library and for the used coreference resolution model. This limit is sufficient for novels that have around 500 pages or less, so we don't provide a script for running on longer texts. But if you want to run it on longer texts, you may process the text in batches and merge the outputs together.

The command-line arguments for the `main` file are in table 7.1.

When we have all of the dependencies installed and downloaded in the virtual environment, we need to activate the virtual environment and then we can run the program from the root directory. It can be done like this without using coreference, when we want to run it on a sample input in the `data` directory and save the output to the `output` directory:

```
source venv/bin/activate
python src/main.py -i data/Little_Women.txt
    -o output/Little_Women.bin
```

The first line just activates the virtual environment called `venv`, we can omit it, if we have the environment already activated.

We can do the same thing by running:

```
./run_without_coref.sh
```

Option	Argument	Description
`-h`	None	prints help
`-i`	filename	path to the input file with the input text, this argument is required
`-o`	filename or None	path to the binary output file, where the output will be saved, if it is not specified, then it is saved in the output directory with the same path and filename as the input, but with the .bin file extension
`-s`	None	if this option is used, the `SpaCy` is used as a NER tool, otherwise `NameTag 2` is used as default
`-c`	None	if this argument is present, the coreference resolution model is used, otherwise it is not used as a default
`-g`	None	if this argument is present, the second method for the coreference resolution looking at genders is used. The -c argument must be also present; otherwise the coreference resolution model is not used at all, so the -g argument has no effect when used without the -c argument.

Table 7.1: Command-line arguments for the main file.

It will use a `NameTag 2` as a NER tool. This can take a few minutes depending on the length of the text and the hardware.

We can run it like this using also the coreference resolution model and the `SpaCy` as a NER tool:

```
source venv/bin/activate
python src/main.py -i data/Little_Women.txt
    -o output/Little_Women.bin -s -c
```

We can do the same thing by running:

```
./run_coref.sh
```

I couldn't run the program with the coreference resolution model on my computer because of the high memory requirements, so I used the `UFAL AIC` cluster. You may face problems running it on the ordinary computer. On the cluster it runs a few minutes (circa 2 minutes for 2000 tokens) for short texts and a few hours for long texts.

The output is saved in a binary file. The list of characters with all the information about them, such as their occurrences and interactions, is saved. This output file can be used as an input for the `graphs.py` file whose usage is described below.

7.3.2 Graphs File

When we have a processed text and save the characters with all the information to a binary file (it is done automatically in `main.py`), then we can display the interactive graphs for the text using the `graphs.py` file. The command-line arguments we can use are in table 7.2.

Option	Argument	Description
`-h`	None	prints help
`-i`	filename	path to the input file with the input text, this argument is required
`-c`	filename	path to the binary file with the saved characters for the given input text, this argument is required
`-o`	None	if this argument is present, the graph with occurrences is displayed, otherwise the graph with interactions is displayed by default
`-m`	number	natural number (that can be used as `int`) that determines the minimum number of occurrences a character must have to be displayed on graph, if not present the default value is 10

Table 7.2: Command-line arguments for the graphs file.

The input file with text is expected to be in the same format as for the `main` file and it must be the same as the file that was processed to create the binary output.

We can activate the virtual environment and run the `graphs.py` from the root of the project directory like this for displaying occurrences on a sample input:

```
source venv/bin/activate
python src/graphs.py -i data/Little_Women.txt
    -c data/binary_outputs/Little_Women.bin -o
```

And for displaying the interactions graph, we run:

```
source venv/bin/activate
python src/graphs.py -i data/Little_Women.txt
    -c data/binary_outputs/Little_Women.bin
```

We can omit activating the virtual environment if it is already activated.

We can display both graphs, one after the other, by running:

```
./run_graphs.sh
```

The graphs are described also in the chapter 4.

The graph is in the upper part of the window. In the bottom part of the window, there are two text fields. In the left one, there is the entire text and it can be scrolled. In the right one, there will be the sentences that correspond to occurrences in the graph after clicking on them.

Occurrences Graph

Each point on the occurrences graph corresponds to the occurrence of the given character on the given position measured by tokens. So when the point has the x coordinate equal to 100, then this point represents that character was mentioned in the token number 100. The text is tokenized and divided into sentences by using `SpacyTokenizer`. When we click on the point, the sentence with the given occurrence is displayed in the right text box, and the token that corresponds to the occurrence is highlighted. The text in the left box is scrolled so that the sentence displayed on the right should be visible. So we can find the given sentence in the left box and look at the surrounding sentences if we want.

Interactions Graph

Each point on the interactions graph corresponds to the interaction of the given character in the given position measured by sentences with other characters that have points in the same position. Position on the x axis of the given point represents the number of a sentence from the beginning in which the interaction is happening. The text is tokenized and divided into sentences by using `SpacyTokenizer`. When we click on the point, the corresponding sentence is displayed in the right text box. The text in the left box is scrolled so that the sentence displayed on the right should be visible. So we can find the given sentence in the left box and look at the surrounding sentences if we want. This is useful because most of the interactions for ordinary books are dialogues that consist of more sentences, so we probably want to look at the dialogue as a whole.

Clicking on the graphs is inaccurate especially if the graph represents many tokens or sentences, where some of the points overlap. Sometimes, it is necessary to click more times on one point to work or to try to click on the different position on the point. Sometimes it can happen that a point corresponds to the empty sentence (consisting only of newline) that is detected to be a part of the dialogue. The clicking and getting the position of the clicked point is handled by the `Tkinter` library.

It is necessary to tokenize the text to the tokens and sentences in the same way as was tokenized when character occurrences and interactions were detected. We use the `SpacyTokenizer` for tokenization in our work. Therefore, when we create the `.bin` file with characters using `main.py` and then run `graphs.py`, it should tokenize the text in the same way.

Conclusion

In this thesis, we explored how characters, their occurrences, and interactions between them, can be found automatically. We divided the work into four parts that can be used separately, even when our work connects them together.

In the first part, we generate the list of named characters that occurred in the text. We use a NER tool as a starting point and then try to merge different namings of one character using rules that were inspired by the previous works that we described. We compared using the `Nametag 2` and `SpaCy` library as the tool for the Named Entity Recognition task. We evaluated this part on 30 novels and analyzed the main problems that occurred.

In the second part, we use the model for the coreference resolution as a tool to find out all mentions of the detected characters in the text. We proposed two different methods for processing the output of the coreference resolution model. We found out that there is only a small difference between these methods when we tried them on the unseen data.

In the third part, our goal was to detect places in the text where two or more characters interact together. When we analyzed this problem and researched some of the existing works, we found out that this is a complex topic that was not explored much. We defined the interaction our own way and decided to only focus on dialogues and sentences with mentions of two or more characters. We used rules for deciding which characters are part of a dialogue and for detecting sentences that describe interaction besides the dialogues. We created these rules by analyzing previous works and analyzing novels from the LitBank dataset. We evaluated the rules on the unseen texts. We found out that assigning characters to dialogues works pretty well, but the detection of interaction sentences has a lot of space for improvement. We analyzed the most common errors and proposed what can be improved or explored more.

In the fourth part, we created interactive graphs that show us the occurrences and interactions of characters throughout the book. We can use them to quickly see which characters occurred in which parts of the book, which characters interacted together, and when. We can use them for displaying the sentences in which the occurrence or interaction happened.

We believe that this work can make you think about how the interaction in books is described and that it proposes new ideas that can be explored in the future. We think that all of the parts of our work could be further improved and should be evaluated on a bigger dataset.

Bibliography

David Bamman, Olivia Lewke, and Anya Mansoor. An annotated dataset of coreference in english literature. *CoRR*, abs/1912.01140, 2019a. URL `http://arxiv.org/abs/1912.01140`.

David Bamman, Sejal Popat, and Sheng Shen. An annotated dataset of literary entities. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2138–2144, Minneapolis, Minnesota, June 2019b. Association for Computational Linguistics. doi: 10.18653/v1/N19-1220. URL `https://aclanthology.org/N19-1220`.

Mariona Coll Ardanuy and Caroline Sporleder. Structure-based clustering of novels. In *Proceedings of the 3rd Workshop on Computational Linguistics for Literature (CLFL)*, pages 31–39, Gothenburg, Sweden, April 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-0905. URL `https://aclanthology.org/W14-0905`.

Micha Elsner. Character-based kernels for novelistic plot structure. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 634–644, Avignon, France, April 2012. Association for Computational Linguistics. URL `https://aclanthology.org/E12-1065`.

Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

Kevin R. Glass and Shaun Bangay. A naïve, salience-based method for speaker identification in fiction books. 2007.

Hua He, Denilson Barbosa, and Grzegorz Kondrak. Identification of speakers in novels. In *ACL*, 2013.

Vincent Labatut and Xavier Bost. Extraction and analysis of fictional character networks: A survey. *CoRR*, abs/1907.02704, 2019. URL `http://arxiv.org/abs/1907.02704`.

Weronika Lajewska and Anna Wr'oblewska. Protagonists' tagger in literary domain - new datasets and a method for person entity linkage. *ArXiv*, abs/2110.01349, 2021.

Alireza Mansouri, Lilly Suriani Affendey, and Ali Mamat. Named entity recognition approaches. *International Journal of Computer Science and Network Security*, 8(2):339–344, 2008.

Grace Muzny, Michael Fang, Angel Chang, and Dan Jurafsky. A two-stage sieve approach for quote attribution. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 460–470, Valencia, Spain, April 2017. Association for Computational Linguistics. URL `https://aclanthology.org/E17-1044`.

Joakim Nivre. Dependency parsing. *Language and Linguistics Compass*, 4(3): 138–152, 2010.

Jana Straková, Milan Straka, and Jan Hajič. Neural architectures for nested ner through linearization. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5326–5331, Stroudsburg, PA, USA, 2019. Association for Computational Linguistics. ISBN 978-1-950737-48-2.

Shubham Toshniwal, Sam Wiseman, Allyson Ettinger, Karen Livescu, and Kevin Gimpel. Learning to Ignore: Long Document Coreference with Bounded Memory Neural Networks. In *EMNLP*, 2020.

Shubham Toshniwal, Patrick Xia, Sam Wiseman, Karen Livescu, and Kevin Gimpel. On Generalization in Coreference Resolution. In *CRAC (EMNLP)*, 2021.

Hardik Vala, David Jurgens, Andrew Piper, and Derek Ruths. Mr. bennet, his coachman, and the archbishop walk into a bar but only one of them gets recognized: On the difficulty of detecting characters in literary texts. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 769–774, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1088. URL `https://aclanthology.org/D15-1088`.

List of Figures

List of Tables

A. Attachments

A.1 List of Action Verbs

nod, encourage, go, meet, witness, sail, touch, hunt, climb, shout, walk, search, grasp, give, look, announce, tell, say, warn, beg, see, watch, assure, glance, gaze, sit, greet, help, offer, understand, bring, recieve, seize, hear, take, ask, talk, put, hunt, mumble, shout, run, hit, play, notice, attack, hold, follow, call, shoot, wander, approach, move, reply, show, cry, drive, shake, point, stand, thank, bow, face, reach, beat, laugh, amuse, explain, add, encounter, report, slap, smile, stab, tease, pull, return, support, mutter, interrupt, dance, chat, cling, carry, observe, kick, yell, kiss, hug, frown, listen, fight, ride, clap, scream, dine, eat, drink, come, hunt, escort, lie

A.2 List of Novels from Evaluation Dataset

- The Scarlett Letter
- The Time Machine
- Ivanhoe
- A Connecticut Yankee
- A Tale of Two Cities
- Treasure Island
- Northanger Abbey
- The Turn of the Screw
- The Call of the Wild
- Heart of Darkness
- Dracula
- An Occurrence at Owl Creek Bridge
- Kidnapped
- The Ambassadors
- The Age of Innocence
- Wuthering Heights
- White Fang
- The Last of the Mohicans
- The Count of Monte Cristo

- The Three Musketeers
- Anna Karenina
- The Yellow Wallpaper
- Crime and Punishment
- War and Peace
- The Hunchback of Notre-Dame
- The Idiot
- The Hound of the Baskervilles
- The Picture of Dorian Gray
- Ulysses
- The Trial