Politecnico di Torino

Master's Degree in Data Science and Engineering

Enhanced Correction and Multi Language Support of Transcription on "Tell your story" Digital Platform

By

Chen Zifan

******

Supervisor(s):

Prof. Luciano Lavagno

Politecnico di Torino

2023

Acknowledgements

Firstly, I wish to express my gratitude to my supervisor, Professor Luciano Lavagno, for his generous guidance, encouragement, and patience throughout my thesis journey.

My sincere thanks also go to Dr. Paolo Pasini, whose practical advice was instrumental in the completion of my work. Without his assistance, the completion of this work would not have been feasible.

I am profoundly grateful for the emotional support extended by Simona, Ilaria, and Dr. Huang, who provides psychological counseling and their assistance has been invaluable to me.

I also deeply appreciate the economic support provided by my parents during my studies.

Lastly, I dedicate my work to my cousin who was murdered and my best friend who died by suicide. The privilege of being alive and having the chance to learn more about the world is indeed a blessing.

Abstract

This thesis represents a practical work, concentrating on the application of a dynamic programming algorithm to enable correction and multi-language support of audio transcriptions on the web platform, *Ti Racconto Una Storia*, or *Tell Your Story* in English.

Ti Racconto Una Storia is a digital platform that incorporates an interview system that automatically records stories via voice messages. Users can record their stories as brief or detailed voice messages, either in response to a series of questions or in an unrestricted format. The arriving story is set as private by default, but it's possible for the story-teller to allow the audio being broadcasted to public on the website, and concurrently, an audio transcription is displayed to enhance comprehension and sharing. Each word in the transcription is timestamped, linking it to the corresponding segment in the audio.

In this thesis, we have developed a series of methodologies using Django to facilitate two functionalities. **The first objective** of this thesis is to empower the owner of story to rectify the machine-generated audio transcription, while automatically recalculating or preserving the timestamp of each word to the corresponding audio segment. **The second objective** is to enable the story-teller or the platform administrator to generate a translated rendition of the audio transcription, while striving to retain the alignment of each translated term as closely as possible to its original position in the audio.

The core algorithm employed in this thesis is known as the Levenshtein Distance. This dynamic programming algorithm is frequently utilized to gauge the similarity between two strings or even broader sequences such as time series and DNA, and to ascertain the optimal alignment between two sequences. It finds extensive application in fields such as computational biology, machine translation, speech recognition, and named entity extraction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis embodies a practical undertaking, mainly focusing on the extensibility, portability and incorporation of Levenshtein Distance to enhance correction and multi language support of audio transcriptions on the web platform *Ti Racconto Una Storia*, or *Tell Your Story* in English.

In this chapter, we present a comprehensive overview of the thesis, detailing the scope and objectives of our research. The focus of this study is to enhance *Ti Racconto Una Storia* by developing and integrating new functionalities. *Ti Racconto Una Storia*, central to our research, serves as a platform for users to engage with audio content. Our initial step involves a thorough introduction to this website, including its primary purpose, user interface, and current features.

We then delve into identifying and discussing the existing limitations and challenges faced by the website. The limitations may encompass a range of issues, from technical constraints, such as inadequate audio transcription accuracy, to user experience problems, like limited accessibility for non-native speakers or those with hearing impairments. By pinpointing these issues, we can more effectively tailor our development efforts to address these specific challenges.

Following the identification of these limitations, we outline the primary objectives of the thesis. These objectives are formulated with the aim of not only overcoming the current challenges but also enhancing the overall functionality and user experience of the website. The goals are set to be both realistic and impactful, ensuring that the outcomes of this thesis will provide improvements to the website.

1.1 *Ti Racconto Una Storia*

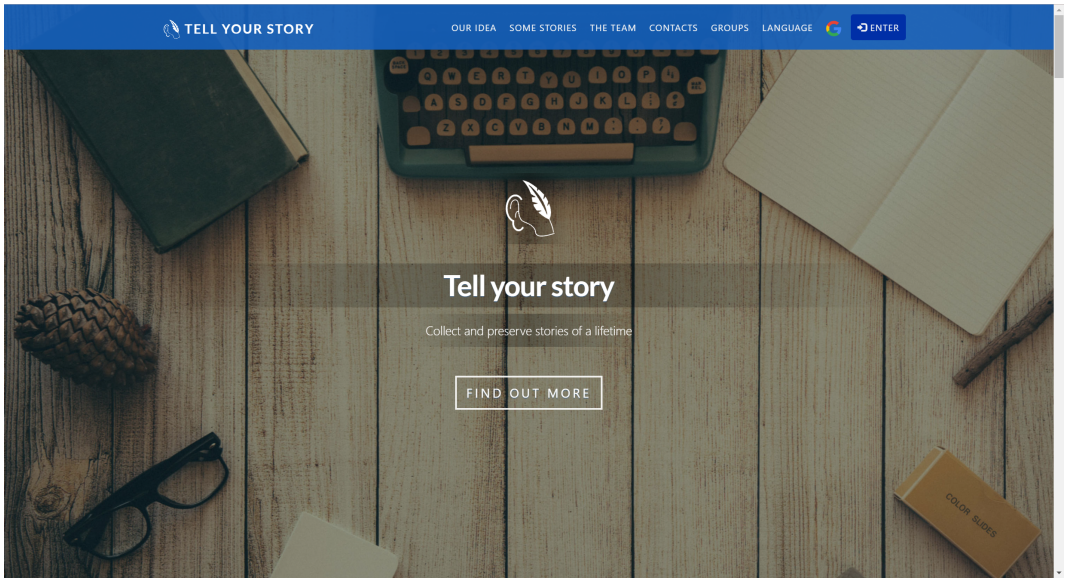

Fig. 1.1 Home page 1[1]

1.1.1 Brief introduction on the platform

Ti Racconto Una Storia is a unique web platform, conceived by a diverse team of professionals, that serves a crucial societal function by preserving, sharing, and listening to individuals' lifetime memories. The platform features an interview system that records stories automatically via voice messages, making the process quick and intuitive. The platform's main features include:

1. **Recording Stories**: Users have the flexibility to record their narratives as either short or extensive voice messages, either responding to a set of questions or in a free-flowing format. All that's required is a smartphone and an enabled Telegram account.

2. **Listening and Remembering**: Beyond just listening to the live voice of the narrator, the platform also offers automatic transcription of story audios, and extracts keywords to assist in pinpointing intriguing aspects. The transcriptions retain the genuine essence of the spoken language, capturing the raw emotions and freshness of the storyteller's words, similar to podcasts to some extent.

3. **Sharing**: The narratives are made accessible on individual web pages, available exclusively to the ones who recorded them, to family members or friends with whom they choose to share, or, if they so wish, to the global community. The ownership of the stories always remains with the user.

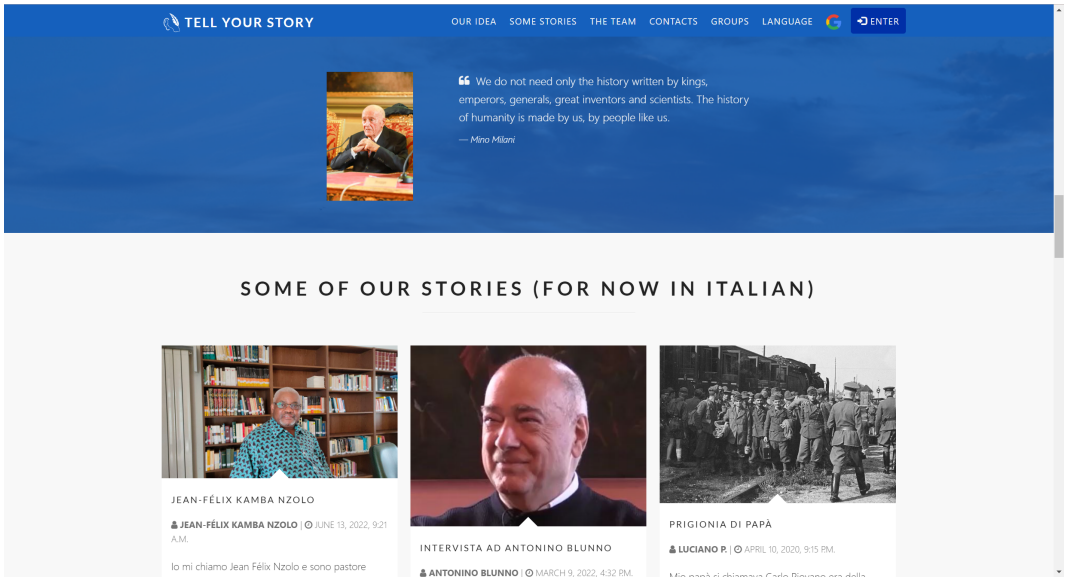

Fig. 1.2 Home page 2[1]

The platform is crafted to restore the importance of personal experiences, with a special emphasis on the elderly, by reducing their feelings of isolation and assisting in the recollection of memories. Furthermore, it provides a mechanism for relatives and friends to delve into and discover specific information within these oral narratives.

Contrary to other social platforms such as Facebook or Twitter that prioritize fleeting moments, concise texts, and images, *Ti Racconto Una Storia* places emphasis on enduring narratives, the human voice, and the unfolding of stories. This makes it an invaluable resource for the creation of family archives and for contributing to a shared historical record.

1.1.2 Usage

Narratives are initially captured and documented through Telegram voice messages by users. Subsequently, these recordings are aggregated and organized on a website.

This digital platform transforms them into a combination of audio tracks, written transcripts, and key terms. To commence this process, it is essential for users to download and install Telegram on their smartphones. Once installed, users initiate a conversation with a Telegram bot, known as @*tiraccontobot*, which is searchable among the Telegram user base. Ensuring the accuracy of the bot's name during this search is crucial.

Upon the first interaction, the bot prompts users to register on the website *Ti Racconto Una Storia*. This is facilitated through a direct link, presented as a button within the chat interface. By following this link, users can create a personal account, agree to the terms of informed consent, and submit their personal information. Notably, providing an email address is critical for password recovery purposes in the future.

After registration, users may resume their interaction with the bot on Telegram. If there is a delay in the bot's response, sending a */start* message can reactivate the conversation.

When initiating a recording, the bot offers users a selection of interviews. These interviews consist of a set of questions to be posed to the interviewee. Options include a standard interview format, focusing on family history, a free-form interview with a single open-ended question, or the opportunity to design a custom interview. For the custom option, users are required to send both the text of their question and the corresponding voice response to the bot. Conversely, for pre-structured interviews, only the voice responses need to be recorded and sent.

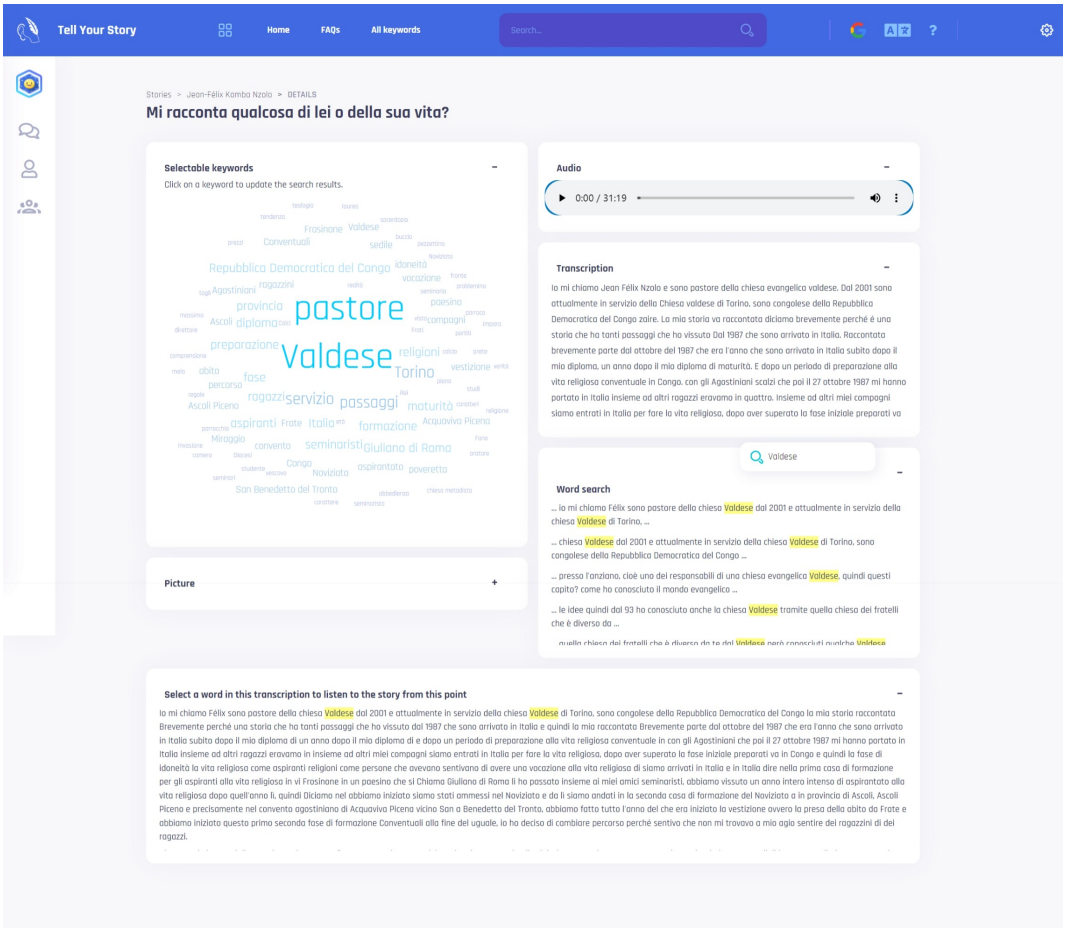

Fig. 1.3 A free-form interview

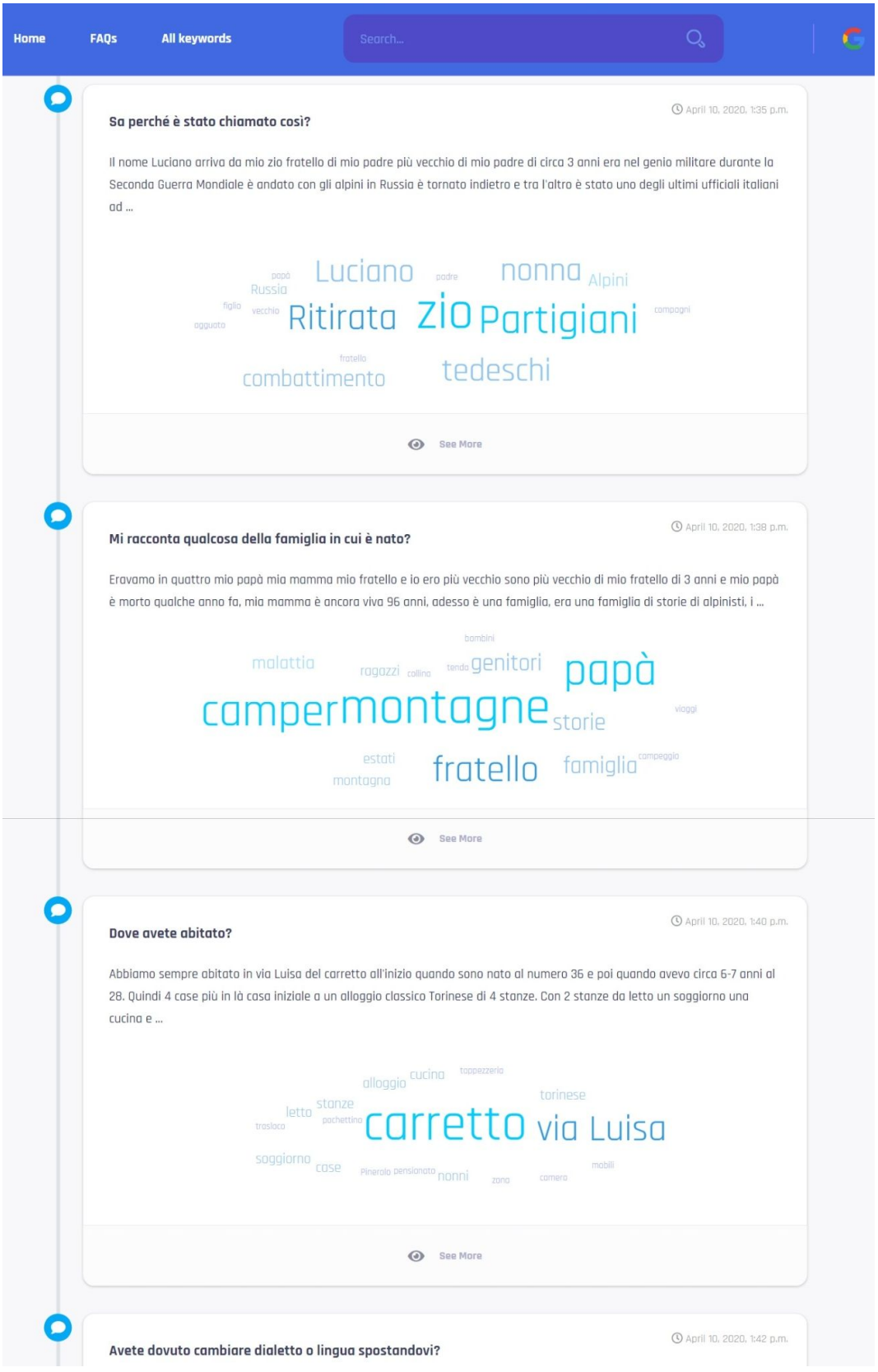

Fig. 1.4 A standard interview

It is important to note that after completing an interview or if users leave the questionnaire incomplete, they must select the 'End interview' button. This action ensures that all data is transmitted to the website for transcription and subsequent playback.

Upon processing their story, the bot provides users with a link for them to access and listen to it on the website. This link also enables them to share the story with others. Interviews can be conducted either in person, by recording voice messages directly on Telegram, or remotely, by forwarding voice messages to @*tiraccontobot*.

After recording a story, it is initially set to private, accessible only to the individual user. To share it with others, users log in to *Ti Racconto Una Storia* and navigate to *Stories* or *My Stories* in their profile. Each story is cataloged by the interviewee's identity and the questions posed. Through the *Sharing* option, users can choose to make a story public or share it privately via email, Telegram, WhatsApp, or other preferred methods. However, users should be mindful that anyone with access to the shared link can further distribute it.

1.2 Current limitations

In the current state, the website *Ti Racconto Una Storia* hosts a collection of stories and responses, with the original audio recordings preserved. To enhance the user experience, previous efforts have utilized the Google Speech-to-Text (STT) service to transcribe these audio files. Each word in these transcriptions is meticulously linked to a corresponding timestamp within the audio file. This link between text and time creates an interactive and user-friendly interface, allowing users to navigate the narrative with ease. By simply clicking on a word in the transcription or searching for a specific term, users can initiate audio playback from the exact point of interest, thus providing a seamless and focused listening experience.

However, this system is not without its limitations. A common issue arises during the recording of interviews, where often, superfluous words are captured. These can include verbal fillers, colloquial phrases, or even inappropriate language. When such automated transcriptions are displayed on the website or shared with family, friends, or the public, there is a high likelihood that users will want to edit or refine the subtitles to remove these unnecessary elements. This need for revision highlights

a gap in the current system's ability to revise irrelevant or undesirable content from the final transcription.

Additionally, while Google's STT technology is advanced and generally reliable, it is not infallible. Errors in transcription can occur, stemming either from the user's speech patterns, such as accents or diction, or from limitations within the STT technology itself, such as its ability to handle diverse languages and dialects accurately. These inaccuracies can lead to misunderstandings or misrepresentations of the original audio, necessitating further review and correction by the user.

Furthermore, there is an ambition to expand the website's capabilities by introducing a feature for translating transcriptions into various languages. This feature aims to maintain the original audio while providing translated subtitles that are as closely aligned with the audio as possible. The goal is to achieve word-level alignment, which is more precise than the common sentence-level alignment found in platforms like YouTube. This finer level of synchronization between translated text and audio is crucial for preserving the context and nuance of the original narrative, especially in a multilingual setting. Achieving this would not only enhance the accessibility of the content to a global audience but also set a new standard in the field of audio transcription and translation, moving beyond the limitations of current technologies.

1.3 Goal of the thesis

In this thesis, a comprehensive series of methodologies has been meticulously developed, utilizing the Django framework, to introduce and refine two key functionalities. The primary focus of *Ti Racconto Una Storia* is to uphold the genuine nature and emotional depth of spoken language. This focus underscores the importance of preserving the original character and nuances of the narrative, rather than merely correcting linguistic anomalies or idiosyncrasies. However, recognizing the diverse needs of our users, we have implemented an automatic transcription feature to enhance the readability of the content for various audiences, including the storyteller, their family, friends, and the general public.

In this project, based on previous work, we have incorporated a significant feature that enables users to actively engage in editing and refining the automated transcriptions of their voice messages. This functionality not only supports user

autonomy but also underscores the critical role of accuracy and personalization in the art of storytelling. **The primary objective** of this research is to empower storytellers – those who are the creators and owners of the narratives – to rectify any inaccuracies present in the machine-generated transcription. This capability is crucial in ensuring that the essence and authenticity of their stories are preserved.

The enhancement of this process is achieved through the implementation of the dynamic programming algorithm known as the Levenshtein Distance. This algorithm plays a pivotal role in maintaining the integrity of the transcription process. It operates by either recalculating or preserving the timestamps associated with each word in the transcription. This technical aspect is of paramount importance, as it ensures that any modifications made to the text are accurately reflected in the corresponding segments of the audio. In other words, when a storyteller alters a word or a phrase in the transcription, the Levenshtein Distance algorithm adjusts the timestamps so that these changes are perfectly aligned with the audio timeline. This alignment is essential for maintaining the coherence between the spoken word and its written representation, thereby enhancing the overall listening and reading experience.

The second objective of this research is to enhance linguistic inclusivity and accessibility by enabling the translation of audio transcriptions into a variety of languages. This functionality is particularly beneficial for storytellers or platform administrators who are keen to engage a broader audience. A significant challenge in this endeavor is the development of a translation mechanism that not only achieves accurate linguistic conversion but also ensures that each translated word is aligned as closely as possible with its corresponding segment in the original audio stream. This precise alignment is vital for maintaining the narrative's rhythm and ensuring that the translated text reflects the timing and context of the spoken words accurately.

Unlike current video or podcast platforms, which typically provide only sentence-level alignment between source paragraphs and their translations, this research strives to align corresponding words with each other. This word-level alignment is more granular and sophisticated than the conventional sentence-level alignment, offering a more nuanced and contextually accurate translation. It allows listeners to follow along more closely with the original narrative flow, enhancing their understanding and engagement with the content. In this thesis, by offering multiple methods for word alignment, from rapid local mechanics to advanced BERT-based techniques, we

provide a range of solutions to meet diverse needs and scenarios, thereby significantly enhancing the reach and impact of audio narratives across different languages and cultures.

Chapter 2

Methodologies

This chapter delves into the methodologies employed in this thesis, focusing on the specific algorithms and their theoretical foundations that are pivotal to our research. At the heart of our project are two key algorithms: the Levenshtein Distance algorithm and multilingual BERT. These were selected due to their proven effectiveness in processing and understanding language-based data, a core component of our study. The Levenshtein Distance algorithm is renowned for its ability to measure the difference between two sequences of text, which is crucial in our task of refining audio transcriptions. On the other hand, multilingual BERT stands out for its advanced capabilities in understanding and interpreting multiple languages, an essential feature for our project's aim to support diverse languages. The subsequent sections will provide a comprehensive explanation of these algorithms, shedding light on their functionalities, and why they are particularly suited for the objectives of this research.

2.1 Django - the back-end framework of our project

In our project, Django, a high-level Python web framework, plays a crucial role in the development of our web application. Django is celebrated for its pragmatic design and its ability to facilitate rapid development. A key feature of Django is its adoption of the Model-Template-View (MTV) architectural pattern, which is a variant of the widely known Model-View-Controller (MVC) architecture. This

pattern is instrumental in structuring and organizing code in a clean and manageable way.

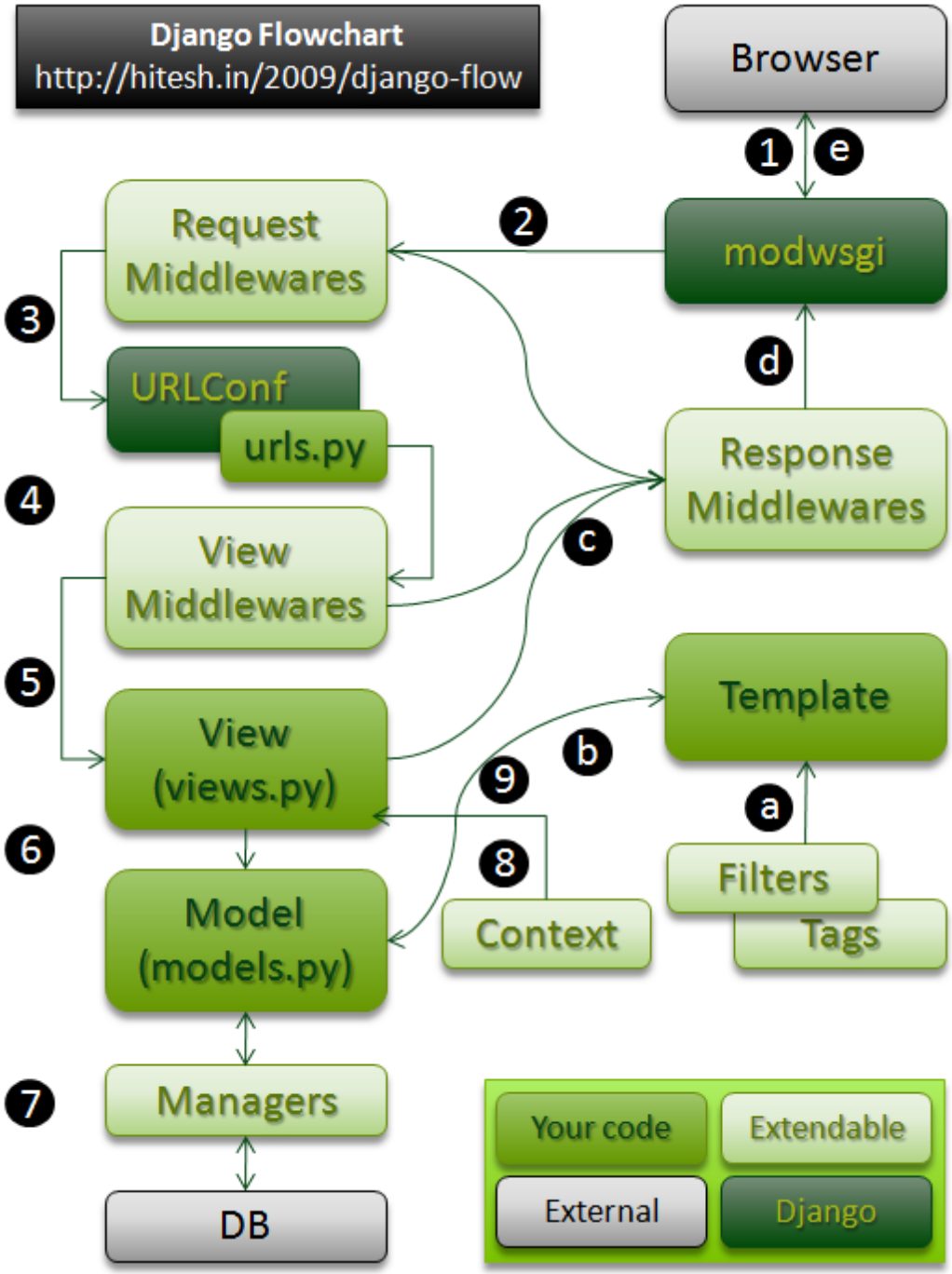

Fig. 2.1 Django dataflow[2]

The MTV architecture divides the web application into three interconnected components. The Model, the focus of our work, is responsible for handling the data and the business logic of the application. It is the layer that manages the database, defining the data structure and providing the essential mechanisms for storing, retrieving, and manipulating the data. The Template refers to the presentation layer, which handles the user interface part of the application. Lastly, the View is concerned with the logic that controls what data is displayed in the Template.

In this thesis, the emphasis is placed predominantly on the Model component of Django's MTV architecture. This decision aligns with our research objectives that are centered around data handling and processing. By leveraging Django's robust and efficient Model framework, we can focus on designing and implementing the data structures and algorithms necessary for our application, without the need to delve deeply into the other components of the MTV pattern. This approach allows for a more concentrated effort on the backend logic and data management, which are critical to the success of our project.

A pivotal aspect of Django's Model framework is its QuerySet API, which encapsulates the intricacies of database operations, including the creation, retrieval, updating, and deletion of data (commonly known as CRUD operations). This encapsulation is particularly advantageous for our project as it simplifies the development of data structures. The QuerySet API provides a high-level, Pythonic way to interact with the database, allowing us to focus on the logic of our application rather than the underlying database queries. Furthermore, this abstraction layer means that our application can interact with various types of databases seamlessly. Whether we choose to use SQLite, PostgreSQL, MySQL, or another database system, Django's Model framework ensures that the majority of our database interactions remain consistent, thus minimizing the need for code modifications when changing database systems.

2.2 Choice of minimum edit distance algorithm

In this section, we delve into the rationale behind selecting the minimum edit distance algorithm, specifically the Levenshtein Distance, for our project, which focuses on enhancing the accuracy and efficiency of user-corrected audio transcriptions. Our primary goal is to adeptly modify existing text transcriptions, stored as strings in

our server database, in response to user corrections submitted through the network. This task involves transforming the original text into the new, user-requested version while minimizing the database operations required, such as insertions, deletions, or substitutions. This requirement is perfectly aligned with the essence of the minimum edit distance algorithm, which is designed to determine the smallest number of edits needed to change one string into another.

In the landscape of string comparison algorithms, there are several notable ones, including Hamming Distance, Longest Common Subsequence, N-Gram and Cosine similarity[4]. Each of these algorithms has its unique strengths and applications. For instance, Cosine similarity are effective in measuring string similarity in terms of character composition and arrangement, while the N-Gram approach focus on the similarity between sets of characters or sequences. However, for our specific application, these algorithms do not precisely address our need to minimize database operations with each transcription correction.

It is important to note that among the various string similarity measures, some are metric distances, meaning they satisfy the triangle inequality $d(x,y) \leq d(x,z) + d(z,y)$. The Levenshtein distance, for instance, is a metric distance, whereas the Normalized Levenshtein distance is not. This distinction is crucial as many nearest-neighbor search algorithms and indexing structures depend on the triangle inequality for efficiency and accuracy. Non-metric similarity measures, therefore, might not be compatible with these algorithms and structures.

The concept of the triangle inequality in the context of string distances is quite intuitive and particularly relevant to our situation, where we calculate the distance between two strings. Consider a scenario where we have an original string in state X and a new string in state Y. According to the triangle inequality, if $d(x,y) > d(x,z) + d(z,y)$, it implies that it would be more cost-efficient to first modify the string from state X to an intermediate string, state Z, and then from state Z to state Y. This scenario, however, is counterintuitive in the realm of string manipulation and violates the basic principles of dynamic programming, which seeks to optimize the process by breaking it down into simpler, more direct steps.

Dynamic programming, a method often employed in computing the Levenshtein distance, relies on the principle of optimally solving smaller sub-problems to efficiently solve larger problems. In the context of string manipulation, it means directly transforming string X into string Y in the most efficient manner, without

detouring through an intermediate state Z. If the triangle inequality were violated, as in the case of $d(x,y) > d(x,z) + d(z,y)$, it would suggest that a less direct route (via state Z) is more efficient, which contradicts the optimization strategy inherent in dynamic programming. This contradiction highlights why metric distances, like the Levenshtein distance, are preferred in scenarios where efficiency and directness of transformation are key, as they adhere to the triangle inequality and thus align with the principles of dynamic programming.

There are also several advanced options, such as Smith-Waterman, Bowtie, and the Burrows-Wheeler Transform (BWT), commonly employed in bioinformatics for complex sequence alignment tasks. Some advanced data structures like Trie and Suffix Tree is leveraged for speeding up computation. However, these algorithms, while powerful, may not be ideally suited for our project's scope due to their complexity and specific optimization for biological data.

The structure of our data further informs our choice. Each word in the original transcription is encapsulated in a model named *AnswerByWord*, which includes keys for the answer (as a foreign key), the word itself, and its start and end timestamps within the current answer. When insertions, deletions, or substitutions are applied, they affect not only the entire text but also any influenced word and its corresponding *AnswerByWord* record. Therefore, minimizing these operations is crucial to maintain database efficiency and integrity.

Therefore the Levenshtein Distance algorithm emerges as the most suitable for our goal due to its direct approach to measuring the difference between two strings. It precisely quantifies the minimum number of single-character edits (insertions, deletions, substitutions) required to change one word into another. This aligns seamlessly with our need to efficiently update the *AnswerByWord* records with minimal database operations, ensuring that each user correction leads to the least amount of change necessary in both the text and the associated data structures.

2.2.1 Brief introduction on Levenshtein Distance

The pivotal computational tool employed is the Levenshtein Distance algorithm. This algorithm, rooted in the principles of dynamic programming, serves as a fundamental technique for measuring the degree of similarity between two sequences, which can be strings of text, time series data, or even complex biological sequences like DNA.

Its versatility extends to determining the most efficient way to align these sequences, a process that involves identifying the minimum number of operations required to transform one sequence into the other.

The Levenshtein Distance algorithm calculates this similarity by considering three primary types of edit operations: insertions, deletions, and substitutions. For instance, when comparing two text strings, the algorithm assesses how many insertions, deletions, or substitutions are necessary to convert one string into the other. The fewer the required operations, the more similar the two strings are considered to be.

This algorithm finds widespread application across various domains. In computational biology, it is instrumental in comparing genetic sequences, aiding in the identification of genetic variations and evolutionary relationships. The algorithm's ability to handle DNA sequences makes it a valuable tool for genetic research and diagnostics.

In the realm of machine translation and natural language processing, the Levenshtein Distance is used to evaluate and enhance the accuracy of translations between languages. By comparing translated text with a reference standard, it helps in refining translation models, thereby improving the quality of machine translation outputs.

Furthermore, in speech recognition technology, this algorithm plays a critical role in improving the accuracy of transcribed text. It helps in comparing the spoken word (converted into text) with a database of known words or phrases, thereby enabling the system to correct errors and understand context better.

Additionally, the Levenshtein Distance algorithm is often employed in named entity extraction, a process crucial in information retrieval and data mining. It aids in accurately identifying and classifying key information from unstructured text data, such as names of people, organizations, locations, and more.

2.3 Efforts on multi-lingual word-align

Addressing the multifaceted challenge of multi-lingual word-level alignments in user audio transcriptions and their translations requires a nuanced approach, considering the semantic complexities inherent in different languages. A simplistic approach

based on string matching or regular expressions is inadequate for this task due to the intricate nature of language semantics and structure.

To tackle this challenge, we have explored various methods, ultimately narrowing down to three feasible approaches, each with its own set of advantages and limitations. The first approach is the simplest, ensuring alignment only at the beginning and end of each sentence. This method, akin to sentence-level alignment, demands the least computational resources. However, its simplicity also means that the alignment accuracy is relatively low, as it does not consider the intricacies of the content within the sentences.

The second approach involves the use of Natural Language Processing (NLP) techniques, particularly focusing on Part-of-Speech (POS) tagging. Initial experiments indicated that POS tagging outperforms Named Entity Recognition (NER) in this context. By aligning words with the same lexical properties in both the original text and its translation, this method offers a more nuanced alignment than the first approach, though it still has limitations in capturing the full semantic context.

The third and most sophisticated approach employs multi-lingual BERT, a state-of-the-art NLP model. This method leverages the advanced capabilities of BERT in understanding and interpreting multiple languages, offering a more comprehensive and context-aware alignment. While this approach is computationally more intensive, it holds the potential for significantly higher accuracy in aligning words across different languages by understanding their contextual meanings.

Each of these approaches represents a different point in the trade-off spectrum between computational resource requirements and alignment accuracy. The choice of method depends on the specific requirements and constraints of the project, including the desired level of accuracy and available computational resources.

2.3.1 POS

Part-of-Speech (POS) tagging is a fundamental technique in the field of Natural Language Processing (NLP) that involves the classification of words in a text (corpus) into their respective parts of speech, such as nouns, verbs, adjectives, and adverbs. This classification is based on both the definition of the word and its context within the sentence. POS tagging is essential for understanding the grammatical structure

of sentences and is a precursor to more complex NLP tasks like syntactic parsing and semantic analysis.

The process of POS tagging typically involves assigning tags to words from a predefined set of parts of speech. This set can vary depending on the tagging system used but generally includes the major word classes and sub-classes like singular/plural nouns, past/present verbs, comparative/superlative adjectives, and so on. The accuracy and complexity of POS tagging can vary significantly based on the language's grammatical rules and the context in which words are used.

Common methods for POS tagging include rule-based, stochastic, and machine learning approaches. Rule-based systems rely on a set of handcrafted rules and dictionaries to assign the correct tags. These systems are often limited by the completeness and complexity of their rule sets. Stochastic methods, such as Hidden Markov Models (HMMs), use probabilistic models trained on annotated corpora to predict the most likely tag for each word based on its context. More recently, machine learning approaches, particularly those using deep learning, have become prevalent. These methods, which include using neural networks, can learn complex patterns from large datasets, leading to more accurate and contextually nuanced tagging.

Several tools and libraries are available for POS tagging across different languages. Popular examples include NLTK for Python, which provides access to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning. Another example is the Stanford POS Tagger, a part of the Stanford NLP suite, which is known for its high accuracy and support for multiple languages.

The application of POS tagging in our project is centered on aligning words with similar lexical properties from the original text and its translated version. By identifying and matching words that share the same POS tag in both languages, this method provides a more nuanced alignment compared to basic sentence-level alignment. For instance, if a noun in the original text is aligned with a noun in the translated text, there is a higher probability of these words being contextually related or equivalent.

However, while POS tagging offers a more refined approach than aligning merely based on sentence boundaries, it is not without limitations. One significant challenge

is its potential inadequacy in capturing the full semantic context of words. POS tags primarily provide syntactic information, which may not always correlate with semantic meaning. For example, a noun in one language could be translated into a different part of speech in another language while retaining the same meaning. This discrepancy can lead to misalignments in cross-lingual contexts.

Moreover, the effectiveness of POS tagging in cross-lingual alignment is also contingent on the accuracy of the POS taggers used for each language. Variations in linguistic structures and nuances across languages can pose challenges, as POS taggers trained on one language may not perform with the same level of accuracy on another. This can result in errors or inconsistencies in alignment, particularly in languages with complex morphologies or those that significantly differ from the language the POS tagger was originally trained on.

2.3.2 BERT

Bidirectional Encoder Representations from Transformers (BERT) represents a significant advancement in the field of Natural Language Processing (NLP), particularly in the area of word embeddings. Developed by researchers at Google, BERT has revolutionized how machines understand and interpret human language, setting new standards for a range of NLP tasks.

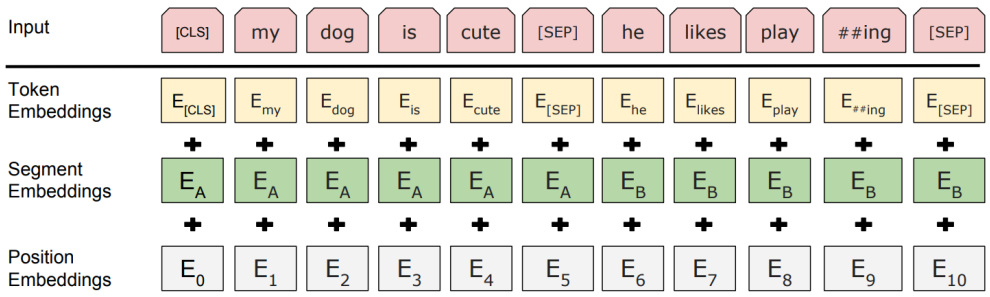

Fig. 2.2 BERT input representation[3]

Word embeddings, a foundational element in NLP, are a form of word representation that allows words with similar meanings to have a similar representation. They are essentially vectors in a high-dimensional space, where each dimension captures some aspect of the word's meaning and the relative position of a word in this space

encodes semantic similarities with other words. This concept is a cornerstone in the field of NLP, enabling deep learning models to process text by converting words into numerical form.

Prior to BERT, word embeddings were typically generated using models like Word2Vec or GloVe, which create a fixed embedding for each word. However, these models have limitations, as they generate the same embedding for a word regardless of its context, leading to a loss of meaning in cases where a word has multiple meanings based on its usage.

BERT addresses this limitation through its context-aware embeddings. Unlike previous models, BERT generates embeddings for words based on the other words in the sentence, meaning the same word can have different embeddings based on its context, allowing for a much richer understanding of language nuances.

The core innovation in BERT is its use of the transformer architecture, a model that relies on attention mechanisms to process words in relation to all other words in a sentence, rather than sequentially. This bidirectional approach is a departure from previous models that processed words either from left to right or right to left. The transformer consists of two parts: the encoder, which reads and processes the input text, and the decoder, used in generating a prediction. BERT utilizes only the encoder part of the transformer. This architecture enables the model to capture the context of a word based on all other words in a sentence, leading to a deeper and more accurate understanding of language.

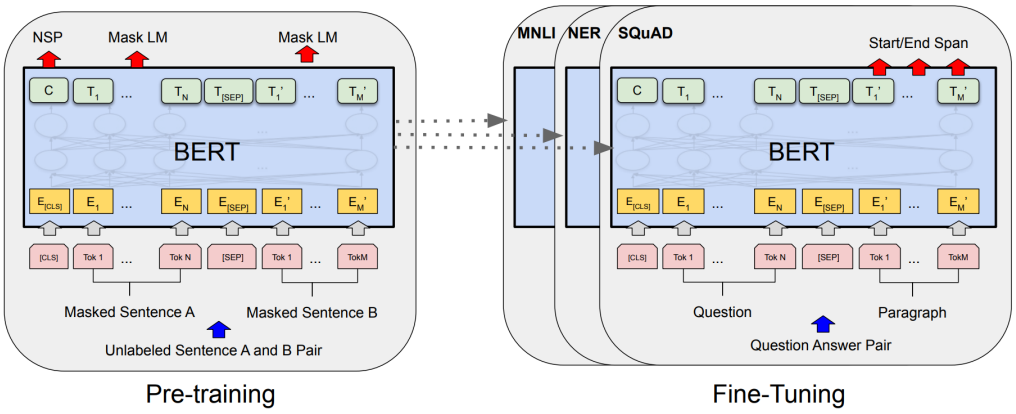

Fig. 2.3 BERT architecture[3]

BERT is pre-trained on a large corpus of text using two innovative strategies: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, some percentage of the input tokens are masked randomly, and the model is trained to predict these masked words based solely on their context. This training method allows BERT to effectively understand the context of words. NSP involves training the model to predict whether a given sentence logically follows another sentence, which helps BERT understand the relationships between sentences.

The pre-training process involves large datasets and requires significant computational power, but the result is a model that has a profound understanding of language structure and context. After pre-training, BERT can be fine-tuned with additional layers to perform a wide range of language tasks, such as sentiment analysis, question answering, and language translation.

Another significant feature of BERT, and particularly relevant to our project, is its ability to handle cross-lingual embeddings. BERT has been extended to multilingual models like BERT Multilingual (mBERT) and XLM, which are pre-trained on text from multiple languages. These models can generate embeddings that capture the semantic nuances across different languages, making them highly effective for tasks involving multiple languages. For our project, we utilize BERT's capability to generate cross-lingual embeddings. By extracting word or subword embeddings from BERT, we can compare and find the closest matches between words in different languages.

Chapter 3

Implementation

In current chapter, we turn our focus to the practical application of the theories and methodologies discussed earlier. This chapter is dedicated to the actual implementation of the Levenshtein Distance algorithm and multilingual BERT in enhancing audio transcription capabilities on a website. The implementation process is not just about writing code; it involves setting up a suitable development environment, carefully designing the system architecture, and meticulously integrating each component to work seamlessly together.

First, we will explore the development environment and tools that were chosen for this project. This includes a detailed description of the software and hardware used, along with the programming languages, frameworks, and libraries that were integral to the development process. The choice of these tools and technologies is crucial, as they form the foundation upon which our application is built.

Following this, we will delve into the system architecture and design. This section will provide a clear picture of how the various components of our system interact and work together. Diagrams and flowcharts will be used to aid in illustrating the structure and flow of the system, making it easier to understand the overall design.

The next part of this chapter will cover the implementation of the Levenshtein Distance algorithm. Here, we will walk through the steps taken to integrate this algorithm into our system, including any modifications or optimizations that were necessary to meet our specific requirements.

We will also discuss the implementation of multilingual BERT. This section will detail how BERT was incorporated into our project, highlighting any challenges faced in handling multiple languages and the solutions we employed to address these challenges.

Lastly, we will talk about the challenges encountered during the implementation phase and the strategies used to overcome them. This part is crucial as it not only reflects the problem-solving skills employed but also provides insights into the practical difficulties faced during the application of theoretical concepts.

Through this chapter, we aim to provide a comprehensive and clear account of how the project was brought to life, from the initial setup to the final implementation stages.

3.1 Development Environment and Tools

The development of this project was carried out using a specific set of hardware and software tools, each chosen for its suitability to the tasks at hand. This section details these tools and the rationale behind their selection.

3.1.1 Hardware

The project was developed on a personal computer provided by the laboratory of the Department of Electronics and Telecommunications (DET) at Politecnico di Torino. This hardware choice was primarily due to its availability and compatibility with the required software tools and frameworks.

3.1.2 Software Environment

Given the nature of the project and building upon the foundation laid by previous work, a Linux-based development environment was deemed most appropriate. Specifically, Ubuntu 20.04.5 LTS was selected. This version of Ubuntu offers a stable and well-supported platform, crucial for development and testing. Its widespread use in similar projects ensures compatibility and ease of troubleshooting.

3.1.3 Deployment

The deployment of the project is planned to be on a web server at Politecnico di Torino. This decision is based on previous work, ensuring continuity and leveraging existing infrastructure and expertise.

3.1.4 Programming Language

Python 3.6 was chosen as the programming language. This decision aligns with the language used in previous related work, ensuring consistency and the ability to integrate with existing codebases. Python's extensive support for scientific computing and natural language processing makes it an ideal choice for this project.

3.1.5 Framework

Django, a high-level Python web framework, was used. Its selection was influenced by its use in previous work, ensuring compatibility and a shorter learning curve. More information about its strength and its special pattern in development is already described in section 2.1.

3.1.6 Libraries

The libraries used in previous work were included to maintain consistency and build upon established foundations. A complete list of these libraries is available in the 'requirements' file, which will be shown in the appendix.

- *numpy* was used for matrix calculations, a necessity when implementing the Levenshtein Distance algorithm.
- *NLTK*, a fundamental library for natural language processing tasks such as tokenization, stopword filtering, and POS tagging, was integral to the project.
- *spaCy* was chosen for more advanced natural language processing tasks, particularly for multi-language support in POS tagging.

- *transformers* library was included to facilitate the use of BERT (Bidirectional Encoder Representations from Transformers) in the project.

3.1.7 Other Tools

- *pyenv* was used for Python version management. This tool allows for easy switching between different Python versions, ensuring compatibility and ease of testing across various environments. *pyenv* is widely recognized for its effectiveness in managing multiple Python versions. More information about this tool can be found at Simple Python Version Management: pyenv[5].
- *Microsoft Azure Translator AI Service* was utilized for translation and word alignment tasks. This service was chosen for its advanced capabilities and reliability. However, the project's design is such that it can accommodate other services, whether local or web-based, for similar purposes. More information about this service can be found at Translator Languages Method - Azure AI services[6].
- *Pyreverse* is a tool contained in Pylint to analyze the source code and generates package and class diagrams. More information can be found at Pyreverse - Pylint 3.1.0-dev0 documentation[7].
- *code2flow* is a tool for generating call graphs for the source code. More information can be found at code2flow: Pretty good call graphs for dynamic languages[8].

3.2 System Architecture and Design

The system architecture of *Ti Racconto Una Storia* is both intricate and multifaceted. This complexity arises from the website's long-standing usage and the continuous evolution of its features and functionalities over the years. The system's overall architecture can be visualized through the following two diagrams: the schema of the system architecture 3.1 and the database 3.2. These diagrams serve as a visual representation, offering a bird's-eye view of the system's structure and the interrelationships between different components. However, due to the comprehensive

nature of the existing system, these figures might not fully capture the intricate details and nuances of the architecture. Therefore, To provide a clear understanding of our project's place within this architecture, we will focus specifically on the components and interactions that are directly relevant to our implementation.

As we mentioned before, our work is designed for the processing, alignment and translations of audio recording transcriptions. This architecture encompasses a variety of data types, including text and audio, and integrates advanced language processing and translation services. The system's design can be broadly categorized into several key components: system workflow, data models, transcription correction and alignment, translation and word-level alignment, and integration with external services.

3.2.1 System workflow

The system's workflow is a sequence of processes, beginning with the user's finishing an interview, as mentioned in 1.1.2. Upon audio upload, the system initiates an automatic transcription process, leveraging advanced speech recognition technology to convert the spoken content into text. This initial transcription is a fundamental step, which includes previous people's effort.

Once the automatic transcription is complete, the user is granted access to the transcribed text. At this stage, the user can review and edit the transcription to ensure accuracy and coherence, a step that underscores the system's recognition of the nuanced nature of human language and the potential limitations of automated transcription.

Following the user's review and editing, if the user requires translation of the transcribed content, the system then engages its translation module. This module is designed to translate the edited transcription into the desired target language, maintaining the fidelity of the original content while adapting it to the linguistic and cultural nuances of the target language.

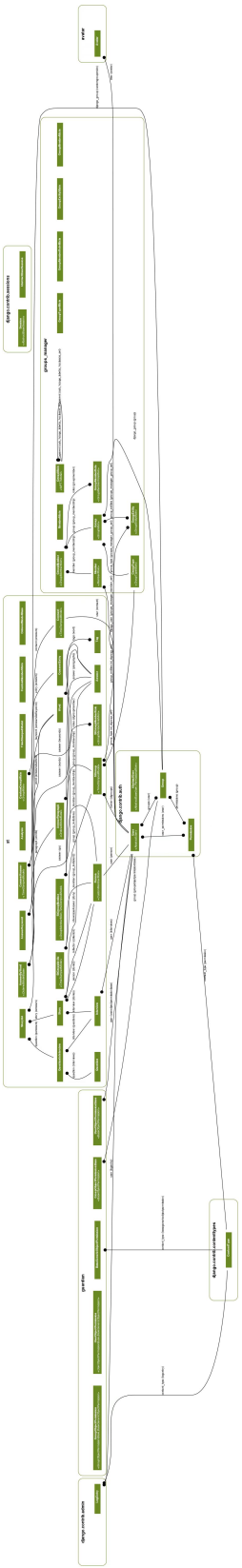

Fig. 3.1 Schema of system architecture

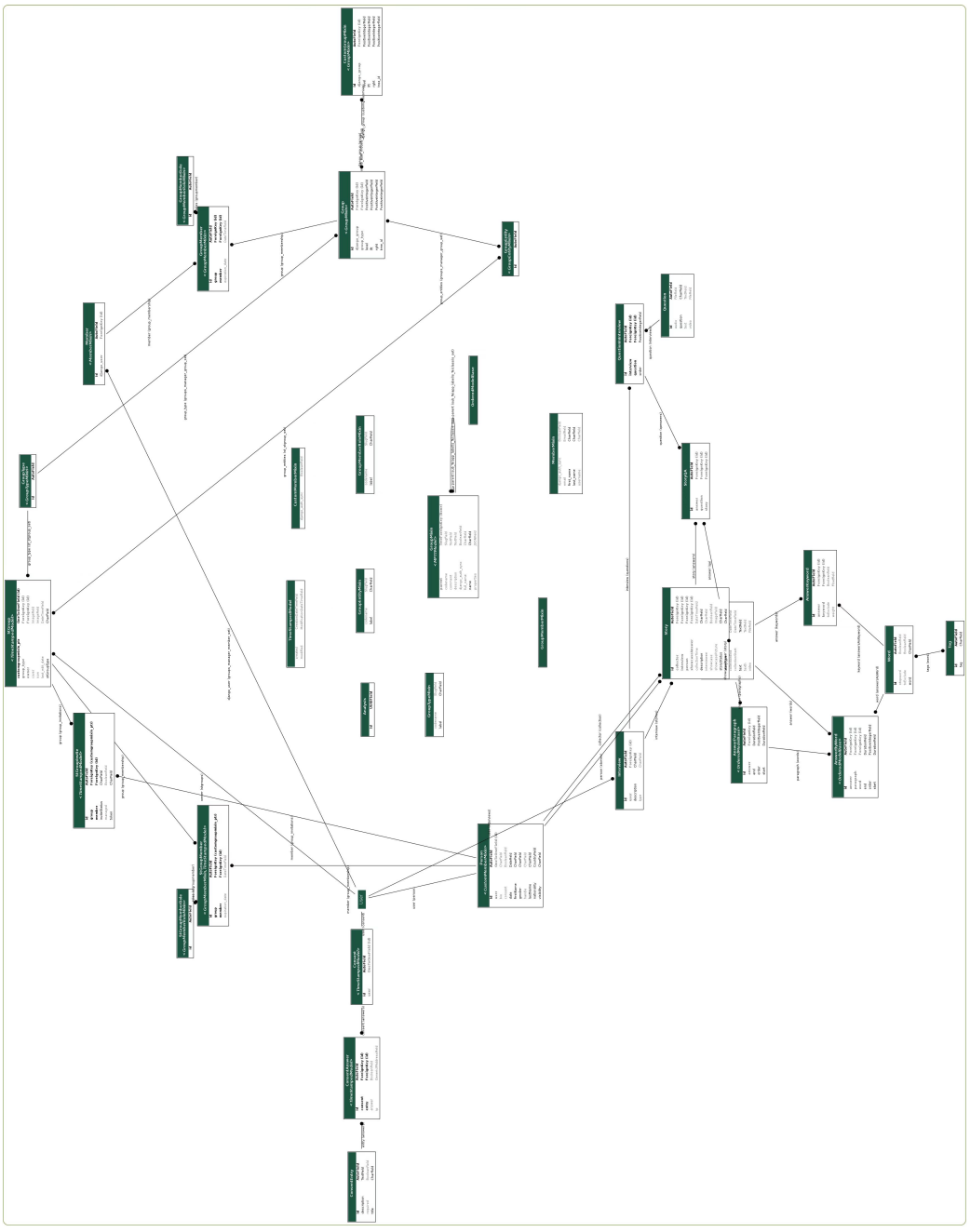

Fig. 3.2 Schema of database

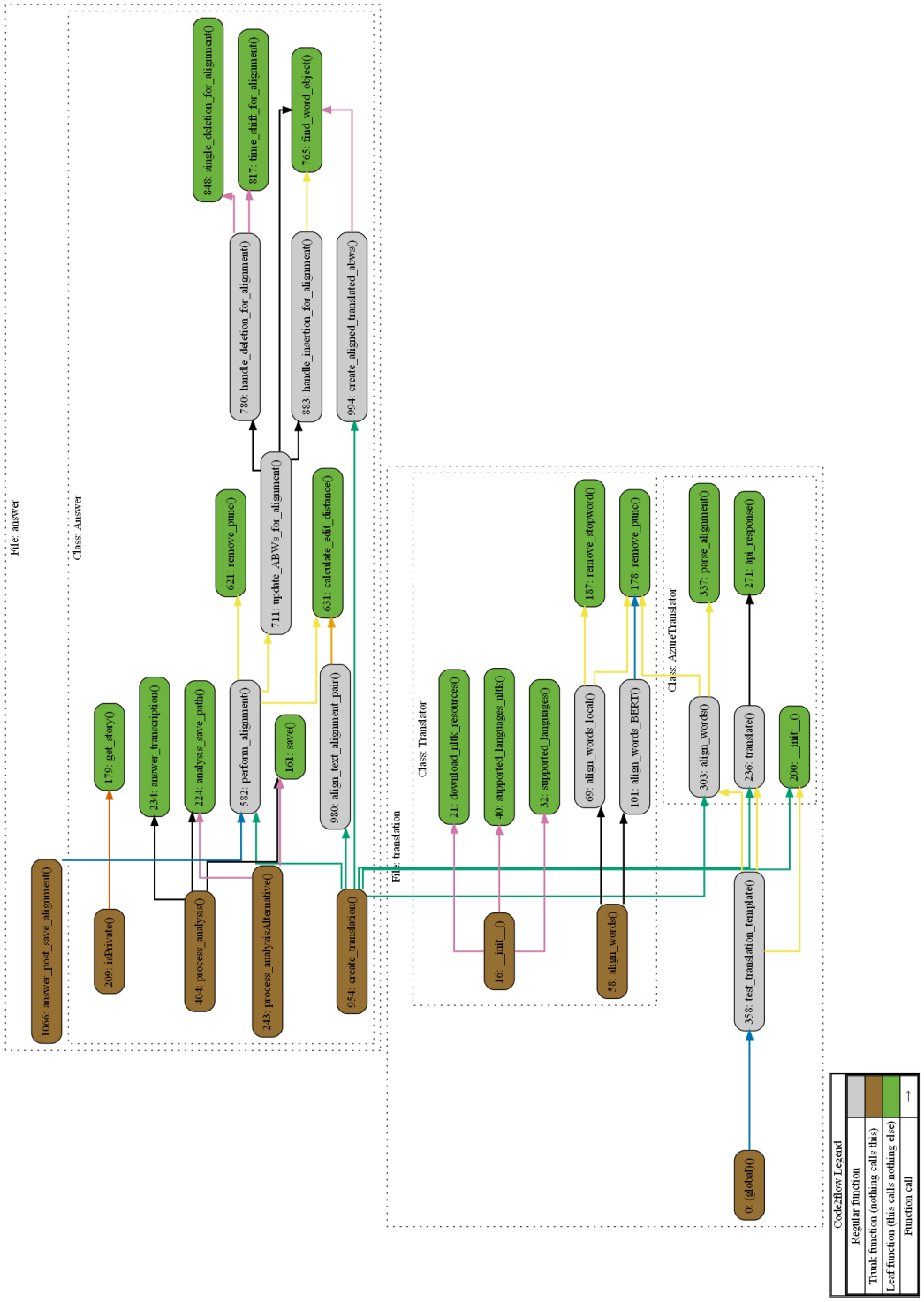

Fig. 3.3 System workflow in our work

3.2.2 Data models

The data models, in our work primarily the *Answer* and *AnswerByWord* models, are foundational to the system's functionality. They are intricately designed structures that facilitate the nuanced handling of transcriptions and translations of audio recordings. Both transcription correction and translation tasks work around these two models.

The *Answer* model is central to the system's architecture. It is designed to encapsulate the transcription or translation of an audio recording. This model includes fields for the text, which represents the transcribed or translated content, and edited text, which allows for the accommodation of revisions or modifications to the original transcription. Additionally, the model includes fields for associated media files, thereby establishing a direct link between the textual content and the corresponding audio or image files. This linkage is crucial for applications that require synchronization between text and media, such as subtitle generation or interactive language learning tools.

Complementing the Answer model is the *AnswerByWord* model, which represents a more granular approach to language processing. This model is designed to store individual words from an *Answer*, along with their corresponding start and end times in the recording. This level of detail is critical for precise alignment and synchronization with the audio content. By breaking down the transcription or translation into individual words, the system can accurately align text with specific segments of the audio, a feature that is invaluable in applications such as automated subtitle generation or detailed linguistic analysis.

The AnswerByWord model also facilitates advanced processing techniques, such as word-level alignment and edit distance calculation. By having access to the timing of each word, the system is able to perform sophisticated operations like aligning the edited transcription with the original at a word level.

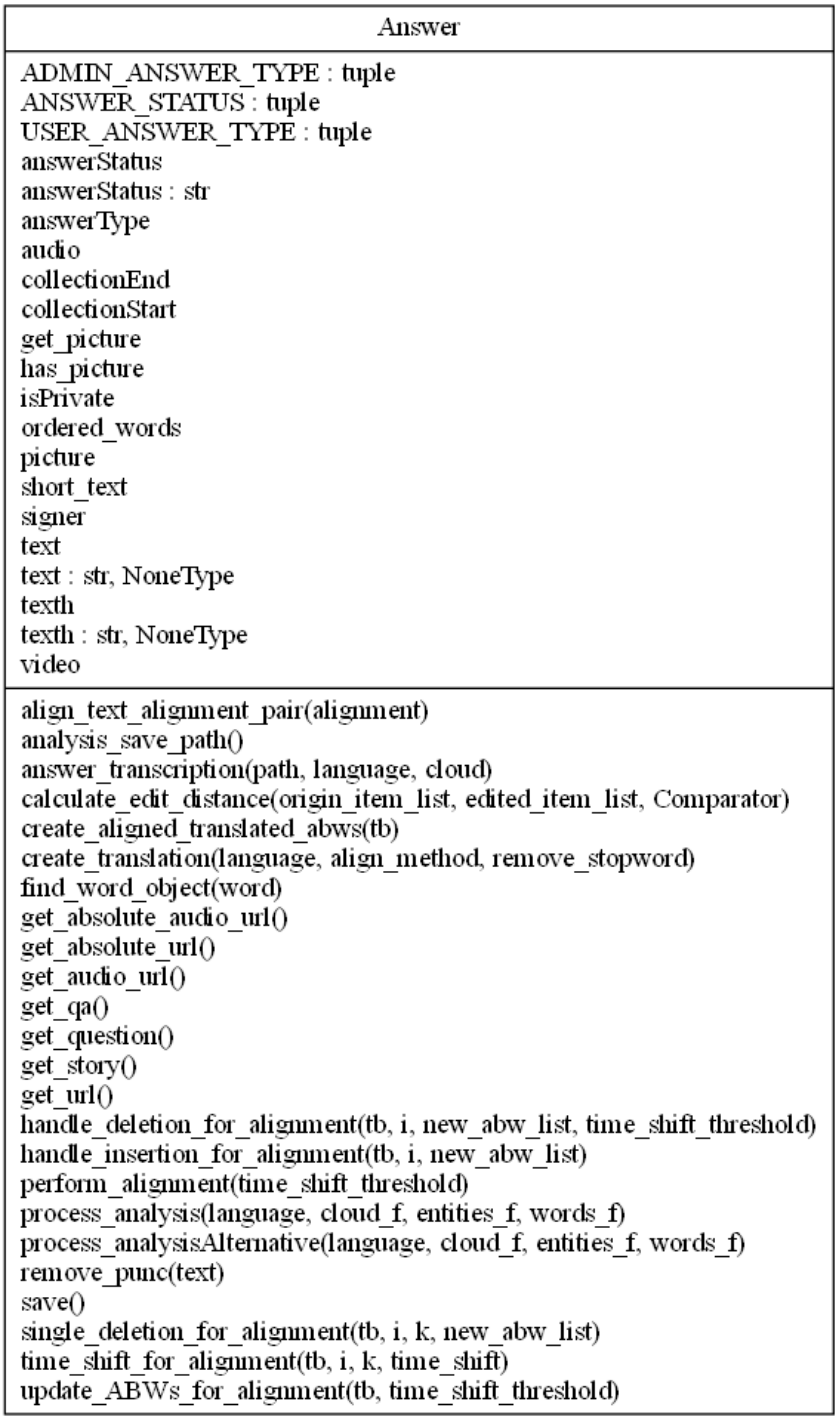

Fig. 3.4 Data structure and methods - *Answer*

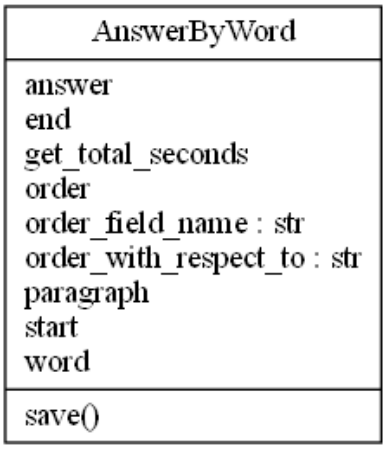

Fig. 3.5 Data structure and methods - *AnswerByWord*

3.3 Transcription correction

3.3.1 Implementation of Levenshtein Distance

As mentioned before again and again, the Levenshtein Distance is an algorithm that quantifies the minimum number of single-character edits required, i.e. insertions, deletions, or substitutions (sometimes the operations of insertion and deletion are also collectively referred to as a gap), to change one sequence (usually string) into the other and indicates how we can achieve it, intuitively aligning with our goal of minimizing the operations needed for enabling user to correct the transcription.

An extremely common example is the minimum edit distance between *kitten* and *sitting* should be 3. Levenshtein Distance will tell us both the minimum edit distance and how we are able to reach it:

1. **k**itten → **s**itten (substitution from k to s)
2. sitt**e**n → sitt**i**n (substitution from e to i)
3. sittin → sittin**g** (insertion of g)

To solve the problem of calculating edit distance, brute force enumeration is impractical, because the number of possible alignments becomes astronomically

large for even fairly short sequences, with a time complexity of $O(3^{min(m,n)})$ where m and n are lengths of the compared sequences, which can be considered as $O(e^n)$. The Levenshtein Distance algorithm provides an efficient solution using dynamic programming, with a time complexity of $O(m \cdot n)$.

Dynamic programming is a method by which a larger problem may be solved by first solving smaller, partial versions of the problem. We demonstrate here how it may be applied to global sequence alignment, where at first we are interested only in the similarity of two sequences, and not the alignment that yields this score.

How to find the minimum edit distance

Let us first agree on some definitions:

Symbol	Definition
X	one sequence to be compared with length m, $X = x_1x_2...x_m$
Y	other sequence to be compared with length n, $Y = y_1y_2...y_n$
X_i	the partial sequence consisting of the first i letters of X
Y_j	the partial sequence consisting of the first j letters of Y
$D(i,j)$	the Levenshtein Distance of X_i and Y_j
g	the gap cost for aligning any letter to a null
$s(a,b)$	the substitution cost for aligning letters a and b

Table 3.1 Definitions in our implementation of Levenshtein Distance

Consider the last item (it may be the last word or character based on the sequence is a sentence or a general string) of an optimal alignment of X_i and Y_j. This column either aligns x_i to y_j, or x_i to a null, or y_j to a null. Because we only consider insertion, deletion or substitution, there are no other possibilities. This observation yields the following recurrence:

The Levenshtein Distance between two strings, a and b (of length i and j respectively), is given by $D(i,j)$. The distance is calculated using the following recursive formula:

$$D(i,j) = \min \begin{cases} D(i-1,j)+g & x_i \text{ aligned with a null} \\ D(i,j-1)+g & y_j \text{ aligned with a null} \\ D(i-1,j-1)+s(x_i,y_i) & x_i \text{ and } y_j \text{ aligned} \end{cases} \tag{3.1}$$

where

$$s(x_i,y_j) = \begin{cases} 0 & \text{if } x_i = y_j \\ s & \text{if } x_i \neq y_j \end{cases} \tag{3.2}$$

and generally $g = s = 1$. In our case where the cost of either a gap or a substitution is a single editing operation on *AnswerByWord*, $g = s = 1$ makes sense.

What formula 3.1 means is that if the last characters of both strings are the same, the distance is the same as that for both strings without their last character. Otherwise, the distance is 1 plus the minimum of:

- The distance between X_i without its last item and Y_j (insertion)
- The distance between X_i and Y_j without its last item (deletion)
- The distance between X_i and Y_j, both without their last items (substitution)

Formula 3.1 acts as the **recursive formula** for dynamic programming. This is the idea of decomposing the problem into optimal sub-problems in dynamic programming.

Besides the recursive case, another basic element of dynamic programming is the **base case**. In the Levenshtein Distance algorithm, if either sequence is empty, the distance should be the length of the other sequence. This is intuitive in the sense that if the user decides to delete all the transcription, the operations should all be deletions and the number of operations is the length of original transcription, and vice versa. Formally,

$$D(i,j) = \begin{cases} i \cdot g & \text{if } j = 0 \\ j \cdot g & \text{if } i = 0 \end{cases} \tag{3.3}$$

How to reach the minimum edit distance

The recursive approach can be highly inefficient due to the repeated calculation of the same sub-problems. A more efficient approach is to use a memorization technique, where results of sub-problems are stored in a traceback route matrix and reused. This common method changes dynamic programming from a top-down recursive pattern to a bottom-up iterative pattern. It uses the idea of trading space for time. It sacrifices a small amount of space used to store matrices in order to reduce the time spent on recursion and repeated calculations.

In the realm of computational linguistics and string processing, the Levenshtein Distance algorithm, particularly its implementation in our project, employs a two-dimensional memorization matrix dp. The dimensions of the matrix are chosen to accommodate the lengths of the two input strings X and Y. Specifically, the matrix is constructed with dimensions $(m+1) \times (n+1)$. This configuration ensures that $dp[i][j]$ represents the minimum edit distance between the first i items of X and the first j items of Y.

The first row and column of the matrix are populated with indices. This step represents the base cases of the algorithm. The first row is initialized with indices ranging from 0 to n, that is the length of Y, and the first column is similarly initialized from 0 to m. These values signify the edit distance from an empty string to the respective sub-strings of X and Y.

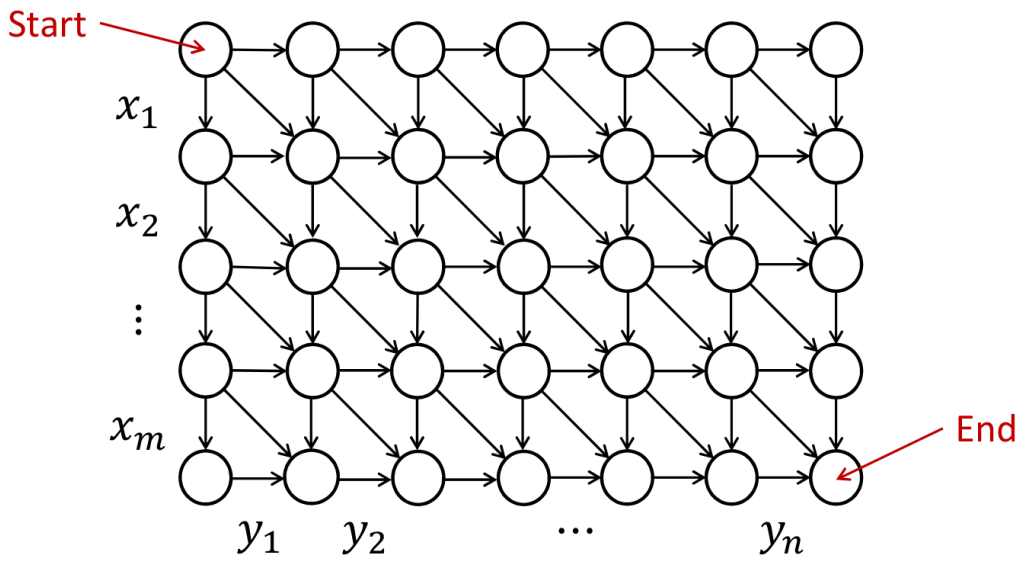

Fig. 3.6 Schema on the Levenshtein Distance algorithm

The core of the method lies in iteratively filling up the matrix. The process of populating the matrix is a systematic exploration of all possible transformations from one sub-string to another, culminating in the transformation of the entire string X into Y. This process involves iterating over each cell of the matrix and determining the minimum number of edit operations required to reach that state.

Iterative Computation: For each cell $dp(i, j)$ in the matrix, the algorithm computes the cost of three potential edit operations: insertion, deletion, and substitution. These operations are evaluated based on their respective neighboring cells:

- Deletion: The cell immediately above $dp(i-1, j)$ represents the scenario where a character from string X is deleted. The cost of this operation is derived from the value of this cell, incremented by one.
- Insertion: The cell immediately to the left $dp(i, j-1)$ symbolizes the insertion of a character into string X. Similar to deletion, the cost is the value of this cell plus one.
- Substitution: The cell diagonally above and to the left $dp(i-1, j-1)$ corresponds to the substitution of an item in string X with an item in string Y. The cost is the value of this cell incremented by one, only if the characters in strings X and Y at positions $i-1$ and $j-1$, respectively, are different.

Minimum Edit Distance Calculation: The algorithm then selects the minimum value among these three computed costs, based on formula 3.1: insertion ($dp[i][j-1]+1$), deletion ($dp[i-1][j]+1$), and substitution ($dp[i-1][j-1]+s(X_i,Y_j)$). This value represents the minimum number of edit operations required to transform the sub-string X_i into Y_j. The calculated minimum cost is then assigned to $dp[i][j]$. It records the minimum number of edits required for the specific subproblem, thereby facilitating the solution of larger problems building on these results.

Upon completion of filling in all the cells, the method returns the value stored in $dp[m][n]$. This value, located at the bottom-right corner of the matrix, represents the minimum edit distance between the entire strings X and Y. It is the culmination of the method's systematic exploration of all possible edit operations and their cumulative costs.

In addition to the dp matrix, a traceback matrix is used. This matrix stores information about which operation (insertion, deletion, substitution) was performed in each step. It helps in tracing back the exact operations (insert, delete, substitute) that were performed to transform string X into string Y. By backtracking from the bottom-right corner of the matrix to the top-left, one can reconstruct the sequence of edits.

The traceback matrix is also constructed of size $(m+1)\times(n+1)$. The first column and row are filled with indicators of deletions and insertions, based on the base case of the Levenshtein Distance, representing transforming a string into an empty string or vice versa.

While filling the DP matrix, the method also populates the traceback matrix. Each cell in this matrix stores the operation (insert, delete, substitute) that resulted in the minimum cost for that cell. Once the matrices are filled, the method backtracks from $dp[m][n]$. This backtracking follows the pointers in the traceback matrix, moving in the opposite direction of the edits (from target string back to source string). This process reconstructs the sequence of edits. As the method backtracks, it can record the specific edits made. This can be used to provide a detailed description of how to transform string X into string Y.

Let's use *kitten* and *sitting* as an example again. Running our implementation, the calculated minimum edit distance, trackback route and the dynamic programming matrix is as follows:

```
['k', 'i', 't', 't', 'e', 'n']
['s', 'i', 't', 't', 'i', 'n', 'g']
Levenshtein distance: 3 / 7
Trackback route:
[('replace', 'k', 's'),
 ('same', 'i', 'i'),
 ('same', 't', 't'),
 ('same', 't', 't'),
 ('replace', 'e', 'i'),
 ('same', 'n', 'n'),
 ('insert', None, 'g')]
```

(a) Levenshtein distance and trackback

```
DP Matrix:
      s  i  t  t  i  n  g
   0  1  2  3  4  5  6  7
k  1  1  2  3  4  5  6  7
i  2  2  1  2  3  4  5  6
t  3  3  2  1  2  3  4  5
t  4  4  3  2  1  2  3  4
e  5  5  4  3  2  2  3  4
n  6  6  5  4  3  3  2  3
```

(b) DP matrix

Fig. 3.7 *kitten* and *sitting* as an example

3.3.2 Workflow

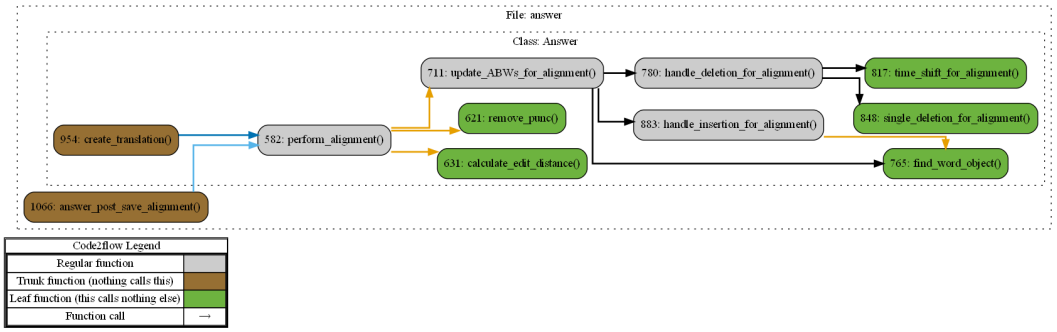

Fig. 3.8 Call function graph of performing alignment

Figure 3.8 is the call function graph of this part of work, i.e. enabling transcription correction. As shown in the image, the *Answer* class is our model that encapsulates

the concept of an answer within *Ti Racconto Una Storia*. It is not merely a data container; it embodies a suite of methods that are pivotal for processing and aligning transcriptions of audio recordings.

The *perform_alignment* method is a cornerstone of the transcription correction functionality. It aligns the edited transcription, stored in the *texth* field, with the original transcription. This method begins by ensuring that the edited transcription is not empty. It then proceeds to preprocess the edited transcription by removing punctuation and converting it to lowercase, a step that normalizes the data for further processing. The method employs NLTK for tokenization, which splits the transcription into a list of words. This method is trigged when an *Answer* object is saved with the help of *answer_post_save_alignment* signal.

Subsequently, the method retrieves the original words associated with the answer, represented as *AnswerByWord* objects, and orders them by their start time. This ordered list of words forms the basis for the alignment process. The core of this method is the invocation of *calculate_edit_distance*, which implements our methods described in 3.3.1. The method concludes by updating the *AnswerByWord* objects to reflect this alignment, using the *update_ABWs_for_alignment* method.

The *remove_punc* method is a utility function used to preprocess transcriptions. It replaces punctuation with spaces and trims leading and trailing spaces. This preprocessing is crucial for normalizing the text, ensuring that the alignment algorithm focuses on the words themselves rather than punctuation.

The *find_word_object* method also serves a utility function, facilitating the retrieval or creation of *Word* objects. Given a word as a string, it searches for an existing *Word* object in the database that matches the string. If such an object does not exist, it creates a new one. This method is instrumental in handling words that are newly introduced into the transcription during the editing process, ensuring that every unique word is accounted for in the database.

The *calculate_edit_distance* method, as mentioned before, implements our methods described in 3.3.1. It is worth mentioning that our method optionally accepts a *comparator* function, allowing for customized comparison logic between words. This flexibility is particularly useful in handling different languages or specific comparison rules. We will see this later again and again.

The *update_ABWs_for_alignment* method processes the trackback route obtained from the *calculate_edit_distance* method. It iterates through the route, handling each operation (same, replace, delete, insert) accordingly to transform the original transcription into the edited version. For each operation, the method invokes specific handling procedures. In cases of "replace" operations, it updates the corresponding *AnswerByWord* object with the new word. For "delete" and "insert" operations, it delegates to the *handle_deletion_for_alignment* and *handle_insertion_for_alignment* methods, respectively. This method ensures that each word in the transcription is accurately represented post-editing, maintaining the integrity of the data.

The *handle_deletion_for_alignment* method manages the deletion operations within the transcription alignment process. It identifies continuous stretches of deletions and decides whether to perform a time shift or handle each deletion individually. If the duration of the deletion exceeds a specified threshold (*time_shift_threshold*), a time shift is applied to all subsequent words, adjusting their start and end times accordingly. This method is for maintaining temporal accuracy in the transcription, especially when substantial portions of the original text are removed.

The *time_shift_for_alignment* method is invoked when a significant portion of the transcription is deleted, necessitating a time shift in the subsequent words. It adjusts the start and end times of all *AnswerByWord* objects following the deletion. There is a switch to decide whether this method will be invoked, and usually it is set to False by default.

In contrast to *time_shift_for_alignment*, the *single_deletion_for_alignment* method is called when deletions in the transcription are not extensive enough to warrant a time shift. It handles deletions on a word-by-word basis, adjusting the timing of the words immediately preceding and following the deleted word to evenly distribute the resultant gap. This method is essential for ensuring that minor edits in the transcription do not disrupt the overall timing of the text.

The *handle_insertion_for_alignment* method manages the insertion of new words into the transcription. It calculates the appropriate time offsets for these insertions, ensuring that the new words are seamlessly integrated into the existing sequence of *AnswerByWord* objects. This involves adjusting the timing of surrounding words to accommodate the inserted words, thereby preserving the temporal coherence of the transcription.

Upon the cases of deletion and insertion, we need to consider some boundary cases, such as deleting or inserting the first or last word. We conducted a lot of experiments to ensure that our code would run correctly and reasonably in these boundary conditions.

3.4 Translation and word-level alignment

This section delves into the implementation of multilingual support and word-level alignment in the transcription service of the website *Ti Racconto Una Storia*. The primary focus is on the integration of web service and multilingual BERT to facilitate the translation of audio transcriptions into multiple languages. The system architecture is built upon two main modules: *Answer* and *Translator*.

Answer, as mentioned in the previous section 3.2.2, represents an answer in the interview format of the answers shared on the platform. In this part of work, the crucial method in this model is *create_translation*, which is responsible for generating a translated version of the answer's transcription. This method leverages the *AzureTranslator* class defined in *Translator* module to perform the actual translation and alignment tasks.

The *Translator* module encapsulates the translation logic. It defines a base *Translator* class and a specific implementation *AzureTranslator*, which uses Microsoft Azure's translation services. The *AzureTranslator* class extends the functionality of the base class by integrating Azure's API for language translation and alignment. The base *Translator* class also includes methods for word-level alignment, which can use a local BERT-based alignment in case the online service is unavailable. This ensures that we can always provide stable services to users.

The system's operation begins with the invocation of the *create_translation* method on an *Answer* instance. This method calls the Azure translation web service to translate the edited transcription into the desired language. It then proceeds to align the translated text with the original at a word level, using either the Azure service or local BERT-based alignment, depending on the specified parameters. The aligned words are then used to create new *AnswerByWord* instances, linking translated words to their corresponding time offsets in the original answer.

3.4.1 Workflow

In this section, we will first explain the overall workflow to ensure that the explanation is logically coherent. Then, we will explain the use of Microsoft Azure Translator AI Service and multilingual BERT in our project in a sequence.

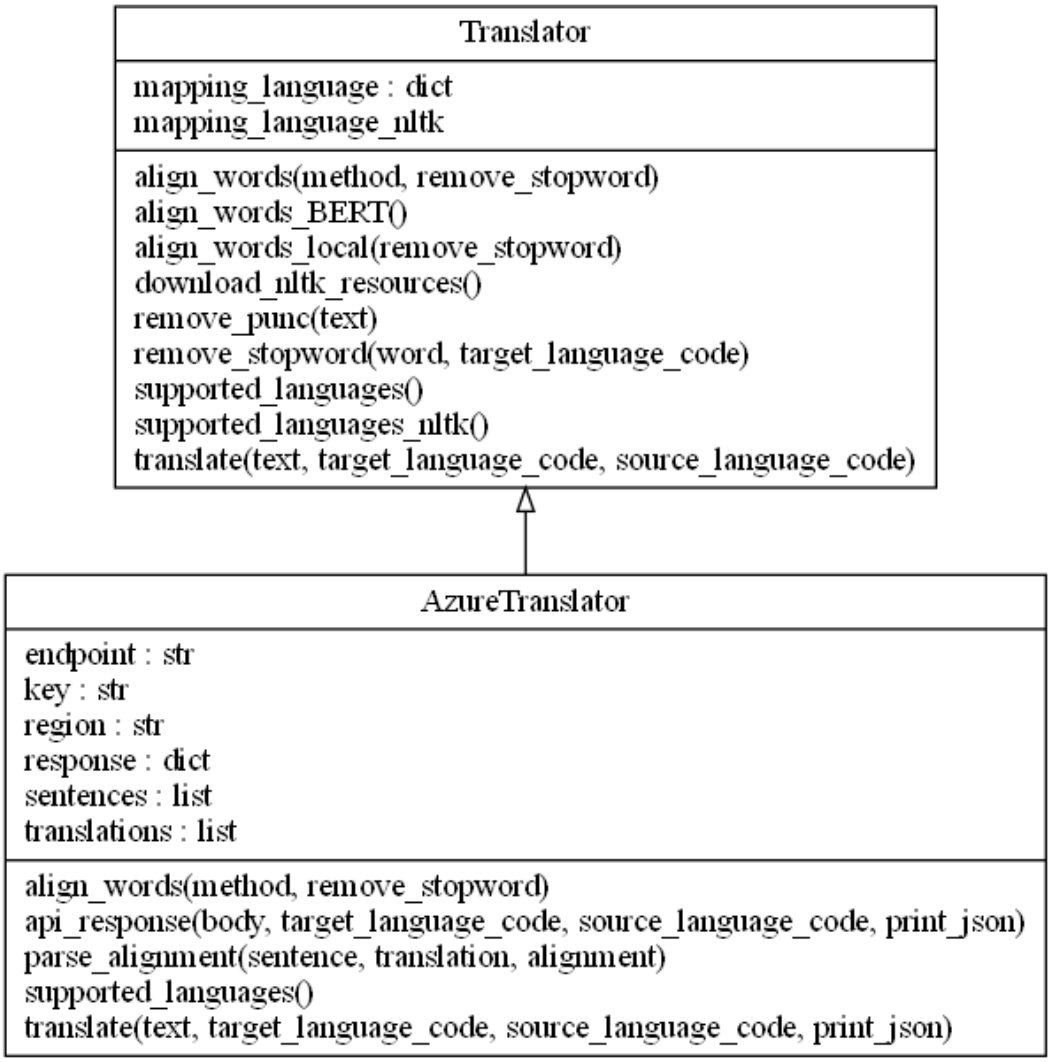

Fig. 3.9 UML diagram of *Translator* class

Translator **base class**

The *Translator* class is designed as a generic base class, providing a template for various translation services. It encapsulates common functionalities needed for

language translation and processing, setting a foundation that can be extended by specific translation service implementations.

In the constructor (*__init__*), the *Translator* class begins by calling some utility functions, *download_nltk_resources*, *supported_languages* and *supported_languages_nltk*, to prepare the necessary NLTK resources.

The *download_nltk_resources* method ensures that essential NLTK resources are available. It checks for the presence of *stopwords* and *punkt* tokenizer datasets, downloading them only if they are missing to avoid the waste on time and computing resources. And through *supported_languages* and *supported_languages_nltk*, the class establishes a mapping between language codes and names. This mapping is essential for aligning the language support between the translation service and the NLTK's capabilities, especially for stopwords processing in various languages. However we have to say that creating this mapping is actually an act of desperation because some NLTK services only accept full name of languages as parameters while most online services accept language codes following ISO 639-1[9].

The *translate* and *align_words* methods are declared as abstract, indicating that any subclass of *Translator* needs to provide concrete implementations for these methods. This design enforces a consistent interface for translation and alignment across different translation services. *align_words_local* and *align_word_BERT* are used as alternatives in case the network service is interrupted, or in future Azure no longer provides word alignment services, or the translation service selected by future developers does not come with word alignment information.

The *Translator* class demonstrates a well-structured approach to implementing translation services in an object-oriented manner. It provides a generic template so for the easy integration of additional translation services in the future.

AzureTranslator class

The *AzureTranslator* class extends the *Translator* class, providing a specific implementation that utilizes Microsoft Azure AI Translator service for translation and alignment. By inheriting from the *Translator* class, *AzureTranslator* gains all the generic functionalities of its parent class. This includes the initialization process, NLTK resource management, and the basic structure for translation and alignment methods. In its constructor, *AzureTranslator* adds additional steps specific to Azure's

translation services. It initializes variables such as the API key, endpoint, and region, which are essential for accessing Azure's translation API.

The *supported_languages* method is overridden to fetch and return a dictionary of languages supported specifically by Azure's translation service.

The *translate* method in *AzureTranslator* is a concrete implementation of the abstract method from *Translator*. It sends a request to Azure's translation API and processes the response. The *api_response* method is the core utility function tailored to Azure's API specifications. It constructs and sends a request to the Azure translation service and handles the response, including the management of headers and query parameters. It will be explained in detail in the following section 3.4.2.

In *AzureTranslator*, the *align_words* method is specifically designed to process the alignment information provided by Azure. If Azure's phrase alignment is available, it is used; otherwise, the method falls back to the local alignment method defined in the base *Translator* class.

Detailed workflow triggered by *Answer*

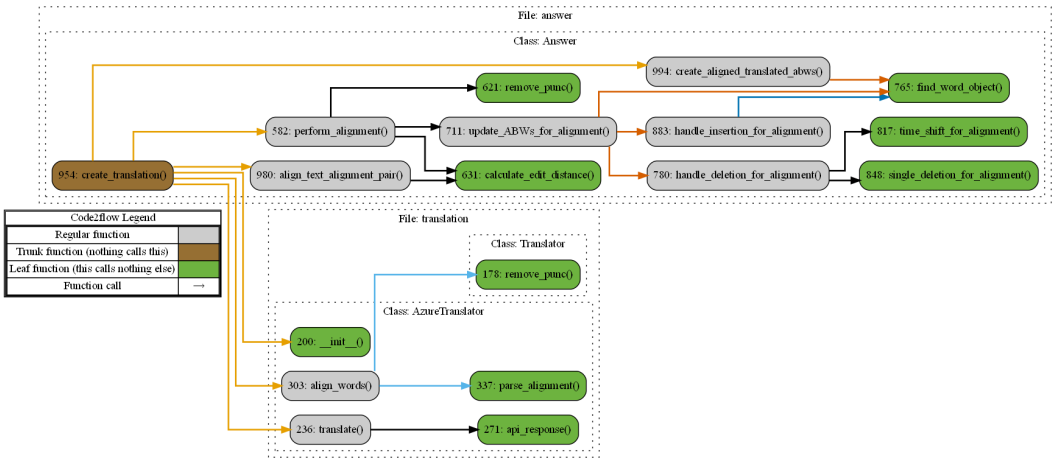

Fig. 3.10 Call function graph of *Answer* and *Translator* module

The workflow begins when *create_translation* is called on an Answer instance. This method serves as the entry point for the translation and alignment process. An instance of *AzureTranslator* is created, leveraging the capabilities of Microsoft

Azure's Cognitive Services. This instance is responsible for handling the translation of the edited transcription text into the desired target language.

The *translate* method of the *AzureTranslator* instance is invoked with the text of the Answer and the target language code. This method sends a request to Azure's translation API, which returns the translated text. The method also processes the response to extract and store sentences and their translations. Along with the translated text, the *AzureTranslator* also retrieves word-level alignment information.

Following the translation and alignment, the workflow proceeds to create a new *Answer* instance that holds the translated text. A new instance of the *Answer* model is created, copying the attributes of the original *Answer* but replacing the text with the translated text. This new instance represents the translated version of the original answer.

The *align_text_alignment_pair* method in the *Answer* model is then called. This method processes the alignment information obtained from the *AzureTranslator*. It involves a detailed comparison of each word in the original transcription with the aligned translated words, creating a mapping between them.

The next step involves creating new *AnswerByWord* instances that link the translated words to their corresponding time offsets in the original answer. The *create_aligned_translated_abws* method iterates over the alignment mappings obtained from the previous step. For each pair of aligned words, this method performs several operations. For each aligned word pair, a new *AnswerByWord* instance is created. This instance links the translated word to the same time offsets as the original word in the *Answer*.

Finally, the workflow concludes with additional alignment adjustments to ensure the accuracy and synchronization of the translated content. The *answer_post_save_alignment* receiver function is automatically triggered after the new *Answer* instance is saved. This function is designed to perform further alignment adjustments based on time shifts.

The *perform_alignment* method within the *Answer* model is called by the receiver function. This method fine-tunes the alignment, taking into account any time shifts that might have occurred during the translation process. It ensures that the translated text remains accurately synchronized with the original audio. What will happen after

calling *perform_alignment* is totally the same with section 3.3.2. The reuse of code shows our efforts on system schema design.

3.4.2 Usage of Microsoft Azure Translator AI Service

We choose to use Microsoft Azure Translation Service [10] because, after conducting research, we found that Azure is the only cloud provider of mainstream online translation services that offers word alignment information between the translated and original text, in addition to basic translation services. During our investigation, we also discovered that Yandex provides non-official word alignment services, although in a concealed manner. However, considering the reliability, legitimacy, and availability of services in Europe, we still choose Microsoft Azure Translation Service as our default translation service provider. It's worth noting that both Azure and Yandex's word alignment services are experimental and may be removed in the future [11]. To address this issue and to prepare for any potential network issues that may cause service disruptions, we have also developed a local BERT-based word alignment strategy, which will be discussed in detail in the next section.

The tutorial about creating a Translator resource in the Azure portal can be found on the official website Create a Translator resource - Azure AI Services[12].

We fetch Azure AI Translator service through RESTful API. Although Microsoft provides Text Translation SDK for Python, we still think RESTful API is more powerful and more stable to use.

In our work the method *api_response* is the core utility function that communicates with the Azure Translator API. The method begins by constructing the URL for the Azure Translator API endpoint. This URL is composed using the base endpoint stored in the *AzureTranslator* instance and appended with the specific path for the translation service (/translate). The method then prepares the parameters and headers required for the API request. These parameters include the target language code, the API version, and an option to include alignment data in the response. The headers primarily consist of the subscription key and region, which are essential for authenticating the request with Azure's service.

The body of the request is prepared by tokenizing the input text into sentences using NLTK's sentence tokenizer. Each sentence is then wrapped in a dictionary structure, forming the payload for the POST request. This structuring is in accordance

with Azure's API requirements, where each text segment to be translated is provided as a separate item in the request body.

The method proceeds to send the POST request to the Azure API with the constructed URL, parameters, headers, and body. The response from the API is then captured and parsed. This response contains the translated text along with alignment data.

Our method also includes robust error handling to manage scenarios where the API request fails or returns an error. In such cases, the method ensures that the process does not halt abruptly, instead providing graceful degradation by returning an empty translation and alignment. In case that translation service works as expected but alignment does not, the method falls back to the local alignment method defined in the base *Translator* class.

3.4.3 Inference of Multilingual BERT

In our project, *align_words_BERT* is invoked as alternatives in case the network service is not available, or in future Azure no longer provides word alignment services, or the translation service selected by future developers does not come with word alignment information. In this specific method, *PyTorch* and *transformers* packages are additionally required in order to enable the capabilities of BERT.

The alignment layer and a threshold value are predetermined. The choice of the 8th layer is grounded in the hypothesis that intermediate layers of BERT models capture a good balance between word-level and context-level representations. In the first layers the contextualization is too weak for high-quality alignments while the last layers are too specialized on the pretraining task (masked language modeling). [13][14][15] In our work finetune on BERT is not needed since BERT already gain enough generic linguistic comprehension capabilities during pretraining phase.

The model is set to evaluation mode, a standard practice in inference tasks to ensure consistency in the model's output by disabling layers like dropout that are only relevant during training. During our experiments, we find that GPU is unnecessary since CPU only is enough for the inference.

The script processes both source and target sentences through mBERT, extracting the hidden states from the specified alignment layer. This step is crucial as it

retrieves the contextual embeddings of each token in both languages. As mentioned in the previous chapter 2.3.2, BERT generates context-aware embeddings instead of traditional static embeddings as NLTK, Word2Vec or GloVe do.

A dot product is computed between the embeddings of the source and target tokens. This mathematical operation is instrumental in quantifying the similarity between token pairs across the two languages. Subsequently, a softmax function is applied to these dot product values, transforming them into probabilities. This step is executed twice, each time focusing on a different direction (source-to-target and target-to-source), thereby ensuring a bidirectional perspective in alignment.

Then we employ a threshold to filter out alignments with lower confidence levels, ensuring that only alignments with a probability exceeding the threshold are considered.

The final step involves mapping the identified subword alignments back to the original words. BERT tokenizes input text into subwords, and the alignment needs to be interpreted at the word level for practical utility.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

Our project has successfully addressed and fulfilled its two primary objectives mentioned at the beginning 1.3 of this thesis, enhancing the *Ti Racconto Una Storia* platform both in terms of transcription accuracy and linguistic inclusivity. Through meticulous development and integration of advanced methodologies within the Django framework, this research has significantly elevated the user experience and broadened the platform's appeal and accessibility.

The first objective, focusing on empowering storytellers to correct inaccuracies in machine-generated transcriptions, has been adeptly achieved. The implementation of the Levenshtein Distance algorithm stands as a testament to this accomplishment. By allowing storytellers to edit transcriptions while maintaining accurate alignment with audio timestamps, the platform ensures that the essence and authenticity of each story are preserved. This feature not only enhances the integrity of the storytelling process but also respects the storyteller's ownership and connection to their narrative. The technical sophistication of this solution, particularly in maintaining coherence between spoken words and their written representation, significantly enhances the listening and reading experience for all users.

The second objective, aimed at transcending language barriers through the translation of audio transcriptions, has been realized with remarkable efficacy. The development of a robust translation mechanism, capable of not just linguistic conversion but also precise word-level alignment with the original audio, marks a significant

advancement over conventional sentence-level alignment methods. This granular approach to translation ensures that the rhythm, timing, and context of the original narrative are faithfully maintained in multiple languages. By offering a spectrum of word alignment methods, from local mechanics to sophisticated BERT-based techniques, this thesis caters to a diverse range of needs and scenarios. This enhancement greatly extends the reach of audio narratives, fostering greater understanding and engagement across different languages and cultures.

4.2 Future work

While this thesis has made significant strides in enhancing the *Ti Racconto Una Storia* platform, there are several avenues for future research and development that could further augment its capabilities.

Drawing from experiences in bioinformatics, where data is often highly structured and regular, there is potential to apply more sophisticated data structures and more advanced algorithms to improve text processing and alignment. The regularity and structure observed in bioinformatics can inspire new approaches to handling textual data, potentially leading to more efficient and accurate transcription and translation processes. Exploring these advanced methodologies could lead to breakthroughs in how we handle complex narrative structures, making the platform more robust and versatile.

The current implementation leverages BERT for its powerful capabilities in language understanding and translation. However, we are now in the era of Large Language Models, which offer even more advanced linguistic processing capabilities. While current LLMs present challenges in terms of stability and predictability – factors that are crucial for our service – the evolving landscape of these models holds great promise. It is anticipated that the underlying logic and mechanisms of LLMs will become more transparent and reliable in the future. Once this occurs, integrating these models into our platform could significantly enhance our translation and alignment accuracy, offering a more nuanced and context-aware approach to multilingual storytelling.

Another area of potential development is the incorporation of adaptive learning mechanisms. These mechanisms could enable the platform to learn from each

transcription and translation, improving accuracy and efficiency over time. This self-improving system could adapt to various dialects, idiomatic expressions, and unique narrative styles, further personalizing the user experience.

References

[1] Luciano Lavagno. Home | ti racconto una storia. https://tiraccontounastoria.org/, 2023.

[2] Patricio Astudillo. Django data flow diagrams - stack overflow. https://stackoverflow.com/questions/9628885/django-data-flow-diagrams, 2023.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] Thibault Debatty. Java string similarity. https://github.com/tdebatty/java-string-similarity, 2022.

[5] pyenv. pyenv: Simple python version management. https://github.com/pyenv/pyenv, 2023.

[6] Microsoft. Translator languages method - azure ai services. https://learn.microsoft.com/en-us/azure/ai-services/translator/reference/v3-0-languages, 2023.

[7] Pylint. Pyreverse - pylint 3.1.0-dev0 documentation. https://pylint.readthedocs.io/en/latest/pyreverse.html, 2023.

[8] Scott Rogowski. code2flow: Pretty good call graphs for dynamic languages. https://github.com/scottrogowski/code2flow, 2023.

[9] Wikipedia. List of iso 639-1 codes. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes, 2023.

[10] Microsoft. Translator translate method - azure ai services. https://learn.microsoft.com/en-us/azure/ai-services/translator/reference/v3-0-translate, 2023.

[11] Microsoft. Limitations - translator translate method - azure ai services. https://learn.microsoft.com/en-us/azure/ai-services/translator/reference/v3-0-translate#limitations, 2023.

[12] Microsoft. Create a translator resource - azure ai services. https://learn.microsoft.com/en-us/azure/ai-services/translator/create-translator-resource, 2023.

[13] Zi-Yi Dou and Graham Neubig. Word alignment by fine-tuning embeddings on parallel corpora. *arXiv preprint arXiv:2101.08231*, 2021.

[14] Masoud Jalili Sabet, Philipp Dufter, François Yvon, and Hinrich Schütze. Simalign: High quality word alignments without parallel training data using static and contextualized embeddings. *arXiv preprint arXiv:2004.08728*, 2020.

[15] John Hewitt and Christopher D. Manning. A structural probe for finding syntax in word representations. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *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 4129–4138, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.