
BOĞAZİÇİ ÜNİVERSİTESİ
1863

CMPE 492

Turkish Preprocessing Operations Using Deep Learning Approaches

Umut Deniz Şener

Koray Tekin

Advisor:

Tunga Güngör

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Broad Impact

One of this project's broad impacts is contributing to the Turkish language NLP operations, even if on a smaller scale. Preprocessing operations such as tokenization, sentence splitting, and normalization are essential for NLP and the project aims to implement these operations by using deep learning approaches. Hence, the project has indirect beneficial impacts on the development of the practical applications of NLP applications for the Turkish language such as sentiment analysis, and text classification. Consequently, there exists a potential for the project to have effects on industries like business analytics, educational applications, content moderation, customer support, etc.

Also, considering that Turkish is an underrepresented language, the project has the potential to increase awareness about developing NLP tools for this kind of language.

In addition, since Turkish like any other language has some unique features, the development of preprocessing operations creates the further possibility of penetrating the language's features such as the head-final structure of it.

To summarize, this project's broad impacts may include enhancing NLP tools for the Turkish language to raise awareness, providing linguistic insights, and serving as a foundation for potential future applications.

1.2. Ethical Considerations

This project involves the usage of large datasets of Turkish text from different sources. Considering, the secure storage and anonymity of these texts has high impor-

tance to protect the privacy of individuals whose text might be included in the dataset. Also, It is crucial to ensure that the dataset should be designed in a way that includes text instances from various concepts in order to maintain an unbiased approach; because the preprocessing methods represent different aspects of the Turkish language and cultural domain. Moreover, it has possible negative effects on society by abusing the developed tools such as disinformation campaigns, or harmful content generation. It is important to consider these effects and restrict users to prevent undesirable consequences.

2. PROJECT DEFINITION AND PLANNING

2.1. Project Definition

This project's aim is to develop and implement preprocessing operations for the Turkish language using deep learning approaches. The preprocessing operations include tokenization (segmenting the text into tokens), sentence splitting (dividing the text into sentences), deasciifier (the process of converting "English versions" (c, g, i, o, s, u, C, G, I, O, S, U) of Turkish letters (ç, ğ, ı, ö, ş, ü, Ç, Ğ, İ, Ö, Ş, Ü, respectively) within the words into their correct forms). The project will start with a literature review to analyze similar systems for English, such as UDPipe and Stanza. Then, for the different steps of preprocessing, deep learning models will be built and trained with the Turkish corpus, taking into account the unique characteristics of the language. Finally, the system will be tested again on Turkish corpora within the Universal Dependencies framework.

2.2. Project Planning

2.2.1. Project Time and Resource Estimation

The project is estimated to take approximately four months to complete, with the following:

- Literature review and analysis of similar systems: 2 weeks
- Developing deep learning models for preprocessing operations and adapting models to the Turkish language: 2 months
- Testing and evaluation on Turkish corpora: 1 month
- Documentation and finalizing the project: 2 weeks

Resources required for the project include:

- Access to relevant research articles and publications
- Computing resources for training and testing deep learning models
- Access to Turkish corpora and datasets
- NLP and deep learning libraries and frameworks (e.g., TensorFlow, PyTorch, NLTK, etc.)

2.2.2. Success Criteria

The success of the project can be evaluated based on the following criteria:

- Completion of a literature review and analysis of similar systems
- Successful development and adaptation of deep learning models for preprocessing Turkish text
- Performance of the developed system on Turkish corpora, compared to existing benchmarks
- Clear documentation and presentation of the project, including methodology, results, and potential applications

2.2.3. Risk Analysis

Potential risks for this project include:

(i) Insufficient access to relevant research articles and publications
(ii) Limited availability and performance of cooperative computing resources such as Google Colaboratory
(iii) Long training times of the neural network models
(iv) Difficulties in adapting deep learning models to complex Turkish language characteristics
(v) Incomplete or biased Turkish corpora and datasets

2.2.4. Team Work (if applicable)

The following tasks can be divided among team members but in our case, we proceeded collaboratively:

- Literature review and analysis of similar systems
- Developing deep learning models for preprocessing operations
- Adapting models to Turkish language characteristics
- Testing and evaluation on Turkish corpora
- Documentation and presentation of the project

Regular meetings are scheduled among team members and the project advisor for effective communication and collaboration.

3. RELATED WORK

During our exploration of the literature, we examined various universal preprocessing operations implemented using deep learning approaches, such as UDPipe and Stanza. In addition, we analyzed different approaches for individual operations, including tokenization and sentence splitting. Furthermore, we researched libraries and model functionalities that could be utilized for our project.

3.1. Universal Preprocessing Tools

- **UDPipe**: Straka et al. (2016) introduced a trainable pipeline for processing CoNLL-U files, performing tokenization, morphological analysis, POS tagging, and parsing [1].
- **Stanza**: Qi et al. (2020) presented a Python NLP toolkit for multiple human languages, offering various preprocessing functionalities [2].

3.2. Individual Preprocessing Operations

- **Tokenization**: Doyle et al. (2019) developed a character-level LSTM network model for tokenizing Old Irish text in the Würzburg Glosses on the Pauline Epistles [3].
- **Sentence Splitting**: Sheik et al. (2022) proposed an efficient deep learning approach for sentence boundary detection in legal texts, which was presented at the Natural Legal Language Processing Workshop [4].

3.3. Resources and Frameworks

- **Preprocessing Techniques in NLP**: A comprehensive guide to preprocessing techniques in NLP can be found at `https://exchange.scale.com/public/blogs/preprocessing-techniques-in-nlp-a-guide`.

- **Universal Dependencies**: The Universal Dependencies framework provides a platform for cross-linguistic, consistent grammatical annotation and is available at `https://universaldependencies.org/`.

REFERENCES

1. Straka, M., Hajic, J., Strakova, J. (2016) UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing, In Proc. of the Tenth International Conference on Language Resources and Evaluation (LREC), p.4290-4297.

2. Qi, P., Zhang, Y., Zhang, Y., Bolton, J., Manning, C.D. (2020) Stanza: A Python Natural Language Processing Toolkit for Many Human Languages, In. Proc. of the 58th Annual Meeting of the Association for Computational Linguistics, p.101-108.

3. Doyle, A., McCrae, J. P., & Downey, C. (2019). A Character-Level LSTM Network Model for Tokenizing the Old Irish Text of the Würzburg Glosses on the Pauline Epistles. In Proceedings of the Celtic Language Technology Workshop 2019 Dublin, 19–23 Aug. 2019 (pp. 70-79).

4. Sheik, R., Adethya, G.T., & Nirmala, S.J. (2022). Efficient Deep Learning-based Sentence Boundary Detection in Legal Text. In Proceedings of the Natural Legal Language Processing Workshop 2022 (pp. 208-217). Association for Computational Linguistics.

4. METHODOLOGY

4.1. Tokenizer

We have designed and implemented a deep learning-based tokenizer for Turkish text. The process is divided into several steps:

4.1.1. Data Loading and Preprocessing

(i) Load the ConLL-U formatted data, extract sentences and corresponding tokenized sentences from it.
(ii) Use a character-level tokenizer to convert the sentences into sequences of characters.
(iii) Determine the vocabulary size and the maximum sequence length in the training data.
(iv) Pad the sequences to the maximum sequence length.

4.1.2. Label Creation

(i) For each sentence, identify the boundaries of the tokens in the corresponding tokenized sentence.
(ii) Create binary labels for the characters in the sentence, where a '1' indicates the start of a token and '0' otherwise.
(iii) Pad the binary labels to match the sequence length.

4.1.3. Model Creation and Training

(i) Split the padded sequences and corresponding labels into training and testing sets.
(ii) Define a bidirectional LSTM model with two LSTM layers and an output dense

layer for token boundary predictions.

(iii) Compile the model using the Adam optimizer and binary cross-entropy loss.

(iv) Train the model using the training data and validate it with the testing data.

4.1.4. Tokenization of New Sentences

(i) Convert the input sentence into a sequence of characters using the same character-level tokenizer used during training.

(ii) Pad the sequence to match the maximum sequence length.

(iii) Use the trained model to predict token boundaries for the input sentence.

(iv) Extract tokens from the input sentence based on the predicted token boundaries.

4.2. Sentence Splitting

We have designed a sentence splitting system for Turkish text, using the deep learning approaches. This process consists of several steps:

4.2.1. Data Loading and Generation of Shuffled Texts

(i) Load the ConLL-U formatted data and extract sentences from it.

(ii) Generate shuffled texts and corresponding sentence lists using a sentence shuffling function that takes a list of sentences, group size, number of texts, and batch size as parameters.

4.2.2. Sentence Boundary Identification and Label Creation

(i) Identify sentence boundaries in the shuffled texts.

(ii) Create binary labels indicating whether a character marks the beginning of a sentence.

4.2.3. Data Preprocessing

(i) Convert the shuffled texts to sequences of characters using a character-level tokenizer.
(ii) Pad the sequences to the maximum length found in the training data.
(iii) Pad the binary labels to match the sequence length.

4.2.4. Model Creation and Training

(i) Split the sequences and labels into training and testing sets.
(ii) Define a bidirectional LSTM model with two LSTM layers and an output dense layer for sentence boundary predictions.
(iii) Compile the model using the Adam optimizer and binary cross-entropy loss.
(iv) Train the model using the training data and validate it with the testing data.

4.2.5. Sentence Splitting of New Texts

(i) Convert the input text into a sequence of characters using the same tokenizer as before.
(ii) Pad the sequence to the maximum length.
(iii) Use the trained model to predict sentence boundaries for the input text.
(iv) Split the text into sentences based on the predicted sentence boundaries.

4.3. Deasciifier Model

We have designed a deep learning model for correcting de-asciified Turkish sentences. The process involves the following steps:

4.3.1. Data Loading and Preprocessing

(i) Load the ConLL-U formatted data, extract original sentences, and convert them to the ASCII form.
(ii) Save these original and de-asciified sentence pairs in a JSON file for easy access in the future.
(iii) Define the character sets for Turkish characters and their corresponding ASCII forms.
(iv) Create a combined character set including all characters appearing in both the original and de-asciified sentences.
(v) Map all characters to unique indices and vice-versa to facilitate numerical computation.

4.3.2. Data Preparation

(i) Convert original and de-asciified sentences to sequences of corresponding character indices.
(ii) Ensure that each target character index corresponds to a Turkish character if it is altered in the de-asciified form, else it should correspond to the same ASCII character.
(iii) Pad the sequences to have the same length as the longest sentence.
(iv) Convert the target sequences to one-hot encoded form.

4.3.3. Model Creation and Training

(i) Define a Sequential model consisting of an Embedding layer, a Bidirectional LSTM layer, and a TimeDistributed Dense layer.
(ii) Compile the model using categorical cross-entropy loss and the Adam optimizer.
(iii) Train the model with a validation split of 0.1 and for a specified number of epochs.

4.3.4. Deasciification of New Sentences

(i) Convert the input sentence into a sequence of character indices using the same mapping as before.

(ii) Pad the sequence to match the maximum sequence length.

(iii) Use the trained model to predict the character indices for the input sentence.

(iv) Convert the predicted indices back to characters.

(v) Replace the de-asciified characters in the input sentence with the predicted Turkish characters.

5. REQUIREMENTS SPECIFICATION

5.0.1. Functional Requirements

(i) Conduct a literature review on existing systems for English, such as UDPipe and Stanza.

(ii) Develop deep learning models for each preprocessing operation (tokenization, sentence splitting, deasciifier).

(iii) Adapt models to Turkish language characteristics, e.g., using embeddings for suffixes.

(iv) Test the system on Turkish corpora, preferably on Turkish treebanks in the Universal Dependencies (UD) framework.

5.0.2. System Features

- Tokenization
- Sentence splitting
- Deasciifier

5.0.3. Algorithms

(i) Tokenization
  - Bi-directional LSTM (Bi-LSTM) model

(ii) Sentence Splitting
  - Bi-directional LSTM (Bi-LSTM) model

(iii) Deasciifier
  - Bi-directional LSTM (Bi-LSTM) model

6. DESIGN

6.1. Information Structure

Not applicable

6.2. Information Flow

Not applicable

6.3. System Design

Not applicable

6.4. User Interface Design (if applicable)

Not applicable

7. IMPLEMENTATION AND TESTING

7.1. Implementation

7.1.1. Tokenizer Implementation

We have implemented a deep learning-based approach for tokenization using a Bi-LSTM model. The code snippet below explains the implementation process step by step.

(i) Import necessary libraries: numpy, tensorflow, re, and sklearn.
(ii) Define a function to load and preprocess the conllu data.
(iii) Load sentences from the conllu file and preprocess them for the Bi-LSTM model.
(iv) Create a character-level tokenizer, fit it on the preprocessed sentences, and create sequences.
(v) Determine the maximum sequence length and pad the sequences to ensure uniform length.
(vi) Generate labels for the sequences based on punctuation marks(will be improved by using train data to determine the token boundaries) and spaces.
(vii) Split the data into training and test sets using an 80-20 split.
(viii) Define a function to create the Bi-LSTM model with appropriate layers and configurations.
(ix) Create and summarize the model.
(x) Train the model using the training data and validate it on the test data.

The Bi-LSTM model we implemented is capable of identifying token boundaries using punctuation marks and spaces, making it a suitable approach for tokenization tasks. The model can be further improved by fine-tuning its architecture and training parameters.

Related python code

```
import numpy as np
import tensorflow as tf
import re
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Bidirectional,
                                    LSTM, Dense
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split

# Load conllu data and preprocess
def load_conllu_data(file_path):
    with open(file_path, "r", encoding="utf-8") as f:
        data = f.read()
    sentences = data.split("\n\n")
    sentences = [s.split("\n") for s in sentences if len(s) > 0]
    return [" ".join([word.split("\t")[1] for word in s if not word.
                                        startswith("#")]) for s in
                                        sentences]

file_path = "tr_boun-ud-dev.conllu"
sentences = load_conllu_data(file_path)

# Preprocess data for Bi-LSTM model with punctuation as token
                                    boundaries
def preprocess_sentences(sentences):
    punctuations = r'[,.;:!?]'
    processed_sentences = []
    for s in sentences:
        s = re.sub(r'([' + punctuations + r'])', r' \1 ', s)
        processed_sentences.append(s)
    return processed_sentences

processed_sentences = preprocess_sentences(sentences)
tokenizer = Tokenizer(filters="", lower=False, char_level=True)
tokenizer.fit_on_texts(processed_sentences)
```

```
sequences = tokenizer.texts_to_sequences(processed_sentences)
vocab_size = len(tokenizer.word_index) + 1

max_length = max([len(seq) for seq in sequences])
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding
                                 ="post")

punctuations = r'[,.;:!?]'
labels = []
for s in processed_sentences:
    label = [0]
    for i, c in enumerate(s[:-1]):
        if c in punctuations or s[i + 1] in punctuations or s[i + 1] =
                                             = ' ':
            label.append(1)
        else:
            label.append(0)
    labels.append(label)
padded_labels = pad_sequences(labels, maxlen=max_length, padding="post
                                 ")


# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(padded_sequences,
                                 padded_labels, test_size=0.2,
                                 random_state=42)

# Define a more complex Bi-LSTM model
def create_bilstm_model(vocab_size, max_length):
    inputs = Input(shape=(max_length,))
    x = Embedding(vocab_size, 128, input_length=max_length)(inputs)
    x = Bidirectional(LSTM(128, return_sequences=True))(x)
    x = Bidirectional(LSTM(64, return_sequences=True))(x)
    outputs = Dense(1, activation="sigmoid")(x)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer="adam", loss="binary_crossentropy",
                                     metrics=["accuracy"])
```

```
    return model

model = create_bilstm_model(vocab_size, max_length)
model.summary()

# Train the model
model.fit(X_train, y_train, batch_size=32, epochs=5, verbose=1,
                                    validation_data=(X_test, y_test))
```

7.1.2. Sentence Splitter Implementation

We implemented a deep learning-based sentence splitter using a Bi-LSTM model. Below, we provide a step-by-step walkthrough of the implementation process:

(i) Imported the necessary libraries: conllu, numpy, tensorflow, and sklearn.
(ii) Defined a function to load and preprocess data from the conllu file.
(iii) Loaded and preprocessed sentences from the conllu file.
(iv) Developed a function that generates shuffled texts and their corresponding groups of sentences.
(v) Generated shuffled texts and their associated groups of sentences.
(vi) Identified sentence boundaries for each shuffled text.
(vii) Created binary labels to denote the start of each sentence in the shuffled text.
(viii) Set up a character-level tokenizer and fit it on the original sentences. Also, padded all sequences to the same length.
(ix) Split the data into training and testing sets, maintaining an 80-20 split.
(x) Defined a function to create a Bi-LSTM model with the appropriate layers and configurations.
(xi) Created and summarized the model.
(xii) Trained the model on the training data and validated it using the test data.
(xiii) Saved the trained model for future use.
(xiv) Implemented a function to tokenize a given sentence using the trained model.

Related python code

```
!pip install conllu
import numpy as np
import tensorflow as tf
import re
import conllu
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Bidirectional,
                                    LSTM, Dense
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
from sklearn.utils import class_weight
import random
def load_conllu_data(file_path):
    with open(file_path, "r", encoding="utf-8") as f:
        data = f.read().strip()

    all_sentences = []
    for item in conllu.parse(data):
        all_sentences.append(item.metadata['text'])

    return all_sentences

def shuffled_text_generator(sentences, group_size=5, num_texts=2000,
                                    batch_size=1000):
    if num_texts is None:
        num_texts = len(sentences) // group_size

    num_batches = num_texts // batch_size

    for _ in range(num_batches):
        random.shuffle(sentences)
        grouped_sentences = [sentences[i:i+group_size] for i in range(
                                            0, len(sentences),
                                            group_size)]
```

```
        texts_and_groups = [(group, " ".join(group)) for group in
                                            grouped_sentences]

        # Yield a batch of shuffled texts and groups
        for text, group in texts_and_groups[:batch_size]:
            yield text, group

file_path = "tr_boun-ud-train.conllu"
all_sentences = load_conllu_data(file_path)
shuffled_texts_gen = shuffled_text_generator(all_sentences)

shuffled_texts = []
splitted_sentences_list = []
for splitted_sentences, shuffled_text in shuffled_texts_gen:
    shuffled_texts.append(shuffled_text)
    splitted_sentences_list.append(splitted_sentences)

print(len(shuffled_texts))
print(len(splitted_sentences_list))
def find_sentence_boundaries(text, splitted_sentences):
    boundaries = []
    start = 0
    for sentence in splitted_sentences:
        start = text.find(sentence, start)
        end = start + len(sentence) - 1
        boundaries.append((start, end))
        start = end
    return boundaries
def create_binary_labels(text, boundaries):
    binary_labels = [0] * len(text)
    for start, end in boundaries:
        binary_labels[start] = 1
    return binary_labels
tokenizer = Tokenizer(filters="", lower=False, char_level=True)
tokenizer.fit_on_texts(all_sentences)
vocab_size = len(tokenizer.word_index) + 1
```

```
sequences = tokenizer.texts_to_sequences(shuffled_texts)
max_length = max([len(seq) for seq in sequences])
padded_sequences = pad_sequences(sequences, maxlen=max_length, padding
                                 ="post")

labels = []
for i, s in enumerate(shuffled_texts):
    boundaries = find_sentence_boundaries(s, splitted_sentences_list[i
                                          ])
    label = create_binary_labels(s, boundaries)
    labels.append(label)

padded_labels = pad_sequences(labels, maxlen=max_length, padding="post
                              ")

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(padded_sequences,
                                   padded_labels, test_size=0.2,
                                   random_state=42)
# Define a more complex Bi-LSTM model
def create_bilstm_model(vocab_size, max_length):
    inputs = Input(shape=(max_length,))
    x = Embedding(vocab_size, 128, input_length=max_length)(inputs)
    x = Bidirectional(LSTM(128, return_sequences=True))(x)
    x = Bidirectional(LSTM(64, return_sequences=True))(x)
    outputs = Dense(1, activation="sigmoid")(x)
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer="adam", loss="binary_crossentropy",
                                    metrics=["accuracy"])
    return model

model = create_bilstm_model(vocab_size, max_length)
model.summary()

model.fit(X_train, y_train, batch_size=32, epochs=6, verbose=1,
                                   validation_data=(X_test, y_test))
model.save("sentence_splitter.h5")
```

```
def tokenize_sentence(sentence, model, tokenizer, max_length):
    seq = tokenizer.texts_to_sequences([sentence])[0]
    padded_seq = pad_sequences([seq], maxlen=max_length, padding="post
                                        ")
    predictions = model.predict(padded_seq)[0]
    predictions = predictions[:len(sentence)]
    print(len(predictions), len(sentence))
    for i, pred in enumerate(predictions[:len(sentence)]):
      print(sentence[i], pred)
    #predictions = np.round(predictions[:len(sentence)]).flatten()
    tokens = []
    current_token = ""

    for i, c in enumerate(sentence):

        if predictions[i] > 0.5:
          tokens.append(current_token)
          current_token = c
        else:
          current_token += c
        if i == len(sentence)-1:
            tokens.append(current_token)
    return [s.strip() for s in tokens]
input_sentence = "Umut su al. Neden Ahmet kahve almak gibi bir eylemde
                                     bulundun? Kahve almaya gidelim.
                                    Nereye gidiyorsun?"
tokens = tokenize_sentence(input_sentence, model, tokenizer,
                                    max_length)
print("Tokens:", tokens)
```

7.1.3. Deasciifier Implementation

We implemented a deasciifier using another Bi-LSTM model. Below is a step-by-step walkthrough of the implementation:

(i) Imported necessary libraries: conllu, numpy, and keras.

(ii) Defined a function to convert Turkish characters to ASCII characters.

(iii) Loaded data from the conllu file, converted each sentence to its ASCII form, and saved the original and ASCII versions as pairs in a JSON file.

(iv) Defined the character set and created dictionaries for character-to-index and index-to-character mappings.

(v) Prepared the input and target data for the model. For each pair of original and ASCII sentences, the ASCII sentence was used as input, and the target was the original sentence.

(vi) Padded all input and target sequences to the same length and one-hot encoded the target data.

(vii) Defined and compiled a Bi-LSTM model suitable for the task.

(viii) Trained the model using the input and target data.

(ix) For prediction, preprocessed the input sentence and fed it to the model. The model predicted the probability distribution for each character in the sentence.

(x) Converted these probabilities back into characters, and replaced the ASCII characters in the input sentence with the predicted characters to get the deasciified sentence.

Through this implementation, the deasciifier can transform text containing ASCII characters into its original Turkish form, and the sentence splitter can correctly identify the boundaries of sentences within a text. Both models can be improved by fine-tuning and by using more diverse training data.

Related python code

```
!pip install conllu
import conllu
import json
import numpy as np
from keras.utils import to_categorical
from keras.utils import pad_sequences
from keras.models import Sequential
```

```
from keras.layers import LSTM, Dense,TimeDistributed, Bidirectional,
                                    Embedding
def turkish_to_ascii(text):
    turkish_chars = "" # Can't include in the Project report, Latex
                                        format causes error.
    ascii_chars = "gGuUsSiIoOcC"
    translation_table = str.maketrans(turkish_chars, ascii_chars)
    return text.translate(translation_table)

file_path = "tr_boun-ud-train.conllu"

with open(file_path, "r", encoding="utf-8") as f:
    data = f.read().strip()
    all_sentences = []
    deasciified_sentences = []
    for i, item in enumerate(conllu.parse(data)):
        original_sentence = item.metadata['text']
        all_sentences.append(original_sentence)
        deasciified_sentence = turkish_to_ascii(original_sentence)
        deasciified_sentences.append(deasciified_sentence)

# Save the original and deasciified sentences to a JSON file
sentence_pairs = [{"original": orig, "deasciified": deasc} for orig,
                                    deasc in zip(all_sentences,
                                    deasciified_sentences)]

with open("tr_boun-ud-train-deasciified.json", "w", encoding="utf-8")
                                    as f:
    json.dump(sentence_pairs, f, ensure_ascii=False, indent=2)
# Load sentence pairs from the JSON file
with open("tr_boun-ud-train-deasciified.json", "r", encoding="utf-8")
                                    as f:
    sentence_pairs = json.load(f)

# Define character set and create dictionaries for character-to-index
                                    and index-to-character mappings
target_chars = ""
```

```
input_chars = "gGuUsSiIoOcC"

#target_char_to_idx = {c: i for i, c in enumerate(target_chars)}

# Define character set and create dictionaries for character-to-index
                                    and index-to-character mappings
all_chars = sorted(set("".join(target_chars) + "".join(input_chars) +
                                    "".join(all_sentences) + "".join(
                                    deasciified_sentences)))
char_to_idx = {c: i for i, c in enumerate(all_chars)}
idx_to_char = {i: c for i, c in enumerate(all_chars)}
# Prepare input (X) and target (y) data
X_data = []
y_data = []

for pair in sentence_pairs:
    orig = pair["original"]
    deasc = pair["deasciified"]

    X_sentence = [char_to_idx[c] for c in deasc]
    y_sentence = []

    for orig_char, deasc_char in zip(orig, deasc):
        if deasc_char in input_chars:
            if orig_char in target_chars:
                y_sentence.append(char_to_idx[orig_char])
            else:
                y_sentence.append(char_to_idx[deasc_char])
        else:
            y_sentence.append(char_to_idx[orig_char])

    X_data.append(X_sentence)
    y_data.append(y_sentence)
print(sentence_pairs[1])
print(X_data[1], y_data[1])
# Pad sequences to the same length
max_length = max([len(s) for s in X_data])
```

```
X_data = pad_sequences(X_data, maxlen=max_length, padding="post")
y_data = pad_sequences(y_data, maxlen=max_length, padding="post")

# One-hot encode the target data
y_data = np.array([to_categorical(y, num_classes=len(all_chars)) for y
                                    in y_data])
# Define and compile the bi-LSTM model
model = Sequential()
model.add(Embedding(len(all_chars), 64, input_length=max_length))
model.add(Bidirectional(LSTM(128, return_sequences=True)))
model.add(TimeDistributed(Dense(len(all_chars), activation="softmax"))
                                    )
model.compile(loss="categorical_crossentropy", optimizer="adam",
                                    metrics=["accuracy"])

# Train the model
history = model.fit(X_data, y_data, epochs=5, batch_size=32,
                                    validation_split=0.1)
model.summary()

# Preprocess the input sentence
input_sentence = "Kahvaltiya kadar 2 saat cografya calisirim."
X_test = [char_to_idx[c] for c in input_sentence]
input_length = len(X_test)
X_test = pad_sequences([X_test], maxlen=max_length, padding="post")

# Make predictions
y_pred = model.predict(X_test)

predicted_chars = []
for p in y_pred[0][:input_length]:
    idx = np.argmax(p)
    char = idx_to_char[idx]
    predicted_chars.append(char)

    sorted_indices = np.argsort(p)[::-1]
    sorted_probs = sorted(p, reverse=True)
```

```
    sorted_chars = [idx_to_char[i] for i in sorted_indices]
    print(f"Char: {char}, Probabilities: {sorted_probs[:5]}, Chars: {
                                        sorted_chars[:5]}")
print(predicted_chars)


# Convert predictions to characters
predicted_chars = [idx_to_char[np.argmax(p)] for p in y_pred[0][:
                                    input_length]]

# Replace the original characters in the input sentence with the
                                    predicted characters
output_sentence = []
i = 0
while i < len(input_sentence):
    c = input_sentence[i]
    if c in input_chars:
        output_sentence.append(predicted_chars[i])
    else:
        output_sentence.append(c)
    i += 1

output_sentence = "".join(output_sentence)


print("Input sentence:", input_sentence)
print("Output sentence:", output_sentence)
```

7.2. Testing

7.2.1. Tokenizer

In order to evaluate the effectiveness of the tokenization model, we have conducted several tests using the test data set. We assess the performance of the model by comparing its predictions with the ground truth and calculating the F-measure. We

present two methods for calculating the F-measure: average F-measure and cumulative F-measure. The testing and evaluation process is outlined below:

(i) Implement the tokenize_sentence function, which tokenizes the input sentence using the trained model. This function converts the input sentence into a sequence of characters, pads the sequence to the maximum sequence length, and feeds it into the trained model for prediction. The predicted boundaries are then used to split the sentence into tokens.

(ii) Implement the f_measure function to calculate the F-measure score given the true and predicted tokens of a sentence. This function computes the precision and recall of the predictions, and then calculates the F-measure score as the harmonic mean of precision and recall.

(iii) Implement the average_f_measure function, which computes the F-measure for each sentence in the test data and returns their average. This function provides a global measure of how well the model is performing across all sentences.

(iv) Implement the cumulative_f_measure function, which computes the cumulative F-measure across all sentences in the test data. This function provides an alternative measure of the model's performance which is less sensitive to the variability in the number of tokens per sentence.

(v) Use the tokenize_sentence function to tokenize all sentences in the test data.

(vi) Evaluate the performance of the model using both the average_f_measure and cumulative_f_measure functions.

```
def tokenize_sentence(sentence, model, tokenizer, max_length):
    seq = tokenizer.texts_to_sequences([sentence])[0]
    padded_seq = pad_sequences([seq], maxlen=max_length, padding="post
                                        ")
    predictions = model.predict(padded_seq)[0]
    predictions = np.round(predictions[:len(sentence)]).flatten()
    tokens = []
    current_token = ""

    for i, c in enumerate(sentence):
```

```
        if predictions[i] == 1:
          tokens.append(current_token)
          current_token = c
        else:
          current_token += c

        if i == len(sentence)-1:
            tokens.append(current_token)
    return [s.strip() for s in tokens]

def f_measure(true_tokens, predicted_tokens):
    true_positives = len(set(true_tokens) & set(predicted_tokens))
    false_positives = len(predicted_tokens) - true_positives
    false_negatives = len(true_tokens) - true_positives

    if true_positives == 0:
        return 0

    precision = true_positives / (true_positives + false_positives)
    recall = true_positives / (true_positives + false_negatives)
    f_measure_score = 2 * precision * recall / (precision + recall)

    return f_measure_score

def average_f_measure(ground_truth, predicted):
    f_measures = [f_measure(true_tokens, pred_tokens) for true_tokens,
                                          pred_tokens in zip(
                                         ground_truth, predicted)]
    return sum(f_measures) / len(f_measures)

# Tokenize test sentences
predicted_tokenization = [tokenize_sentence(sentence, model, tokenizer
                                    , max_length) for sentence in
                                    original_sentences]

# Calculate F-measure
f_measure_score = average_f_measure(tokenized_sentences,
```

```
                                    predicted_tokenization)
print(f"F-measure: {f_measure_score:.4f}")

def cumulative_f_measure(ground_truth, predicted):
    total_true_positives = 0
    total_false_positives = 0
    total_false_negatives = 0

    for true_tokens, pred_tokens in zip(ground_truth, predicted):
        true_positives = len(set(true_tokens) & set(pred_tokens))
        false_positives = len(pred_tokens) - true_positives
        false_negatives = len(true_tokens) - true_positives

        total_true_positives += true_positives
        total_false_positives += false_positives
        total_false_negatives += false_negatives

    if total_true_positives == 0:
        return 0

    precision = total_true_positives / (total_true_positives +
                                        total_false_positives)
    recall = total_true_positives / (total_true_positives +
                                     total_false_negatives)
    f_measure_score = 2 * precision * recall / (precision + recall)

    return f_measure_score

# Calculate cumulative F-measure
cumulative_f_measure_score = cumulative_f_measure(tokenized_sentences,
                                    predicted_tokenization)
print(f"Cumulative F-measure: {cumulative_f_measure_score:.4f}")
```

7.2.2. Sentence Splitter

For a more granular analysis, a sentence splitter function has been implemented and tested. This function aims to identify sentence boundaries in a paragraph of text.

The testing process is as follows:

(i) Define a function, get_predicted_splits, that predicts sentence splits in a paragraph of text. It utilizes the trained model to tokenize the paragraph and returns the predicted sentence boundaries.

(ii) Define a function, calculate_confusion_matrix, to calculate the True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN) based on the comparison between the predicted and the true sentence boundaries.

(iii) Define a function, f_measure_single_paragraph, that calculates the F-measure for a single paragraph. It gets the predicted splits for the paragraph, calculates the confusion matrix, and uses these values to calculate the precision, recall, and accuracy, which are then used to calculate the F-measure.

(iv) Define a function, f_measure_multiple_paragraphs, that calculates the F-measure for multiple paragraphs. It sums up the TP, FP, FN, and TN for all the paragraphs and calculates the overall precision, recall, and accuracy. These values are used to calculate the F-measure for all paragraphs.

The F-measure gives us an understanding of the accuracy of the sentence splitter model. It considers both precision (how many selected sentences are relevant) and recall (how many relevant sentences are selected), providing a balanced measure of its performance.

```
def get_predicted_splits(paragraph, model, tokenizer, max_length):
    predicted_splits = tokenize_sentence(paragraph, model, tokenizer,
                                         max_length)
    return predicted_splits

def calculate_confusion_matrix(true_splits, predicted_splits):
```

```
    TP = 0
    FP = 0
    FN = 0
    TN = 0

    for true_boundary in true_splits:
        if true_boundary in predicted_splits:
            TP += 1
        else:
            FN += 1

    for predicted_boundary in predicted_splits:
        if predicted_boundary not in true_splits:
            FP += 1

    # TN is the total number of non-split characters that are
                                        correctly non-split
    TN = len(predicted_splits[0]) - (FN + FP + TP)
    print(TN)
    return TP, FP, FN, TN

def f_measure_single_paragraph(paragraph, true_splits, model,
                                    tokenizer, max_length):
    predicted_splits = get_predicted_splits(paragraph, model,
                                        tokenizer, max_length)
    print(predicted_splits)
    print(true_splits)
    tp, fp, fn, tn = calculate_confusion_matrix(true_splits,
                                        predicted_splits)

    precision = tp / (tp + fp)
    recall = tp / (tp + fn)

    # Modified to include TN in the denominator
    accuracy = (tp + tn) / (tp + fp + fn + tn)
    print("TP: ", tp,"FP: ", fp,"TN: ", tn,"FN: ", fn )
```

```
    # Beta is considered to be 1 in this case
    f_measure = (2 * precision * recall) / (precision + recall)

    # Modified to include accuracy
    #f_measure = (2 * precision * recall * accuracy) / (precision +
                                        recall + accuracy)
    print(accuracy)
    print(recall)
    print(precision)
    return f_measure

def f_measure_multiple_paragraphs(paragraphs, true_splits_list, model,
                                    tokenizer, max_length):
    tp_total, fp_total, fn_total, tn_total = 0, 0, 0, 0
    for paragraph, true_splits in zip(paragraphs, true_splits_list):
        predicted_splits = get_predicted_splits(paragraph, model,
                                            tokenizer, max_length)
        tp, fp, fn, tn = calculate_confusion_matrix(true_splits,
                                            predicted_splits, paragraph
                                            )
        tp_total += tp
        fp_total += fp
        fn_total += fn
        tn_total += tn
    precision = tp_total / (tp_total + fp_total)
    recall = tp_total / (tp_total + fn_total)
    accuracy = (tp_total + tn_total) / (tp_total + fp_total + fn_total
                                        + tn_total)

    # Beta is considered to be 1 in this case
    f_measure = (2 * precision * recall) / (precision + recall)

    # Modified to include accuracy
    f_measure = (2 * precision * recall * accuracy) / (precision +
                                        recall + accuracy)

    return f_measure
```

```
print(f_measure_multiple_paragraphs(all_test_paragraphs,
                                    true_splits_list, model, tokenizer,
                                     max_length))
```

7.2.3. Deasciifier

To evaluate the performance of the deasciifier, an F-measure is calculated for all sentences in the dataset. The process is as follows:

(i) For each pair of sentences in the dataset (an original sentence and its deasciified version), a prediction is made based on the deasciified sentence.
(ii) The counts of True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN) are calculated for each pair.
(iii) These counts are summed across all sentence pairs to obtain total counts.
(iv) The precision, recall, and F-measure are then calculated using these total counts.

The F-measure provides a comprehensive measure of the performance of the deasciifier, considering both its precision and recall.

```
def calculate_f_measure_for_all_sentences(sentence_pairs, model,
                                    max_length, char_to_idx,
                                    idx_to_char):
true_positives = 0
false_positives = 0
false_negatives = 0
true_negatives = 0
for pair in sentence_pairs:
original_sentence = pair["original"]
deasciified_sentence = pair["deasciified"]
X_test = [char_to_idx[c] for c in deasciified_sentence]
input_length = len(X_test)
X_test = pad_sequences([X_test], maxlen=max_length, padding="post")
y_pred = model.predict(X_test)
```

```
predicted_chars = [idx_to_char[np.argmax(p)] for p in y_pred[0][:
                                    input_length]]
output_sentence = []
i = 0
while i < len(deasciified_sentence):
c = deasciified_sentence[i]
if c in input_chars:
output_sentence.append(predicted_chars[i])
else:
output_sentence.append(c)
i += 1
output_sentence = "".join(output_sentence)
tp, fp, fn, tn = calculate_counts(original_sentence,
                                    deasciified_sentence,
                                    output_sentence)
true_positives += tp
false_positives += fp
false_negatives += fn
true_negatives += tn
precision = true_positives / (true_positives + false_positives) if
                                    true_positives + false_positives >
                                    0 else 0
recall = true_positives / (true_positives + false_negatives) if
                                    true_positives + false_negatives >
                                    0 else 0
f_measure = 2 * precision * recall / (precision + recall) if precision
                                     + recall > 0 else 0
return f_measure

f_measure = calculate_f_measure_for_all_sentences(sentence_pairs,
                                    model, max_length, char_to_idx,
                                    idx_to_char)
print("F-measure for all sentences:", f_measure)
```

7.3. Deployment

7.3.1. Google Colab Deployment

The project is hosted on Google Colab, which provides an easy-to-use environment with pre-installed necessary libraries, such as TensorFlow. To run the project, follow these steps:

(i) Open the Google Colab notebook in your browser using the provided links below for each model.
- For the tokenizer:
`https://colab.research.google.com/drive/1rqWBmh1uhbQ1_gKlOr3BUvdr42vrFmxr?usp=sharing`
- For the sentence splitter:
`https://colab.research.google.com/drive/1etDGKxNfJY2uGg-pLk-_938BecDYhpqv?usp=sharing`
- For the deasciifier:
`https://colab.research.google.com/drive/1SDs4zGDApAMfct0zAkocfaEVXK01hCvO?usp=sharing`

(ii) Sign in to your Google account if prompted.

(iii) Upload the training data and the test data named "tr_boun-ud-dev.conllu", "tr_boun-ud-test.conllu" files to the Google Colab environment by using the file upload feature in the left-hand panel. Make sure to adjust the file paths in the code accordingly.

(iv) Run the cells in the notebook sequentially to train the corresponding model and test it with a sample instance.

8. RESULTS

8.1. Example Inputs and Outputs

8.1.1. Tokenization Example

8.1.1.1. Example Inputs. We present three sample Turkish texts for tokenization:

(i) `input1`: Yurt dışında çalışmak hedeflerim içerisinde umarım da gerçekleşir; uluslar arası öğretmenlik diploması almak istiyorum bu yüzden de.
(ii) `input2`: - Çocuklar, gördüğünüz gibi dinleneceğimiz, oynayacağımız, yemeklerimizi yiyip, yarışmalar yapacağımız yere geldik.
(iii) `input3`: Ancak bu bir teferruat gayet tabii ki.

8.1.2. Example Outputs

Using the implemented Bi-LSTM tokenizer, the sample texts are tokenized as follows:

(i) `output1`: ["Yurt", "dışında", "çalışmak", "hedeflerim", "içerisinde", "umarım", "da", "gerçekleşir", ";", "uluslar", "arası", "öğretmenlik", "diploması", "almak", "istiyorum", "bu", "yüzden", "de", "."]
(ii) `output2`: ["Çocuklar", ",", "gördüğünüz", "gibi", "dinleneceğimiz", ",", "oynayacağımız", ",", "yemeklerimizi", "yiyip", ",", "yarışmalar", "yapacağımız", "yere", "geldik", "."]
(iii) `output3`: ["Ancak", "bu", "bir", "teferruat", "gayet", "tabii", "ki", "."]

8.1.3. Sentence Splitter Example

8.1.3.1. Input. The following is a sample input to the sentence splitter:

Meğerse ne kadar yanılmışız. İnsanların ve toplulukların birbirine üstünlüğü ve egemenliği o zaman söz konusu olmaya başlar. .İstanbul Arena'da yapılacak konsere Kıraç, Haluk Levent, Bulutsuzluk Özlemi, Pentagram, Okan Karacan ve Nejat Yavaşoğulları ile birçok sanatçı katılacak. Benjamin dramından yaklaşık yirmi beş yıl sonra. Tamam, deyip görüşmeye gider.

8.1.3.2. Output. After the sentence splitting process, the sentences obtained from the above input are as follows:

(i) "Meğerse ne kadar yanılmışız."
(ii) "İnsanların ve toplulukların birbirine üstünlüğü ve egemenliği o zaman söz konusu olmaya başlar."
(iii) "İstanbul Arena'da yapılacak konsere Kıraç, Haluk Levent, Bulutsuzluk Özlemi, Pentagram, Okan Karacan ve Nejat Yavaşoğulları ile birçok sanatçı katılacak."
(iv) "Benjamin dramından yaklaşık yirmi beş yıl sonra."
(v) " Tamam, deyip görüşmeye gider."

8.1.4. Deasciifier Example

8.1.4.1. Example Inputs and Outputs. Below are some examples of deasciified Turkish sentences and their original versions:

(i) `Example 1`:
  - `input`: "Eseklerin sirtlarina yuklenmis sepetlerle tasinirdi uzumler."
  - `output`: "Eşeklerin sırtlarına yüklenmiş sepetlerle taşınırdı üzümler."

(ii) `Example 2`:
  - `input`: "Benjamin dramindan yaklasik yirmi bes yil sonra."
  - `output`: "Benjamin dramından yaklaşık yirmi beş yıl sonra."

(iii) `Example 3`:
  - `input`: "Istanbul Arena'da yapilacak konserlere cok sanatci katilacak."
  - `output`: "İstanbul Arena'da yapılacak konserlere çok sanatçı katılacak."

(iv) `Example 4`:

- `input`: "Cocuklar, gordugunuz gibi dinlenecegimiz, oynayacagimiz, yemeklerimizi yiyip, yarismalar yapacagimiz yere geldik."
- `output`: "Çocuklar, gördüğünüz gibi dinleneceğimiz, oynayacağımız, yemeklerimizi yiyip, yarışmalar yapacağımız yere geldik."

(v) `Example 5`:

- `input`: "Ancak bu bir teferruat gayet tabii ki."
- `output`: "Ancak bu bir teferruat gayet tabii ki."

9. CONCLUSION

9.1. Implementation

9.1.1. Tokenizer Implementation

We have implemented a tokenizer using a bi-directional LSTM (Bi-LSTM) model. The model has been trained on the Turkish treebanks in the Universal Dependencies (UD) framework. The tokenizer utilizes a character-level tokenization approach, considering punctuation as token boundaries.

9.1.2. Sentence Splitter Implementation

We have implemented a Sentence Splitter using a Bi-LSTM model, which has been trained on Turkish treebanks in the Universal Dependencies (UD) framework. The sentence splitter is designed to identify the boundaries of sentences in a text. It utilizes shuffled texts and their corresponding sentence groupings to generate binary labels to denote the start of each sentence in a shuffled text. The model uses a character-level tokenization approach and is capable of accurately identifying sentence boundaries even in complex, shuffled texts.

9.1.3. Deasciifier Implementation

We have implemented a Deasciifier using a Bi-LSTM model. The model is designed to convert Turkish text written in ASCII characters back into its original Turkish form. This model has been trained on paired datasets consisting of original Turkish sentences and their ASCII form, learning to replace ASCII characters with their Turkish counterparts. The deasciifier model is beneficial for processing texts that have been affected by keyboard layout issues, typographical errors, or other forms of text corruption, allowing for more accurate natural language processing in Turkish.

9.2. Model Architecture

The Bi-LSTM model that was used for the training of the models consists of the following layers:

- Input layer
- Embedding layer
- Bi-directional LSTM layer (128 units)
- Bi-directional LSTM layer (64 units)
- Dense layer (1 unit, sigmoid activation)

9.3. Training and Evaluation

The models were trained on a training set and evaluated on a test set, with an 80% to 20% split for validation of sentence splitter and tokenizer; with an 90% to 10% split for validation of deasciifier. The training was performed for five epochs with a batch size of 32 for tokenizer and deasciifier; for six epochs with a batch size of 32 for sentence splitter. The performance metrics were binary cross-entropy loss and accuracy for all of the models.

9.4. Results

Results show that the Bi-LSTM based tokenizer, sentence splitter, and deasciifier perform well on the test set which has been adjusted for various evaluations. Results of measurement for the models are as follows:

9.4.1. Results for Tokenizer

- Aggregate f-measure of the Tokenizer is calculated as: 95% Overall consideration for the accuracy of the Tokenizer can be summarized as follows; since our training data can be considered to be adequate in terms of various unbiased context

based words, the model could be trained in an un-biased manner with the right adjustment of the hyperparameters.

9.4.2. Results for Sentence Splitter

- Aggregate f-measure of the sentence splitter is calculated as: 98% Overall consideration for the accuracy of the Sentence Splitter can be summarized as follows; The task of sentence splitting is easier compared to the other two, and we were able to create the almost optimal training data for the sentence splitter via shuffling the sentences for creating paragraphs. Also since our training data can be considered to be adequate in terms of various unbiased context based sentences, the model could be trained in an un-biased manner after hyperparameter adjustments via grid search.

9.4.3. Results for Deasciifier

- Aggregate f-measure of the deasciifier is calculated as: 84% Considering the overall accuracy for the deasciifier, we can say that; This task was the most cumbersome one among the three. Since we didn't have specific data for this purpose we had to create the data set ourselves, even though this creation resulted successful, the pseudo data still wasn't sufficient for the enough training of deasciifier for all replacement types.