Dokumentation des finalen Softwaretools

Dokumentation des Moduls "Projektrealisierung"

im Bereich Wirtschaftsinformatik: Data Science
an der Dualen Hochschule Baden-Württemberg Mannheim

Autoren:	Niklas Koch, Niclas Cramer, Jasmina Pascanovic & Antoine Fuchs
Matrikelnummern:	6699912, 7607733, 5711726 & 3008441
Kurs:	WWI20DSA
Studiengangsleiter:	Prof. Dr.-Ing. habil. Dennis Pfisterer
Dozent:	Michael Lang, Enzo Hilzinger
Abgabedatum:	27.07.2023

Inhaltsverzeichnis

Codeverzeichnis

1 Einführung

Dieses Dokument beinhaltet sämtliche Funktionen, die während der Bearbeitung des Projekts „SynTex“ genutzt wurden. Das Dokument komplettiert die ReadMe-Dokumente und den Projektabschlussbericht, die ebenfalls in Abgaberepository auffindbar sind. Die Repositories sind über folgende Links erreichbar:

- Abgabe 1: https://github.com/NICFRU/Abgabe-MS-1
- Abgabe 2: https://github.com/NICFRU/Abgabe-MS-2
- Abgabe 3: https://github.com/NICFRU/Abgabe-MS-3
- Projekt: https://github.com/NICFRU/Projektrealisierung

2 Datenverarbeitung

Die Funktionen, die in Listing 2.1 abgebildet sind, zeigen, wie die Texte aus den verschiedenen Quellen zu einem großen Gesamtdataframe zusammengefasst werden. Da die wissenschaftlichen Texte nur als txt-Datei vorlagen, musst hier ein etwas komplizierterer Ansatz verwendet werden, als bei den anderen Textarten, die jeweils als csv-Dateien vorlagen. Es wird sich auf 500 Trainings- und 200 Testdaten beschränkt. Danach werden die Daten jeweils zu einem Dataframe zusammengefasst und exportiert.

```
1  with open('../data/sources/val.txt','r') as f:
2    texts = []
3    for i in range(700):
4      line = f.readline()
5      obj = json.loads(line)
6      text = obj["article_text"]
7      new_list = []
8      for string in text:
9        string = string.replace(" .", ".")
10       new_list.append(string)
11     texts.append(new_list)
12
13   for i in range(len(texts)):
14     x = ''
15     x = x.join(texts[i][j] for j in range(len(texts[i])))
16     texts_2.append(x)
17
18 scientific_texts = pd.DataFrame(columns=["classification", "text"])
19 scientific_texts["text"] = texts_2
20 scientific_texts["classification"] = "Scientific"
21 scientific_texts_train = scientific_texts[:500]
22 scientific_texts_test = scientific_texts[500:]
23
24 news = pd.read_csv("../data/sources/news.csv", sep=",", encoding="ISO
       -8859-1")
25 news = news[:700]
26 news["classification"] = "news"
27 news = news[["text", "classification"]]
28 news_train = news[:500]
29 news_test = news[500:]
30
31 reviews = pd.read_csv("../data/sources/reviews.csv", sep=",", header=None
       )
32 reviews = reviews[:700]
33 reviews["classification"] = "reviews"
34 reviews = reviews[[2, "classification"]]
35 reviews.rename(columns={2: "text"}, inplace=True)
36 reviews_train = reviews[:500]
37 reviews_test = reviews[500:]
38
39 stories = pd.read_excel("../data/sources/stories.xlsx")
40 stories = stories[:700]
41 stories["classification"] = "story"
```

```
42 stories = stories[["story", "classification"]]
43 stories.rename(columns={"story": "text"}, inplace=True)
44 stories_train = stories[:500]
45 stories_test = stories[500:]
46
47 data_train = pd.concat([scientific_texts_train, news_train, reviews_train
       , stories_train])
48 data_test = pd.concat([scientific_texts_test, news_test, reviews_test,
       stories_test])
49
50 data_train.to_csv("../data/data_train.csv")
51 data_test.to_csv("../data/data_test.csv")
```

Listing 2.1: Funktion, welche die Daten aus verschiedenen Quellen zusammenfügt

3 Klassifikation

In diesem Kapitel werden die wichtigsten Funktionen erklärt, die für die Klassifikation und die Entwicklung des neuronalen Netzes notwendig waren, gezeigt und erklärt.

3.1 Daten einlesen und vorbereiten

Listing 6.13 zeigt die Funktion, die zum Extrahieren von zusätzlichen Features aus den gegebenen Texten verantwortlich ist. Es werden folgende Features extrahiert:

- alle Sätze des Textes als eine Liste
- die Anzahl der Sätze im Text
- alle Wörter des Textes als eine Liste
- die Anzahl der Wörter im Text
- alle Wörter des Textes als eine Liste ohne Füllwörter
- die Anzahl der Wörter im Text ohne Füllwörter
- die Anzahl der Füllwörter im Text
- alle Füllwörter des Textes als Liste
- alle Wörter des Textes in ihrer lemmatisierten Form

Danach werden die zusätzlichen Features mit dem Originaltext in den Dataframes „data_train_with_features.csv“ und „data_test_with_features.csv“ gespeichert.

```
1  def preprocess_text(text):
2      doc = nlp(text)
3      sentences = [sent.text for sent in doc.sents]
4      num_sentences = len(sentences)
5
6      words_without_stopwords = []
7      words_with_stopwords = []
8      lemmas = []
9      stops = []
10
11     for token in doc:
12         if not token.is_stop and not token.is_punct:
13             words_without_stopwords.append(token.text)
14             lemmas.append(token.lemma_)
15         elif not token.is_punct:
16             words_with_stopwords.append(token)
17             stops.append(token)
18
19     num_words_without_stopwords = len(words_without_stopwords)
20     num_words_with_stopwords = len(words_with_stopwords)
21     num_stops = len(stops)
22
23     return sentences, num_sentences, words_with_stopwords,
           num_words_with_stopwords, words_without_stopwords,
           num_words_without_stopwords, lemmas, stops, num_stops
```

```
24
25  data_train['sentences'], data_train['num_sentences'], data_train['
        words_with_stopwords'], data_train['num_words_with_stopwords'],
        data_train['words_without_stopwords'], data_train['
        num_words_without_stopwords'], data_train['lemmas'], data_train['
        stops'], data_train['num_stops'] = zip(*data_train['text'].apply(
        preprocess_text))
26  data_test['sentences'], data_test['num_sentences'], data_test['
        words_with_stopwords'], data_test['num_words_with_stopwords'],
        data_test['words_without_stopwords'], data_test['
        num_words_without_stopwords'], data_test['lemmas'], data_test['stops'
        ], data_test['num_stops'] = zip(*data_test['text'].apply(
        preprocess_text))
27  data_train.to_csv("../../data/data_with_features/data_train_with_features
        .csv")
28  data_test.to_csv("../../data/data_with_features/data_test_with_features.
        csv")
```

Listing 3.1: Funktionen zum Extrahieren von zusätzlichen Features aus den gegebenen Texten

3.2 Visualisierungen

Listing 3.2 zeigt die Funktion, mit welcher Histogramme erstellt wurden, um die zusätzlichen Features zu visualisieren, die im vorherigen Kapitel extrahiert wurden. Mit diesem Code können Visualisierungen erzeugt werden, wie die, in Abbildung 3.1, bei welcher die Verteilung der Anzahl an Wörtern inklusive Füllwörter für die Nachrichtenartikel in unserem Trainingsdatensatz veranschaulicht.

```
1  def visualize_histogram(data, bin_size, x_label, y_label, edgecolor="
       black", linewidth=1):
2      plt.hist(data, bins=bin_size, color="steelblue", edgecolor=edgecolor,
           linewidth=linewidth)
3      plt.xlabel(x_label)
4      plt.ylabel(y_label)
5      plt.show()
```

Listing 3.2: Funktion zum Visualisieren von Features in Form eines Histograms

Listing 3.3 zeigt den Code, der genutzt wurde, um die durchschnittlichen Werte numerischer Features für unterschiedliche Klassen zu visualisieren. Zum Verständnis sind in dem Code auch Kommentare eingebaut. Damit können Visualisierungen wie Abbildung 3.2 erzeugt werden.

```
# Group the data by 'classification' and calculate the mean values for other variables

data_train_metric = data_train[["num_sentences", "num_words_with_stopwords", "num_words_without_stopwords", "num_stops"]].copy()
ttest_data = data_train.groupby("classification").mean()

# Get the variable names and their mean values
variable_names = grouped_data.columns.tolist()
mean_values = grouped_data.values.tolist()

# Set the color palette for the bars
colors = ["steelblue", "salmon", "lightgreen", "purple"]

# Plot the grouped bar chart
fig, ax = plt.subplots()
x = np.arange(len(variable_names))
width = 0.2

bars = []
for i, values in enumerate(mean_values):
    bar = ax.bar(x + (i * width), values, width=width, label=data_train["classification"].unique()[i], color=colors[i])
    bars.append(bar)

# Customize the plot
ax.set_xlabel("Variables")
ax.set_ylabel("Mean Values")
ax.set_title("Means of numeric variables on Classification")
ax.set_xticks(x + (len(data_train["classification"].unique()) * width) / 2)
ax.set_xticklabels(variable_names)

# Create a legend
handles = [bar[0] for bar in bars]
labels = data_train["classification"].unique()
ax.legend(handles, labels)
plt.tight_layout()
plt.xticks(rotation=45, ha='right')

# Show the plot
plt.show()
```

Listing 3.3: Funktion zum Visualisieren von Durchschnittswerten für numerische Zusatzfeatures (inkl. Kommentare).

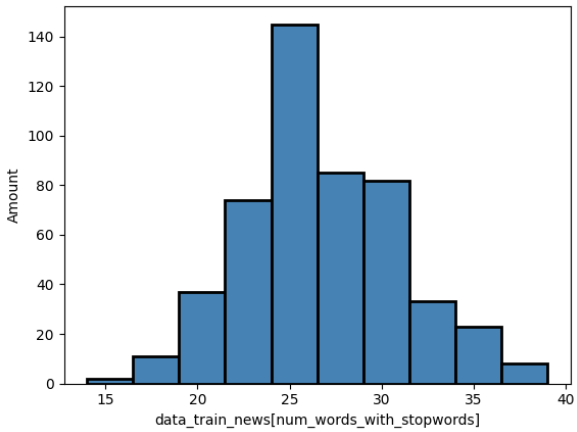

Abbildung 3.1: Verteilung der Anzahl an Wörtern inklusive Füllwörter für Nachrichtenartikel in unserem Trainingsdatensatz

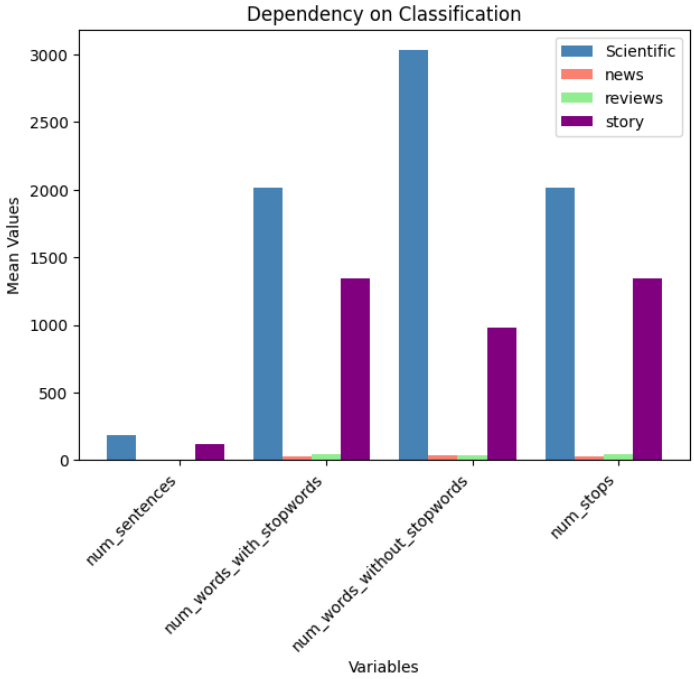

Abbildung 3.2: Durchschnittliche Werte numerischer Features für unterschiedliche Klassen

3.3 Support-Vector-Machine (SVM)

Listing 3.4 zeigt die Funktion, die genutzt wurde, um die Dataframes einzulesen und zufällig durchzumischen.

```
1 data_train = pd.read_csv("../../data/data_with_features/
      data_train_with_features.csv").drop(["Unnamed: 0"], axis=1)
2 data_test = pd.read_csv("../../data/data_with_features/
      data_test_with_features.csv").drop(["Unnamed: 0"], axis=1)
3
4 data_train = data_train.sample(frac=1).reset_index(drop=True)
5 data_test = data_test.sample(frac=1).reset_index(drop=True)
```

Listing 3.4: Einlesen und Durchmischen der Datensätze

In Listing 3.5 werden gleich mehrere Funktionen gezeigt. Zuerst wird ein Vectorizer auf die Texte des Dataframes trainiert, um Texte und Wörter in numerische Vektoren umzuwandeln. Danach werden die Trainingsdaten nochmal mittels eines Splits in Trainings- und Testdaten (bzw. Validierungsdaten) aufgeteilt. Das Modell wird dann trainiert, getestet und evaluiert.

```
1  vectorizer = TfidfVectorizer()
2  features = vectorizer.fit_transform(data_train["text"])
3
4  X_train, X_test, y_train, y_test = train_test_split(features, data_train[
       "classification"], test_size=0.2, random_state=42)
5
6  model = svm.SVC(probability=True)
7  model.fit(X_train, y_train)
8
9  y_pred = model.predict(X_test)
10 print(classification_report(y_test, y_pred))
```

Listing 3.5: Training des Vectorizers, cross-splitten der Trainingsdaten und Erstellung der SVM

Die Daten, welche am Anfang als „data_test“ gespeichert wurden, sind für das Modell nun vollkommen unbekannt und können zum Testen & Evaluieren der SVM genutzt werden. Ein Beispiel dafür zeigt Listing 3.6. Hier wird der Eintrag mit Index 12 der Testdaten geprüft. Der Text wird mittels des Vectorizers umgewandelt und das Modell daraufhin nach einer Klassifikation angefragt. Dabei sollen die jeweiligen Wahrscheinlichkeiten für jede der 4 Klassen mitangegeben werden. Der Output von Listing 3.6 ist in dem Fall bspw:

- Klasse: Scientific
- Wahrscheinlichkeit: 99.97%
- Wahre Klasse: Scientific

```
1  new_text = data_test["text"][12]
2  new_text_features = vectorizer.transform([new_text])
3  probabilities = model.predict_proba(new_text_features)
4  true_class = data_test["classification"][12]
5  predicted_class = model.predict(new_text_features)
6
7  for i, probs in enumerate(probabilities):
8      class_probabilities = ["{:.2f}%".format(prob * 100) for prob in probs
           ]
9      print("Klasse {}: {}".format(predicted_class, class_probabilities))
10 print("Vorhergesagte Klasse:", predicted_class, " | Wahre Klasse:",
       true_class)
```

Listing 3.6: SVM-Test mit beliebigem Beispieltext testen

3.4 Neuronales Netz

Für das Neuronale Netz wurden die Daten genauso eingelesen und der Vectorizer wurde genauso trainiert, wie in dem Code für die Support-Vector-Machine. Listing 3.7 zeigt dann den Code, der verwendet wird, um ein sehr simples neuronales Netz mit drei Layern zu bauen. Diesem Netz werden die Textvektoren als Input gegeben und einer der 4 Klassen werden daraufhin ausgegeben. Als Optimizer werden Adam, als Kostenfunktion die Kreuzentropie und als Metric die Accuracy genommen. Das ist soweit meistens der Standard.

```
model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(features.shape[1],))
        ,
    layers.Dense(64, activation='relu'),
    layers.Dense(4, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Listing 3.7: Aufbau des simplen neuronalen Netzes

Listing 3.8 zeigt den Code zum Umwandeln der Klassen in numerische Werte mittels eines eigenen Labelizers. Danach folgt die Funktion, die das Modell trainiert. Es werden 10 Trainingsepochen durchgeführt.

```
label_to_int = {label: i for i, label in enumerate(np.unique(data_train["
    classification"]))}
y_train = np.array([label_to_int[label] for label in y_train])
y_test = np.array([label_to_int[label] for label in y_test])

model.fit(X_train.toarray(), y_train, epochs=10, batch_size=16, verbose
    =1)
```

Listing 3.8: Umwandeln der Klassen in numerische Werte & Trainieren des Modells

Listing 3.9 zeigt, wie das Modell evaluiert wurde, d.h. wie der Loss und die Accuracy ermittelt wurden.

```
loss, accuracy = model.evaluate(X_test.toarray(), y_test, verbose=1)
print('Test Loss:', loss)
print('Test Accuracy:', accuracy)

model.save("../../models/classification/neuro_net_1.h5")
```

Listing 3.9: Evaluierung des neuronalen Netzes

Listing 3.10 zeigt dann einen Test des Modells auf bis dahin ungesehene Daten aus dem Testdatensatz. Die Predictions werden gespeichert und für jede Klasse eine Wahrscheinlichkeit

angegeben. Die Ausgabe ist quasi dieselbe, wie die bei der SVM.

```
test_number = 78
new_text = data_test["text"][test_number]
new_class = data_test["classification"][test_number]
new_text_features = vectorizer.transform([new_text])

predictions = model.predict(new_text_features.toarray())
predicted_class = np.argmax(predictions, axis=1)
predicted_probability = np.max(predictions, axis=1)

int_to_label = {i: label for label, i in label_to_int.items()}

predicted_labels = [int_to_label[prediction] for prediction in
    predicted_class]
for label, probability in zip(predicted_labels, predicted_probability):
    print(f"Vorhergesagte Klasse: {label}, Wahrscheinlichkeit: {
        probability}, Wahre Klasse: {new_class}. ", predicted_class)
```

Listing 3.10: Test des neuronalen Netzes auf ungesehene Daten

3.5 Transformer-Modelle

Listing 3.11 enthält den Code, mit welchem die Daten und ein Transformer zum Training vorbereitet wurden. Die Daten werden auch hier gelabelt, diesmal mit einem vordefinierten Label-Encoders, die Daten werden erneut gesplittet und ein vortrainiertes Modell wird heruntergeladen. In dem Fall handelt es sich um das „uncased-Bert“ Modell von Google.

```
texts = data_train['text'].tolist()
labels = data_train['classification'].tolist()

label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(labels)

train_texts, test_texts, train_labels, test_labels = train_test_split(
    texts, encoded_labels, test_size=0.2, random_state=42)

tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased'
    )
train_encodings = tokenizer(train_texts, truncation=True, padding=True,
    return_tensors='tf')
test_encodings = tokenizer(test_texts, truncation=True, padding=True,
    return_tensors='tf')

model = TFDistilBertForSequenceClassification.from_pretrained('distilbert
    -base-uncased', num_labels=4)

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])

# Dauer: ca. 1 Minute
```

Listing 3.11: Initialisieren des Transformer-Modells und Vorbereitung der Daten

Listing 3.12 zeigt, wie das Transformer-Modell trainiert bzw. optimiert wird. Es wird in diesem Fall eine eigene Trainingsfunktion mit einem Tensorflow-Dekorator genutzt. Der Grund dafür ist, dass der Code aufgrund von Bugfixing soweit verändert wurde und lediglich so funktioniert hat. Das Modell trainiert über 10 Epochen und erreicht sehr schnell eine Accuracy von über 99%. Dieser Code dauert etwa 2h bis er komplett durchgelaufen ist.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='
    train_accuracy')

@tf.function
def train_step(inputs, labels):
    with tf.GradientTape() as tape:
        logits = model(inputs)[0]
        current_loss = loss(labels, logits)
    gradients = tape.gradient(current_loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    train_loss(current_loss)
    train_accuracy(labels, logits)

epochs = 10
batch_size = 16
steps_per_epoch = len(train_texts) // batch_size

for epoch in range(epochs):
    train_loss.reset_states()
    train_accuracy.reset_states()

    for step in range(steps_per_epoch):
        batch_inputs = {key: value[step * batch_size:(step + 1) *
            batch_size] for key, value in train_encodings.items()}
        batch_labels = train_labels[step * batch_size:(step + 1) *
            batch_size]
        train_step(batch_inputs, batch_labels)

    print(f'Epoch {epoch + 1}: Loss {train_loss.result()}, Accuracy {
        train_accuracy.result()}')

try:
    model.save("../../models/classification/model")
except:
    print("Save did not work!")

# Dauer: etwa 2h
```

Listing 3.12: Training des Transformer-Modells

Listing 3.13 zeigt den Code, der zum Testen bzw. zum Nutzen des Transformer-Modells geplant war. „Geplant war“, weil diese Funktion nicht zum Laufen gebracht werden konnte. Es gab beim Ausführen des Gesamtskripts immer entweder ein Problem beim Training des Transformers oder beim Testen. Wurde ein Problem gelöst, entstand ein neues beim jeweils anderen Prozessschritt. Teilweise gab es Fehlermeldungen, die weder mit der Dokumentation, noch mit Fragen von Experten auf diversen Plattformen wie StackOverflow gelöst werden konnten. Aufgrund von mangelnder Zeit und ausreichender Performance der anderen genutzten Modelle, wurde der Transformer-Ansatz für die Klassifikation vorerst verworfen.

```
1 test_inputs = {
2     'input_ids': test_encodings['input_ids'],
3     'attention_mask': test_encodings['attention_mask']
4 }
5
6 test_loss, test_accuracy = model.evaluate(test_inputs, test_labels)
7 print('Test Accuracy:', test_accuracy)
```

Listing 3.13: (Geplantes) Testen bzw. Nutzen des Transformer-Modells

4 Zusammenfassung Teil I

Die Funktionen der Zusammenfassung werden in zwei Teile unterteilt entsprechend, der wichtigsten Files. Teil eins bezieht sich auf „paraphrasing_trainpipeline.ipynb “. Manche Funktionen können zweimal auftauchen, um die Struktur der Files nicht zu unterbrechen, werden diese jedoch beibehalten.

4.1 Preprocessing

```
import ast
df_explotde=df_train[['batch_texts','batch_output']]
df_explotde=df_explotde.applymap(ast.literal_eval)
df_explotde['C'] = df_explotde.apply(lambda row: list(zip(
row['batch_texts'], row['batch_output'])), axis=1)
df_explotde = df_explotde.explode('C')

Get 'A' and 'B' values back
df_explotde[['batch_texts', 'batch_output']] =

pd.DataFrame(df_explotde['C'].tolist(), index=df_explotde.index)
df = df_explotde.drop(columns=['C'])
```

Listing 4.1: (Weitere) Vorverarbeitung der Daten

Beschreibung: Listing 4.1 befasst sich mit der Verarbeitung eines Pandas DataFrame, um die Spalten 'batch_texts' und 'batch_output' zu kombinieren und zu manipulieren.

```
from deepmultilingualpunctuation import PunctuationModel
model = PunctuationModel()
result = model.restore_punctuation(text)
```

Listing 4.2: Hinzufügen von Interpunktion

Beschreibung: Listing 4.2 fügt einem gegebenen Text Punkte hinzu. Dies ist vor allem bei der Verarbeitung von Speech-to-Text-Lösungen nötig.

4.2 token_count()

```
def token_count(text):
    tokens = text.split()
    return len(tokens)
```

Listing 4.3: Funktionen zum Zählen von Token

Eingabe: `text` (String): Ein Text, dessen Tokenanzahl berechnet werden soll.
Ausgabe: Die Anzahl der Tokens (Wörter) im gegebenen Text.

Beschreibung: Die Funktion **_token_count** aus Listing 4.3 nimmt einen Text als Eingabe und zählt die Anzahl der Tokens (Wörter) in diesem Text. Der Text wird mithilfe der **split()**-Methode in eine Liste von Wörtern (Tokens) aufgeteilt. Die Anzahl der Elemente in der resultierenden Liste (die Anzahl der Wörter im Text) wird mit der **len()**-Funktion ermittelt. Die Funktion gibt die Anzahl der Tokens (Wörter) im Text als Ergebnis zurück.

4.3 adjust_length()

```
1  def adjust_length(text):
2      length = token_count(text)
3      if length < 20:
4          min_length = length + int(length * 0.05)
5          max_length = min_length + min_length
6      elif length < 50:
7          min_length = length + int(length * 0.05)
8          max_length = min_length + min_length * 0.5
9      elif length < 60:
10         min_length = length + int(length * 0.05)
11         max_length = min_length + min_length * 0.4
12     elif length < 80:
13         min_length = length + int(length * 0.05)
14         max_length = min_length + min_length * 0.25
15     elif length < 100:
16         min_length = length + int(length * 0.3)
17         max_length = min_length + 100
18     else:
19         min_length = math.ceil(length / 50) * 70
20         max_length = min_length + 100
21     return min_length, max_length
```

Listing 4.4: Funktion, die minimale und maximale Länge des Outputs festlegt

Eingabe:

- `sentenc` (String): Ein langer Text, der in Sätze aufgeteilt werden soll.
- `splitt` (optional, Integer): Die maximale Tokenanzahl, die ein Batch enthalten darf. Standardwert ist 180.
- `split` (optional, String): Das Trennzeichen, um den Text in Sätze aufzuteilen. Standardwert ist „.“ .

Ausgabe: Eine Liste von Batches, wobei jeder Batch eine Teilmenge von Sätzen ist, die eine maximale Tokenanzahl von `splitt` nicht überschreitet.
Beschreibung: Die Funktion `batch_sent` aus Listing 4.3 erhält einen langen Text (`sentenc`) und optional die Parameter `splitt` und `split`. Sie teilt den Text in Sätze auf und erstellt Batches von Sätzen, wobei jeder Batch eine maximale Tokenanzahl von `splitt` nicht überschreiten darf.

4.4 batch_sent()

```
1  def batch_sent(sentenc, splitt=180, split='.'):
2      sentences = sentenc.split(split)
3      batches = []
4      batch = []
5      batch_len = 0
6      for sentence in sentences:
7          sentence_len = len(tokenizer.tokenize(sentence))
8          if sentence_len + batch_len > splitt:
9              if sentence_len < splitt:
10                 batches.append(batch)
11                 batch = [sentence]
12                 batch_len = sentence_len
13             else:
14                 batch.append(sentence)
15                 batch_len += sentence_len
16     batches.append(batch)
17     return batches
```

Listing 4.5: Funktion, welche einen Text in Batches aufteilt

Beschreibung:

Die Funktion **batch_sent** aus Listing 7.3 nimmt einen Text (**sentenc**) und optionale Parameter (**splitt** und **split**) als Eingabe und erstellt Batches von Sätzen, wobei jeder Batch eine maximale Tokenanzahl von **splitt** nicht überschreitet.

4.5 paraphrase_of_text()

```
1      batch_text_list = []
2
3      # Auslesen der Text- und Kompressionsinformationen aus dem
           W rterbuch
4      text = dictionary[text_name]
5      komp = dictionary[komp_name]
6
7      # Iteration ueber die Batches des Texts
8      for batch in tqdm(batch_sent(text, split=split), desc='Verarbeite
           Batches'):
9          # Ueberpruefen Sie, ob der aktuelle Batch nicht leer ist
10         if len(batch):
11             # Zusammenfuegen der Saetze in einem Batch
12             batch_text = '. '.join(batch)
13             batch_text += "."
14             batch_text_list.append(batch_text)
15
16             # Anpassung der Laenge und Generierung der Zusammenfassung
17             min_length_test, max_length_test = adjust_length(batch_text)
18             ext_summary = summarizer(batch_text, max_length=int(round(
                   max_length_test * komp, 0)), min_length=int(round(
```

```
                min_length_test * komp, 0)), length_penalty=100,
                num_beams=2)

            # Hinzufuegen der generierten Zusammenfassung zur Gesamtliste
            text_gesamt_list.append(ext_summary[0]['generated_text'])

    # Erstellen der Gesamtzusammenfassung und Berechnung der endgueltigen
        Kompressionsrate
    text_gesamt = '. '.join(text_gesamt_list)
    actual_compression_rate = len(text_gesamt.split(' ')) / len(text.
        split(' ')) * 100

    # Aktualisierung des Woerterbuchs mit den neuen Informationen
    dictionary['Zusammenfassung'] = text_gesamt
    dictionary['Endgueltige_Kompressionsrate'] = actual_compression_rate
    dictionary['laenge Zusammenfassung'] = len(text_gesamt.split(' '))
    dictionary['laenge Ausgangstext'] = len(text.split(' '))
    dictionary['batch_texts'] = batch_text_list
    dictionary['batch_output'] = text_gesamt_list

    return dictionary
```

Listing 4.6: Funktion, die den Paraphraser ausführt und alle relevanten Informationen in einem Dataframe festhält

Eingabe:

- `df_s` (DataFrame): Ein DataFrame, der die Texte und Kompressionsraten enthält.
- `text_name` (optional, String): Der Name der Spalte im DataFrame, die die Texte enthält. Standardwert ist 'text'.
- `komp_name` (optional, String): Der Name der Spalte im DataFrame, die die Kompressionsraten enthält. Standardwert ist 'reduction_multiplier'.
- `split` (optional, String): Das Trennzeichen, um den Text in Sätze aufzuteilen. Standardwert ist '. '.

Ausgabe: Ein DataFrame `df_summary_testing`, der Zusammenfassungen der Texte basierend auf den gegebenen Kompressionsraten enthält.
Beschreibung: Die Funktion `paraphrase_of_text` aus Listing **??** erhält einen DataFrame `df_s`, in dem Texte und Kompressionsraten angegeben sind, sowie optionale Parameter `text_name`, `komp_name` und `split`. Sie gibt einen neuen DataFrame `df_summary_testing` zurück, der Zusammenfassungen der Texte basierend auf den gegebenen Kompressionsraten enthält.

4.6 calculate_compression()

```
def calculate_compression(df, total_tokens_col, current_tokens_col,
```

```
2 desired_compression_rate):
3     df['current_compression_rate'] = df[current_tokens_col] /
4     df[total_tokens_col]
5     df['compression_difference'] = desired_compression_rate -
6     df['current_compression_rate']
7     df['reduction_multiplier'] = desired_compression_rate /
8     df['current_compression_rate']
9     return df
```

Listing 4.7: Funktion, welche Informationen zu Kompressionsrate zusammenführt

Eingabe:

- `df` (DataFrame): Der Eingabe-DataFrame, der die Informationen zu den Texten enthält.
- `total_tokens_col` (String): Der Name der Spalte, die die Gesamtzahl der Tokens im Text enthält.
- `current_tokens_col` (String): Der Name der Spalte, die die aktuelle Tokenanzahl (nach der Zusammenfassung) im Text enthält.
- `desired_compression_rate` (float): Die gewünschte Kompressionsrate (Verhältnis der Tokens nach der Zusammenfassung zur ursprünglichen Tokenanzahl).

Ausgabe: Der DataFrame `df` mit zusätzlichen Spalten, die Informationen zur Kompression enthalten.
Beschreibung: Die Funktion `calculate_compression` aus Listing 4.7 berechnet die Kompressionsrate für jeden Text im DataFrame `df` und fügt dem DataFrame zusätzliche Spalten hinzu, die Informationen zur Kompression enthalten.

4.7 clean_text()

```
1 def clean_text(text):
2     sentences = nltk.sent_tokenize(text.strip())
3     sentences_cleaned = [s for sent in sentences for s
4     in sent.split("\n")]
5     sentences_cleaned_no_titles = [sent for sent in sentences_cleaned
6     if len(sent) > 0 and
7     sent[-1] in string.punctuation]
8     text_cleaned = "\n".join(sentences_cleaned_no_titles)
9     return text_cleaned
```

Listing 4.8: Funktion zum Reinigen (Vorverarbeiten) von den Texten

Eingabe: `text` (String): Ein Text, der gereinigt werden soll.
Ausgabe: Ein gereinigter Text ohne Zeilenumbrüche innerhalb der Sätze und ohne Sätze, die keine Interpunktion am Ende haben.

Beschreibung: Die Funktion `clean_text` aus Listing 4.8 erhält einen Text (`text`) und reinigt diesen, indem sie Zeilenumbrüche innerhalb der Sätze entfernt und Sätze eliminiert, die keine Interpunktion am Ende haben.

4.8 preprocess_data()

```
1  def preprocess_data(examples):
2      texts_cleaned = [clean_text(text) for text
3      in examples["batch_texts"]]
4      inputs = [prefix + text for text in texts_cleaned]
5
6      model_inputs = tokenizer(inputs,
7      max_length=max_input_length, truncation=True)
8
9      # Setup the tokenizer for targets
10     with tokenizer.as_target_tokenizer():
11         labels = tokenizer(examples["batch_output"],
12         max_length=max_target_length,truncation=True)
13
14     model_inputs["labels"] = labels["input_ids"]
15     return model_inputs
```

Listing 4.9: Funktion, die den Text weiter vorverabeitet

Eingabe:

- `examples` (DataFrame): Ein Dataframe mit Beispielen, das Spalten `"batch_texts"` und `"batch_output"` enthält. Jede Zeile enthält einen Eingabetext `"batch_texts"` und das entsprechende Ziel-Output `"batch_output"`.

Ausgabe: Ein Dictionary von vorverarbeiteten Modell-Eingaben, die für das Training oder die Vorhersage verwendet werden können. Das Dictionary enthält die Tokenisierung der gereinigten Eingabetexte mit den entsprechenden Token-IDs für das Modell und die Tokenisierung der Ziel-Outputs.
Beschreibung: Die Funktion `preprocess_data` aus Listing 4.9 übernimmt ein DataFrame mit Beispielen (`examples`), reinigt die Eingabetexte und führt die Tokenisierung für das Modell durch. Sie gibt ein vorverarbeitetes Dictionary zurück, das für das Training oder die Vorhersage verwendet werden kann.

4.9 Initialisierung des Trainingsmodells

```
1  batch\size = 8
2  model\name = "bart-base-paraphrasing"
3  model\dir = f"drive/MyDrive/Models/{model_name}"
```

```
args = Seq2SeqTrainingArguments(
    model_dir,
    evaluation_strategy="steps",
    eval_steps=100,
    logging_strategy="steps",
    logging_steps=100,
    save_strategy="steps",
    save_steps=200,
    learning_rate=4e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    weight_decay=0.01,
    save_total_limit=3,
    num_train_epochs=1,
    predict_with_generate=True,
    fp16=True,
    load_best_model_at_end=True,
    metric_for_best_model="rouge1",
    report_to="tensorboard"
)
```

Listing 4.10: Initialisierung des Trainingsmodells

Beschreibung: Der Code aus Listing 4.10 definiert eine Instanz der Klasse Seq2Seq, um die Hyperparameter für das Training eines Seq2Seq-Modells festzulegen. Die hiermit erstellten Modelle sind später für die Zusammenfassungen zuständig.

4.10 Ausführen des Modells

```
def model_init():
    return AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)

trainer = Seq2SeqTrainer(
    model_init=model_init,
    args=args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)

trainer.train()
trainer.save_model()
```

Listing 4.11: Ausführung des initialisierten Modells

Beschreibung: `model_init`: Die Funktion `model_init` aus Listing 4.11 wird definiert, um das Modell zu initialisieren. Sie lädt das BART-Modell für Seq2Seq-Aufgaben aus dem vortrainierten Modell-Checkpoint (`model_checkpoint`) und gibt das Modell zurück, das für das Training verwendet wird.

4.11 Testen und Evaluieren der Modelle

```
# get test split
test_tokenized_dataset = tokenized_datasets["test"]

# pad texts to the same length
def preprocess_test(examples):
    inputs = [prefix + text for text in examples["text"]]
    model_inputs = tokenizer(inputs,
    max_length=max_input_length, truncation=True,
    padding="max_length")
    return model_inputs

test_tokenized_dataset = test_tokenized_dataset.map(preprocess_test,
batched=True)

# prepare dataloader
test_tokenized_dataset.set_format(type='torch', columns=['input_ids',
'attention_mask'])
dataloader = torch.utils.data.DataLoader(test_tokenized_dataset,
batch_size=32)

# generate text for each batch
all_predictions = []
for i, batch in enumerate(dataloader):
    predictions = model.generate(**batch)
    all_predictions.append(predictions)

# flatten predictions
all_predictions_flattened = [pred for preds in all_predictions
for pred in preds]

# tokenize and pad titles
all_titles = tokenizer(test_tokenized_dataset["title"],
max_length=max_target_length,truncation=True,
padding="max_length")["input_ids"]

# compute metrics
predictions_labels = [all_predictions_flattened, all_titles]
compute_metrics(predictions_labels)
```

Listing 4.12: Durchlaufen aller wichtigen Funktionen und Messen der Performance

Beschreibung: Der Code aus Listing 4.12 führt die Vorhersage und Berechnung von Metriken für das getestete Modell durch.

- `test_tokenized_dataset`: Das vorverarbeitete Datenset für die Testdaten wird aus `tokenized_datasets` abgerufen und in `test_tokenized_dataset` gespeichert.
- `preprocess_test`: Die Funktion `preprocess_test` wird definiert, um die Testdaten vorzubereiten. Sie fügt den Präfix (`prefix`) zu den Texten hinzu und führt die Tokenisierung und das Padding durch, um alle Texte auf die gleiche Länge zu bringen.

- `test_tokenized_dataset` wird durch Anwendung der `map`-Funktion auf das vorverarbeitete Datenset `test_tokenized_dataset` bearbeitet. Dadurch werden die Testdaten vorbereitet und in ein batchfähiges Format gebracht.
- Ein PyTorch DataLoader (`dataloader`) wird erstellt, um die vorverarbeiteten Testdaten in Batches mit einer Batch-Größe von 32 zu laden.
- Das Modell (`model`) generiert Texte für jede Batch von Eingaben aus dem DataLoader, indem die Methode `generate` aufgerufen wird. Die generierten Texte werden in der Liste `all_predictions` gespeichert.
- Die generierten Texte werden in `all_predictions_flattened` zusammengeführt, um sie einfacher mit den Ziel-Outputs zu vergleichen.
- Die Titel (Ziel-Outputs) werden mit dem Tokenizer tokenisiert und auf die gleiche Länge gepaddet, um sie mit den generierten Texten zu vergleichen.
- Schließlich werden die Vorhersagen (`all_predictions_flattened`) und die Ziel-Outputs (`all_titles`) als Eingaben für die Funktion `compute_metrics` übergeben, um die Metriken zu berechnen und die Leistung des Modells auf den Testdaten zu evaluieren.

5 Zusammenfassung Teil II

Das Kapitel Zusammenfassung II bezieht sich auf das Python-Notebook „dict_input.ipynb“.

5.1 reduce_repetitions()

```
1  def reduce_repetitions(text):
2      text = re.sub(r'\.{2,}', '.', text)
3      # Ersetzt zwei oder mehr Punkte durch einen Punkt.
4      text = re.sub(r'\!{2,}', '!', text)
5      # Ersetzt zwei oder mehr Ausrufezeichen durch ein Ausrufezeichen.
6      text = re.sub(r'\,{2,}', ',', text)
7      # Ersetzt zwei oder mehr Kommas durch ein Komma.
8      text = re.sub(r'\;{2,}', ';', text)
9      # Ersetzt zwei oder mehr Semikolons durch ein Semikolon.
10     return text  # Gibt den bereinigten Text zurueck.
```

Listing 5.1: Funktion, welche Wiederholungen minimiert

Beschreibung: Die Funktion „reduce_repetitions“ aus Listing 5.1 verwendet reguläre Ausdruecke (regex), um wiederholte Zeichen in einem Text zu reduzieren. Dabei werden bestimmte Zeichenmuster durch ein einzelnes Zeichen ersetzt, wenn sie zwei oder mehr Mal hintereinander vorkommen. **Eingabe:** „text“ - Ein Text als Eingabe, in dem wiederholte Zeichen reduziert werden sollen.
Ausgabe: Der bereinigte Text, in dem wiederholte Zeichen durch ein einzelnes Zeichen ersetzt wurden.

5.2 textrank_extractive()

```
1  def textrank_extractive(text, compression_rate=0.5, split='\. '):
2      nlp = spacy.load("en_core_web_lg")
3      doc = re.split(fr'(?<!\b\w\w){split}', reduce_repetitions(re.sub(
4      ' +', ' ', text.replace("\n", " ").replace(
5      '-', ' ').replace('_', ' ').replace("\'", "").replace(
6      "!", ".").replace("?", ".").replace(";", "")))
7
8      sentences = [sent for sent in doc if len(sent.replace(
9      "-", " ").split()) > 2]
10
11     sentence_docs = [nlp(sentence) for sentence in sentences]
12
13     # Hier verwenden wir TextRank, um wichtige Saetze zu extrahieren.
14     num_sentences = max(1, int(len(sentences) * compression_rate))
15     extracted_sentences = summarize(text, words=num_sentences,
16     split=True)
17     #Aehnlichkeitsmatrix
18     similarity_matrix = np.zeros((len(sentences), len(sentences)))
```

```
19      for i, doc_i in enumerate(sentence_docs):
20          for j, doc_j in enumerate(sentence_docs):
21              similarity = similarity_function(doc_i, doc_j)
22              similarity_matrix[i, j] = similarity
23
24      #Aehnlichkeitsgraph
25      graph = nx.from_numpy_array(similarity_matrix)
26
27      # Nun berechnen wir den TextRank-Score fuer jeden Satz.
28      scores = nx.pagerank_numpy(graph)
29
30      # Auswaehlen derbesten Saetze basierend auf ihrem TextRank-Scores
31      top_sentences = sorted(scores, key=scores.get,
32      reverse=True)[:num_sentences]
33
34      # Sortierung nach Position im Text
35      top_sentences = sorted(top_sentences)
36
37      # Schluesselsaetze extrahieren
38      extracted_sentences = [sentences[index] for index
39      in top_sentences]
40
41      return extracted_sentences
```

Listing 5.2: Textrank-Funktion

Beschreibung: Die Funktion „textrank_extractive“ aus Listing 5.2 implementiert den TextRank, um extraktive Schlüsselsätze aus einem gegebenen Text zu extrahieren.
Eingabe:

- „text“: Der Text, aus dem Schlüsselsätze extrahiert werden sollen.
- „compression_rate“ (optional): Ein Parameter, der die Anzahl der extrahierten Sätze im Vergleich zur Gesamtanzahl der Sätze im Text steuert. Standardmäßig ist es auf 0.5 eingestellt.
- „split“ (optional): Ein Trennzeichen oder ein regulärer Ausdruck, der den Text in Sätze aufteilt. Standardmäßig ist es auf ‘. ’ (Punkt gefolgt von Leerzeichen) eingestellt.

Ausgabe: Eine Liste der extrahierten Schlüsselsätze aus dem gegebenen Text.

5.3 similarity_function()

```
1   def similarity_function(doc1, doc2):
2       # WCosinus-Aehnlichkeit
3       similarity = doc1.similarity(doc2)
4       return similarity
```

Listing 5.3: Die Ähnlichkeitsfunktion für den Textrank

Beschreibung: Die Funktion „similarity_function“ aus Listing 5.3 berechnet die Cosinus-Ähnlichkeit zwischen zwei gegebenen Dokumenten, die mit Spacy als Dokumentenobjekte („doc1“ und „doc2“) repräsentiert werden.
Eingabe:

- „doc1“: Ein Spacy-Dokumentenobjekt, das das erste Dokument repräsentiert.
- „doc2“: Ein Spacy-Dokumentenobjekt, das das zweite Dokument repräsentiert.

Ausgabe: Die Cosinus-Ähnlichkeit zwischen den beiden gegebenen Dokumenten. Die Ähnlichkeit ist ein Wert zwischen 0 und 1, wobei 1 eine perfekte Ähnlichkeit bedeutet und 0 eine völlige Unähnlichkeit. Je höher der Wert, desto ähnlicher sind die beiden Dokumente.

5.4 compression_ratio()

```
1 def compression_ratio(text, summary):
2     # Verhaeltnisberechnung
3     num_words_text = len(text.split())
4     num_words_summary = len(summary.split())
5     ratio = num_words_summary / num_words_text
6     return ratio
```

Listing 5.4: Berechnung der Kompressionsrate

Beschreibung: Die Funktion „compression_ratio“ aus Listing 5.4 berechnet das Verhältnis der Anzahl der Wörter in der gegebenen Zusammenfassung zur Anzahl der Wörter im gegebenen Ausgangstext. Dieses Verhältnis gibt an, wie viel der Text komprimiert wurde, um die Zusammenfassung zu erstellen.
Eingabe:

- „text“: Der ursprüngliche Text, aus dem die Zusammenfassung erstellt wurde.
- „summary“: Die erstellte Zusammenfassung des Textes.

Ausgabe: Das Verhältnis der Anzahl der Wörter in der Zusammenfassung zur Anzahl der Wörter im Ausgangstext. Ein Wert größer als 1 bedeutet, dass die Zusammenfassung mehr Wörter enthält als der ursprüngliche Text (keine Kompression). Ein Wert kleiner als 1 bedeutet, dass die Zusammenfassung weniger Wörter enthält als der ursprüngliche Text (Kompression).

5.5 compression()

```
1 def compression(text, compression_rate, split='\. '):
2     max_iterations = 20
```

```
3     iterations = 0
4     extracted = textrank_extractive(text, compression_rate, split)
5     summary = '. '.join(extracted)
6     compression_rate_renewed = compression_rate
7
8     # Kompressionsrate anpassen
9     while compression_ratio(text, summary) <
10    compression_rate and iterations < max_iterations:
11        iterations += 1
12        compression_rate_renewed += 0.05
13        if compression_rate_renewed > 1:
14            compression_rate_renewed = 1
15        extracted = textrank_extractive(text,
16        compression_rate=compression_rate_renewed)
17        summary = '. '.join(extracted)
18    return summary  # Gibt die komprimierte Zusammenfassung zurueck.
```

Listing 5.5: Feinabstimmung der Kompressionsrate

Beschreibung: Die Funktion „compression“ aus Listing 5.5 führt die Textkompression durch, indem sie extraktive Schlüsselsätze mit dem TextRank-Algorithmus extrahiert und eine Zusammenfassung erstellt. Wenn das erzielte Kompressionsverhältnis kleiner ist als die gewünschte Rate, wird versucht, das Kompressionsverhältnis zu erhöhen, um die gewünschte Kompression zu erreichen. **Eingabe:**

- „text“: Der ursprüngliche Text, der komprimiert werden soll.
- „compression_rate“: Der gewünschte Kompressionsgrad als Prozentsatz der Anzahl der extrahierten Sätze im Vergleich zur Gesamtanzahl der Sätze im Text.
- „split“: Ein Trennzeichen oder regulärer Ausdruck, der den Text in Sätze aufteilt. Standardmäßig ist es auf ‘. ‘ (Punkt gefolgt von Leerzeichen) eingestellt.

Ausgabe: Die komprimierte Zusammenfassung des Textes.

5.6 token_count()

```
1  def token_count(text):
2      tokens = text.split()
3      return len(tokens)
```

Listing 5.6: Funktion zum Zählen der Länge der Token

Beschreibung: Die Funktion „token_count“ zählt die Anzahl der Wörter (Tokens) in einem gegebenen Text. Ein Token ist eine einzelne Einheit, die durch Leerzeichen oder andere Trennzeichen getrennt ist. **Eingabe:** „text“: Der Text, dessen Wörter gezählt werden sollen. **Ausgabe:** Die Anzahl der Wörter (Tokens) im gegebenen Text.

5.7 adjust_length()

```
def adjust_length(text):
    length = token_count(text)
    if length <20:
        min_length = length + int(length * 0.05)
        max_length = min_length +min_length
    elif length <50:
        min_length = length + int(length * 0.05)
        max_length = min_length +min_length* 0.5
    elif length <60:
        min_length = length + int(length * 0.05)
        max_length = min_length +min_length* 0.4
    elif length < 80:
        min_length = length + int(length * 0.05)
        max_length = min_length + min_length* 0.25
    elif length < 100:
        min_length = length + int(length * 0.3)
        max_length = min_length + 100
    else:
        min_length = math.ceil(length / 50) * 70
        max_length = min_length + 100
    return min_length, max_length
```

Listing 5.7: Anpassen der minimalen und maximalen Outputlänge

Beschreibung: Die Funktion „adjust_length“ aus Listing 5.7 berechnet die minimale und maximale Länge eines Textes basierend auf der aktuellen Länge des Textes. Diese Funktion wird verwendet, um die Längenbegrenzung für die Textkompression festzulegen. **Eingabe:** „text“: Der Text, dessen Längenbegrenzung angepasst werden soll. **Ausgabe:** Ein Tupel mit der berechneten minimalen und maximalen Länge des Textes.

5.8 batch_sent()

```
def batch_sent(sentenc, splitt=180, split='\. '):
    # Saetze weden durch ". " getrennt
    # ". " folgt nicht auf einzelne Woerter
    sentences = re.split(fr'(?<!\b\w\w){split}', sentenc.lower())

    batches = []
    batch = []
    batch_len = 0

    # Durchlaufen Sie jeden Satz in den Saetzen.
    for sentence in sentences:
        # Berechnen Sie die Anzahl der Tokens im Satz.
        sentence_len = len(tokenizer.tokenize(sentence))

        if sentence_len + batch_len > splitt:
            if sentence_len < splitt:
```

```
17                  batch = [sentence]
18                  batch_len = sentence_len
19          else:
20              batch.append(sentence)
21              batch_len += sentence_len
22      batches.append(batch)
23
24      return batches
```

Listing 5.8: Funktion, welche einen Text in Batches aufteilt

Beschreibung: Die Funktion „batch_sent“ aus Listing 5.8 teilt den eingegebenen Text in Batches von Sätzen auf, wobei jeder Satz durch das Trennzeichen „.“ getrennt ist. Die Funktion stellt sicher, dass das Trennzeichen nicht auf ein einzelnes Wort folgt, um die Aufteilung der Sätze korrekt durchzuführen. **Eingabe:**

- „sentenc“: Der zu verarbeitende Text, der in Sätze aufgeteilt werden soll.
- „splitt“ (optional): Die maximale Anzahl von Tokens, die ein Batch enthalten kann. Standardmäßig ist der Wert auf 180 gesetzt.
- „split“ (optional): Das Trennzeichen, das zum Aufteilen der Sätze verwendet wird. Standardmäßig ist das Trennzeichen „.“.

Ausgabe: Die Funktion gibt eine Liste von Batches zurück, wobei jeder Batch eine Liste von Sätzen ist. Jeder Satz innerhalb eines Batches enthält nicht mehr Tokens als die maximale Batch-Länge („splitt“).

5.9 text_rank_algo()

```
1  def text_rank_algo(dictionary, komp='compression', split='\\. ',
2  random_T=True, column='text'):
3      # Text aus dem Woerterbuch extrahieren und bearbeiten
4      text = dictionary[column].replace("\n", " ")
5      if random_T:
6          random_value = dictionary[komp]
7      else:
8          if dictionary['reduction_multiplier'] < 0.8:
9              random_value = dictionary['desired_compression_rate']
10         elif dictionary['reduction_multiplier'] < 0.9:
11             random_value = dictionary['reduction_multiplier']
12         else:
13             random_value = 1
14
15     # Durchfuehren der Textkompression
16     text_rank_text = compression(text.replace("\n\n", " "),
17     random_value, split)
18     compression_ratio_value = compression_ratio(text,
19     compression(text, random_value, split))
20
```

```
21      # Weitere Textbearbeitung
22      text = re.sub(' +', ' ', text.replace("\n", " ").replace('-','
23      ').replace('_',' ').replace("\'", "").replace("!", ".").replace(
24      "?",".").replace(";", ""))
25      text_rank_text = re.sub(' +', ' ', text_rank_text.replace("\n", "
26      ").replace('-',' ').replace('_',' ').replace("\'", "").replace(
27      "!", ".").replace("?", ".").replace(";", ""))
28
29      # Hinzufuegen von Ergebnissen zum Woerterbuch
30      if random_T:
31          dictionary['text'] = text
32          dictionary['text_rank_text'] = text_rank_text
33          dictionary['tokens_gesamt'] = len(text.split(' '))
34          dictionary['token_text_rank'] = len(
35          text_rank_text.split(' '))
36          dictionary['desired_compression_rate'] = random_value
37          dictionary['text_rank_compression_rate'] =
38          compression_ratio_value
39      else:
40          dictionary['text_rank_text_2'] = text_rank_text
41          dictionary['tokens_gesamt_2'] = len(text.split(' '))
42          dictionary['token_text_rank_2'] = len(
43          text_rank_text.split(' '))
44          dictionary['desired_compression_rate_2'] = random_value
45          dictionary['text_rank_compression_rate_2'] =
46          compression_ratio_value
47
48      # Rueckgabe des aktualisierten Woerterbuchs
49      return dictionary
```

Listing 5.9: Komplettes Durchführen des Textranks sowie Zusammenführen von wichtigen Daten

Beschreibung: Die Funktion „text_rank_algo“ aus Listing 5.9 führt den TextRank für die Textkompression durch und aktualisiert das übergebene Wörterbuch „dictionary“ mit den Ergebnissen. Die Funktion verwendet den Textkompressionswert „komp“ aus dem Wörterbuch, um die Kompressionsrate zu bestimmen. Je nach Wert von „random_T“ wird entweder der zufällig ausgewählte Kompressionswert verwendet oder der Wert aus dem Wörterbuch.
Eingabe:

- „dictionary“: Ein Wörterbuch, das Informationen über den Text und die Kompressionsrate enthält.
- „komp“ (optional): Der Schlüssel für die Kompressionsrate im Wörterbuch. Standardmäßig ist der Wert auf „compression“ gesetzt.
- „split“ (optional): Das Trennzeichen, das zum Aufteilen der Sätze verwendet wird. Standardmäßig ist das Trennzeichen „.“.
- „random_T“ (optional): Ein boolscher Wert, der angibt, ob ein zufällig ausgewählter Kompressionswert verwendet werden soll. Standardmäßig ist der Wert auf True gesetzt.

- „column“ (optional): Der Schlüssel für den Text im Wörterbuch. Standardmäßig ist der Wert auf „text“ gesetzt.

Ausgabe: Das aktualisierte Wörterbuch „dictionary“ mit zusätzlichen Informationen über den komprimierten Text und die Kompressionsrate. Die aktualisierten Schlüssel sind abhängig von „random_T“ und werden entweder als Suffix „_2“ oder direkt im Wörterbuch gespeichert.

5.10 check_class_and_get_model_name()

```
1  def check_class_and_get_model_name(input_dict, class_key):
2      # Aus dem Woerterbuch den Wert des gegebenen Schluessels abrufen
3      class_value = input_dict.get(class_key)
4
5      # ueberpruefen, ob der Wert existiert
6      if class_value is None:
7          raise ValueError(f"'{class_key}' nicht im
8          Eingabedictionary gefunden")
9
10     # Entsprechend dem Wert den Modellnamen zuweisen
11     if class_value == 'Scientific':
12         model_name = 'NICFRU/bart-base-paraphrasing-science'
13     elif class_value == 'news':
14         model_name = 'NICFRU/bart-base-paraphrasing-news'
15     elif class_value == 'story':
16         model_name = 'NICFRU/bart-base-paraphrasing-story'
17     elif class_value == 'reviews':
18         model_name = 'NICFRU/bart-base-paraphrasing-review'
19     else:
20         return False
21
22     # Rueckgabe des Modellnamens
23     return model_name
```

Listing 5.10: Überprüfung des Klassifkationsergebnisses und Modellwahl auf der Grundlage der Klassifikation

Beschreibung: Die Funktion „check_class_and_get_model_name“ aus Listing 5.10 überprüft den gegebenen Schlüssel „class_key“ in dem übergebenen Wörterbuch „input_dict“ und gibt den entsprechenden Modellnamen basierend auf dem Wert des Schlüssels zurück.
Eingabe:

- „input_dict“: Ein Wörterbuch, das Informationen enthält, einschließlich des Schlüssels „class_key“.
- „class_key“: Der Schlüssel, dessen Wert überprüft werden soll, um den Modellnamen zu erhalten.

Ausgabe: Der Modellname, der dem Wert des Schlüssels „class_key“ entspricht. Wenn der Wert nicht einer der definierten Klassen („Scientific“, „news“, „story“, „reviews“) entspricht, wird False zurückgegeben.

5.11 create_model()

```
1 def create_model(dictionary):
2     # Finde passendes Model
3     model_name = check_class_and_get_model_name(dictionary,
4     'classification')
5     global tokenizer, summarizer
6
7     # Lade passenden Tokenizer und Modell von Huggingface
8     tokenizer = AutoTokenizer.from_pretrained(model_name)
9     summarizer = pipeline("text2text-generation", model=model_name)
```

Listing 5.11: Funktion zum Laden des gewählten Modells

Beschreibung: Die Funktion „create_model“ aus Listing 5.11 erstellt ein Text-zu-Text Generierungsmodell basierend auf dem Klassifikationswert, der im übergebenen Wörterbuch „dictionary“ unter dem Schlüssel „classification“ enthalten ist.
Eingabe: „dictionary“ - Ein Wörterbuch, das Informationen enthält, einschließlich des Klassifikationswerts unter dem Schlüssel „classification“. **Funktionsweise:**

1. Die Funktion ruft die „check_class_and_get_model_name“ Funktion auf, um den Namen des Modells basierend auf dem Klassifikationswert zu ermitteln und in der Variablen „model_name“ zu speichern.
2. Die Tokenizer- und Summarizer-Variablen werden als globale Variablen deklariert, um sicherzustellen, dass sie außerhalb der Funktion verwendet werden können.
3. Der Tokenizer wird mit dem ausgewählten Modellnamen von Huggingface initialisiert und in der globalen Variable „tokenizer“ gespeichert.
4. Der Summarizer wird als Pipeline mit dem ausgewählten Modellnamen von Huggingface initialisiert und in der globalen Variable „summarizer“ gespeichert. Dabei wird der Text als Eingabe für die Zusammenfassung verwendet.

Die globalen Variablen „tokenizer“ und „summarizer“ können nun in anderen Funktionen verwendet werden, um Texte zu tokenisieren und Zusammenfassungen zu generieren.

5.12 paraphrase_of_text()

```
1 def paraphrase_of_text(dictionary, text_name='text',
2 komp_name='reduction_multiplier', split='\. '):
```

```
    # Initialisierung der Listen
    text_gesamt_list = []
    batch_text_list = []

    # Auslesen der Text- und Kompressionsinformationen
    text = dictionary[text_name]
    komp = dictionary[komp_name]

    # Iteration ueber die Batches des Texts
    for batch in tqdm(batch_sent(text, split=split),
    desc='Verarbeite Batches'):
        # Ueberpruefen Sie, ob der aktuelle Batch nicht leer ist
        if len(batch):
            # Zusammenfuegen der Saetze in einem Batch
            batch_text = '. '.join(batch)
            batch_text += "."
            batch_text_list.append(batch_text)

            # Anpassung der Laenge und Generierung der Zusammenfassung
            min_length_test, max_length_test =
            adjust_length(batch_text)
            ext_summary = summarizer(batch_text,
            max_length=int(round(max_length_test * komp, 0)),
            min_length=int(round(min_length_test * komp, 0)),
            length_penalty=100, num_beams=2)


            # Hinzufuegen der generierten Zusammenfassung
            text_gesamt_list.append(ext_summary[0]['generated_text'])

    # Erstellen der Gesamtzusammenfassung und Berechnung
    der endgueltigen Kompressionsrate
    text_gesamt = '. '.join(text_gesamt_list)
    actual_compression_rate = len(text_gesamt.split(' ')) /
    len(text.split(' ')) * 100

    # Aktualisierung des Woerterbuchs mit den neuen Informationen
    dictionary['Zusammenfassung'] = text_gesamt
    dictionary['Endgueltige_Kompressionsrate'] =
    actual_compression_rate
    dictionary['laenge Zusammenfassung'] = len(text_gesamt.split(' '))
    dictionary['laenge Ausgangstext'] = len(text.split(' '))
    dictionary['batch_texts'] = batch_text_list
    dictionary['batch_output'] = text_gesamt_list

    return dictionary
```

Listing 5.12: Ausführen des gesamten Paraphrasers sowie das Sammeln wichtiger Informationen

Beschreibung: Die Funktion „paraphrase_of_text“ aus Listing 5.12 dient dazu, einen gegebenen Text zu paraphrasieren und eine Zusammenfassung zu erstellen, indem der Text in Batches aufgeteilt wird, die jeweils eine bestimmte Kompressionsrate erhalten.

Eingabe:

- „dictionary“: Ein Wörterbuch, das den Text und die Kompressionsrate enthält.
- „komp_name“ (optional): Der Schlüssel im Wörterbuch, unter dem die Kompressionsrate gespeichert ist. Standardwert ist „reduction_multiplier“.
- „split“ (optional): Das Trennzeichen, das verwendet wird, um den Text in Sätze aufzuteilen. Standardwert ist „.“.

5.13 calculate_compression()

```
1  def calculate_compression(input_dict, total_tokens_col,
2  current_tokens_col, desired_compression_rate):
3      # Berechne die aktuelle Kompressionsrate
4      input_dict['current_compression_rate'] =
5      input_dict[current_tokens_col] / input_dict[total_tokens_col]
6      # Berechne die Differenz zur gewuenschten Kompressionsrate
7      input_dict['compression_difference'] =
8      input_dict[desired_compression_rate] -
9      input_dict['current_compression_rate']
10     # Berechne den Reduktionsmultiplikator
11     input_dict['reduction_multiplier'] =
12     input_dict[desired_compression_rate] /
13     input_dict['current_compression_rate']
14     return input_dict
```

Listing 5.13: Informationen zur Kompressionsrate berechnen

Beschreibung: Die Funktion „calculate_compression“ aus Listing 5.13 berechnet die aktuelle Kompressionsrate, die Differenz zur gewünschten Kompressionsrate und den Reduktionsmultiplikator basierend auf den im übergebenen Wörterbuch enthaltenen Werten für die Gesamtanzahl der Tokens, die aktuelle Anzahl der Tokens und die gewünschte Kompressionsrate.

Eingabe:

- „input_dict“: Ein Wörterbuch, das die relevanten Informationen enthält, um die Kompression zu berechnen.
- „total_tokens_col“: Der Schlüssel im Wörterbuch, unter dem die Gesamtanzahl der Tokens gespeichert ist.
- „current_tokens_col“: Der Schlüssel im Wörterbuch, unter dem die aktuelle Anzahl der Tokens gespeichert ist.
- „desired_compression_rate“: Der Schlüssel im Wörterbuch, unter dem die gewünschte Kompressionsrate gespeichert ist.

Ausgabe: Die Funktion calculate_compression gibt das übergebene Wörterbuch zurück, das um drei zusätzliche Schlüssel-Wert-Paare erweitert wurde:

- „current_compression_rate“: Die aktuelle Kompressionsrate, berechnet als Verhältnis der aktuellen Anzahl der Tokens zur Gesamtanzahl der Tokens.
- „compression_difference“: Die Differenz zwischen der gewünschten Kompressionsrate und der aktuellen Kompressionsrate.
- „reduction_multiplier“: Der Reduktionsmultiplikator, der angibt, um welchen Faktor die Textlänge reduziert werden muss, um die gewünschte Kompressionsrate zu erreichen.

5.14 execute_text_gen()

```
1  def execute_text_gen(dictionary, split='\. ', seed=10):
2      # Kopieren Sie das Woerterbuch fuer Manipulationen
3      dictionary_copy = dictionary.copy()
4
5      # Fuehre den Text-Rank-Algorithmus aus
6      dictionary_copy = text_rank_algo(dictionary_copy, split=split)
7
8      # Berechne die Kompression
9      dictionary_copy = calculate_compression(dictionary_copy,
10     'tokens_gesamt', 'token_text_rank', 'desired_compression_rate')
11
12     # Erstelle das Modell
13     create_model(dictionary_copy)
14
15     # Paraphrasiere den Text
16     dictionary_copy = paraphrase_of_text(dictionary_copy,
17     text_name='text_rank_text', split=split)
18
19     # Berechne die endgueltige Kompressionsrate
20     dictionary_copy['ent_com_rate'] =
21     dictionary_copy['laenge Zusammenfassung'] /
22     dictionary_copy['tokens_gesamt']
23
24     # Berechne erneut die Kompression
25     dictionary_copy = calculate_compression(dictionary_copy,
26     'tokens_gesamt', 'laenge Zusammenfassung',
27     'desired_compression_rate')
28
29     # Fuehre erneut den Text-Rank-Algorithmus aus
30     dictionary_copy = text_rank_algo(dictionary_copy,
31     random_T=False, column='Zusammenfassung')
32
33     # Berechne die endgueltige Kompressionsrate erneut
34     dictionary_copy['ent_com_rate'] =
35     dictionary_copy['laenge Zusammenfassung'] /
36     dictionary_copy['tokens_gesamt']
37
38     return dictionary_copy
```

Listing 5.14: Führe alle wichtigen Funktionen durch

Eingabe: „dictionary“: Ein Wörterbuch (Dictionary) mit den Informationen zum Text und zur gewünschten Kompressionsrate.
Ausgabe: Die Funktion „execute_text_gen“ gibt ein neues Wörterbuch zurück, das mit zusätzlichen Schlüssel-Wert-Paaren erweitert wurde, um die Ergebnisse der Textgenerierung und Kompressionsberechnungen zu speichern.

6 Frontend

Im Rahmen des Projekts soll ein funktionales Frontend erstellt werden. Dieses wurde mittels streamlit umgesetzt. Dabei sollen Funktionalitäten, wie das Hochladen von Texten, sowie Textdateien in .pdf, .docx und .txt Formaten und das Einsprechen von Texten vorhanden sein. Danach soll eine Auswahl zu der Textklassifikation und/oder der Zusammenfassung von Texten, unter Angabe der gewünschten Kompressionsrate, erfolgen. Letztendlich erscheinen die Ergebnisse zu den ausgewählten Aspekten.

6.1 Python Bibliotheken für das Frontend

```
1  import streamlit as st
2  import plotly.graph_objects as go
3  from keras.models import load_model
4  from tkinter import Tk
5  from tkinter.filedialog import askopenfilename
6
7  import joblib
8  from joblib import load
9  import numpy as np
10 import streamlit as st
11 import PyPDF2
12 from PyPDF2 import PdfReader
13 from docx import Document
14 import time
15
16
17 # Laden der Imports fuer die Speech to Text Funktion
18 import speech_recognition as sr
19 from time import ctime # get time details
20 import time
21 import playsound
22 import os
23 import random
24 from gtts import gTTS
25
26 from python_scripts.transcript_creation import modell_speak, record_audio
27
28 ################################################
29
30 # Laden der Imports fuer die Zusammenfassung
31 import re
32 import networkx as nx
33 import spacy
34 from summa import keywords
35 from summa.summarizer import summarize
36
37 # Laden des Spacy-Modells
38 import evaluate
39 import nltk
```

```
40 from nltk.tokenize import sent_tokenize
41 from transformers import AutoModelForSeq2SeqLM, AutoTokenizer,
       PreTrainedTokenizerFast
42 import matplotlib.pyplot as plt
43 import math
44 from tqdm import tqdm
45 from transformers import pipeline
46 import torch
47 from deepmultilingualpunctuation import PunctuationModel
48
49 from python_scripts.zusammenfassung import execute_text_gen
```

Listing 6.1: Imports der Python Bibliotheken und Laden der Modelle aus anderen Dateien

6.2 Laden der Modelle

```
1 ############ Laden der Modelle ###############
2 model_text_korrigieren = PunctuationModel()
3 ## Laden des Modells fuer die Klassifikation
4 model_classification = load_model("classification_modells/neuro_net_1.h5"
      )
5 vectorizer = load("classification_modells/vectorizer_1.joblib")
6
7 ## Laden des Modells fuer die Zusammenfassung
8 for model_name in ['NICFRU/bart-base-paraphrasing-science','NICFRU/bart-
      base-paraphrasing-news','NICFRU/bart-base-paraphrasing-story','NICFRU
      /bart-base-paraphrasing-review']:
9     tokenizer_zusammenfassung = AutoTokenizer.from_pretrained(model_name)
10    paraphrase_zusammenfassung = pipeline("text2text-generation", model=
          model_name)
```

Listing 6.2: Einladen der Modelle

6.3 Angabe des Designs von Schrift und Knöpfen

```
1 # Schriftgroesse fuer Buttons anpassen
2 st.write('<style>div.row-widget button{font-size: 30px !important;}</
      style>', unsafe_allow_html=True)
3
4 # Schriftgroesse fuer Text Area anpassen
5 st.write('<style>div.row-widget textarea{font-size: 35px !important;}</
      style>', unsafe_allow_html=True)
6
7 m = st.markdown("""
8 <style>
9 div.stButton > button:nth-of-type(2) {
10     background-color: #ce1126;
11     color: white;
12     height: 3em;
13     width: 12em;
```

```
14      border-radius:10px;
15      border:3px solid #000000;
16      font-size:20px;
17      font-weight: bold;
18      margin: auto;
19      display: block;
20  }
21
22  div.stButton > button:hover {
23    background:linear-gradient(to bottom, #ce1126 5%, #ff5a5a 100%);
24    background-color:#ce1126;
25  }
26      /* 1st button */
27      .element-container:nth-child(3) {
28        left: 10px;
29        top: -60px;
30      }
31  div.stButton > button:active {
32    position:relative;
33    top:3px;
34  }
35
36  </style>""", unsafe_allow_html=True)
```

Listing 6.3: Angabe des Designs von Schrift und Knöpfen

6.4 extract_text_from_element(element)

```
1  def extract_text_from_element(element):
2      # Extrahiere den Text aus einem Streamlit-Element
3      if hasattr(element, 'text'):
4          return element.text
5      elif isinstance(element, (list, tuple)):
6          return ' '.join([extract_text_from_element(item) for item in
               element])
7      elif isinstance(element, dict):
8          return ' '.join([extract_text_from_element(item) for item in
               element.values()])
9      else:
10         return str(element)
```

Listing 6.4: Funktion zum Extrahieren von Text

Diese Funktion extrahiert den Text aus dem Eingabefeld. Folgende Schritte werden dabei ausgeführt:

Zuerst wird geprüft, ob das übergebene Element das Attribut text hat. Wenn dies der Fall ist, wird der Text des Elements zurückgegeben. Dies ist der Basisfall, der den rekursiven Prozess beendet. Falls das Element keine text-Eigenschaft hat, wird überprüft, ob es sich um eine Liste oder ein Tuple handelt. Wenn ja, wird der Text aller Elemente in der Liste oder im Tuple mithilfe einer List Comprehension gesammelt und mit einem Leerzeichen getrennt.

Wenn das Element ein Dictionary ist, wird der Text aller Werte im Dictionary gesammelt und mit einem Leerzeichen getrennt. Wenn keine der vorherigen Bedingungen erfüllt ist, wird das Element in eine Zeichenkette (String) umgewandelt und zurückgegeben

6.5 extract_text_from_page(page)

```
1 def extract_text_from_page(page):
2     # Extrahiere den Text aus einer Streamlit-Seite
3     text = ''
4     for element in page:
5         text += extract_text_from_element(element)
6     return text
```

Listing 6.5: Extrahieren des Texts des gesamten Frontends

Diese Funktion extrahiert den Text aus einer gesamten Streamlit-Seite, die in Form einer Liste von Elementen vorliegt. Sie nutzt die vorher definierte extract_text_from_element-Funktion. Sie initialisiert eine leere Zeichenkette text, in der der gesamte extrahierte Text gespeichert wird. Dann durchläuft sie jede Element in der übergebenen page. Für jedes Element ruft sie die extract_text_from_element-Funktion auf, um den Text daraus zu extrahieren. Der extrahierte Text wird der Variablen text hinzugefügt. Am Ende der Schleife wird die gesamte extrahierte Textmenge zurückgegeben.

6.6 Möglichkeiten zur Eingabe von Dateien

```
1  def read_docx_file(file_path):
2      doc = Document(file_path)
3      paragraphs = [p.text for p in doc.paragraphs]
4      return " ".join(paragraphs)
5
6  def read_txt_file(file_path):
7      with open(file_path, 'r') as file:
8          content = file.read()
9      return content
10
11 def read_pdf_file(file_path):
12     content = ""
13     with open(file_path, 'rb') as file:
14         pdf = PyPDF2.PdfReader(file)
15         for page in pdf.pages:
16             content += page.extract_text()
17     return content
18
19 def read_pdf(file):
20     data_folder = "Testfiles"
21     file_path = os.path.join(data_folder, file)
22     pdf = PdfReader(file_path)
```

```
23      text = ""
24      for page in range(len(pdf.pages)):
25          text += pdf.pages[page].extract_text()
26      return text
27
28  def read_docx(file):
29      data_folder = "Testfiles"
30      file_path = os.path.join(data_folder, file)
31      doc = Document(file_path)
32      text = ""
33      for paragraph in doc.paragraphs:
34          text += paragraph.text + "\n"
35      return text
36
37  def read_txt(file):
38      data_folder = "Testfiles"
39      file_path = os.path.join(data_folder, file.name)
40      with open(file_path, 'r', encoding='utf-8') as f:
41          text = f.read()
42      return text
```

Listing 6.6: Verschiedene Möglichkeiten zum Einlesen von Texten

Die gegebenen Funktionen ermöglichen das Lesen von Textinhalten aus verschiedenen Dateiformaten: .docx, .txt und .pdf. Sie fassen den Text aus den Dateien zusammen und geben ihn als eine zusammenhängende Zeichenkette (String) zurück.

read_docx_file(file_path): Diese Funktion liest den Inhalt einer .docx-Datei, die durch den übergebenen file_path spezifiziert ist. Sie verwendet die Python-Bibliothek python-docx, um den Inhalt zu extrahieren. Die Funktion gibt den Text als eine einzige Zeichenkette zurück, indem sie die einzelnen Absätze in der Datei mit Leerzeichen verbindet.

read_txt_file(file_path): Diese Funktion liest den Inhalt einer .txt-Datei, die durch den übergebenen file_path spezifiziert ist. Sie öffnet die Datei, liest den gesamten Inhalt und gibt ihn als Zeichenkette zurück.

read_pdf_file(file_path): Diese Funktion liest den Inhalt einer .pdf-Datei, die durch den übergebenen file_path spezifiziert ist. Sie verwendet die Python-Bibliothek Py PDF2, um den Text von jeder Seite der PDF-Datei zu extrahieren. Die Funktion gibt den Text als eine einzige Zeichenkette zurück, indem sie den Text von jeder Seite miteinander verbindet.

read_pdf(file): Diese Funktion liest den Inhalt einer .pdf-Datei, deren Dateiname durch das Argument file angegeben wird. Sie verwendet die Python-Bibliothek PyPDF2, um den Text von jeder Seite der PDF-Datei zu extrahieren. Die Funktion gibt den Text als eine einzige Zeichenkette zurück, indem sie den Text von jeder Seite miteinander verbindet.

read_docx(file): Diese Funktion liest den Inhalt einer .docx-Datei, deren Dateiname

durch das Argument file angegeben wird. Sie verwendet die Python-Bibliothek python-docx, um den Inhalt zu extrahieren. Die Funktion gibt den Text als eine einzige Zeichenkette zurück, indem sie den Text jedes Absatzes mit einem Zeilenumbruch verbindet.

read_txt(file): Diese Funktion liest den Inhalt einer .txt-Datei, deren Dateiname durch das Argument file angegeben wird. Die Funktion öffnet die Datei mit UTF-8-Encoding, liest den gesamten Inhalt und gibt ihn als Zeichenkette zurück.

6.7 summarize_text(text, compression_rate,predicted_class)

```
1
2 def summarize_text(text, compression_rate,predicted_class):
3     # Hier erfolgt die Zusammenfassung des Textes unter Beruecksichtigung
          der Kompressionsrate
4     # Implementiere hier deine Logik zur Zusammenfassung des Textes
5     dic=execute_text_gen({'classification':predicted_class,'text':text,'
          compression':compression_rate*0.01})
6     paraphrase_zusammenfassung = dic['text_rank_text_2']
7     text_compression_rate=round(dic['ent_com_rate']*100,2)
8     print('Text: ',paraphrase_zusammenfassung,'Kompression: ' ,
          text_compression_rate)
9     return paraphrase_zusammenfassung, text_compression_rate
```

Listing 6.7: Funktion zur Zusammenfassung des Texts

Diese Funktion führt eine Zusammenfassung eines Textes unter Berücksichtigung einer vorgegebenen Kompressionsrate durch. Dabei werden detailliert folgende Schritte durchgeführt:
Die Funktion summarize_text(text, compression_rate, predicted_class) erhält drei Argumente: text ist der zu summarisierende Text, compression_rate ist der Kompressionsfaktor in Prozent und predicted_class ist die vorhergesagte Klasse oder Kategorie des Textes (wahrscheinlich im Kontext von maschinellem Lernen oder Klassifikation).
Die Funktion verwendet eine externe Funktion execute_text_gen, um die tatsächliche Zusammenfassung durchzuführen. Es verwendet den text und die predicted_class, um den Text zusammenzufassen, wobei die Kompressionsrate compression_rate berücksichtigt wird.
Die Zusammenfassung des Textes wird in der Variablen paraphrase_zusammenfassung gespeichert.
Die tatsächliche Kompressionsrate wird in der Variablen text_compression_rate gespeichert, die auf zwei Dezimalstellen gerundet wird.
Die Funktion druckt dann die Zusammenfassung und die Kompressionsrate auf der Konsole und gibt die Zusammenfassung (paraphrase_zusammenfassung) und die Kompressionsrate (text_compression_rate) als Tupel zurück.

6.8 classify_text(text)

```
1 def classify_text(text):
2     new_text_features = vectorizer.transform([text])
3     predictions = model_classification.predict(new_text_features.toarray
          ())
4     predicted_class = np.argmax(predictions, axis=1)
5     predicted_probability = np.max(predictions, axis=1)
6     return predicted_class, predicted_probability
```

Listing 6.8: Funktion zur Durchführung der Klassifikation

Zusammengefasst führt die Funktion eine Klassifikation für den gegebenen Text durch, indem sie den Text in ein numerisches Format umwandelt und ein zuvor trainiertes Modell verwendet, um die Klassenlabel und die Wahrscheinlichkeiten für diese Klassen zu bestimmen. Die Funktion classify_text(text) erhält einen Text als Eingabe (text), den sie klassifizieren soll. vectorizer ist ein zuvor trainierter Vektorisierer (wie CountVectorizer oder TF-IDF-Vectorizer), der verwendet wird, um den Text in ein numerisches Format umzuwandeln, damit er von einem Klassifikationsmodell verarbeitet werden kann. vectorizer.transform([text]) wandelt den gegebenen text in ein numerisches Feature-Vektor um. Das [text] ist eine Liste mit nur einem Element, da der Vektorisierer in der Regel eine Liste von Texten erwartet, auch wenn es nur einen gibt. model_classification ist ein trainiertes Klassifikationsmodell, das auf den numerischen Features arbeitet und Klassenlabels vorhersagt. Es könnte ein Decision Tree, Random Forest, Support Vector Machine (SVM) oder ein anderes Modell sein. model_classification.predict(new_text_features.toarray()) verwendet das trainierte Klassifikationsmodell, um Vorhersagen für den gegebenen Text zu machen. new_text_features.toarray() wandelt die numerischen Features in ein 2D-Array um, das vom Modell verstanden wird. predictions enthält die Vorhersagen des Modells, eine Wahrscheinlichkeitsverteilung der Klassen. np.argmax(predictions, axis=1) ermittelt den Index der vorhergesagten Klasse mit der höchsten Wahrscheinlichkeit (argmax). Es wird der vorhergesagte Klassenindex (predicted_class) zurückgegeben. np.max(predictions, axis=1) ermittelt den maximalen Wahrscheinlichkeitswert aus den Vorhersagen und gibt ihn als predicted_probability zurück. Die Funktion gibt schließlich ein Tupel mit den vorhergesagten Klassenindex (predicted_class) und der dazugehörigen Wahrscheinlichkeit (predicted_probability) zurück.

6.9 style_button_row(clicked_button_ix, n_buttons)

```
1 def style_button_row(clicked_button_ix, n_buttons):
2     def get_button_indices(button_ix):
3         return {
4             'nth_child': button_ix,
```

```
            'nth_last_child': n_buttons - button_ix + 1
        }

    clicked_style = """
    div[data-testid*="stHorizontalBlock"] > div:nth-child(%(nth_child)s):
        nth-last-child(%(nth_last_child)s) button {
        border-color: rgb(255, 75, 75);
        color: rgb(255, 75, 75);
        box-shadow: rgba(255, 75, 75, 0.5) 0px 0px 0px 0.2rem;
        outline: currentcolor none medium;
    }
    """
    unclicked_style = """
    div[data-testid*="stHorizontalBlock"] > div:nth-child(%(nth_child)s):
        nth-last-child(%(nth_last_child)s) button {
        pointer-events: none;
        cursor: not-allowed;
        opacity: 0.65;
        filter: alpha(opacity=65);
        -webkit-box-shadow: none;
        box-shadow: none;
    }
    """
    style = ""
    for ix in range(n_buttons):
        ix += 1
        if ix == clicked_button_ix:
            style += clicked_style % get_button_indices(ix)
        else:
            style += unclicked_style % get_button_indices(ix)
    st.markdown(f"<style>{style}</style>", unsafe_allow_html=True)
```

Listing 6.9: Dynamisches Styling von Schaltflächen

Die Funktion get_button_indices(button_ix) ist eine interne Hilfsfunktion, die den Index der geklickten Schaltfläche (button_ix) verwendet, um die Indizes für CSS-Styling-Selektoren zu berechnen. Es verwendet dabei die Pseudoklassen nth-child und nth-last-child, um das entsprechende Styling auf die geklickte Schaltfläche und die restlichen Schaltflächen anzuwenden. clicked_style und unclicked_style sind CSS-Stilvorlagen, die definiert, wie die Schaltflächen aussehen sollen, wenn sie angeklickt (clicked_style) oder nicht angeklickt (unclicked_style) sind. In der Hauptfunktion wird eine leere Zeichenkette style initialisiert, die später die gesamten CSS-Stilregeln für alle Schaltflächen enthalten wird. Die Funktion durchläuft eine Schleife von ix = 1 bis ix = n_buttons, um jede Schaltfläche zu behandeln. Wenn ix dem clicked_button_ix entspricht, wird die CSS-Stilvorlage clicked_style auf die Schaltfläche mit dem entsprechenden Index angewendet, indem die berechneten Indizes mit get_button_indices(ix) eingesetzt werden. Wenn ix nicht dem clicked_button_ix entspricht, wird die CSS-Stilvorlage unclicked_style auf die Schaltfläche mit dem entsprechenden Index angewendet, indem die berechneten Indizes mit get_button_indices(ix) eingesetzt werden. Schließlich wird der gesamte generierte CSS-Stil als HTML <style>-Tag über st.markdown()

in der Streamlit-Anwendung angewendet, um die Schaltflächen entsprechend zu stylen. Zusammengefasst ermöglicht die Funktion das dynamische Styling einer Zeile von Schaltflächen, indem sie die Optik der angeklickten und nicht angeklickten Schaltflächen anpasst, abhängig vom Index der geklickten Schaltfläche und der Gesamtzahl der Schaltflächen.

6.10 execute_the_classification_and_summary(input_text,compression _rate,classification, summary)

```
1
2  def execute_the_classification_and_summary(input_text,compression_rate,
       classification, summary):
3      predicted_class, predicted_probability = classify_text(input_text)
4      predicted_probability = round(predicted_probability[0]*100, 2)
5      if predicted_class == 0:
6          predicted_class = "Scientific Paper"
7      elif predicted_class == 1:
8          predicted_class = "News"
9      elif predicted_class == 2:
10         predicted_class = "Review"
11     else:
12         predicted_class = "Story"
13
14
15     if predicted_probability >= 95:
16         if classification:
17             st.subheader("Classification")
18             st.write(f"The class calculated by the model is:  \"{
                   predicted_class}\". The model is very confident with a
                   probability of {predicted_probability}%.")
19         if summary:
20             paraphrase_zusammenfassung, text_compression_rate =
                   summarize_text(input_text, compression_rate,
                   predicted_class)
21             st.subheader("Summary")
22             st.write("This is a summary of the input text has a
                   compression rate of {}%:".format(text_compression_rate))
23             st.write(paraphrase_zusammenfassung)
24     else:
25         st.write(f"Attention! The model is not entirely certain. The
               class calculated by the model is: \"{predicted_class}\".
               However, the model predicts this class with a probability of
               only {predicted_probability}%.")
```

Listing 6.10: Funktion zur Ausgabe der Ergebnisse auf der Oberfläche

Die Funktion namens execute_the_classification_and_summary(input_text,compression_ rate,classification, summary) führt eine Klassifikation und eine Zusammenfassung für den gegebenen Text (input_text) durch, wobei die Optionen für die Kompressionsrate (compression_rate), die Klassifikation (classification) und die Zusammenfassung (summary) berück-

sichtigt werden. Die Funktion nimmt als Eingabe den input_text, die gewünschte Kompressionsrate (compression_rate), eine Flagge für die Klassifikation (classification) und eine Flagge für die Zusammenfassung (summary) entgegen. Die Funktion ruft die Funktion classify_text(input_text) auf, um den gegebenen Text zu klassifizieren und die vorhergesagte Klasse (predicted_class) sowie die Wahrscheinlichkeit (predicted_probability) der Vorhersage zu erhalten. Die predicted_probability wird in Prozent (auf zwei Dezimalstellen gerundet) umgerechnet, um sie besser zu interpretieren. Basierend auf der vorhergesagten Klasse wird predicted_class in einen verständlichen Klassenbezeichner umgewandelt (z. B. von numerischen Klassenindizes zu sprechenden Klassennamen). Wenn die predicted_probability größer oder gleich 95% ist (hohe Wahrscheinlichkeit), wird der nächste Schritt ausgeführt:

- Wenn die Klassifikationsflagge classification aktiviert ist, wird die vorhergesagte Klasse und die Wahrscheinlichkeit auf der Streamlit-Oberfläche als Text ausgegeben.
- Wenn die Zusammenfassungsflagge summary aktiviert ist, wird die Zusammenfassungsfunktion summarize_text(input_text, compression_rate, predicted_class) aufgerufen, um eine Zusammenfassung des Eingabetextes mit der gegebenen Kompressionsrate zu erstellen.
- Die Zusammenfassung und die Kompressionsrate werden auf der Streamlit-Oberfläche angezeigt.

Wenn die predicted_probability unter 95% liegt (geringe Wahrscheinlichkeit), wird der nächste Schritt ausgeführt:

- Eine Warnung wird auf der Streamlit-Oberfläche ausgegeben, die besagt, dass das Modell nicht sicher ist.
- Die vorhergesagte Klasse und die Wahrscheinlichkeit werden auf der Streamlit- Oberfläche als Text ausgegeben.

Insgesamt führt die Funktion eine Klassifikation und eine Zusammenfassung durch und gibt die Ergebnisse basierend auf den übergebenen Optionen auf der Streamlit-Oberfläche aus. Je nach Modellvertrauen und Benutzereinstellungen kann die Klassifikation und Zusammenfassung angezeigt werden.

6.11 split_into_batches(text, batch_size=500)

```
1 def split_into_batches(text, batch_size=500):
2     # First, tokenize the text
3     nltk.download('punkt')
4     tokens = nltk.word_tokenize(text)
5
```

```
6       # Split into batches
7       batches = [tokens[i:i + batch_size] for i in range(0, len(tokens),
            batch_size)]
8
9       # Join tokens back into strings
10      batches = [' '.join(batch) for batch in batches]
11
12      return batches
```

Listing 6.11: Funktion zum Einteilen der Eingabe in Batches

Die Funktion split_into_batches(text, batch_size=500) teilt den gegebenen Text in Batches (Teilmengen) auf, um eine zu lange Textsequenz in kleinere, handhabbare Abschnitte zu zerlegen. Hier ist eine Beschreibung des Codes:
Die Funktion verwendet die Natural Language Toolkit (NLTK) und benötigt möglicherweise die Daten für die Tokenisierung, daher wird nltk.download('punkt') ausgeführt, um sicherzustellen, dass die erforderlichen Ressourcen verfügbar sind. Der Text wird in Tokens aufgeteilt, wobei Tokenisierung den Text in einzelne Wörter oder Satzteile (Tokens) aufspaltet. Anschließend wird der Text in Batches (Teilmengen) aufgeteilt. Jeder Batch enthält batch_size Anzahl von Tokens. Dabei wird die Funktion nltk.word_tokenize verwendet, um den Text in Tokens zu teilen, und eine List Comprehension erstellt die Batches. Die Batches werden dann wieder in Zeichenketten umgewandelt, indem die Tokens jedes Batches wieder zu einem zusammenhängenden Text verknüpft werden. Schließlich gibt die Funktion die Liste der Batches zurück.

6.12 is_file(path)

```
1   def is_file(path):
2       return os.path.isfile(path)
```

Listing 6.12: Funktion zur Überprüfung eines existierenden Pfades

Die Funktion is_file(path) überprüft, ob der gegebene Pfad (path) zu einer existierenden Datei führt. Hier ist eine Beschreibung des Codes: Die Funktion verwendet die os.path.isfile()-Funktion, um zu überprüfen, ob der Pfad, der als Argument path übergeben wird, zu einer existierenden Datei führt. Die os.path.isfile()-Funktion gibt True zurück, wenn der angegebene Pfad eine existierende Datei ist, andernfalls gibt sie False zurück. Die Funktion gibt das Ergebnis der Überprüfung zurück, d.h., True, wenn path eine Datei ist, und False, wenn path keine Datei ist.

6.13 main()

```
1  def main():
2
3      # Fuehre eine Wartezeit von wait_time Sekunden durch
4      #time.sleep(30)
5      global content
6      try:
7          if content != "":
8              content=content
9          else:
10             content = ""
11     except:
12         content = ""
13     # Texteingabe
14
15     # text_vorlesen = "Welcome to Syntex, to proceed further with the
           screenreader function, tap the big red botton on the left in the
           middle of your screen"
16     # modell_speak(text_vorlesen)
17
18
19     st.title("SynTex")
20     file = st.file_uploader("Please select a document", type=["pdf", "
           docx", "txt"])
21     print(file)
22
23     # Add custom CSS style
24     st.markdown(
25         """
26         <style>
27         .red-button {
28             background-color: red;
29             color: white;
30             padding: 0.5rem 2rem;
31             font-size: 1.5rem;
32             border-radius: 0.5rem;
33             border: none;
34             cursor: pointer;
35         }
36         </style>
37         """,
38         unsafe_allow_html=True
39     )
40
41     # if st.button("Upload a document",key=1,type='primary'):
42     #     root = Tk()
43     #     root.withdraw()
44     #     file_path = askopenfilename(filetypes=[("Textdokumente", "*.txt
           "), ("Word-Dokumente", "*.docx"), ("PDF-Dateien", "*.pdf")])
45
46     #     if file_path:
47     #         if file_path.endswith('.docx'):
48     #             input_text = read_docx_file(file_path)
49     #             print(input_text)
50     #         elif file_path.endswith('.txt'):
51     #             input_text = read_txt_file(file_path)
```

```
52      #             print(input_text)
53      #         elif file_path.endswith('.pdf'):
54      #             input_text = read_pdf_file(file_path)
55      #             print(input_text)
56      #         else:
57      #             print("Ungueltiger Dateityp. Nur DOCX-, TXT- und PDF-
            Dateien werden unterstuetzt.")
58      #     else:
59      #         print("Keine Datei ausgewaehlt.")
60
61
62
63
64      if file is not None:
65
66          content = ""
67
68          try:
69              file_type = file.type
70              if file_type == 'application/pdf':
71                  content = read_pdf(file)
72              elif file_type == 'application/vnd.openxmlformats-
                    officedocument.wordprocessingml.document':
73                  content = read_docx(file)
74              elif file_type == 'text/plain':
75                  content = read_txt(file)
76
77              st.header("Inhalt des Dokuments")
78              input_text = st.text_area("Ausgabe", value=content, height
                    =200, key=20)
79          except:
80              st.write("Add the absolute link into the text field to get a
                    file outside of the repo")
81
82      if st.button("Speech to Text", on_click=style_button_row, kwargs={
83          'clicked_button_ix': 2, 'n_buttons': 2
84      }):
85          st.write("Click to start recording")
86
87          content=record_audio(filename='transcript.txt')
88
89
90          input_text = st.text_area("Speech to Text an copy the text into
                the Textbox", value=content, height=200, key=30)
91
92
93
94      # Texteingabe
95      if file is None:
96          input_text = st.text_area("Enter a text", '', height=200, key=10)
97          if is_file(input_text):
98              print(True)
99              if input_text.endswith('.docx'):
100                 content = read_docx_file(input_text)
101                 print(1)
102             elif input_text.endswith('.txt'):
```

```
103                 content = read_txt_file(input_text)
104                 print(2)
105                 print(content)
106             elif input_text.endswith('.pdf'):
107                 content = read_pdf_file(input_text)
108                 print(3)
109             input_text=content
110             input_text = st.text_area("Copy the text into the above text
                    field ", input_text, height=200, key=100)
111             st.write("Copy the Data into above text field ")
112
113
114
115     if st.button("read text", on_click=style_button_row, kwargs={
116     'clicked_button_ix': 1, 'n_buttons': 2}):
117         if input_text == "":
118                 st.warning("Please enter a text first")
119                 modell_speak("Please enter a text first.")
120         else:
121             modell_speak(input_text)
122
123
124
125
126
127     # if st.button("read text"):
128     #     if input_text == "":
129     #         st.warning("Please enter a text first")
130     #         modell_speak("Please enter a text first.")
131     #     else:
132     #         modell_speak(input_text)
133     # Dokument auswaehlen
134
135
136     # Checkboxen fuer Klassifikation und Zusammenfassung
137     classification_enabled = st.checkbox("Activate classification")
138     summarization_enabled = st.checkbox("Activate summary")
139
140     # Schieberegler fuer die Kompressionsrate
141     compression_rate = st.slider("Compression rate (%) to compress the
            text to the value", min_value=20, max_value=80, value=50, step=1)
142
143     # Knopf zum Zusammenfassen und Klassifizieren
144     if st.button("Execute",key=2,type='secondary'):
145         if input_text:
146             try:
147                 input_text_neu=''
148                 for batch in split_into_batches(input_text):
149                     input_text_neu += model_text_korrigieren.
                            restore_punctuation(batch)
150                 input_text= input_text_neu
151
152             except:
153                 pass
154             if classification_enabled or summarization_enabled:
```

```
155                 execute_the_classification_and_summary(input_text,
                        compression_rate,classification_enabled,
                        summarization_enabled)
156             else:
157                 no_box_text='Please check the Box classification and / or
                         summary to proceed further!'
158                 st.warning(no_box_text)
159                 modell_speak(no_box_text)
160         else:
161             st.warning("Please enter a text first.")
162
163     if st.button("Helper",type='primary'):
164         scren_text='''Welcome to Syntex. Please input your text in the
                text area and press the read text button to hear the text.
165         If you want to upload a document, please select the document type
                 and press the execute button.
166         You can also edit the text in the designated textbox.
167         If you want to activate the classification and the summary,
                please check the corresponding boxes.
168
169         You can adjust the compression rate with the slider.
170         To execute the classification and the summary, please press the
                execute button.
171         '''
172         modell_speak(scren_text)
173 if __name__ == '__main__':
174
175     main()
```

Listing 6.13: Funktion der main() im Frontend

Die Funktion main() ist der Einstiegspunkt der Anwendung und enthält den Hauptcode für die Benutzeroberfläche. Zuerst werden alle benötigten Bibliotheken und Funktionen importiert, und eine globale Variable content wird initialisiert. Es wird eine Benutzeroberfläche erstellt, die es Benutzern ermöglicht, entweder einen Text hochzuladen oder selbst einen Text in ein Textfeld einzugeben. Es gibt auch eine Schaltfläche Speech to Text, die es Benutzern ermöglicht, Audio aufzunehmen und in Text umzuwandeln. Es gibt Checkboxen, um die Klassifikation und Zusammenfassung zu aktivieren, sowie einen Schieberegler, um die Kompressionsrate für die Zusammenfassung anzupassen. Nach Eingabe des Textes oder Hochladen einer Datei kann der Benutzer die Execute-Schaltfläche betätigen, um die Klassifikation und/oder die Zusammenfassung auszuführen. Eine Helper-Schaltfläche gibt eine Sprachausgabe aus, die eine Erklärung und Anleitung zur Verwendung der Anwendung bietet. Die Funktion main() wird schließlich durch die Bedingung if __name__ == '__main__': aufgerufen, um das Streamlit-Webanwendungsskript zu starten. Die Benutzeroberfläche interagiert mit verschiedenen Funktionen wie read_docx_file, read_txt_file, read_pdf_file, split_into_batches, model_text_korrigieren.restore_punctuation, execute_the_classification_and_summary und modell_speak, die in anderen Teilen des Codes definiert sind und verschiedene Aufgaben wie das Lesen von Dateien, die Textklassifikation, die Text-

zusammenfassung und die Text-zu-Sprache-Umwandlung durchführen.

7 Zusatz-Features

7.1 Dokument-Upload

7.1.1 `read_docx_file()`

```
1 def read_docx_file(file_path):
2     doc = Document(file_path)
3     paragraphs = [p.text for p in doc.paragraphs]
4     return paragraphs
```

Listing 7.1: Funktion zum Einlesen von DocX-Files

Beschreibung: Diese Funktion liest den Inhalt einer .docx-Datei ein und extrahiert die Absätze daraus.
Eingabe: Der Dateipfad zur .docx-Datei.
Ausgabe: Eine Liste von Absätzen aus der .docx-Datei.

7.1.2 `read_txt_file()`

```
1 def read_txt_file(file_path):
2     with open(file_path, 'r') as file:
3         content = file.read()
4     return content
```

Listing 7.2: Funktion zum Einlesen von Textdateien

Beschreibung: Diese Funktion liest den Inhalt einer .txt-Datei ein.
Eingabe: Der Dateipfad zur .txt-Datei.
Ausgabe: Der gesamte Textinhalt der .txt-Datei.

7.1.3 `read_pdf_file()`

```
1 read_pdf_file(file_path):
2     content = ""
3     with open(file_path, 'rb') as file:
4         pdf = PyPDF2.PdfReader(file)
5         for page in pdf.pages:
6             content += page.extract_text()
7     return content
```

Listing 7.3: Funktion zum Einlesen von PDFs

Beschreibung: Diese Funktion liest den Inhalt einer .pdf-Datei ein und extrahiert den Text aus den Seiten.

Eingabe: Der Dateipfad zur .pdf-Datei.
Ausgabe: Der gesamte Textinhalt der .pdf-Datei.

7.2 Speech-to-Text-Eingabe

7.2.1 `modell_speak()`

```
1 def modell_speak(text):
2     text = str(text)
3     engine.say(text)
4     engine.runAndWait()
```

Listing 7.4: Funktion zur Sprachausgabe des übergebenen Textes

Beschreibung: Diese Funktion verwendet das Text-to-Speech-Modul, um den übergebenen Text als Sprache auszugeben.
Eingabe: Der Text, der als Sprache ausgegeben werden soll.
Ausgabe: Es erfolgt eine Sprachausgabe des übergebenen Textes.

7.2.2 `there_exists()`

```
1 def there_exists(terms, voice_data):
2     for term in terms:
3         if term.lower() in voice_data.lower():
4             return True
```

Listing 7.5: Funktion zur Überprüfung, ob bestimmte Begriffe in Spracheingabedaten enthalten sind

Beschreibung: Diese Funktion überprüft, ob bestimmte Begriffe aus der Liste `terms` im Spracheingabedaten `voice_data` enthalten sind.
Eingabe:

- `terms`: Eine Liste von Begriffen, die überprüft werden sollen.
- `voice_data`: Der Text der Spracheingabe, in dem die Begriffe gesucht werden.

Ausgabe: Die Funktion gibt `True` zurück, wenn mindestens einer der Begriffe aus der Liste `terms` im Spracheingabedaten `voice_data` enthalten ist. Ansonsten gibt sie `False` zurück.

7.2.3 `modell_speak()`

```
1 def modell_speak(audio_string):
2     tts = gTTS(text=audio_string, lang='en') # Text-zu-Sprache (Stimme)
```

```
3       r = random.randint(1, 20000000)
4       audio_file = 'audio-' + str(r) + '.mp3'
5       tts.save(audio_file) # Als MP3-Datei speichern
6       playsound.playsound(audio_file) # Wiedergabe der Audiodatei
7       print(audio_string) # Ausgabe des gesprochenen Textes
8       os.remove(audio_file) # Entfernen der Audiodatei
```

Listing 7.6: Funktion zur Sprachausgabe eines gegebenen Textes

Beschreibung: Diese Funktion erzeugt eine Sprachausgabe für den gegebenen Text audio_string.
Eingabe: Der Text, der in Sprache umgewandelt und ausgegeben werden soll. **Ausgabe:** Die Funktion erzeugt eine Sprachausgabe, die den gegebenen Text wiedergibt, und gibt den Text zusätzlich in der Konsole aus. Die temporäre Audiodatei wird nach der Wiedergabe automatisch gelöscht.

7.2.4 record_audio()

```
1   def record_audio(ask="", filename='transcription.txt'):
2       with sr.Microphone() as source: # Mikrofon als Quelle
3           # if ask:!
4           #     print(ask)
5           #audio = r.listen(source, 5, 5)  # Aufnahme des Audios ueber die
                Quelle
6           voice_data = ''
7           modell_speak("Starting transcription")
8           modell_speak("You can speak now!")
9           voice_data = r.listen(source)
10          modell_speak("Done Listening")
11
12          try:
13              voice_data = r.recognize_google(voice_data)
14              modell_speak("I repeat")
15              modell_speak(voice_data)
16              modell_speak('If incorrect press the button again')  #
                    Umwandlung des Audios in Text
17          except sr.UnknownValueError: # Fehler: Der Spracherkenner
                versteht die Eingabe nicht
18              modell_speak('I did not get that')
19          except sr.RequestError:
20              modell_speak('Sorry, the service is down')
21          #print(">>", voice_data.lower()) # Ausgabe des vom Benutzer
                gesprochenen Textes
22
23          # Den Text in eine Datei schreiben
24          with open(filename, 'w') as file:
25              file.write(voice_data)
26          #print(voice_data)
27
28          return voice_data
```

Listing 7.7: Funktion zur Aufnahme von Audiodaten und Umwandlung in Text

Beschreibung: Diese Funktion ermöglicht die Aufnahme von Audiodaten über ein angeschlossenes Mikrofon und wandelt die gesprochene Sprache in Text um.
Eingabe: `ask` ist ein optionales Argument, das als Frage oder Aufforderung vor der Aufnahme ausgegeben werden kann. `filename` ist der Dateiname, in den der transkribierte Text geschrieben werden soll.
Ausgabe: Der von der Benutzerin oder dem Benutzer gesprochene Text wird in Textform zurückgegeben und zusätzlich in die Datei `filename` geschrieben. Falls der Spracherkenner die Eingabe nicht versteht oder ein Fehler auftritt, werden entsprechende Meldungen ausgegeben.