NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Thesis title in English

DIMITRIOS T. ORINOS

Supervisors: **Manolis Koubarakis,** Professor
Konstantinos Plas, Postgraduate student

ATHENS

MARCH 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Τίτλος της πτυχιακής εργασίας στα Ελληνικά

ΔΗΜΗΤΡΙΟΣ Θ. ΟΡΕΙΝΟΣ

Επιβλέποντες: **Μανώλης Κουμπαράκης,** Καθηγητής
Κωνσταντίνος Πλάς, Μεταπτυχιακός Φοιτητής

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2024

BSc THESIS

Thesis title in English

DIMITRIOS T. ORINOS

S.N.: 7115132100006

SUPERVISORS: **Manolis Koubarakis,** Professor
Konstantinos Plas, Postgraduate student

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Τίτλος της πτυχιακής εργασίας στα Ελληνικά

ΔΗΜΗΤΡΙΟΣ Θ. ΟΡΕΙΝΟΣ

Α.Μ.: 7115132100006

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Μανώλης Κουμπαράκης,** Καθηγητής
Κωνσταντίνος Πλάς, Μεταπτυχιακός Φοιτητής

ABSTRACT

The main goal of my thesis is the creation of a question answering engine that is able to answer simple questions from the dataset SimpleQuestions, which is relying on the Freebase and it is translated into Wikidata, where as a result each question of the dataset consists of an entity id and a relation id from this translation. In order to achieve this goal, firstly i find through SPARQL the entity label for each question by using the entity id which was resulted from the translation of the SimpleQuestions dataset to Wikidata. Secondly i create a relation vocabulary which consists of the unique relation ids that exists in the training and the validation dataset of the SimpleQuestions dataset. Later i find the index of each unique relation id of the relation vocabulary in it. Next, i create a neural network consisting of a bert transformer layer, a linear layer, an activation and an dropout layer with the output neurons equal to 129, as long as the length of the relation vocabulary. The relation indexes of the unique relation ids combined with this network conduce in order to predict the index for the relation id for each question of training and validation dataset of SimpleQuestions dataset. Also i create a similar neural network with two outputs one for the entity label start word index and one for entity label end word index with output neurons equal to 33 each, which is the biggest length is found for all SimpleQuestions dataset questions. In order to find the entity label start and entity label end index for each training and the validation dataset entity label, i split the training and the validation dataset questions into parts with length equal to its corresponding entity labels length and i create for each subset (training, validation) question by using the equal question part with the entity label, a list that consists of 0 and 1, where 1 is the case when the entity label word belongs to the entity label and the corresponding question and 0 if not. I also tried to find and later predict these indexes through text similarity either with cosine similarity by using spacy library combined with the Sentence Transformer model of the Transformers library or jaccard similarity metric by using only spacy library. These entity span indexes combined the entity span neural network that i created conduce in order to predict the start and the end index of the entity label in the corresponding question for each SimpleQuestions dataset question. Finally i create some versions of question answering engine by using spacy library, Sentence Transformer combined with the cosine similarity metric and jaccard similarity metric. Finally the best entity span evaluation scores are achieved for the methodology of finding entity span indexes by creating lists consisting of 0 and 1 only for SimpleQuestions training and validation dataset questions that their entity label consists a part of their wording. Also the biggest percentage of totally correct answered questions are observed for the QA engine with python library spacy for the methodology of finding entity span indexes (start,end) through cosine similarity metric and the methodology of finding entity span indexes by creating lists consisting of 0 and 1 in the case of training and validating all SimpleQuestions training and validation dataset questions respectively, regardless if their entity label consists a part of their wording.

SUBJECT AREA: Artificial Intelligence(AI)-Natural Language Processing(NLP)

KEYWORDS: Wikidata, Question Answering engine, entity span index prediction, Relation id prediction, Text similarity

ΠΕΡΙΛΗΨΗ

Ο κύριος στόχος της διπλωματικής μου εργασίας είναι η δημιουργία μιας μηχανής απάντησης ερωτήσεων που είναι σε θέση να απαντήσει σε απλές ερωτήσεις από το σύνολο δεδομένων SimpleQuestions, η οποία βασίζεται στη βάση γνώσεων Freebase και μεταφράζεται στη βάση γνώσεων της Wikidata, όπου ως αποτέλεσμα κάθε ερώτηση του συνόλου δεδομένων SimpleQuestions αποτελείται από ένα entity id και ένα relation id από αυτή τη μετάφραση. Για την επίτευξη αυτού του στόχου, αρχικά βρήκα μέσω της SPARQL το entity label για κάθε ερώτηση, χρησιμοποιώντας το entity id το οποίο προέκυψε από τη μετάφραση του συνόλου δεδομένων SimpleQuestions στη Wikidata. Δεύτερον δημιούργησα ένα relation vocabulary το οποίο αποτελείται από τα μοναδικά relation ids που υφίστανται στα σύνολα εκπαίδευσης και επαλήθευσης του συνόλου δεδομένων SimpleQuestions. Αργότερα βρήκα τη θέση (index) του κάθε μοναδικού relation id του relation vocabulary σε αυτό. Έπειτα δημιούργησα ένα νευρωνικό δίκτυο που αποτελείται από ένα επίπεδο bert transformer, ένα γραμμικό (linear) επίπεδο, ένα επίπεδο συνάρτησης ενεργοποίησης (activation) και ένα dropout επίπεδο αριθμό νευρώνων εξόδου ίσο με 129, όσο το μήκος του relation vocabulary. Οι σχεσιακοί (relation) δείκτες των μοναδικών relation ids σε συνδυασμό με αυτό το νευρωνικό δίκτυο συμβάλλουν στο να γίνει η πρόβλεψη το δείκτη για το relation id της κάθε ερώτησης του συνόλου εκπαίδευσης και επαλήθευσης του συνόλου δεδομένων (dataset) SimpleQuestions. Επίσης δημιούργησα ένα όμοιο νευρωνικό δίκτυο με δύο εξόδους, μία για τον δείκτη της αρχικής λέξης του entity label στην αντίστοιχη ερώτηση και μία για τον δείκτη της τελικής λέξης του entity label με αριθμό νευρώνων ίσο με 33 η κάθε έξοδος, που είναι το μεγαλύτερο μήκος ερώτησης από όλες τις ερωτήσεις του συνόλου δεδομένων SimpleQuestions. για να γίνει η πρόβλεψη των δεικτών της αρχικής και της τελικής λέξης του entity label μέσα στην ερώτηση για κάθε ερώτηση του συνόλου δεδομένων SimpleQuestions. Προκειμένου να βρω τον δείκτη της αρχικής και της τελικής λέξης για κάθε entity label του συνόλου εκπαίδευσης και επαλήθευσης, χώρισα τις ερωτήσεις των συνόλων εκπαίδευσης και επαλήθευσης σε τμήματα με μήκος ίσο με αυτό του αντίστοιχου τους entity label και δημιούργησα για κάθε ερώτηση αυτών των υποσυνόλων χρησιμοποιώντας, μια λίστα που αποτελείται από 0 και 1, όπου το 1 υπάρχει μόνο στις αντίστοιχες θέσεις της λίστας όπου το entity label ανήκει στην αντίστοιχη ερώτηση αν το entity label και η αντίστοιχη ερώτηση διασπαστούν σε λέξεις και 0 εαν δεν ανήκει. Επίσης προσπάθησα να βρω αυτούς τους δείκτες μέσω ομοιότητας κειμένου είτε με τη μετρική του jaccard similarity είτε με τη μετρική του cosine similarity χρησιμοποιώντας τη βιβλιοθήκη spacy συνδυαζόμενη με το μοντέλο του Sentence Transformer της βιβλιοθήκης των transformer είτε με τη μετρική του cosine similarity χρησιμοποιώντας μόνο τη βιβλιοθήκη spacy. Αυτοί οι entity span δείκτες σε συνδυασμό με το νευρωνικό δίκτυο για τη πρόβλεψη του entity span που δημιούργησα συμβάλλουν ώστε να γίνει η πρόβλεψη the start and the end index of the entity label in the corresponding question for each SimpleQuestions dataset question. Τελικά δημιούργησα ορισμένες εκδόχες της μηχανής απάντησης ερωτήσεων χρησιμοποιώντας τη βιβλιοθήκη spacy, τη βιβλιοθήκη του Sentence Transformer συνδυαζόμενη με τη μετρική του cosine similarity και τη μετρική του jaccard similarity. Τέλος, τα καλύτερα σκόρ αξιολόγησης για το entity span επιτυγχάνονται για τη μεθοδολογία εύρεσης των δεικτών του entity span με τη δημιουργία λιστών που αποτελούνται από 0 και 1 μόνο για τις ερωτήσεις του συνόλου δεδομένων εκπαίδευσης και επικύρωσης SimpleQuestions, των οποίων to entity label αποτελεί μέρος της διατύπωσής τους. Επίσης, το μεγαλύτερο ποσοστό απόλυτα σωστών απαντημένων ερωτήσεων παρατηρείται για τη μηχανή QA με τη βιβλιοθήκη spacy της python για τη μεθοδολογία εύρεσης δεικτών για το entity span(αρχή, τέλος) μέσω της μετρικής του cosine similarity και τη μεθοδολογία εύρεσης δεικτών για το entity span με τη δημιουργία λιστών που αποτελούνται από 0 και 1 στην περίπτωση εκπαίδευσης και επικύρωσης όλων των ερωτήσεων του συνόλου δεδομένων εκπαίδευσης και επικύρωσης SimpleQuestions αντίστοιχα, ανεξάρτητα από το αν το entity label τους αποτελεί μέρος της διατύπωσής τους.

Translated with DeepL.com (free version)

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνητή Νοημοσύνη-Επεξεργασία Φυσικής Γλώσσας

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Wikidata, Μηχανή απάντησης ερωτήσεων, Πρόβλεψη δεικτών αρχής και τέλους του entity label, Πρόβλεψη relation id, Ομοιότητα

Στη σελίδα αυτή αναφέρονται οι αφιερώσεις. Η σελίδα αυτή είναι προαιρετική.

ACKNOWLEDGEMENTS

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέποντα, καθ .Μανώλη Κουμπαράκη, για τη συνεργασία, την ουσιαστική επίβλεψη και την πολύτιμη συμβολή του στην ολοκλήρωση της. Όπως επίσης θα ήθελα να ευχαριστήσω τον μεταπυχιακό φοιτητή του Μεταπτυχιακού Προγράμματος του τμήματος Πληροφορικής και Τηλεπικοινωνιών "Πληροφορική" και συνεργάτη του επιβλέποντα, καθ .Μανώλη Κουμπαράκη, Κωνσταντίνο Πλά για τις καθοριστικές επεμβάσεις και ουσιαστικές συμβουλές για την διεκπεραίωση αυτής της διπλωματικής εργασίας.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

Στον πρόλογο αναφέρονται θέματα που δεν είναι επιστημονικά ή τεχνικά, όπως το πλαίσιο που διενεργήθηκε η εργασία, ευχαριστίες, ο τόπος διεξαγωγής κλπ.

1. INTRODUCTION

1.1 Problem Statement

One of the most important desired aims of “the AI” would be the ability for it to communicate naturally with humans in a human language. As opposed to formal languages that are used to precisely instruct machines to perform certain actions, human language expressions can be difficult to truly understand without an huge body of knowledge about the physical world and human perception of it. While this knowledge about the world can help resolve some ambiguity, or at least make certain interpretations more probable, natural language remains ambiguous, occasionally fooling even humans [1].

Developing systems that hold a certain degree of semantic understanding of human language is one of the most important problems in natural language processing (NLP) as well as information retrieval (IR), but in this thesis it is given more focus on NLP. Understanding human languages is especially interesting for the purpose of question answering, where the goal is to compute a direct answer to a question given by the user. Question answering enables users to go through a simple keyword-based search that returns a ranked list of relevant pages (e.g. Google Search) and transition to a more focused and effective search enables the system to reason over the available knowledge and provide a direct answer. In fact, Google Search already embeds question answering in its search results when the answer is clear [1].

In this thesis, we focus on deep learning for semantic parsing and question answering over knowledge graphs (KGQA), which is a extensive research field. More specifically question answering can be performed using different sources of knowledge: relational databases, graph databases and knowledge graphs, text, images, other multimedia, and any combination of these. These different sources of knowledge can be categorized as structured and unstructured.

With structured sources, for example knowledge graphs and relational databases, we refer to those sources where the meaning of the stored information is unambiguously defined in a formal logical system. Structured sources include . With unstructured sources, such as text and images, we refer to sources where the meaning of the stored information is not clearly indicated and where the data must be interpreted in their raw form in order to compute an answer[1].

Structured sources of knowledge usually provide some automatic way of querying the contained information, for example, relational databases typically support retrieval using SQL queries and RDF (Resource Description Framework) knowledge graphs can be queried using SPARQL [2,3,4,5]. The task of building a query given a question is a form of semantic parsing, for example text-to-SPARQL for question answering over knowledge graphs (KGQA) [1].

Generally the knowledge can be provided in several different ways, such as text, databases, knowledge graphs and even images. When structured data sources are used, such as relational databases or knowledge graphs, question answering is usually performed by relying on a semantic parser that translates the natural language question into a formal query in a compatible query language. This query can then be executed by a database to retrieve the answer to the question in the form of a narrow subset of its data (e.g. a set of entities) [1].

So semantic parsing is the task of translating a natural language expression (e.g question) into a machine-understandable form that maintains the meaning of the original expression [1].

Also, it is worth mentioning that there has been increased interest in neural network based methods from both academic and industrial communities. As with other application areas in NLP and beyond, deep learning automatically learns to extract relevant features, which are used by machine learning methods to describe examples numerically in order to be able to distinguish between them and make decisions. Thus the users could be able to avoid manual feature engineering, which was characteristic for earlier machine learning methods [1].

In general the larger size a dataset has the probabilities a model a neural network to be trained successfully are higher. However the difficulty and the cost of constructing datasets with large quantities of high-quality data is significant. This has as a consequence a lack of data, which can be moderated by relying on transfer learning from other tasks with more data. In transfer learning, (neural network) models are first trained on a different but related task, with the goal of capturing relevant knowledge in the pretrained model. Then, the pretrained model is finetuned on the target task, with the goal of reusing the knowledge captured in the pretraining phase to improve performance on the target task [6].

Recently proposed transfer learning methods [7,8,9,10,11,12] show that significant improvement on downstream natural language processing (NLP) tasks can be obtained by finetuning a neural network that has been trained for language modeling (LM) over a large corpus of text data without task-specific annotations. The use of this family of techniques is an emerging research topic in the NLP community, because these techniques are proved to be very useful [9].

SimpleQuestions dataset [13] is a very well-studied dataset that characterizes core challenges of KGQA. The large size of this dataset is particularly appealing for my study, because it enables me to investigate performance for a wider range of sizes of the data used for training [1,6]. These are the main reasons that i use this dataset for my thesis.

Based on all the aforementioned information, in this thesis the main problem that is stated is the creation of the best question answering (QA) engine over the knowledge graph of Wikidata [14] for the SimpleQuestions dataset [13] by using BERT [1,6,7,15,16,17] as the pretrained model for finetuning, in order to provide users directly answers to their natural language questions, by translating natural language (NL) inputs (e.g questions) to their logical forms (queries) through RDF query language SPARQL [2,3,4,5] (semantic parsing). From this problem occurs that knowledge graph question answering (KGQA) system has to understand the intent of the given question, formulate a query, and retrieve the answer by querying the underlying knowledge base [1,6].

1.2 Project Scope

The subject that this thesis is directly related is deep learning [1,6]. In the recent decades, and most notably in the recent years, incredible progress has been made in the field of deep learning. Compared to traditional machine learning methods (e.g. Support Vector Machines[18]), where features must be manually predefined and are then used in a classifier or regression model, deep learning methods aim to automatically learn to extract useful features. All the above combined with the rapid increase in computing power and availability of data, resulted in the development of novel learning techniques for various applications [1].

One example of these techniques was the development of BERT [1,6,7,15,16,17] and other pre-trained language models in the field of NLP, which enabled transfer learning from more powerful pre-trained models. One of these pre-trained models is the recently released OpenAI GPT3 [19], which was impressive in its ability to generate coherent looking and eloquently written text and even other things (e.g. HTML code).

Semantic parsing and KGQA can similarly benefit from deep learning and it is important to investigate related questions. The ability to learn and pre-train representations rather than engineer features can make it easier to train and adapt semantic parsers for new domains. Also semantic parsing and KGQA have certain characteristics that make them a unique structured prediction problem, compared to the well-studied sequence generation methods used for machine translation. In particular, semantic parsing can often be formulated as a sequence-to-tree (or more generally sequence-to-graph) task, where the tree-structured output can inspire certain modeling choices[1].

1.3 Aim and Objectives

About this thesis the aim is to investigate how deep learning and semantic parsing could be used in order to create a question answering engine over knowledge graphs by using models pretrained for language modeling and also the RDF query language SPARQL [2,3,4,5].

In order to achieve this aim from this thesis the following list of objectives must be achieved:

1. The first objective is the creation of a simple SPARQL [2,3,4,5] query that uses the entity id of each one of the questions of the training and the validation dataset of the SimpleQuestions dataset [13] in order to find its entity label. This simple query is the only involvement of semantic parsing with this thesis.

2. Another objective is to investigate transfer learning for question answer ing over knowledge graphs (KGQA) using models pretrained for language modeling. More specifically we want to see by creating for each of the SimpleQuestions dataset [13] questions a sequence of 0 and 1 (entity span encoded), where 1 if its entity label word is part of the question and 0 if not, and by using BERT transformer model [1,6,7,15,16,17] in order to create neural networks to predict the right relation id and the entity label start and entity label end index in the question correspondingly, if can we achieve high scores in all the metrics, i.e F1, recall, precision and accuracy [20].

3. An important objective is to investigate how text similarity models can effect the performance of the neural networks that predict the relation id and the entity label start and end indexes. In fact we want to achieve the maximum score for all the evaluation metrics(F1, recall, precision and accuracy [20]) for the relation and entity span neural networks, by finding the entity label start and entity label end index using text similarity model of Sentence Transformers [21,22] combined with cosine similarity metric [23,24], the jaccard similarity metric [23,24] and also the spacy library [25,26].

4. Also we want to check for the case of entity span encoded with 0 and 1 (mentioned in the objective 2) through evaluation metrics if the performance of the neural networks for relation ids and entity label start and end indexes,(mentioned in the objective 2),

increases or decreases if we ignore questions that its entity label is not part of them or not. Additionally this is an important objective the corresponding SimpleQuestions subsets [13] that contain only questions that are answerable over wikidata [14].

5. Another objective is to check if the neural networks about relation ids and about entity label start and entity label end index in its corresponding question (mentioned in the objective 2) give predictions that could lead to reliable answers both to other questions that are different from the questions of the test subset of the SimpleQuestions dataset [13] and to the SimpleQuestions dataset questions [13] themselves.

6. Additionally we want to investigate if and how text similarity models and similarity metrics such as cosine and jaccard [23,24] effect the performance of a question answering engine model. More specifically we want to check which of these three models: the QA engine model consisting of python library spacy [25,26], the corresponding model that include the Sentence Transformer similarity model [21,22] combined with cosine similarity metric [23,24] and the QA engine model consisting jaccard similarity metric [23,24] are preferable in order to create a question answering engine that is able to give reliable answers to each question given by the user.

7. Also another objective is to investigate based on the correct predictions for entity id and relation id, when the answers given from QA engines in a question are totally correct, at some point correct or totally incorrect.

1.4 Methodology

In my thesis the methodology that i followed two main tasks are executed: the first is the entity span prediction and the second is the relation prediction. Nevertheless before finding the indices for both the relation id and the start and end word of the entity labels (entity span start and entity span end indexes), i made the following steps:

1. I import some useful python libraries for this thesis through the lines of code below [20,26,27,28,29,30,31,32,33,34]:

```python
import difflib
import operator as op
import random
import numpy as np
import pandas as pd
import torch
import torch.backends.cudnn
from numpy.random import MT19937
from numpy.random import RandomState, SeedSequence
from torch.optim import Adam
from transformers import get_linear_schedule_with_warmup
import os
from google.colab import drive
from sklearn.metrics import f1_score, accuracy_score, precision_score,
    recall_score
import matplotlib.pyplot as plt
import torch.nn as nn
from transformers import BertModel
from transformers import BertTokenizerFast
```

Listing 1.1: Code in Python from which some useful python libraries are imported.

2. I mount google drive through the code below, because all the files (for example .csv) that i used for the implementation of my thesis are saved on my google drive[27,29].

```python
drive.mount('/content/drive')
```

Listing 1.2: Code in Python from which the google drive is mounted.

3. Through the following lines of code in python is checked the device that i used for the implementation of the notebooks of my thesis, that is determined from google colab. Also through the following code this device is printed[27,35]:

```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Device available for running: ")
print(device)
```

Listing 1.3: Code in Python from which the device that is used for this thesis is first checked and then printed.

4. Also i read the created datasets (including either all training and validation questions or only answerable over wikidata [14] questions [13]) as .csv file format by using pandas method read_csv() [36] through the following lines of code [27]:

```python
train_dataset = pd.read_csv("drive/MyDrive/train_dataset.csv", sep = ',')
validation_dataset = pd.read_csv("drive/MyDrive/valid_dataset.csv", sep =
    ',')
```

Listing 1.4: Code in Python where the SimpleQuestions training and validation dataset that include all questions that are either answerable over wikidata or not is read through pandas method read$_csv()wherethecomma("$

```python
train_dataset = pd.read_csv("drive/MyDrive/
    train_answerable_questions_dataset.csv", sep = ',')
validation_dataset = pd.read_csv("drive/MyDrive/
    valid_answerable_questions_dataset.csv", sep = ',')
```

Listing 1.5: Code in Python where the SimpleQuestions training and validation dataset that include only questions that are answerable over wikidata is read through pandas method read$_csv().$

Additionally i made a check in the recently stored train and validation datasets (described in Chapter 4), for any lines, where each line consists of the queries, the entity labels, the relation ids and the entity ids, which have a null value, via the dropna() [27,37] function of the python pandas library. If any such row is found, it is removed from any dataset that has it either training or validation, and the removal is done in the dataset itself, without creating a copy of it [37].

```python
train_dataset.dropna(inplace = True)
validation_dataset.dropna(inplace = True)
```

Listing 1.6: Code in Python where is checked if the SimpleQuestions training and validation dataset has at least one line with null value through pandas function dropna(). These lines of code are the same both for SimpleQuestions dataset that includes only answerable over wikidata questions and for the corresponding dataset that includes all the questions.

5. I create lists through the pandas method .to_list() [27,38] for the entity labels found from SPARQL [2,3,4,5], the entity ids, the relation ids for all the questions of the training and validation dataset of the SimpleQuestions dataset either this dataset includes all questions (answerable, non-answerable over wikidata [13]) or only answerable questions over wikidata [13]. This step is very useful, because this step could make the implementation of the next steps of the methodology of this thesis much more easier.

```python
train_entity_labels = train_dataset['entity_label'].to_list()
validation_entity_labels = validation_dataset['entity_label'].to_list()
train_entityids = train_dataset['entity_id'].to_list()
validation_entityids = validation_dataset['entity_id'].to_list()
train_questions = train_dataset['Question'].to_list()
validation_questions = validation_dataset['Question'].to_list()
train_relationids = train_dataset['relation_id'].to_list()
validation_relationids = validation_dataset['relation_id'].to_list()
```

Listing 1.7: Code in Python from which are created lists for all the questions

6. Another useful step is to find the biggest question length from all the questions of SimpleQuestions dataset [13]. This is achieved by making list concatenation by using ”+” operator [39] through the following lines of code [27]:

```python
#Below the max questions length is found for all questions.
quests = train_questions + validation_questions
print(len(quests), len(train_questions), len(validation_questions))
# Printing concatenated list
q_lengths = []
for s in range(len(quests)): #padding
    q_lengths.append(len(quests[s].split()))
max_question_length = max(q_lengths)
print(max_question_length)#Max question length
```

Listing 1.8: Code in Python where is found the biggest question length from all the questions of SimpleQuestions dataset by using list concatenation through ”+” operator.

Also when i refer the words ”full” or ”whole” for the SimpleQuestions dataset [13] i mean that this dataset contains answerable and non-answerable questions [13] over wikidata [14].

More analytically:

Relation Prediction Firstly i create the relation vocabulary and i found the index of each unique relation id of the relation vocabulary in this vocabulary through the functions relation_vocabulary and find_relations_index [26,27] respectively:

```python
#Create relation vocabulary for each question of train and validation dataset
def relation_vocabulary(relation_column_dataset):
 relation_vocabulary = []
 #In order to have the unique relation ids it is important to check if an
   relation id is already in the relation_vocabulary
 for relation in range(len(relation_column_dataset)):
  g = 0
  for i in range(len(relation_vocabulary)):
    if relation_column_dataset[relation] != relation_vocabulary[i]:
      g += 1
    else:
      continue
  if g == len(relation_vocabulary) and relation_column_dataset[relation] not
   in relation_vocabulary:
      relation_vocabulary.append(relation_column_dataset[relation]) #The
   relation id is not inside the relation vocabulary
      #which has some relations already or the relation vocaulary has no
   relations
  return relation_vocabulary
```

Listing 1.9: Code in python in order to create a function by which the relation vocabulary is created. This vocabulary includes the unique relation ids. The same function creates this vocabulary both for the dataset with only answerable over wikidata questions or the dataset including answerable and non-answerable over wikidata questions.

```python
def find_relation_index(relation_vocabulary,relation_ids_column):
 relations_count = []
 for relation in range(len(relation_ids_column)):
   relations_count.append(relation_vocabulary.index(relation_ids_
   column[relation]))

  return relations_count
```

Listing 1.10: Code in python in order to create a function the index in relation vocabulary for each unique id in this relation vocabulary is found.

The function relation_vocabulary [26,27], which takes as an argument the list of all relation ids of the training and validation dataset, creates and returns the relation vocabulary which is the list of unique relation ids present in the training and validation dataset of SimpleQuestions dataset [13].

Briefly through relation_vocabulary function[26,27], the relation vocabulary is constructed by creating a list, by checking each time if a relation id is already in this list. If it is not already in the list (relation vocabulary), then is added as a new unique relation id in the relation vocabulary, else the relation id is already in the list (relation vocabulary) as a unique relation id.

Respectively through the function find_relation_index [26,27], which takes as an argument the relation vocabulary, returns for each relation id its position (index) in the relation vocabulary through the index() function [40].

The length of the relation vocabulary in the case of training and validating the full SimpleQuestions dataset [13] is equal to 129 and the length of the relation vocabulary for the case of training and validating the SimpleQuestions subset [13] that includes only questions that are answerable over wikidata[14] is equal to 125.

Finally i create a neural network [26,27,41] through a class in python in order to predict the relation index and thus to predict the relation id. This neural network includes the same layers as the entity span neural network, i.e a bert layer (bert-base-uncased model) [15,17,32,33], a linear layer [42] with 768 neurons for input and 129 neurons for output ,one layer consisting of an activation function [43], by a dropout layer[44], and softmax activation function layer[45,46,47,48], to pass the result obtained from the previous four layers and convert it into probability.

```python
class BertRelationsClassifier(nn.Module):
  def __init__(self, activation, dropout_prob, rel_vocab_len):
    super(BertRelationsClassifier, self).__init__()
    activations = {
        "ELU" : nn.ELU,
        "LeakyReLU"    : nn.LeakyReLU,
        "Softplus"     : nn.Softplus,
        "Tanhshrink"     : nn.Tanhshrink
  }
    self.bert_model = BertModel.from_pretrained('bert-base-uncased')
    self.rel_vocab_len = rel_vocab_len
    self.fc = nn.Linear(768, self.rel_vocab_len)
    self.activation_type = activation
    self.dropout_probability = dropout_prob
    self.activation = activations[self.activation_type]()
    self.dropout = nn.Dropout(self.dropout_probability)
    self.softmax = nn.Softmax(dim=1)

 def forward(self, input_ids, attention_mask):
   out = self.bert_model(input_ids, attention_mask)
```

```python
    out = self.fc(out[1])#CLS token-pooled output
    out = self.activation(out)
    out = self.dropout(out)
    out = self.softmax(out)
    return out
```

Listing 1.11: Code in Python for the neural network for relation prediction which is a python class named BertRelationsSpanClassifier.

The output of the linear layer, which is equal to 129, is the length of the corresponding relation vocabulary created to group the unique relations and to find the relation index for each relation of the respective question. Regarding the dataset used for this work [13], initially all those .txt files, which either contains all questions or only the questions [13] that could be answered over wikidata [14], were used which contain the questions with entity ids and relation ids that have been obtained by annotating (let us say transforming) the questions from the Knowledge Base Freebase to the Knowledge Base Wikidata [14] (e.g. annotated_wd_data_train.txt, annotated_wd_data_valid.txt, annotated_wd_data_train_answearable.txt, annotated_wd_data_valid_answearable.txt [13]).

Entity Span Prediction

The prediction of the entity start index and entity end index is concerned, could be done following the steps below :

- Initially each question, as well as the corresponding entity label, is splitted into words using python's split() method[49]. Then, using python method join()[50], parts of the query are created equal in length to that of their splitted entity label in words. These parts are stored in a list as well as the indexes of the initial word and final word of these question parts are stored in dictionaries with keys the question parts and values the indexes(positions) of the initial words and final words for each question. Everything mentioned above is done via the create_question_parts function [26,27], shown below, which takes as arguments, the splitted question and the splitted entity label and returns the list of question parts and dictionaries containing the indexes(positions) of the initial words and final words for each question part .

```python
def create_question_parts(question_splitted,entity_splited_length):
 Quests_Parts = []
 Quest_Parts_Start = {}
 Quest_Parts_End = {}
 for i in range(len(question_splitted)-entity_splited_length+1):
  quest_part = " ".join(question_splitted[i:i+entity_splited_length])
  Quests_Parts.append(quest_part)
  Quest_Parts_Start[quest_part] = i
  Quest_Parts_End[quest_part] = i+entity_splited_length-1
 return Quests_Parts, Quest_Parts_Start, Quest_Parts_End
```

Listing 1.12: Code for python function named create_question_parts from which the parts with legth equal to the splitted entity label length for each question are created.

- After this first step it is more easier for me to find the entity start index and entity end indexes. About this indexes i can find them for each question of the training and the validation dataset of SimpleQuestions through the following ways:

  1. The python spacy library and specifically the difflib.get_close_matches() [25,26] function could be used to find out for each entity label which parts of its corresponding question are the same or very close to it either lexically (most times) or semantically. If such words are found that are identical or very close to the

entity label, its position as entity start index and the position of the last word as entity end index are finally obtained from them.

If no such question part is found, then via python's Sentence-transformers[13,14] the corresponding embeddings are created for both the entity label and each part of its question with a length equal to the length of the splitted entity label as mentioned above, starting from the first, second word of the question until the end of the question is reached.

More specifically, the pretrained model all-MiniLM-L6-v2 is used for the Sentence Transformer, which combines speed (5 times faster than all other pretrained models of the Sentence Transformer and (sentence) transformers in general) and good quality in terms of results [22].

The similarity model for Sentence Transformer is loaded via the following commands [21]:

```python
!pip install sentence-transformers
from sentence_transformers import SentenceTransformer, util
similarity_model = SentenceTransformer("all-MiniLM-L6-v2")
```

Listing 1.13: Code in python in order to install Sentence Transformers library.

These embeddings are compared through the metric of cosine similarity [23,24] by creating a cosine similarity function [23,24,27], which is shown below that takes as an argument the two encoded vectors, one for the entity label and one for a part of its corresponding question, and returns the cosine similarity score between the two vectors.

```python
def cosine_similarity(encoding1, encoding2):
  return np.dot(encoding1, encoding2) / (np.linalg.norm(encoding1) *
    np.linalg.norm(encoding2)) if np.linalg.norm(encoding1) * np.
   linalg.norm(encoding2) != 0 else 0
```

Listing 1.14: Code for python function named cosine_similarity from which the cosine similarity metric score is calcuated.

This function give as a result the part of the question that has the highest possible similarity to the entity label. Therefore, it becomes very easy through the corresponding part of the question to define the index (position) of the first word of the question part as entity start index and the index (position) of the last word in this part of the question as entity end index.

The function through which the embeddings are created via Sentence Transformer [21,22] and uses the cosine similarity function which i mentioned earlier [23,24], fast_best_match_with_cosine_similarity [26,27] which takes as arguments the entity label, the list of question parts, the dictionaries containing the indexes(positions) of the start words and end words for each question segment mentioned above, as well as the similarity model of Sentence Transformer and returns the question part that has the highest similarity and the position of the initial and final word of that question part, otherwise if there is no such question part it returns None as the question part and as the index and for the initial and final word of the entity label the value -1.

```python
# Finds best match based on their similarity
def best_match_with_cosine_similarity(entity, question_parts,
    Start_Indexes, End_Indexes, similarity_model):
          similarity_scores = {}
          entity_vector = similarity_model.encode(entity)
          for q in range(len(question_parts)):
```

```
# Find similarity scores between entity and each question part in question
            quest = similarity_model.encode(question_parts[q])
            similarity_scores[question_parts[q]] = cosine_similarity(entity_vector, quest)
# Return the highest similarity score from all question parts as a list
           if similarity_scores == {}:
            return None, -1, -1
           Q = dict(sorted(similarity_scores.items(), key = lambda x: x[1], reverse = True))
           EntitiesList = [key for key in Q]
           return EntitiesList[0], Start_Indexes[EntitiesList[0]], End_Indexes[EntitiesList[0]]
```

Listing 1.15: Code in python in order to create the best_match_with_cosine_similarity function where by using jaccard similarity metric the question part with the highest similarity score with the correspondning entity label is found.

Finally, for the questions where no question part identical or similar to the corresponding entity label has been found through all the above methodology, which have been measured through the entity_span function [26,27] and are 17 for the train dataset of the SimpleQuestions subset that include all questions [13] (answerable,non-answerable over wikidata [14]), 10 for the train dataset of the SimpleQuestions subset that include only answerable over wikidata [14] questions [13] and 2 for the validation dataset both for the SimpleQuestions dataset including all questions and the corresponding dataset including only answerable over wikidata questions [14], are set as entity start index and entity end index the positions (indices) of the penultimate (question length-2) and the last word (question length-1) are set respectively. In this way the lengths of the SimpleQuestions training and validation dataset [13] are kept constant.

The entity_span function where the entity span start and entity span end indexes are found for each question, takes as arguments the list of all questions of SimpleQuestions training or validation dataset, the list of all entity labels of SimpleQuestions training or validation dataset in question and the similarity model of Sentence Transformer.

This function calls the functions create_question_parts and best_match_with _cosine_similarity in order to create the parts of each question and find the question part that is equal to the entity label of the question with its start word index and end word index and returns the list of indexes (positions) of the initial words of the entity labels within the query (entity span start indexes) and the list of indexes (positions) of the final words of the entity labels within the query (entity span end indexes) [26,27].

```
def entity_span(question_column, entity_label_column, similarity_model):
  entity_start_column = []
  entity_end_column = []
  #Count for how many questions there is no question part that is equal or similar(finding with difflib.get_close_matches() or SentenceTransformer)
  #to the corresponding entity label.
  count_entity_with_no_similarity = 0
  for q in range(len(question_column)):
    # Find the word that performs the biggest similarity
      quest = question_column[q].lower().split()
      entity_label = entity_label_column[q].lower()
```

```python
    ent_label = entity_label.split()
    entity_label_splitted_length = len(ent_label)
    Quests_Parts, Quest_Parts_Start, Quest_Parts_End = create_question_parts(quest, entity_label_splitted_length)
    Entity = difflib.get_close_matches(entity_label, Quests_Parts, 1)
    if (Entity != []):
      # Beginning and end of span
      Entity_start_index = Quest_Parts_Start[Entity[0]]
      Entity_end_index = Quest_Parts_End[Entity[0]]
      entity_start_column.append(Entity_start_index)
      entity_end_column.append(Entity_end_index)
    else:
      Entity, Entity_start_index, Entity_end_index = best_match_with_cosine_similarity(entity_label, Quests_Parts, Quest_Parts_Start, Quest_Parts_End, similarity_model)
      if Entity is None:
      # Beginning and end of span
       entity_start_column.append(len(quest)-2)
       entity_end_column.append(len(quest)-1)
       count_entity_with_no_similarity += 1
      else:
       print(q, Entity, Entity_start_index, Entity_end_index)
       entity_start_column.append(Entity_start_index)
       entity_end_column.append(Entity_end_index)
print(count_entity_with_no_similarity)
return entity_start_column, entity_end_column
train_entity_span_start, train_entity_span_end = entity_span(train_questions, train_entity_labels, similarity_model)
validation_entity_span_start, validation_entity_span_end = entity_span(validation_questions, validation_entity_labels, similarity_model)
```

Listing 1.16: Code in python in order to create the entity_span function where the entity span start and entity span end indexes are found through the entity span of the corresponding question by using spacy library or the combination of Sentence Transformers library with the cosine similarity metric. Also after the function are the lines of code which are the execution of the function the entity_span for the corresponding SimpleQuestions training and validation dataset.

2. The second way to find the entity span start and entity span end indexes is the same with the previously described methodology, with the only differences that instead of cosine similarity metric, i use jaccard similarity metric [23,24] in order to find the similarity score for a question between its entity label and a question part. The basic difference is that in order to calcuate the jaccard similarity metric, i do not use the Sentence Transformer [21,22] library, but only the list of words for entity label and each question part respectively. In this function i create this list for each question part through python's split() function [48], while the corresponding list for entity label is created through entity span function, which is described earlier. Also the lengths of the SimpleQuestions training and validation dataset [13] are kept constant. Essentially the main changes are that instead of the functions cosine_similarity and best_match_with_cosine_similarity, i create the functions jaccard_similarity [23,24,27] and best_match_with_jaccard_similarity [26,27] respectively, where the entity_span function has changed accordingly [26,27]. (encoding1, encoding2 are variables for the lists of the words for the splitted entity label and the splitted question part in words for whichever question.)

```python
def jaccard_similarity(encoding1, encoding2):
```

```python
""" returns the jaccard similarity between two encodings """
intersection_cardinality = len(set.intersection(*
[set(encoding1), set(encoding2)]))
union_cardinality = len(set.union(*[set(encoding1),
set(encoding2)]))
return intersection_cardinality/float(union_cardinality)
if union_cardinality != 0 else 0
```

Listing 1.17: Code for Python function named jaccard$_{similarity from which the jaccard similarity metric score is calcuated.}$

```python
# Finds best match based on their similarity
def best_match_with_jaccard_similarity(entity, question_parts,
    Start_Indexes, End_Indexes):
           similarity_scores = {}
           for q in range(len(question_parts)):
    # Find similarity scores between entity and each question part
    in question
            similarity_scores[question_parts[q]] =
    jaccard_similarity(entity, question_parts[q].split())
    # Return the highest similarity score from all question parts as
     a list
           if similarity_scores == {}:
            return None, -1, -1
           Q = dict(sorted(similarity_scores.items(), key = lambda x
    : x[1], reverse = True))
           EntitiesList = [key for key in Q]
           return EntitiesList[0], Start_Indexes[EntitiesList[0]],
    End_Indexes[EntitiesList[0]]
```

Listing 1.18: Code in python in order to create the best_match_with_jaccard_similarity function where by using jaccard similarity metric the question part with the highest similarity score with the correspondning entity label is found.

```python
def entity_span(question_column, entity_label_column):
  entity_start_column = []
  entity_end_column = []
  #Count for how many questions there is no question part that is
    equal or similar(finding with difflib.get_close_matches() or
    jaccard similarity metric)
  #to the corresponding entity label.
  count_entity_with_no_similarity = 0
  for q in range(len(question_column)):
    # Find the word that performs the biggest similarity
      quest = question_column[q].lower().split()
      entity_label = entity_label_column[q].lower()
      ent_label = entity_label.split()
      entity_label_splited_length = len(ent_label)
      Quests_Parts, Quest_Parts_Start, Quest_Parts_End =
    create_question_parts(quest, entity_label_splited_length)
      Entity = difflib.get_close_matches(entity_label, Quests_Parts,
     1)
      if (Entity != []):
        # Beginning and end of span
        Entity_start_index = Quest_Parts_Start[Entity[0]]
        Entity_end_index = Quest_Parts_End[Entity[0]]
        entity_start_column.append(Entity_start_index)
        entity_end_column.append(Entity_end_index)
      else:
        Entity, Entity_start_index, Entity_end_index =
    best_match_with_jaccard_similarity(ent_label, Quests_Parts,
    Quest_Parts_Start, Quest_Parts_End)
```

```
        if Entity is None:
        # Beginning and end of span
         entity_start_column.append(len(quest)-2)
         entity_end_column.append(len(quest)-1)
         count_entity_with_no_similarity += 1
        else:
         print(q, Entity, Entity_start_index, Entity_end_index)
         entity_start_column.append(Entity_start_index)
         entity_end_column.append(Entity_end_index)
  print(count_entity_with_no_similarity)
  return entity_start_column, entity_end_column
  train_entity_span_start, train_entity_span_end = entity_span(train_questions, train_entity_labels, similarity_model)
  validation_entity_span_start, validation_entity_span_end = entity_span(validation_questions, validation_entity_labels, similarity_model)
```

Listing 1.19: Code in python in order to create the entity_span function where the entity span start and entity span end indexes are found through the entity span of the corresponding question by using spacy library or the jaccard similarity metric. Also after the function are the lines of code which are the execution of the function the entity_span for the corresponding SimpleQuestions training and validation dataset.

3. An alternative way of finding the entity span start index and entity span end index is to create for each question an entity span of length equal to the length of the question, which will consist of 1 if the corresponding entity label word is present in the words of the question and 0 if it is not [1,6]. Essentially this approach focuses on the existence of the entity label itself exclusively within its corresponding question, rather than searching for some similar part of the question to the entity label.

   So, i use the function create_question_parts [26,27] mentioned in the first approach, the list with all parts of each question, the dictionaries with keys the parts for each question and values the indexes (positions) of the initial words and final words for each question.

   Then through the create_entity_span [26,27] function that takes as arguments the question splitted in words by the python split() function [48] and the indexes of the initial and final word of the part of the question that is equal to the corresponding entity label, the entity span for each question is created. In this entity span because the words that make up the entity label must be in consecutive positions within the question, it is always necessary that when creating the entity span, the index specifying the position of the starting word of the equal question part with the corresponding entity label for the question must be less than or equal to the corresponding index of the ending word of that equal question part. This indirectly ensures that the ones that enter the entity span are consecutive and equal in number to the length of the entity label splitted in words by the split() function [48]. This function returns the entity span, as well as the number of consecutive ones in it for each question.

```
#Create entity span
def create_entity_span(question_splitted_length, quest_part_start, quest_part_end):
   entity_span = []
   count_successive_ones = 0
   for k in range(question_splitted_length):
     if k >= quest_part_start and k <= quest_part_end:
        entity_span.append(1)
        count_successive_ones += 1
```

```python
    else:
        entity_span.append(0)

  return entity_span, count_successive_ones
```

Listing 1.20: Code in python in order to create the create_entity_span function where the entity span for each question is created for all questions regardless if their entity label is part of their wording or not. The entity span that is created through this function will consists of zeros and ones which are consecutive equal in number with the length of the entity label when it is splitted in words.

Through the find_start_and_end_index function [26,27], which takes as arguments the entity span, the entity label, the list of all the parts of the question, the dictionaries with keys the parts for each question and values the indexes (positions) of the start words and end words for each question, the length of the question and the number of consecutive ones of the entity span which are counted in this function, the question part that is equal with the entity label, the position of the initial word (index of the first one in the entity span) of the entity label and respectively the position of the final word (index of the last one in the entity span) could be found. If the entity span that is created for a question is not full of zeros, then a logical variable is used to confirm that all ones present are contiguous.

Thus, the index of the beginning of the entity label is considered to be the index of the first ones that existing in the entity span, through the index()[40] function, and the index of the end of the entity label is considered to be the value of the index of the first one of the entity span to which the number of consecutive ones of the entity span is added and the value 1 is subtracted. Essentially through this methodology the entity start index and the entity end index must be exactly the same with the indexes of the start word and the end word of the question part that is equal to the entity label, from the corresponding dictionaries including start and end indexes of question parts of a question which are returned from the function create_question_parts [26,27], respectively. In the case, of course, that no word of the entity label is present in the words of the corresponding question, the entity span will consist only of zeros, so there is no word to derive the corresponding indices for entity start and entity end.

To deal with this situation, initially from the list of question parts as obtained from the create_question_parts function[26,27] through difflib.get_close_matches() [25,26], it is searched, which part is closest to the entity label and for that question part from the dictionaries containing the indexes(positions) of the initial words and the final words for it obtained from the same function, the corresponding positional indexes of its initial and final words are found. Next, if no question part that is found by difflib.get_close_matches() [25,26] is even minimally similar to the entity label, then the positions (indices) of the penultimate (question length-2) and the last word (question length-1) are set as the entity start index and entity end index, respectively. In this way the lengths of the train and validation dataset are kept constant. The logical variable that confirms the existence of consecutive ones in the span is returned by this function, as well as the positional indices of the initial and final word of the entity label.

```python
#Create for each question the entity span and find the entity span
    start index and the entity span end index.
def find_start_and_end_index(entity_label,
    entity_label_splitted_length, question_parts,
    question_splitted_length, start_indexes, end_indexes):
```

```
  entity_span_start_index = -1
  entity_span_end_index = -1
  quest_part_start_index = -1
  quest_part_end_index = -1
  entity_span = []
  entity_span_with_successive_ones = 0
  for q_p in range(len(question_parts)):
    if question_parts[q_p] == entity_label:
      quest_part = question_parts[q_p]
      quest_part_start_index = start_indexes[quest_part]
      quest_part_end_index = end_indexes[quest_part]
  entity_span, count_successive_ones = create_entity_span(
   question_splitted_length, quest_part_start_index,
   quest_part_end_index)
  if entity_span != [0]*question_splitted_length:
   entity_span_with_successive_ones = 1
   entity_span_start_index = entity_span.index(1)
   entity_span_end_index = entity_span_start_index +
   count_successive_ones-1
  else:
   Entity = difflib.get_close_matches(entity_label, question_parts,
   1)
   if Entity != []:
    entity_span_start_index = start_indexes[Entity[0]]
    entity_span_end_index = end_indexes[Entity[0]]
   else:
    entity_span_start_index = question_splitted_length-2
    entity_span_end_index = question_splitted_length-1

  return entity_span, entity_span_with_successive_ones,
   entity_span_start_index, entity_span_end_index
```

Listing 1.21: Code in python in order to create the find_start_and_end_index function where the entity span start and entity span end index for each question are found. This function finds entity span indexes for all questions regardless if their entity label is part of their wording or not.

All of the above is implemented via the entity_span_encoded function [26,27], which takes as arguments the list of all questions of the given SimpleQuestions dataset (training or validation) [13](either the dataset with only answerable over wikidata [14] questions [13] or the dataset including answerable and non-answerable over wikidata [14] questions [13]), the list of all entity labels of the SimpleQuestions dataset (training or validation) and returns the list of entity spans, as well as the lists of start and end position indices for each entity label for all questions. It is worth noting that this function counts the number of questions whose entity span consists of 0 and 1, where the ones are contiguous.

```
#Create entity span by create an list which consists of zeros and
   ones. By checking in which part for each question the
   correspoding entity
#exists and in these indexes of the span to replace with ones.
#This is applied for each one of the train and validation questions
def entity_span_encoded(question_column_dataset,
   entity_label_column_dataset):
  entity_spans = []
  span_start_index = []
  span_end_index = []
  count_spans_with_successive_ones = 0
  for q in range(len(question_column_dataset)):
      quest = question_column_dataset[q].lower().split()
      quest_splitted_length  = len(quest)
      ent_label = entity_label_column_dataset[q].lower()
      s = ent_label.split()
      entity_label_splitted_length = len(s)
      Quests_Parts, Quest_Parts_Start, Quest_Parts_End = create_question_parts(quest, entity_label_splitted_length)
      entity_span, span_with_successive_ones, entity_span_start_index, entity_span_end_index = find_start_and_end_index(ent_label, entity_label_splitted_length, Quests_Parts, quest_splitted_length, Quest_Parts_Start, Quest_Parts_End)
      entity_spans.append(entity_span)
      span_start_index.append(entity_span_start_index)
      span_end_index.append(entity_span_end_index)
      if span_with_successive_ones == 1:#Boolean variable
       count_spans_with_successive_ones += 1
  return entity_spans, span_start_index, span_end_index
  train_span, train_entity_span_encoded_start_indexes, train_entity_span_encoded_end_indexes = entity_span_encoded(train_questions, train_entity_labels)
  validation_span, validation_entity_span_encoded_start_indexes, validation_entity_span_encoded_end_indexes = entity_span_encoded(validation_questions, validation_entity_labels)
```

Listing 1.22: Code in python in order to create the entity_span_encoded function where the entity span and entity span indexes (start and end) for each question are found. This function finds entity span indexes for all questions regardless if their entity label is part of their wording or not.Also after the function are the lines of code which are the execution of the function the entity_span_encoded for the corresponding SimpleQuestions training and validation dataset.

4. The last methodology that i created in order to find entity span start index and entity span end index is to create for each Simple Questions training and validation dataset question is the same as the previous methodology, with the only difference that i use for training and validation, only the questions that one of its parts is equal to their corresponding entity label. Thus the lengths of the training and of the validation dataset are not kept constant.

   So the first change compared to the previous methodology is that in the function find_start_and_end_index [26,27], there is no need to find the entity span indexes for the questions with no equal part with the entity label either through the spacy library [25,26] or by putting some specific values as the values of indexes. Another change is that the boolean value that the find_start_and_end_index function returns is useful for the entity_span_encoded function not only to count how many questions have their entity label equal to one of their parts, but also to insert into lists all these questions, their corresponding entity labels, entity spans, entity span start indexes, entity span end indexes and relation ids. The corresponding functions find_start_and_end_index and entity_span_encoded [26,27] are shown below:

```
#Create for each question the entity span and find the entity span start index and the entity span end index.
def find_start_and_end_index(entity_label, entity_label_splitted_length, question_parts, question_splitted_length, start_indexes, end_indexes):
 entity_span_start_index = -1
 entity_span_end_index = -1
 quest_part_start_index = -1
 quest_part_end_index = -1
 entity_span = []
 entity_span_with_successive_ones = 0
```

```
  for q_p in range(len(question_parts)):
    if question_parts[q_p] == entity_label:
      quest_part = question_parts[q_p]
      quest_part_start_index = start_indexes[quest_part]
      quest_part_end_index = end_indexes[quest_part]
  entity_span, count_successive_ones = create_entity_span(
   question_splitted_length, quest_part_start_index,
   quest_part_end_index)
  if entity_span != [0]*question_splitted_length:
   entity_span_with_successive_ones = 1
   entity_span_start_index = entity_span.index(1)
   entity_span_end_index = entity_span_start_index +
   count_successive_ones-1
  return entity_span, entity_span_with_successive_ones,
   entity_span_start_index, entity_span_end_index
```

Listing 1.23: Code in python in order to create the find_start_and_end_index function where the entity span start and entity span for each question are found only for questions that their entity label is part of their wording or not.

```
#Create entity span by create an list which consists of zeros and
   ones. By checking in which part for each question the
   correspoding entity
#exists and in these indexes of the span to replace with ones.
#This is applied for each one of the train and validation questions
def entity_span_encoded(question_column_dataset,
   entity_label_column_dataset, relation_indexes):
  span_start_index = []
  span_end_index = []
  relations_index_list = []
  entity_spans = []
  entity_labels_list = []
  questions_list = []
  count_spans_with_successive_ones = 0
  for q in range(len(question_column_dataset)):
      quest = question_column_dataset[q].lower().split()
      quest_splitted_length  = len(quest)
      ent_label = entity_label_column_dataset[q].lower()
      s = ent_label.split()
      entity_label_splitted_length = len(s)
      Quests_Parts, Quest_Parts_Start, Quest_Parts_End =
   create_question_parts(quest, entity_label_splitted_length)
      entity_span, span_with_successive_ones,
   entity_span_start_index, entity_span_end_index =
   find_start_and_end_index(ent_label, entity_label_splitted_length,
    Quests_Parts, quest_splitted_length, Quest_Parts_Start,
   Quest_Parts_End)
      if span_with_successive_ones == 1:#Boolean variable
        span_start_index.append(entity_span_start_index)
        span_end_index.append(entity_span_end_index)
        relations_index_list.append(relation_indexes[q])
        entity_spans.append(entity_span)
        questions_list.append(question_column_dataset[q])
        entity_labels_list.append(entity_label_column_dataset[q])
        count_spans_with_successive_ones += 1
  print(count_spans_with_successive_ones)
  return entity_spans, span_start_index, span_end_index,
   relations_index_list, questions_list, entity_labels_list
  train_span, train_entity_span_encoded_start_indexes,
   train_entity_span_encoded_end_indexes,
   train_entity_span_encoded_relations_list,
    train_entity_span_encoded_questions_list,
    train_span_encoded_entity_labels_list = entity_span_encoded(
    train_questions, train_entity_labels, train_relations_index)
  validation_span, validation_entity_span_encoded_start_indexes,
    validation_entity_span_encoded_end_indexes,
    validation_entity_span_encoded_relations_list,
    validation_entity_span_encoded_questions_list,
    validation_span_encoded_entity_labels_list = entity_span_encoded(
    validation_questions, validation_entity_labels,
    validation_relations_index)
```

Listing 1.24: Code in Python in order to create the entity_span_encoded function where the entity span and entity span indexes (start and end) are found only for questions that their entity label is part of their wording or not. Also after the function are the lines of code which are the execution of the function the entity_span for the corresponding SimpleQuestions training and validation dataset.

It is worth mentioning that all the words in entity labels and the questions have been converted to words with lowercase letters, regardless of whether they are capitalized or not, in order to have better comparison between question parts and entity labels.

- Finally i create the neural network in order to predict the entity start index and entity end index [26,27,41], through a class in python which includes two heads one entity start head and one entity end head. Each of these heads consists of a bert layer (bert-base-uncased model) [15,17,32,33], a linear layer [42] with 768 neurons for input and 33 neurons for output, which is the maximum question length that is observed in all the SimpleQuestions dataset, one layer consisting of an activation function [43], by a dropout layer[44], where at the end the result obtained from the above four layers is passed through the softmax activation function [45,46,47,48], to convert the results into probabilities.

```
class BertEntitySpanClassifier(nn.Module):
  def __init__(self, activation, dropout_prob, max_quest_len):
    super(BertEntitySpanClassifier, self).__init__()
    self.bert_model = BertModel.from_pretrained('bert-base-uncased')

    activations = {
        "ELU" : nn.ELU,
        "LeakyReLU"    : nn.LeakyReLU,
        "Softplus"     : nn.Softplus,
        "Tanhshrink"     : nn.Tanhshrink
  }
    self.activation_type = activation
    self.activation = activations[self.activation_type]()
    self.max_length = max_quest_len
    self.dropout_probability = dropout_prob
    self.start_head = nn.Sequential(
          nn.Linear(768, self.max_length),
          self.activation,
          nn.Dropout(p=self.dropout_probability),
          nn.Flatten(),
          nn.Softmax(dim=1)
    )

    self.end_head = nn.Sequential(
          nn.Linear(768, self.max_length),
          self.activation,
          nn.Dropout(p=self.dropout_probability),
          nn.Flatten(),
          nn.Softmax(dim=1)
    )

  def forward(self, input_ids, att_mask):
     out = self.bert_model(input_ids, att_mask)
     out = out[0]
     out_start = self.start_head(out)
     out_end = self.end_head(out)
     return out_start , out_end
```

Listing 1.25: Code in python for the neural network for entity span prediction which is a python class named BertEntitySpanClassifier.

After describing the above methodologies which for finding entity span start, entity span end and relation indexes, there are some final steps that i made in order to be able to predict all these indexes (relation, entity span start, entity span end). More specifically these steps are:

1. Loading of the tokenizer BertTokenizerFast [17,34]:

```
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

Listing 1.26: Code in python that loads the tokenizer of Transformers Bert model ”bert-base-uncased”.

2. Creation of a custom dataset [51] for the SimpleQuestions dataset (either for the dataset with only answerable over wikidata [14] questions [13] or for the corresponding dataset including answerable and non-answerable over wikidata [14] questions [13])through a python class, where its questions are encoded through BertTokenizerFast method encode_plus() [34,52]. This custom dataset includes the entity span indexes(start, end), the relation indexes, the questions, the ids and the attention mask occurred from encoding of questions.

   Also since the span start and end correspond to actual words, but here I use the fast version of BERT tokenizer BertTokenizerFast [34] i need to locate the corresponding start and end tokens in the BERT tokenized question for each entity [26,27]. This dataset includes a pytorch generator in cpu device[53].

   Also in order to create this custom dataset i used the lists of questions, relation, entity span start, entity span end indexes, the maximum question length of all SimpleQuestions training and validation dataset questions [13]. Of course depending of which methodology and which SimpleQuestions dataset i use(either subset including only the answerable over wikidata [14] questions [13] or the full dataset), i use the corresponding lists.

   The input ids and the attention masks that occurred from the question encoding through BertTokenizerFast[34] are the inputs of the bert layer [6,7,15,16,17,32,33] the relation prediction neural network [26,27] that i have created earlier (**Relation Prediction**) and of the bert layer of the corresponding entity span prediction network [26,27] which is described later.

```
g = torch.Generator()
print(g.device)
```

Listing 1.27: Generator from python library pytorch

```python
class BertQuestionAnsweringDataset:
  def __init__(self, questions, tokenizer, entity_starts, entity_ends,
   relation_indexes, length, generator, shuffle=True):
        self.questions = questions
        self.max_length = length
        self.tokenizer = tokenizer
        self.relation_indexes = relation_indexes
        self.g = generator
        self.shuffle = shuffle
        ids = []
        masks = []
        start_ent = []
        end_ent = []
        # Encode questions
        for question in range(len(self.questions)):
          encoding = self.tokenizer.encode_plus(
             text = self.questions[question],
             max_length=self.max_length,
             truncation=True,
             pad_to_max_length=True,
             return_attention_mask=True,
             padding='max_length',
             add_special_tokens=False
          )
          Question = self.questions[question].split()
          # Locate the corresponding start in the BERT tokenized question
          start_token = tokenizer.encode_plus(
            text = Question[entity_starts[question]],
            return_attention_mask=True,
            add_special_tokens=False
        )
          # Locate the corresponding end in the BERT tokenized question
          end_token = tokenizer.encode_plus(
            text = Question[entity_ends[question]],
            return_attention_mask=True,
            add_special_tokens=False
        )
          ids.append(encoding['input_ids'])
          masks.append(encoding['attention_mask'])
          start_ent.append(encoding['input_ids'].index(start_token['
   input_ids'][0]))
          end_ent.append(encoding['input_ids'].index(end_token['input_ids
   '][-1]))
        self.input_ids = ids
        self.attention_masks = masks
        self.entity_start_indexes = start_ent
        self.entity_end_indexes = end_ent

  def __len__(self):
        return len(self.questions)

  def __getitem__(self, idx):
        return { 'input_ids': torch.tensor(self.input_ids[idx]),
                'attention_mask':  torch.tensor(self.attention_masks[idx
   ]),
                'relation_indexes': torch.tensor(self.relation_indexes[
   idx]),
                'entity_start': torch.tensor(self.entity_start_indexes[
   idx], dtype = torch.long),
                'entity_end': torch.tensor(self.entity_end_indexes[idx],
    dtype = torch.long)
                }

  def set_epoch(self, epoch):
      self.epoch = epoch
      if self.shuffle:
          self.g.manual_seed(123456789 + self.epoch)
```

Listing 1.28: Generator from python library pytorch

Also the above class includes a boolean variable shuffle, with default value equal to True and an function set_epoch() in python with arguments the epoch of the training and validation process and the instance of the custom dataset class, i.e self, which helps to the achievement of reproducible results during the training and validation process [54].

Neural network projects are full of non-deterministic processes that result in different outcomes in each execution. To perform a fair comparison, i have to make all these outcomes from different execution to be equal with each other. In other words i need to enable reproducibility and thus to have reproducible results [54].

So this function helps the reproducibility to be kept by shuffling differently but reproducibly every epoch, you should reset the generator by creating an instance (self.g) of the generator of the custom dataset class [54]. As a result i ensure that my data loading process is reproducible. The data loaded to my model in each execution of the whole algorithm should be the same to make the result comparable [54].

3. Also in the above custom dataset class i reseed the workers using worker_init_fn() to preserve reproducibility through by creating the following function seed_worker[54]:

```python
def seed_worker(worker_id):
    worker_seed = torch.initial_seed() % 2**32
    np.random.seed(worker_seed)
    random.seed(worker_seed)
```

Listing 1.29: Python function named seed_worker that preserves reproducibility by resseding of workers using worker_init_fn().

4. Additionally i create the function set_seed which has as a parameter the SEED number in order to preserve reproducibility [54]:

```python
#To preserve reproducibility
SEED = 42
def set_seed(SEED):
 random.seed(SEED)# Set python seed for custom operators.
 torch.manual_seed(SEED)
 np.random.seed(SEED)
 torch.cuda.manual_seed_all(SEED)
 torch.backends.cudnn.benchmark = False
 torch.backends.cudnn.deterministic = True
```

Listing 1.30: Python function named set_seed that preserves reproducibility.

The reproducibilty is preserved through this function, because the command random.seed(SEED) sets python seed for custom operators and the command torch .manual _seed(SEED) seeds the RNG (Random Number Generators) for all devices (both CPU and CUDA).The command torch.cuda.manual_seed_all(SEED) is used for multi-GPU [54].

Another reason for the maintenance of the reproduciblity by this function is the deterministic behavior achievement by these commands:

- torch.backends.cudnn.benchmark = False, which causes CUDA Deep Neural Network (cuDNN) to deterministically select an algorithm, possibly at the cost of reduced performance (the algorithm itself may be nondeterministic).
- torch.backends.cudnn.deterministic = True, which causes CUDA Deep Neural Network (cuDNN) to use a deterministic convolution algorithm, but may slow down performance. It will not guarantee that the training process is deterministic, if are used other libraries that may use nondeterministic algorithms.

5. Creation of the dataloader through the previously created custom dataset from pytorch [51] for the training and validation dataset of SimpleQuestions dataset [13]. The batch size is equal to 32.

```python
train_span_encoded_bert_dataset = BertQuestionAnsweringDataset(
    train_questions, tokenizer, train_entity_span_encoded_start_indexes,
    train_entity_span_encoded_end_indexes, train_relations_index, max(
    q_lengths), g)
valid_span_encoded_bert_dataset = BertQuestionAnsweringDataset(
    validation_questions, tokenizer ,
    validation_entity_span_encoded_start_indexes,
    validation_entity_span_encoded_end_indexes,
    validation_relations_index, max(q_lengths), g)
train_dataloader_span_encoded = torch.utils.data.DataLoader(
    train_span_encoded_bert_dataset, batch_size=32, num_workers = 0,
    worker_init_fn=seed_worker, shuffle=True, generator =
    train_span_encoded_bert_dataset.g)
val_dataloader_span_encoded = torch.utils.data.DataLoader(
    valid_span_encoded_bert_dataset, batch_size=32, num_workers = 0,
    worker_init_fn=seed_worker, shuffle=True, generator =
    valid_span_encoded_bert_dataset.g)
```

Listing 1.31: Code in python in order to create the dataloader from pytorch for the training and validation dataset of SimpleQuestions dataset. This is an example from the methodology of entity spans to be lists consisting of 0 and 1 regardless if i train and validate the full SimpleQuestions dataset or not.

6. I create the create_file function with a goal to create a file in .pt, in which the parameters for each model are stored. This function takes as an argument a name for the folder where the model will be saved (e.g. ”drive/MyDrive/”) and a name for the name of the file where each model is saved (e.g. ”relation_prediction.pt”).

```python
def create_file(model_dir, model_save_name):
  if not os.path.exists(model_dir):
    os.makedirs(model_dir)
  model_save_path = os.path.join(model_dir, model_save_name)
  return model_save_path
```

Listing 1.32: Python function named create_file which creates a file in .pt in which the parameters for each model are stored.

7. Additionally i create two functions for training and finding the best possible parameters for the best possible prediction of the relation and entity span, where the first named relation_prediction_function and the second entity_prediction_function respectively [26,27,55,56]. These functions have as arguments the corresponding training dataloader [51,55], the corresponding validation dataloader [51,55], the number of epochs, the optimizer [57,58,59,60,61,62], the scheduler [63], the learning rate [64,65] the corresponding neural network (relation prediction, entity span prediction) model [26,27], the loss function [66,67] and the dropout probability value [44].

The function relation_prediction_function returns the best possible model, the list including numbers from 0 to number of training epochs-1, the lists including the values of training losses and validation losses for all epochs respectively and the largest value of f1, while the function entity_prediction_function returns the same, with the only difference that returns two f1-score values: the largest value of f1 for entity span start and the largest value of f1 for entity span end.

```python
def relation_prediction_function(train_dataset, valid_dataset, train_dataloader, val_dataloader, model, loss_func, optimizer, scheduler, epochs, activation, lr, dropout_prob):
 list_loss_train = []
 list_epochs = []
 list_loss = []
 the_last_f1_score = 0
 patience = 3
 max_f1 = -1
 trigger_times = 0
 for epoch in range(epochs):
   train_dataset.set_epoch(epoch)
   valid_dataset.set_epoch(epoch)
   model.train()
   batch_losses = []
   for batch in train_dataloader:
    ids = batch['input_ids'].to(device)
    attention_masks = batch['attention_mask'].to(device)
    relation_indexes = batch['relation_indexes'].to(device)
    y_pred = model(ids, attention_masks)
    loss = loss_func(y_pred.cpu(),relation_indexes.cpu())
    batch_losses.append(loss.detach().numpy())
    #Delete previously stored gradients
    optimizer.zero_grad()

    #Perform backpropagation starting from the loss calculated in this epoch
    loss.backward()
    nn.utils.clip_grad_norm_(model.parameters(), 0.7)
    #Update model's weights based on the gradients calculated during backprop
    optimizer.step()
    scheduler.step()
   print(f"Epoch {epoch:3}: Train Loss = {sum(batch_losses)/len(train_dataloader):.5f}")
   list_loss_train.append(sum(batch_losses)/len(train_dataloader))
   with torch.no_grad():
     model.eval()
     F1 = []
     precision = []
     recall = []
     accuracy = []
     batch_losses_val = []
     for batch in val_dataloader:
       ids = batch['input_ids'].to(device)
       attention_masks = batch['attention_mask'].to(device)
       relation_indexes = batch['relation_indexes'].to(device)
       y_predict = model(ids, attention_masks)
       val_loss = loss_func(y_predict.cpu(),relation_indexes.cpu())
       batch_losses_val.append(val_loss)
       y_pr = torch.argmax(y_predict, dim=1).squeeze()
       Y_p = y_pr.tolist()
       y_val = relation_indexes.tolist()
```

```
      F1.append(f1_score(y_val, Y_p, average = 'micro'))
      precision.append(precision_score(y_val, Y_p, average = 'micro'))
      recall.append(recall_score(y_val, Y_p, average = 'micro'))
      accuracy.append(accuracy_score(y_val,Y_p))
    Valid_f1 = sum(F1)/len(val_dataloader)
    Valid_accuracy = sum(accuracy)/len(val_dataloader)
    print(f"Epoch {epoch:3}: Valid Loss = {sum(batch_losses_val)/len(
   val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid f1_score = {Valid_f1:.5f}")
    print(f"Epoch {epoch:3}: Valid Acurracy = {Valid_accuracy:.5f}")
    print(f"Epoch {epoch:3}: Valid Recall score = {sum(recall)/len(
   val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid Precision score = {sum(precision)/len
   (val_dataloader):.5f}")
    list_loss.append(sum(batch_losses_val)/len(val_dataloader))
    list_epochs.append(epoch)
    if Valid_f1 > max_f1:
      max_f1 = Valid_f1
      best_model = model
      print('The current f1 score:', Valid_f1)
    if Valid_f1 < the_last_f1_score:
      trigger_times += 1
      print('trigger times:', trigger_times)
      if trigger_times >= patience:
        print('Early stopping!\nStart to test process.')
        break;
    the_last_f1_score = Valid_f1
  return max_f1, list_loss_train, list_loss, list_epochs, best_model
```

Listing 1.33: Python function named relation_prediction_function in which the training and validation process of each methodology for the prediction of the relation index is applied.

Essentially the function named relation_prediction_function [26,27] gives through argmax function [68] the index of the highest probability value from the output of either 129 or 125 softmax function probability values [45,46,47,48] of the corresponding relation neural network that i created previously [26,27], because the length of the relation vocabulary in the case of training and validating the full SimpleQuestions dataset [13] is equal to 129 and the length of the relation vocabulary for the case of training and validating the SimpleQuestions subset [13] that includes only questions that are answerable over wikidata[14] is equal to 125.

So this predicted relation index as a number must be either between 0 and 128 or between 0 and 124, where this number correspond to the position of the unique relation ids in the corresponding relation vocabulary. This predicted index is correspond to an unique relation id of the relation vocabulary that i created previously [26,27]. So this relation id is the predicted relation id for the corresponding question from the relation prediction neural network [26,27].

```
def entity_prediction_function(train_dataset, valid_dataset,
   train_dataloader, val_dataloader, model, loss_func, optimizer,
   scheduler, epochs, activation, lr, dropout_prob):
  list_loss_train = []
  list_loss = []
  list_epochs = []
  the_last_f1_start_score = 0
  the_last_f1_end_score = 0
  patience = 3
  max_f1_start = -1
  max_f1_end = -1
  trigger_times = 0
```

```python
for epoch in range(epochs):
 train_dataset.set_epoch(epoch)
 valid_dataset.set_epoch(epoch)
 model.train()
 batch_losses = []
 for batch in train_dataloader:
  input_ids = batch['input_ids'].to(device)
  attention_masks = batch['attention_mask'].to(device)
  entity_span_start_indexes = batch['entity_start'].to(device)
  entity_span_end_indexes = batch['entity_end'].to(device)
  y_pred_start, y_pred_end = model(input_ids, attention_masks)
  y_pred_start = y_pred_start.squeeze()
  y_pred_end = y_pred_end.squeeze()
  start_preds = torch.argmax(y_pred_start,dim = 1)
  loss = loss_func(y_pred_start.cpu(),entity_span_start_indexes.cpu())+loss_func(y_pred_end.cpu(),entity_span_end_indexes.cpu())
  batch_losses.append(loss.detach().numpy())
   #Delete previously stored gradients
  nn.utils.clip_grad_norm_(model.parameters(), 0.7)
  optimizer.zero_grad()

   #Perform backpropagation starting from the loss calculated in this epoch
  loss.backward()

   #Update model's weights based on the gradients calculated during backprop
  optimizer.step()
  scheduler.step()
 print(f"Epoch {epoch:3}: Train Loss = {sum(batch_losses)/len(train_dataloader):.5f}")
 list_loss_train.append(sum(batch_losses)/len(train_dataloader))
 with torch.no_grad():
   model.eval()
   F1_start = []
   F1_end = []
   accuracy_start = []
   accuracy_end = []
   precision_start = []
   precision_end = []
   recall_start = []
   recall_end = []
   batch_losses_val = []
   for batch in val_dataloader:
     input_ids = batch['input_ids'].to(device)
     attention_masks = batch['attention_mask'].to(device)
     entity_span_start_indexes = batch['entity_start'].to(device)
     entity_span_end_indexes = batch['entity_end'].to(device)
     y_predict_start, y_predict_end = model(input_ids, attention_masks)
     y_predict_start = y_predict_start.squeeze()
     y_predict_end = y_predict_end.squeeze()
     val_loss = loss_func(y_predict_start.cpu(), entity_span_start_indexes.cpu())+loss_func(y_predict_end.cpu(), entity_span_end_indexes.cpu())
     batch_losses_val.append(val_loss)
     y_pr_start = torch.argmax(y_predict_start, dim=1)
     y_pr_end = torch.argmax(y_predict_end, dim=1)
     Y_p_start = y_pr_start.tolist()
     Y_p_end = y_pr_end.tolist()
     y_val_start = entity_span_start_indexes.tolist()
```

```
      y_val_end = entity_span_end_indexes.tolist()
      F1_start.append(f1_score(y_val_start, Y_p_start, average = 'micro'))
      F1_end.append(f1_score(y_val_end, Y_p_end, average = 'micro'))
      accuracy_start.append(accuracy_score(y_val_start, Y_p_start))
      accuracy_end.append(accuracy_score(y_val_end, Y_p_end))
      precision_start.append(precision_score(y_val_start, Y_p_start, average = 'micro'))
      precision_end.append(precision_score(y_val_end, Y_p_end, average = 'micro'))
      recall_start.append(recall_score(y_val_start, Y_p_start, average = 'micro'))
      recall_end.append(recall_score(y_val_end, Y_p_end, average = 'micro'))
    Valid_f1_start = sum(F1_start)/len(val_dataloader)
    Valid_f1_end = sum(F1_end)/len(val_dataloader)
    print(f"Epoch {epoch:3}: Valid Loss = {sum(batch_losses_val)/len(val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid f1_score for entity start span = {Valid_f1_start:.5f}")
    print(f"Epoch {epoch:3}: Valid Acurracy for entity start span = {sum(accuracy_start)/len(val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid Recall for entity start span = {sum(recall_start)/len(val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid Precision for entity start span = {sum(precision_start)/len(val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid f1_score for entity end span = {Valid_f1_end:.5f}")
    print(f"Epoch {epoch:3}: Valid Acurracy for entity end span = {sum(accuracy_end)/len(val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid Recall for entity end span = {sum(recall_end)/len(val_dataloader):.5f}")
    print(f"Epoch {epoch:3}: Valid Precision for entity end span = {sum(precision_end)/len(val_dataloader):.5f}")
    list_loss.append(sum(batch_losses_val)/len(val_dataloader))
    list_epochs.append(epoch)
    if Valid_f1_start >= max_f1_start and Valid_f1_end >= max_f1_end:
      max_f1_start = Valid_f1_start
      max_f1_end = Valid_f1_end
      best_model = model
      print('The current f1 score for entity start:', Valid_f1_start)
      print('The current f1 score for entity end:', Valid_f1_end)
    if Valid_f1_start < the_last_f1_start_score and Valid_f1_end < the_last_f1_end_score:
      trigger_times += 1
      print('trigger times:', trigger_times)
      if trigger_times >= patience:
        print('Early stopping!\nStart to test process.')
        break;
    the_last_f1_start_score = Valid_f1_start
    the_last_f1_end_score = Valid_f1_end

  return max_f1_start, max_f1_end, list_loss_train, list_loss, list_epochs, best_model
```

Listing 1.34: Python function named entity_prediction_function in which the training and validation process of each methodology for the prediction of the entity span start index and entity span end index is applied.

Essentially the function named entity_prediction_function [26,27] gives through argmax function [68] the index of the highest probability value from the output of 33 softmax function probability values [45,46,47,48] of the corresponding relation neural network that i created previously [26,27], because the biggest length of a question in words that is observed in SimpleQuestions dataset [13] is equal to 33. Thus these predicted indexes as numbers must be either between 0 and 32, where these numbers correspond to the position of these words in their question for each question.

So these words that their positions in the question correspond to these predicted indexes which are occurred from the entity span neural network [26,27] and the entity_prediction_function [26,27] are the predicted start and end words of entity label (span) of the corresponding question.

Additionally in order to improve the learning curve of the losses, but also the performance of the algorithms in general, i applied in the functions relation_prediction_function and entity_prediction_function [26,27] the method of early stopping, according to which if in one epoch no increase of the unique f1 value [20] for the prediction of the relation is observed for the validation dataset, but also the f1 values [20] for the entity start index and for the entity end index for predicting the corresponding indices for the entity span, then the training and validation process of the algorithm is stopped, because the performance of the model will not improve and there is a risk of overfitting [69,70,71].

8. I create two functions finding the results for the neural networks concerning the relation index and the entity span indexes (start,end), where the first named get_relation_results and the second get_entity_results respectively [27]. These functions have as arguments the corresponding training dataloader [51,55], the corresponding validation dataloader [51,55], the number of epochs ,the optimizer [57,58,59,60,61,62], the scheduler get_linear_schedule_with_warmup [63], the corresponding neural network model (relation prediction, entity span prediction) [26,27], the loss function [66,67], the learning rate [64,65] and the SEED number.

   These two functions return the list of training losses, the list of validation losses, and the list of how many epochs each model with each activation function needs to be fully trained.

```python
learning_rate  = 2e-5
dropout_prob = 0.3
epochs = 20
list_relation_training_losses = []
list_relation_validation_losses = []
list_relation_epochs = []
loss_func = nn.CrossEntropyLoss()
optimizer_name = "Adam"
def get_relation_results(activation, rel_vocab_length, learning_rate,
   optimizer_name, epochs, dropout_prob, loss_func, train_dataset,
   valid_dataset, train_dataloader, val_dataloader, SEED):
 set_seed(SEED)
 model = BertRelationsClassifier(activation, dropout_prob,
   rel_vocab_length).to(device)
 print(model)
 loss_func = loss_func.to(device)
 optimizer = getattr(torch.optim, optimizer_name)(model.parameters(), lr=
   learning_rate)
 scheduler = get_linear_schedule_with_warmup(optimizer, 0, len(
   train_dataloader)*epochs)
 f1, list_loss_train, list_loss_valid, list_epochs, best_relation_model =
    relation_prediction_function(train_dataset, valid_dataset,
   train_dataloader, val_dataloader, model, loss_func, optimizer,
   scheduler, epochs, activation, learning_rate, dropout_prob)
```

```python
    print(f1)
    torch.save(best_relation_model.state_dict(), create_file(f"drive/MyDrive/Relation_indexes_models_best_parameters/All_Questions/{activation}", f"relation_prediction_span_encoded_not_ignoring_questions_{activation}.pt") )
    print(f"The best parameters are saved for the activation function: {activation}")
    return list_loss_train, list_loss_valid, list_epochs
```

Listing 1.35: Python function named get_relation_results in which the training and validation process of each methodology for the prediction of the entity span start index and entity span end index is applied.

```python
learning_rate  = 2e-5
dropout_prob = 0.3
epochs = 20
list_entity_training_losses = []
list_entity_validation_losses = []
list_entity_span_encoded_epochs = []
loss_func = nn.CrossEntropyLoss()
optimizer_name = "Adam"
def get_entity_results(activation, max_question_length, learning_rate, optimizer_name, epochs, dropout_prob, loss_func, train_dataset, valid_dataset, train_dataloader, val_dataloader, SEED):
    set_seed(SEED)
    model = BertEntitySpanClassifier(activation, dropout_prob, max_question_length).to(device)
    print(model)
    loss_func = loss_func.to(device)
    optimizer = getattr(torch.optim, optimizer_name)(model.parameters(), lr=learning_rate)
    scheduler = get_linear_schedule_with_warmup(optimizer, 0, len(train_dataloader)*epochs)
    f1_start, f1_end, list_loss_train, list_loss_valid, list_epochs, best_entity_model = entity_prediction_function(train_dataset, valid_dataset, train_dataloader, val_dataloader, model, loss_func, optimizer, scheduler, epochs, activation, learning_rate, dropout_prob)
    print(f1_start,f1_end)
    torch.save(best_entity_model.state_dict(), create_file(f"drive/MyDrive/Entity_span_indexes_models_best_parameters/All_Questions/{activation}",f"entity_span_encoded_prediction_not_ignoring_questions_{activation}.pt") )
    print(f"The best parameters are saved for the activation function: {activation}")
    return list_loss_train, list_loss_valid, list_epochs
```

Listing 1.36: Python function named get_entity_results in which the training and validation process of each methodology for the prediction of the entity span start index and entity span end index is applied.

9. Additionally for all these methodologies in order to plot the learning curves between the training and validation losses that occurred from the Cross Entropy loss function [67] both for relation prediction and entity span prediction, i create the function plot_losses, which takes as arguments the the list including numbers from 0 to number of training epochs-1, the lists including the values of training losses and validation losses for all epochs respectively, a list with the names of the activation functions that i use to make my experiments ('LeakyReLU','Tanhshrink', 'Softplus','ELU') [43], the learning rate and the dropout probability [44,64,65] value that i use for my best experiments which are equal to 2e-3 and 0.3 respectively.

```python
def plot_losses(list_epochs, list_training_losses, list_validation_losses
    , activations, lr, dropout_prob):
 for activation in range(len(activations)):
  plt.plot(list_epochs[activation], list_training_losses[activation],
    label = f"{activations[activation]},{lr},{dropout_prob}-Training")
  plt.plot(list_epochs[activation], list_validation_losses[activation],
    label = f"{activations[activation]},{lr},{dropout_prob}-Validation")
  plt.legend(loc='center left', bbox_to_anchor=(1, 0.5), fontsize = '
    small')
  plt.xlabel('Number of epochs')
  plt.ylabel('Loss')
  plt.title("loss-score")
  plt.ylim(ymin = 3.5)
activations = ['LeakyReLU','Tanhshrink', 'Softplus','ELU']
plot_losses(list_relation_epochs, list_relation_training_losses,
    list_relation_validation_losses, activations, learning_rate,
    dropout_prob)
plot_losses(list_entity_span_encoded_epochs, list_entity_training_losses,
     list_entity_validation_losses, activations, learning_rate,
    dropout_prob)
```

Listing 1.37: Through the function plot_losses the learning curves for losses (relation prediction losses

1.5 Thesis Layout

My thesis consists of the following layers:

1. **Chapter 1: Introduction**

   This chapter states the basic problem of this thesis, explains the general scope of the thesis, presents the aim and the objectives of this thesis and describes in detail the main methodology that is followed in the thesis.

2. **Chapter 2: Preliminaries**

   This chapter provides the background knowledge necessary to understand the rest of the thesis. It refers mainly on question answering over knowledge graphs and deep learning and more briefly on semantic parsing (especially through SPARQL).

3. **Chapter 3: Related Work**

   In this chapter recent work that is related with the question answering over knowledge graphs and semantic parsing are presented and discussed briefly.

4. **Chapter 4: Dataset Generation** This chapter explains concisely how the SimpleQuestions dataset, which is created on Freebase knowledge graph, is transformed from Freebase knowledge graph to Wikidata. Also in this chapter it is explained how the entity labels for the SimpleQuestions dataset questions is found through SPARQL in order to add them to the initial SimpleQuestions dataset annotated on Wikidata as a new column.

5. **Chapter 5: The Question Answering model**

   This chapter explains how the QA engines for this thesis are created. More specifically three types of QA engines are created: The first one including only spacy library, the second one including the Sentence Transformers similarity model combined with cosine similarity metric and the thirf including only jaccard similarity metric.

6. **Chapter 6: Evaluation** This chapter displays the experiments with the best evaluation results, i.e loss learning curve, training and validation losses values, as well as precision, accuracy, f1 and recall scores. Additionally for each one of these experiments, their results are interpreted and thus some conclusions are drawn.

7. **Chapter 7: Conclusions and future Work** Finally, we summarize our contributions and findings. Also based on our findings, the creation of a question answering engine for datasets with more complex questions(e.g ComplexQuestions) for is discussed as a direction for future research.

2. PRELIMINARIES

2.1 Summary

This thesis is referring to the following concepts: RDF data, linked data, Knowledge Graphs, RDF query language SPARQL [2,3,4,5], semantic parsing, Question Answering over Knowledge Graphs, Transformers and BERT.

Before i explain the above concepts, it is important to explain concisely some terminologies. The natural language question (NLQ), is also referred to as question or utterance. The meaning representation is also referred to as logical forms or formal queries.The execution results or answers for a formal query are also referred to as denotations.

RDF data

RDF stands for Resource Description Framework and is a standard for describing web resources and data interchange, developed and standardized with the World Wide Web Consortium (W3C). W3C includes the foundational concepts, semantics and specifications for different formats. Essentially RDF is a general method of describing and representing data by defining relationships between resources such as documents, physical objects, people, abstract concepts and data objects in a graph format [72,73,74].

The first syntax defined for RDF was based on the Extensible Markup Language (XML). Although the XML syntax was the only syntax option specified at first, standards for encoding RDF statements now include the following three syntaxes[72,73,74]:

1. Terse RDF Triple Language (Turtle) which is the most popular text syntax for RDF statements. The W3C describes it as a ”compact and natural text form” that includes abbreviations for commonly used patterns.
2. JavaScript Object Notation for Linked Data (JSON-LD), which uses the JSON syntax for RDF statements.
3. N-Triples, which is a subset of the Turtle syntax, designed to be a simpler text-based format for RDF statements for improved ease of use by humans writing statements. The simpler format also makes it easier for programs to create and parse RDF statements.

RDF allows describing anything: persons, animals, objects, and concepts of any kind. They are considered resources. Also RDF is a standard way to make statements about resources. Every resource is identified by a URI or IRI. Uniform Resource Identifier (URI) is a standardized format for identifying a resource, whether abstract or physical. IRI (Internationalized Resource Identifier) is a protocol standard that expands the Uniform Resource Identifier (URI) one. IRIs are similar and complementary to URIs,enabling the use of international character sets. URI standard only uses the US-ASCII character set. IRI allows containing characters from Unicode character set. IRI allows using Chinese, Japanese, Korean, and Cyrillic characters. IRIs could appear in all the positions of an RDF.

For example, the IRI for the Rosetta Stone is:

https://dbpedia.org/describe/?url=http

RDF allows combining information from different datasets like Wikidata [14], DBpedia [75,76] etc. RDF statements are also known as triples and consist of three components [72,73,74]:

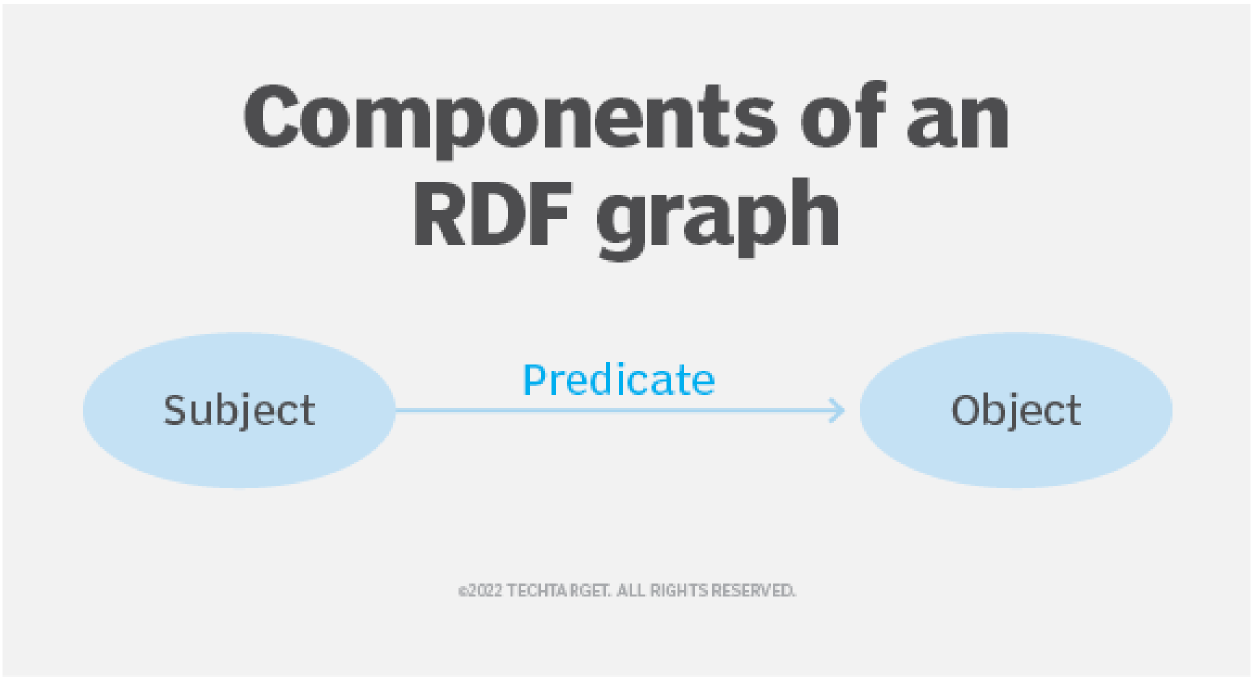
Components of an RDF graph
Subject
Predicate
Object
©2022 TECHTARGET. ALL RIGHTS RESERVED.

Figure 2.1: An RDF statement consists of three components, referred to as a triple: subject, predicate, object

1. Subject is a resource being described by the triple.
2. Predicate describes the relationship between the subject and the object.
3. Object is a resource that is related to the subject.

The RDF standard provides for three different types of nodes [72,73]:

1. Uniform Resource Identifier (URI) (mentioned earlier): Uniform Resource Locator (URL) is a type of URI that is commonly used in RDF statements. When W3C updated the RDF specification to version 1.1 in 2014, it added Internationalized Resource Identifier (IRI) as a node type.
2. Literal is a specific data value and can be a string, a date or a numerical value. They could not appear in the subject or predicate positions, just in the object position. Literal values are expressed using the URI or IRI format.
3. Blank node identifier is also known as an anonymous resource or a *bnode*. It represents a subject about which nothing is known other than the relationship. Blank node identifiers use special syntax to identify them.

The benefits of RDF include the following [72]:

1. A consistent framework encourages the sharing of metadata about internet resources.
2. RDF's standard syntaxes for describing and querying data enable software that uses metadata to work more easily.
3. The standard syntax and query capability enable applications to exchange information more easily.

4. Searchers get more precise results from searching based on metadata than they would from indexes derived from full-text gathering.

5. Intelligent software agents have more precise data to work with and are more precise in what they deliver to users.

RDF has the following difficulties[72]:

1. Standardization of vocabulary for describing RDF resources can be difficult.

2. Choosing the most appropriate syntax format for RDF statements takes practice. Different formats may be easier to use for specific implementations, especially for the systems that generate the RDF statements.

3. Choosing the most appropriate RDF query language for an application depends on the features of the query language and the specific needs of the application.

Linked data

In order to understand better the meaning if the term ”Linked data”, suppose we have the following example [77]: Let us say that there two relational tables of two different databases. The first contains data about movies, with titles, actor and director names. The second is a collection of theaters in a town where such movies are playing. So It becomes clear that there is an implicit link between the rows of those two tables.

let’s say that we wanted to publish this information so that it could be easily consumed by other Web sites or computer programs. For example, so that a program can automatically query our Web site and any other site that has movie scheduling information in order to show a complete view in one place.

The traditional Web could not provide an standard way to make such tables available on the Web and have them be accessible from an application. For example we could export the values in CSV and putting them on a HTTP Server.

This approach has the important problem that CSV files could be understandable for machines but not human readable. Even a simple difference such as changing the order of the columns can make it impossible for an automated computer program to consume the data, since they would have to have separate rules for every kind of CSV anywhere that talks about movies or theaters. So the Linked data comes with the goal to publish structured data in such a way that it can be easily consumed and combined with other Linked Data. About Linked data in order to achieve their goal, these four basic principles must be respected [77,78] which are explained through the previous example with the tables of different databases (movies and theatres):

1. **Use URIs as names for things**: Instead of using application-specific identifiers—database keys, UUIDs, incremental numbers, etc.—mapping names to a set of URIs. Each identifier must map to one single URI. For example, each row of those two tables is now uniquely identifiable using its URI.

2. **Use HTTP URIs so that people can look up those names**: Make URIs roughly, accessible via HTTP as we do for every human-readable Web page, i.e dereference-able. So every single row of our tables is now fetch able and uniquely identifiable anywhere on the Web.

3. **When someone looks up a URI, provide useful information, using the standards such as RDF* and SPARQL**: Have the web server reply with some structured data when invoked and model the data with RDF. Here there is the need to perform a paradigm shift from a relational data model to a graph one.

4. **Include links to other URIs so that they can discover more things**:Once all the rows of the tables have been uniquely identified, made dereferenceable through HTTP, and described with RDF, the last step is providing links between different rows across different tables. The main goal here is to make explicit those links that were implicit before shifting to the Linked data approach. In our example, movies would be linked to the theaters in which they are playing.

Once these tables have been published, the Linked Data rules achieve their goal because people across the Web can start referencing and consuming the data in our rows easily. If there is a link from movies of one of the tables to external popular data sets such Wikipedia and IMDB then it is even more easier for people and computers to consume these data and combine it with other data [77].

It is easily understood that Linked Data is one of the core pillars of the Semantic Web. The Semantic Web is about making links between datasets that are understandable not only to humans, but also to machines, and Linked Data provides the best practices for making these links possible [78]. Essentially Linked Data is a set of design principles for sharing machine-readable interlinked data on the Web [78].

Linked data is not only about exposing data using Web technologies, but a Web of Data could be built through Linked data[78]. Imagine that there are hundreds of different datasets published on the Web according to the Linked Data principles: thousands different identifiers that people can rely to grab data about books, movies, actors, cities, or anything else. Consequently all these datasets form a giant Web-scale database which could be potentially embodied in many applications and the reference in this database whenever there is a need becomes more easier [78].

By considering the example with the databases of books and movies, instead of having links citations simply between books or Web pages, this allows links between anything to be followed for more information. For example if a single author has published in two different journals and both journals expose their catalogs as Linked Data, and the author's bio is on DBpedia [75,76], then the corresponding application can easily mash it all together with a simple query, automatically. As if all the data was in one database [77].

Open Data can be freely used and distributed by anyone, i.e can be made available to everyone without links to other data. At the same time, data can be linked without being freely available for reuse and distribution [78]. Linked Open Data is a powerful mixture of Linked Data and Open Data: it is both linked and uses open sources. One notable example of an Linked Open Data set is DBpedia[75,76] – a crowd-sourced community effort to extract structured information from Wikipedia [79] and make it available on the Web [78].

A benefit of Linked Data is that breaks down the information that exist between various formats and brings down the fences between various sources. Also it facilitates the extension of the data models and allows easy updates. So, data integration and browsing through complex data become easier and much more efficient [78].

Semantic Parsing

Semantic parsing is the task of translating a natural language (NL) question $x \in X$, $X \subset$

N into a machine-interpretable representation $q \in Q$ of its meaning in a given formal language (FL), such as SPARQL [2,3,4,5]. Here, X is a subset of the natural language N, which specifies the scope (domain) of some semantic parser, which is a concept that refers to the knowledge or competence area in which a particular semantic parser might be specialized. N contains all possible questions that are considered to be part of a natural language (for example English) and the X, which is subset of it, may be restricted to those questions that are in the domain/scope of the semantic parser. Thus the set X is the set of questions for which the system using the semantic parser should provide a response that is not an out-of-domain response. For example, if one can make a semantic parser to interpret commands for a basic autonomous vacuum cleaner, then only a small fraction of possible NL questions is meaningful, that is, should result in a change of movement or air flow. Other questions, for example asking about the weather, should be considered out-of-domain. [1].

The produced logical form *q* could be assumed as correct if it most accurately reflects the meaning of the NL question *x* under the restrictions of the scope of the parser and the used formal language. However, it is difficult to clearly express in a formal way the concept of meaning. One property of the correct logical form is that its execution in the environment *E* (which is denoted by *q(E)*) leads to the desired change of the environment. For example, executing a query in a computer system to retrieve the results from a database yields the set of results. However, the correctness of execution results is not sufficient because multiple logical forms could give the expected results but not all of them correctly capture the meaning of the NL question. For example, translating “What is two plus two” to $2*2$ yields the same result but is not the correct query [1].

Regardless of the specific semantic parsing approach taken, the construction of logical forms for KGQA requires taking several decisions regarding their structure and content. So, a semantic parser must perform the tasks that are described briefly below [1]. In order to understand better these tasks consider the question “Where was the author of Dune born?” as an example:

1. **Entity Linking**: In the context of KGQA entity linking is the task of deciding which KG entity is referred to by (a certain phrase in) the NLQ *q*. In the case of our example, the entity linker must identify :Dune_(novel) as the entity being referred to by “Dune” in the NLQ. The large number (often millions) of entities forms a core challenge in entity linking with large-scale KGs. A particular phrase (e.g. “Dune”), if considered without context, can refer to one of possibly many different entities (Dune can refer to the novel, the David Lynch film, the Denis Villeneuve film, the landform, or several other possibilities). It is essential to take the context of the phrase into account to correctly disambiguate the phrase to the correct entity. The large number of entities makes it practically unfeasible to create fully annotated training examples to learn lexical mappings for all entities. Consequently, statistical entity linking systems need to generalise well to unseen entities. Many modern KGQA systems externalise the task of entity linking by employing a standalone entity linking system like S-Mart [80] for Freebase [81].

2. **Identifying Relations**: It is essential to determine which relation must be used in a certain part of the logical form. Similarly to entity linking, we need to learn to map natural language expressions to the relations of the KG. However, unlike in entity linking, where entities are expressed by entity-specific noun phrases, relations are typically expressed by noun and verb phrase patterns that use less specific words. For instance, in our example NLQ, the relations :author and :birthplace are explicitly

referred to by the phrases “author of” and “born”, respectively. In other words birthplace is a relation mapping from a person to a place where that person was born and author of is a relation mapping from a person to a book about which this person is the author of. However, depending on the KG schema, relations may also need to be inferred, like in “Which American wrote Dune?” where “American” specifies an additional (SPARQL) constraint ?answer :residentOf :USA without explicitly mentioning the relation :residentOf [1].

3. **Identifying Logical/Numerical Operators**: Sometimes questions contain additional operators on intermediate variables/sets. For example, ”How many books did Frank Herbert write?” implies a COUNT operation on the set of books written by Frank Herbert. Other operators can include ARGMAX/MIN, comparative filters (e.g. “older than 50”), set inclusion (e.g. ”Did Frank Herbert write Dune?”) etc. Like entities and relations, identifying such operators is primarily a lexicon learning problem. However, compared to entities and relations, there is a fixed small set of operators, which depends on the chosen formal language and not on the KG.

4. **Determining Logical Form Structure**: In order to arrive at a concrete logical form, a series of decisions must be made regarding the structure of the logical form, i.e. how to arrange the operators, relations and entities such that the resulting object accurately captures the meaning of the question and executes to the intended answer. For example, if FunQL is used as target formal language, a decision must be taken to supply :Dune_(novel) as the argument to the relation function :author, and to pass this subtree as the argument to :birthplace, yielding the final query :birthplace(:author(:Dune_(book))).

Knowledge Graphs

A knowledge graph (KG) is a formal representation of facts pertaining to a particular domain (including the general domain). It consists of entities denoting the subjects of interest in the domain, and relations (The words predicate or property are often used interchangeably with relations) denoting the interactions between these entities. A relation can link two entities and also an entity to an data value of the entity (such as a date, a number, a string etc.) which is referred to as a literal. Several large-scale knowledge graphs have been developed, which contain general factual knowledge about the world. Some of the most prominently known examples are WikiData[14], Freebase [81] and DBpedia [75,76].

Formally, let E = $\{e_1, \ldots, e_{n_e}\}$ be the set of entities, L be the set of all literal values, and P = $\{p_1, \ldots, p_{n_e}\}$ be the set of relations connecting two entities, or an entity with a literal. A triple $t \in E \times L \times (E \cup L)$ is defined as a statement comprising of a subject entity, a relation, and an object entity or literal, describing a fact. Then, a KG K is a subset of $E \times L \times (E \cup L)$ of all possible triples representing facts that are assumed to hold [1].

SPARQL

SPARQL, a recursive acronym for “SPARQL Protocol and RDF Query Language”, is one of the most commonly used query languages for KGs and is supported by many publicly available KGs like WikiData[14], Freebase [81] and DBpedia [75,76]. The central part of a SPARQL query [2,3,4,5] is a graph pattern, composed of resources from the KG (i.e. entities and relations) and variables to which multiple KG resources can be mapped. This graph patterns specifies conditions on variables, whereas the SELECT clause determines which variables are returned as the answer [1,2,3,4,5].

Essentially SPARQL is a query language that requests and retrieves data that use the RDF format (RDF data section). Thus, the database is a set of items with the format subject-predicate-object previously explained. Also SPARQL allows using query operations such as JOIN, SORT, AGGREGATE, along with others [73].

The following example permits obtaining the names and emails of all the persons in the FOAF (Friend Of A Friend) dataset [73]:

```
PREFIX foaf:
SELECT ?name
       ?email
WHERE
  {
    ?person  a          foaf:Person .
    ?person  foaf:name  ?name .
    ?person  foaf:mbox  ?email .
  }
```

Listing 2.1: Example in RDF language SPARQL which permits obtaining the names and emails of all the persons in the FOAF (Friend Of A Friend)vocabulary is found.

The PREFIX clause declares the label foaf representing the URI indicated between angle brackets. The SELECT clause joins together all the RDFs where the predicate a corresponds to a person according to the foaf dataset, and the person's name and mailbox [73].

The join result consists of a set of rows with the name and email of each person in the dataset. As a person may have multiple names and mailboxes, then the returned set of results could contain multiple rows for the same person [73]. It is worth mentioning one for the cases that SPARQL used is for geospatial data. For example work [82] presents the dataset GEOQUESTIONS1089 for benchmarking geospatial question answering engines. GEOQUESTIONS1089 is the largest such dataset available presently and it contains 1089 questions, their corresponding GeoSPARQL or SPARQL queries and their answers over the geospatial knowledge graph YAGO2geo. The dataset GEOQUESTIONS1089 is used to evaluate the effectiveness and efficiency of geospatial question answering engines GeoQA2 (an extension of GeoQA developed by the AI team of the University of Athens (UoA)) and the system of Hamzei et al. (2021).

Also about question answering engine GeoQA2 is presented more analytically in [83]. Essentially GeoQA2 consist the most recent version of the engine GeoQA, originally proposed by Punjani et al and has been designed to work over the geospatial knowledge graph YAGO2geo. GeoQA2 improves GeoQA by being able to answer more complex geospatial questions (e.g., questions involving aggregates and arithmetic comparisons), having more effective individual components (e.g., for named entity recognition and disambiguation), supporting more templates for generating GeoSPARQL queries, and executing these queries more efficiently by using a technique based on materializing topological relations.

Large amounts of geospatial data [82,83,84] have been made available recently on the linked open data cloud and on the portals of many national cartographic agencies (e.g., OpenStreetMap data, administrative geographies of various countries, or land cover/land use data sets). These datasets use various geospatial vocabularies and can be queried using SPARQL or its OGC-standardized extension GeoSPARQL. [84] as a paper their authors go beyond these approaches to offer a question answering service on top of linked geospatial data sources. For the evaluation of the algorithms for the architecture that this paper follows is used a set of 201 natural language questions.

Question Answering over Knowledge Graphs

KGQA can be defined as the task of retrieving answers to NL questions or commands from a KG. While the expected answers could take on complex forms, such as ordered lists (e.g. for the command “Give me Roger Deakins movies ranked by date.”), or even a table (e.g. for the question “When was which Nolan film released?”), the vast majority of works on KGQA assume the answer to be of a simpler form: a set of entities and/or literals or booleans [1].

A more formal explanation it can be defined as [1]: Let G be a KG and let q be a natural language question. Then the set of all possible answers is specified A as the union of the power set $P(E \cup L)$ of entities E and literals L in G, the set of the numerical results of all possible aggregation functions $f : P(E \cup L) \mapsto \mathbb{R}$ (such as SUM or COUNT), and the set True,False of possible boolean results, which is needed for yes/no questions. The task of KGQA is to return the correct answer $a \in A$, for a given NL question q. The most common way in which KGQA approaches accomplish this is by creating a formal query representing the semantic structure of the question q, which when executed over G, returns A. Thus, KGQA can be assumed as an application area of semantic parsing.

Transformers and BERT

Transformer [1,6,7,15,16,17,85] networks have been recently proposed for NLP tasks and are fundamentally different from the common RNN and CNN architec tures. Especially compared to RNNs, which maintain a recurrent state, transformers use multi head self-attention to introduce conditioning on other timesteps. This allow for more parallel training, and offer better interpretability, unlike RNNs, which process the input sequence one time step at a time.[1,6].

Essentially, transformers consists of several layers of multi-head self-attention with feed-forward layers and skip connections. Multi-head self-attention is an extension of the standard attention mechanism [86], with two major differences [6]:

1. Attention is applied only within the input sequence
2. Multiple attention heads enable one layer to attend to different places in the input sequence

In order to understand how transformers work, let the transformer consist of L layers, each ($l \in \{1, \ldots, L\}$) producing vectors $h_1^{l+1}, ..., h_N^{l+1}$, which are also used as inputs of the l+1-th transformer layer. The inputs $h_1^1, ..., h_N^1$ to the first transformer layer are the embeddings of the input tokens $h_1, ..., h_T$. Based on the above hypothesis we have about the transformer encoder layer and transformer decoder layer:

Transformer Encoder Layer: A single transformer layer for an encoder consists of two parts [1]:

1. a self-attention layer
2. a two-layer MLP.

The self-attention layer collects information from neighbouring tokens using the multi-head attention mechanism that attends from every position to every position. With $h_t^{(l-1)}$ being the outputs of the previous transformer layer for position t, as referred before, the multi-head self-attention mechanism computes M different attention distributions (one for every head) over the *t* positions. For every head ($m \in \{1, \dots, L\}$), self-attention score in each layer *l* for position t is computed through the following equations:

$$\begin{aligned} \boldsymbol{q}_t^{l,m} &= \boldsymbol{W}_Q^{l,m}\boldsymbol{h}_t^l \\ \boldsymbol{k}_t^{l,m} &= \boldsymbol{W}_K^{l,m}\boldsymbol{h}_t^l \\ \boldsymbol{v}_t^{l,m} &= \boldsymbol{W}_V^{l,m}\boldsymbol{h}_t^l \end{aligned} \tag{2.1}$$

$$\begin{aligned} a_{t,j,l,m} &= \frac{(\boldsymbol{q}_t^{l,m})^T\boldsymbol{k}_j^{l,m}}{\sqrt{\boldsymbol{d}_k}} \\ a_{t,j,l,m} &= \frac{e^{a_{t,j,l,m}}}{\sum_{i=0}^{T} e^{a_{t,i,l,m}}} \\ \boldsymbol{s}_t^l &= \sum_{i=0}^{T} a_{t,j,l,m}\boldsymbol{v}_j \end{aligned} \tag{2.2}$$

Essentially the attention score is calculated through a softmax function [45,46,47,48] of dot products between the input vectors h_t of position t and h_j of position j in the layer l after multiplication with the input vectors h_t, h_j and W_Q, and W_K , which are trainable matrices for head *m* of layer *l*. Self-attention score is normalized by the square root of the dimension d_k of the $k_t^{l,m}$ vectors.

From the equation (2.2) $\boldsymbol{s}_t^l$ are defined as the intermediate representation vectors for each input position, which are computed as the concatenation of the *M* heads' summary vectors, each computed as a $a_{t,j}$ -weighted sum of input vectors $h_1^1, ..., h_N^1$, which are first projected using the matrix $W_V^{l,m}$

The summaries $\boldsymbol{s}_t^{(m)}$ for head *m*, as computed above, are concatenated and passed through a linear transformation to get the final self-attention output vector $\boldsymbol{u}_j$:

$\boldsymbol{u}_j = \boldsymbol{W}_O \oplus_{m=0}^{M} (\boldsymbol{s}_t^l)^{(k)}$

where $\boldsymbol{W}_O$ is a trainable matrix and $\oplus$ denotes concatenation.

The output of the *l*-th transformer layer (which is also the input to the *l*+1-th layer as referred before) is then given by applying a two-layer feed-forward network [87] with a ReLU activation function [88] on $\boldsymbol{h}_t^l$. Finally the whole transformer layer is shown through the following equations [1,6]:

$$\begin{aligned} \boldsymbol{q}_t^{l,m} &= \boldsymbol{W}_Q^{l,m}\boldsymbol{h}_t^l \\ \boldsymbol{k}_t^{l,m} &= \boldsymbol{W}_K^{l,m}\boldsymbol{h}_t^l \\ \boldsymbol{v}_t^{l,m} &= \boldsymbol{W}_V^{l,m}\boldsymbol{h}_t^l \end{aligned} \tag{2.3}$$

where W_A W_B are also trainable matrices [1,6].

Transformer Decoder Layer:

The decoder should take into account the previously decoded tokens, as well as the encoded input. Because of this the following two changes are made in the decoder, compared to the presented transformer encoder layer [1]:

1. The usage of a causal self-attention mask in self-attention
2. The addition of a cross-attention layer.

The causal self-attention mask is necessary to prevent the decoder from monitoring to future tokens, which are available during training but not during test. As such, this is more of a practical change. The cross-attention layer enables the decoder to monitor to the encoded input. Denoting the encoded input sequence vectors as $\boldsymbol{x}_i$, cross-attention is implemented in the same way as self-attention, except the vectors used for computing the key and value vectors are the $\boldsymbol{x}_i$ vectors. The decoder layer then becomes:

A more formal explanation it can be defined as: Let G be a KG and let q be a natural language question. Then the set of all possible answers is specified A as the union of the power set $P(E \cup L)$ of entities E and literals L in G, the set of the numerical results of all possible aggregation functions $f : P(E \cup L) \mapsto \mathbb{R}$ (such as SUM or COUNT), and the set True,False of possible boolean results, which is needed for yes/no questions. The task of KGQA is to return the correct answer $a \in A$, for a given NL question q. The most common way in which KGQA approaches accomplish this is by creating a formal query representing the semantic structure of the question q, which when executed over G, returns A. Thus, KGQA can be assumed as an application area of semantic parsing [1].

BERT consists a model that is created from Devlin et al. [7], who pretrained transformers on a large collection of unsupervised language data, following previous works on transfer learning from pretrained transformer based language models [11]. In particular BERT [1,6,7,15,16,17,32,33,85] is a transformer-based bidirectional model which builds on pre-training a masked language model (MLM).

The fact that BERT [1,6,7,15,16,17,32,33,85] has been pre-trained as a masked language model allows this model to use tokens both from the preceding (left) tokens as well as following (right) tokens to build representations of tokens. In contrast, a left-to-right language model of OpenAI-GPT [11] would only be allowed to use tokens to the left of the current position. In other words the left-to-right LM pretraining of OpenAI-GPT constrained the model to look only at the past [1,6].

More specifically BERT is pre-trained on a large corpus of text (BookCorpus (800M words) and English Wikipedia (2500M words)) using an objective that consists of two tasks:

1. **Masked LM task**

   The MLM task trains the model to predict words that have been omitted from the input. In order to do this, 15% of all tokens in a sentence are replaced either with a special [MASK] token in 80% of the cases, feeding this resulting partially masked sequence into the model and training the model to predict the words that have been masked out, given the other words or a random token (in 10% of the cases). In the remaining 10% of the cases, the selected token is not replaced. This is done to reduce the inconsistency between training and test, because during testing, no special mask tokens are used. The main objective of the masked LM task is to predict the (20%) masked words in the sentence [1,6].

2. **Next Sentence Prediction task**

   This task consists of predicting whether two sentences follow each other in the corpus (or generally in a text) or not. To train for this task, in 50% of the cases, two adjacent sentences are taken from the corpus, and in the other 50%, two random sentences are taken. The model is trained to distinguish between the two cases. The Next Sentence Prediction (NSP) task is useful for downstream tasks such as single sentence classification and also entailment, which is formulated as classification of sentence pairs, but also for single sentence classification [1,6].

The fact that BERT is pre-trained as a masked language model enables BERT's feature vectors to include information both from the previous tokens as well as the following tokens [1,6]. Additionally BERT is pre-trained on a sentence pair classification task. Specifically, it is trained to predict whether one sentence follows another in a text [6].

The model used in BERT is simply a transformer encoder with learned positional embeddings rather than the original fixed sinusoid ones. Before the input (for example a sentence such as "What songs have Nobuo Uematsu produced") is fed to the model, it is first tokenized to (sub)word level using a WordPiece [89,90] vocabulary of 30000 tokens (["What", "songs", "have", "no", "#buo", "u", "#ema", "#tsu", "pro duced"], for the previous sentence). More common words are taken as words("What","songs","have"),while uncommon words are split into subword units ("nobuo" ["no", "#buo"]) [1,6].

This method significantly reduces vocabulary size and the amount of rare words without dramatically increasing sequence length. The input sequence is also padded with a special [CLS] token at the beginning and a special [SEP] token at the end[6]. Also if the input is a sentence pair, such as "The dog ate." and "Man bites dog", they are separated using another [SEP] token and they would be fed to the model as, "[CLS] The dog ate. [SEP] Man bites dog [SEP]" [1]. Also in addition to positional embeddings, a special segment embedding is added to the WordPiece embeddings. This segment embedding simply specifies whether the token is part of the first sentence or the second [1].

Finally the WordPiece [89,90] token sequence is then embedded into a sequence of vectors. The resulting embedding vectors are fed through the transformer, which uses several lay ers of multi-head self-attention [1,6] and feedfoward [87] layers [6]. The output vectors for each token can be used for sequence tagging tasks, while the vector asso ciated with the [CLS] token at the beginning of the sequence is assumed to capture relevant information about the input sequence as a whole, since BERT has been pre-trained for sentence pair classification [1].

3. RELATED WORK

3.1 Summary

Firstly i refer analytically in works that are related with the Question Answering over Knowledge Graphs (KGQA). The most of the related works that i refer about this thesis are related with the Question Answering over Knowledge Graphs (KGQA) and also with the SimpleQuestions dataset, because these are the main points the main points around which my thesis has been created. Additionally i refer some related works about semantic parsing, which consists a useful tool in order to retrieve e.g SPARQL [2,3,4,5] queries for each question of the SimpleQuestions dataset [13] its corresponding entity label by using its corresponding Wikidata [14] entity id that it is already exist in the SimpleQuestions dataset [13].

Especially in the case of this thesis, where in the SimpleQuestions dataset [13] exists single fact based questions and consequently a single subject entity and a relation need to be predicted, the corresponding queries could be created by using SPARQL [2,3,4,5]. For questions such as, ”Where was Frank Herbert born?” and other examples of this type of questions, queries can be created following this logic: given a knowledge graph *G* that contains triples of the form (*s*, *p*, *o*), where *s* is a subject entity, *p* a predicate (also denoted as relation), and *o* an object entity. Therefore given a natural language question represented as a sequence of words $q = w_1,......,w_T$, then through simple QA must be found the the set of triples ($\hat{s}$, $\hat{p}$, $\hat{o}$) , for which right subject $\hat{s}$ and predicate $\hat{p}$ that question *q* refers to and which characterize the set of triples in G that contain the answer to *q*.

For fixed-structure prediction tasks like these described above, simple text classification and ranking methods could be used to predict the different parts of the target formal query given the natural language question as input [1]. Also for single-fact based prediction tasks, systems relying on standard classification models, can achieve state-of-the-art performance [91,92].

But for such approaches the target formal queries consist only of one subject entity and one relation and this means that they could be predicted using two separate classifiers, receiving the NLQ as input and producing an output distribution over all entities and all relations in the KG, respectively. So, while the KG relation mentioned or implied in the question, could be predicted successfully with this approach, the entities could not be predicted with equal success, because large KGs like Freebase [81] contain a huge amount of entities and training datasets can only cover a small fraction of these and therefore many classes remain unseen after training. This makes it difficult to apply the approach described above for relation classification to entity prediction. Consequently this approach applied with success to the relation classification, but not the same is the case with entity classification [1].

Therefore, several works on SimpleQuestions [13] initially only perform relation classification and rely on a two-step approach for entity linking instead. In such an two-step approach, first an entity span detector, which does not need to learn representations for entities (like encoder networks used in classifiers), avoiding the need for data covering all entities, is applied to identify the entity mention in the NLQ. Secondly, simple text based retrieval methods can be used to find a limited number of suitable candidate entities. The best fitting entity, given the question, can then be chosen from this candidate set based on simple string similarity and graph-based features. The entity span detector can be implemented based on classifiers as well, either as a pair of classifiers (one for predicting the start position and one for predicting the end position of the span) or as a collection of independent classifiers, where in this case the entity span detector representing a sequence tagger (one for every position in the input, similar to a token classification network), where each classifier predicts whether the token at the corresponding position belongs to the entity span or not [1].

A simple concrete example of a full KGQA semantic parsing algorithm, which gives competitive performance on SimpleQuestions [13], relying on classification models, is the approach proposed by Mohammed et al. [1,91]. This QA system follows the following steps for predicting the entity-relation pair constituting the formal query:

1. Entity span detection: A bidirectional Long-Short Term Memory network (BiLSTM) [93] is used for sequence tagging, i.e. to identify the span *a* of the NLQ *q* that mentions the entity. In the example “Where was Frank Herbert born?”, the entity span could be “Frank Herbert”.
2. Entity candidate generation: Given the entity span, all KG entities are selected whose labels (almost) exactly match the predicted span.
3. Relation classification: A bidirectional gated recurrent unit (BiGRU) [94] encoder is used to encode *q*. The final state of the BiGRU is used to compute probabilities over all relations using a softmax output layer, which is given from the equation [45,46,47,48]:
$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3.1}$$
.
4. Logical form construction: The top-scoring entities and relations from previous steps are taken, and all their possible combinations are ranked based on their component scores and other heuristics. The top-scoring pair is then returned as the predicted logical form of the question.

For training the full system, the relations given in the formal queries of the dataset are used as correct output classes. For the span detector, we first need to extract “pseudo-gold” entity spans since they are not given as part of the dataset. This is done by automatically aligning NLQs with the entity label(s) of the correct entity provided in the training data, i.e. the part of the NLQ that best matches the entity label is taken as the correct span to train the sequence tagging BiLSTM.

I choose to refer this work because it has some similarities with the methodology that i followed in this thesis (described in 1.4 methodology section in chapter 1 - Introduction). These similarities (and also the differences) are:

1. Comparing with the above entity span detection and entity candidate generation steps, the simlarity is that i create entity spans for each question of the SimpleQuestions dataset [13], which includes the entity label, in order to find the entity label start and entity label end index for each question.

   The first difference is that in this thesis firstly based on the entity id of each question of the training and the validation dataset, which occurred from the transformation of SimpleQuestions dataset from Freebase to Wikidata [95], i find directly, but through SPARQL [2,3,4,5] queries and not through sequence tagging with BiLSTM, the unique entity label with the most accurate match. I do not try to find ,the entity label with the best match by finding all the entity labels of KG (Wikidata) entities with the almost exact match with the (predicted) entity label (span) and ranking them to find the entity label with the most exact match.

Another difference is the next step, where based to this unique label in order to find the entity label (span) start and end indexes i create the corresponding unique entity span for each question either with the lists containing 0 and 1 or with other ways, such as Sentence Transformers similarity model [21,22] combined with spacy library [25,26]. This step is not part of the algorithm of Mohammed et al [1,91].

2. About the relation classification, instead of BiGRU [94] encoder, i used transformers [1,6,7,15,16,17,85] (more especially BERT [15,17,32,33]) in my neural network for relations and to encode all the questions of the SimpleQuestions training and validation dataset. Also i used a softmax output layer [45,46,47,48] in order to compute probabilities over all relation ids in the relation vocabulary (described in 1.4) in the final state of the aforementioned neural network.

3. About the logical form construction, i do something similar with this step as procedure in the question answering engine creation. Initially from the neural network for relations i get an relation id prediction and also from the corresponding neural network for entity span indexes (described in 1.4) i get an entity span start index and also an entity span end index prediction.

   The entity id for this entity label in the question that it is occurred from these prediction indexes, i get it either from SPARQL [2,3,4,5] or by finding the cosine similarity [23,24] of this label with all the corresponding labels of training and validation dataset of SimpleQuestions [13] either through the similarity model which consists of combination of spacy [25,26] and Sentence Transformer library [21,22], rank them and get the label with the highest similarity score (described in 1.4). This unique combination of the relation id and the entity id the predicted logical form of the question and gives the answer to the question.

   Essentially the main difference between my methodology and methodology of Mohammed et al. [1,91] is in the methodology of Mohammed et al, there are a multitude of top-scoring entities and a multitude of top-scoring entities from relation classification and entity candidate generation steps, where in order to get the predicted logical form all of them are combined and ranked, something that it is not applied in my thesis, because i have only find one relation id and only one entity id in order to generate the predicted logical form of the question, without the need for comparison.

Also another work that is related with my thesis, because in this work the entity span is identified, is the work by Mohammed et al. [91] and Petrochuk and Zettlemoyer [92] with the difference that in this work the entity is disambiguated without using neural networks. Petrochuk and Zettlemoyer [92] employed a BiLSTM-CRF model for span prediction, which can no longer be considered a simple collection of independent classifiers and instead forms a structured prediction model that captures dependencies between different parts of the output formal query (in this case the binary labels for every position).

Additionally Bordes et al. [1, 6, 96] propose a memory network (MemNN)-based solution to SimpleQuestions [13]. They use bag-of-words representations for triples and question and train a model that predicts the right triple by minimizing a margin-based ranking loss as defined in the section above. They compute scores between questions and whole triples, including the triple objects. However, triple objects are answers and thus might not be present in the question, which may affect the performance adversely. The same work introduces the SimpleQuestions dataset [13], consisting of approximately 100,000 question-answer pairs.

Referring in summary to other works on SimpleQuestions dataset [13] typically predict the subject entity and predicate separately,unlike [96], which ranks whole triples. There are works, such as [97] which explore fully character level encoding of questions, entities and predicates, and use an attention-based decoder [86]. The model of He and Golub [97] consists of a character-level RNN-based question encoder and an attention-enhanced RNN decoder coupled with two separate character-level CNN-based entity label and predicate URI encoders. Given that their model works purely on character-level, which results in much longer input sequences, the effect of an attention mechanism could be more pronounced than with word-level models. From the output size perspective, it is reasonable to assume that compared to machine translation an attention mechanism is of lesser importance in Simple QA where only two predictions need to be made per input sequence.

Additionally there are works that explore building question representations on both word- and character level [98].

Also there are works on SimpleQuestions dataset [13] that have the aim to explore relation detection in-depth and propose a hierarchical word-level and symbol-level residual representation [99]. More specifically Yu et al. [99] employ CNNs and propose an attentive pooling approach. Firstly, the question is split into entity mention and relation pattern. Then, the entity mention is encoded using a character-level CNN and matched against a character-level encoded subject label. The relation pattern is encoded using a separate word-level CNN with careful max-pooling and matched against a word-level encoded predicate URI. Essentially in this approach an attention mechanism is used to obtain better matches for predicates instead of implicitly perform segmentation. Both [98] and [100] works improve the performance of [99] by in corporating structural information such as entity type for entity linking. Also Dai et al. [100] investigate a word-level RNN-based approach,in which they encode the question on word-level only and train/use Freebase-specific predicate and entity representation. [101] uses a BiLSTM+CRF tagger to improve the performance of [100]. The tagger is trained to predict which parts of the input correspond to the entity mention. The detected entity span is then used for filtering entity candidates. Also [91] investigates different RNN and CNN-based relation detectors and a BiLSTM and CRF-based entity mention detectors. [92] estimates the upper-bound on accuracy for SimpleQuestions [13], which is less than 83.4% due to unresolvable ambiguities (which is caused by the question lacking information to correctly disambiguate entities). Both [91] and [92] first identify the entity span, similarly to previous works, but disambiguate the entity without using neural networks.

4. DATASET GENERATION

4.1 Summary

Regarding the SimpleQuestions dataset, is a dataset originally designed for Knowledge Base Freebase. However because of the termination of Freebase [38] since August 31, 2016, has been made migration of its content to Wikidata [95]. This migration became through mapping of Freebase triples (subject, (property or relation), object) to knowledge base of Wikidata [14]. More specifically subjects and the objects of the Freebase triples, which consists Freebase topics mapping to Wikidata items [14], using automatically generated mappings and the properties (relations) are mapped using a handmade mapping. When there is no equivalent property in Wikidata [14], but the Freebase inverse property has an equivalent property PXX in Wikidata [14], we map the Freebase property to a fake Wikidata [14] property RXX (R indicating reverse). Note that not every translated triple is required to exist in Wikidata [14]. When migrating the data from Freebase to Wikidata [14] part of the information was lost. We therefore have created two versions, the first containing questions that can be answered in Wikidata [14], the second containing all questions [13].

This new dataset contains 49.202 questions (22.302 of which are answerable over Wikidata [14]). One of the reason of the gap in size between the two datasets is that we have only mapped 404 Freebase properties to Wikidata [95] even if the dataset contains 1.837 properties [13,102].

Also it is worth mentioning that, as in chapter 1.4, when i talk about full or whole SimpleQuestions dataset [13] i mean that the SimpleQuestions dataset [13] contains answerable and non-answerable questions [13] over wikidata [14].

For my thesis on the above in order to renew (regenerate in a way) the already existing SimpleQuestions dataset [13] and more specifically the training and the validation dataset by finding entity labels and creating a new dataset column that contains them, i made the following steps:

1. I import some useful python libraries [20,26,27,28,29,30,31,32,33,34].

2. I mount google drive, because all the files (for example .csv) that i used for the implementation of my thesis are saved on my google drive[27,29].

3. Also i read the training and validation datasets that are in .txt file format [13] (including either all training and validation questions or only answerable over wikidata [14] questions [13])and drop the column of their answers ids by using pandas method read_csv() [36] and pandas method drop() [103] through the following lines of code [27]:

```
train_dataset = pd.read_csv('annotated_wd_data_train.txt', sep="\t",
    names=['entity_id','relation_id', 'answer', 'Question']) #read train
    dataset
validation_dataset = pd.read_csv('annotated_wd_data_valid.txt', sep="\t",
     names=['entity_id','relation_id', 'answer', 'Question'])#read
    validation dataset
train_dataset = train_dataset.drop(labels='answer', axis=1) #drop answer
validation_dataset = validation_dataset.drop(labels='answer', axis=1) #
```

```
    drop answer
```

Listing 4.1: Code in python where the SimpleQuestions training and validation dataset in .txt file that include all questions that are either answerable over wikidata or not is read through pandas method read$_csv()wherethetab("")istheseparator.$

4. I made some preprocessing for each question by creating the function preprocessing, which takes as argument the text (in this case the question) and performs accent removal, as well as punctuation mark removal (period, question mark, exclamation mark), s removal with apostrophe('s) and comma(,) removal. Finally this function, which is shown below, returns the original text with the above mentioned processing [26,27].

```
def Preprocessing(text):
  text = unidecode(text.replace("?", "")
  .replace("'s", "").replace(".", "").replace("!", "")
  .replace(",", ""))
  return text
train_dataset['Question'] = train_dataset['Question'].apply(lambda x:
    Preprocessing(x))
validation_dataset['Question'] = validation_dataset['Question'].apply(
    lambda x:Preprocessing(x))
```

Listing 4.2: Code in python for the creation of the function Preprocessing which makes the question preprocessing. Also the last two lines of code (after the function) are the execution of the function Preprocessing for SimpleQuestions training and validation dataset correspondingly.

5. The next step is to use the SPARQLWrapper from python [104], in order to find the entity labels through SPARQL later. I downloaded using the piece of code below [27]:

```
!pip install sparqlwrapper
from SPARQLWrapper import SPARQLWrapper, JSON, CSV
sparql = SPARQLWrapper("https://query.wikidata.org/sparql",
agent= "WDQS-example Python/%s.%s"
% (sys.version_info[0], sys.version_info[1]))
```

Listing 4.3: Code in python from which the SPARQLWrapper is installed in order to create SPARQL queries.

6. Another step is to find the divisors of the lengths of the full SimpleQuestions training and validation dataset, which are equal to 34373 and 4867 respectively [27]. This is implemented through the following function find_divisors which receives as an argument a number, in our case the length of either the SimpleQuestions training or the SimpleQuestions validation dataset[27]:

```
def find_divisors(a):
    divisors = []
    for i in range(1, a + 1):
        if a % i == 0:
            divisors.append(i)
    return divisors

# Example usage
number_to_check_1 = 34374
number_to_check_2 = 4867
result_1 = find_divisors(number_to_check_1)
result_2 = find_divisors(number_to_check_2)
```

Listing 4.4: Code in python from which the function find_divisors is created in order to. Also the last two lines of code (after the function) are the execution of the function find_divisors for SimpleQuestions training and validation dataset correspondingly.

7. With the help of SPARQL [2,3,4,5], which is a semantic database query language used to schematize, retrieve, manipulate and return linked data from a triplet, i.e. data stored in RDF (Resource Description Framework) format, i found the entity id and the entity label from wikidata for each SimpleQuestions training and validation dataset entity id[13]. This was done via the following piece of code for training and validation dataset respectively [26,27,104,105]:

```
train_entity_names = []
train_entity_ids = []
for i in range(0,len(train_dataset),102):
  str = ""
  for entity_id in train_dataset['entity_id'][i:i+102]:
   str = str + "wd:" + entity_id + " "
  sparql.setQuery("""
  SELECT ?item ?itemLabel
  WHERE
  {
    VALUES ?item {""" + str + """}
    SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
  }
""")
  sparql.setReturnFormat(JSON)
  results = sparql.query().convert()
  train_results_df = pd.json_normalize(results['results']['bindings'])
  for j in train_results_df['item.value']:
      train_entity_ids.append(j)
  for j in train_results_df['itemLabel.value']:
      train_entity_names.append(j)
```

Listing 4.5: Code in python that includes a SPARQL query from which the entity ids and the entity labels are found for the training subset of either the full SimpleQuestions dataset or the corresponding SimpleQuestions subset containing only questions that are answerable over wikidata.

```
validation_entity_names = []
validation_entity_ids = []
for i in range(0,len(validation_dataset),157):
  str = ""
  for entity_id in validation_dataset['entity_id'][i:i+157]:
   str = str + "wd:" + entity_id + " "
  sparql.setQuery("""
  SELECT ?item ?itemLabel
  WHERE
  {
    VALUES ?item {""" + str + """}
    SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
  }
""")
  sparql.setReturnFormat(JSON)
  results = sparql.query().convert()
  validation_results_df = pd.json_normalize(results['results']['bindings'
    ])
  for j in validation_results_df['item.value']:
      validation_entity_ids.append(j)
  for j in validation_results_df['itemLabel.value']:
      validation_entity_names.append(j)
```

Listing 4.6: Code in python that includes a SPARQL query from which the entity ids and the entity labels are found for the validation subset of either the full SimpleQuestions dataset or the corresponding SimpleQuestions subset containing only questions that are answerable over wikidata.

These pieces of code is about the .txt files that contain the full SimpleQuestions dataset[13]. The numbers 102 and 157 are divisors of the full SimpleQuestions training and validation dataset (full dataset means that this dataset contains answerable and non-answerable questions [13] over wikidata [14]), respectively. These numbers are used in order to capture the entity labels for all the questions from SPARQL [2,3,4,5] in the shortest possible time.

About the .txt files that contain only the answerable questions [13] over wikidata [14], i create the same piece of code with the only difference that i have different divisors, due to the different dataset length, in order to conceive the entity labels as fast as possible, i.e 161 for the training dataset length equal to 19481 and 217 for the validation dataset length equal to 2821.

8. Also I put the entity ids and the questions of the initial SimpleQuestions training and validation datasets [13] in lists by using the method to_list() [26,27,38]. These lines of code are the same for both full SimpleQuestions dataset [13] and SimpleQuestions dataset that contains only answerable over wikidata [14] questions [13].

```
train_dataset_questions = train_dataset['Question'].to_list()
validation_dataset_questions = validation_dataset['Question'].to_list()
train_dataset_entity_ids = train_dataset['entity_id'].to_list()
validation_dataset_entity_ids = validation_dataset['entity_id'].to_list()
```

Listing 4.7: Code in Python that creates list for the SimpleQuestions dataset questions and their corresponding entity ids by using the method to_list().

9. From the step of finding entity labels and entity ids from SPARQL, the entity ids that are occurred are expressions that contain an encoded name, for example for the question of the SimpleQuestions (full) training dataset [13] ”who is a musician born in detroit”, the entity id that occurred is https://www.wikidata.org/wiki/Q12439, which is related with Detroit (Q12439, if i search in wikidata [14]).

   From this expression and any other similar expression i want to keep the encoded name, i.e named start from ”Q” and in this case ”Q12439”. So in order to achieve that i create the python function entity_ids[26,27], which uses the python library os [28] and the python function split(), which splits expression such as https://www.wikidata.org/wiki/Q1
   [49]:

```
#Split Spaqrl URI's in order to get the entity ids
import os
def enitity_ids(ids): #get ids from sparql query ids
 entity_ids = []
 for i in range(len(ids)):
   entity_ids.append(ids[i].split(os.path.sep)[-1])
 return entity_ids
train_ids = enitity_ids(train_entity_ids)
validation_ids = enitity_ids(validation_entity_ids)
```

Listing 4.8: Code for function entity_ids in python which splits the expressions of entity ids in order to get the encoded name start from ”Q” which are the entity ids. Also the last two lines of code (after the function) are the execution of the function entity_ids for SimpleQuestions training and validation dataset correspondingly.

This function is the same for both full SimpleQuestions dataset [13] and SimpleQuestions dataset that contains only answerable over wikidata [14] questions [13].

Next, i create the function labels [26,27] from which i confirm the entity id of each question, which occurred from SPARQL [2,3,4,5], through the entity id obtained for each question from the transfer of SimpleQuestions [13] from Freebase [81] to Wikidata [14.95]. This function takes as arguments the list of questions, the list of the entity ids both from the corresponding SimpleQuestions dataset column and from the entity_ids function [26,27] and the list of entity labels occured from SPARQL [2,3,4,5].

Essentially i assigned to each question and entity id its corresponding entity label by searching in the list of training and validation entity ids obtained from SPARQL [2,3,4,5]. If any of these entity ids exist in the SimpleQuestions [13] training and validation entity labels lists [26,27,38] that are created before and matching the entity label and thus a new column with entity labels for the training and the validation dataset [26,27]:

```python
def labels(questions, entity_id_dataset, entity_ids, entity_names):#entity_ids and dataset must have the same length
#In this step we find the entity ids from the initial dataset to the list occured from Sparql Query
#for each of the train and validation dataset.
#This will become the train or validation data new column, containing entities' labels

  label_df_column = []
  for entity_id in entity_id_dataset:
   j=0
   for ent in entity_ids: #find its id on list
    if entity_id == ent:
      label_df_column.append(entity_names[j]) #and find corresponding label
      break
    j += 1
  return label_df_column
train_dataset['entity_label'] = labels(train_dataset_questions, train_dataset_entity_ids, train_ids, train_entity_names)
validation_dataset['entity_label'] = labels(validation_dataset_questions, validation_dataset_entity_ids ,validation_ids, validation_entity_names)
```

Listing 4.9: Code in Python for function labels that has as aim to assign to each question and entity id to their corresponding entity label by searching in the list of entity ids obtained from SPARQL. The last two lines of code (after the function) are the execution of the function labels for SimpleQuestions training and validation dataset correspondingly.

10. Thus the preprocessing of the entity labels column, that are found through labels function [26,27], through function Preprocessing [26,27]. Next the index of the columns (entity id, relation id, questions, entity label) of the initial SimpleQuestions dataset (either full or the dataset containing only questions that are answerable over wikidata [13]) must be reindexed in order to include the entity labels as a new column in the initial SimpleQuestions dataset [13]. I achieved that by using the reindex() method from pandas [106].

```python
train_dataset['entity_label'] = train_dataset['entity_label'].apply(lambda x:Preprocessing(x))
validation_dataset['entity_label'] = validation_dataset['entity_label'].apply(lambda x:Preprocessing(x))
```

Listing 4.10: Code in Python for function Preprocessing that is applied for the entity labels. The last two lines of code (after the function) are the execution of the function Preprocessing for SimpleQuestions training and validation dataset correspondingly.

```
new_columns = ["entity_id","entity_label","relation_id","Question"]
train_dataset = train_dataset.reindex(columns=new_columns)
validation_dataset = validation_dataset.reindex(columns=new_columns)
```

Listing 4.11: Code in Python from which the column of the entity labels for SimpleQuestions training and validation dataset correspondingly.

Finally the SimpleQuestions training and validation dataset [13] is exporting to a .csv file by importing csv library [28] and using the to_csv() pandas method [107]. The names that the full SimpleQuestions training and validation datasets [13] saved are ”drive/MyDrive/train_dataset.csv” and ”drive/MyDrive/valid_dataset.csv”. Respectively the names that the SimpleQuestions training and validation datasets [13] including only answerable over wikidata [14] questions saved by using the same piece of code are ”drive/MyDrive/train_answerable_questions_dataset.csv” and ”drive/MyDrive/valid_ar

```
import csv
# Specify the file name
train_csv_name = "drive/MyDrive/train_dataset.csv"
# Writing to CSV file
train_dataset.to_csv(train_csv_name, index=False)

valid_csv_name = "drive/MyDrive/valid_dataset.csv"
validation_dataset.to_csv(valid_csv_name, index=False)
```

Listing 4.12: Code in Python for function labels that has as aim to assign to each question and entity id to their corresponding entity label by searching in the list of entity ids obtained from SPARQL. These lines of code (after the function) are the execution of the function labels for the full SimpleQuestions training and validation dataset correspondingly.

The function labels is the same for both full SimpleQuestions dataset [13] and SimpleQuestions dataset that contains only answerable over wikidata [14] questions [13].

Additionally all these steps are followed in order to find the entity labels of the test dataset of SimpleQuestions dataset [13] both for the case that it is full and for the case that contains only questions that are answerable over wikidata [14]. Respectively two similar test datasets are created.

Finally the first objective of this thesis, i.e the creation of a simple SPARQL [2,3,4,5] query that uses the entity id of each one of the questions of the training and the validation dataset of the SimpleQuestions dataset [13] in order to find its entity label, it is completely achieved.

For any step, that the corresponding piece of code in python is not referred in this section it could be found on chapter 1.4 in the Section 1-Introduction and in [27].

5. THE SENTENCE TRANSFORMER QUESTION ANSWERING MODEL

5.1 Summary

Another very important part of my thesis is the creation of the Question Answering Engine. I have created three types of Question Answering Engines (QA Engine):

1. QA engine by using the Sentence Transformers library [21,22] combined with the cosine similarity metric [23,24].
2. QA engine by using only the jaccard similarity metric [23,24].
3. QA engine by using only the spacy library [25,26].

Before the creation of the QA engines, i made the following:

1. I import some useful python libraries for this thesis [20,26,27,28,29,30,31,32,33,34] The corresponding piece of code in python is found on Section 1.4-Methodology.
2. I mount google drive through the code below, because all the files (for example .csv) that i used for the implementation of my thesis are saved on my google drive[27,29]. The corresponding piece of code in python is found on Section 1.4-Methodology.
3. Definition through Google colab of the device that is going to be used for all this thesis. The corresponding piece of code in python, which is found on Section 1.4-Methodology, checks the device that i used for the implementation of my thesis, that is determined from google colab, as i mentioned earlier. Also through the following code this device is printed[27,35].
4. Also i read the created datasets (including either all training, validation and test questions or only answerable over wikidata [14] questions [13]) as .csv file format by using pandas method read_csv() [36] The corresponding piece of code in python is found on Section 1.4-Methodology. An example for the case of SimpleQuestions dataset which contains only answerable over wikidata [14] questions [13]:

```python
train_answerable_dataset = pd.read_csv("drive/MyDrive/
    train_answerable_questions_dataset.csv", sep = ',')
validation_answerable_dataset = pd.read_csv("drive/MyDrive/
    valid_answerable_questions_dataset.csv", sep = ',')
test_answerable_dataset = pd.read_csv("drive/MyDrive/
    test_answerable_dataset.csv", sep = ',')
print(train_answerable_dataset)
print(validation_answerable_dataset)
print(test_answerable_dataset)
```

Listing 5.1: Code in Python where the SimpleQuestions training and validation dataset that include only questions that are answerable over wikidata is read through pandas method read$_csv()wherethecomma('$

5. I create an .txt file named ”Questions_testing_with_answers” [27] that contains 60 questions with their corresponding entity id entity label relation id and answer id. In order to create this .txt file i use questions from SimpleQuestions dataset [13] either

the questions themselves or other questions similar with them, with regard to the wording. Also i create some questions on my own, for which of them i search for answer ids,answer label, relation id, entity id on wikidata [14].

This .txt file is going to be use for all question answering engines in order to see which QA engine answers either correctly or generally giving an answer to the most questions out of these 60 questions. Thus i will have a clearer view about which QA engine works better. This file is read as .csv file through pandas method read_csv() [36].

```python
testing_questions_list = pd.read_csv("drive/MyDrive/Questions_testing_with_answers.txt", sep = ',', names=['Question','entity_id', 'relation_id','answer_id', 'answer_label'])
```

Listing 5.2: Code in Python where the SimpleQuestions training and validation dataset that include only questions that are answerable over wikidata is read through pandas method read$_{c}sv()wherethecomma(”$

6. Additionally i made a check in the recently stored training validation and test datasets (described in Chapter 4) and also in .txt file ”Questions_testing_with_answers” , for any lines, where each line consists of the queries, the entity labels, the relation ids and the entity ids, which have a null value, via the dropna() [27,37] function of the python pandas library. If any such row is found, it is removed from any dataset that has it either training or validation, and the removal is done in the dataset itself, without creating a copy of it [37]. An example for the case of SimpleQuestions dataset which contains only answerable over wikidata [14] questions [13]. The corresponding piece of code is on Section 1.4-Methodology and below is shown an example of this piece of code for the .txt file ”Questions_testing_with_answers” [27]

```python
testing_questions_list.dropna(inplace = True)
```

Listing 5.3: Code in Python where is checked if the SimpleQuestions .txt file ”Questions$_{t}esting_{w}ith_{a}nswers”hasatleastonelinewithnullvaluethroughpandasfunctiondropna().$

7. I create lists through the pandas method .to_list() [27,38] for the entity labels found from SPARQL [2,3,4,5], the entity ids, the relation ids for all the questions of the training and validation dataset of the SimpleQuestions dataset either this dataset includes all questions (answerable, non-answerable over wikidata [13]) or only answerable questions over wikidata [13]. This step is very useful, because this step could make the implementation of the next steps of the methodology of this thesis much more easier. This step is also applied for the ”Questions_testing_with_answers” .txt file and the corresponding code for the SimpleQuestions datasets [13] is on Section 1.4-Methodology.

```python
train_answerable_entity_labels = train_answerable_dataset['entity_label'].to_list()
validation_answerable_entity_labels = validation_answerable_dataset['entity_label'].to_list()
test_answerable_entity_labels = test_answerable_dataset['entity_label'].to_list()
train_answerable_entityids = train_answerable_dataset['entity_id'].to_list()
validation_answerable_entityids = validation_answerable_dataset['entity_id'].to_list()
test_answerable_entityids = test_answerable_dataset['entity_id'].to_list()
train_answerable_relationids = train_answerable_dataset['relation_id'].to_list()
```

```python
validation_answerable_relationids = validation_answerable_dataset['
    relation_id'].to_list()
test_answerable_relationids = test_answerable_dataset['relation_id'].
    to_list()
train_answerable_questions = train_answerable_dataset['Question'].to_list
    ()
validation_answerable_questions = validation_answerable_dataset['Question
    '].to_list()
test_answerable_questions = test_answerable_dataset['Question'].to_list()
```

Listing 5.4: Code in Python from which are created lists for all the questions

8. The preprocessing for each question is made by creating the function preprocessing, which takes as argument the text (in this case the question) and performs accent removal, as well as punctuation mark removal (period, question mark, exclamation mark), s removal with apostrophe('s) and comma(,) removal.Finally this function, which is shown in Section 1.4-Methodology, returns the original text with the above mentioned processing [26,27].

9. Another useful step is to find the biggest question length from all the questions of SimpleQuestions dataset [13]. This is achieved by making list concatenation by using "+" operator [39] through the following lines of code [27]:

```python
#Below the max questions length is found for all questions.
quests = train_questions + validation_questions + test_questions
print(len(quests), len(train_questions), len(validation_questions))
# Printing concatenated list
q_lengths = []
for s in range(len(quests)): #padding
    q_lengths.append(len(quests[s].split()))
max_question_length = max(q_lengths)
print(max_question_length)#Max question length
```

Listing 5.5: Code in Python where is found the biggest question length from all the questions of SimpleQuestions dataset by using list concatenation through "+" operator.

10. Also i create a jaccard similarity metric [23,24] custom function. The corresponding lines of code are on Section 1.4-Methodology.

11. I create the function relation_vocabulary [26,27], which creates and returns the relation vocabulary which is the list of unique relation ids present in the training and validation dataset of SimpleQuestions dataset [13]. The lines of code that describe this function are in section 1.4-Methodology.

12. Additionally i create the function find_relation_index [26,27], which returns for each relation id its position (index) in the relation vocabulary.

13. Also in order to make predictions for the relation index, entity span start index, entity span end index predictions i construct neural networks where the corresponding lines of code that describe them are in section 1.4-Methodology.

14. In order to create SPARQL queries that are needed for this work i download the SPARQLWrapper from python [104], with the piece of code from which the SPARQLWrapper [104] is downloaded could be found on Section 4-Dataset Generation.

More analytically i create the Question Answering Engine through the following steps:

1. I create the get_entity_id function that takes an entity label as an argument, which involves creating a SPARQL query [2,3,4,5] that, given an entity label, finds and extracts its corresponding entity id [26,103,104,105].

```python
def get_entity_id(entity_label):
 sparql = SPARQLWrapper('https://query.wikidata.org/sparql'
 , agent = "WDQS-example Python/%s.%s"
 % (sys.version_info[0], sys.version_info[1]))
 sparql.setQuery("""
 SELECT distinct ?item ?itemLabel ?itemDescription
 WHERE
  {
    ?item ?label "%s"@en.
    ?article schema:about ?item .
    ?article schema:inLanguage "en" .
    ?article schema:isPartOf <https://en.wikipedia.org/>.
    SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
  }"""  % (entity_label.title()))
  sparql.setReturnFormat(JSON)
  results = sparql.query().convert()
  entityids = []
  for result in results["results"]["bindings"]:
    entityids.append(result["item"]["value"].split(sep='/')[-1])
  return entityids
```

Listing 5.6: Code in Python for function get_entity_id which involves creating a SPARQL query that finds and extracts its corresponding entity id through an entity label

The next step is the creation of two dictionaries, where one named Endict will have as keys the entity labels of the training validation and test dataset and as values the corresponding entity ids of the training, validation and test dataset [26,27].

```python
# Create a dictionary with entity labels as keys and entity_ids as values
entity_ids = train_entityids + validation_entityids + test_entityids
entity_labels = train_entity_labels + validation_entity_labels +
    test_entity_labels
Endict = {}
for h in range(len(entity_ids)):
  label = entity_labels[h]
  id = entity_ids[h]
  Endict[label] = id
entities = list(Endict.keys())
quests = {}
```

Listing 5.7: Code in Python for the creation of dictionary named Endict which have as keys the entity labels of the training validation and test dataset and as values the corresponding entity ids of the whole SimpleQuestions dataset.

2. The other dictionary named quests, which is needed when we create the QA engine by using Sentence Transformers library [21,22] and cosine similarity metric [23,24], has as keys the entity labels of the training, validation and test dataset and as values the corresponding encoding vectors from the Sentence Transformer [21,22] for each entity label of the training, validation and test dataset [26,27].

```python
# Create a dictionary with Sentence Transfomer embeddings for each entity
    label.
for entity_label, id in Endict.items():
    quest = similarity_model.encode(entity_label)
    quests[entity_label] = quest
```

Listing 5.8: Code in Python for the creation of the dictionary named quests which has as keys the entity labels of the full SimpleQuestions dataset and as values the corresponding encoding vectors from the Sentence Transformer for each entity label of the full SimpleQuestions dataset.

The step of creating the above two dictionaries is extremely helpful when in the QA engine are entered questions of the SimpleQuestions dataset [13]

3. I create the function question_encoding that takes as arguments the question and the corresponding tokenizer, in this case BertTokenizerFast [34]. Finally the function question_encoding encode the given question through the command tokenizer.encode_plus() [52]. The ids and masks resulting from the corresponding encoding are returned.

```python
def question_encoding(question,tokenizer):
    question = Preprocessing(question)
    ids = []
    mask = []
    # Encode the question
    encoding = tokenizer.encode_plus(
        text = question,
        return_attention_mask=True,
        add_special_tokens=False
    )

    ids.append(encoding['input_ids'])
    mask.append(encoding['attention_mask'])

    return torch.tensor(ids), torch.tensor(mask)
```

Listing 5.9: Code in Python for the function question_encoding where either a question from SimpleQuestions dataset or a question given by a user is encoded through command tokenizer.encode_plus().

4. Through the function question_encoding, the question preprocessing is done by calling the function preprocessing, which takes as argument the text (in this case the question) and performs accent removal, as well as punctuation mark removal (period, question mark, exclamation mark), s removal with apostrophe('s) and comma(,) removal. Finally this function returns the original text with the above mentioned processing [27](The python code of this function could be found on the Section 4:Dataset Generation.)

5. Creation of the function find_closest_match [26,27] that takes as arguments the entity label, the dictionary Endict with keys the entity labels and values the entity ids, a list of all keys of Endict, the dictionary entity_quests with keys the entity labels and values the encodings obtained for each entity label through the pretrained similarity model of Sentence Transformer [21,22]. Through this function the entity id for the entity of a question is found by spacy library [25,26] between the entity label of a user question and the list of Endict keys, i.e the entity labels of the SimpleQuestions training and validation dataset [13] and the more similar entity label is returned, if the entity id for the entity of a query cannot be found through the get_entity_id function. In this case the entity id of the entity that has the best cosine similarity score with the entity for which we are looking for the entity id is returned, if available.

```python
#Find the best match from the list of train dataset entity labels
def find_closest_match(entity_label, Endict, entities):
  best_match = difflib.get_close_matches(entity_label, entities, n=1)
  if best_match != []:
    return Endict[best_match[0]]
    print("Cosine similarity metric is used!")
    return entityid
  else:
    return None
```

Listing 5.10: Code in Python for the creation of the function find_closest_match which finds the most similar entity label of the SimpleQuestions dataset to the entity label of a question given by a user through spacy library in the corresponding QA engine that include spacy library.

The same function, if i want to use only the cosine similarity metric [23,24], takes the entity label, the dictionary Endict with keys the entity labels and values the entity ids, a list of all keys of Endict, the dictionary quests with keys the entity labels and values the encodings obtained for each entity label through the pretrained similarity model of Sentence Transformer [21,22].

Through this function the entity id of a question is found by the evaluation of the cosine similarity metric score between the pretrained similarity model encoding of this entity label for a question and each value of the dictionary quests. The entity label with the best cosine similarity metric score is kept and through Endict dictionary the corresponding relation id is found.

```
def find_closest_match(entity_label, Endict, entities, quests,
    similarity_model):
  ent = similarity_model.encode(entity_label)
  similarity_scores = {}
  # Find similarity scores between entity and each entity in dictionary
  for ent_label, Ent in quests.items():
    similarity_scores[ent_label] = cosine_similarity(ent,Ent)
  Q = dict(sorted(similarity_scores.items(),
  key = lambda x: x[1], reverse = True))
  EntitiesList = [key for key in Q]
  if EntitiesList != []:
    entityid = Endict[EntitiesList[0]]
    print("Cosine similarity metric is used!")
    return entityid
  else:
    return None
```

Listing 5.11: Code in Python for the creation of the function find_closest_match which finds the most similar entity label of the SimpleQuestions dataset to the entity label of a question given by a user through the corresponding QA engine which include the combination of Sentence Transformers library and cosine similarity metric.

Where cosine similarity metric [23,24] function is:

```
def cosine_similarity(encoding1, encoding2):
  return np.dot(encoding1, encoding2) / (np.linalg.norm(encoding1) * np.
    linalg.norm(encoding2)) if np.linalg.norm(encoding1) * np.linalg.norm
    (encoding2) != 0 else 0
```

Listing 5.12: Code in Python for the creation of the function cosine_similarity which calcuates the cosine similarity metric between two encodings. In this case the encodings are the Sentence Transformers library encodings of the entity label of the question given by a user through the corresponding QA engine and he Sentence Transformers library encodings of an entity label of the SimpleQuestions dataset.

If i want to use only the jaccard similarity metric instead of cosine similarity metric [23,24], the find_closest_match [26,27] remains the same compared with the corresponding function [26,27] including cosine similarity metric [23,24] with the differences that a different similarity metric is used and Sentence Transformers library [21,22] is not used:

```
def find_closest_match(entity_label, Endict, entities, quests,
    similarity_model):
```

```python
  ent = similarity_model.encode(entity_label)
  similarity_scores = {}
  # Find similarity scores between entity and each entity in dictionary
  for ent_label, Ent in quests.items():
    similarity_scores[ent_label] = jaccard_similarity(ent,Ent)
  Q = dict(sorted(similarity_scores.items(),
  key = lambda x: x[1], reverse = True))
  EntitiesList = [key for key in Q]
  if EntitiesList != []:
    entityid = Endict[EntitiesList[0]]
    print("Jaccard similarity metric is used!")
    return entityid
  else:
    return None
```

Listing 5.13: Code in Python for the creation of the function find_closest_match which finds the most similar entity label of the SimpleQuestions dataset to the entity label of a question given by a user through the corresponding QA engine which include the combination of Sentence Transformers library and jaccard similarity metric.

Creation of functions to predict entity span start index, entity span end index and relation index. The function for predicting the entity span start index and entity span end index takes as arguments the ids and the mask resulting from the encoding of the question via the question_encoding function, the aforementioned dictionaries Endict and entity_quests, the pretrained model of the Sentence Transformer similarity model [21,22], the entity_model_path, i.e. the path where the best parameters for the best predictive model of entity span start index and entity span end index are stored, the activation function for the best predictive model of entity start index and entity end index, the dropout probability, dropout_prob and the SEED number which helps to have the same and reproducible answers. The predictions for entity span start index and entity span end index, are returned by this function. Correspondingly the function for the prediction of relation index, takes as arguments the ids and mask resulting from the encoding of the question through the question_encoding function, the length of the relation vocabulary, where its creation is described in 1.4 chapter, the relation_model_path, i.e. the path where the best parameters for the best prediction model of relation index are stored, the activation function of the best prediction model of relation index prediction,the dropout probability, dropout_prob and the SEED number. The prediction for the relation index gives through the relation vocabulary which relation id is in the question. This relation index is returned.

```python
def entity_prediction(ids, mask, Endict, entities,
tokenizer, quests, similarity_model,
max_question_length, activation,
entity_model_path, dropout_prob, SEED):
    SEED = 42
    set_seed(SEED)
    entity_model = BertEntitySpanClassifier(activation, dropout_prob,
   max_question_length).to(device)
    entity_model.load_state_dict(torch.load(entity_model_path))
    entity_model.eval()
    y_start, y_end = entity_model(ids, mask)
    y_pred_start = torch.argmax(y_start, dim=1)
    #y_end = y_end * start_mask
    y_pred_end = torch.argmax(y_end, dim=1)
    entity = tokenizer.convert_tokens_to_string(tokenizer.convert_ids
    _to_tokens(ids[0][y_pred_start.item():y_pred_end.item()+1]))
    print(entity)
    if entity in Endict:
        return Endict[entity]
    else:
        entityid = get_entity_id(entity)

        if entityid != []:
            return entityid[0]
    return find_closest_match(entity, Endict, entities,
    quests, similarity_model)
```

Listing 5.14: Code in Python for the creation of the function entity_prediction which has as aim to predict the entity span start index and entity span end index which gives the entity label that exists from the wording of whichever question.

```
def relation_prediction(ids, mask, rel_vocab,
rel_vocab_len, activation_relation,
dropout_prob, relation_model_path,
SEED):
    set_seed(SEED)
    relation_model = BertRelationsClassifier(activation_relation,
    dropout_prob, rel_vocab_len).to(device)
    relation_model.load_state_dict(torch.load(relation_model_path))
    relation_model.eval()
    rel = relation_model(ids, mask).to(device)
    rel_pred = torch.argmax(rel, dim=1)
    relation = rel_vocab[rel_pred]
    return relation
```

Listing 5.15: Code in Python for the creation of the function relation_prediction which has as aim to predict the relation indexwhich gives the relation id that exists from the wording of whichever question.

Also in the case that for a QA engine i do not use the Sentence Transformers similarity model [21,22], i.e for the QA engine include either only spacy library [25,26] or jaccard similarity metric [23,24], the function entity_prediction does not have the dictionary quests and the Sentence Transformers similarity model [21,22] as arguments.

```
def entity_prediction(ids, mask, Endict, entities, tokenizer,
   max_question_length, activation, entity_model_path, dropout_prob,
   SEED):
    #Loading the best model for entity span prediction
    set_seed(SEED)
    entity_model = BertEntitySpanClassifier(activation, dropout_prob,
   max_question_length).to(device)
    entity_model.load_state_dict(torch.load(entity_model_path))
    entity_model.eval()
    y_start, y_end = entity_model(ids, mask)
    y_pred_start = torch.argmax(y_start, dim=1)
    y_pred_end = torch.argmax(y_end, dim=1)
    entity = tokenizer.convert_tokens_to_string(tokenizer.
   convert_ids_to_tokens(ids[0][y_pred_start.item():y_pred_end.item()
   +1]))
    print(entity)
        # Initially check if it's in its current form (without accents
   etc)
    if entity in Endict:
        return Endict[entity]
    else:
        entityid = get_entity_id(entity)

        if entityid != []:
```

```python
            return entityid[0]
    # Fallback to the most similar from the dictionary
    return find_closest_match(entity, Endict, entities)
```

Listing 5.16: Code in Python for the creation of the function entity_prediction which has as aim to predict the entity span start index and entity span end index which gives the entity label that exists from the wording of whichever question.

To extract the final answer or answers for each question, i create the function named answer which takes as arguments the question itself, the dictionary Endict the dictionary entity_quests if the Sentence Transformer similarity model is used, the tokenizer BertTokenizerFast [60], the pretrained similarity model of the Sentence Transformer [21,22], if the cosine similarity metric [23,24] is used, the entity_model_path, i.e. the path where the best parameters for the each one of the models created (described in 1.4 chapter) of entity start index and entity end index are stored, the relation vocabulary, the relation_model_path, i.e. the path where the best parameters for the best prediction model of the relation index are stored, the activation function of the best prediction model (described in 1.4 chapter) of the relation index, the activation functions of the best prediction models of the entity start index and entity end index (described in 1.4 chapter) and the dropout probability, dropout_prob. Also the answer function calls the aforementioned function question_encoding in order to encode the question and get the ids and the mask that are needed for the functions entity_prediction and relation_prediction, which are also called from the function answer. This function returns a boolean value which shows if a question has an totally correct answer and also a boolean value which shows if a question has answer(s) either correct or incorrect or both of them, regardless if these answers include the expected answer of a question. Additionally the answers of this question, with the limitation of maximum 15 answers, are printed [26,57,58,59].

More specifically if the QA engine contains only spacy library [25,26] the answer is totally correct if at least one of the relation id and entity id has predicted correctly and the list of predicted answers has one common element with the real answer according to the Questions_testing_with_answers.txt file. Also when at least one of the relation id and entity id has predicted correctly, while the question to be answered with either only correct or only incorrect answers or both of them, but the expected answer is not one of these answers, then the questions is assumed as answered. For these three QA engines that i create a question is not answered correctly if both of relation id and entity id is not predicted correctly. When i refer to the correct prediction of entity id and relation id, i mean the predictions to give the expected values for entity id and relation id, according to the ”Questions_testing_with_answers” .txt file. Also For these three QA engines that i create a question has no answer if for the combination of the predicted relation id and entity the corresponding SPARQL query [2,3,4,5] gives an empty list with no answers. So, the function called answer for this QA engine which includes only spacy library [25,26] has the following form:

```python
def answer(question, Endict, entities, rel_vocab, rel_vocab_len,
    tokenizer, max_question_length, activation_relation,
    activation_entity, relation_model_path, entity_model_path,
    dropout_prob, SEED, real_answer, real_entity_id, real_relation_id):
    #Boolean variable that denotes if a question is answered
    question_answered = 0
    question_correct_answered = 0
    #Encode question
    ids, mask = question_encoding(question, tokenizer)
    ids = ids.to(device)
    mask = mask.to(device)
    # Loading the best model for relation prediction
    relation = relation_prediction(ids, mask, rel_vocab, rel_vocab_len, activation_relation, dropout_prob, relation_model_path, SEED)
    print(relation)
    #Predict Entity
    entity = entity_prediction(ids, mask, Endict, entities, tokenizer, max_question_length, activation_entity, entity_model_path, dropout_prob, SEED)
    print(entity)
    if entity == None:
        print("Sorry, no answer available")
    else:
        answers = get_answer(entity, relation)
        if answers == []:
          print("Sorry, no answer available")
        else:
          model_answers = []
          for iter, answer in enumerate(answers):
             # To only print top 15 results
             if iter == 15:
                break
             model_answers.append(answer["itemLabel"]["value"])
          # Print the answer
          for iter, answer in enumerate(model_answers):
              print('answer: ', answer)
          ans = difflib.get_close_matches(real_answer, model_answers, n=1)
          if entity != real_entity_id and relation != real_relation_id:
            print("The question is not answered correctly!")
          elif entity == real_entity_id and relation == real_relation_id:
            if ans != []:
             question_correct_answered = 1
             print("The question is answered correctly!")
            else:
             question_answered = 1
             print("The question is answered with either correct answer(s) or not correct answer(s) or both of them, but with not the expected answer!")
          else:
            if ans != []:
             question_correct_answered = 1
             print("The question is answered correctly, although either entity id or relation id is not predicted correctly!")
            else:
             question_answered = 1
             print("The question is answered with either correct answer(s) or not correct answer(s) or both of them, but with not the expected answer!")
    return question_correct_answered, question_answered
```

Listing 5.17: Code in Python for the creation of the function answer for the QA engine that includes only spacy library.

If the QA engine contains either the combination of Sentence Transfromers library [21,22] with the cosine similarity metric [23,24] or only jaccard similarity metric [23,24] the answer is totally correct if at least one of the relation id and entity id has predicted correctly and either for a list with only one predicted answer, this answer has cosine similarity metric score with the real answer greater than 80% or also for a list with more than one predicted answers the maximum cosine similarity metric score between the real answer of a question according to the Questions_testing_with_answers.txt file and one of more answers of the predicted relation id and entity id occurred from SPARQL [2,3,4,5] for the same question is greater than 80% and the corresponding maximum cosine similarity metric score between the real answer and one of more predicted answers are lower than 5

For any other cosine similarity metric scores when at least one of the relation id and entity id has predicted correctly, while the question to be answered with either only correct or only incorrect answers or both of them, regardless if the expected answer is one of these answers, then the questions is assumed as answered. The function called answer for the QA engine including Sentence Transformers [21,22] and cosine similarity metric [23,24]) has the following form:

```python
def answer(question, Endict, entities, rel_vocab, rel_vocab_len,
    tokenizer, quests, similarity_model, max_question_length,
    activation_relation, activation_entity, relation_model_path,
    entity_model_path, dropout_prob, SEED, real_answer, real_entity_id,
    real_relation_id):
     #Boolean variables that denotes if a question is answered
     question_answered = 0
     question_correct_answered = 0
     #Encode question
     ids, mask = question_encoding(question, tokenizer)
     ids = ids.to(device)
     mask = mask.to(device)
     # Loading the best model for relation prediction and predict relation
     id
     relation = relation_prediction(ids, mask, rel_vocab, rel_vocab_len,
    activation_relation, dropout_prob, relation_model_path, SEED)
     print(relation)
     # Loading the best model for entity span prediction and predict
    entity id
     entity = entity_prediction(ids, mask, Endict, entities, tokenizer,
    quests, similarity_model, max_question_length, activation_entity,
    entity_model_path, dropout_prob, SEED)
     print(entity)
     real_ans = similarity_model.encode(real_answer)
     if entity == None:
         print("Sorry, no answer available")
     else:
         answers = get_answer(entity, relation)
         if answers == []:
           print("Sorry, no answer available")
         else:
           model_answers = []
           for iter, answer in enumerate(answers):
              if iter == 15:
                 break
              model_answers.append(answer["itemLabel"]["value"])
           answer_similarity_scores = {}
           for iter, answer in enumerate(model_answers):
               ans = similarity_model.encode(answer)
               answer_similarity_scores[answer] = cosine_similarity(
    real_ans,ans)
           Answers_sorted = dict(sorted(answer_similarity_scores.items(),
    key = lambda x: x[1], reverse = True))
           best_similarity_scores = list(Answers_sorted.values())
           best_answers_sorted = list(Answers_sorted.keys())
           # Print the answers occured from best model parameters
           # and the similarity scores of each answer with the expected
       answer
           for sorted_answer in range(len(best_answers_sorted)):
             print('answer: ', best_answers_sorted[sorted_answer],',','
    similarity score with the expected answer: {:.2f}'.format(
    best_similarity_scores[sorted_answer]))
           if best_similarity_scores != []:
             min_similarity_answer_score = min(best_similarity_scores)
             max_similarity_answer_score = max(best_similarity_scores)
           if entity != real_entity_id and relation != real_relation_id:
              print("The question is not answered correctly!")
           elif entity == real_entity_id and relation == real_relation_id:
             if (max_similarity_answer_score > 0.8 and len(
    best_answers_sorted) == 1) or (max_similarity_answer_score > 0.8 and
    min_similarity_answer_score > 0.05 and len(best_answers_sorted) > 1):
              question_correct_answered = 1
              print("The question is answered correctly!")
             else:
              question_answered = 1
              print("The question is answered with either only correct
    answer(s), regardless if the expected answer is one of these correct
    answer(s) or not, \nwhich are different but even a little bit similar
     with the expected answer or only incorrect answer(s) or with both of
     them!")
           else:
             if (max_similarity_answer_score > 0.8 and len(
    best_answers_sorted) == 1) or (max_similarity_answer_score > 0.8 and
    min_similarity_answer_score > 0.05 and len(best_answers_sorted) > 1):
              question_correct_answered = 1
              print("The question is answered correctly, although either
    entity id or relation id is not predicted correctly!")
             else:
              question_answered = 1
              print("The question is answered with either only correct
    answer(s), regardless if the expected answer is one of these correct
    answer(s) or not, \nwhich are different but even a little bit similar
     with the expected answer or only incorrect answer(s) or with both of
     them!")
     return question_correct_answered, question_answered
```

Listing 5.18: Code in Python for the creation of the function answer for the QA engine that includes only spacy library.

The function called answer for the QA engine including only jaccard similarity metric [23,24]) has the following form:

```
def answer(question, Endict, entities, rel_vocab, rel_vocab_len,
    tokenizer, max_question_length, activation_relation,
    activation_entity, relation_model_path, entity_model_path,
    dropout_prob, SEED, real_answer, real_entity_id, real_relation_id):
     #Boolean variables that denotes if a question is answered
     question_answered = 0
     question_correct_answered = 0
     #Encode question
     ids, mask = question_encoding(question, tokenizer)
     ids = ids.to(device)
     mask = mask.to(device)
     # Loading the best model for relation prediction and predict relation
      id
     relation = relation_prediction(ids, mask, rel_vocab, rel_vocab_len,
    activation_relation, dropout_prob, relation_model_path, SEED)
     print(relation)
```

```
# Loading the best model for entity span prediction and predict entity id
entity = entity_prediction(ids, mask, Endict, entities, tokenizer, max_question_length, activation_entity, entity_model_path, dropout_prob, SEED)
print(entity)
if entity == None:
    print("Sorry, no answer available")
else:
    answers = get_answer(entity, relation)
    if answers == []:
      print("Sorry, no answer available")
    else:
      model_answers = []
      for iter, answer in enumerate(answers):
         if iter == 15:
            break
         model_answers.append(answer["itemLabel"]["value"])
      answer_similarity_scores = {}
      for iter, answer in enumerate(model_answers):
          answer_similarity_scores[answer] = jaccard_similarity(real_answer.split(),answer.split())
      Answers_sorted = dict(sorted(answer_similarity_scores.items(), key = lambda x: x[1], reverse = True))
      best_similarity_scores = list(Answers_sorted.values())
      best_answers_sorted = list(Answers_sorted.keys())
      # Print the answers occured from best model parameters
      # and the similarity scores of each answer with the expected answer
      for sorted_answer in range(len(best_answers_sorted)):
        print('answer: ', best_answers_sorted[sorted_answer],',',' similarity score with the expected answer: {:.2f}'.format(best_similarity_scores[sorted_answer]))
      if best_similarity_scores != []:
        min_similarity_answer_score = min(best_similarity_scores)
        max_similarity_answer_score = max(best_similarity_scores)
      if entity != real_entity_id and relation != real_relation_id:
         print("The question is not answered correctly!")
      elif entity == real_entity_id and relation == real_relation_id:
        if (max_similarity_answer_score > 0.8 and len(best_answers_sorted) == 1) or (max_similarity_answer_score > 0.8 and min_similarity_answer_score > 0.05 and len(best_answers_sorted) > 1):
         question_correct_answered = 1
         print("The question is answered correctly!")
        else:
         question_answered = 1
         print("The question is answered with either only correct answer(s), regardless if the expected answer is one of these correct answer(s) or not, \nwhich are different but not (or maybe a little bit) similar with the expected answer or only incorrect answer(s) or with both of them!")
      else:
        if (max_similarity_answer_score > 0.8 and len(best_answers_sorted) == 1) or (max_similarity_answer_score > 0.8 and min_similarity_answer_score > 0.05 and len(best_answers_sorted) > 1):
         question_correct_answered = 1
         print("The question is answered correctly, although either entity id or relation id is not predicted correctly!")
        else:
         question_answered = 1
```

```
            print("The question is answered with either only correct
   answer(s), regardless if the expected answer is one of these correct
   answer(s) or not, \nwhich are different but not (or maybe a little
   bit) similar with the expected answer or only incorrect answer(s) or
   with both of them!")
    return question_correct_answered, question_answered
```

Listing 5.19: Code in Python for the creation of the function answer for the QA engine that includes only spacy library.

To get the answer (or answers) the function answer calls the get_answer function which takes as arguments the entity and relation obtained by calling the entity_prediction and relation_prediction functions, through which the best parameters of the best prediction models for relation index, entity start index and entity end index are loaded. The get_answer function includes the corresponding SPARQL query [26,27,103,105] to find the answers, depending on whether the relation of the given question starts from R or from P. The answers of the question are returned.

```
def get_answer(entityid, relation):
 sparql = SPARQLWrapper('https://query.wikidata.org/sparql',
 agent = "WDQS-example Python/%s.%s"
 % (sys.version_info[0], sys.version_info[1]))
 if entityid is not None:
   if (relation[0] == 'P'):
    query = """
    SELECT ?item ?itemLabel
    WHERE
    {
      wd:""" + entityid + """ wdt:""" + relation + """ ?item.
      SERVICE wikibase:label { bd:serviceParam wikibase:language
      "[AUTO_LANGUAGE],en". }
    }"""
   else:
    relation = relation.replace('R', 'P')
    query = """
    SELECT ?item ?itemLabel
    WHERE
    {
      ?item wdt:""" + relation + """ wd:""" + entityid + """.
      SERVICE wikibase:label { bd:serviceParam wikibase:language
      "[AUTO_LANGUAGE],en". }
    }"""
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    results = sparql.query().convert()
    return results["results"]["bindings"]
```

Listing 5.20: Code in Python for the creation of the function get_answer which gives the answer(s) of a question through a SPARQL query for the relation id and the entity id predictions that occurred from the functions relation_prediction and entity_prediction respectively.

So finally having the question answering engine ready i called the answer function to get the answer(s) for each question we put in it, thus calculating the percentage of questions that were answered.

```
print('This is the question answering engine. Type exit to quit')
dropout_prob = 0.3
count_train_questions_total = 0
count_train_questions_answered = 0
while(1):
  question = input()
  if question == "exit":
    break;
  else:
    count_train_questions_total += 1
  question_answered = answer(question, Endict, entities,
  rel_vocab, len(rel_vocab), tokenizer, quests,
  similarity_model, max(q_lengths), "ELU", "ELU",
  activation_relation_best_parameters,
  activation_entity_best_parameters,
  dropout_prob, SEED)
  if question_answered == 1:
    count_train_questions_answered += 1
print(count_train_questions_answered)
print(count_train_questions_total)
print("The percentage of the train questions that are answered correct is
    : {:.2f}%".format((count_train_questions_answered/
   count_train_questions_total)*100))
```

Listing 5.21: Code in Python which is an example of how a QA engine runs.

The questions i posed in both Question Answering Engines I created were either my own or from the annotated_wd_data_test_answerable.txt file regarding questions included in the SimpleQuestions test dataset and are answerable over Wikidata [13,14].

6. EVALUATION

6.1 Summary

In this section the best arithmetic results about the relation indexes, entity span start indexes and entity span end indexes for all the methodologies that i applied are shown. For the neural networks for relation and entity span prediction (described in chapter 1.4), i made experiments by using hyperparameters such as learning rate, activation function, optimizer, loss function and dropout probability [43,44,64,65,66,67].

More specifically in these experiments i use dropout probability values of 0.1, 0.2, 0.3 and 0.4 [30] and learning rate values of 2e-5 and 4e-5 [64]. Also i use the ELU, LeakyReLU, Tanhshrink and Softplus activation functions [43], the optimizer Adam [60,61] and the cross-entropy loss function [67]. The arithmetic results include the training loss, the validation loss, the accuracy score, the f1 score, the precision score and the recall score [20]. Also i plotted the learning curve between the training and the validation loss both on the relation prediction and on entity span prediction.

From the procedure that i follow in order to find the entity label using SPARQL [2,3,4,5] and is decribed in Section 4, i observed that there are cases that the entity label which is found from SPARQL [2,3,4,5] is not appearing in the question wording. For example a question of the train dataset containing answerable over wikidata [14] questions only is ”What a superhero movie that premiered on toonami” and the corresponding entity label for this question that occurred from SPARQL [2,3,4,5] is superhero film. This means that the entity label is not part of its question wording, although they have the same meaning.

This is an example of some similar cases for both the datasets with all the questions and the dataset with only the answerable questions over wikidata [13]. Other examples in this datasets that this phenomenon appears is when the entity label could not be found from SPARQL [2,3,4,5] for a question and as entity label is assumed the corresponding entity id.

The tables 6.1 and 6.2 show the best arithmetic results for the methodology of creating a list consisting of 0 and 1, depending on if the question includes its entity label, then there are ones in the list, else the list consists only of zeros. This list is the entity span for each SimpleQuestions training and validation dataset [13]. More specifically there are trained only the questions of the SimpleQuestions training dataset that include their corresponding entity label, which is found from SPARQL (described in Chapter 4). Respectively for the validation process there are used only the questions of the SimpleQuestions validation dataset that include their entity label in their wording. The best evaluation results for all the methodologies that i applied in this thesis are achieved for the hyperparameter values of learning rate = 2e-5 [64] and dropout probability = 0.3 [44].

It is worth mentioning that when i refer in this section the word wording i mean the wording of a SimpleQuestions dataset question [13]. Also i assume as the best evaluation results for each methodology, the results where the max evaluation metric score i.e accuracy,f1,precision and recall [20] occurred for the relation index prediction and at least for one of the entity span start index or entity span end index prediction.

The figures 6.1 and 6.2 display the learning curves between the training loss and the validation loss for the relation index prediction and the prediction of entity span start and entity span end index respectively for the case of using the SimpleQuestions dataset which in-

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.95998	0.97032	0.96066
Recall-score	0.95998	0.97032	0.96066
Precision-score	0.95998	0.97032	0.96066
Accuracy-score	0.95998	0.97032	0.96066

Table 6.1: Best evaluation results about relation index, entity span start and entity span end index for the methodology of creating a list consisting of 0 and 1, depending on if the question includes its entity label in its wording, then there are ones in the list, else only 0. For these evaluation results is used the SimpleQuestions dataset which includes all the training and validation questions either they are answerable over wikidata or not. For these questions their entity label must exists in their wording.

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.94207	0.97180	0.95732
Recall-score	0.94207	0.97180	0.95732
Precision-score	0.94207	0.97180	0.95732
Accuracy-score	0.94207	0.97180	0.95732

Table 6.2: Best evaluation results about relation index, entity span start and entity span end prediction for the methodology of creating a list consisting of 0 and 1, depending on if the entity label exists in question wording, then there are ones in the list, else 0. For these evaluation results is used the SimpleQuestions subset which includes only training and validation dataset questions that are answerable over wikidata. For these questions their entity label must exists in their wording.

cludes all the training and validation dataset questions (e.g annotated_wd_data_train.txt) [13] that their entity label, which is found from SPARQL [2,3,4,5], is a part of their wording. Correspondingly the figures 6.3 and 6.4 display the learning curves between the training loss and the validation loss for the relation index prediction and the prediction of entity span start and entity span end index respectively for the case of using the SimpleQuestions dataset which includes only the training and validation dataset questions that are answerable over wikidata (e.g annotated_wd_data_train_answerable.txt [13]) [14]. This applies to these questions that their entity label is a part of their wording.

For the above methodology (Entity span as a list consisting of O and 1- All questions that their entity label, which is found from SPARQL [2,3,4,5] queries, exist in their wording):

1. About entity span start and end index prediction, when i use the corresponding dataset (training and validation dataset), which includes only the questions that are answerable over wikidata [14], the best results are achieved for the activation function Softplus [43].
2. When i use the corresponding dataset (training and validation dataset) which includes all the questions (answerable and non answerable over wikidata [14]), the best results about entity span start and end index prediction are reached for the activation function ELU [43]. It is worth mentioning here that, i choose the experiment that includes ELU as the best for this case because it has the highest possible value of entity span start and entity span end index prediction scores simultaneously ,which are equal to 0.97032 and 0.96066 respectively, from all other experiments (with activation functions LeakyReLU, Softplus and Tanhshrink).
3. Through the activation function ELU [43] the best results for relation index prediction are achieved both in the case of using the dataset with all questions and in the case

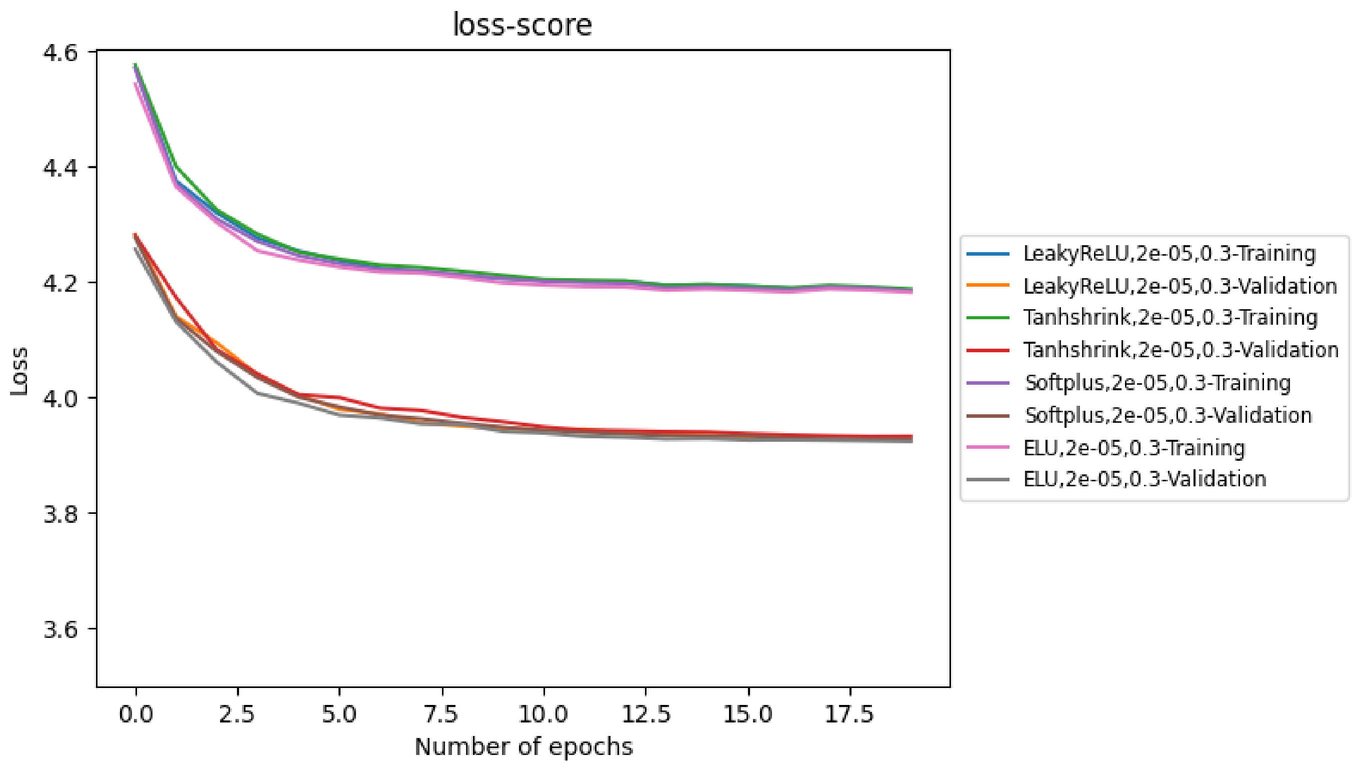

Figure 6.1: Learning curve between relation training and validation loss- methodology of finding entity span indexes (start and end) through entity span, which is a list exclusively consisting of 0 and 1 for the SimpleQuestions dataset that includes all the questions regardless of whether questions are answerable over wikidata or not, where their entity label is a part of their wording.

of using the dataset with only the answerable over wikidata [14] questions.

The tables 6.3 and 6.4 show the best arithmetic results for the methodology of creating the entity span as a sequence 0 and 1, depending on if the question includes its entity label in its wording, then there are ones in the list, else the list consists only of zeros. The main difference with the previous results is that i trained and validate the questions included in the SimpleQuestions training and validation dataset with all the questions (e.g annotated_wd_data_train.txt) [13] respectively either include in their wording their corresponding entity label, which is found from SPARQL [2,3,4,5](described in Chapter 4), or not.

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.96058	0.93403	0.93648
Recall-score	0.96058	0.93403	0.93648
Precision-score	0.96058	0.93403	0.93648
Accuracy-score	0.96058	0.93403	0.93648

Table 6.3: Best evaluation results about relation index, entity span start and entity span end index for the methodology of creating a list consisting of 0 and 1, depending on if the question has its entity label as a part of its wording, else a list full of zeros. For these evaluation results is used the SimpleQuestions dataset that includes all the questions regardless of whether questions are answerable over wikidata or not.

For the above methodology (Entity span as a sequence of O and 1- All questions either if their entity label exist in their wording or not) about entity span start and end index prediction, when i train and validate the corresponding dataset questions (training, validation,

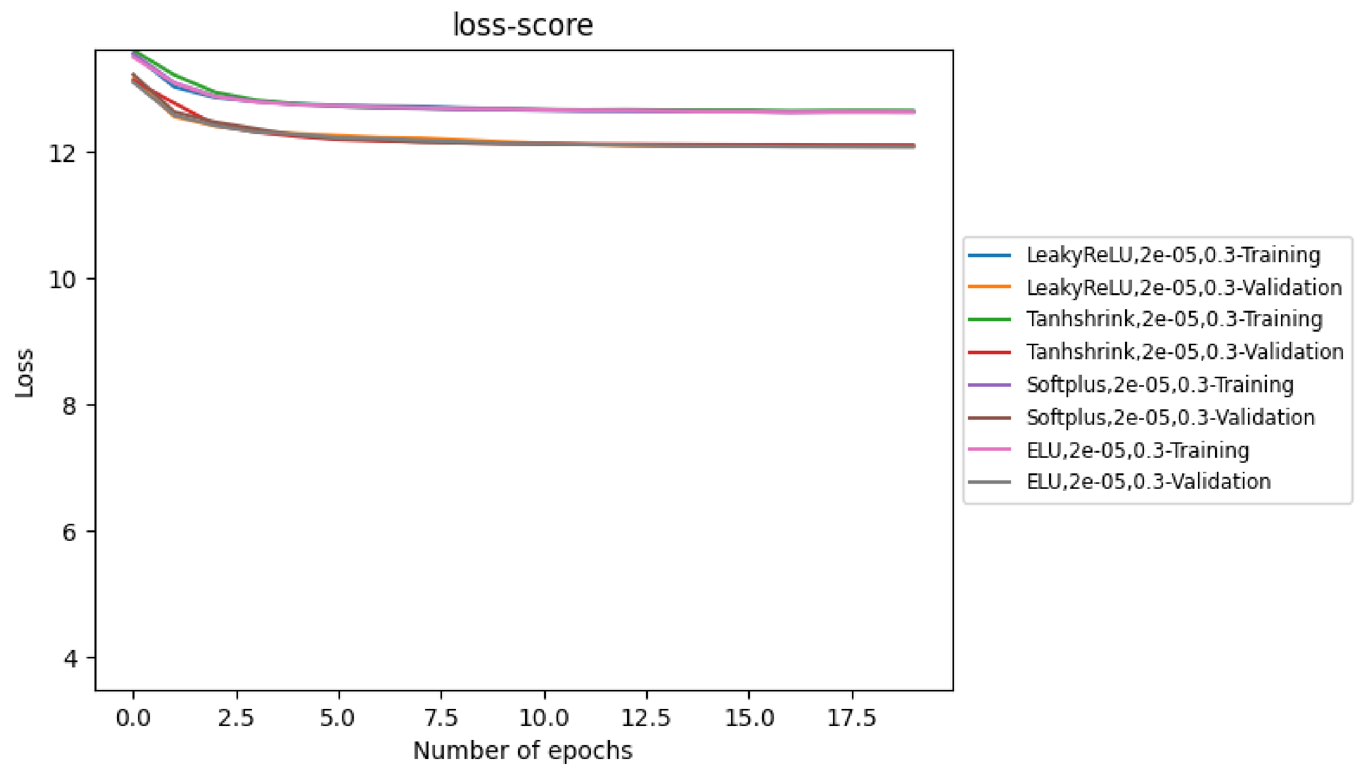

Figure 6.2: Learning curve between entity span training and validation loss- methodology of finding entity span indexes (start and end) through entity span, which is a list exclusively consisting of 0 and 1, for the SimpleQuestions dataset that includes all the questions regardless of whether questions are answerable over wikidata or not. The entity label of these questions is a part of their wording.

test), for the SimpleQuestions dataset which includes only the answerable over wikidata questions [13], the best results are achieved for the activation function ELU [29], while for the case of SimpleQuestions dataset which includes all the questions (answerable, non-answerable over wikidata [14]), the best results are achieved for the activation function Softplus[43]. About relation index prediction the best results are achieved for the activation function ELU [43], regardless if i train and validate all SimpleQuestions dataset questions (answerable and non-answerable over wikidata [13,14]) or only questions that are answerable over wikidata [14].

The figures 6.5 and 6.6 display the learning curves between the training loss and the validation loss for the relation index prediction and the prediction of entity span start and entity span end index respectively for the case of using the SimpleQuestions dataset with all the training and validation questions (either answerable over wikidata [14] or not, e.g annotated_wd_data_train.txt[13]) regardless of whether their entity label is a part of their wording.

Also the figures 6.7 and 6.8 show the learning curves between the training loss and the validation loss for the relation index prediction and the prediction of entity span start and entity span end index respectively for the case of using SimpleQuestions dataset with training and validation questions that are answerable over wikidata [13] (e.g annotated_wd_data_train_answerable.txt) [13], regardless of whether their entity label, which is found from SPARQL, is a part of their wording.

The tables 6.5 and 6.6 show the best arithmetic results for the methodology of finding the entity span start and entity span end index by using spacy library [25,26] or the Sentence

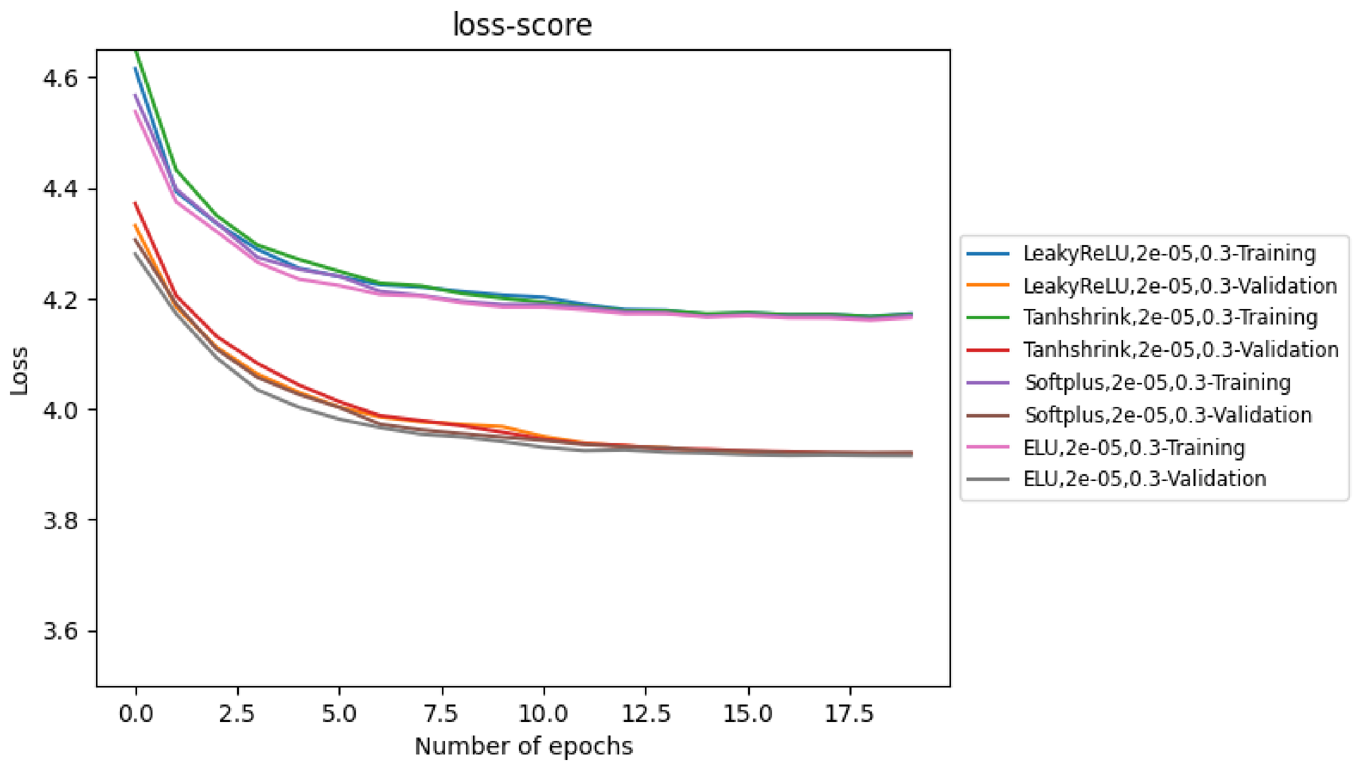

Figure 6.3: Learning curve between relation training and validation loss-methodology of finding entity span indexes (start and end) through entity span , which is a list consisting of 0 and 1 or only of zeros, for the SimpleQuestions dataset that includes all the questions that answerable over wikidata. The entity label of these questions is a part of their wording.

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.94979	0.93153	0.92521
Recall-score	0.94979	0.93153	0.92521
Precision-score	0.94979	0.93153	0.92521
Accuracy-score	0.94979	0.93153	0.92521

Table 6.4: Best evaluation results about relation index, entity span start and entity span end prediction for the methodology of creating a list consisting of 0 and 1, depending on if the question has its entity label as a part of its wording, else a list full of zeros. For these evaluation results is used the SimpleQuestions dataset that includes questions that are answerable over wikidata.

Transformers library [13] with the cosine similarity metric as the similarity metric [23,24]. This methodology is described analytically in chapter 1.4.

For this methodology of finding entity span start and entity span end index by using spacy library [25,26] or Sentence Transformers library [21,22] combined with the cosine similarity metric [23,24]) about entity span start and end index prediction, when i train and validate the corresponding dataset questions (training, validation, test), for the SimpleQuestions dataset which includes only the answerable over wikidata questions [13], the best results are achieved for the activation function Softplus [43], while for the case of SimpleQuestions dataset which includes all the questions (answerable, non-answerable over wikidata [14]), the best results are achieved for the activation function ELU[43]. About relation index prediction the best results are achieved for the activation function ELU [43], both for the whole SimpleQuestions dataset and the dataset including only the answerable over wikidata questions [13,14].

The tables 6.7 and 6.8 show the best arithmetic results for the methodology of finding the

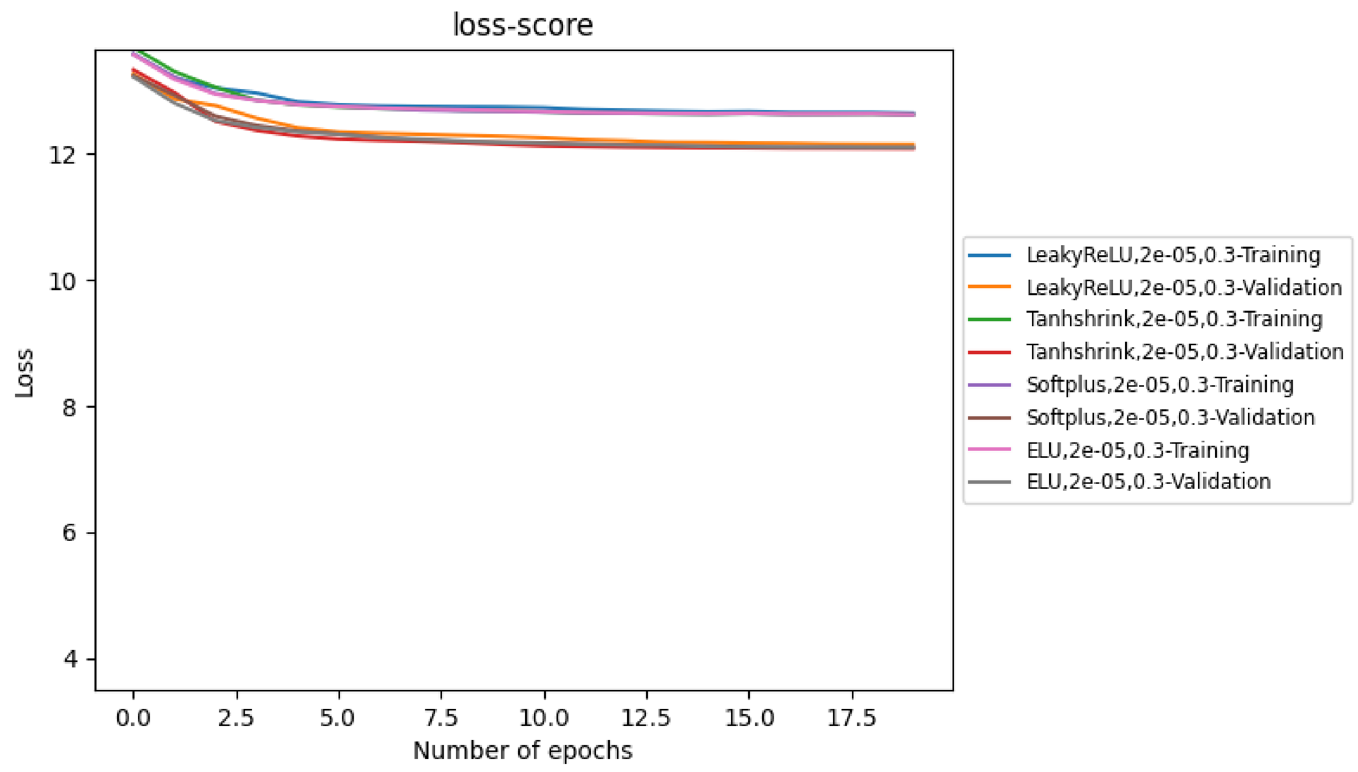

Figure 6.4: Learning curve between entity span training and validation loss-methodology of finding entity span indexes (start and end) through entity span, which is a list consisting of 0 and 1 or only of zeros, for the SimpleQuestions dataset that includes all the questions that answerable over wikidata. The entity label of these questions is a part of their wording).

entity span start and entity span end index by using spacy library [17] or the Sentence Transformers library [13] with the jaccard similarity metric as the similarity metric [23,24]. This methodology is described analytically in chapter 1.4.

For this methodology (Finding entity span start and entity span end index by using spacy library [25,26] or jaccard similarity metric [23,24]):

1. About relation index prediction, the best results are achieved for the activation function ELU [43] either for full SimpleQuestions dataset, or with the subset contains only answerable over wikidata questions [13,14].
2. About entity span indexes (start and end) prediction, when i use the full SimpleQuestions dataset and also when i use (train and validate) the corresponding SimpleQuestions dataset [13] (training and validation), which contains only questions that are answerable over wikidata [14], the best results are achieved for the activation function Softplus[43].

So about entity span prediction the experiment that achieve the highest possible score for the entity span start prediction and the entity end prediction, i assume them as the most successful.

For example in the methodology that i use either Sentence Transformer [21,22] with cosine similarity metric [23,24] or spacy library [25,26] to find entity span start and end indexes for questions that are answerable over wikidata [13] in order to predict them layer, in entity span prediction the experiment that i use the activation function ELU [43], gives entity span start index prediction score for all types of scores (accuracy, f1, precision, recall) equal

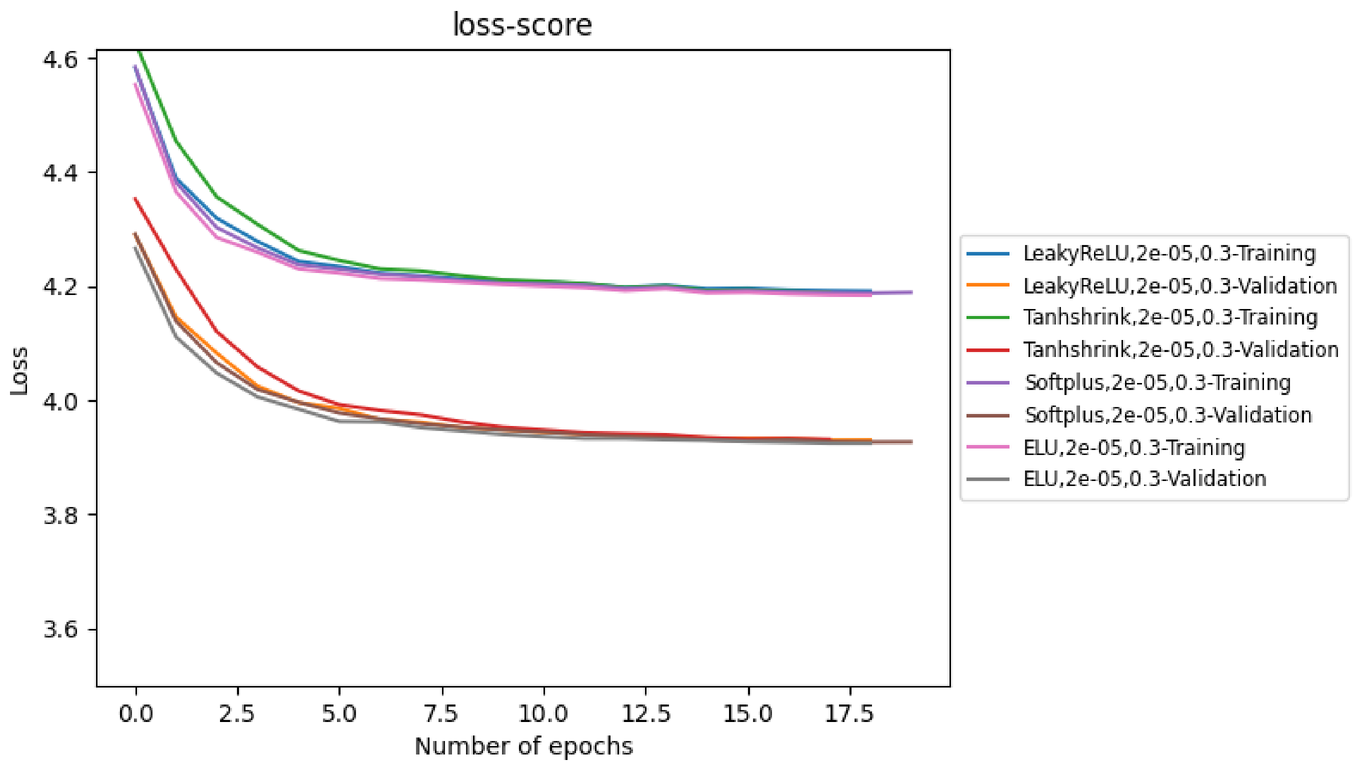

Figure 6.5: Learning curve between relation training and validation loss for the SimpleQuestions dataset which contains all questions (answerable and non-answerable over wikidata). This learning curve is the same for the methodology of creating entity spans as lists consisting of 0 and 1 or only of 0, for the methodology of finding entity span start and end indexes through either spacy library or Sentence Transformers library combined with cosine similarity metric and for the methodology of finding entity span start and end indexes through through either spacy library or by using jaccard similarity metric (Described in chapter 1.4).

to 0.93610 and entity span end index prediction score equal to 0.92065. The respective experiment that i use the activation function Softplus [43] give entity span start and end prediction scores equal to 0.93083 and 0.92591 correspondingly.

I finally kept the experiment with Softplus [43], because it has the max prediction score for entity span end compared with the corresponding score of experiment with Softplus [43], and also because although Softplus [43] has higher entity span start score, the difference between these entity span start prediction scores is small.

Also when i refer the words ”full” or ”whole” for the SimpleQuestions dataset [13] i mean that this dataset contains answerable and non-answerable questions [13] over wikidata [14].

From all these experiments, the best evaluation relation or entity span model parameters and the best evaluation scores that are occurred, i observe that the relation prediction scores on all the SimpleQuestions dataset [31], regardless of the methodology that i followed in each case, gives very good results after 20 epochs, with the best values of all scores, i.e f1 score, recall score, precision score, accuracy score ranging from 94% to 96%. For the dataset that contains only answerable questions over wikidata [13] the corresponding scores are a little bit lower in the range of 94% to 95% for all the methodologies that i applied. The best f1, accuracy, precision and recall scores are achieved with the activation function of ELU [43] for the SimpleQuestions dataset that includes all the questions (e.g annotated_wd_data_train.txt) [13] (table 6.3), for the methodology of finding the entity span indexes (start and end) by creating lists that consists of 0 and 1 or only zeros

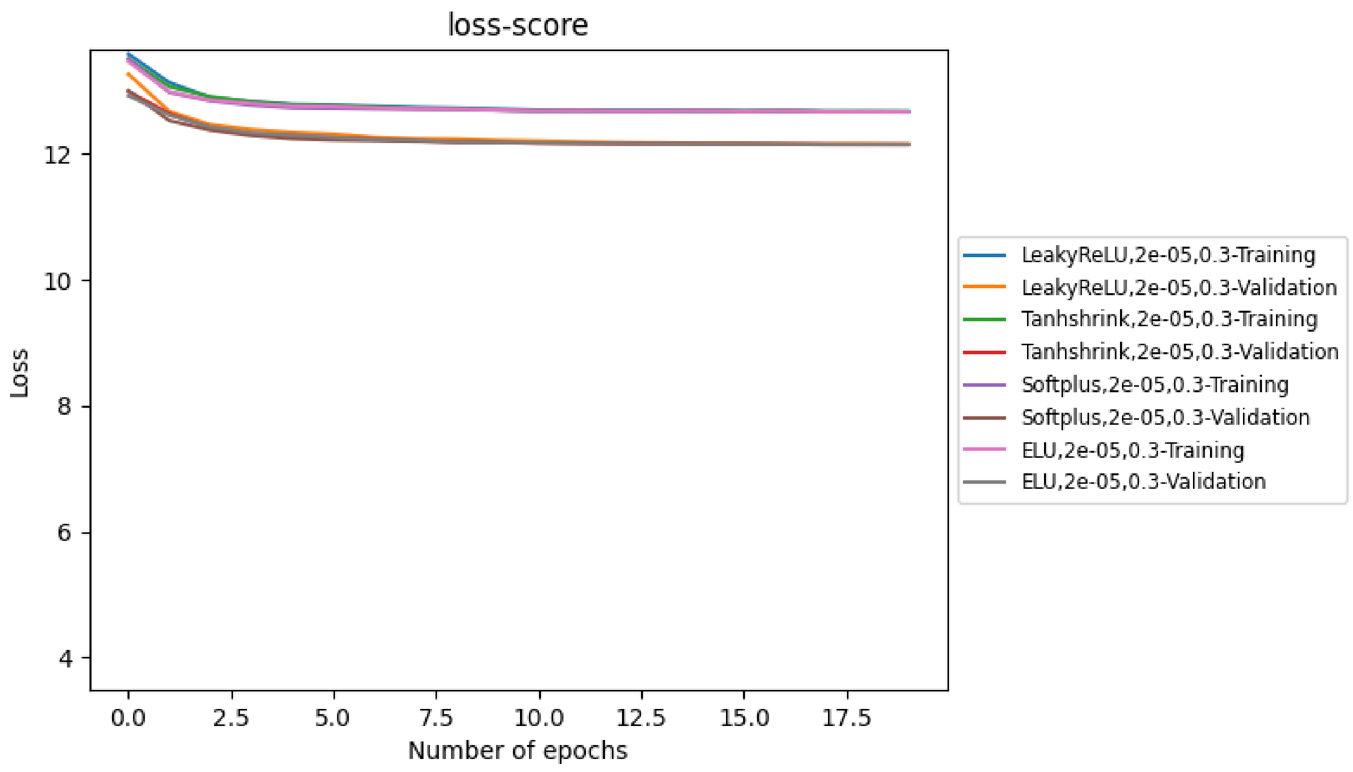

Figure 6.6: Learning curve between entity span training and validation loss- methodology of finding entity span indexes (start and end) through entity span, which is a list consisting of 0 and 1 for the SimpleQuestions dataset which includes all questions (answerable and non-answerable over wikidata.

and assuming these lists as entity spans.

About the entity span prediction the methodology that i achieve the best evaluation scores is the methodology that i create the lists that consists of 0 and 1 or only zeros and assuming these lists as entity spans only for the questions that have their entity label, which is found from SPARQL [2,3,4,5,104,105], as a part of their wording. More specifically i achieve the highest scores for the whole SimpleQuestions dataset [13] with the ELU [43] activation function (table 6.1) and for the SimpleQuestions subset that includes only the questions that are answerable over wikidata(e.g annotated_wd_data_train_answerable.txt [13]) (table 6.2) with the activation function Softplus [43].

Also the learning curves between the training losses and the validation losses for all the methodologies both for relation prediction and entity span prediction are very close to each other, i.e these curves converge a lot, as can be seen from figures 6.1-6.12. This fact and in combination with the very good evaluation results for relation index prediction and entity span indexes (start, end) prediction scores (f1,precision,recall,accuracy) [20], shows that the training and the validation of each relation prediction and entity span prediction model is made with great success.

For all the other methodologies of finding entity span start and end indexes(entity spans as a sequence of 0 and 1, regardless if their corresponding entity label is included in their wording, spacy library [25,26], Sentence Transformers [13,14] combined with cosine similarity metric and the methodology that uses only jaccard similarity metric[23,24]), the relation prediction scores are the same with each other for each SimpleQuestions dataset [13] which contains all questions and respectively are equal with each other SimpleQuestions subset containing only the answerable questions [13]), but different between these

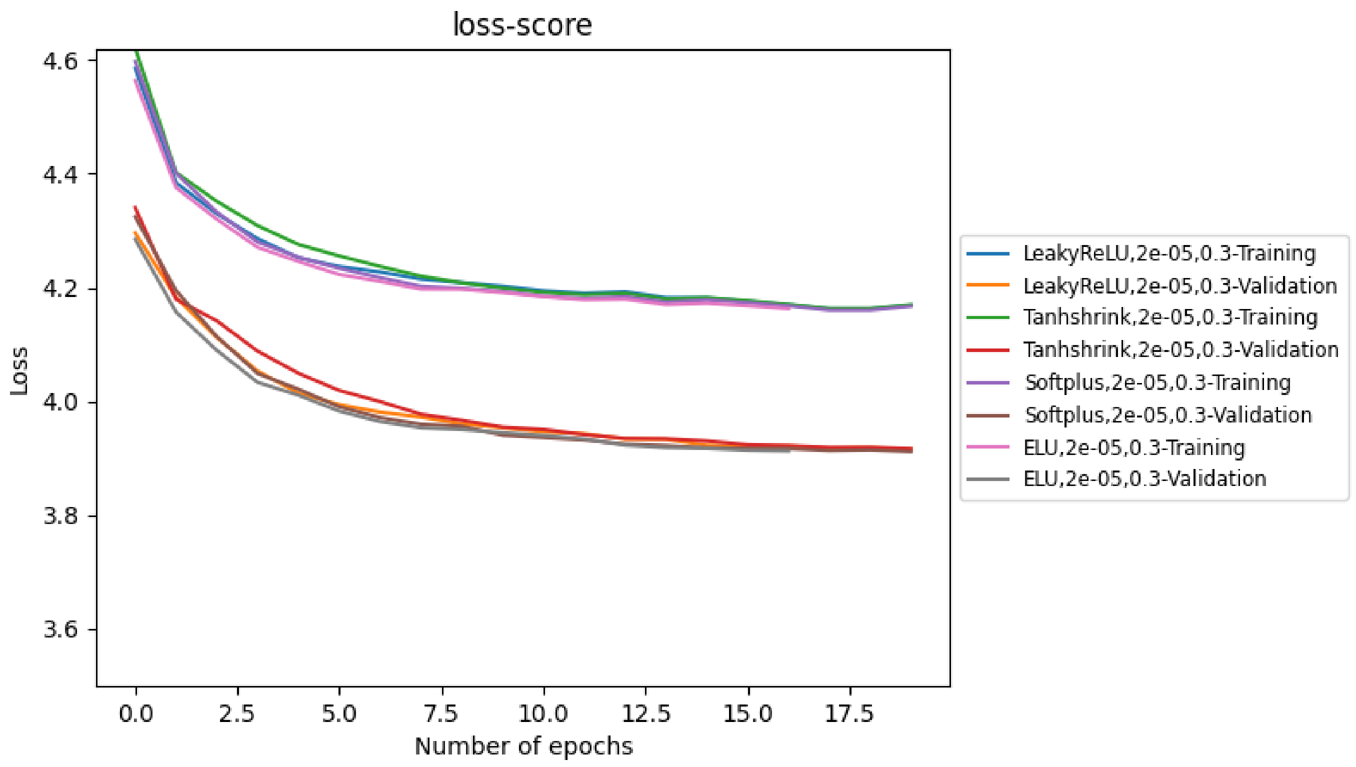

Figure 6.7: Learning curve between relation training and validation loss for the SimpleQuestions dataset which includes only questions which are answerable over wikidata. This learning curve is the same for the methodology of creating entity spans as lists consisting of 0 and 1 or only of 0, for the methodology of finding entity span start and end indexes through either spacy library or Sentence Transformers library combined with cosine similarity metric and for the methodology of finding entity span start and end indexes through through either spacy library or by using jaccard similarity metric (Described in chapter 1.4).

two different datasets, because the process in order to find the relation index through the creation of a vocabulary with all unique relation ids from training and validation questions and make predictions for them through relation neural network (chapter 1.4) [26,27] is completely independent from the methodologies of finding entity span indexes (start and end) [26,27]. In other words the relation index predictions are not affected at all from any methodology applied for entity span prediction.

All these other methodologies have very good results but a little bit lower compared to the methodology of creation of entity spans as sequences of 0 and 1 only for the questions that have their entity label as a part of their wording, because the methodology of finding entity span indexes with spacy library [25,26] mixed either with the jaccard similarity metric [23,24] or with the use of Sentence Transformers [21,22] library combined with cosine similarity metric [23,24] in a question tries to find not only the equality between entity label and a question part with equal length with it through spacy library [25,26], but, if the equality is not achievable, i tried to find similar question parts through spacy library [25,26] mixed either with jaccard similarity metric [23,24] or with the combination of Sentence Transformers [25,26] library with cosine similarity metric [23,24].

The finding of similar question parts with the corresponding entity label is a work that it is not only focused in the equality but also on the similarity between an entity label and a question part. So for a percentage of the questions have finally entity span indexes which are corresponding to question phrases that are lexically (most of times) or semantically (a few times) similar to their entity labels, where this similarity in several cases is not so

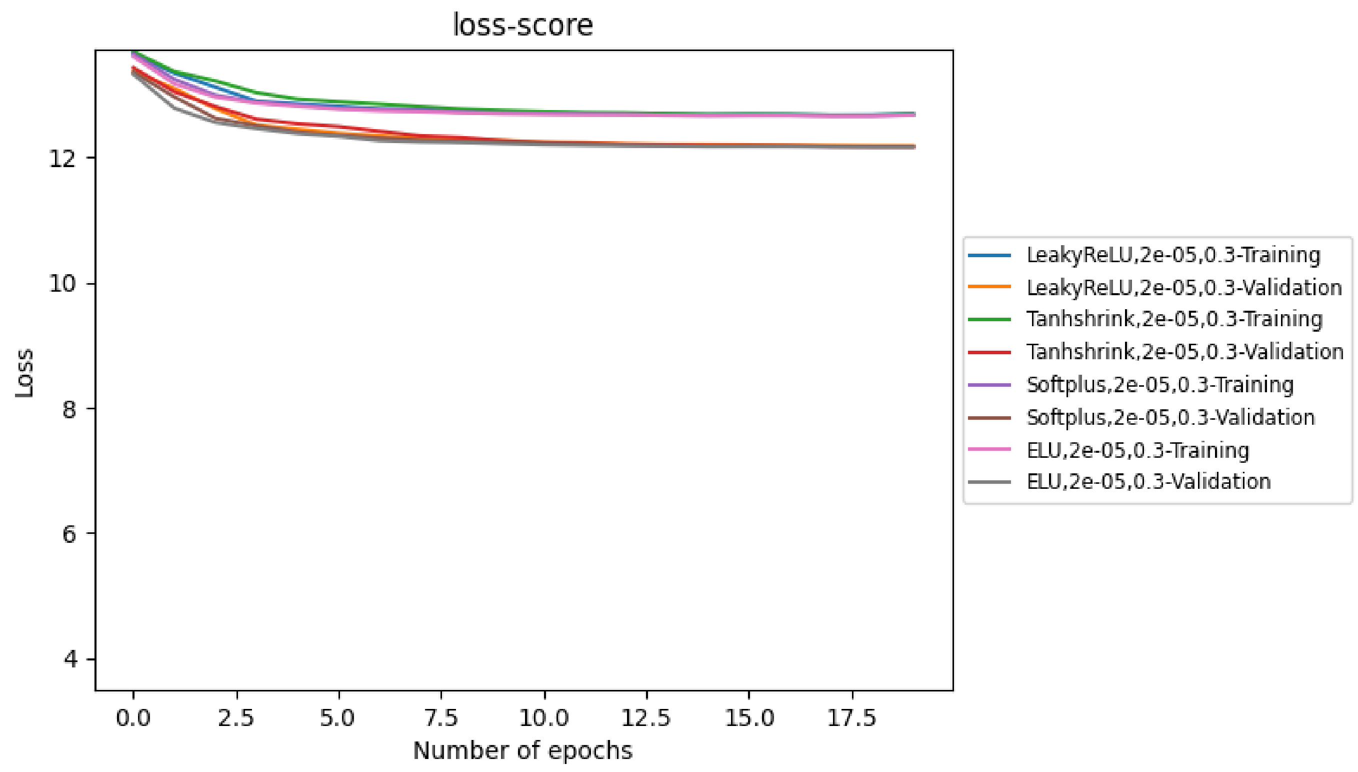

Figure 6.8: Learning curve between entity span training and validation loss -methodology of finding entity span indexes (start and end) through entity span, which is a list consisting of 0 and 1 for the SimpleQuestions dataset which includes only questions which are answerable over wikidata.

great and in fact it could be quite small through cosine or jaccard similarity metric [23,24]. Finally this could lead to a small decrease on the entity span neural network performance for this methodology and as a sequence on the entity span prediction scores compared to the methodology of entity spans to be sequences of 0 and 1 only for the questions that have their entity label as a part of their wording. Thus the predictions could lose a bit of their credibility. Thus cosine similarity metric in general leads to better model parameters, so and evaluation results than the jaccard similarity metric for this thesis [23,24].

Also about the methodology of creating entity spans as a sequence of 0 and 1 for SimpleQuestions dataset [13], regardless if the corresponding entity label of each question is included in their wording, as a methodology has as aim to find the entity label itself, not something similar with it in the question. Based on the previous paragraph that i want to find the equality or the similarity of the entity label with an question part with the same length as its own, now there are the same amount questions that their entity label is not part of their question wording, so their entity spans are full of zeros.

This is something that is expected to cause some decrease on the entity span neural network performance for this methodology compared with the same methodology that is applied only for questions that their entity label is a part of their wording and as a result on the entity span prediction scores both for the SimpleQuestions dataset [13] containing all questions and the corresponding SimpleQuestions dataset [13] containing only answerable questions [13] over wikidata.

All the above best evaluation results, i.e the best parameters that lead the neural network to achieve these results, from each one of the methodologies that i implemented, help me in order to investigate how many questions can the question answering engines that

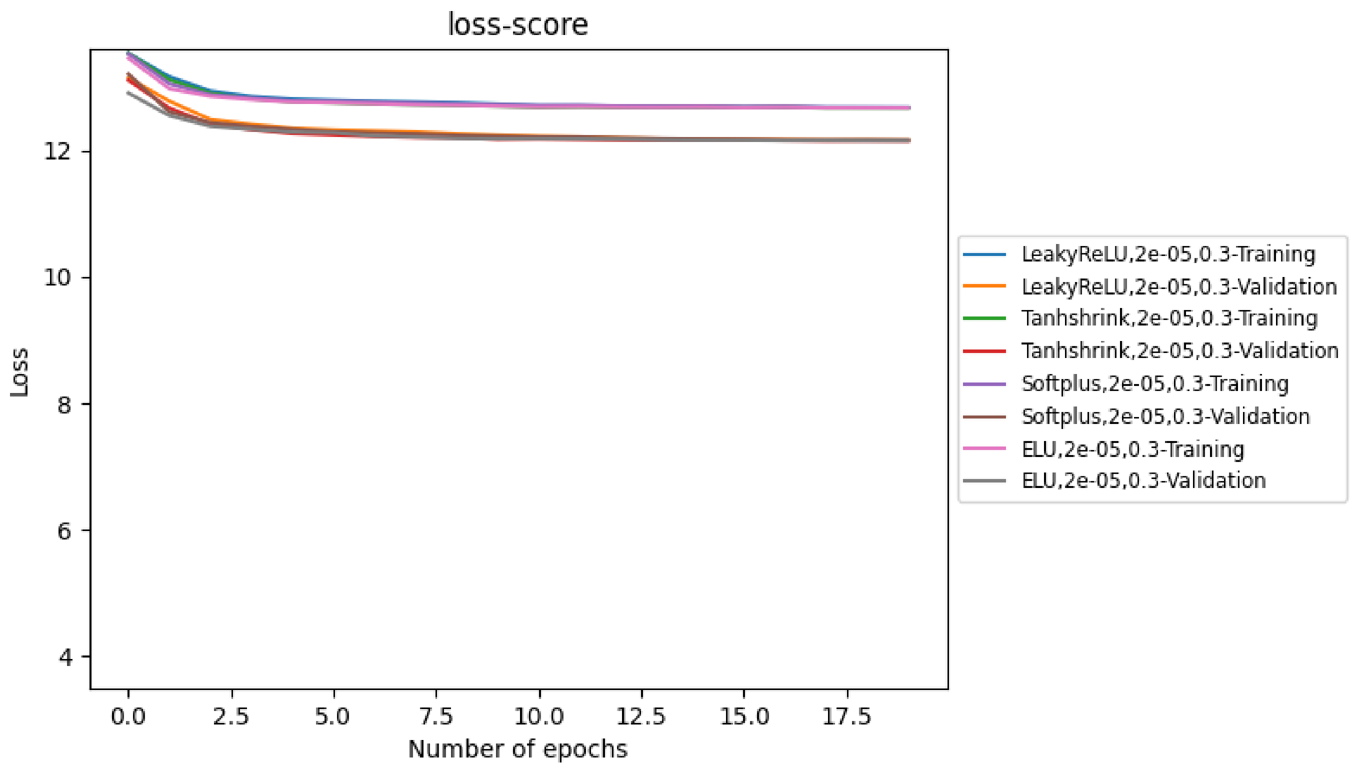

Figure 6.9: Learning curve between entity span training and validation loss-methodology of finding entity span indexes either with spacy library or Sentence Transformers + cosine similarity metric for the SimpleQuestions dataset (training and validation dataset) that includes all the questions (answerable, non-answerable over wikidata).

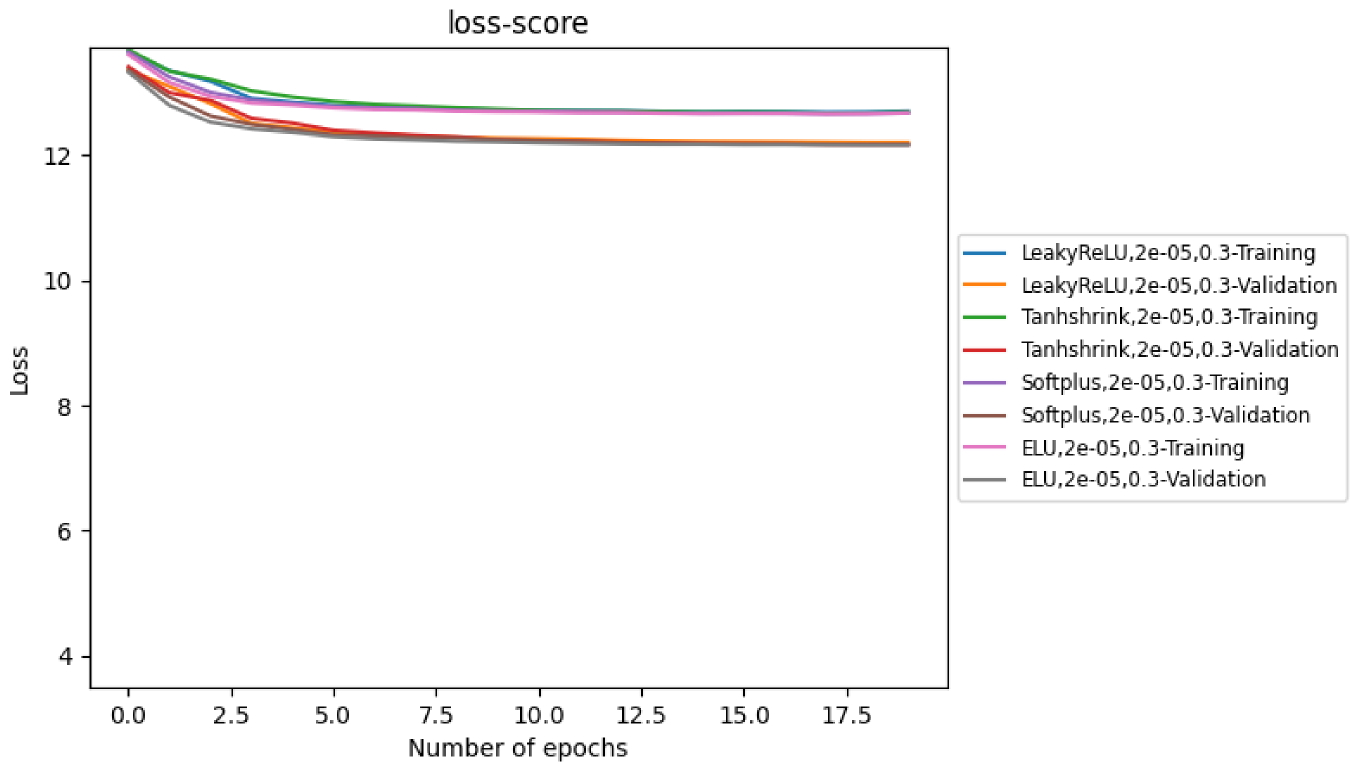

Figure 6.10: Learning curve between entity span training and validation loss-methodology of finding entity span indexes either with spacy library or Sentence Transformers + cosine similarity metric for the SimpleQuestions dataset (training and validation dataset) that includes only questions that are answerable over wikidata.

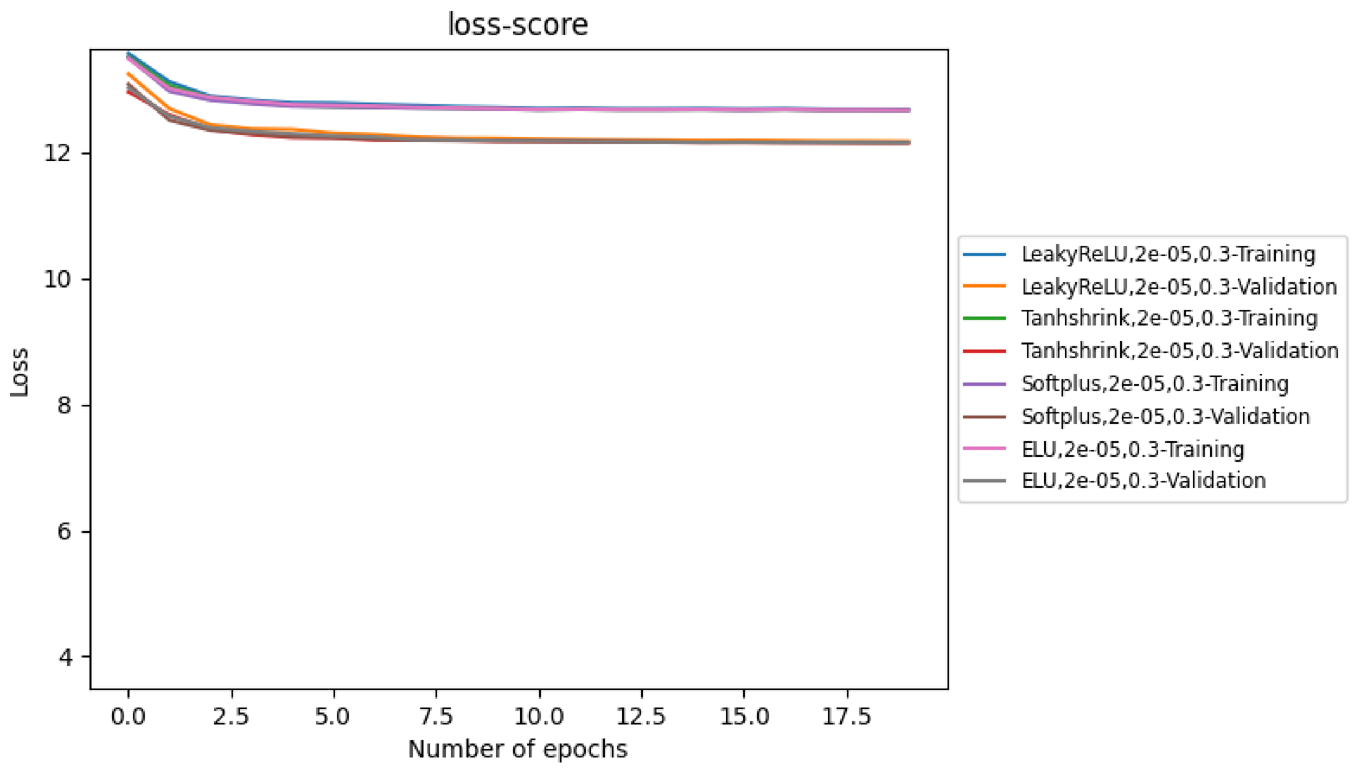

Figure 6.11: Learning curve between entity span training and validation loss-methodology of finding entity span indexes either with spacy library or jaccard similarity metric for the SimpleQuestions dataset which contains all questions (answerable, non-answerable over wikidata).

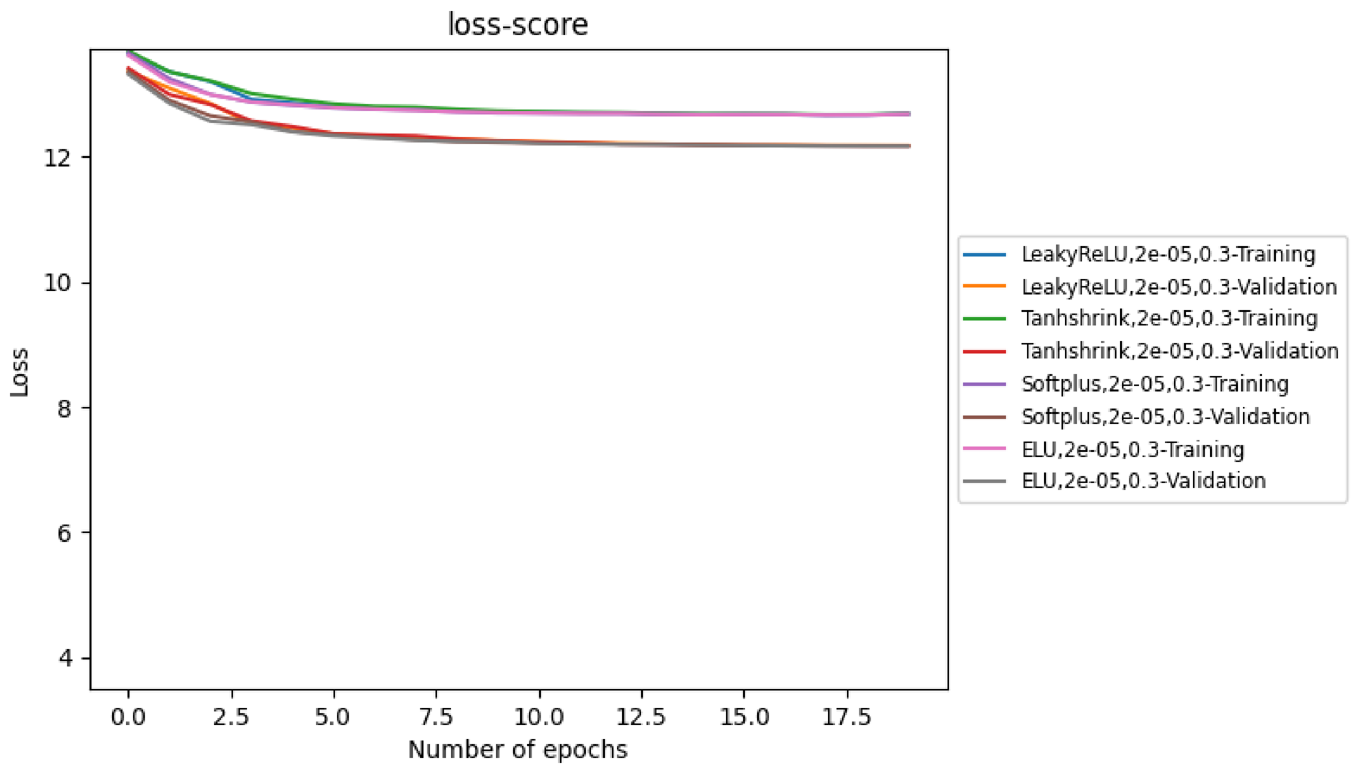

Figure 6.12: Learning curve between entity span training and validation loss-methodology of finding entity span indexes either with spacy library or jaccard similarity metric for the SimpleQuestions dataset which contains questions that are answerable over wikidata.

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.96058	0.92688	0.92872
Recall-score	0.96058	0.92688	0.92872
Precision-score	0.96058	0.92688	0.92872
Accuracy-score	0.96058	0.92688	0.92872

Table 6.5: Best evaluation results about relation index, entity span start and entity span end index for the methodology of finding entity span indexes (start and end), by using sentence transformers with cosine similarity metric, or else through difflib.get_close_matches() for the SimpleQuestions dataset which contains all (training, validation, test) the questions (answerable, non-answerable over wikidata.

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.94979	0.93083	0.92591
Recall-score	0.94979	0.93083	0.92591
Precision-score	0.94979	0.93083	0.92591
Accuracy-score	0.94979	0.93083	0.92591

Table 6.6: Best evaluation results about relation index, entity span start and entity span end index for the methodology of finding entity span indexes (start and end), sentence transformers with cosine similarity metric, or else through difflib.get_close_matches() for the SimpleQuestions dataset (training and validation dataset) that includes only questions that are answerable over wikidata.

i created (Section 5 - The Sentence Transformer Question Answering Model) and the degree of reliability for these answers. More specifically for all the methodologies for the question answering engines i have created, the percentage (%) of the answered questions in an indicative number of 60 questions that i insert in a chatbox is shown below.

These 60 questions i have inserted them in a .txt file named ”Questions_testing_with_answers” and i read them as a .csv file consisting of the question, relation id, entity id, answer id and answer label. The creation of this .cscv file is described in Section 5 - The Sentence Transformer Question Answering Model

For the tables below: QA Engine 1: Question answering engine by using spacy library [25,26]

QA Engine 2: Question answering engine by using Sentence Transformers [21,22] combined with cosine similarity metric [23,24]

QA Engine 3: Question answering engine by using jaccard similarity metric [23,24]

Also in order to understand better how the above QA engines work, i assume that a question is answered correctly if the QA engine finds the correct answer that exists in the .txt file ”Questions_testing_with_answers” for each question and answers that have the highest possible simialrity score with this answer. So i assume these questions as correctly answered questions.

Additionally i assume if for a question a QA engine finds answers that are different and even a little bit from the answer of this question included in the ”Questions_testing_with_answers” .txt file. The similarity between the answer of the .txt file and an answer that is found through a QA engine for these questions is lower than the corresponding similarity between the answer of the .txt file and an answer that is calcuated for the correctly answered questions as i mentioned earlier.

Methodology 1: Creation of list consisting exclusively of 0 and 1 (entity span) for questions

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.96058	0.92443	0.92729
Recall-score	0.96058	0.92443	0.92729
Precision-score	0.96058	0.92443	0.92729
Accuracy-score	0.96058	0.92443	0.92729

Table 6.7: Best evaluation results about relation index, entity span start and entity span end index for the methodology of finding entity span indexes (start and end), by using jaccard similarity metric, or else through difflib.get_close_matches() for the SimpleQuestions dataset (training, validation, test) which comprises all questions (either answerable over wikidata or not).

Evaluation Metrics	Relation index	Entity span start index	Entity span end index
F1-score	0.94979	0.92907	0.92205
Recall-score	0.94979	0.92907	0.92205
Precision-score	0.94979	0.92907	0.92205
Accuracy-score	0.94979	0.92907	0.92205

Table 6.8: Best evaluation results about relation index, entity span start and entity span end index for the methodology of finding entity span indexes (start and end), by using jaccard similarity metric, or else through difflib.get_close_matches() for the SimpleQuestions dataset (training, validation, test) which comprises questions that are answerable over wikidata.

that their entity label, which is found from SPARQL [2,3,4,5], is a part of their wording.

Methodology 2: Creation of list consisting exclusively of 0 and 1 or only of 0 (entity span) for all questions regardless if their entity label, which is found from SPARQL, is a part of their wording.

Methodology 3: Finding entity span start and entity span end index by using spacy library [25,26] or Sentence Transformers library [21,22] combined with the cosine similarity metric [23,24].

Methodology 4: Finding entity span start and entity span end index by using spacy library [25,26] or Sentence Transformers library [21,22] combined with the jaccard similarity metric [23,24].

Correctly Answered questions(%)	QA engine 1	QA engine 2	QA engine 3
Methodology 1	68.33	65	50
Methodology 2	70	66.67	53.33
Methodology 3	70	65	53.33
Methodology 4	68.33	68.33	53.33

Table 6.9: Percentage (%) of correctly answered questions (Questions that each QA engine finds the answer from the "Questions_testing_with_answers" .txt file and some quite similar answers) for each one of the Question Answering engines and methodlogies that are implemented for the SimpleQuestions dataset (training, validation, test) which comprises all questions (answerable, non answerable over wikidata).

From all the previous evaluation scores (f1, precision, recall, accuracy [20]-tables 6.1-6.8), it is easy to understand that both in case of answerable SimpleQuestions dataset questions [13] over wikidata [14] and in case of all SimpleQuestions dataset questions [13], the combination of spacy library [25,26] with the Sentence Transformers library [21,22] either with cosine similarity metric or jaccard similarity metric [23,24], leads to very good neural network either relation or entity span performance and as a result to very good evaluation results.

Answered questions(%)	QA engine 1	QA engine 2	QA engine 3
Methodology 1	15	16.67	30
Methodology 2	15	20	26.67
Methodology 3	15	20	28.33
Methodology 4	16.67	18.33	28.33

Table 6.10: Percentage (%) of answered questions (Questions that each QA engine finds answers that at least most of them have either (very) small or no similarity with the answer from the ”Questions_testing_with_answers” .txt file. One of these answers could maybe be the answer of the .txt file itself) for each one of the Question Answering engines and methodlogies that are implemented for the SimpleQuestions dataset (training, validation, test) which comprises all questions (answerable, non answerable over wikidata).

Correctly Answered questions(%)	QA engine 1	QA engine 2	QA engine 3
Methodology 1	65	61.67	48.33
Methodology 2	66.67	65	50
Methodology 3	65	66.67	50
Methodology 4	68.33	68.33	51.67

Table 6.11: Percentage (%) of correctly answered questions (Questions that each QA engine finds the answer from the ”Questions_testing_with_answers” .txt file and some quite similar answers) for each one of the Question Answering engines and methodlogies that are implemented for the SimpleQuestions datasets (training, validation, test) which contain only answerable over wikidata questions.

Nevertheless, from these evaluation results it cannot be made clear the impact of the spacy [25,26] and Sentence Transformers [21,22] libraries and respectively the impact of the cosine and jaccard similarity metric [23,24] separately. So in order to investigate this effect for each one of these libraries and similarity metrics i create the three types of answering engines that are described in Section 5-The Question Answering Model.

From these percentages (%) of answered questions (tables 6.9-6.12) for the 60 questions of ”Questions_testing_with_answers” .txt file, i observe that for all the methodologies either for all or for answerable questions over wikidata [14] the best percentage of totally correct answered questions, which is equal to 70%, is achieved by the QA engine that includes spacy library [25,26] for the best parameters of the methodology including Sentence Transformers similarity model [21,22] and cosine similarity metric [23,24] and for the best parameters of the methodology of creating entity spans consisting of 0 and 1, regardless if an entity label of a question consists part of its wording. The most remarkable is that for this multitude of questions for these two aforementioned methodologies, the percentages of totally correct answered questions is exactly the same.

This observation is strengthen more by the percentages of questions that are neither answered totally correct nor answered totally incorrect that is achieved for these two aforementioned methodologies, which is equal to 15% for both of these methodologies.

The QA engine including only spacy library [25,26] for all the methodologies achieves very good percentages(%) of totally correct answered questions, which ranges from 65% to 70% and also not insignificant, but much more lower than percentages(%) of totally correct answered questions, percentages of only answered questions with answers neither totally correct nor totally incorrect, which ranges from 8.33% to 18.33%.

The QA engine including the Sentence Transformer similarity model [21,22] combined with the cosine similarity metric[25,26] for all the methodologies attain very good percent-

Answered questions(%)	QA engine 1	QA engine 2	QA engine 3
Methodology 1	8.33	11.67	23.33
Methodology 2	18.33	23.33	31.67
Methodology 3	18.33	21.67	33.33
Methodology 4	16.67	18.33	26.67

Table 6.12: Percentage (%) of answered questions (Questions that each QA engine finds answers that at least most of them have either (very) small or no similarity with the answer from the "Questions_testing_with_answers" .txt file. One of these answers could maybe be the answer of the .txt file itself) for each one of the Question Answering engines and methodlogies that are implemented for the SimpleQuestions datasets (training, validation, test) which contain only answerable over wikidata questions.

ages(%) of totally correct answered questions, which ranges from 61.67% to 68.33%, which consists lower percentages than the percentages(%) that QA engine including only spacy library [25,26] performs. Also the percentages(%) of answered questions with answers neither totally correct nor totally incorrect for this QA engine, which ranges from 11.67% to 23.33%, are a little bit higher than the corresponding percentages(%) that QA engine including only spacy library [25,26] performs.

From all these percentages is also shown that the jaccard similarity metric [23,24] is not the best similarity metric compared with cosine similarity metric [23,24] and also compared spacy library [25,26]. The percentages(%) of totally correct answered questions which ranges from 48.33% to 53.33% are not very high and also the percentages(%) of answered questions with answers neither totally correct nor totally incorrect for this QA engine, which ranges from 23.33% to 33.33% are the highest that are observed across the three QA engines.

Essentially the QA engine that includes spacy library,with a slight difference from the combination of Sentence Transformer similarity model [21,22] with the cosine similarity metric [23,24], is the best QA engine model because the difference between the ranges of the percentages(%) of totally correct answered questions and the ranges of the percentages(%) of answered questions with answers neither totally correct nor totally incorrect are more farer than all the other QA engines. This shows that spacy library [25,26], is able not only to find the correct (real) answer to the question with success and also answers that have an common property (described through relation id) with the correct answer, for example the birthplace, the type of film etc. Also these percentages(%) shows that answers with an common property with the real answer are found from spacy library [25,26] more frequently than answers that are neither totally right nor totally wrong, but even a little close lexically (or semantically) to the real answer.

In order to explain the important difference in the performance of the QA engine models that include cosine and jaccard similarity metric [23,24], consider the sentences below: "The bottle is empty", "There is nothing in the bottle", "The bottle is empty empty", "David Beckham was playing at Manchester United", "Rolling Stones is a very-well known group playing rock music".

More specifically for the sentences "The bottle is empty" and "There is nothing in the bottle" by using the cosine similarity score function [26,27] that is described in Section 1.4-Methodology and in Section 5-The Question Answering model with the help of the Sentence Transformer similarity model [21,22], the cosine similarity score between these two sentences is equal to 0.66. Also by using the jaccard similarity score function [26,27] that is described in Section 1.4-Methodology and in Section 5-The Question Answering model, the jaccard similarity score between these two sentences is equal to 0.43 [27].

As before for the sentences ”There is nothing in the bottle” and ”The bottle is empty empty”, the cosine similarity metric is equal to 0.64, while the jaccard similarity remains equal to 0.43. Also for the sentences ”David Beckham was playing at Manchester United” and ”Rolling Stones is a very-well known group playing rock music”, the cosine similarity metric is equal to 0.19, while the jaccard similarity remains equal to 0.06 [27].

Thus from all the above examples it is shown that in general cosine similarity metric [23,24] gives much better similarity scores than the jaccard similarity metric [23,24]. So cosine similarity metric [23,24] could find better more similar answer to the real answer of an question and works better with text data and with text embeddings than jaccard similarity metric [23,24]. Generally jaccard similarity metric is rarely used with text data and it does not work with text embeddings. This is the reason why in many questions testing for each methodology in the QA engine including only jaccard similarity metric [26,27], the similarity score between an answer from the QA engine and the real answer for each question is equal to 0.

Essentially cosine similarity metric is not affected by the magnitude/length of the feature vectors, in this case the lengths of the questions and jaccard similarity metric takes into account only the set of unique words for each text document [24]. The difference between the first ”The bottle is empty” and the third sentence ”The bottle is empty empty” is the word ”empty” that exists twice in the third sentence. Based on what has been referred in this paragraph it is worth noting that jaccard similarity metric score [23,24], for both comparing the sentence ”The bottle is empty” with the sentence ”There is nothing in the bottle” and comparing the sentence ”The bottle is empty” with the sentence ”There is nothing in the bottle” remains unchanged, while the cosine similarity metric scores although they are higher would drop a little bit and this means that cosine similarity metric [23,24] is less sensitive to a difference in lengths [24].

Previously i observe that for the methodology of entity spans to be lists consisting of 0 and 1, in the case that i train and validate only the questions that their entity label, which is found from SPARQL queries [2,3,4,5,104,105], is part of their wording, the evaluation scores (f1,accuracy,recall,precision) [20], are a little bit better that the case of the same methodology in the case when i train and validate all the SimpleQuestions dataset questions [13], regardless if their entity label is part of their wording.

Instead from all these percentages (%) of totally correct answered questions and questions answered neither entirely correct nor entirely incorrect(tables 6.9-6.12) it is shown that when i train and validate all the SimpleQuestions dataset questions [13], regardless if their entity label is part of their wording, these percentages are generally higher than the case of train and validate only the questions that their entity label is part of their wording for the same methodology.

Finally it is occurred that when i train and validate the SimpleQuestions dataset (training, validation datasets) [13] that contain only answerable over wikidata questions the percentages (%) of totally correct answered questions are lower and the percentages (%) of questions that are answered neither totally correct nor totally incorrect are generally higher than in the case that i train and validate the full SimpleQuestions dataset (training, validation datasets) [13].

All these mentioned in the last two paragraphs could be explained by the fact that when i train and validate the full SimpleQuestions dataset [13] (including answerable, non-answerable questions over wikidata [14]), although there are questions in the dataset that could not be answered through wikidata [14], the bigger length of the dataset helps the neural network models to cope with other types of questions (with no answer over wikidata or generally a knowledge graph for example) and thus makes the QA engines (spacy library, jaccard similarity metric, Sentence Transfromer similarity model + cosine similarity metric) to give answers to questions different from the SimpleQuestions dataset (especially test dataset) [13]. Thus through SimpleQuestions dataset [13] that includes only answerable questions over wikidata [14], due to its lower length, all the aforementioned αρε impracticable to be carried out.

In addition all these show that the number of the 60 questions is a very indicative number in order to check if the QA engine models work well.

Based on the correct predictions or not relation id and entity id are predicted correctly and thus on the answers that are given for each question from QA engine, there are observed these types of questions:

1. Questions that are totally correct answered, with the entity id and the relation id are predicted correctly. An example of these questions is the question ”which company produced hot enough for june” which has answer ”Rank Organization”. This answer is the real and is proved to be the correct predicted for all the QA engines and all the methodologies. The real (and the correct predicted) relation ids and entity ids are P272 and Q12124859 correspondingly [26,27]. These entity id and the relation id predictions for this question are achieved either from the QA engine with spacy library [25,26] or the QA engine with cosine similarity metric [23,24] or the QA engine with jaccard similarity metric [23,24] for all the methodologies.

   This shows that it is very important when a user puts a question to the QA engine box, the wording of the question should be as simple as possible and have as little and concise information as possible. When this happens it is more possible to have entirely correct answers.

2. Questions that are totally correct answered, with one of the entity id and relation id to be predicted correctly. An example of these questions is the question ”which was the country of citizehship of christopher robinson” ,which has answer ”Canada”, for the QA engine that include jaccard similarity metric [23,24] through the best parameters of the methodology of finding entity span indexes (start, end) through cosine similarity metric [23,24] for the activation function ELU [43] in the case of training and validating the full training and validation SimpleQuestions training and validation dataset [13]. The real relation ids and entity ids are P27 and Q2966640 correspondingly. The real answer ”Canada” and the relation id P27 are predicted correctly [26,27]. The prediction of this entity id from the aformentioned QA engine and methodology is not the correct one compared with the real entity id Q2966640, because this prediction gives Q370264 as entity id [26,27].

3. Questions that have no answers, regardless if even one of the entity id and relation id were predicted correctly. An example is the question ”who is the composer of the song liar?” which for all the QA engines and all the methodologies has no answer. The real entity id, relation id and answer label are Q1049271,P86,”Freddie Mercury”.

   More specifically for the QA engine including the Sentence Transformer similarity model [21,22] and the cosine similarity metric through the best parameters of the methodology of finding entity span indexes (start, end) where the entity spans are lists consisting of 0 and 1 for all questions (regardless if the question has its entity label as part of its wording) for the activation function ELU [43] in the case of training and validating the training and validation SimpleQuestions training and validation subset [13] only with answerable over wikidata [14] questions, gives an incorrect prediction for the entity id: Q124707535 [26,27]. Also the same QA engine through the aforementioned model parameters predict correct the relation id:P86 [26,27]. The combination of this correctly predicted relation id and of the incorrectly predicted entity id gives no answers from the corresponding SPARQL query [2,3,4,5,104,105].

4. Questions that have answers that are assumed as totally wrong, because these answers have either small or no similarity with the real answer of their question, with none of the entity id and relation id to be predicted correctly.

   An example is the question ”what alternative rock band from Chicago is the author of of the blue colour of the sky” where for the QA engine including the Sentence Transformer similarity model [21,22] and the cosine similarity metric through the best parameters of the methodology of finding entity span indexes (start, end) where the entity spans are lists consisting of 0 and 1 for all questions (regardless if the question has its entity label as part of its wording) for the activation function Softplus [43] in the case of training and validating the full training and validation SimpleQuestions training and validation dataset [13] the answers that are given from the corresponding SPARQL query [2,3,4,5,104,105] (for example Linkin Park with cosine similairty score metric equal to 0.21), are assumed as wrong, because the entity id and the relation id are predicted incorrectly (Q11366 instead of Q2071360 and R136 instead of P175) and as a sequence the predicted entity label is not the expected (alternative rock instead of alternative rock band from Chicago) [26,27].

5. Questions that have answers that are assumed as neither totally correct not totally incorrect, where these answers have either significant or small similarity with the real answer of their question, regardless if the real answer is a part of the predicted answers and if at least one of the entity id and relation id to be predicted correctly.

   An example is the question ”An example is the question ”What is the name of a folk rock singer (or group)?” where for the QA engine including the Sentence Transformer similarity model [21,22] and the cosine similarity metric through the best parameters of the methodology of finding entity span indexes (start, end) where the entity spans are lists consisting of 0 and 1 for questions that have their entity label, which are found from SPARQL query [2,3,4,5,104,105], as part of their wording for the activation function ELU [43] in the case of training and validating the full training and validation SimpleQuestions training and validation dataset [13] the predicted entity id, relation id and answer are Q186472,R136,”Neil Young” respectively. The real entity entity id, relation id and answer are also the same. But also there are some other significant or a little similar answers with the real answer such as Paul Simon , with cosine similarity score with the expected answer equal to 0.53 [26,27].

   Another example for the QA engine including spacy library [25,26] and the same methodology is ”Who was born in dakar?” where from the corresponding SPARQL query [2,3,4,5,104,105], the answers that are occurred do not include the real answer, i.e ”Djibril Diawara”, although the predicted relation id and entity id are the same with the real ones, i.e R19 and Q3718 respectively. Another answer that is similar due to the common property of the town Dakar to be the birthplace (R19), is the answer ”Henri Saivet”.

Also i want to mention that when i referred to an answer or relation id or entity id of a question as real, i mean that i assumed the answers, relation ids and entity ids that are included for this question of ”Questions_testing_with_answers” .txt file as real.

For all these methodologies, although the equality or the similarity of the entity label with an part of their corresponding question, the prediction of the relation ids or the entity span indexes is based on indexes and not in a phrase or sentence embedding and as a result the entity span indexes mainly are not predicted always correctly. This fact, although the results are importantly reliable, could be a further obstacle to improving results of QA engines.

It is finally occurred that due to the fact that i have a different neural network for relation prediction and different for entity span prediction, although that the prediction scores are very high as number, it is possible when the relation id to be predicted right, the entity id is not predicted right and vice versa. This is a reason that some questions either are not answered or answered not right.

7. CONCLUSIONS AND FUTURE WORK

In this thesis i create three question answering engines the one with cosine similarity metric and Sentence Transformer, one with spacy library and the other with jaccard similarity metric. I create before two neural networks one for the relation prediction and one for the entity span prediction, which give very good predictions. These predictions helps the QA engines to predict generally satisfyingly the relation ids and the entity span indexes and as a result the entity labels for all the best parameters of the neural networks for each one of the methodologies either for SimpleQuestions dataset questions that are answerable over wikidata or for all SimpleQuestions dataset questions (entity span creation as a sequence of 0 and 1 either for all questions or only for questions having their entity label as part of their wording, using of spacy library or Sentence Transformers library combined with similarity metrics (cosine, jaccard)).

More specifically the best evaluation results (scores) for entity span prediction are achieved for the methodology of finding entity span indexes (start,end) by creating lists consisting of 0 and 1 only for SimpleQuestions training and validation dataset (for both the full dataset and the corresponding dataset that inlcudes only answerable over wikidata questions) questions that their entity label consists a part of their wording.

Also the biggest percentage of totally correct answered questions are observed for the QA engine with spacy library for the methodology of finding entity span indexes (start,end) through cosine similarity metric and the methodology of finding entity span indexes by creating lists consisting of 0 and 1 in the case of training and validating all SimpleQuestions training and validation dataset questions respectively, regardless if their entity label consists a part of their wording.

These results, although that are generally satisfactory, are based on indexes, i.e on numbers and not on sentence or phrase embeddings and it is proved a little bit difficult to have a much more bigger amount of correct predictions for relation indexes, but mainly for entity span indexes, than it is achieved in this thesis. This fact is a good change for the future to create neural networks that accept as entry phrase, word, sentence embeddings and explore ways to create question answering engines that their predictions will base on these neural network inputs. This work could help more the QA engines to understand better the semantics of an entity or a sentence.

About neural networks another future work to be not only to accept as input phrase or sentence embeddings, but also to create a neural network that predict both the relation index and as a result the relation id and the entity span indexes and so the entity labels. The main reason for this future work is that in this thesis i create two neural networks for relation and entity span prediction respectively and in some questions in QA engine the relation id could be predicted correctly, but not the entity label and vice-versa. This is a reason for some questions the QA engine gives wrong or no answers.

ABBREVIATIONS - ACRONYMS

RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
OWL	Web Ontology Language
OGC	Open Geospatial Consortium

APPENDIX A. FIRST APPENDIX

APPENDIX B. SECOND APPENDIX

APPENDIX C. THIRD APPENDIX

BIBLIOGRAPHY

1. https://bonndoc.ulb.uni-bonn.de/xmlui/bitstream/handle /20.500.11811/9810/6670.pdf ?sequence=5isAllowed=y
2. https://en.wikipedia.org/wiki/SPARQL
3. https://medium.com/wallscope/constructing-sparql-queries-ca63b8b9ac02
4. https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial
5. https://query.wikidata.org/
6. https://arxiv.org/pdf/2001.11985.pdf
7. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
8. Howard, J., Ruder, S.: Universal language model fine-tuning for text classification. In: Pro ceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Vol ume 1: Long Papers). pp. 328–339 (2018)
9. Liu, X., He, P., Chen, W., Gao, J.: Multi-task deep neural networks for natural language understanding. arXiv preprint arXiv:1901.11504 (2019)
10. Peters, M.E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., Zettlemoyer, L.: Deep contextualized word representations. In: Proc. of NAACL (2018)
11. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I.: Improving language understanding by generative pre-training transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
12. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners. Tech. rep.
13. https://github.com/askplatypus/wikidata-simplequestions
14. D.Vrandecic and M. Krötzsch, Wikidata: a free collaborative knowledgebase, Commun. ACM57(2014) 78, url: https://doi.org/10.1145/2629489.
15. https://huggingface.co/google-bert/bert-base-uncased
16. J. Devlin et al., “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), 2019 4171
17. https://huggingface.co/docs/transformers/index
18. C. Cortes and V. Vapnik, Support-vector networks, Machine learning 20 (1995) 273.

19. T. B. Brown et al., Language models are few-shot learners, arXiv preprint arXiv:2005.14165 (2020).
20. https://scikit-learn.org/stable /modules/classes.htmlmodule-sklearn.metrics
21. SentenceTransformers Documentation — Sentence-Transformers documentation
22. Pretrained Models — Sentence-Transformers documentation
23. Top 7 Ways To Implement Text Similarity In Python
24. Ultimate Guide To Text Similarity With Python - NewsCatcher
25. Python | Find all close matches of input string from a list - GeeksforGeeks
26. https://github.com/Themiscodes/Question-Answering-Transformers
27. https://github.com/DimOriCoding/Question-Answering-Engine-with-Transformers
28. https://docs.python.org/3/library/index.html
29. https://stackoverflow.com/questions/50168315/mounting-google-drive-on-google-colab
30. https://www.w3schools.com/python/numpy/numpy_intro.asp
31. https://www.w3schools.com/python/pandas/default.asp
32. https://huggingface.co/transformers/v3.0.2/model_doc/bert.htmloverview
33. https://huggingface.co/docs/transformers/model_doc/berttransformers.BertModel
34. https://huggingface.co/docs/transformers/model_doc/berttransformers.BertTokenizerFast
35. https://pytorch.org/docs/stable/tensor_attributes.htmltorch.device
36. https://www.w3schools.com/python/pandas/pandas_csv.asp
37. https://www.w3schools.com/python/pandas/ref_df_dropna.asp
38. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.to_list.html
39. https://www.geeksforgeeks.org/python-ways-to-concatenate-two-lists/
40. https://www.geeksforgeeks.org/python-list-index/
41. https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html
42. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html
43. https://pytorch.org/docs/stable/nn.functional.htmlnon-linear-activation-functions
44. https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html
45. https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html
46. https://developers.google.com/ machine-learning/ crash-course/multi-class-neural-networks/softmax
47. https://deepai.org/machine-learning-glossary-and-terms/softmax-layer

48. https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html
49. https://www.geeksforgeeks.org/python-string-split/
50. https://pythonbasics.org/join/
51. https://pytorch.org/tutorials/beginner/data_loading_tutorial.html
52. https://huggingface.co/transformers/v4.8.2/internal/tokenization_utils.htmltransformers.
tokenization_utils_base.PreTrainedTokenizerBase.encode_plus
53. https://pytorch.org/docs/stable/generated/torch.Generator.html
54. https://darinabal.medium.com/deep-learning-reproducible-results-using-pytorch-42034da5ad7
55. https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html
56. https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html
57. https://huggingface.co/docs/transformers/main_classes/optimizer_schedulesoptimization
58. https://machinelearningknowledge.ai/pytorch-optimizers-complete-guide-for-beginner/
59. https://machinelearningmastery.com/using-optimizers-from-pytorch/
60. https://pytorch.org/docs/stable/optim.html
61. https://pytorch.org/docs/stable/generated/torch.optim.Adam.html
62. https://pythonguides.com/adam-optimizer-pytorch/
63. https://huggingface.co/docs/transformers /main_classes/ optimizer_schedulestransformers
.SchedulerType
64. https://towardsdatascience.com/https-medium-com-dashingaditya-rakhecha-understanding-
learning-rate-dd5da26bb6de
65. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-
learning-neural-networks/
66. https://neptune.ai/blog/pytorch-loss-functions
67. https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html
68. https://pytorch.org/docs/stable/generated/torch.argmax.html
69. https://pythonguides.com/pytorch-early-stopping/
70. https://debuggercafe.com/using-learning-rate-scheduler-and-early-stopping-with-pytorch/
71. https://clay-atlas.com/us/blog/2021/08/25/pytorch-en-early-stopping/
72. https://www.techtarget.com/searchapparchitecture/definition/Resource-Description-
Framework-RDF
73. https://www.baeldung.com/cs/rdf-intro
74. https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf/

75. P. N. Mendes et al., “DBpedia spotlight: shedding light on the web of documents,” Proceedings of the 7th international conference on semantic systems, ACM, 2011 1.

76. J. Lehmann et al., DBpedia- A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia, Semantic Web Journal 6 (2015) 167, Outstanding Paper Award (Best 2014 SWJ Paper).

77. https://cambridgesemantics.com/blog/semantic-university/intro-semantic-web/intro-linked-data/

78. https://www.ontotext.com/knowledgehub/fundamentals/linked-data-linked-open-data/

79. https://www.wikipedia.org/

80. Y. Yang and M.-W. Chang, S-mart: Novel tree-based structured learning algorithms applied to tweet entity linking, arXiv preprint arXiv:1609.08075 (2016)

81. K.Bollacker et al., “Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge,” Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD’08, ACM, 2008 1247, isbn: 978-1-60558-102-6, url: http://doi.acm.org/10.1145/1376616.1376746.

82. https://cgi.di.uoa.gr/ koubarak/publications/2023/GeoKG___GeoAI_2023_GeoQA2-3.pdf

83. https://dl.acm.org/doi/10.1145/3281354.3281362

84. https://link.springer.com/chapter/10.1007/978-3-031-47243-5_15

85. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems. pp. 5998–6008 (2017)

86. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 (2014)

87. https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network

88. https://builtin.com/machine-learning/relu-activation-function

89. M.Schuster and K. Nakajima, “Japanese and korean voice search,” 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2012 5149

90. Wu, Y., Schuster, M., Chen, Z., Le, Q.V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al.: Google’s neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144 (2016)

91. S. Mohammed et al., “Strong Baselines for Simple Question Answering over Knowledge Graphs with and without Neural Networks,” Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers), vol. 2, 2018 291.

92. M. Petrochuk and L. Zettlemoyer, SimpleQuestions Nearly Solved: A New Upper-bound and Baseline Approach, arXiv preprint arXiv:1804.08798 (2018)

93. S. Hochreiter and J. Schmidhuber, Long short-term memory, Neural computation 9 (1997) 1735

94. K. Cho et al., "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, 2014 1724, url: http://aclweb.org/anthology/D/D14/D14-1179.pdf

95. Pellissier Tanon, T., Vrandecic, D., Schaffert, S., Steiner, T., Pintscher, L.: From Freebase to Wikidata: The great migration. In: Proc. of WWW. pp. 14191428 (2016)

96. A. Bordes et al., Large-scale simple question answering with memory networks, arXiv preprint arXiv:1506.02075 (2015)

97. He, X., Golub, D.: Character-level question answering with attention. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. pp. 1598–1607 (2016)

98. Lukovnikov, D., Fischer, A., Lehmann, J., Auer, S.: Neural network-based question answer ing over knowledge graphs on word and character level. In: Proceedings of the 26th inter national conference on World Wide Web. pp. 1211–1220. International World Wide Web Conferences Steering Committee (2017)

99. Yu, M., Yin, W., Hasan, K.S., dos Santos, C., Xiang, B., Zhou, B.: Improved neural relation detection for knowledge base question answering. In: Proceedings of the 55th Annual Meet ing of the Association for Computational Linguistics (Volume 1: Long Papers). vol. 1, pp. 571–581 (2017)

100. Dai, Z., Li, L., Xu, W.: Cfo: Conditional focused neural question answering with large scale knowledge bases. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). vol. 1, pp. 800–810 (2016)

101. Yin, W., Yu, M., Xiang, B., Zhou, B., Sch¨ utze, H.: Simple question answering by atten tive convolutional neural network. In: Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers. pp. 1746–1756 (2016)

102. https://ceur-ws.org/Vol-1963/paper555.pdf

103. https://www.w3schools.com/python/pandas/ref_df_drop.asp

104. https://sparqlwrapper.readthedocs.io/en/latest/SPARQLWrapper.Wrapper.html

105. https://people.wikimedia.org/ bearloga/notes/wdqs-python.html

106. https://www.geeksforgeeks.org/reindexing-in-pandas-dataframe/

107. https://www.geeksforgeeks.org/saving-a-pandas-dataframe-as-a-csv/

108. https://opendata.stackexchange.com/questions/9685/get-qid-from-wikidata-label-name-via-sparql