ÇUKUROVA UNIVERSITY

ENGINEERING AND ARCHITECTURE FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

GRADUATION THESIS

AUTOMATIC SHORT ANSWER GRADING

By
2016555060 Nuri Özgür Sarıgöz

Advisor
Doç.Dr. Umut Orhan

ADANA

ABSTRACT

This study aims to create a application that automatically grades the short answers given to a question with a referenced answer. The similarity between the answer given to the question by the examinee and the reference answer decided by the instructor are calculated by different language processing methods. Those methods are used to find the similarity values of the reference answers set for a collection example questions and student answers given to the questions which are marked as correct or incorrect before-hand and to create a dataset. After that, this dataset is used to evaluate new answers. The similarity values of the new answer and the reference answer is calculated and compared with the ones in the dataset, A model predicts if the answer is correct or not by checking the correctness of the answers with the similar similarity values in the dataset. Lastly, the answer is scored depending on its probability of being correct, decided by the model.

CONTENTS

DEFINITIONS

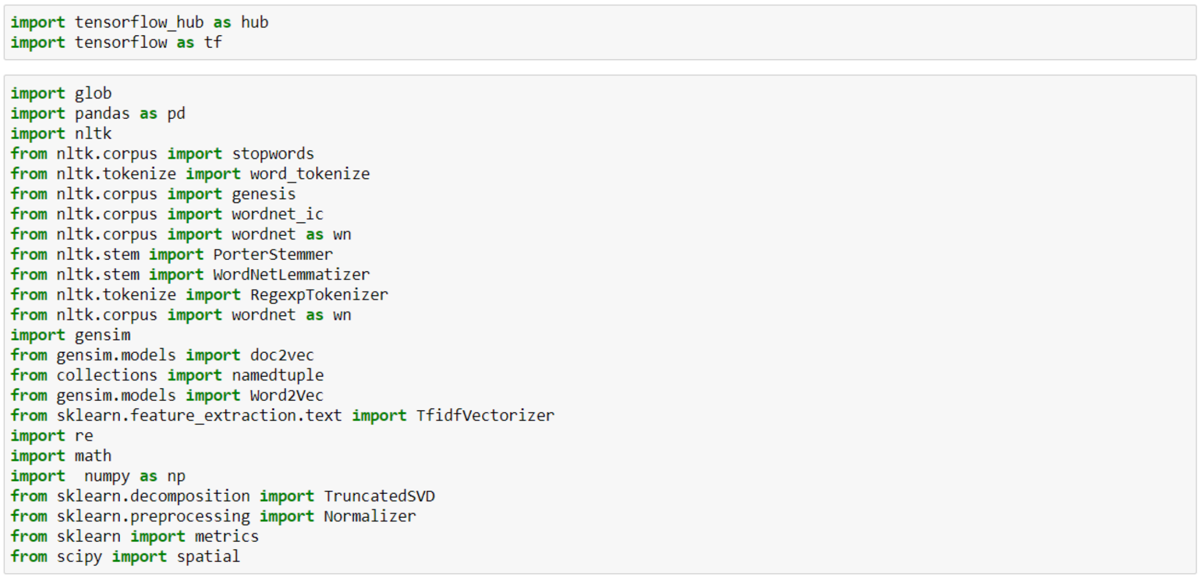
student answer: The answer given by the person who the question has been directed.

reference answer: The answer that has been decided to be the correct answer of the question by the person who directs to question.

LIST OF FIGURES

1. Preparation

1.1 Libraries



```python
import tensorflow_hub as hub
import tensorflow as tf
```

```python
import glob
import pandas as pd
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.corpus import genesis
from nltk.corpus import wordnet_ic
from nltk.corpus import wordnet as wn
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import wordnet as wn
import gensim
from gensim.models import doc2vec
from collections import namedtuple
from gensim.models import Word2Vec
from sklearn.feature_extraction.text import TfidfVectorizer
import re
import math
import  numpy as np
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import Normalizer
from sklearn import metrics
from scipy import spatial
```

Figure 1: Libraries

1. TensorFlow: Contains the necessary functions for processing the models used in the application.
2. Glob: Contains glob() function which provides easy access to saved files.
3. Pandas: Used for reading and creating data tables.
4. NLTK: The primary tool kit for linguistics functions and text processing.
5. Gensim: Contains Doc2Vec function, which is used for model training.
6. Collections: Used for creating tuples with named fields.
7. Scikit-Learn: Contains the functions used for machine learning.
8. RE: Provides regular expression matching.
9. Math: Provides access to the mathematical functions.
10. Numpy: Used for creating necessary large arrays.
11. SciPy: Contains the functions used for computations on matrices.

1.2 Models

1.2.1 Universel Sentence Encoder

The Universal Sentence Encoder encodes text into high-dimensional vectors that can be used for text classification, semantic similarity, clustering and other natural language tasks.

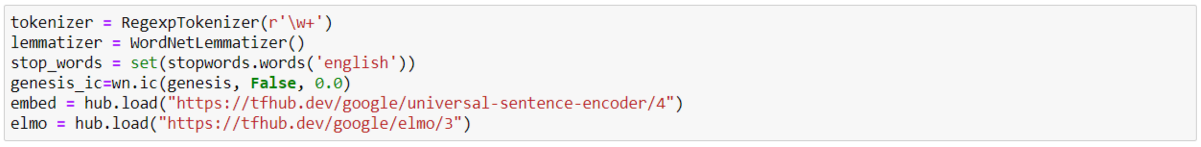
The model is trained and optimized for greater-than-word length text, such as sentences, phrases or short paragraphs. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide variety of natural language understanding tasks. The input is variable length English text and the output is a 512 dimensional vector. The model is trained with a deep averaging network (DAN) encoder. In this project, it is used for semantic similarity calculations.

1.2.2 Elmo Module

The Elmo Module computes contextualized word representations using character-based word representations and bidirectional LSTMs.

This module supports inputs both in the form of raw text strings or tokenized text strings. The module outputs fixed embeddings at each LSTM layer, a learnable aggregation of the 3 layers, and a fixed mean-pooled vector representation of the input. In this project, it is used for word embedding.

1.3 Text Processing Tools



```
tokenizer = RegexpTokenizer(r'\w+')
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
genesis_ic=wn.ic(genesis, False, 0.0)
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4")
elmo = hub.load("https://tfhub.dev/google/elmo/3")
```

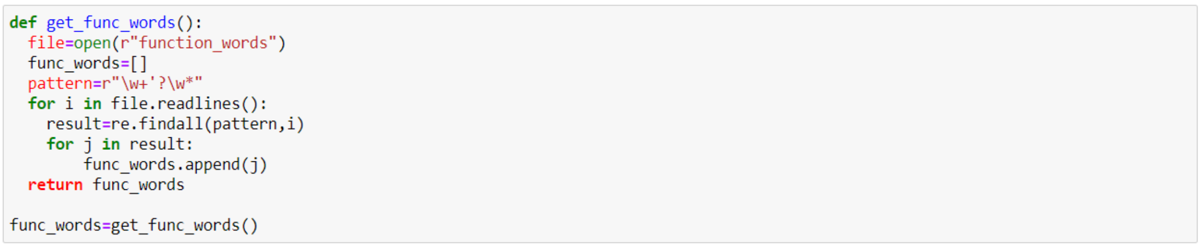
Figure 2 Models and NLTK Functions

1. Regexp Tokenizer: A tokenizer that splits a string using a regular expression, which matches either the tokens or the separators between tokens.
2. WordNet Lemmatizer: Lemmatizes the text, using WordNet's built-in morphy function. Returns the input word unchanged if it cannot be found in WordNet.
3. stop_words: Provides a list of ‘stop words’ for the selected language which are commonly ignored in text processing.
4. Genesis: A corpus provided by WordNet.
5. wordnet_ic: Creates an information content dictionary from a corpus.

2. Features Module

2.1 Text Processing & Analyzing Functions

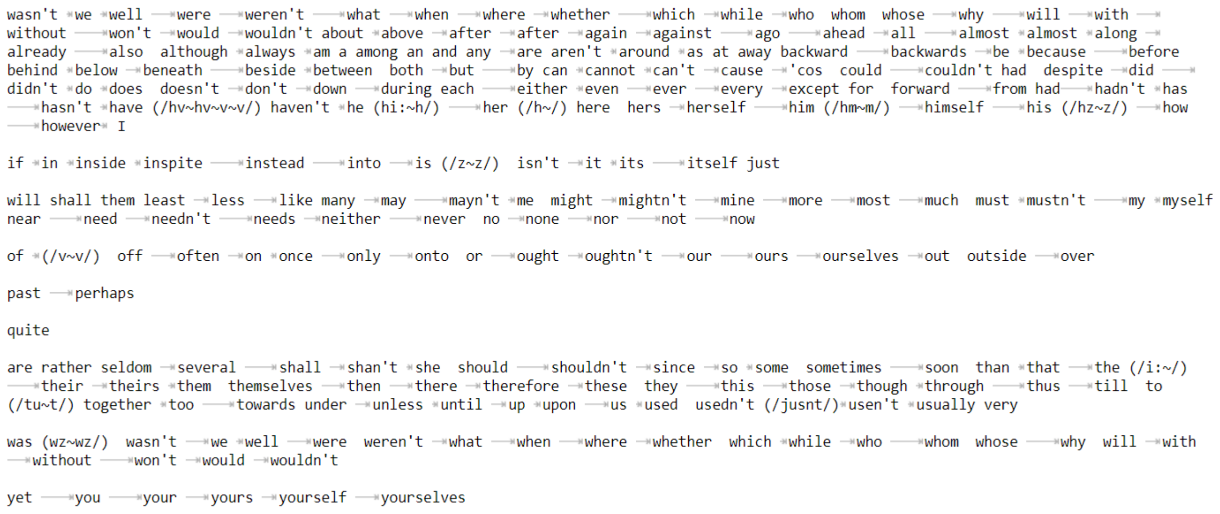
2.1.1 get_func_words



```
def get_func_words():
  file=open(r"function_words")
  func_words=[]
  pattern=r"\w+'?\w*"
  for i in file.readlines():
    result=re.findall(pattern,i)
    for j in result:
        func_words.append(j)
  return func_words

func_words=get_func_words()
```

Figure 3: Code for 'get_func_words' function

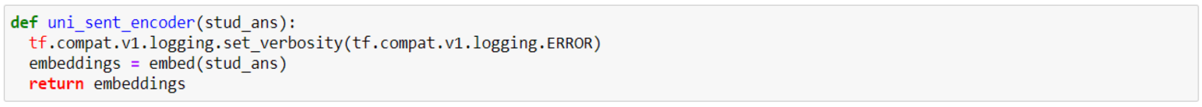
In linguistics, function words (also called functors) are words that have little lexical meaning or have ambiguous meaning and express grammatical relationships among other words within a sentence, or specify the attitude or mood of the speaker. This words are needed be omitted during the lexical analysis. This function uses a file named 'function_words' as the reference for which English words are function words. The file extracts the words from the file by a pattern recognition loop and saves them into a list called 'func_words'.



```
wasn't	we	well	were	weren't	what	when	where	whether	which	while	who	whom	whose	why	will	with	
without	won't	would	wouldn't about	above	after	after	again	against	ago	ahead	all	almost	almost	along	
already	also	although	always	am a among an and any	are aren't	around	as at away backward	backwards	be	because	before
behind	below	beneath	beside	between	both	but	by can	cannot	can't	cause	'cos	could	couldn't had	despite	did	
didn't	do	does	doesn't	don't	down	during each	either	even	ever	every	except for	forward	from had	hadn't	has
	hasn't	have (/hv~hv~v~v/) haven't	he (hi:~h/)	her (/h~/) here	hers	herself	him (/hm~m/)	himself	his (/hz~z/)	how
	however	I

if	in	inside	inspite	instead	into	is (/z~z/)	isn't	it	its	itself just

will shall them least	less	like many	may	mayn't	me	might	mightn't	mine	more	most	much	must	mustn't	my	myself
near	need	needn't	needs	neither	never	no	none	nor	not	now

of	(/v~v/)	off	often	on	once	only	onto	or	ought	oughtn't	our	ours	ourselves	out	outside	over

past	perhaps

quite

are rather seldom	several	shall	shan't	she	should	shouldn't	since	so	some	sometimes	soon	than	that	the (/i:~/)
	their	theirs	them	themselves	then	there	therefore	these	they	this	those	though	through	thus	till	to
(/tu~t/) together	too	towards under	unless	until	up	upon	us	used	usedn't (/jusnt/)	usen't	usually very

was (wz~wz/)	wasn't	we	well	were	weren't	what	when	where	whether	which	while	who	whom	whose	why	will	with
	without	won't	would	wouldn't

yet	you	your	yours	yourself	yourselves
```

Figure 4: Content of 'func_words' file

2.1.2 uni_sent_encoder

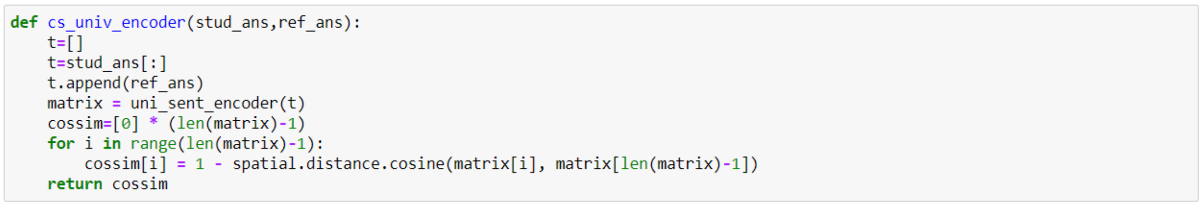


```python
def uni_sent_encoder(stud_ans):
  tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
  embeddings = embed(stud_ans)
  return embeddings
```

Figure 5: Code for 'uni_sent_encoder' function

This function simply gets the embedding of the answer given by the student by using Universal Sentence Encoder. Additionally, it reduces to logging output by limiting it to error messages, increasing the performance of the application.

2.1.3 cs_univ_encoder

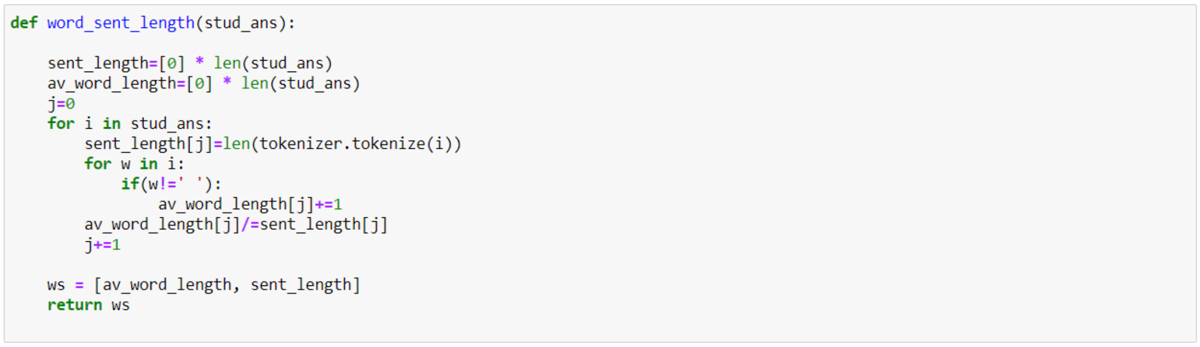


```python
def cs_univ_encoder(stud_ans,ref_ans):
    t=[]
    t=stud_ans[:]
    t.append(ref_ans)
    matrix = uni_sent_encoder(t)
    cossim=[0] * (len(matrix)-1)
    for i in range(len(matrix)-1):
        cossim[i] = 1 - spatial.distance.cosine(matrix[i], matrix[len(matrix)-1])
    return cossim
```

Figure 6: Code for 'cs_univ_encoder' function

Cosine similarity measures the similarity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction. When plotted on a multi-dimensional space, where each dimension corresponds to a word in a text, the cosine similarity can capture the orientation (the angle) of different texts; hence can be used to measure their similarity. This function finds the embedding of the student answer and the reference answer by using 'uni_sent_encoder' function. After that saves the results into matrices. Then calculates the cosine similarity between them by using 'spatial.distance.cosine' function, defined in SciPy library.

2.1.4 word_sent_length

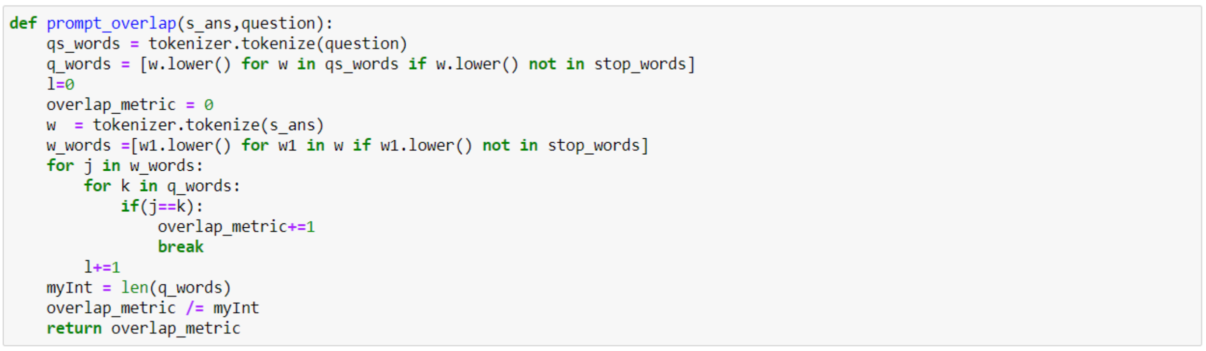


```python
def word_sent_length(stud_ans):

    sent_length=[0] * len(stud_ans)
    av_word_length=[0] * len(stud_ans)
    j=0
    for i in stud_ans:
        sent_length[j]=len(tokenizer.tokenize(i))
        for w in i:
            if(w!=' '):
                av_word_length[j]+=1
        av_word_length[j]/=sent_length[j]
        j+=1

    ws = [av_word_length, sent_length]
    return ws
```

Figure 7: Code for 'word_sent_length' function

This function calculates the sentence length and the average word length of the student answer by counting the spaces.

2.1.5 prompt_overlap

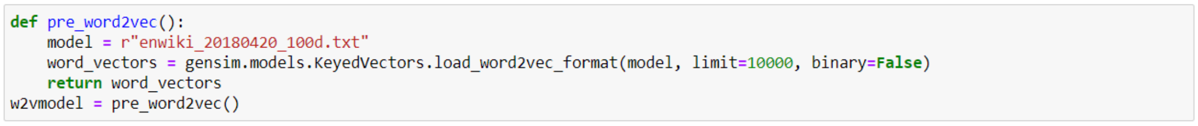


```
def prompt_overlap(s_ans,question):
    qs_words = tokenizer.tokenize(question)
    q_words = [w.lower() for w in qs_words if w.lower() not in stop_words]
    l=0
    overlap_metric = 0
    w  = tokenizer.tokenize(s_ans)
    w_words =[w1.lower() for w1 in w if w1.lower() not in stop_words]
    for j in w_words:
        for k in q_words:
            if(j==k):
                overlap_metric+=1
                break
        l+=1
    myInt = len(q_words)
    overlap_metric /= myInt
    return overlap_metric
```

Figure 8: Code for 'prompt_overlap' function

Another way of measuring the similarity between two sentences is to look for overlapping words. This function eliminates stop words from the student answer and the question, and turns them into lower-case. Then it checks if any word has been used in both sentences. It increases the value of ‘overlap_metric’ by 1 for each shared word between the two sentences, then divides it with the number of words in the question, returning it as the final value.

2.1.6 pre_word2vec

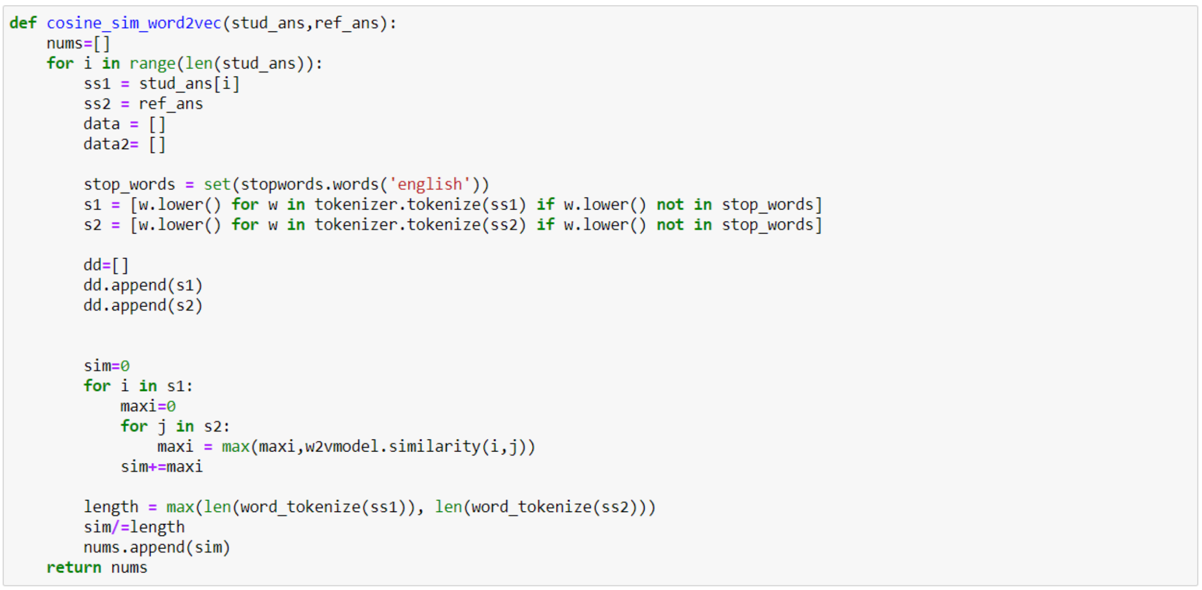


```
def pre_word2vec():
    model = r"enwiki_20180420_100d.txt"
    word_vectors = gensim.models.KeyedVectors.load_word2vec_format(model, limit=10000, binary=False)
    return word_vectors
w2vmodel = pre_word2vec()
```

Figure 9: Code for 'pre_word2vec' function

This function extracts the word vectors from a text file. ‘KeyedVectors.load_word2vec_format’ is a function, defined in Gensim library, which instantiates the vectors from an existing file as KeyedVectors instance (each vector is identified by a key). The model used for this function is provided by Wikipedia2Vec and contains large amount of pre-trained embedding in text format. The ‘limit’ variable in extraction function is used for reducing the quantity of vectors in order to match the hardware limitations during the development process. However, limiting the word vectors causes inconsistencies in the application. Therefore, final version of the application uses the complete collection of the word vectors.

2.1.7 cosine_sim_word2vec

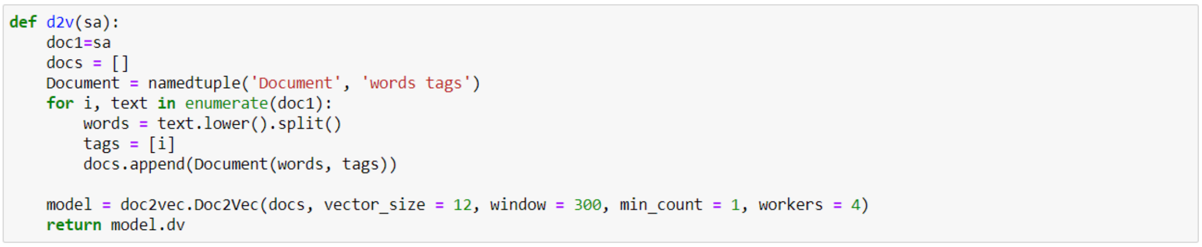


```
def cosine_sim_word2vec(stud_ans,ref_ans):
    nums=[]
    for i in range(len(stud_ans)):
        ss1 = stud_ans[i]
        ss2 = ref_ans
        data = []
        data2= []

        stop_words = set(stopwords.words('english'))
        s1 = [w.lower() for w in tokenizer.tokenize(ss1) if w.lower() not in stop_words]
        s2 = [w.lower() for w in tokenizer.tokenize(ss2) if w.lower() not in stop_words]

        dd=[]
        dd.append(s1)
        dd.append(s2)


        sim=0
        for i in s1:
            maxi=0
            for j in s2:
                maxi = max(maxi,w2vmodel.similarity(i,j))
            sim+=maxi

        length = max(len(word_tokenize(ss1)), len(word_tokenize(ss2)))
        sim/=length
        nums.append(sim)
    return nums
```

Figure 10: Code for 'cosine_sim_word2vec' function

This function calculates the cosine similarity between the student answer and the reference answer by using the same method as 'cs_univ_encoder' function. In contrast to 'cs_univ_encoder' function, this function uses the word vectors obtained from 'pre_word2vec' as reference point instead of Universal Sentence Encoder. The function clears the stop words and turns all the characters into lower case. Then compares two sentences word-by-word, using the Word2Vec model.

2.1.8 d2v

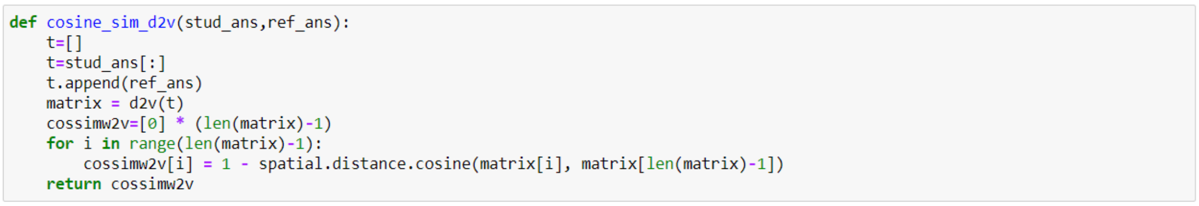


```
def d2v(sa):
    doc1=sa
    docs = []
    Document = namedtuple('Document', 'words tags')
    for i, text in enumerate(doc1):
        words = text.lower().split()
        tags = [i]
        docs.append(Document(words, tags))

    model = doc2vec.Doc2Vec(docs, vector_size = 12, window = 300, min_count = 1, workers = 4)
    return model.dv
```

Figure 11: Code for 'd2v' function

This function creates a class named 'Document' with two named tuples, 'words' and 'tag'. 'words' stores the each word of the student answer and the reference answer while 'tag' is used to differentiate the student answer from the reference answer. Then it creates and trains a model using the 'Document' class by using 'doc2vec.Doc2Vec' from Gensim library. 'vector_size' sets the dimensionality of the feature vectors. 'window' variable is the maximum distance between the current and predicted word within a sentence. 'min_count' is the lowest total frequency for a word to be take into account and 'workers' is the number of worker threads to train the model.

2.1.9 cosine_sim_d2v

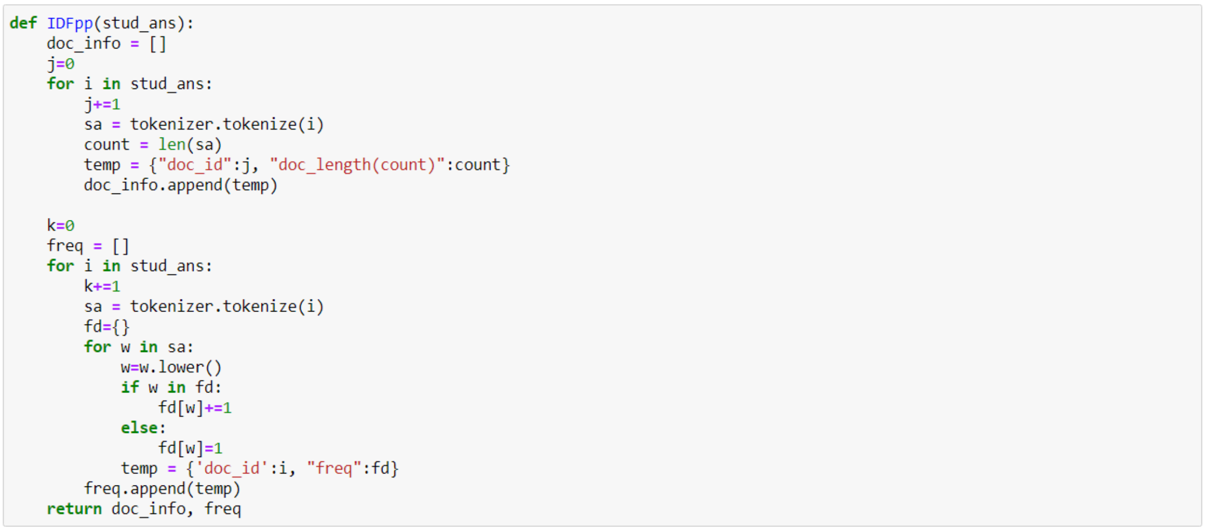


```
def cosine_sim_d2v(stud_ans,ref_ans):
    t=[]
    t=stud_ans[:]
    t.append(ref_ans)
    matrix = d2v(t)
    cossimw2v=[0] * (len(matrix)-1)
    for i in range(len(matrix)-1):
        cossimw2v[i] = 1 - spatial.distance.cosine(matrix[i], matrix[len(matrix)-1])
    return cossimw2v
```

Figure 12 Code for 'cosine_sim_d2v' function

Another function that calculates the cosine similarity between the student answer and the reference answer. This time, the function utilizes the Doc2Vec approach, by using the model built in d2v function. Similar to ‘cs_univ_encoder’ function, this function saves the words and their embedding into a matrix to calculate their cosine similarity with ‘spatial.distance.cosine’ function from SciPy library.

2.1.10 IDFpp

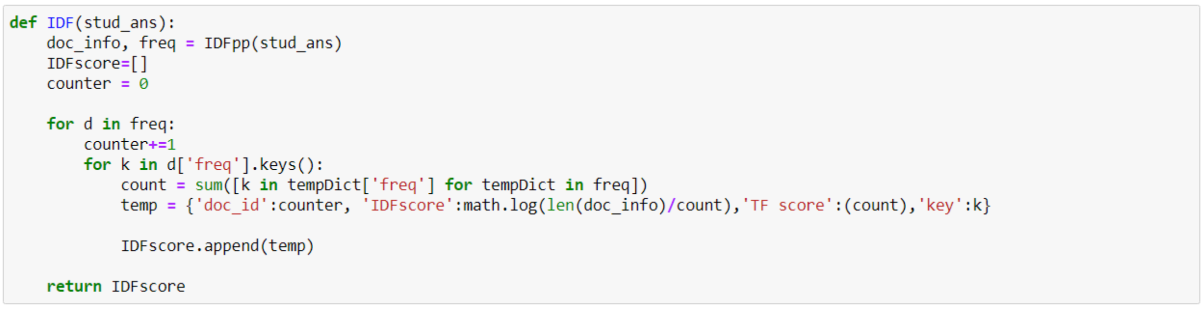


```
def IDFpp(stud_ans):
    doc_info = []
    j=0
    for i in stud_ans:
        j+=1
        sa = tokenizer.tokenize(i)
        count = len(sa)
        temp = {"doc_id":j, "doc_length(count)":count}
        doc_info.append(temp)

    k=0
    freq = []
    for i in stud_ans:
        k+=1
        sa = tokenizer.tokenize(i)
        fd={}
        for w in sa:
            w=w.lower()
            if w in fd:
                fd[w]+=1
            else:
                fd[w]=1
            temp = {'doc_id':i, "freq":fd}
        freq.append(temp)
    return doc_info, freq
```

Figure 13: Code for 'IDFpp' function

This function creates and returns two classes. ‘doc_info’ and ‘freq’. ‘doc_info’ contains two lists called ‘doc_id’ and ‘doc_lenght’. ‘doc_lenght’ is the sentence length of the student answer while ‘doc_id’ is the ID index given to it by the function (for answers that contains multiple sentences). ‘freq’ class contains two lists as well, ‘doc_id’ and ‘freq’. For this clas, ‘doc_id’ hold the sentence(s) in the student answer and ‘freq’ records the frequency (how many time the word is repeated in the answer) of each word in the student answer. The functions returns

2.1.11 IDF



```python
def IDF(stud_ans):
    doc_info, freq = IDFpp(stud_ans)
    IDFscore=[]
    counter = 0

    for d in freq:
        counter+=1
        for k in d['freq'].keys():
            count = sum([k in tempDict['freq'] for tempDict in freq])
            temp = {'doc_id':counter, 'IDFscore':math.log(len(doc_info)/count),'TF score':(count),'key':k}

            IDFscore.append(temp)

    return IDFscore
```

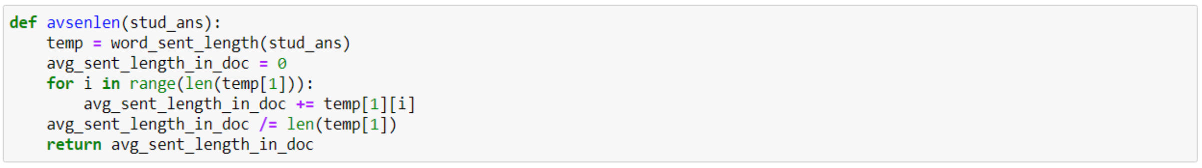
Figure 14: Code for 'IDF' function

In information retrieval, tf–idf, or TFIDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is calculated by multiplying two metrics: how many times a word appears in a document and the inverse document frequency of the word across a set of documents.

This function calculates IDF and TF scores of the words in the student answer. For TFIDF is a statistic meant to be used in corpora that contains multiple documents, the function requires a multiple-sentence student answers to be fully executed. Otherwise, IDF score will be evaluated as zero since there is no collection documents to calculate a frequency.

The function calls ‘IDFpp’ function to index words and sentences in the student answer and find the frequency of the words. It adds the frequency of the words between the sentences and saves it as TF score for each unique word. Then it calculates the IDF score of the by finding the natural logarithm of the sentence length, divided by the term frequency. Final scores are returned by the function in a list named ‘IDFscore’

2.1.12 avsenlen

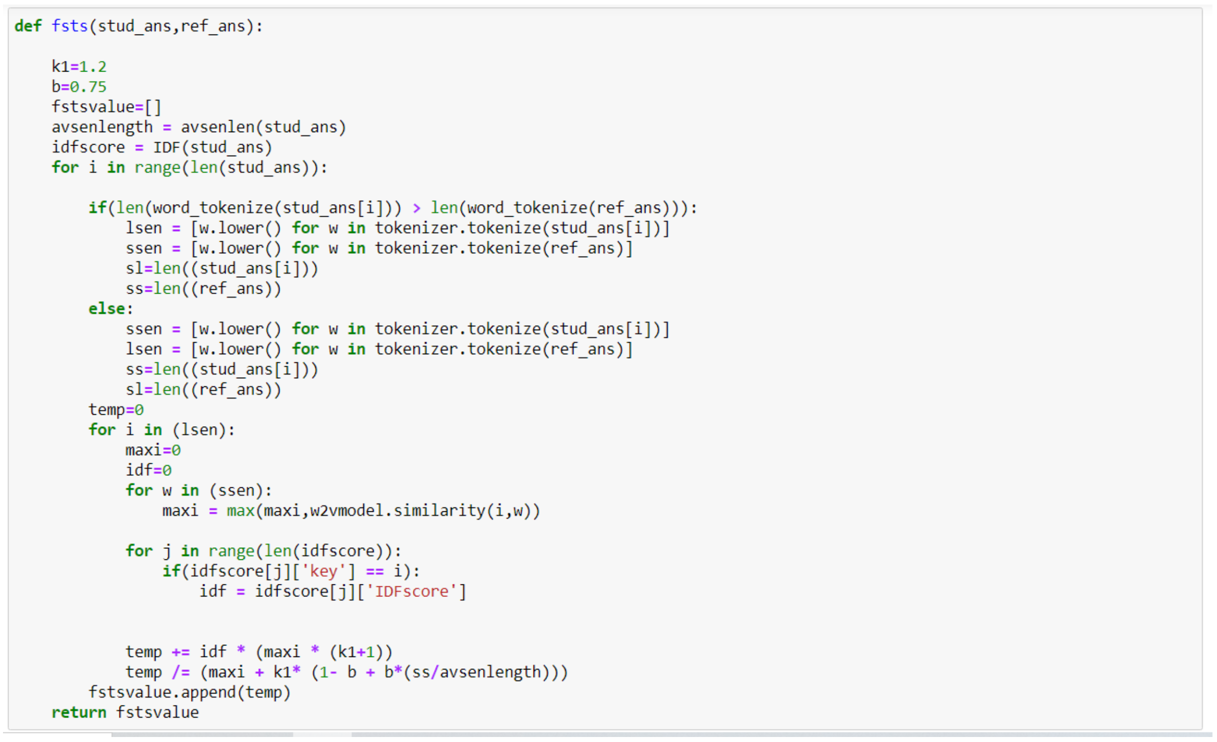


```python
def avsenlen(stud_ans):
    temp = word_sent_length(stud_ans)
    avg_sent_length_in_doc = 0
    for i in range(len(temp[1])):
        avg_sent_length_in_doc += temp[1][i]
    avg_sent_length_in_doc /= len(temp[1])
    return avg_sent_length_in_doc
```

Figure 15: Code for 'avsenlen' function

This function finds the average length of the sentences in a multiple-sentence student answer, separated into documents.

2.1.13 fsts

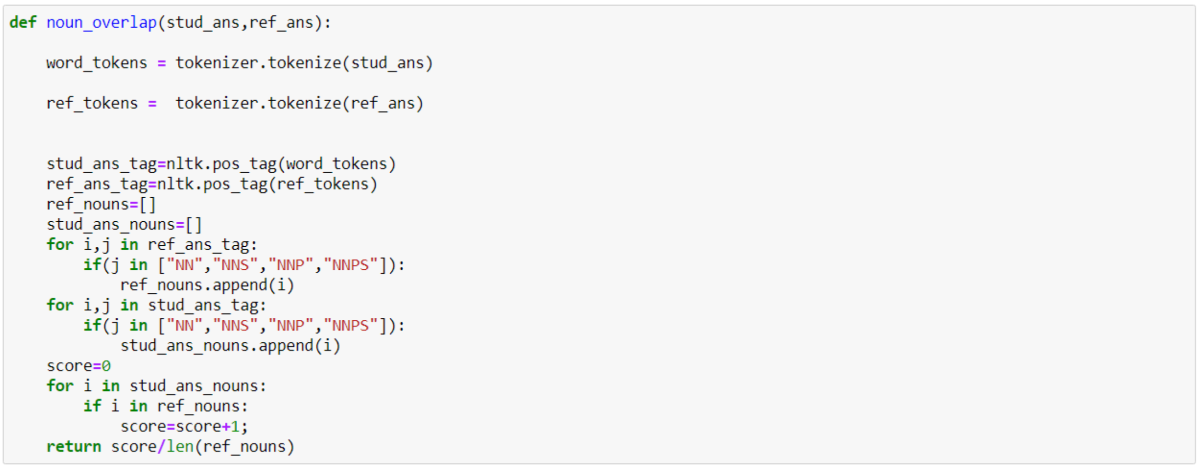


```
def fsts(stud_ans,ref_ans):

    k1=1.2
    b=0.75
    fstsvalue=[]
    avsenlength = avsenlen(stud_ans)
    idfscore = IDF(stud_ans)
    for i in range(len(stud_ans)):

        if(len(word_tokenize(stud_ans[i])) > len(word_tokenize(ref_ans))):
            lsen = [w.lower() for w in tokenizer.tokenize(stud_ans[i])]
            ssen = [w.lower() for w in tokenizer.tokenize(ref_ans)]
            sl=len((stud_ans[i]))
            ss=len((ref_ans))
        else:
            ssen = [w.lower() for w in tokenizer.tokenize(stud_ans[i])]
            lsen = [w.lower() for w in tokenizer.tokenize(ref_ans)]
            ss=len((stud_ans[i]))
            sl=len((ref_ans))
        temp=0
        for i in (lsen):
            maxi=0
            idf=0
            for w in (ssen):
                maxi = max(maxi,w2vmodel.similarity(i,w))

            for j in range(len(idfscore)):
                if(idfscore[j]['key'] == i):
                    idf = idfscore[j]['IDFscore']


            temp += idf * (maxi * (k1+1))
            temp /= (maxi + k1* (1- b + b*(ss/avsenlength)))
        fstsvalue.append(temp)
    return fstsvalue
```

Figure 16: code for 'fsts' function

This function find the similarity between the student answer and the reference answer by using tf-idf values, obtained from 'IDF' function. Firstly, the student answer and the reference answers are compared to find which one is longer. They are also tokenized (separated into words) and set to lower-case during the process. The longer one is set to 'lsen' variable while the short one is set to 'ssen' variable. Each word in 'lsen' is compared with each word from 'ssen' using 'similarity' function from Gensim library and 'w2vmodel' built in 'pre_word2vec' function. For each word combination, the value of the highest similarity score is assigned to 'maxi' variable, 'idfscore' value is assigned to 'idf' variable and 'fstsvalue' is calculated accordingly in a loop. The function returns the list of final values.

2.1.14 noun_overlap

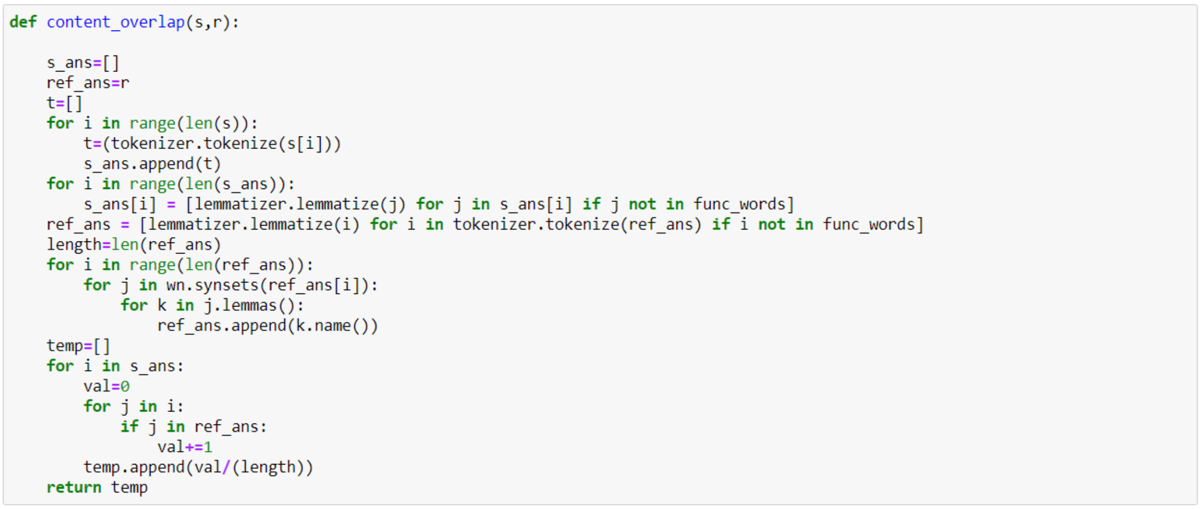


```python
def noun_overlap(stud_ans,ref_ans):

    word_tokens = tokenizer.tokenize(stud_ans)

    ref_tokens =  tokenizer.tokenize(ref_ans)


    stud_ans_tag=nltk.pos_tag(word_tokens)
    ref_ans_tag=nltk.pos_tag(ref_tokens)
    ref_nouns=[]
    stud_ans_nouns=[]
    for i,j in ref_ans_tag:
        if(j in ["NN","NNS","NNP","NNPS"]):
            ref_nouns.append(i)
    for i,j in stud_ans_tag:
        if(j in ["NN","NNS","NNP","NNPS"]):
            stud_ans_nouns.append(i)
    score=0
    for i in stud_ans_nouns:
        if i in ref_nouns:
            score=score+1;
    return score/len(ref_nouns)
```

Figure 17: Code for 'noun_overlap' function

This function, similar to ‘promp_overlap’ function, aims to find the identical words, but this time, between the student answer and the reference answer. Additionally, this function checks specifically noun words. After tokenizing both the student answer and the reference answer, the function marks up each word as corresponding to a particular part of speech (noun, verb, preposition etc.) by using ‘pos_tog’ function, provided by NL Toolkit. After that, the function removes words that are not noun and check if any noun is shared between the student answer and the reference answer. The function adds to score by one for each shared word and divides if by the number of nouns in the reference answers, returning it as the result.

2.1.15 content_overlap

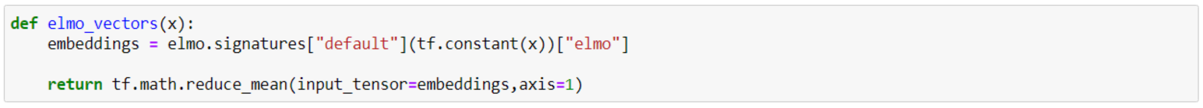


```python
def content_overlap(s,r):

    s_ans=[]
    ref_ans=r
    t=[]
    for i in range(len(s)):
        t=(tokenizer.tokenize(s[i]))
        s_ans.append(t)
    for i in range(len(s_ans)):
        s_ans[i] = [lemmatizer.lemmatize(j) for j in s_ans[i] if j not in func_words]
    ref_ans = [lemmatizer.lemmatize(i) for i in tokenizer.tokenize(ref_ans) if i not in func_words]
    length=len(ref_ans)
    for i in range(len(ref_ans)):
        for j in wn.synsets(ref_ans[i]):
            for k in j.lemmas():
                ref_ans.append(k.name())
    temp=[]
    for i in s_ans:
        val=0
        for j in i:
            if j in ref_ans:
                val+=1
        temp.append(val/(length))
    return temp
```

Figure 18: The code for 'content_overlap' function

The function checks if there are any words with the similar content in the student answer and the reference answers. The function tokenizes the the student answer. Then lemmatizes (turns all words into their base form) all words in the student answer and the reference answer that are not listed in ‘stop_words’ file. After that, for each word in the reference answer, their content-related words are found in WordNet database by using ‘synsets’, ‘lemmas’ and ‘name’ functions from NL Toolkit library and added to ‘ref_ans’ list. Lasty, the student answer is analyzed if there are any words in the student answer that also exist in the list of content related words to the reference answer. The number of shared words is divided by the length of the reference answer and returned as the final score.

2.1.16 elmo_vectors

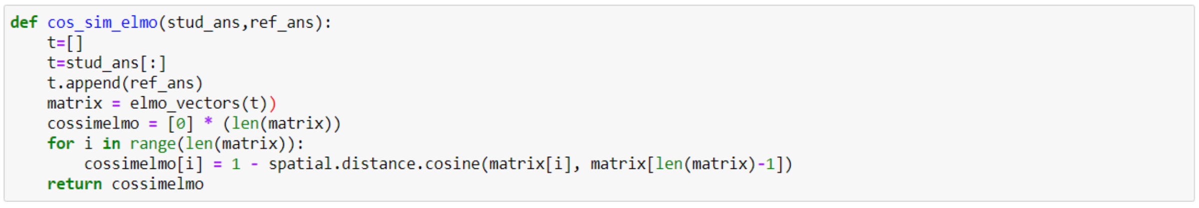


```
def elmo_vectors(x):
    embeddings = elmo.signatures["default"](tf.constant(x))["elmo"]

    return tf.math.reduce_mean(input_tensor=embeddings,axis=1)
```

Figure 19: Code for ‘elmo_vectors’

This function obtains the embedding of the student answer and turns them into a tensor by using ‘signatures’ function from TensorFlow library. Then it returns the average of the features by using another function from TensorFlow, ‘math.reduce_mean’.

2.1.17 cos_sim_elmo

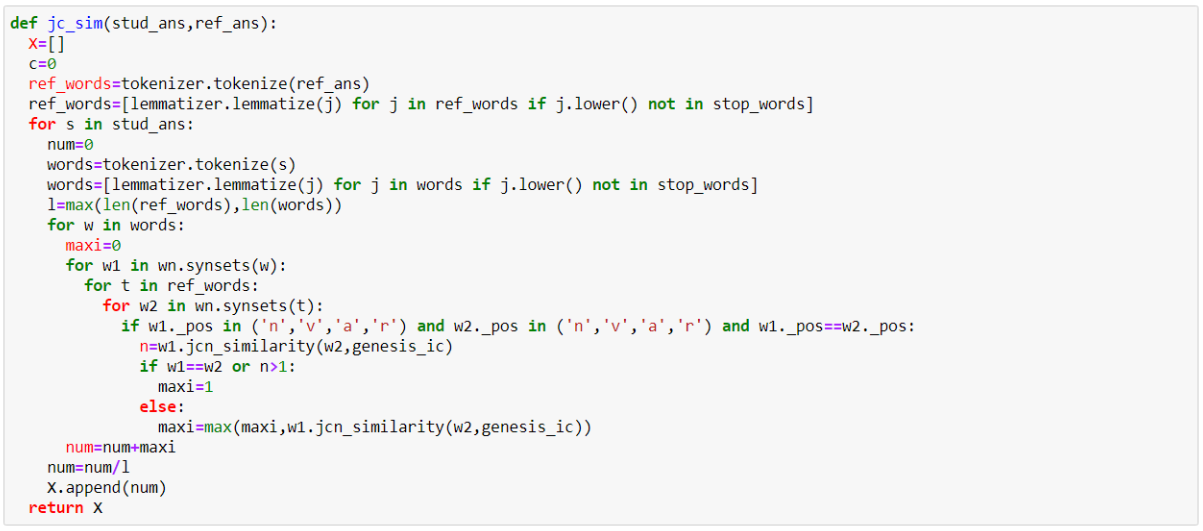


```
def cos_sim_elmo(stud_ans,ref_ans):
    t=[]
    t=stud_ans[:]
    t.append(ref_ans)
    matrix = elmo_vectors(t))
    cossimelmo = [0] * (len(matrix))
    for i in range(len(matrix)):
        cossimelmo[i] = 1 - spatial.distance.cosine(matrix[i], matrix[len(matrix)-1])
    return cossimelmo
```

Figure 20: Code for 'cos_sim_elmo' function

Another function that calculates cosine similarity between the student answer and the reference answer. This function uses the vectors received from ‘elmo_vectors’ function to make the calculation. The calculation method is the same as previous cosine similarity functions.

2.1.18 jc_sim

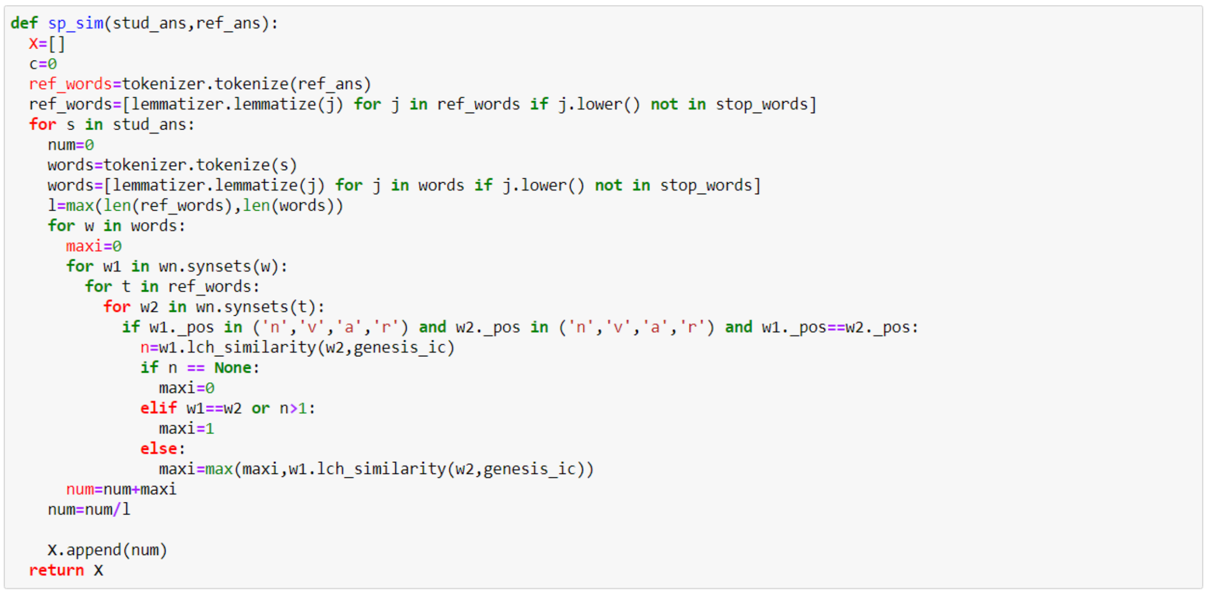


```
def jc_sim(stud_ans,ref_ans):
  X=[]
  c=0
  ref_words=tokenizer.tokenize(ref_ans)
  ref_words=[lemmatizer.lemmatize(j) for j in ref_words if j.lower() not in stop_words]
  for s in stud_ans:
    num=0
    words=tokenizer.tokenize(s)
    words=[lemmatizer.lemmatize(j) for j in words if j.lower() not in stop_words]
    l=max(len(ref_words),len(words))
    for w in words:
      maxi=0
      for w1 in wn.synsets(w):
        for t in ref_words:
          for w2 in wn.synsets(t):
            if w1._pos in ('n','v','a','r') and w2._pos in ('n','v','a','r') and w1._pos==w2._pos:
              n=w1.jcn_similarity(w2,genesis_ic)
              if w1==w2 or n>1:
                maxi=1
              else:
                maxi=max(maxi,w1.jcn_similarity(w2,genesis_ic))
      num=num+maxi
    num=num/l
    X.append(num)
  return X
```

Figure 21: Code for 'jc_sim' function

This function is used for calculating JCN similarity. The function tokenizes and lemmatizes the words from the reference answer and the student answer that are not listed in ‘stop_words’ file. Next, it selects a word from both answers and list their content related words, using ‘synets’ function. Then it compares words from both lists that are either noun, verb, adjective or adverb (identified with ‘_pos’ function) using ‘jcn_similarity’ function and ‘genesis_ic’ model. This process continues until all words from both lists are compared and their maximum similarity values are found. The values are stored in a list and returned by the function.

2.1.19 sp_sim

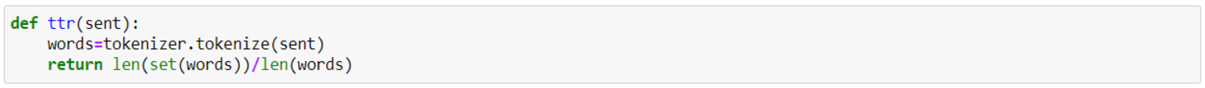


```
def sp_sim(stud_ans,ref_ans):
  X=[]
  c=0
  ref_words=tokenizer.tokenize(ref_ans)
  ref_words=[lemmatizer.lemmatize(j) for j in ref_words if j.lower() not in stop_words]
  for s in stud_ans:
    num=0
    words=tokenizer.tokenize(s)
    words=[lemmatizer.lemmatize(j) for j in words if j.lower() not in stop_words]
    l=max(len(ref_words),len(words))
    for w in words:
      maxi=0
      for w1 in wn.synsets(w):
        for t in ref_words:
          for w2 in wn.synsets(t):
            if w1._pos in ('n','v','a','r') and w2._pos in ('n','v','a','r') and w1._pos==w2._pos:
              n=w1.lch_similarity(w2,genesis_ic)
              if n == None:
                maxi=0
              elif w1==w2 or n>1:
                maxi=1
              else:
                maxi=max(maxi,w1.lch_similarity(w2,genesis_ic))
      num=num+maxi
    num=num/l

    X.append(num)
  return X
```

Figure 22: Code for 'sp_sim' function

This function is used for calculation of LCH similarity. The function is nearly identical to ‘jc_sim’ function. Only ‘jcn_similarity’ function is replaced with ‘lch_similarity’ function and an additional statement is added for zero case, because ‘lch_similarity’ functions returns *None* value when there is no similarity between the words.

2.1.20 ttr



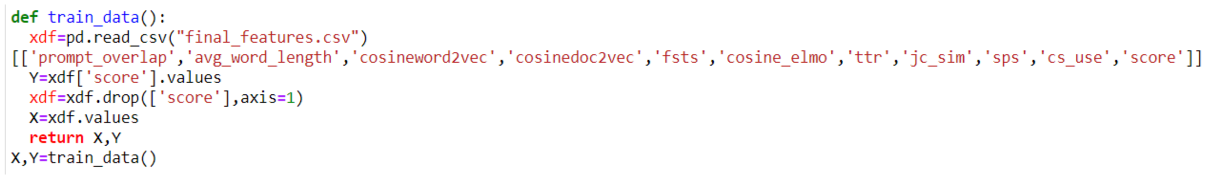
```
def ttr(sent):
    words=tokenizer.tokenize(sent)
    return len(set(words))/len(words)
```

Figure 23: Code for 'ttr' function

This function tokenizes the student answer, creates a set from its words and returns the length of the set divided by the length of the answer.

2.2 Modeling Functions

2.2.1 train_data

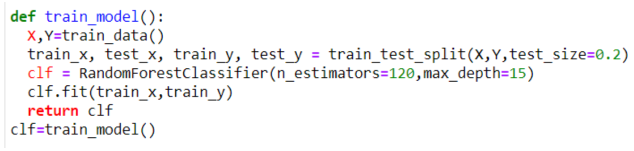


```
def train_data():
  xdf=pd.read_csv("final_features.csv")
[['prompt_overlap','avg_word_length','cosineword2vec','cosinedoc2vec','fsts','cosine_elmo','ttr','jc_sim','sps','cs_use','score']]
  Y=xdf['score'].values
  xdf=xdf.drop(['score'],axis=1)
  X=xdf.values
  return X,Y
X,Y=train_data()
```

Figure 24: Code for 'train_data' function

This function is used for training the data contained in ‘final_features.csv’ file. ‘final_features.csv’ contains the similarity scores of a collection of example questions, their reference answers and student answers. Each example has a binary ‘score’ index, which marks the student answers as correct (1) or incorrect (0). The ‘scores’ are separated from the main data in this function and saved as a new data.

2.2.2 train_model



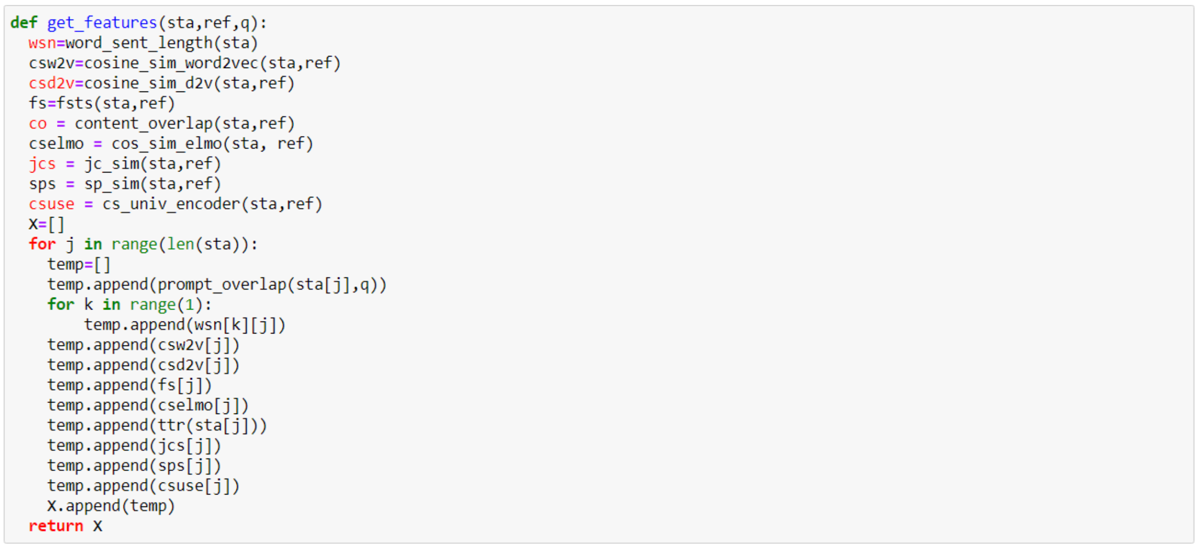
```
def train_model():
  X,Y=train_data()
  train_x, test_x, train_y, test_y = train_test_split(X,Y,test_size=0.2)
  clf = RandomForestClassifier(n_estimators=120,max_depth=15)
  clf.fit(train_x,train_y)
  return clf
clf=train_model()
```

Figure 25 Code for 'train_model' function

This function is used for training the scoring model. A test set is created for both X (similarity scores) and Y (scores) datasets as 20% of the train sets. ‘RandomForestClassifier’ function from Scikit-Learn library is used to improve the accuracy of the prediction made by the model.

2.3 Main Functions

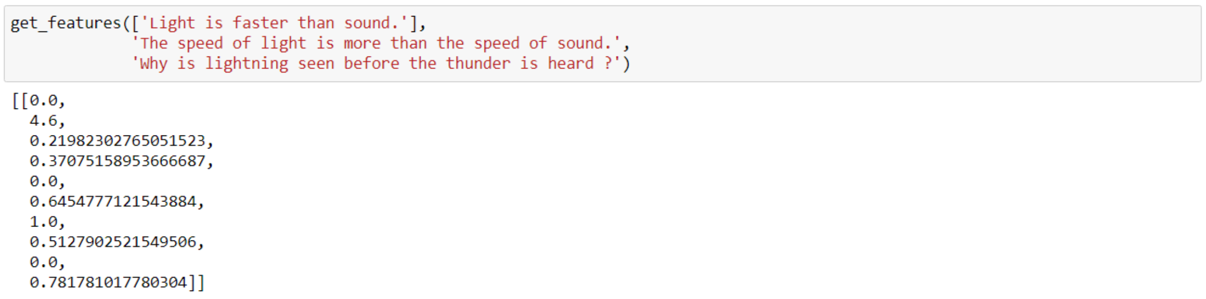
2.3.1 get_features



```
def get_features(sta,ref,q):
  wsn=word_sent_length(sta)
  csw2v=cosine_sim_word2vec(sta,ref)
  csd2v=cosine_sim_d2v(sta,ref)
  fs=fsts(sta,ref)
  co = content_overlap(sta,ref)
  cselmo = cos_sim_elmo(sta, ref)
  jcs = jc_sim(sta,ref)
  sps = sp_sim(sta,ref)
  csuse = cs_univ_encoder(sta,ref)
  X=[]
  for j in range(len(sta)):
    temp=[]
    temp.append(prompt_overlap(sta[j],q))
    for k in range(1):
        temp.append(wsn[k][j])
    temp.append(csw2v[j])
    temp.append(csd2v[j])
    temp.append(fs[j])
    temp.append(cselmo[j])
    temp.append(ttr(sta[j]))
    temp.append(jcs[j])
    temp.append(sps[j])
    temp.append(csuse[j])
    X.append(temp)
  return X
```

Figure 26: Code for 'get_features' function

This is the main function that executes all text processing and analyzing operations. After the execution, the similarity values obtained from each of the functions are assigned into a list and returned by the function.

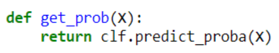


```
get_features(['Light is faster than sound.'],
             'The speed of light is more than the speed of sound.',
             'Why is lightning seen before the thunder is heard ?')
```

```
[[0.0,
  4.6,
  0.21982302765051523,
  0.37075158953666687,
  0.0,
  0.6454777121543884,
  1.0,
  0.5127902521549506,
  0.0,
  0.781781017780304]]
```

Figure 27: Example output of 'get_features' function, showing all similarity values, calculated by the text analyzing functions

2.3.2 get_prob



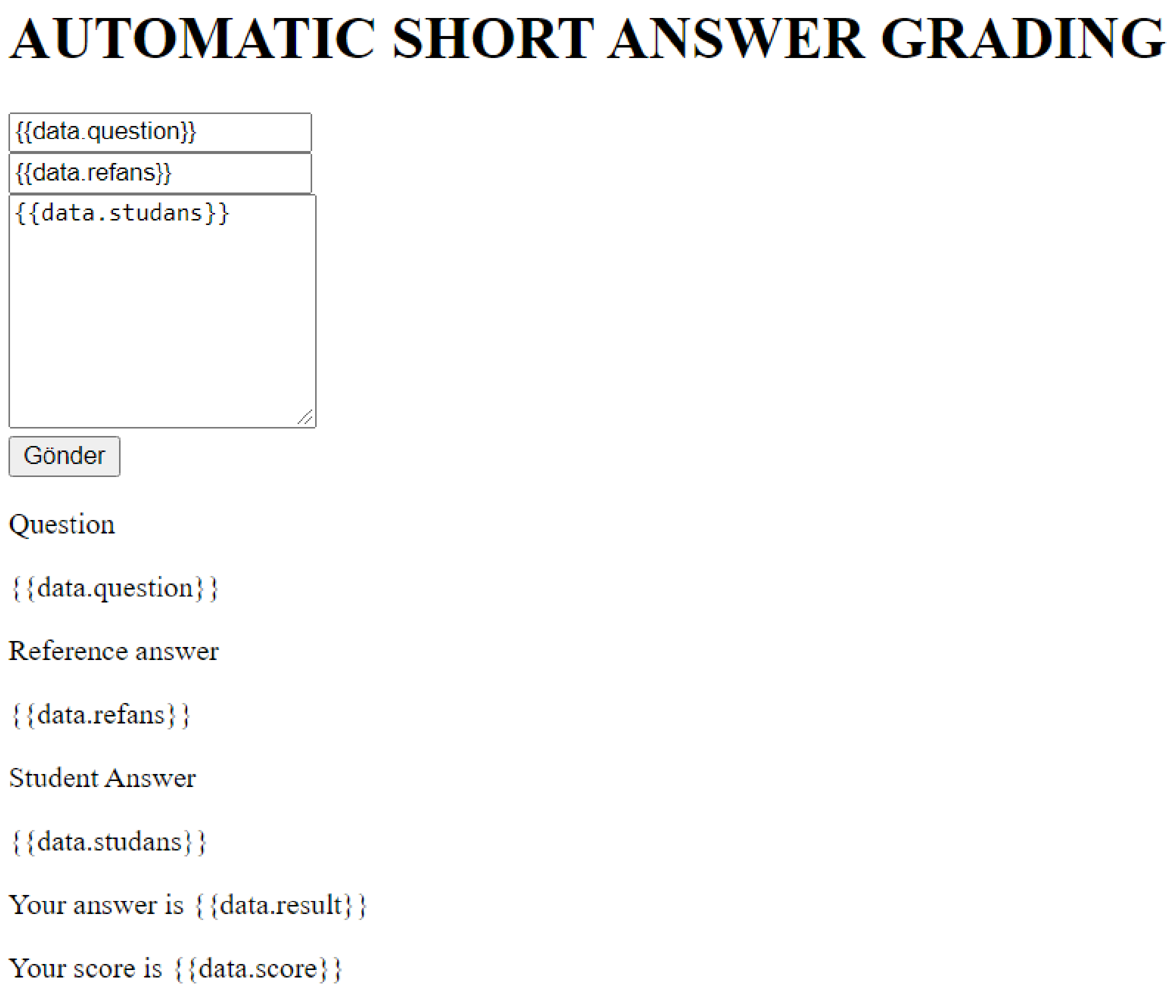
```
def get_prob(X):
    return clf.predict_proba(X)
```

Figure 28: Code for 'get_prob' function

This function uses ‘predict_proba’ function from Scikit-Learn library to evaluate the probability of the student answer being correct. The function uses the model built by ‘train_model’ function. It compares the similarity values of the student answers with the similarity values of example answers from the dataset. It attempts to predict if the student answer is correct by checking how many example answers with close similarity scores to the student answer are correct. If there are many example answer with similar similarity scores to the student answer that are marked as correct in the dataset, the model considers the student answer as likely to be correct and vice versa.

3. Application

3.1 Interface


AUTOMATIC SHORT ANSWER GRADING

{{data.question}}

{{data.refans}}

{{data.studans}}

Question

{{data.question}}

Reference answer

{{data.refans}}

Student Answer

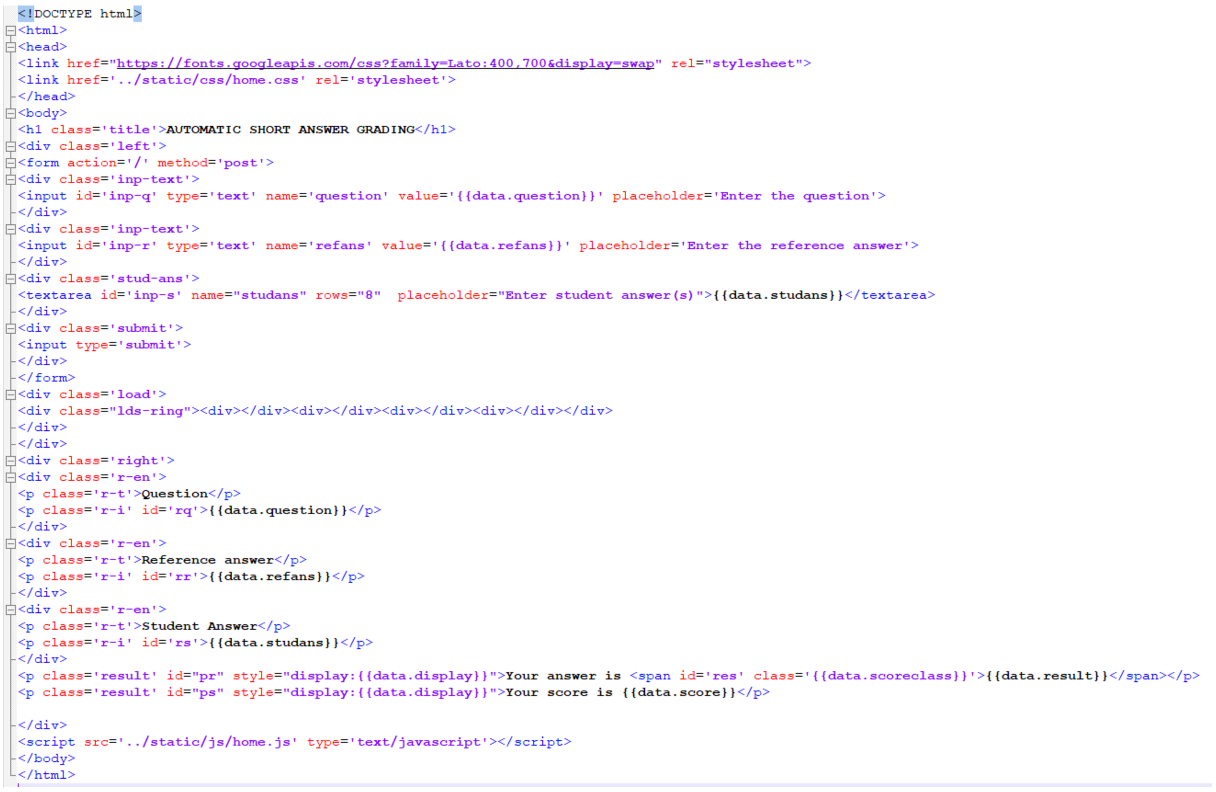
{{data.studans}}

Your answer is {{data.result}}

Your score is {{data.score}}

Figure 29: Blank form of the interface

This application is created to show how the program works and not meant for public use. The application has only one main interface. The interface contains 3 text boxes. From top to bottom, they receive the question, the reference answer and the student answer inputs from the user. After the inputs are received by clicking the 'Send' button at the bottom, each of the inputs are shown at the right side of the application. After the calculations are complete, a text message indicating whether the answer is correct or incorrect appear under the displayed inputs, with the score earned for the answer. The user can try different question and answer combinations n the same interface without needing re-run the application.

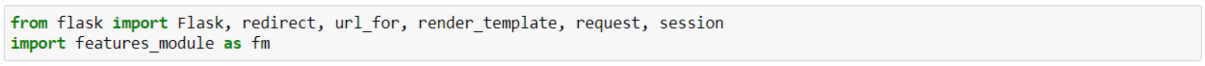


```html
<!DOCTYPE html>
<html>
<head>
<link href="https://fonts.googleapis.com/css?family=Lato:400,700&display=swap" rel="stylesheet">
<link href='../static/css/home.css' rel='stylesheet'>
</head>
<body>
<h1 class='title'>AUTOMATIC SHORT ANSWER GRADING</h1>
<div class='left'>
<form action='/' method='post'>
<div class='inp-text'>
<input id='inp-q' type='text' name='question' value='{{data.question}}' placeholder='Enter the question'>
</div>
<div class='inp-text'>
<input id='inp-r' type='text' name='refans' value='{{data.refans}}' placeholder='Enter the reference answer'>
</div>
<div class='stud-ans'>
<textarea id='inp-s' name="studans" rows="8"  placeholder="Enter student answer(s)">{{data.studans}}</textarea>
</div>
<div class='submit'>
<input type='submit'>
</div>
</form>
<div class='load'>
<div class="lds-ring"><div></div><div></div><div></div><div></div></div>
</div>
</div>
<div class='right'>
<div class='r-en'>
<p class='r-t'>Question</p>
<p class='r-i' id='rq'>{{data.question}}</p>
</div>
<div class='r-en'>
<p class='r-t'>Reference answer</p>
<p class='r-i' id='rr'>{{data.refans}}</p>
</div>
<div class='r-en'>
<p class='r-t'>Student Answer</p>
<p class='r-i' id='rs'>{{data.studans}}</p>
</div>
<p class='result' id="pr" style="display:{{data.display}}">Your answer is <span id='res' class='{{data.scoreclass}}'>{{data.result}}</span></p>
<p class='result' id="ps" style="display:{{data.display}}">Your score is {{data.score}}</p>

</div>
<script src='../static/js/home.js' type='text/javascript'></script>
</body>
</html>
```

Figure 30: HTML code used for the interface

3.2 Structure

3.2.1 Imports



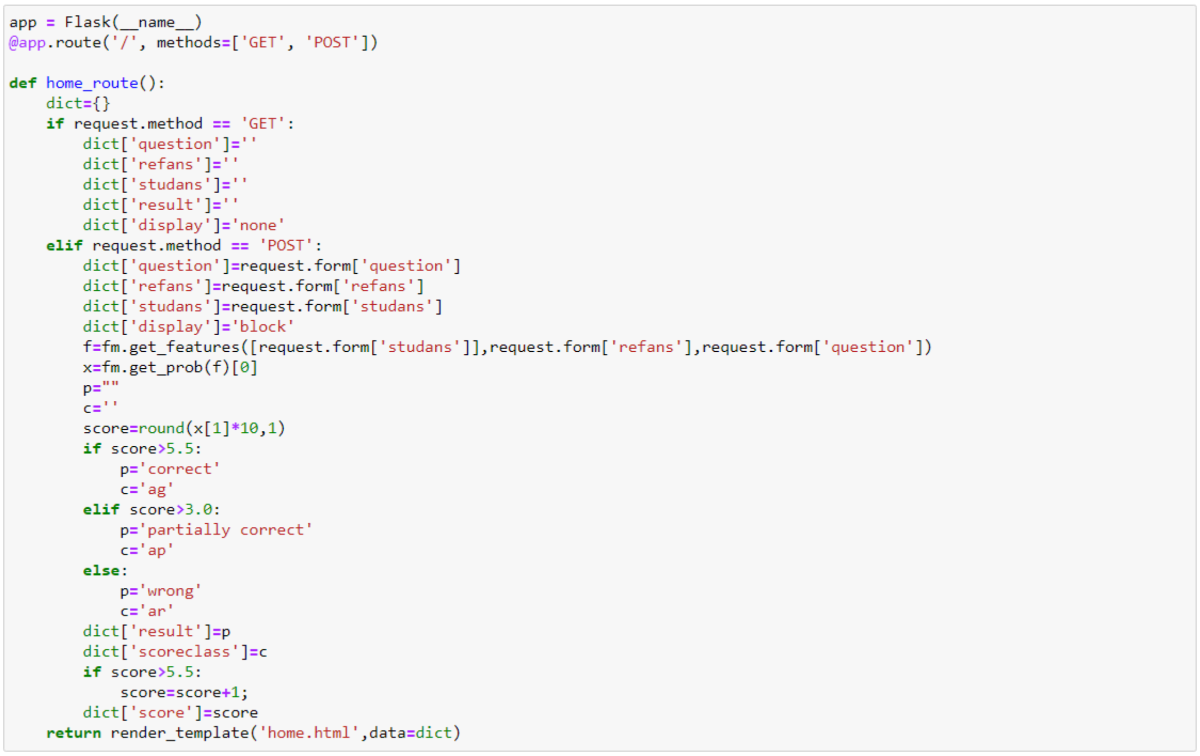
```python
from flask import Flask, redirect, url_for, render_template, request, session
import features_module as fm
```

Figure 31: Import of the application

Flask: The web framework used to create the web application.

features_module: The module that contains coded text processing, analyzing and the main functions of the program.

3.2.2 Home Function



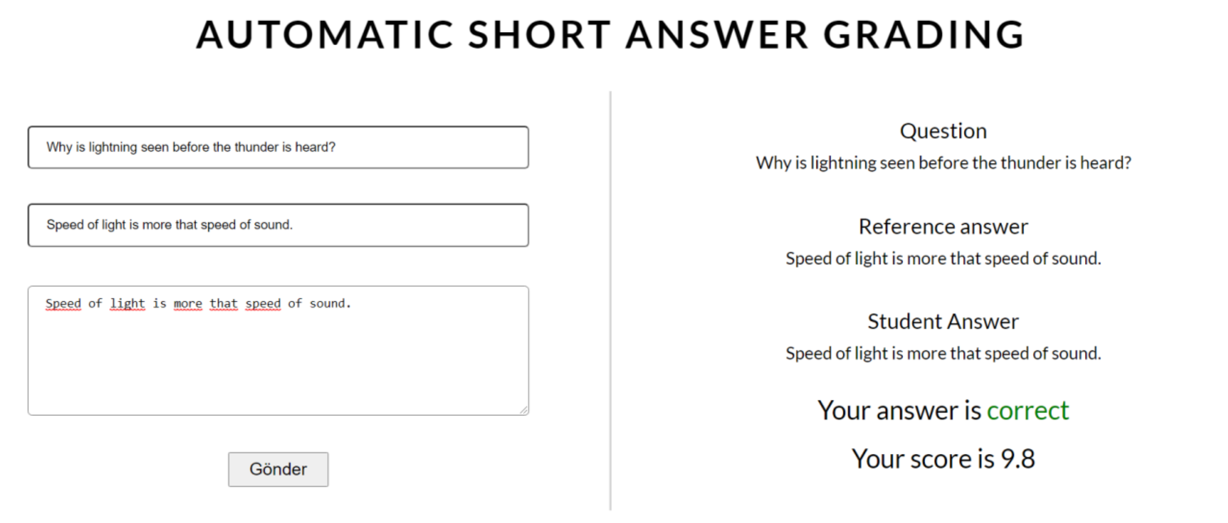
```
app = Flask(__name__)
@app.route('/', methods=['GET', 'POST'])

def home_route():
    dict={}
    if request.method == 'GET':
        dict['question']=''
        dict['refans']=''
        dict['studans']=''
        dict['result']=''
        dict['display']='none'
    elif request.method == 'POST':
        dict['question']=request.form['question']
        dict['refans']=request.form['refans']
        dict['studans']=request.form['studans']
        dict['display']='block'
        f=fm.get_features([request.form['studans']],request.form['refans'],request.form['question'])
        x=fm.get_prob(f)[0]
        p=""
        c=''
        score=round(x[1]*10,1)
        if score>5.5:
            p='correct'
            c='ag'
        elif score>3.0:
            p='partially correct'
            c='ap'
        else:
            p='wrong'
            c='ar'
        dict['result']=p
        dict['scoreclass']=c
        if score>5.5:
            score=score+1;
        dict['score']=score
    return render_template('home.html',data=dict)
```

Figure 32: Home function of the web application

There are 5 dictionaries defined in the home function.

- ‘question’: The question asked by the user or the instructor.
- ‘refrans’: The reference answer set by the user.
- ‘studan’: The answer given by the user or the student
- ‘result’: The score of the answer.
- ‘display’: The input to be displayed in the screen.

The function receives the question, the reference answer and the student answer inputs from the user and passes them to ‘get_feature’ function in ‘feature_module’ module. Meanwhile the inputs are stored in ‘display’ dictionary to be shown in screen. ‘get_feature’ function returns the similarity scores of the student answer. These values are sent to get_prob() function in the module to calculate the probability of the answer being correct. After the probability of being correct is received from the function, it’s rounded to a value between 1 to 10. If the final score is below 3.0, the answer is marked as incorrect. If the score is above 3.0 but below 5.5, it is marked as partially correct. If the score is above 5.5, then the answer is considered correct and increased by 1. The score is saved in ‘score’ dictionary and displayed in the screen.


AUTOMATIC SHORT ANSWER GRADING

Why is lightning seen before the thunder is heard?

Speed of light is more that speed of sound.

```
Speed of light is more that speed of sound.
```

Question

Why is lightning seen before the thunder is heard?

Reference answer

Speed of light is more that speed of sound.

Student Answer

Speed of light is more that speed of sound.

Your answer is correct

Your score is 9.8

Figure 33: Example usage of web application

4. Deficiencies

1. Due to large size of models and datasets used in text processing and analyzing part of the development, the application takes up too much space in temporary and permanent memory.
2. The processing speed is below the desired speed.
3. The scoring is inconsistent and same inputs can give slightly different results.

5. Results

The final program is capable of evaluating the correctness of the student answers with a successful rate. Although the scoring might change in a different run of the application, it achieves to be consistent in detecting correct and incorrect answers. The application does not require any instruction to be used correctly. The usage is very straightforward and the interface is easy to navigate. It can be used easefully for experimental purposes. Further development might be required for public use.

6. Sources

[1] Elmo Module https://tfhub.dev/google/elmo/2

[2] Universal Sentence Encoder https://tfhub.dev/google/universal-sentence-encoder/4

[3] Wikipedia2Vec Pre-trained Embedding
https://wikipedia2vec.github.io/wikipedia2vec/pretrained/

[4] Jupyter Notebook https://jupyter.org/