StackReQA: Stack Overflow Retrieval Question Answering

Alyssa Augsburger, John Lee, Sanjay Saravanan
UC Berkeley MIDS students
{alyssaaugsburger,johncslee,sanjayms}@berkeley.edu

Abstract

Retrieval Question-Answering (ReQA) tasks identify the most relevant answer to a question. This paper presents StackReQA, which retrieves the most relevant answer to a given Stack Overflow question. We provide a solution that utilizes both the text and code of an input question in order to predict the most useful and appropriate answer. We present several approaches using different embeddings, ordering techniques, and architectures. Specifically, we embed the data using the Universal Sentence Encoder (USE), Bidirectional Encoder Representations from Transformers (BERT), Word2Vec and various combinations of these embeddings. We experiment with the ordering of questions and answers and text and code. Additionally, we compare models utilizing a dual encoder architecture and neural networks with early versus late fusion. We find that our models utilizing USE embeddings perform best. Specifically, our best-performing model embeds the data as is with USE and uses a late fusion neural network architecture.

1 Introduction

Retrieval Question-Answering (ReQA) tasks provide a path for systems to retrieve relevant answer(s) to a question. This is an end-to-end solution to the traditional Question Answering (QA) tasks in which systems require an intermediate contextual document in order to retrieve an answer within a specific span of text. This is important because it is uncommon that a person would provide both a question and a document in order to retrieve an answer.

We are motivated by helpful solutions for Python programming questions from Stack Overflow. Our goal is to identify the best Stack Overflow answer to a proposed question. Additionally, we wish to measure the effectiveness of utilizing different natural language processing (NLP) methods in order to retrieve answers to programming questions. Therefore, we apply data preprocessing techniques in conjunction with a variety of architectures to learn which combinations produce the best results. Specifically, we focus on the following:

- Representation of code snippets and natural language text
- Ordering techniques
- Combinations of embeddings in varied model architectures

The challenge in this task is handling both natural language text and Python code snippets. This data is embedded separately with the Universal Sentence Encoder (USE) (Yang et al., 2019), Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2018), and Word2Vec (Mikolov et al., 2013). We then evaluate on our test data based on similarities between questions and answers. The performance of each strategy is measured by the mean reciprocal rank (MRR) (Ahmad et al., 2019) and precision.

2 Background

Question answering is important as it allows users to interact with systems and retrieve answers to questions asked in natural language rather than asked in machine specific code. We are motivated by this task as we want the initial predicted response from our models to return applicable solution(s) so time is not wasted on irrelevant suggestions.

Question answering is a difficult problem and becomes more challenging when there is a large corpus of documents to retrieve relevant answers from. We are also motivated by the characteristics of a good answer retrieval model as proposed by

(Ahmad et al., 2019), which include the necessity of an end-to-end solution and that the model be contextually-aware. To limit the scope of our work, we strive to meet these characteristics without the use of a large corpus of documents for retrieval. We also use the same evaluation metric, MRR.

Furthermore, we are motivated by the work of (Kulkarni and Rosich, 2020) that predicts the best response to questions from Stack Overflow textual data. They do not include code snippets in order to reduce record size for their models. We solve a similar problem with similar architectures and dataset. We strive to improve on their findings by including code fragments in our models, as we believe the code fragments can provide additional context.

Similarly, (Guo et al., 2020) evaluate eight different retrieval tasks using two supervised neural models: one based on fine-tuning BERT and another based on USE-QA. They describe that both models perform well and in their experiments, USE-QA is used for the purposes of retrieving a sentence answer from a paragraph. However, the answer texts in our dataset are paragraphs instead of single sentences, so we therefore choose USE instead to produce embeddings of our data given the desirable performance of USE-QA.

3 Methods

3.1 Task

The overarching task of our work involves the development of question-answering model architectures to perform best response prediction given input questions. For our experiments, we associate a set of ten answers to each question, consisting of the actual answers for the question and randomly selected answers from our data.[1] This best response prediction task involves creating a set of rankings for each set of ten question-answer pairs, from which the top-ranking answer is predicted as the best answer for each input question. In subsequent sections, we discuss our experimentation with strategies for representation, ordering, and architectures.

3.2 Data

The supervised dataset employed in our models is a filtered subset of the 200+GB corpus of (Stack Overflow, 2019), accessible through Google BigQuery. The following criteria are applied to create the subset of data where:

- Questions contain the $Python$ tag: We are interested in Python language code questions which allows us to utilize our domain knowledge.
- Questions contain an $accepted\ answer$: The $accepted\ answer$ indicator (yes or no) provides labeled data to fine-tune supervised NLP models.
- Number of answers per question is greater than 0 and less than 11: Questions with 0 answers do not provide labeled data for supervised training. Limiting answers to less than 11 (10 answers) already accounts for 99.6% of the $Python$ tag dataset and meets the criteria established in our task.
- Question creation date is between 2016 and 2020: We aim for a dataset of greater than 10,000 (for training purposes) and less than 100,000 (to reduce high computational cost) which is satisfied within this range of data.

Our filtered data contains 30,623 unique questions and 60,336 answers, which averages 1.97 answers per question. Additional exploratory data analysis (EDA) shows that 75% of the questions have less than two answers which means that random answers will appear frequently in each set of ten answers per question.

Further EDA indicates 11,504 answers that contain outlinks, which is a concern as discussed in (Kulkarni and Rosich, 2020). We do not believe these answers are a concern as the answers also include text that can provide sufficient information for the NLP models to use to train or infer on.[2] Additionally, there is a set of two duplicated answers associated with two different but similar questions. These duplicated answers are not removed.

Minimum, maximum, and average lengths (number of tokens) of the input data are shown in Table 1. Long lengths can lead to poor performance due to a model's limited ability to capture long-term context, which we consider when fine-tuning. The average lengths help guide our decisions when selecting maximum sequence lengths on BERT models.

[1]See Appendix B Figure 1

[2]See Appendix B Figure 5

	Lengths		
Data	Min.	Max.	Avg.
Question Text+Code	6	3064	135.7
Answer Text+Code	2	3509	86.1

Table 1: Minimum, maximum, and average lengths (number of tokens).

No Code vs. No Text	Percentage
Questions without code	4.27%
Questions without text	0.19%
Answers without code	4.61%
Answers without text	2.48%

Table 2: Percentage of questions and answers that include or do not include code in the dataset

The percentage of data without code or text is shown in Table 2. These low percentages indicate that most questions and answers contain both text and code. Therefore, we decide to keep both text and code in our models to capture the full context of our data.

The train/development/test split are 80/10/10 of the unique number of questions which result in 24,498/3,062/3,063 unique questions and approximately twice as many in number of answers.

3.3 Evaluation Metric and Baseline

For evaluation, we compute the MRR (Ahmad et al., 2019) and precision metrics to compare each of our models. The final models are compared against a BM-25 (Seitz, 2020) model.

MRR We use MRR as the primary metric because it evaluates a ranked list of answers. We measure the rank that the model assigns to the correct answer for each question then use Eq (1) to calculate MRR. A MRR score of 1.0 indicates that the model predicts the correct answer as rank 1 every time. In Eq (1), Q is the set of all questions and $rank_i$ is the rank of the correct answer for the ith question.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (1)$$

Additionally, we use precision as a secondary metric to describe how many of the positive predictions were actually correct.

BM-25 Following suit with (Ahmad et al., 2019) and considering the low computational cost to run the algorithm, we select BM-25 as a baseline.[3] It is interesting to note that using BM-25, with text only (0.4141 MRR), performed worse than using both text and code (0.6392 MRR).[4] Moving forward, all of our models will include both text and code applied to different strategies discussed below.

3.4 Strategies

3.4.1 Representations

We embed the questions and answers using USE, BERT, and Word2Vec and experiment with various combinations. Additionally, for each type of embedding we apply different methods of embedding the data, such as averaging sentences and varying the sequence selections and maximum sequence lengths. For each embedding variation, we run a dual encoder model to evaluate the performance of the embedding technique and the results are shown in Table 3. For each type of embedding, we then use the best performing technique in our subsequent neural models.

USE We employ USE to embed the data because it encodes entire sentences and does not limit the length of the input. This fits our data well as most questions and answers consist of at least one or more sentences. Additionally, USE embeddings can be used to compute sentence-level semantic similarity scores. We use the universal-sentence-encoder (v4) from TensorFlow Hub. This is the Deep Averaging Network (DAN) version of the model which allows for efficient inference, but results in reduced accuracy. We try two different methods of embedding the data with USE, each creating 512-dimensional feature vectors. With method 1, we embed each question and answer as is, regardless of the number of sentences they consist of. With method 2, we tokenize each question and answer into sentences. After, we embed each sentence individually and then average the sentence embeddings together to produce a single embedding for each question and answer. We find that the method of embedding the data as is performs better than the method of averaging the sentence embeddings, as shown in Table 3.

BERT We use the BERT architecture to develop contextually-aware embeddings for our question and answer texts. In order to successfully use BERT, we build a custom tokenizer utilizing out-

[3]Refer to Appendix A for more detail
[4]See Appendix B Table A1

Embeddings Experiment [1]	**MRR**	**Precision**
USE - Data As Is	**0.7567**	**0.6108**
USE - Averaged Sentences	0.6535	0.4900
USE - Text Then Code	0.7374	0.5856
USE - Code Then Text	0.7361	0.5814
BERT - Last 512 Tokens	0.5656	0.3939
BERT - Last 136/87 Tokens	0.5633	0.3963
BERT - First And Last 136/87 Tokens	0.5862	0.4140
BERT - First And Last 136/87 Tokens - Text Then Code	0.6165	0.4508
BERT - First And Last 136/87 Tokens - Code Then Text	0.6147	0.4461
BERT - First And Last 136/87 Tokens - Text Then Code - Hidden States	0.5975	0.4369
Word2Vec - Code Reserved Words	0.2317	0.0200
Word2Vec - Top 50 Code	0.3214	0.0797
Word2Vec - Text And Code	0.5170	0.3475

[1]Dot product is applied on USE embeddings. Cosine Similarity is applied on BERT and Word2Vec embeddings.

Table 3: Performance of different embedding techniques on a dual encoder architecture.

puts from the pre-trained bert-base-cased BertTokenizer to retrieve n-dimensional vectors of token IDs, masks, and segment IDs. We experiment with varying sequence lengths as a parameter to our custom tokenizer. As discussed in (Sun et al., 2019), we also only parse x beginning and y end tokens from texts, leaving out the middle when constructing input vectors for the BERT model. To construct our embeddings, we decide to use the pre-trained bert-base-cased TFBertModel because it is specialized for tasks that use entire sentences. We pass each of our input vectors into this model, retrieve the last hidden states for each token in the resulting output, and then average the last hidden states to create 768-dimensional feature vectors. As depicted in Table 3, we run several experiments for different variations of embeddings. Our findings show that using the first and last 136/87 tokens for questions and answers respectively, which are derived from the average token lengths from Table 1, provide better results than simply using the last 512 or 136/87 tokens for texts.

Word2Vec Since USE and BERT are not explicitly trained on Python code data, a Word2Vec model from gensim's model package is used to represent code. This decision is based on the simplicity of utilizing a continuous-bag-of-words (CBOW) unsupervised training model. A window size of 5 is used to train a subset of code data over 10 epochs. Since code typically contains many interchangeable characters, such as variable name x in $x = 5$, we filter out the noisy training code data in two different ways. In Model 1, we keep only the 35 Python reserved words and create 100-dimensional feature vectors. In Model 2, we keep the top 50 most frequent tokens with a single semantic meaning, such as the addition indicator operator += or the *print* function. These are not part of the 35 Python reserved words but are still popular tokens in the data and can provide a canonical way of representing the code. We create 13-dimensional feature vectors, selected based on the the average number of tokens per sentence after filtering. In model 3, we train on all text and code data for the purposes of running a final combined embedding model. Similarly, we create 135-dimensional feature vectors based on the average number of tokens per question. Our findings indicate that all models utilizing Word2Vec code representation performed worse than using other code representations.[5]

3.4.2 Ordering

Ordering is the second strategy with which we decide to experiment with in regards to the construction of embeddings and our models. The objective here is to identify if ordering has any effect in our models' performance. Table 3 shows the following variations in ordering for embeddings: *Text Then Code* and *Code Then Text*. In the former, we ensure all text in questions and answers appear before the code, whereas reversed in the latter. For the USE embeddings, our findings show that neither method of ordering performs better than using the data as is. For BERT embeddings, we note that the *Code Then Text* ordering method applied on top of the first and

[5]See Appendix B Table A1

last 136/87 tokens performs the best. Another approach that we apply with BERT is concatenating the bottom layer hidden states with the top layer hidden states to form 1536-dimensional vectors. The sheer contrast between the learned weights of the bottom and top layer hidden states are what we try to experiment with to understand if it captures the differentiation of embeddings from early and late states to reduce confusion. We note, however, that using this approach on top of the *Text Then Code* ordering strategy and first and last 136/87 tokens still underperforms. Unlike with the USE DAN model, a possible reason for why ordering seems to positively affect BERT could be that it takes advantage of the self-attention layers within its architecture.

In regards to modeling, we present the following variations in ordering of embedding inputs: *Question + Answer* and *Answer + Question*. As depicted in Table 4, the USE model with the *Question + Answer* approach and the BERT model with the *Answer + Question* approach perform the best. Although ordering of the input embeddings seems to have helped the BERT model, it is difficult to state if there is any other reason for this other than luck with regards to the randomized model weights initialization.

3.4.3 Architectures

Dual Encoder We apply the dot product for USE or cosine similarity for BERT to embedded test data in order to predict answers that are most similar to each question.[6] USE is trained to return high dot products on similar input data (Yang et al., 2019), while cosine similarity is a distance metric which can be used to measure similarity between two embeddings. This dual encoder model allows for comparison of different embedding strategies, as previously described, prior to more extensive fine-tuning of subsequent models.

Concatenated Neural Network We build a neural network layer on top of the embedded data to fine-tune our models.[7] Variations at each level of the pipeline are conducted to determine if there is an ideal strategy. The question and answer text plus code are all embedded with USE, BERT, and Word2Vec. The input to the different neural network models includes various combinations as discussed in section 3.4.1. The ordering of text and code in the questions and answers is also varied as discussed in section 3.4.2. Finally, we fine-tune the neural network by varying hyperparameters and number of dense layers. The softmax output layer produces a probability value, where the predicted answer for each question is determined based on the highest probability score.

Two Towers Neural Network Similar to the concatenated neural network, the two towers architecture includes a neural network layer on top of the embedded data.[8] The only difference is that the question and answer embeddings are concatenated prior to the neural network (early fusion) in the previous model architecture whereas the two towers model computes the dot products of the question and answer embeddings after the neural network (late fusion). Again, we vary the combinations of embeddings and fine-tune the neural network to find the optimal strategy.

4 Results and Discussion

USE embeddings perform best across all three architectures. Specifically, our best model is the two towers model with USE embeddings, which produces the best MRR of 0.7817, approximately a 22% improvement over our BM-25 baseline. Our best performing two towers model uses a single dense layer on top of the answer embedding with 512 nodes, the Adam optimizer with a learning rate of 0.001, sparse categorical cross-entropy loss, a batch size of 64, and 2 epochs. We find that not only does including the code improve our model's performance, but also that USE does a better job of capturing the semantic context of code than Word2Vec and BERT. In alignment with (Ahmad et al., 2019), not much fine-tuning is required for USE to produce accurate predictions regarding similarity between questions and answers, as the USE two towers architecture performed only 3% better than the dual encoder architecture. Additional experimentation can be found in Table A1 of Appendix B.

In regards to error analysis, we note that out of all of the incorrect predictions, the accepted answer to questions was most frequently predicted as the second-ranking answer, as shown in Figure 1. It is interesting to note that the majority of the predicted answers that were ranked first, with the accepted answers being ranked second, were actual answers to

[6]See Appendix B Figure 2

[7]See Appendix B Figure 3

[8]See Appendix B Figure 4

Ordering and Architecture Experiment	**MRR**	**Precision**
Concatenated Neural Network with Ordering Variations		
USE - Question + Answer	0.7794	0.6252
USE - Answer + Question	0.7323	0.5657
BERT - Question + Answer	0.6741	0.4904
BERT - Answer + Question	0.6814	0.5011
Two Towers		
USE	**0.7817**	**0.6290**
BERT	0.6960	0.5185

Table 4: Performance of different ordering and architecture experiments.

Architecture	**Best Models**	**MRR**	**Precision**
Baseline	BM-25 - Text and Code	0.6392	0.4757
Dual Encoder	USE - Data As Is	0.7567	0.6108
Concatenated Neural Network	USE - Question Then Answer	0.7794	0.6252
Two Towers	USE	**0.7817**	**0.6290**

Table 5: Best performing models of each architecture.

the question rather than randomly selected answers. Upon further inspection, the errors in our model can be attributed to the following challenges:

- In the cases where the accepted answer was ranked second, the predicted answer tends to have more matching tokens to the question than the actual accepted answer, causing greater confusion for the model to learn and infer.[9]

- Since it is up to human interpretation on what is considered an *accepted answer*, it is challenging for our model to distinguish between two or more similar and helpful answers.

Another finding we note is that on the average, sequence lengths (number of tokens) of the incorrectly predicted answers were similar to what was previously described in our EDA of the full dataset. In this case, longer lengths are not necessarily a cause of error for our models. We also realize that we do not have a sufficient amount of data for training our concatenated model architectures since they require more parameters. Therefore, we hypothesize that this contributes to why our two towers models outperforms the concatenated models.

5 Conclusion

In this paper, we apply different embedding strategies, ordering techniques, and architectures to retrieve the most relevant answer to a given Stack Overflow question. We specifically do not exclude the code in the data. We observe that embedding the data with USE outperforms models with embeddings using BERT, Word2Vec, or varied combinations. Additionally, we observe that our two towers model with late fusion performs best, yet it is only a slight improvement over the less computationally-intensive dual encoder model. Future work to improve upon our models and their performance includes: better representation of code data, more fine-tuning, an expanded dataset, and an ablation study to better understand which components of the two towers architecture contribute to the model's performance.

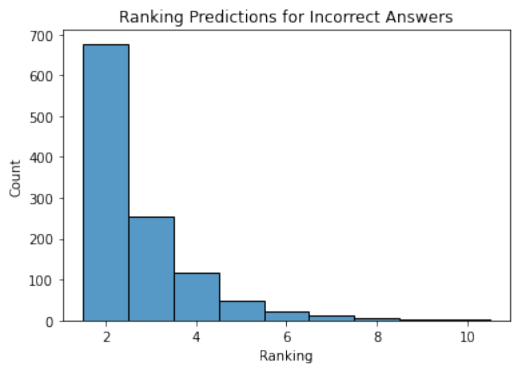


Figure 1: Histogram of Incorrect Answer Ranking Predictions

[9]See Appendix B Figures 6 and 7

References

Amin Ahmad, Noah Constant, Yinfei Yang, and Daniel Cer. 2019. Reqa: An evaluation for end-to-end answer retrieval models. *Proceedings of the 2nd Workshop on Machine Reading for Question Answering*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Mandy Guo, Yinfei Yang, Daniel Cer, Qinlan Shen, and Noah Constant. 2020. Multireqa: A cross-domain evaluation for retrieval question answering models.

Rahul Kulkarni and Ryan Rosich. 2020. Semantic similarity and response prediction for programming qa.

Tomas Mikolov, Greg Corrado, Kai Chen, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space.

Rudi Seitz. 2020. Understanding tf-idf and bm25.

Kaggle Stack Overflow. 2019. Stack overflow data.

Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune BERT for text classification? *CoRR*, abs/1905.05583.

Yinfei Yang, Daniel Cer, Amin Ahmad, Mandy Guo, Jax Law, Noah Constant, Gustavo Hernandez Abrego, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2019. Multilingual universal sentence encoder for semantic retrieval.

Appendices

Appendix A Additional Information

BM-25 Baseline This model is a TF-IDF based method which considers term saturation and document lengths in addition to the term and document frequencies. Without having to train a neural network, running this algorithm takes a few seconds on our test dataset. Additionally, this algorithm did not have an issue on paragraph-level data when comparing the long lengths of the questions and answers, so it fits well with this ReQA task. To use this algorithm, we tokenize the input data to create a document corpus from the answers (10 per question) and a query corpus from the question. Using the gensim summarization.bm25 module, we calculate BM-25 scores for each answer which are used to rank the answers per question. The rank of the actual accepted answer is used to calculate the MRR score.

Appendix B Additional Figures and Table

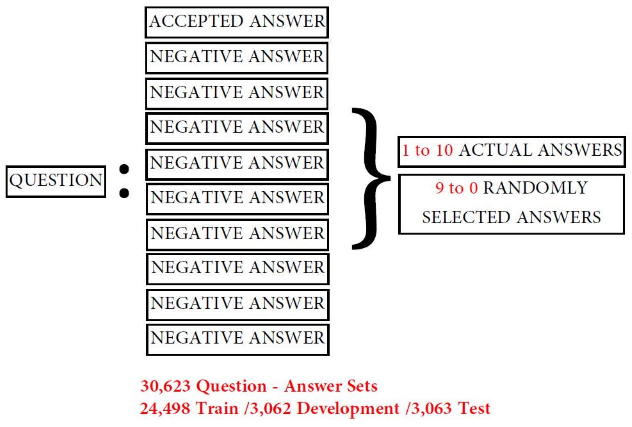


Figure 1: Question-Answer Set

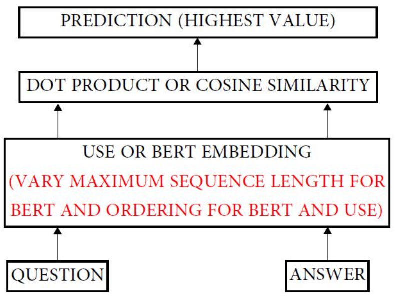

PREDICTION (HIGHEST VALUE)
DOT PRODUCT OR COSINE SIMILARITY
USE OR BERT EMBEDDING
(VARY MAXIMUM SEQUENCE LENGTH FOR BERT AND ORDERING FOR BERT AND USE)
QUESTION
ANSWER

Figure 2: Dual Encoder Architecture

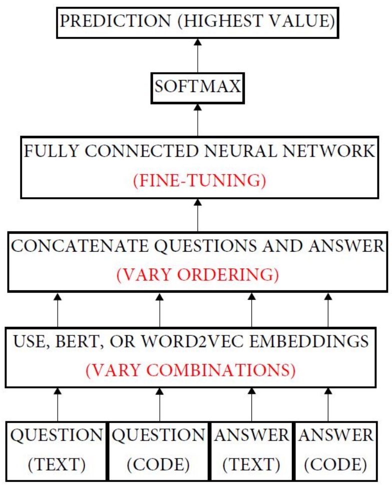

PREDICTION (HIGHEST VALUE)
SOFTMAX
FULLY CONNECTED NEURAL NETWORK
(FINE-TUNING)
CONCATENATE QUESTIONS AND ANSWER
(VARY ORDERING)
USE, BERT, OR WORD2VEC EMBEDDINGS
(VARY COMBINATIONS)
QUESTION (TEXT)
QUESTION (CODE)
ANSWER (TEXT)
ANSWER (CODE)

Figure 3: Concatenated Neural Network Architecture

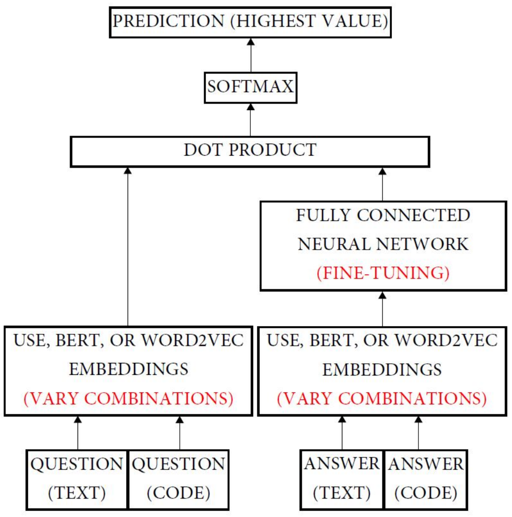
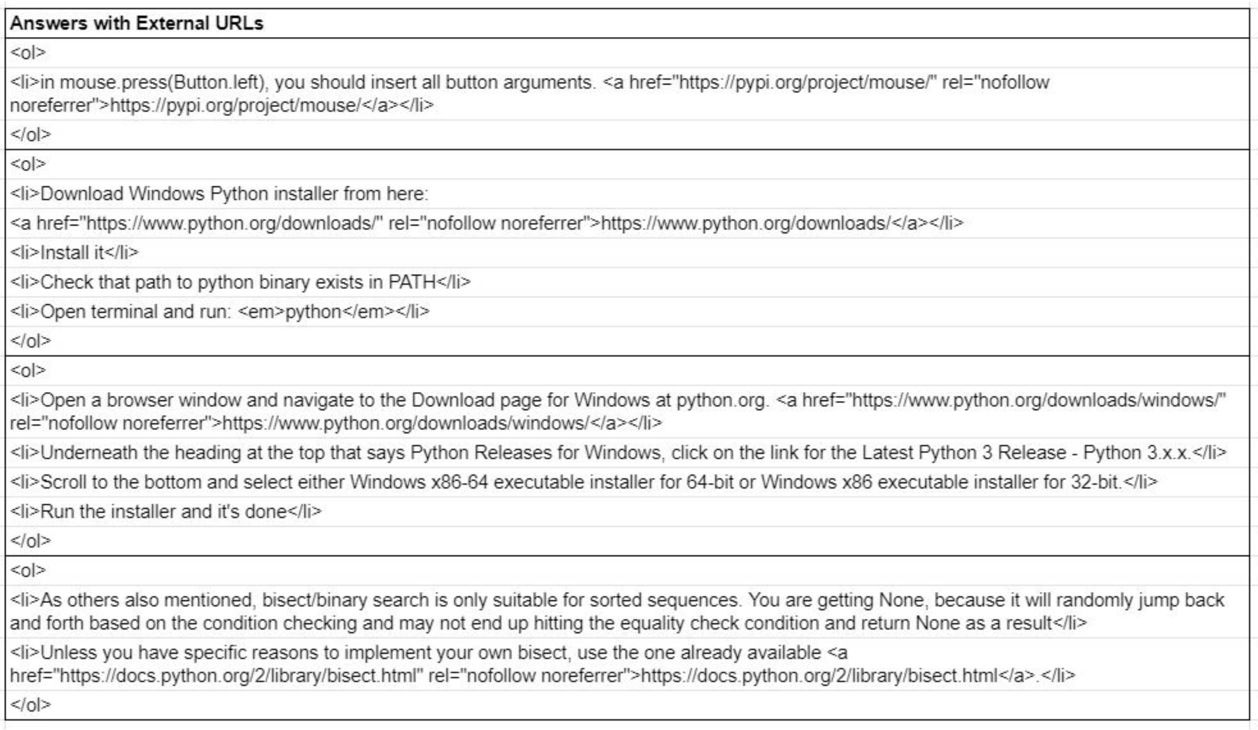
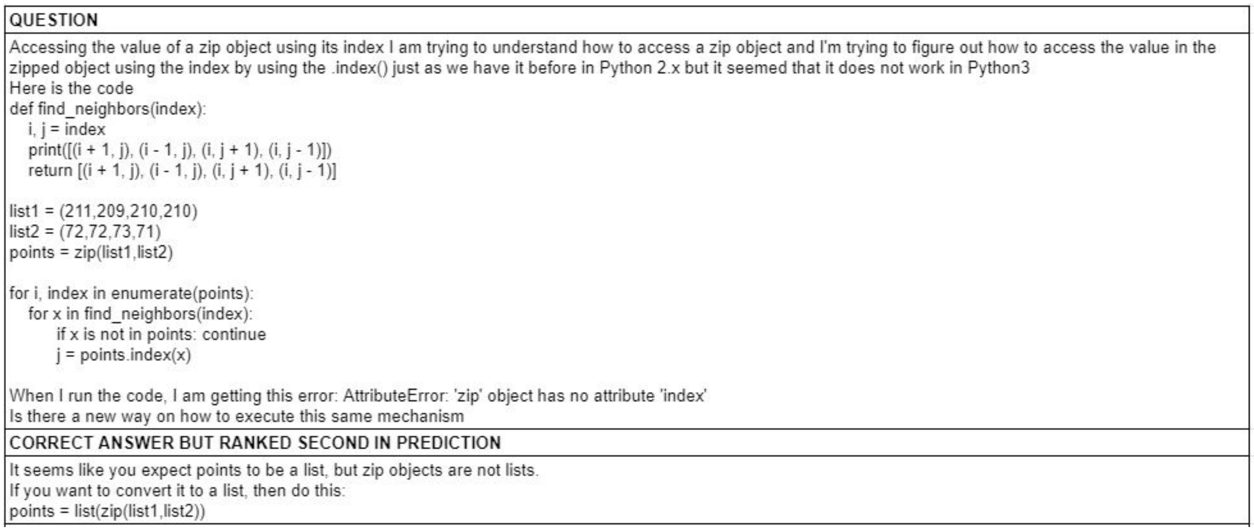


Figure 4: Two Towers Architecture



Answers with External URLs
<ol>
<li>in mouse.press(Button.left), you should insert all button arguments. <a href="https://pypi.org/project/mouse/" rel="nofollow noreferrer">https://pypi.org/project/mouse/</a></li>
</ol>
<ol>
<li>Download Windows Python installer from here:
<a href="https://www.python.org/downloads/" rel="nofollow noreferrer">https://www.python.org/downloads/</a></li>
<li>Install it</li>
<li>Check that path to python binary exists in PATH</li>
<li>Open terminal and run: <em>python</em></li>
</ol>
<ol>
<li>Open a browser window and navigate to the Download page for Windows at python.org. <a href="https://www.python.org/downloads/windows/" rel="nofollow noreferrer">https://www.python.org/downloads/windows/</a></li>
<li>Underneath the heading at the top that says Python Releases for Windows, click on the link for the Latest Python 3 Release - Python 3.x.x.</li>
<li>Scroll to the bottom and select either Windows x86-64 executable installer for 64-bit or Windows x86 executable installer for 32-bit.</li>
<li>Run the installer and it's done</li>
</ol>
<ol>
<li>As others also mentioned, bisect/binary search is only suitable for sorted sequences. You are getting None, because it will randomly jump back and forth based on the condition checking and may not end up hitting the equality check condition and return None as a result</li>
<li>Unless you have specific reasons to implement your own bisect, use the one already available <a href="https://docs.python.org/2/library/bisect.html" rel="nofollow noreferrer">https://docs.python.org/2/library/bisect.html</a>.</li>
</ol>

Figure 5: Example answers with external URLs that still provide context



QUESTION

Accessing the value of a zip object using its index I am trying to understand how to access a zip object and I'm trying to figure out how to access the value in the zipped object using the index by using the .index() just as we have it before in Python 2.x but it seemed that it does not work in Python3
Here is the code

```
def find_neighbors(index):
   i, j = index
   print([(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)])
   return [(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)]

list1 = (211,209,210,210)
list2 = (72,72,73,71)
points = zip(list1,list2)

for i, index in enumerate(points):
   for x in find_neighbors(index):
        if x is not in points: continue
        j = points.index(x)
```

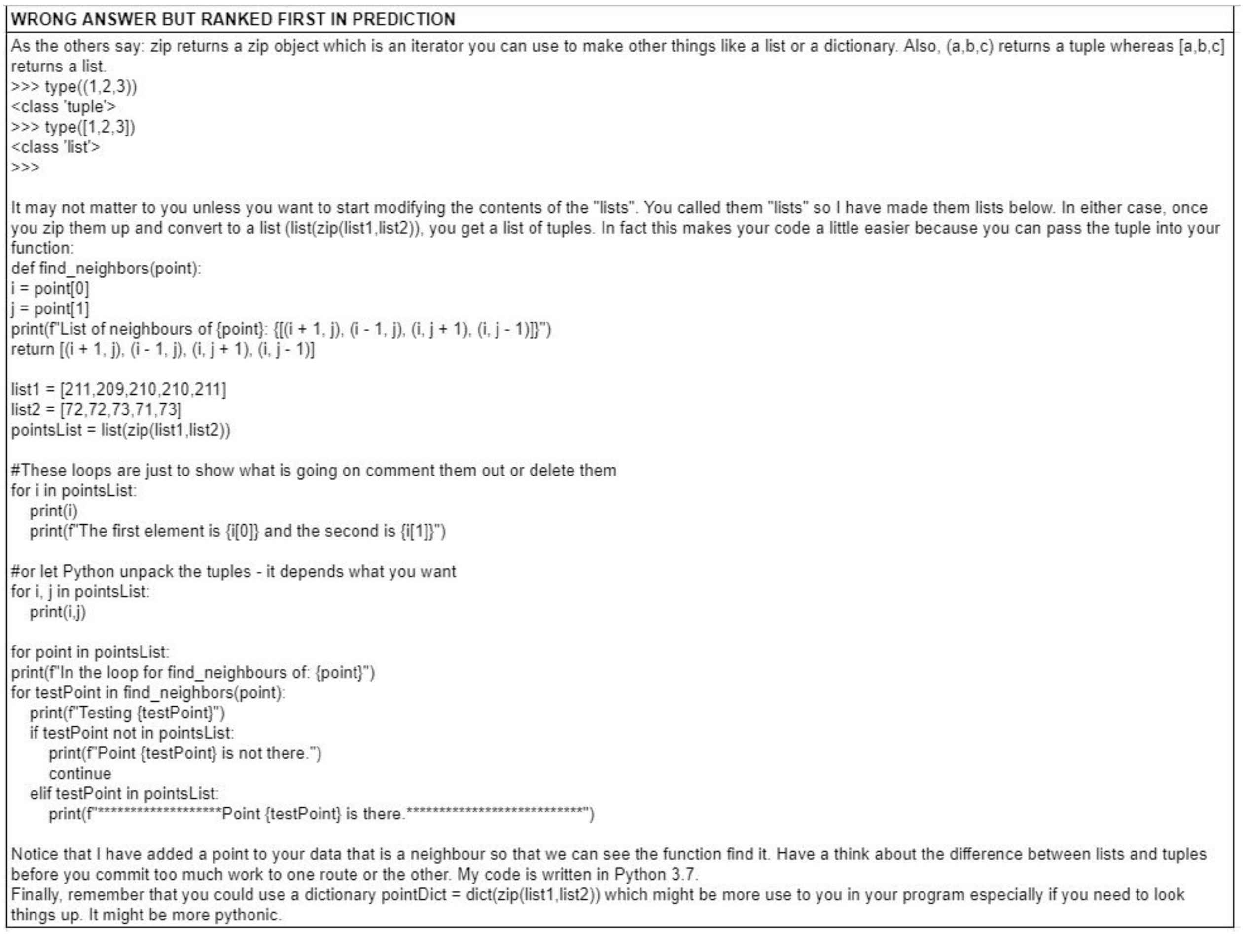
When I run the code, I am getting this error: AttributeError: 'zip' object has no attribute 'index'
Is there a new way on how to execute this same mechanism

CORRECT ANSWER BUT RANKED SECOND IN PREDICTION

It seems like you expect points to be a list, but zip objects are not lists.
If you want to convert it to a list, then do this:

```
points = list(zip(list1,list2))
```

Figure 6: Example incorrect answer prediction Part 1



WRONG ANSWER BUT RANKED FIRST IN PREDICTION

As the others say: zip returns a zip object which is an iterator you can use to make other things like a list or a dictionary. Also, (a,b,c) returns a tuple whereas [a,b,c] returns a list.

```
>>> type((1,2,3))
<class 'tuple'>
>>> type([1,2,3])
<class 'list'>
>>>
```

It may not matter to you unless you want to start modifying the contents of the "lists". You called them "lists" so I have made them lists below. In either case, once you zip them up and convert to a list (list(zip(list1,list2)), you get a list of tuples. In fact this makes your code a little easier because you can pass the tuple into your function:

```
def find_neighbors(point):
i = point[0]
j = point[1]
print(f"List of neighbours of {point}: {[(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)]}")
return [(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)]

list1 = [211,209,210,210,211]
list2 = [72,72,73,71,73]
pointsList = list(zip(list1,list2))

#These loops are just to show what is going on comment them out or delete them
for i in pointsList:
   print(i)
   print(f"The first element is {i[0]} and the second is {i[1]}")

#or let Python unpack the tuples - it depends what you want
for i, j in pointsList:
   print(i,j)

for point in pointsList:
print(f"In the loop for find_neighbours of: {point}")
for testPoint in find_neighbors(point):
   print(f"Testing {testPoint}")
   if testPoint not in pointsList:
      print(f"Point {testPoint} is not there.")
      continue
   elif testPoint in pointsList:
      print(f"*******************Point {testPoint} is there.***************************")
```

Notice that I have added a point to your data that is a neighbour so that we can see the function find it. Have a think about the difference between lists and tuples before you commit too much work to one route or the other. My code is written in Python 3.7.
Finally, remember that you could use a dictionary pointDict = dict(zip(list1,list2)) which might be more use to you in your program especially if you need to look things up. It might be more pythonic.

Figure 7: Example incorrect answer prediction Part 2

Miscellaneous Experiments [1]	**MRR**	**Precision**
BM-25 Baseline		
BM-25 With Text Only	0.4141	0.2151
BM-25 With Text And Code	**0.6392**	**0.4757**
Word2Vec Code Representation [2]		
Concatenated Neural Network		
USE_{QT} + $W2V1_{QC}$ + USE_{AT} + $W2V1_{AC}$	0.6568	0.4659
USE_{QT} + $W2V1_{QC}$ + USE_{AT} + $W2V1_{AC}$	0.6568	0.4659
USE_{QT} + $W2V2_{QC}$ + USE_{AT} + $W2V2_{AC}$	0.6847	0.5049
$BERT_{QT}$ + $W2V1_{QC}$ + $BERT_{AT}$ + $W2V1_{AC}$	0.5372	0.3377
$BERT_{QT}$ + $W2V2_{QC}$ + $BERT_{AT}$ + $W2V2_{AC}$	0.5448	0.3388
Two Towers		
USE_{QAT} + $W2V1_{QAC}$	0.6844	0.5072
USE_{QAT} + $W2V2_{QAC}$	**0.6910**	**0.5154**
$BERT_{QAT}$ + $W2V1_{QAC}$	0.5801	0.3782
$BERT_{QAT}$ + $W2V2_{QAC}$	0.5765	0.3796
BERT Hidden States		
Top And Bottom Layers - Concatenated	0.6795	0.5002
Top And Bottom Layers - Two Towers	**0.7047**	**0.5322**
USE Averaging Sentences		
USE - Concatenated	0.7412	0.5806
USE - Two Towers	**0.7552**	**0.5954**
Combined Embeddings Model		
Concatenated Neural Network		
USE_{QTC} + $W2V3_{QTC}$ + USE_{ATC} + $W2V3_{ATC}$	0.7596	0.5979
USE_{QATC} + $BERT_{QATC}$ + $W2V3_{QATC}$	0.7382	0.5729
Two Towers		
USE_{QTC} + $W2V3_{QTC}$ + USE_{ATC} + $W2V3_{ATC}$	**0.7749**	**0.6206**
USE_{QATC} + $BERT_{QATC}$ + $W2V3_{QATC}$	0.7625	0.5997

[1]Abbreviations in table are formatted as Embedding with subscripts Question Or Answer and Text Or Code. For example, USE_{QAT} represents USE embeddings on Question and Answer Text while $W2V1_{ATC}$ represents Word2Vec embeddings on Answer Text and Code

[2]Word2Vec abbreviations: W2V1 (Python Reserved Words), W2V2 (Common Python Code Tokens), W2V3 (Tokenized All Text and Code)

Table A1: Miscellaneous Experiments.