Natural Language Processing
Project 3: Question Answering System

Team members:
Anant Agarwal (aa2387), Deekshith Belchappada (db786), Pooja RaviShankar (pr438), Taru Saraswat (ts752)

Brief overview of our QA approach:
Following steps summarize the QA approach we followed (detailed steps along with justification are provided in next section):

1. Preparation of the data corpus for the given documents, such that sentences can be retrieved from this corpus based upon their question ID, document ID and the sentence index.
2. NER tagging of documents using Stanford NER tagger. We picked the tagger which tags with 7 different types. Sentences having words tagged as 'location' and 'organization' are considered while answering 'where' questions, words tagged as 'person' are considered while answering 'who' questions, words tagged as 'date' and 'time' are considered while answering 'when' questions.
3. We make a lookup list based on the NER tags for fast access of the relevant sentences at the time of answering the question.
4. After the above preprocessing we go through the questions one by one and try to answer them.
5. For each question we go through the lookup list and look at all the possible candidate answers and try to rank them and return the top 5.
6. The ranking is done using a linear combination of doc_score (score provided with each doc) and word match count.
7. Word match count is calculated by matching the words in the question and the context of the candidate answer. Stemming, stop words removal and lower casing is also done at the time of matching.

Brief Justification: As we need to answer questions of type "Where", "When", "Who" and the answer is supposed to be small, so we think that the words (or word chunks) tagged as "Location", "Organization", "Date", "Time", "Person" should form a good answer. Detailed justification of each small step in the process is provided in the next section.

QA Approach description and justification:
The approach is a continuation of the baseline system we implemented earlier. The improvements done over the baseline approach are detailed in the next section. Following steps describe our final QA system:

1. Data Reading:
    a. We extract the text data from the given format of documents and create a data corpus
    b. Schema of the corpus is:
```
{ <Qid>: [
            [<Sent1>, <Sent2>, <Sent3>,..],  //sentences corresponding to first document
            [ <Sent1>, <Sent2> … ],
            …
            [<Sent1>, .. ]
         ]
}
```

    Justification: We read all the documents for each question and store them, which can be used for any task. The idea is to prevent repetitive reads from disk. Any sentence can be read using Corpus[Qid][Doc_id][Sent_id], where Doc_id is the document number and Sent_id is the sentence number in the corresponding document.

2. Creation of Look-up lists:
    a. We generate lookup list at the time of generating corpus, corresponding to every Question.
    b. We rely on Stanford NER tagger for NER tagging to build the lookup list.

c. We use the 7 class tagger which tags each token of the sentence as {location, person , organization, date, time, money, percent, o}

d. Schema of the lookup dict is:

```
{ <Qid>: {
        "who": [ (<doc_id>, <sent_id>, candidate answer), ... ] ,
        "where": [ (<doc_id>, <sent_id>, candidate answer), ... ]
        "when": [ (<doc_id>, <sent_id>, candidate answer), ... ]
        }
}
```

e. We apply NER to each sentence and on finding a word insert a tuple in the above dict as follows:
   i. A word tagged as 'location', 'organization', is inserted in 'where' list with the corresponding doc_id and sentence id.
   ii. A word tagged as 'person' is inserted in 'who' list with the corresponding doc_id and sentence_id
   iii. A word_tagged as 'date', 'time' is inserted in 'when' list with the corresponding doc_id and sentence_id

f. Answers can have multiple words, but NER tagger splits the sentences into an array of words, and tags each word. So we combine continuous word sequences with same tag and treat them as one entity. This enables us to answer questions having multi word answers using the above lookup list.

Justification:
The idea is that the words tagged as 'location', 'organization', will be the set of candidate answers for 'where' type of questions. Similarly, for 'who' type of question the words tagged as 'person' will form the set of candidate answer, and for 'when' type of questions the set will be formed by the words tagged as 'date', 'time'. So we preprocess all the docs corresponding to each question, and store the candidate answers in advance. When a question needs to be answered, we scan through the candidate set of answers corresponding to the type of question and choose top 5 answers based on our ranking methodology.

3. Question processing:
   a. We read the question.txt and extract the Question Id, Question type and Question description for each question.
   b. These are passed to Answer Generation module below to generate the answers.
   c. answer.txt is generated using the answers returned by the answer generation module.

Justification: This module processes the question.txt and reads and links to the answer matching module.

4. Answer Generation:
   a. For each question, we look at the question type, viz. "Where", "When" and "Who".
   b. For every <Qid , Qtype> pair, we fetch the corresponding tuples from the lookup table.
      i. Every tuple contains doc_id, sent_id and candidate answer. (as shown in 2(d)) To rank all the candidate answers we do the following:
      ii. Fetch the context in which the candidate answer was present in the document
      iii. We specify the context span, which represents the number of sentences to be fetched. For e.g. if the span is 1 then only the sentence containing the answer will be fetched. If the span is 2, then sentences corresponding to sent_id-1 and sent_id will be fetched. If the span is 3, then sentences corresponding to sent_id-1, sent_id, sent_id_+1 will be fetched.
      iv. We tokenize the context, and question text.

v. Apply stemming on the context and the question text. NLTK PorterStemmer has been used for stemming.
vi. We remove stop words from both the context and the question text. NLTK stopword corpora has been used for stop words.
vii. We then count the number of words that are present in both the question and the context. This is done using set intersection.
viii. After the matches are counted we assign a score to each candidate answer as follows:
    1. doc_score*doc_score_weight + Match count
    2. doc_score is the score present on each of the document file.
    3. doc_score_weight is the importance we would like to give to the doc_score.
ix. To generate the top 5 answers we have the following variations:
    1. Pick the tuples corresponding to the highest score assigned in the above step
    2. Pick the top 2 answers based on the highest score assigned with a high doc_score_weight (say 0.9) and pick the next 3 purely based on the match count (i.e. assuming doc_score_weight 0)
x. While deciding the answers we also make sure that duplicate answers aren't present in our answer list.

c. We write the 5 answers obtained for every question in the answer file in the following format: <qid> <document id> <candidate answer>

Justification:

- Fetching more context: In few cases, just by looking at one sentence, it may not be possible to figure out the answer. Idea here is to look at some more surrounding sentences instead of just one sentence and get a better sentence match.
- Stemming and removing stop words: We depend on the count of words matching between the question and context as a parameter to decide the best answers. By stemming we ensure that words in question and context match even though there are grammatical differences. Stop words removal is important so that match word count doesn't include the stop words present in question text or in context.
- doc_score and doc_score_weight: The idea here is that a doc with a high score has a high probability of having the answer. Consider two candidate answers a1, a2, where a1 belongs to d1 with doc_score ds1, and a2 belongs to d2 with doc_score ds2 and ds1 > ds2. Let the match count for a1's context with question text be m1 and for a2 be m2. If m2 > m1 then we would still like to rank a1 better than a2 and ds1> ds2. To achieve this, in a generic fashion we use the function: *doc_score*doc_score_weight* + *Match count* to assign score to each candidate answer.
- Mix doc_score_weight approach: We also consider picking some answers with a doc_score_weight and other answers with a different doc_score_weight. In particular, we pick 2 answers by using a high doc_score_weight (close to 0.5) and 3 answers using a low doc_score_weight (0). The idea is to mix the best of both worlds, i.e. select few answers by giving extra importance to doc score and the rest based on maximum match count.

All the code has been implemented by ourselves, except Stanford NER tagger, NLTK PorterStemmer, NLTK stop words corpora, NLTK tokenizer.

Improvements over Baseline Model:
Following were the enhancements in our final version of QA system compared to our baseline model(as described in the proposal report)

1. Combining NER tags: In NER tagger words like "The International" are being tagged as [("The", ORGANIZATION), ("International", ORGANIZATION)]. The baseline model, was generating two possible candidate answer tuples for the above i.e. (doc_id, sent_id, "The"), (doc_id, sent_id,

"International"). Both the tuples were being ranked same by the answer ranker, hence killing one spot for the answer and also giving wrong answer. In the final system we combined the contiguous word sequences tagged with same NER tag. Resulting in generation of just one candidate answer tuple. Enabling us to answer questions which have more than one word answers and also freeing up one slot for an extra guess and hence increasing performance

.

2. Stemming of question text and document text: Application of stemming to question tokens and the document text at the time of counting matching words ensured a better match between words belonging to the same family of root words. This wasn't done in baseline system.

3. Removal of stop words from question text and document text: We also removed stop words from the question text and document text at the time of counting the word matches. This enables us to get a more accurate count of word matches. This wasn't done in baseline system.

4. Converting question text and document text to lowercase: In our baseline system, we were not converting all the words to lowercase before checking for word match count. Final version of our QA system fixed this gap and provided a flag to enable/disable this option for experimentation.

5. Removal of duplicates from answer list: We make sure that all the 5 answers we guess are unique, increasing our chances to be give out a correct answer. In the baseline system, this wasn't ensured, which was resulting in many duplicate guesses in the answer list.

6. Ranking answers based on document score: We also experimented with a new ranking idea to rank the candidate answer tuples. The motivation of this has been described in the previous section. The idea was to consider doc score along with the word match count to rank the tuples. The baseline system did ranking purely based on the word match count. We also tried a mixture approach, that is generating few guesses for a question by considering the doc score and few guesses without considering doc score.

7. Increasing the context: In our baseline system, we looked at only one sentence as per the data we get from tuple list while considering the candidate answers. In few cases, looking at sentences preceding and following the current sentence along with the current sentence can give better answers. We implemented this in our Final system which was not done in baseline and provided a counter to control number of sentences we should include along with the current sentence to find possible answers.

Detailed walk through for answering one question:

Following steps gives a detailed walk through for answering question number 90

Question: Who invented the paper clip?

Steps:

- By the time we reach question processing part, we will have following 3 data structures as described before:

  a) corpus:
  Data structure which stores all the sentences present in every document for all the questions. While finding answers for question number 90, we will be looking sentences stored in corpus['90']

  b) lookup_dict:
  Data structure which stores tuples list for 3 question types, 'when', 'where' and 'who'. To answer question 90, we will look at all the tuple list in lookup_dict['90']['who'] since question type is 'who'. First 10 elements of the list are shown below (tuple format is described earlier):

[(1, 0, 'Elisha G. Otis'), (1, 1, 'William LeBaron Jenny'), (1, 7, 'Vartan Gregorian'), (1, 13, 'Thomas Alva Edison'), (1, 14, 'James J. Ritty'), (1, 34, 'JOHN BARBOUR'), (2, 1, 'Johan Vaaler'), (2, 2, 'Langanker'), (3, 2, 'Elisha G. Otis'), (3, 3, 'William LeBaron Jenny designedthe')]

c) lookup_score :
Data structure which stores score of every document for all the questions. Lookup_score[90][doc_id] will give the document score for every doc_id present in folder 90. Some samples values are shown below:

[15.11, 14.84, 14.55, ...]

- Question processing module will pass question id, question type and question string to answering module. Question id here is 90, question type here is who, question string is ‘Who invented the paper clip’.
- Answer generating module:

a) Performs stemming and removes stopwords from the question text. Tokens after stemming and removing the stopwords for question 90 are shown below.

(['paper', 'clip', 'invent', '?'])

b) Picks the tuple list from lookup_dict[‘90’][‘who’]. For every tuple in this tuple list, it fetches the sentence from Corpus data structure. Following is a sentence fetched for a tuple having doc_id 2 and sent_index 1:

corpus['90'][2][1]:
"The paper clip, weighing a desk-crushing 1,320 pounds, is afaithful copy of Norwegian Johan Vaaler's 1899 invention, said PerLangaker of the Norwegian School of Management."

Output after stemming and removing stop words:
(['desk-crush', 'School', 'pound', "'s", 'said', 'clip', 'copi', ',', 'invent', '.', 'PerLangak', 'Norwegian', 'paper', '1899', '1,320', 'weigh', 'afaith', 'Johan', 'Manag', 'Vaaler'])

Finds common word match count:
In this example, it finds 3 common words.

Calculates weighted count based on document score:
Finds the doc score from lookup_score data structure and calculates the weighted score. Weighted score for this example is shown below.

lookup_score['90'][2]
14.842588
doc_score_weight: 0.5
Weighted score = 14.84*0.5+ 3 = 10.42

We make 2 copies of the answer tuple, one having weighted score and one with token match count.
We maintain 2 heaps, one to keep track of answer tuples with weighted score and one to keep track of answer tuples with just word match count. The tuples are pushed to their respective heaps.
As per our example, (10.421294, '90', 2, 'Johan Vaaler') gets pushed to heap maintaining weighted count and (3, '90', 2, 'Johan Vaaler') gets pushed to another heap.

c) After processing all the tuples from the tuple list, here are the top 20 tuples picked from each heap.

[(10.42, '90', 2, 'Johan Vaaler'), (9.42, '90', 2, 'Langanker'), (8.55, '90', 1, 'Thomas Alva Edison'), (8.55, '90', 1, 'James J. Ritty'), (8.55, '90', 1, 'Elisha G. Otis'), (8.27, '90', 3, 'Thomas Alva Edison'), (8.27, '90', 3, 'Elisha G. Otis'), (8.279173, '90', 3, 'Ambrose'), (7.881823, '90', 4, 'MARILYN GARDNER'), (7.55, '90', 1, 'William LeBaron Jenny'), (7.55, '90', 1, 'Vartan Gregorian'), (7.55, '90', 1, 'JOHN BARBOUR'), (7.23, '90', 3, 'William LeBaron Jenny designedthe'), (7.27, '90', 3, 'Vartan Gregorian'), (7.27, '90', 3, 'Rinaldo diVillamagna'), (7.27, '90', 3, 'Richard Jordan Gatling'), (7.27, '90', 3, 'Peter Henlein'), (7.27,

```
'90', 3, 'Mary Phelps Jacobinvented'), (7.27, '90', 3, 'Lucien Smith'), (7.27, '90', 3, 'James J. Rittyinvented')]

[(3, '90', 54, 'David'), (3, '90', 2, 'Johan Vaaler'), (2, '90', 84, 'Schott'), (2, '90', 78, 'Sammy Davis Jr.'), (2, '90', 78, 'McKinzie'), (2, '90', 78, 'Christensen'), (2, '90', 71, 'Moynihan'), (2, '90', 68, 'Howardand Bruce Guile'), (2, '90', 67, 'Mr Adham'), (2, '90', 67, 'Mr Adham'), (2, '90', 63, 'Shaye'), (2, '90', 58, 'Dangerfield'), (2, '90', 29, 'LUCY KELLAWAYWhen'), (2, '90', 28, 'Ernie'), (2, '90', 26, 'Harry Bryce'), (2, '90', 18, 'Wolfson'), (2, '90', 15, 'Reagan'), (2, '90', 15, 'Marlim Fitzwater'), (2, '90', 12, 'Steven'), (2, '90', 11, 'Franz Liszt')]
```

d) At first, answer generation module will pick top 2 tuples from heap maintaining weighted count. Then remaining 3 tuples will be picked from other heap. List of all these 5 tuples is returned to the question processing module. The 5 tuples returned to question processing module in our example are given below:

```
(10.42, '90', 2, 'Johan Vaaler'), (9.42, '90', 2, 'Langanker'), (3, '90', 54, 'David'),
(2, '90', 84, 'Schott'), (2, '90', 78, 'Sammy Davis Jr.')
```

- Question processing module will process these 5 tuples shown above and writes following 5 answer to answer.txt. And it turns out that the right answer for this question based on pattern.txt is "Johan Vaaler"

```
90 2 Johan Vaaler
90 2 Langanker
90 54 David
90 84 Schott
90 78 Sammy Davis Jr.
```

Result:

Baseline:

Mean reciprocal rank over 232 questions is 0.135
173 questions had no answers found in top 5 responses.

Final QA system:

Mean reciprocal rank over 232 questions is 0.210
149 questions had no answers found in top 5 responses.

Analysis:

Experiments on Dev Set and results:

We have provided following tunable parameters to perform various experiments on our final version of QA system.

max_rank_check: Number of documents to be consulted to find answer for a given question. A value of 7 will look at only top 7 ranked documents to find an answer.
doc_score_weight: How much of weightage should be given to doc_score. A value of 0.5 will multiply doc score weight by 0.5 and add it to match count.
lower_case_mode: Boolean variable which controls whether question text and sentences having answer should be converted to lowercase before checking for word match between them.
Context_span: Number of sentences to be considered in total while counting word match for a candidate answer tuple. Setting this to 3 will pick one sentence preceding and one sentence following the sentence containing the candidate answer.
Stemming: Whether stemming should be applied or not before checking for word match.
With 2 heaps: Experiments involved 2 heaps, one keeping track of weighted score and other keeping word match count. First 2 guesses for a question were based on the weighted score, and remaining 3 based on unweighted score.

Reversed order: 2 heap setup, first 3 guesses for a question based on unweighted score (just the word match count), and remaining 2 based on weighted score.

System Configuration/Type	MRR	Unanswered questions	Thoughts/Conclusion
Baseline max_rank_check: 7	0.135	173	Baseline performance, with 7 docs.
max_rank_check: 7 doc score weight: 0 lower case mode: 1 context span: 1 stemming: 1	0.198	156	Final system performance with 7 docs
max_rank_check: 100 doc score weight: 0 lower case mode: 1 context span: 1 stemming: 1	0.212	154	With 100 docs the performance of the QA system increases. This can be reasoned by the fact that more docs are considered for finding the answer. Conclusion: So we decided to stick with a Max Rank check of 100

max_rank_check = 100
doc_score_weight = 0
lower case mode = 1
context span = 4
stemming = 1
	0.151	169	We try increasing the context span, and observe that the performance of the system decreases.
			Conclusion: So we decide to stick to a context span of 1

max_rank_check: 100 doc score weight: 0.9 lower case mode: 1 context span: 1 stemming: 1	0.186	163	Next we try tweaking the ranking using the doc score, and we see that system performance decreases. This suggests that giving extra importance to doc score isn't a good idea.
max_rank_check: 100 doc score weight: 0.5 lower case mode: 1 context span: 1 stemming: 1	0.202	157	Based on the findings of the previous experiment, we try reducing the doc_score_weight to a smaller value of 0.5 which gives a better performance as compared to 0.9.
max_rank_check: 100 doc score weight:0.9 (2 heaps) lower case mode: 1 context span: 1 stemming: 1	0.202	149	We start experimenting with 2 heaps, i.e. guessing 2 answers based on weighted score and 3 based on match count. We see a performance improvement (number of questions answered) as compared to all previous system
max_rank_check: 100 doc score weight:0.5 (With 2 heaps) lower case mode: 1 context span: 1 stemming: 1	0.210	149	As the MRR went down in the previous experiment, so we tried experimenting with a lower doc_score_weight and we see that MRR increases and the questions answered remains same as with the weight of 0.9. Conclusion: Giving excessive importance to doc score isn't a good idea, but giving some importance to doc score is. As this system answers the most number of questions till

			now.
max_rank_check: 100 doc score weight: 0.5 (2 heaps) (reversed order) lower case mode: 1 context span: 1 stemming: 1	0.203	149	We wanted to experiment with the relative accuracy of answers generated using weighted word match count and answers generated using just the word match count. We reversed the order of guesses and placed the 3 unweighted answers first and then the 2 weighted answers. We see that the MRR reduced compared to the previous experiment.

How well did the system work?

- We have 232 questions in total and following table gives an analysis of how each question type is answered. We see a performance increase on all type of questions compared to baseline QA system.

Question Type	**Total count**	**Correct in Baseline**	**Correct in Final system**	**Baseline: % correct**	**Final system: % correct**
Who	113	31	41	27.4	30.8
When	48	11	17	22.9	35.4
Where	71	17	25	23.9	35.2

Which features improved the performance?

- NER tagging, doc score weight, 2 heap setup, duplicate guess removal, stemming, lower casing, stop words removal all of them worked in favour of improving the performance.
- Increasing Context span did not work out in favour of the improving the performance. Our understanding is that's because there weren't many contexts having inter-sentence dependencies which could be handled by this feature.
- Out of the things that worked in our favour, combining NER tags gave the major boost to the performance, second major contributor was the doc score weight with 2 heap system. Duplicate answer removal was the third major contributor. Stemming was fourth major. Lower casing and stop words removal had minor contribution to the performance improvement.

Contribution of team members:

Everyone came up with multiple ideas and then we zeroed in on the above implementation together as a team. The 4 modules discussed above were split for implementation as follows:

1. Data Reading and Corpus generation: Deekshith Belchappada (db786)
2. Creation of Lookup lists: Anant Agarwal (aa2387)
3. Question Processing: Pooja RaviShankar (pr438)
4. Answer Generation: Taru Saraswat (ts752)