**Natural Language Processing: Homework 6 Report**

a) **Names of group members:** Suvinay Bothra, Kartikey Sharma

**b) Overview of QA System:**

## Question Processing

There are many different types of questions and each type can help us identify the answer to a question. For example: a "where" question requires a place answer. Similarly, a "who" question requires a person to answer, a "when" question requires a time based answer, and "how" and "why" questions require an explanation, or a word that can provide an explanation. However, questions of the category "what" or "which" are more generic : they can warrant a person, place, explanation, or time.

In our processing, we wanted to determine what the type of question was: so, we traversed the entire question to look for "who", "when", and "where", "what", and "how" words in the question. We simply searched for the occurrence of one of these elements and considered it as the question type. For example, "In what place was Hitler buried" will be considered a "what" question. This approach can have edge cases, but a rule based approach cannot possibly cover all cases, unless all the cases are known (even then it will be difficult to implement given the sheer variations of questions). We considered using a Machine Learning approach to classify questions into categories, but did not implement it because of the run time and implementation time costs. However, we do think that a machine learning approach would better suit our tasks and this is an improvement we want to make in the future. Then we used NLTK to remove all the stop words from the question. A stop word is any word that commonly occurs in a language and does not add much to the semantic meaning of the question. For example "Suvinay is a man" will become "Suvinay man" where "is" and "a" do not add much meaning to a sentence. Then, we used lemmatization with NLTK using POS tags on our question to reduce a word inflection form to its base form (that has actual meaning in the language, unlike stemming)[1]. Consider the case, "Go went goes walks walking walked", all these will be converted to their base form as "Go go go walk walk walk", when we do this we do lose some information information about the tense and exact usage, later on we vectorize the questions to find similarity between questions and answers, and when we do this, the answer we are looking for may use the have the same base word but different usage. To illustrate this better, consider the case where we vectorize the question:"Who goes to walmart?", where the vector uses the unique words as features (mark 1

[1] https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

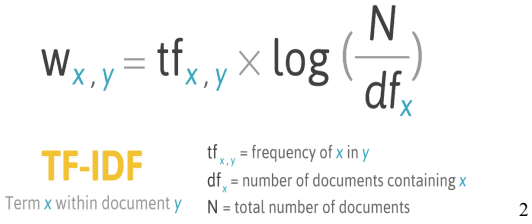
for presence, 0 otherwise) [ Who goes walmart king henry ] this question would have a vector value [ 1 1 1 0 0] where as the answer vector could be “Ruth went to walmart last friday”, now this vector would be represented as [0 0 1 0 0], however if lemmatization would be done the vector would consider the word “goes” as go, and the word “went” in the question as go, which would yield an answer vector of [0 1 1 0 0]; this would make the answer more similar and hence a more likely candidate to be the correct answer. However, it must be noted that it would also increase the similarity between question and candidate answers for all variations of the word “go” making many unviable answers more viable. But we came to conclude that the results will vary on a case by case basis, but, in general, more similar answer candidates will benefit the most from lemmatization as they will get the biggest push in similarity.

Then, we changed all words in the question to lowercase. This is because a lack of uniformity in case sensitivity will cause vectorization of words to be different for the same word with different casing: for example, for the sentence “Who is Lilly?” and the answer “i am lilly” none of the words will be considered the same in the vector since Lilly and lilly are different features due to case sensitivity, hence they will both increment a different index in the feature vector and as such we will lose useful information that will reduce the similarity for a deserving answer candidate otherwise.

Then, we vectorized every question. We implemented different features to find the approach that works best through experimentation, we used:

**1) Bag of words with binary values:** creates a vector of 0s with the length of the number of unique words in all the question, where each feature is a unique word in the set of all questions,when vectorizing, if a feature is present in a sentence set the value of the index associated to that feature as 1.

**2) Bag of words with normalized word frequency:** Same as the approach above, instead for each occurrence of a feature in the question, increment the value of the index associated to that feature by 1 and divide by the number of words in the question to normalize it so as to not provide a boost to larger questions. We thought this approach would better indicate occurrences compared to a simple bag of words matrix.



[2]

3) **TF-IDF:** This feature extraction technique initially creates a vector of 0s with the length of the number of unique words, for every review, then for every unique word that is present in the review being indexed the feature matrix values at that index is decided by the formula shown the image above. This technique is used by search engines and various IR systems to retrieve relevant results.[3]

## Passage Retrieval

Generally, information retrieval systems are used to retrieve relevant information from large chunks of data, for example, the web. However, for the purpose of this assignment, the relevant documents were provided to us and we used a TF-IDF feature that retrieval systems generally use. So, we did not use an information retrieval system. This is because we believe the dataset is not too large, and our computational formulation can find the most similar answers without relying on a IR system. For the training set we were provided the exact document number for where the answer occurs. We used a hashmap to store where each question's answer is in the provided XML files. We did this by associating each question number to its question number with a hashmap, and then we had access to a document called relevant_docs that mapped every question number to the document number where to find the answer. We used another hashmap to map every question number to the document number, and another hashmap to map every document number to its corresponding text. This allowed us to efficiently map (indirectly) every question to it's answer string for the training set. For the training set, we used a hashmap to link every question id to the entire XML parsed string (not just one document). Consequently, the search space for an answer is larger in the test set.

## Answer Processing

The answers to our questions were in XML documents. To parse them, we used simple string parsing line by line and extracted all document numbers with the help of regular expressions. We then read all the text related to a document as a string and parsed this string to

[2] https://ted-mei.medium.com/demystify-tf-idf-in-indexing-and-ranking-5c3ae88c3fa0
[3] https://www.geeksforgeeks.org/tf-idf-model-for-page-ranking/

remove all tags (like <text> etc.), we included the text in all the tokens because we think a lot of information contained in tags like <head> (for example the data and journal something was published in) is relevant information too. We then decided to split the string into sentences, rather than an arbitrary number like 20 token chunks. We did this because usually, a sentence can convey enough meaning to answer a factoid question. Different sentences vary in length, but convey enough meaning, and we are making an assumption that every sentence conveys an idea (while this is not always the case), it is a generalization that seemed reasonable based on structures of sentences and their semantic meaning. We also performed lemmatization and stemming for our answer sentences so that only relevant words would be vectorized and we would not lose information based on the variations of words. This however raises an issue: For example, if an acceptable answer is only "Los Angeles Lakers", lemmatization would make this "Los Angeles Laker", while this might not be too far off, it does not answer the question based on the requirements of this assignment. So we decided to link every lemmatized word to its history using the question number and index it occurs at.

We used the same concepts (bag of words, tf-idf) to make our feature vectors as we did for vectorizing the questions because we need to compare the questions to the answers and based on that we need to find the most similar vectors. Also, the features chosen in the vector were just the unique words from questions. The rationale for this choice is that we are only measuring a similarity between questions and answers, and having features based on all answers words will bring in noise. We decided to use cosine similarity for comparing all vectors over dot products, but the former normalizes the result by a magnitude and gives a clearer sense of similarity and reduces the range to -1 to 1 (from negative infinity to infinity) . After generating these feature vectors, we compared each question vector to their respective candidate answer vectors, and calculated a similarity score and used a heap to get the vectors with the 10 largest scores.

**Answer Extraction**

First, we categorized questions to narrow down the type of entity to return, this is explained in detail in the question processing section. We decided to use a named entity recognizer for extracting answers. For example, if the question was a "who" type question, the named entity recognizer was used to find a person entity. If the named entity recognizer was not able to identify any entity, we used POS tagging to return the noun phrases as answer words. We did this because it would probably be unreasonable to not extract any sort of answer from the

most similar sentence, and the fact that named entity recognizers have limitations and they do not always accurately categorize names and entities.

We used Spacy for the NER because it has functionality for things like Dates, numbers (money), people, places etc, compared to NLTK NER which does not include dates and times. Another design choice we made was that if there are multiple entities recognized by the NER in the current most similar sentence, for the provided sentence, we extracted all these entities and made them our candidate answers. For example if a sentence has entities: Rafael, Roger, Rooney. We made Rafael the first predicted answer, Roger the second predicted answer, and Rooney the third. Then we moved to the next most similar sentence and repeated the same process. We call this approach #Extraction1. While we do realize this approach may not be successful if a question asks for multiple entities: for eg. "who is the match between?" The answer would be "Nadal" and "Federer" but our system would only output a single entity, but regardless of the limitation we made this decision because by eyeballing our dataset, we could tell that most questions did not require multiple entity answers, and that choosing multiple answers from the most similar questions outweighs the chance of finding the right answer compared choosing answers from the more dissimilar sentences. Of course, this is not a general fact and just an assumption based on experimentation which gave us good results.

We also used an alternative approach which we call #Extraction2 which involved obtaining a list of named entities/noun phrases from the answer candidate sentences, and comparing their context (with a window size of 2, because 5 words is a long enough window to get some context) around them with the question using cosine similarity. We returned the 10 answer entities with the highest cosine similarity with the question as answers.

**Analysis**

We ran multiple experiments: First we ran our code with different features without changing anything else with Extraction Technique #1:

Binary Bag of Words : MRR - 0.059181 , Bag of Words Normalized word frequency : MRR-0.058509, TF-IDF : MRR - 0.144795.

These results were as expected, it did it little worse than the baseline system as shown in the spec because lemmatization and stop words were not used in this run. The finding showed that TF-IDF features performed significantly better than other features and this was expected because

tf-idf as a metric reflects how important each word is.

```
(base) Suvinays-MacBook-Pro:QASystem suvinay$ python evaluation.py ./training/qadata/answer_patterns.txt prediction.txt
qid 2: Correct guess "spices" at rank 9
qid 8: Correct guess "tritium" at rank 4
qid 14: Correct guess "south america" at rank 4
qid 18: Correct guess "bethel" at rank 1
qid 20: Correct guess "lexington" at rank 3
qid 21: Correct guess "brinkley" at rank 1
qid 26: Correct guess "cincinnati" at rank 4
qid 29: Correct guess "1936" at rank 2
qid 30: Correct guess "william gibson" at rank 1
qid 36: Correct guess "james boswell" at rank 2
qid 47: Correct guess "pirates" at rank 5
qid 48: Correct guess "parkinson" at rank 8
qid 50: Correct guess "leprechaun" at rank 2
qid 52: Correct guess "stoll" at rank 2
qid 56: Correct guess "carnival cruise lines" at rank 9
qid 62: Correct guess "henson" at rank 1
qid 73: Correct guess "joe penny" at rank 4
qid 74: Correct guess "the netherlands" at rank 4
qid 94: Correct guess "varivax" at rank 5
qid 101: Correct guess "princeton university press" at rank 10
qid 102: Correct guess "alberta" at rank 8
qid 104: Correct guess "1985" at rank 1
qid 105: Correct guess "fruits" at rank 2
qid 106: Correct guess "elizabeth ii" at rank 1
qid 108: Correct guess "sheriff pat garrett" at rank 2
qid 111: Correct guess "david kirk" at rank 1
qid 112: Correct guess "1976" at rank 1
qid 114: Correct guess "seuss" at rank 5
```

The result above is from a tf-idf feature matrix run, while it shows many answers were ranked 1 or 2, many were also ranked very low at about 8, 9, 10. This made us introspect our answer extraction and made us switch to #Extraction2, since we felt we our approach to answering "what" questions was based on randomly choosing noun phrases. Another finding was that, instead of simply using binary features, gave us better results than using the frequency of occurrences of words and normalizing them. This was unlike what we expected, since increased occurrence of a feature should be favored, however the difference was very slight, so we would need a larger dataset to concretely argue which feature is better.

After this, we used extraction strategy #Extraction2 on TF-IDF features and it reduced the performance to 0.105919, and for bag of words with normalized word frequency it reduced performance to 0.055614. This was surprising, because we thought strategy 2 made more sense in terms of extraction answers, but this probably means this strategy does not fit our data well and perhaps is not a very good strategy since it reduced the performance for bag of words slightly and tf-idf significantly. In the future we would like to employ better answer extraction techniques because a look at our answer vectors indicated that it would be possible to find a lot more correct answers if our extraction techniques were superior. Overall, 0.144795 was our best MRR.

## Citations


1. Prabhakaran, S. Lemmatization Approaches with examples in Python. Machine learning plus : https://www.machinelearningplus.com/nlp/lemmatization-examples-python/ We used the code from section 3 of this article to use advanced lemmatization with nltk
2. Upadhyay P. Removing stop words with NLTK in Python. GeeksforGeeks(05/18/2020): https://www.geeksforgeeks.org/removing-stop-words-nltk-python we used the approach described in the article to remove stop words
3. Mei T. Demystify TF-IDF in indexing and ranking. Medium: https://ted-mei.medium.com/demystify-tf-idf-in-indexing-and-ranking-5c3ae88c3fa0
4. Mikolov et al. Efficient Estimation of Word Representations in Vector Space. https://arxiv.org/pdf/1301.3781.pdf
5. http://www.tfidf.com/ For tf-idf explanation.
6. Rit19. Tf-Idf model for page ranking. GeeksforGeeks (08/07/2019) https://www.geeksforgeeks.org/tf-idf-model-for-page-ranking/ to understand tf-idf
7. https://spacy.io/usage/linguistic-features#named-entities For NER explanation.
8. Used Scikit learn library to get CountVectorizer, TfIdfVectorizer, cosine similarity
9. Used NLTK for POS tagging, lemmatization, stopwords, word and sentence tokenization.
10. Used Spacy library to get the NERecognizer class.


## Appendix

**Suvinay:** Wrote the report. Did research on various usable features like tf-idf, word2vec, BERT etc, Provided team leadership and organized tasks to give direction to the team. Implemented the methods: getData(), parseQuestions(), parseRelevantDocs(), getXmlDict(), removeStopWords() corpusCounts(),normalizedWordFrequencyMatrix(), cosineSimilarity(), getTopSimilar(), buildVocab(), writeToFile(), lemmatize(), addPosTags(),def get_word_net(), getQnA(), and integrated all chunks. In terms of coding, read and parsed .txt files, converted XML files to string for Kartikey to parse, associated questions, their id's , their relevant document number, and their text , corresponding answer document to dictionaries. Implemented preprocessing like removing stop words and lemmatization. Generated feature matrix for Bag of words (binary and normalized word frequency) and Tf-IDF. Wrote the driver and getTopSimilar that compares every question vector to answer vectors for similarity and returns top candidates. Wrote the similarity methods and wrote code to produce the output file.

**Kartikey**: Conducted research on using word2vec, its relevance and its usage details as well as named entity recognition alternatives in Python. Wrote classes to implement feature extraction using gensim word2vec model (which we ended up not using), as well as a named entity recognizer using spacy, including code for answer extraction. Furthermore, wrote code to parse relevant text from the topdoc.x files. This included the following methods:
W2V class: getAvgWordVec(), getAvgWordVecFromList(), constructor logic
NERecognizer class: constructor logic, getEntities(), getAnsCandidates(), getAnsFromQuestionList() (answer extraction code.), getAnsFromQuestionListWithContext()
XML parsing: genXMLFromTopDocs()

Worked on stylistic choices and debugging as well.