Natural Language Processing

LI Yongqi
csyqli@comp.polyu.edu.hk

Lab 2 Document-based Question Answering

Objective

This lab is to introduce the task of document-based question answering (machine reading comprehension), including the background, the basic concepts, the benchmarks as well as the main approaches. These benchmarks and main approaches are closely related to the group project this semester. After this lab, students should be able to learn basic question answering techniques.

Session 1: IR-based Question Answering

Two paradigms

The quest for knowledge is deeply human, and so it is not surprising that practically as soon as there were computers we were asking them questions.

By the early 1960s, systems used the two major paradigms of question answering—**information retrieval-based** and **knowledge-based** --- to answer questions about baseball statistics or scientific facts.

In 2011, IBM's Watson question-answering system won the TV game-show *Jeopardy!* using a hybrid architecture that surpassed humans at answering questions like

> WILLIAM WILKINSON'S "AN ACCOUNT OF THE PRINCIPALITIES OF WALLACHIA AND MOLDOVIA" INSPIRED THIS AUTHOR'S MOST FAMOUS NOVEL

Most question answering systems focus on factoid questions, questions that can be answered with simple facts expressed in short texts. The answers to the questions below can be expressed by a personal name, temporal expression, or location:

(1) Who founded Virgin Airlines?
(2) What is the average age of the onset of autism?
(3) Where is Apple Computer based?

In this session we describe the two major paradigms for factoid question answering. **Information-retrieval** or **IR-based** question answering relies on the vast quantities of textual information on the web or in collections like PubMed. Given a user question, information retrieval techniques first find relevant documents and passages. Then systems (feature-based, neural, or both) use **reading comprehension** algorithms to read these retrieved documents or passages and draw an answer directly from spans of text. Thus, these systems are called document-based question answering systems.
In the second paradigm, **knowledge-based question answering**, a system instead builds a semantic representation of the query, mapping *What states border Texas?* to the logical representation. These meaning representations are then used to query databases of facts. Finally, large industrial systems like the DeepQA system in IBM's Watson are often hybrids, using both

text datasets and structured knowledge bases to answer questions. DeepQA finds many candidate answers in both knowledge bases and in textual sources, and then scores each candidate answer using knowledge sources like geospatial databases, taxonomical classification, or other textual sources.

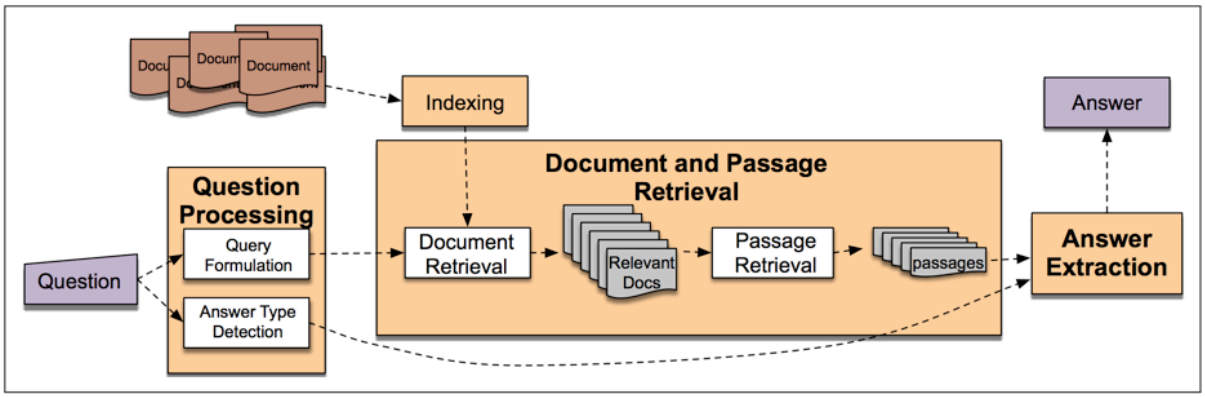
In this lab, we focus on describing IR-based approaches including neural reading comprehension systems, followed by a discussion of evaluation.

IR-based Factoid Question Answering

The goal of information retrieval based question answering is to answer a user's question by finding short text segments on the web or some other collection of documents. The following block shows some sample factoid questions and their answers.

Question	Answer
Where is the Louvre Museum located?	in Paris, France
What's the abbreviation for limited partnership?	L.P.
What are the names of Odin's ravens?	Huginn and Muninn
What currency is used in China?	the yuan
What kind of nuts are used in marzipan?	Almonds
What instrument does Max Roach play?	Drums
What's the official language of Algeria?	Arabic
How many pounds are there in a stone?	14

The below figure shows the three phases of an IR-based factoid question-answering system: question processing, passage retrieval and ranking, and answer extraction.



Question Processing

The main goal of the question-processing phase is to extract the **query**: the keywords passed to the IR system to match potential documents. Some systems additionally extract further information such as:

- **answer type**: the entity type (person, location, time, etc.) of the answer.
- **focus**: the string of words in the question that is likely to be replaced by the answer

in any answer string found.
• **question type**: is this a definition question, a math question, a list question?

For example, for the question *Which US state capital has the largest population?* the query processing might produce:

query: "US state capital has the largest population"
answer type: city
focus: state capital

In the next two sections we summarize the two most commonly used tasks, query formulation and answer type detection.

Query Formulation

Query formulation is the task of creating a query—a list of tokens— to send to an information retrieval system to retrieve documents that might contain answer strings.

For question answering from the web, we can simply pass the entire question to the web search engine, at most perhaps leaving out the question word (*where*, *when*, etc.). For question answering from smaller sets of documents like corporate information pages or Wikipedia, we still use an IR engine to index and search our documents, generally using standard TF-IDF cosine matching, but we might need to do more processing. For example, for searching Wikipedia, it helps to compute TF-IDF over bigrams rather than unigrams in the query and document (Chen et al., 2017).

Or we might need to do query expansion, since while on the web the answer to a question might appear in many different forms, one of which will probably match the question, in smaller document sets an answer might appear only once. Query expansion methods can add query terms in hopes of matching the particular form of the answer as it appears, like adding morphological variants of the content words in the question, or synonyms from a thesaurus.

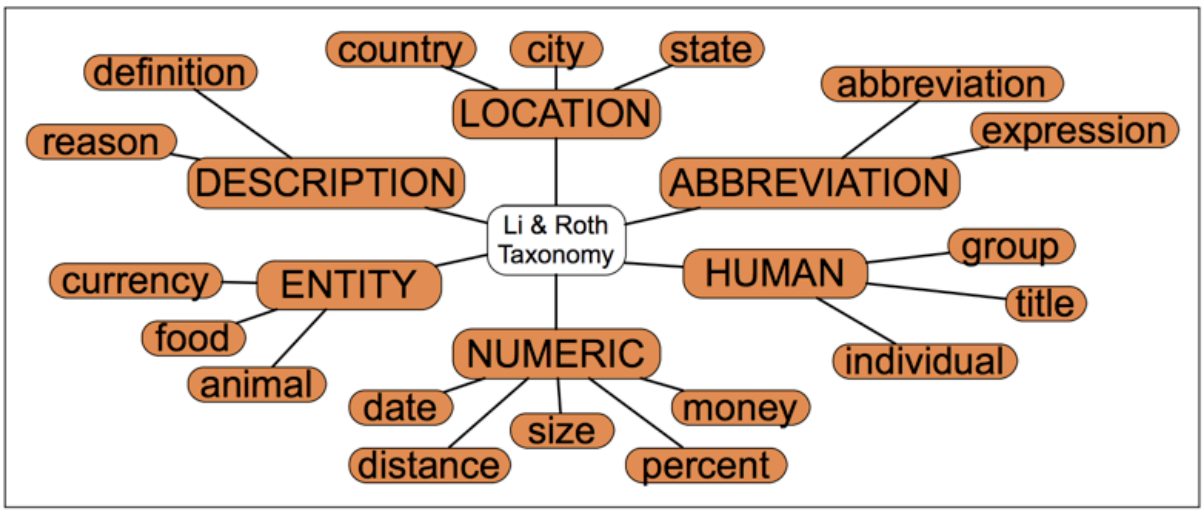
A query formulation approach that is sometimes used for questioning the web is to apply query reformulation rules to the query. The rules rephrase the question to query reformulation make it look like a substring of possible declarative answers. The question "*when was the laser invented?*" might be reformulated as "*the laser was invented*"; the question "*where is the Valley of the Kings?*" as "*the Valley of the Kings is located in*". Here are some sample handwritten reformulation rules from Lin (2007):

(4) *wh-word* did A *verb* B → . . . A *verb+ed* B
(5) Where is A → A is located in

Answer Types

Some systems make use of **question classification**, the task of finding the **answer type**, the named-entity categorizing the answer. A question like "*Who founded Virgin Airlines?*" expects an answer of type PERSON. A question like "*What Canadian city has the largest population?*" expects an answer of type CITY. If we know that the answer type for a question is a person, we can avoid examining every sentence in the document collection, instead focusing on sentences mentioning people.

While answer types might just be the named entities like PERSON, LOCATION, and ORGANIZATION, we can also use a larger hierarchical set of answer types called an answer type taxonomy. Such taxonomies can be built **answer type taxonomy** automatically, from resources like WordNet (Harabagiu et al. 2000, Pasca 2003), or they can be designed by hand. A subset is shown in below.



In this hierarchical tagset, each question can be labeled with a coarse-grained tag like HUMAN or a finegrained tag like HUMAN:DESCRIPTION, HUMAN:GROUP, HUMAN:IND, and so on. The HUMAN:DESCRIPTION type is often called a BIOGRAPHY question because the answer is required to give a brief biography of the person rather than just a name.

Question classifiers can be built by hand-writing rules like the following rule from (Hovy et al., 2002) for detecting the answer type BIOGRAPHY:

(6) who {is | was | are | were} PERSON

Most question classifiers, however, are based on supervised learning, trained on databases of questions that have been hand-labeled with an answer type (Li and Roth, 2002). Either feature-based or neural methods can be used. Feature based methods rely on words in the questions and their embeddings, the part-of-speech of each word, and named entities in the questions. Often, a single word in the question gives extra information about the answer type, and its identity is used as a feature. This word is sometimes called the **answer type word** or **question headword**, and may be defined as the headword of the first NP after the question's *wh-word*; headwords are indicated in boldface in the following examples:

(7) Which **city** in China has the largest number of foreign financial companies?

(8) What is the state **flower** of California?

In general, question classification accuracies are relatively high on easy question types like PERSON, LOCATION, and TIME questions; detecting REASON and DESCRIPTION questions can be much harder.

Session 2: Document-based Question Answering

Document and Passage Retrieval

The IR query produced from the question processing stage is sent to an IR engine, resulting in a set of documents ranked by their relevance to the query. Because most answer-extraction methods are designed to apply to smaller regions such as paragraphs, QA systems next divide the top n documents into smaller passages such as sections, paragraphs, or sentences. These might be already segmented in the source document or we might need to run a paragraph segmentation algorithm.

The simplest form of **passage retrieval** is then to simply pass along every passage to the answer extraction stage. A more sophisticated variant is to filter the passages by running a named entity or answer type classification on the retrieved passages, discarding passages that don't contain the answer type of the question. It's also possible to use supervised learning to fully rank the remaining passages, using features like:

- The number of **named entities** of the right type in the passage
- The number of **question keywords** in the passage
- The longest exact sequence of question keywords that occurs in the passage
- The rank of the document from which the passage was extracted
- The **proximity** of the keywords from the original query to each other (Pasca 2003, Monz 2004).
- The number of **n-grams** that **overlap** between the passage and the question (Brill et al., 2002).

Sentence Similarity

To locate the answers, sometimes we want to compare the similarities between the query and a sentence in the document. We call this task as **sentence similarity**. Besides the sentence similarity features we introduce in the first lab, below are other three useful sentence similarity models that are well capsuled in the python package **gensim**.

```
import gc
import tqdm
import numpy as np
from gensim import corpora, models, similarities
from sentence import Sentence
from collections import defaultdict
class SentenceSimilarity():

    def __init__(self, seg):
        self.seg = seg

    def set_sentences(self, sentences):
        self.sentences = []
        for i in range(0, len(sentences)):
            self.sentences.append(Sentence(sentences[i], self.seg, i))

    # obtain tokenized sentences
    def get_cuted_sentences(self):
        cuted_sentences = []
```

```
        for sentence in self.sentences:
            cuted_sentences.append(sentence.get_cuted_sentence())

        return cuted_sentences

    # for high-level complicated model
    def simple_model(self, min_frequency = 1):
        self.texts = self.get_cuted_sentences()

        # remove low-frequent words
        frequency = defaultdict(int)
        for text in self.texts:
            for token in text:
                frequency[token] += 1
        self.texts = [[token for token in text if frequency[token] > min_frequency]
for text in self.texts]
        self.dictionary = corpora.Dictionary(self.texts)
        self.corpus_simple = [self.dictionary.doc2bow(text) for text in self.texts]

    # tfidf
    def TfidfModel(self):
        self.simple_model()

        # transform model
        self.model = models.TfidfModel(self.corpus_simple)
        self.corpus = self.model[self.corpus_simple]

        # create similarity matrix
        self.index = similarities.MatrixSimilarity(self.corpus)

    # lsi
    def LsiModel(self):
        self.simple_model()

        # transform model
        self.model = models.LsiModel(self.corpus_simple)
        self.corpus = self.model[self.corpus_simple]

        # create similarity matrix
        self.index = similarities.MatrixSimilarity(self.corpus)


    # lda
    def LdaModel(self):
        self.simple_model()

        # transform model
        self.model = models.LdaModel(self.corpus_simple)
        self.corpus = self.model[self.corpus_simple]

        # create similarity matrix
        self.index = similarities.MatrixSimilarity(self.corpus)

    # preprocessing new sentences
    def sentence2vec(self, sentence):
        sentence = Sentence(sentence, self.seg)
        vec_bow = self.dictionary.doc2bow(sentence.get_cuted_sentence())
```

```
        return self.model[vec_bow]

    def bow2vec(self):
        vec = []
        length = max(self.dictionary) + 1
        for content in self.corpus:
            sentence_vectors = np.zeros(length)
            for co in content:
                sentence_vectors[co[0]] = co[1]
            vec.append(sentence_vectors)
        return vec

    # find the most similar sentences
    # input: test sentence
    def similarity(self, sentence):
        sentence_vec = self.sentence2vec(sentence)

        sims = self.index[sentence_vec]
        sim = max(enumerate(sims), key=lambda item: item[1])

        index = sim[0]
        score = sim[1]
        sentence = self.sentences[index]

        sentence.set_score(score)
        return sentence


    def similarity_k(self, sentence, k):
        sentence_vec = self.sentence2vec(sentence)

        sims = self.index[sentence_vec]
        sim_k = sorted(enumerate(sims), key=lambda item: item[1], reverse=True)[:k]

        indexs = [i[0] for i in sim_k]
        scores = [i[1] for i in sim_k]
        return indexs, scores
```

Practice

1. Using LDA models in the gensim to compute the sentence similarities. Input your own data.

Session 3: Answer Extraction

The final stage of question answering is to extract a specific answer from the passage, for example responding 29,029 feet to a question like "How tall is Mt. Everest?". This task is commonly modeled by span labeling: given a passage, identifying the span of text which constitutes an answer.

A simple baseline algorithm for answer extraction is to run a named entity tagger on the candidate passage and return whatever span in the passage is the correct answer type. Thus, in the following examples, the underlined named entities would be extracted from the passages as the answer to the HUMAN and DISTANCE-QUANTITY questions:

"Who is the prime minister of India?"
Manmohan Singh, Prime Minister of India, had told left leaders that the deal would not be renegotiated.
"How tall is Mt. Everest?"
The official height of Mount Everest is 29029 feet

Unfortunately, the answers to many questions, such as DEFINITION questions, don't tend to be of a particular named entity type. For this reason modern work on answer extraction uses more sophisticated algorithms, generally based on supervised learning. The next section introduces a simple feature-based classifier, after which we turn to modern neural algorithms.

Feature-based Answer Extraction

Supervised learning approaches to answer extraction train classifiers to decide if a span or a sentence contains an answer. One obviously useful feature is the answer type feature of the above baseline algorithm. Hand-written regular expression patterns also play a role, such as the sample patterns for definition questions in below.

Pattern	Question	Answer
<AP> such as <QP>	What is autism?	", developmental disorders such as autism"
<QP>, a <AP>	What is a caldera?	"the Long Valley caldera, a volcanic crater 19 miles long"

Other features in such classifiers include:

Answer type match: True if the candidate answer contains a phrase with the correct answer type.
Pattern match: The identity of a pattern that matches the candidate answer.
Number of matched question keywords: How many question keywords are contained in the candidate answer.
Keyword distance: The distance between the candidate answer and query keywords.
Novelty factor: True if at least one word in the candidate answer is novel, that is, not in the query.
Apposition features: True if the candidate answer is an appositive to a phrase containing many question terms. Can be approximated by the number of question terms separated from the candidate answer through at most three words and one comma (Pasca, 2003).
Punctuation location: True if the candidate answer is immediately followed by a comma, period, quotation marks, semicolon, or exclamation mark.
Sequences of question terms: The length of the longest sequence of question terms that occurs in the candidate answer.

N-gram Tiling Answer Extraction

An alternative approach to answer extraction, used solely in Web search, is based **n-gram tiling** on n-gram tiling, an approach that relies on the **redundancy** of the web (Brill et al. 2002, Lin 2007). This simplified method begins with the snippets returned from the Web search engine, produced by a reformulated query.

In the first step, n-gram mining, every unigram, bigram, and trigram occurring in the snippet is extracted and weighted. The weight is a function of the number of snippets in which the n-gram occurred, and the weight of the query reformulation pattern that returned it. In the n-gram filtering step, n-grams are scored by how well they match the predicted answer type. These scores are computed by handwritten filters built for each answer type.

Finally, an n-gram tiling algorithm concatenates overlapping n-gram fragments into longer answers. A standard greedy method is to start with the highest-scoring candidate and try to tile each other candidate with this candidate. The best-scoring concatenation is added to the set of candidates, the lower-scoring candidate is removed, and the process continues until a single answer is built.

Neural network approaches to answer extraction draw on the intuition that a question and its answer are semantically similar in some appropriate way. As we'll see, this intuition can be fleshed out by computing an embedding for the question and an embedding for each token of the passage, and then selecting passage spans whose embeddings are closest to the question embedding. We will focus on these approaches in the later session.

In the last part of this session, we will give some code snippets for you to mine linguistic features for answer extraction. Take the following features for example:

- **Length of sentence**
- **Number of Named Entities**
- **Sentence position in the doc**: sentences in introductions and summaries are more likely to be relevant. The problem is that after conversion, the docs do not have much of a structure, they are basically a huge string. It should be possible to recover some structure based on number of white lines and words such as 'section', 'chapter', '1.' etc.
- **Number of Upper Case words**: often special terminology or names
- **Number of nouns, verbs and adjectives** (lexical semantics)

```
def count_k_important(sent):
    count = 0
    for w in sent:
        if w.lower_ in top_k_words:
            count+=1
    return count

def eliminate_non_english_words(s):
    """takes list of words and eliminates all words that contain non-english
characters, digits or punctuation"""
    english_words = []
    for word in s:
        if word.lower() in authors:
            english_words.append(word)
        else:
            try:
                word.encode(encoding='utf-8').decode('ascii')
                # if re.sub('-', '', word).isalpha():
                    #             english_words.append(re.sub('[%s]'             %
re.escape(string.punctuation), '', word))
```

```
                word = re.sub('[%s]' % re.escape(string.punctuation), '', word)
                if word.isalpha():
                    english_words.append(word)
            except UnicodeDecodeError:
                pass
    return ' '.join(english_words)

def calculate_named_entities(sent):
    count = 0
    entities = []
    for ent in sent.ents:
        count+=1
        entities.append((ent.label_, ent.text))
    return count, entities

def calculate_pos(sent):
    n = 0
    v = 0
    a = 0
    for w in sent:
        if w.pos_ == 'VERB':
            v+=1
        if w.pos_ == 'ADJ':
            a += 1
        if w.pos_ == 'NOUN':
            n+=1
    return n, v, a

def calculate_upper(sent):
    counter = -1
    for w in sent:
        if not w.is_lower:
            counter += 1
    return counter
```

```
def calculate_features(li):
    """takes a list of texts and returns df with sentences and their features"""
    all_sentences = []
    for a in li:
        posi = 0
        for sent in nlp(a).sents:
            out = [str(sent), len(list(sent))]
            upper = calculate_upper(sent)
            sent = nlp(eliminate_non_english_words(str(sent).split()))
            named_entities, _ = calculate_named_entities(sent)
            k_important = count_k_important(sent)
            pos = 100/len(list(nlp(a).sents))*posi
            nouns, verbs, adjectives = calculate_pos(sent)
            out+=[named_entities, k_important, pos, upper, nouns, verbs,
adjectives]
            all_sentences.append(out)
            posi += 1
    features = pd.DataFrame(all_sentences, columns=['sentence', 'length',
'named_entities', 'k_important', 'position',
```

```
                                                    'upper', 'nouns', 'verbs',
'adjectives'])
  return features
```

Practice

1. Input a list of sentences (or a document), and calculate the aforementioned features using the code above. Summarize the output features in a form. Please see the output format as:

	Sentence	Length	#NE	Position	#Upper	#Nouns	#Verbs	#ADJs
0								
1								
2								
3								

2. Read more here: https://github.com/alvercau/Q-A-System/tree/master/notebooks

Session 4: Reading Comprehension

Neural answer extractors are often designed in the context of the reading comprehension task. It was Hirschman et al. (1999) who first proposed to take children's reading comprehension tests—pedagogical instruments in which a child is given a passage to read and must answer questions about it—and use them to evaluate machine text comprehension algorithm. They acquired a corpus of 120 passages with 5 questions each designed for 3rd-6th grade children, built an answer extraction system, and measured how well the answers given by their system corresponded to the answer key from the test's publisher.

Since then reading comprehension has become both a task in itself, as a useful way to measure natural language understanding performance, but also as (sometimes called the reader component of question answerers).

SQuAD

Modern reading comprehension systems tend to use collections of questions that are designed specifically for NLP, and so are large enough for training supervised learning systems.

For example the Stanford Question SQuAD Answering Dataset (SQuAD) consists of passages from Wikipedia and associated questions whose answers are spans from the passage, as well as some questions that are designed to be unanswerable (Rajpurkar et al. 2016, Rajpurkar et al. 2018); a total of just over 150,000 questions. Fig. 25.6 shows a (shortened) excerpt from a SQUAD 2.0 passage together with three questions and their answer spans.

Beyoncé Giselle Knowles-Carter (born September 4, 1981) is an American singer, songwriter, record producer and actress. Born and raised in Houston, Texas, she performed in various singing and dancing competitions as a child, and rose to fame in the late 1990s as lead singer of R&B girl-group Destiny's Child. Managed by her father, Mathew Knowles, the group became one of the world's best-selling girl groups of all time. Their hiatus saw the release of Beyoncé's debut album, Dangerously in Love (2003), which established her as a solo artist worldwide, earned five Grammy Awards and featured the Billboard Hot 100 number-one singles "Crazy in Love" and "Baby Boy".
Q: "In what city and state did Beyoncé grow up?" A: "Houston, Texas"
Q: "What areas did Beyoncé compete in when she was growing up?" A: "singing and dancing"
Q: "When did Beyoncé release Dangerously in Love?" A: "2003"

SQuAD was built by having humans write questions for a given Wikipedia passage and choose the answer span. Other datasets used similar techniques; the NewsQA dataset consists of 100,000 question-answer pairs from CNN news articles. For other datasets like WikiQA the span is the entire sentence containing the answer (Yang et al., 2015); the task of choosing a sentence rather than a smaller answer span is sometimes called the sentence selection task.

Group Project Datasets

We provide three multiple-choice document-based question answering dataset to evaluate your approaches, i.e., MCTest, RACE, and DREAM. Their basic statistics and required QA skills are presented below.

	#Docs	#Questions	#Words/Doc	#Words/Q	Required Skill
MCTest	500	2,000	205	8.0	Matching
RACE	27,933	97,687	323	10.0	Matching Multi-Sentence Reasoning
DREAM	6,444	10,197	86	8.6	Matching Multi-Sentence Reasoning Commonsense Reasoning

From the table, we can see:

1. MCTest is the smallest benchmark, while DREAM is the second large dataset, and RACE is the largest. It will require a relatively long time to run your system on RACE and DREAM datasets.
2. Besides the dataset size, these three datasets require different levels of skills for QA systems to answer correctly. MCTest is the simplest dataset where the majority of the questions can be handled using word matching techniques. RACE is more difficult since answering 60% of its questions require summarizing multiple evidences across the documents, i.e., the so-called multi-hop reasoning skill. Besides the textual information in the document (dialogue), 30% questions in DREAM require additional commonsense knowledge that cannot be obtained from the document (dialogue). Hence, DREAM is the most difficult one.

MCTest

MCTest is a freely available set of stories and associated questions intended for research on the machine comprehension of text.

Previous work on machine comprehension (e.g., semantic modeling) has made great strides, but primarily focuses either on limited-domain datasets, or on solving a more restricted goal (e.g., open-domain relation extraction). In contrast, MCTest requires machines to answer **multiple-choice** reading comprehension questions about fictional stories, directly tackling the high-level goal of open-domain machine comprehension.

Reading comprehension can test advanced abilities such as causal reasoning and understanding the world, yet, by being multiple-choice, still provide a clear metric. By being fictional, the answer typically can be found only in the story itself. The stories and questions are also carefully limited to those a young child would understand, reducing the world knowledge that is required for the task.

MCTest presents the scalable crowd-sourcing methods that allow us to cheaply construct a dataset of *500 stories and 2000 questions*. By screening workers (with grammar tests) and stories (with grading), we have ensured that the data is the same quality as another set that we manually edited, but at one tenth the editing cost. By being open-domain, yet carefully restricted, we hope MCTest will serve to encourage research and provide a clear metric for advancement on the machine comprehension of text. Below is an example of MCTest.

James the Turtle was always getting in trouble. Sometimes he'd reach into the freezer and empty out all the food. Other times he'd sled on the deck and get a splinter. His aunt Jane tried as hard as she could to keep him out of trouble, but he was sneaky and got into lots of trouble behind her back. One day, James thought he would go into town and see what kind of trouble he could get into. He went to the grocery store and pulled all the pudding off the shelves and ate two jars. Then he walked to the fast food restaurant and ordered 15 bags of fries. He didn't pay, and instead headed home. His aunt was waiting for him in his room. She told James that she loved him, but he would have to start acting like a well-behaved turtle. After about a month, and after getting into lots of trouble, James finally made up his mind to be a better turtle.

1) What is the name of the trouble making turtle?
A) Fries
B) Pudding
C) James
D) Jane

2) What did James pull off of the shelves in the grocery store?
A) pudding
B) fries
C) food
D) splinters

3) Where did James go after he went to the grocery store?
A) his deck
B) his freezer
C) a fast food restaurant
D) his room

4) What did James do after he ordered the fries?
A) went to the grocery store
B) went home without paying
C) ate them
D) made up his mind to be a better turtle

RACE

Although questions in MCTest are of high-quality ensured by careful examinations through crowdsourcing, it contains only 500 stores and 2000 questions, which substantially restricts its usage in training advanced machine comprehension models. Later some researchers present **RACE**, a new dataset for benchmark evaluation of methods in the reading comprehension task.

Collected from the **English exams for middle and high school** Chinese students in the age range between 12 to 18, RACE consists of near *28,000 passages and near 100,000 questions* generated by human experts (English instructors), and covers a variety of topics which are carefully designed for evaluating the students' ability in understanding and reasoning. Moreover, while MCTest is designed for 7 years old children, RACE is constructed for middle and high school students at 12–18 years old hence is more complicated and requires stronger reasoning skills. In other words, RACE can be viewed as a larger and more difficult version of the MCTest dataset.

In particular, the proportion of questions that **requires reasoning** is much larger in RACE than that in other benchmark datasets for reading comprehension, and there is a significant gap between the performance of the state-of-the-art models (43%) and the ceiling human performance (95%). The dataset is freely available at http://www.cs.cmu.edu/~glai1/data/race/ and the code is available at https://github.com/qizhex/RACE_AR_baselines.

Below is an example from RACE.

Passage:
In a small village in England about 150 years ago, a mail coach was standing on the street. It didn't come to that village often. People had to pay a lot to get a letter. The person who sent the letter didn't have to pay the postage, while the receiver had to.
"Here's a letter for Miss Alice Brown," said the mailman.

" I'm Alice Brown," a girl of about 18 said in a low voice.
Alice looked at the envelope for a minute, and then handed it back to the mailman.
"I'm sorry I can't take it, I don't have enough money to pay it", she said.
A gentleman standing around were very sorry for her. Then he came up and paid the postage for her.
When the gentleman gave the letter to her, she said with a smile, " Thank you very much, This letter is from Tom. I'm going to marry him. He went to London to look for work. I've waited a long time for this letter, but now I don't need it, there is nothing in it."
"Really? How do you know that?" the gentleman said in surprise.
"He told me that he would put some signs on the envelope. Look, sir, this cross in the corner means that he is well and this circle means he has found work. That's good news."
The gentleman was Sir Rowland Hill. He didn't forgot Alice and her letter.
"The postage to be paid by the receiver has to be changed," he said to himself and had a good plan.
"The postage has to be much lower, what about a penny? And the person who sends the letter pays the postage. He has to buy a stamp and put it on the envelope." he said . The government accepted his plan. Then the first stamp was put out in 1840. It was called the "Penny Black". It had a picture of the Queen on it.

Questions:
1): The first postage stamp was made_____.
A. in England B. in America C. by Alice D. in 1910

2): The girl handed the letter back to the mailman because_____.
A. she didn't know whose letter it was
B. she had no money to pay the postage
C. she received the letter but she didn't want to open it
D. she had already known what was written in the letter

3): We can know from Alice's words that_____.
A. Tom had told her what the signs meant before leaving
B. Alice was clever and could guess the meaning of the signs
C. Alice had put the signs on the envelope herself
D. Tom had put the signs as Alice had told him to

4): The idea of using stamps was thought of by_____.
A. the government B. Sir Rowland Hill C. Alice Brown D. Tom

Answer: ADABC

RACE dataset is distinct in its reasoning types of the questions. To get a comprehensive picture about the reasoning difficulty requirement of RACE, we conduct human annotations of questions types. Following Chen et al. (2016); Trischler et al. (2016), the authors stratify the questions into five classes as follows with ascending order of difficulty:

• **Word matching**: The question exactly matches a span in the article. The answer is self-evident.
• **Paraphrasing**: The question is entailed or paraphrased by exactly one sentence in the passage. The answer can be extracted within the sentence.
• **Single-sentence reasoning**: The answer could be inferred from a single sentence of the article by recognizing incomplete information or conceptual overlap.
• **Multi-sentence reasoning**: The answer must be inferred from synthesizing information distributed across multiple sentences.
• **Insufficient/Ambiguous**: The question has no answer or the answer is not unique based on the given passage.

DREAM

DREAM is a multiple-choice Dialogue-based REAding comprehension exaMination dataset. In contrast to existing reading comprehension datasets, DREAM is the first to focus on in-depth multi-turn multi-party dialogue understanding.

DREAM contains *10,197 multiple choice questions for 6,444 dialogues*, collected from English-as-a-foreign-language examinations designed by human experts. DREAM is likely to present significant challenges for existing reading comprehension systems: 84% of answers are non-extractive, 85% of questions require reasoning beyond a single sentence, and 34% of questions also involve commonsense knowledge.

Below is an example from DREAM dataset.

Dialogue:
W: Tom, look at your shoes. How dirty they are! You must clean them.
M: Oh, mum, I just cleaned them yesterday.
W: They are dirty now. You must clean them again.
M: I do not want to clean them today. Even if I clean them today, they will get dirty again tomorrow.
W: All right, then.
M: Mum, give me something to eat, please.
W: You had your breakfast in the morning, Tom, and you had lunch at school.
M: I am hungry again.
W: Oh, hungry? But if I give you something to eat today, you will be hungry again tomorrow.

Q1 Why did the woman say that she wouldn't give him anything to eat?

A. Because his mother wants to correct his bad habit.
B. Because he had lunch at school.
C. Because his mother wants to leave him hungry.

Compared to existing datasets built from formal written texts, the vocabulary size of DREAM is relatively small since spoken English by its nature makes greater use of high-frequency words and needs a smaller vocabulary for efficient real-time communication (Nation, 2006).

The authors categorize questions into two main categories according to the types of knowledge required to answer them: matching and reasoning.

Matching

A question is entailed or paraphrased by exactly one sentence in a dialogue. The answer can be extracted from the same sentence. For example, we can easily verify the correctness of the question-answer pair ("What kind of room does the man want to rent?", "A two-bedroom apartment.") based on the sentence "M: I'm interested in renting a two-bedroom apartment". This category is further divided into two categories word matching and paraphrasing in previous work (Chen et al., 2016; Trischler et al., 2017).

Reasoning

Questions that cannot be answered by the surface meaning of a single sentence belong to this category. The author further define four subcategories as follows.

– *Summary* Answering this kind of questions requires the whole picture of a dialogue, such as the topic of a dialogue and the relation between speakers. Under this category, questions such as "What are the two speakers talking about?" and "What are the speakers probably doing?" are frequently asked.

– *Logic* We require logical reasoning to answer questions in this category. We usually need to identify logically implied relations among multiple sentences in a dialogue. To reduce the ambiguity during the annotation, we regard a question that can only be solved by considering the content from multiple sentences and does not belong to the summary subcategory that involves all the sentences in a dialogue as a logic question.

– *Arithmetic* Inferring the answer requires arithmetic knowledge.

– *Commonsense* To answer questions under this subcategory, besides the textual information in the dialogue, we also require additional commonsense knowledge that cannot be obtained from the dialogue.

Practice

1. Please compare SQuAD, MCTest, RACE, DREAM datasets with respect to data size, document length, question length, as well as reasoning skills.

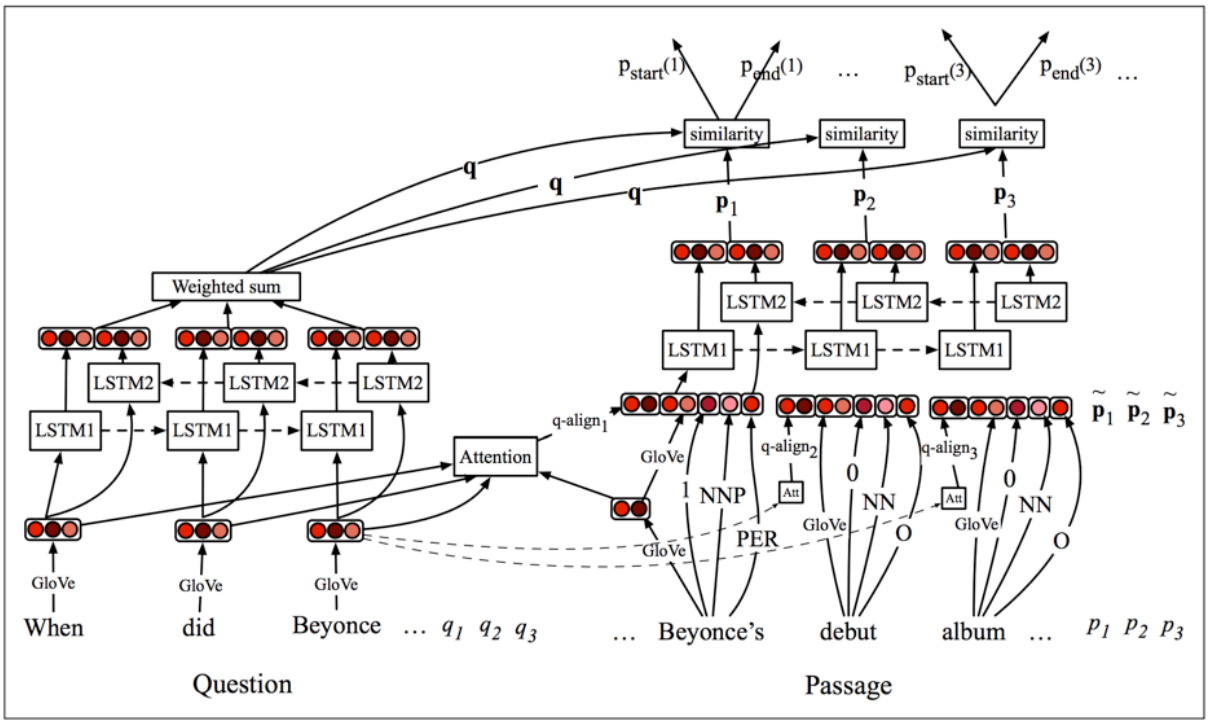
Session 5: Neural Answer Extraction

Neural algorithms for reading comprehension are given a question q of l tokens $q_1,...,q_l$ and a passage p of m tokens $p_1,..., p_m$. Their goal is to compute, for each token pi the probability $p_{start}(i)$ that p_i is the start of the answer span, and the probability $p_{end}(i)$ that p_i is the end of the answer span. Or, it is to select the most probable candidate from the multiple choices.

A bi-LSTM-based Reading Comprehension Algorithm

The below figure shows the architecture of the Document Reader component of the DrQA system of Chen et al. (2017). Like most such systems, DrQA builds an embedding for the

question, builds an embedding for each token in the passage, computes a similarity function between the question and each passage word in context, and then uses the question-passage similarity scores to decide where the answer span starts and ends.



Let's consider the algorithm in detail, following closely the description in Chen et al. (2017). The question is represented by a single embedding $\boldsymbol{q}$, which is a weighted sum of representations for each question word q_i. It is computed by passing the series of embeddings $\mathbf{E}(q_1),...,\mathbf{E}(q_l)$ of question words through an RNN (such as a bi-LSTM). The resulting hidden representations $\{q_1,...,q_l\}$ are combined by a weighted sum $\mathbf{q} = \sum_j b_j \mathbf{q}_j$

The weight b_j is a measure of the relevance of each question word, and relies on a learned weight vector $\mathbf{w}$: $b_j = \frac{exp(\mathbf{w} \cdot \mathbf{q}_j)}{\sum_{j'} exp(\mathbf{w} \cdot \mathbf{q}'_j)}$

To compute the passage embedding $\{\mathbf{p}_1,...,\mathbf{p}_m\}$ we first form an input representation by concatenating four components: $\tilde{p} = \{\tilde{\mathbf{p}}_1,...,\tilde{\mathbf{p}}_m\}$

- An embedding for each word $\mathbf{E}(p_i)$ such as from GLoVE (Pennington et al., 2014).
- Token features like the part of speech of p_i, or the named entity tag of p_i, from running POS or NER taggers.
- Exact match features representing whether the passage word pi occurred in the question. Separate exact match features might be used for lemmatized or lower-cased versions of the tokens.
- Aligned question embedding: In addition to the exact match features, many QA systems use an attention mechanism to give a more sophisticated model of similarity between the passage and question words, such as similar but non-identical words like *release* and *singles*. For example a weighted similarity $\sum_j a_{i,j} \mathbf{E}(q_j)$ can be used, where the attention weight $a_{i,j}$ encodes the similarity between p_i and each question word q_j. This attention weight can be computed as the dot product between functions α of the word embeddings of the question and passage:

$$q_{i,j} = \frac{\exp(\alpha(\mathbf{E}(p_i)) \cdot \alpha(\mathbf{E}(q_j)))}{\sum_{j'} \exp(\alpha(\mathbf{E}(p_i)) \cdot \alpha(\mathbf{E}(q'_j)))}$$

$\alpha(\cdot)$ can be a simple feed forward network.

We then pass the input into a biLSTM:

$$\{\mathbf{p}_1, ..., \mathbf{p}_m\}) = RNN(\{\tilde{\mathbf{p}}_1, ..., \tilde{\mathbf{p}}_m\}$$

The result of the previous two steps is a single question embedding **q** and a representation for each word in the passage $\{\mathbf{p}_1,...,\mathbf{p}_m\}$. In order to find the answer span, we can train two separate classifiers, one to compute for each p_i the probability $p_{start}(i)$ that p_i is the start of the answer span, and one to compute the probability $p_{end}(i)$. While the classifiers could just take the dot product between the passage and question embeddings as input, it turns out to work better to learn a more sophisticated similarity function, like a bilinear attention layer **W**:

$$p_{start}(i) \propto exp(\mathbf{p}_i \mathbf{W}_s \mathbf{q})$$
$$p_{end}(i) \propto exp(\mathbf{p}_i \mathbf{W}_e \mathbf{q})$$

These neural answer extractors can be trained end-to-end by using datasets like SQuAD.

Stanford Attentive Reader

```
def build_fn(args, embeddings):
    """
        Build training and testing functions.
    """
    in_x1 = T.imatrix('x1')
    in_x2 = T.imatrix('x2')
    in_mask1 = T.matrix('mask1')
    in_mask2 = T.matrix('mask2')
    in_l = T.matrix('l')
    in_y = T.ivector('y')

    l_in1 = lasagne.layers.InputLayer((None, None), in_x1)
    l_mask1 = lasagne.layers.InputLayer((None, None), in_mask1)
    l_emb1 = lasagne.layers.EmbeddingLayer(l_in1, args.vocab_size,
                                           args.embedding_size, W=embeddings)

    l_in2 = lasagne.layers.InputLayer((None, None), in_x2)
    l_mask2 = lasagne.layers.InputLayer((None, None), in_mask2)
    l_emb2 = lasagne.layers.EmbeddingLayer(l_in2, args.vocab_size,
                                           args.embedding_size, W=l_emb1.W)

    network1     =     nn_layers.stack_rnn(l_emb1,     l_mask1,     args.num_layers,
args.hidden_size,
                                       grad_clipping=args.grad_clipping,
                                       dropout_rate=args.dropout_rate,
                                       only_return_final=(args.att_func == 'last'),
                                       bidir=args.bidir,
                                       name='d',
                                       rnn_layer=args.rnn_layer)
```

```
    network2     =     nn_layers.stack_rnn(l_emb2,     l_mask2,     args.num_layers,
args.hidden_size,
                                     grad_clipping=args.grad_clipping,
                                     dropout_rate=args.dropout_rate,
                                     only_return_final=True,
                                     bidir=args.bidir,
                                     name='q',
                                     rnn_layer=args.rnn_layer)

    args.rnn_output_size = args.hidden_size * 2 if args.bidir else args.hidden_size

    if args.att_func == 'mlp':
        att       =       nn_layers.MLPAttentionLayer([network1,       network2],
args.rnn_output_size,
                                              mask_input=l_mask1)
    elif args.att_func == 'bilinear':
        att      =      nn_layers.BilinearAttentionLayer([network1,      network2],
args.rnn_output_size,
                                                 mask_input=l_mask1)
    elif args.att_func == 'avg':
        att = nn_layers.AveragePoolingLayer(network1, mask_input=l_mask1)
    elif args.att_func == 'last':
        att = network1
    elif args.att_func == 'dot':
        att     =     nn_layers.DotProductAttentionLayer([network1,     network2],
mask_input=l_mask1)
    else:
        raise NotImplementedError('att_func = %s' % args.att_func)

    network = lasagne.layers.DenseLayer(att, args.num_labels,

nonlinearity=lasagne.nonlinearities.softmax)

    if args.pre_trained is not None:
        dic = utils.load_params(args.pre_trained)
        lasagne.layers.set_all_param_values(network,                dic['params'],
trainable=True)
        del dic['params']
        logging.info('Loaded pre-trained model: %s' % args.pre_trained)
        for dic_param in dic.iteritems():
            logging.info(dic_param)

    logging.info('#params:     %d'     %     lasagne.layers.count_params(network,
trainable=True))
    for layer in lasagne.layers.get_all_layers(network):
        logging.info(layer)

    # Test functions
    test_prob = lasagne.layers.get_output(network, deterministic=True) * in_l
    test_prediction = T.argmax(test_prob, axis=-1)
    acc = T.sum(T.eq(test_prediction, in_y))
    test_fn = theano.function([in_x1, in_mask1, in_x2, in_mask2, in_l, in_y], acc)

    # Train functions
    train_prediction = lasagne.layers.get_output(network) * in_l
    train_prediction = train_prediction / \
```

```
        train_prediction.sum(axis=1).reshape((train_prediction.shape[0], 1))
    train_prediction = T.clip(train_prediction, 1e-7, 1.0 - 1e-7)
    loss      =      lasagne.objectives.categorical_crossentropy(train_prediction,
in_y).mean()
    #       TODO:       lasagne.regularization.regularize_network_params(network,
lasagne.regularization.l2)
    params = lasagne.layers.get_all_params(network, trainable=True)

    if args.optimizer == 'sgd':
        updates = lasagne.updates.sgd(loss, params, args.learning_rate)
    elif args.optimizer == 'adam':
        updates = lasagne.updates.adam(loss, params)
    elif args.optimizer == 'rmsprop':
        updates = lasagne.updates.rmsprop(loss, params)
    else:
        raise NotImplementedError('optimizer = %s' % args.optimizer)
    train_fn = theano.function([in_x1, in_mask1, in_x2, in_mask2, in_l, in_y],
                               loss, updates=updates)

    return train_fn, test_fn, params
```

Practice

1. Find more QA models here: https://github.com/l11x0m7/Question_Answering_Models