PAPER—SUPPLEMENTAL

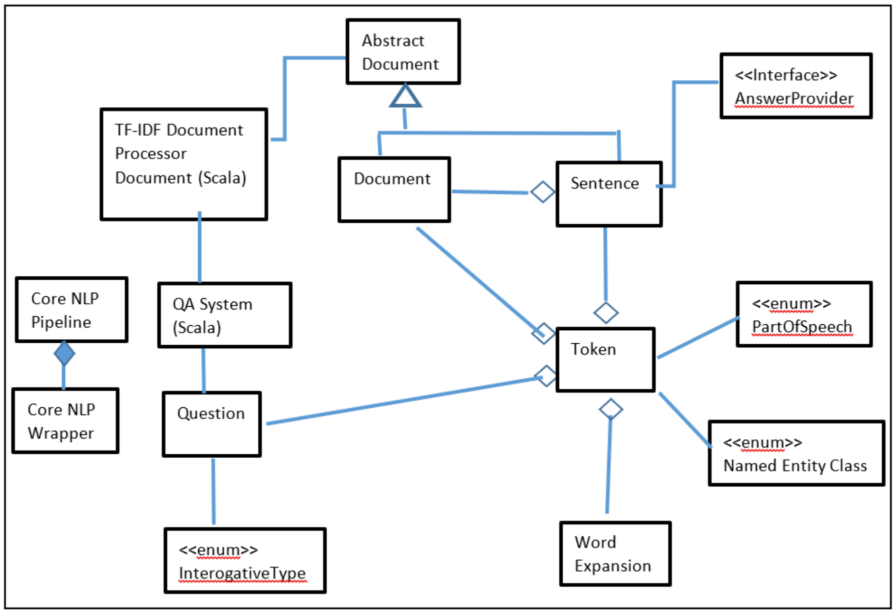
Belinda Copus, CS 5560 Knowledge Discovery and Management, Summer 2017

BACKGROUND

This paper is an informal update to the paper titled, Applying Knowledge Discovery and Question/Answering to Intelligent Tutoring System, submitted during Summer, 2017. The objective was to develop a natural language question answering system as a proof-of-concept for potential inclusion in a future intelligent tutoring system (ITS) for beginning programming students. A substantial amount of work was done on the QA system in 2017, leading up to the paper presented before. The work was re-engaged since that time, and further developments and enhancement have been made to the system. This paper documents those changes and the results obtained, and suggests areas of subsequent research and development.

ARCHITECTURE

The structural abstractions produced in the earlier version of the QA system remained essentially unchanged as additional features were added to document processing and answer extraction. I feel that the early work I put into the logical decomposition of the system into appropriate class abstractions is a significant strength of my QA system implementation, because it is both coherent and extensible. As I did not find the Stanford `CoreNLP` abstractions to be satisfactory in this regard, I built my own abstractions to encapsulate logical entities like Documents, Sentences, and Tokens. These provide a clean interface to Stanford `CoreNLP` that is easy and intuitive to use in client code. Documents and Sentences derive from the abstract class, `AbstractDocument`, so that common functionality is refactored into the abstract parent class. I had also provided a wrapper class for the `CoreNLP` pipeline, but this class ended up being a much thinner abstraction layer and, unlike the Document and Sentence abstractions, is probably superfluous.



The system is divided into two components: Document Processing and Question Answering. Both are implemented on Apache Spark in Scala, with auxiliary processing code in Java. The first version of the QA system relied more heavily on Java because of my familiarity with the language. As Scala familiarity has increased, I have relied more and more on Scala, since it is typically much more concise than Java and is well-suited to applications involving data pipelines.

QA System Updates

N-Gram

I implemented N-Grams in this project with N = 2. It is my belief that there are only marginal gains to be expected from a larger size for N, since a 3-Gram would be resolved to two N-gram matches in my system, which would have an additive effect in any case.

Word2Vec

Word2Vec was implemented in document processing (to produce the Word2Vec model) and in question answering. When a question is posed, the system first determines the type of the interrogative:

```
private InterrogativeType extractInterrogativeType() {
  //  given a token with pos == "WP" or pos == "WRB", extract its lemma, and
  //  use it to determine the question type.
  for(Token tok : tokens) {
    if (tok.getPos() == PartOfSpeech.WP || tok.getPos() == PartOfSpeech.WRB) {
      switch (tok.getLemma().toLowerCase()) {
        case "who"   : return InterrogativeType.WHO;
        case "what"  : return InterrogativeType.WHAT;
        case "when"  : return InterrogativeType.WHEN;
        case "where" : return InterrogativeType.WHERE;
        case "why"   : return InterrogativeType.WHY;
      }//switch
    }
  }
  return InterrogativeType.UNKNOWN;
}//extractInterrogativeType()
```

Based on the type of the interrogative, a set of target terms is derived. Target terms will be used during sentence scoring, in which candidate sentences that might contain an answer to the question are ranked. The target terms are limited to nouns and verbs as being most relevant in ranking answers, given the kind of factual content that exists in our corpus.

```
private Token[] extractTargetTerms() {
  //collect the nouns and verbs from the question
  ArrayList<Token> targets = new ArrayList<>();
  for(Token tok : tokens)
    if(tok.isNoun() || tok.isVerb())
      targets.add(tok);
  return targets.toArray(new Token[targets.size()]);
}//extractTargetTerms()
```

After this, the terms are expanded using the Word2Vec model to include synonyms. I allowed the three closest synonyms to be included in the expansion:

```
public static String[] expand(String word) {
  Tuple2<String, Object>[] synonyms;
  try {
    synonyms = w2vModel.findSynonyms(word, 3);
  } catch (java.lang.IllegalStateException ex) {
    return null;
  }
  String[] rval = new String[synonyms.length];
  for(int i = 0; i < rval.length; i++) {
    rval[i] = synonyms[i]._1;
  }
  return rval;
}//expand()
```

LDA

The LDA model was constructed during document processing, with a vocabulary size of about 3,000 words (less stop words, very short words, and words with strange characters) and 20 topic clusters. The number of topic clusters is the result of some research I performed into appropriate cluster sizes from available literature, most of which is anecdotal in nature. Each sentence in the corpus was assigned a single, “best” topic from the twenty topics.

In question answering, the lemmas in the question were compared with the top lemmas in each topic, and a score accumulated. The topic with the highest score for that question was determined to be that question’s best topic.

Where a candidate sentence (determined by TF/IDF) and a question has a matching best topic, the sentence’s score was increased by a confidence scaling multiple greater than 1.0.

Thus, the whole sentence scoring algorithm became:

```
public double score(Question question) {
  double score = 0.0;
  int[] indices = this.getTfIdfVector().indices();
  double[] values = this.getTfIdfVector().values();

  for(int hash : question.getTargetTermHashes()) {
    int index = Arrays.binarySearch(indices, hash);
    if(index >= 0)
      score += values[index];
  }

  for(int i = 0; i < question.getTargetNgrams().length; i++)
    for(int j = 0; j < ngrams.length; j++)
      if(ngrams[j].matchesNgram2(question.getTargetNgrams()[i]))
        score *= NGRAM_MATCH_MULTIPLIER;
  if(mainTopic != NO_TOPIC)
    if(mainTopic == question.mainTopic())
      score *= LDA_TOPIC_MATCH_MULTIPLIER;
  return score;
}//score()
```

Technologies Considered but Not Implemented

OpenIE and WordNet were investigated for this QA system implementation. As both target conceptual relationships in a broad and general context, it was determined that they would not be a good fit for the kind of specialized conceptual relationships found in computer science textbooks. For that reason, I chose to focus my implementation on the statistical, rather than knowledge-based, technologies for this iteration.

Document Preparation

Prior to programmatic document preparation, the corpus was prepared by selecting computer science programming texts, exporting to plain text, and dividing into chapters. Each chapter represents a "Document" to the QA system, and each newline-terminated string within each chapter represents a "Sentence".

The document preparation engine is separate from the question-answering system but relies on the same structural base classes and NLP pipeline. When document processing begins, all the text documents in the corpus subdirectory are scanned, and TF/IDF vectors are generated. Sentences are then also processed in the same way, so that each sentence has a representative vector. A Word2Vec model is then generated from the corpus, and LDA topic processing proceeds with 20 topics specified. Each sentence is then further processed to determine its closest topic match according to the LDA model. Finally, the Word2Vec model and LDA model are saved for future use by the QA system, along with the Documents and Sentences RDDs.

Document processing is implemented in Scala using Java classes and the Java-based Stanford CoreNLP pipeline. The Scala code is included below.

```
def main(args: Array[String]): Unit = {
  System.setProperty("hadoop.home.dir", "C:\\winutils")
  val sparkConf = new SparkConf().setAppName("QASystem").setMaster("local[*]")
  val sc: SparkContext = new SparkContext(sparkConf)

  FileUtils.deleteDirectory(new File("preparedCorpus"))
  new File("preparedCorpus").mkdir()

  val files = sc.wholeTextFiles("corpus")

  //val stops : Broadcast[Array[String]] =sc.broadcast(sc.textFile("data/stopwords.txt").collect())

  //
  //Process the corpus documents with full annotation;
  //  Save the TF-IDF data per-document and per-sentence;
  //  Save documents and sentences in object files for subsequent use.
  //    First: The documents
  //
  val documents: Array[Document] = files.map(file => CoreNlpWrapper.prepareText(file._2)).toArray()
  val docsRDD : RDD[AbstractDocument] = sc.parallelize(documents)

  //
  //      Calculate TF-IDF for documents and save the vector in each document
  //
  val dd: Seq[AbstractDocument] = processElements(docsRDD)
```

```
//
//    Second: the sentences
//
val sentences = documents.flatMap(doc => doc.getSentences)
val sentencesRDD : RDD[AbstractDocument] = sc.parallelize(sentences)

val ss: Seq[AbstractDocument] = processElements(sentencesRDD)
//defer saving sentenced object file until LDA topic extraction has been performed

//
//...Done with TF/IDF processing
//

//
//Use Word2Vec to create a vector space from the corpus for subsequent use
//
val w2vecInput : RDD[Seq[String]] = sc.parallelize(sentences.map(
  s => s.getTokens.map(t => t.getLemma).toSeq
).toSeq)
val word2vec : Word2Vec = new Word2Vec().setVectorSize(300)
val model: Word2VecModel = word2vec.fit(w2vecInput)
model.save(sc, "preparedCorpus/w2vec")

//
//Prepare for LDA topic extraction
//

val params: Params = Params(ss, 20, "em")

val topic_output: PrintStream = new PrintStream("preparedCorpus\\LDAResults.txt")
val (corpus, vocabArray, actualNumTokens) = preprocess(sc, params.input)
corpus.cache()
val actualVocabSize = vocabArray.length

//
// Run LDA topic extraction
//
val lda = new LDA()

lda.setOptimizer(new EMLDAOptimizer)
  .setK(params.k)
  .setMaxIterations(20)

//val cvm : CountVectorizerModel = new CountVectorizerModel(corpus)
val ldaModel: LDAModel = lda.run(corpus)

//
// Save the LDA model for later use by the QA system
//
ldaModel.save(sc, "preparedCorpus\\LDA")
sc.parallelize(vocabArray).saveAsObjectFile("preparedCorpus\\vocab.obj")
```

```
    //
    // Apply LDA to extract best topics for each sentence, and save the sentences and document
    // to object files
    //
    Models.setLDAModel(ldaModel)
    //Models.setVocabulary(vocabArray)
    val ti = ldaModel.describeTopics(maxTermsPerTopic = math.min(actualVocabSize, 50))
    val topWords: Array[Array[String]] = ti.map {
      case (terms, termWeights) => terms.zip(termWeights).map {
        case (term, weight) => vocabArray(term.toInt)
      }
    }

    extractBestTopics(sc, ss, topWords)
    val tw: Seq[Seq[String]] = topWords.map(topic => topic.toSeq).toSeq
    sc.parallelize(tw).saveAsObjectFile("preparedCorpus\\ldaTopicsTopWords.obj")
    sc.parallelize(ss).saveAsObjectFile("preparedCorpus\\sentences.obj")
    sc.parallelize(dd).saveAsObjectFile("preparedCorpus\\documents.obj")
    //
    // End of LDA processing
    //

  }//main()

  private def preprocess(sc: SparkContext, docs: Seq[AbstractDocument]): (RDD[(Long, Vector)], Array[String], Long)
= {
    val stopWords: Array[String] =sc.textFile("data/stopwords.txt").collect()
    val stopWordsBroadCast: Broadcast[Array[String]] =sc.broadcast(stopWords)

    val filteredDocs: RDD[Array[String]] = sc.parallelize(
      docs.map(
      (doc: AbstractDocument) => doc.getTokens.map((tok: Token) => tok.getLemma)
         .filterNot((lemma: String) => stopWordsBroadCast.value.contains(lemma.toLowerCase))
        .filterNot((lemma: String) => lemma.contains("[^a-zA-Z]".r))
        .filterNot((lemma : String) => lemma.length <= 3)
      )
    )
    filteredDocs.cache()

    val dfseq: RDD[Seq[String]] = filteredDocs.map(_.toSeq)
    dfseq.cache()

    val hashingTF = new HashingTF(filteredDocs.count().toInt)
    val tf = hashingTF.transform(dfseq)
    tf.cache()

    val idf = new IDF().fit(tf)
    val tfidf = idf.transform(tf).zipWithIndex().map(_.swap)

    val dff = dfseq.flatMap(f => f)
    val vocab: Array[String] = dff.distinct().collect()

    (tfidf, vocab, dff.count()) // Vector, Vocab, total token count
  }//preprocess

  private def extractBestTopics(sc: SparkContext, ss: Seq[AbstractDocument], topWords: Array[Array[String]]): Unit =
{
    ss.foreach((sentence: AbstractDocument) => {
      val topicAffinities = ListBuffer[Int]()
      val lemmas = sentence.getTokens.map((tok: Token) => tok.getLemma)
      topWords.foreach(topic => {
        val accum = sc.accumulator(0)
        lemmas.foreach(lemma => {
          topic.foreach( keyWord =>
            if(keyWord == lemma)
              accum += 1
          )
        })
        topicAffinities += accum.value
      })
      val bestTopic = {
        if (topicAffinities.forall(score => score == 0))
          AbstractDocument.NO_TOPIC
        else
          topicAffinities.zipWithIndex.maxBy(_._1)._2
      }
      sentence.setMainTopicFromLDA(bestTopic)
    })
  }

  def processElements(rDD: RDD[AbstractDocument]): Seq[AbstractDocument] = {
    //process each document into a sequence of lemmatized tokens
    val elementSequences =
      rDD.map(doc => {
        doc.getTokens.map(tok => {
          tok.getLemma
        }).toSeq
      })
    elementSequences.foreach(doc => println(doc))

    //calculate TF-IDF data
    val hashingTF = new HashingTF()
    val termFrequencies = hashingTF.transform(elementSequences)
    termFrequencies.cache()

    val idf = new IDF().fit(termFrequencies) //returns IDFModel
    val tfIdf = idf.transform(termFrequencies) //returns TF-IDF SparseVectors

    //collect elements and TF-IDF data and save the vectors in each element
    val dd = rDD.collect()
    val tt = tfIdf.collect()

    for (i : Int <- 0 until rDD.count().toInt) {
      dd.apply(i).setTfIdfVector(tt.apply(i).toSparse)
    }
    return dd
  }//processElements()
```

ANSWER EXTRACTION

Answer extraction has become somewhat more sophisticated in this iteration of the QA system. I begin by selecting the whole text of the sentence as the proposed answer. In some cases, that will be the best that the system can do. However, for certain question types, we may be able to improve the answer localization.

For example, “Why” questions are best answered by a “because” clause, so I limit the response to the “because” clause in the sentence, if one exists.

Other question types, such a “what,”, “when,” “where,” and “who” may best be answered by an appropriate named entity. In those cases, I establish a list of the kinds of named entities that might answer the type of question that has been asked, and then scan the document, looking for those kinds of named entities. If found, the named entity provides the answer. It is important to note that the corpus—and this kind of content in general—does not lend itself to much named entity extraction. There is further discussion of the observed impact from this later in the paper.

Finally, some questions seek a definition, e.g., “what is…” questions. In those cases, the system seeks an is-clause, and returns the appropriate portion of the sentence that appears to deliver a definition.

```
  public Answer extractAnswer(Question question) {
    //construct the default answer to return if we cannot improve upon it
    Answer answer = new Answer(this.getOriginalText(), Answer.QUALITY_VERY_LOW);

    //For "why" questions, look for a "because" clause...
    if(question.getInterrogativeType() == InterrogativeType.WHY) {
      Token[] tokens = this.getTokens();
      int becauseIndex = -1;
      for (int i = 0; i < tokens.length; i++) {
        if(tokens[i].getPos()==PartOfSpeech.IN && tokens[i].getLemma().toLowerCase().equals("because"))
          becauseIndex = i;
      }
      if(becauseIndex >= 0) {
        StringBuffer sb = new StringBuffer();
        for (int i = becauseIndex; i < tokens.length; i++) {
          sb.append(tokens[i].getText() + " ");
        }
        answer.setAnswerText(sb.toString()).setAnswerQuality(Answer.QUALITY_VERY_HIGH);
      }
      return answer;
    }
    //For other question types, look for an appropriate named entity...
    ArrayList<NamedEntityClass> objectives = new ArrayList<>();
    switch (question.getInterrogativeType()) {
      case WHO   :
      case WHAT  :
        objectives.add(NamedEntityClass.PERSON);
        objectives.add(NamedEntityClass.ORGANIZATION);
        break;
    case WHEN  :
      objectives.add(NamedEntityClass.DATE);
      objectives.add(NamedEntityClass.TIME);
      break;
    case WHERE :
      objectives.add(NamedEntityClass.LOCATION);
      break;
    default: return answer; //no idea what to extract, use default answer
  }

  //we have at least one target class to extract
  NamedEntityClass[] targets = new NamedEntityClass[objectives.size()];
  objectives.toArray(targets);
  Token[] tokens = this.getTokens();
  int start = -1, end = -1;
  for (int i = 0; i < tokens.length; i++) {
    for (int j = 0; j < targets.length; j++) {
      if(tokens[i].getNec() == targets[j]) {
        if (start < 0) {
          start = i;
        }
        end = i;
      }
    }//inner for
  }//outer for

  //if no matching named entities found, and it is a WHAT question, seek a definition
  if(start == -1) {
    for(int i = 0; i < tokens.length; i++) {
      if(question.getInterrogativeType()==InterrogativeType.WHAT
              && (tokens[i].getLemma().equals(":") || tokens[i].getLemma().equals("be")))
        start = i < tokens.length - 1 ? i + 1 : -1;
    }
    if(start > -1)
      end = tokens.length - 1;
  }

  //if nothing found, return the default answer
  if(start == -1)
    return answer;

  //otherwise, extract the named entity tokens and indicate a high quality answer
  StringBuffer sb = new StringBuffer();
  for(int i = start; i < end; i++)
    sb.append(tokens[i].getText() + " ");
  return answer.setAnswerText(sb.toString()).setAnswerQuality(Answer.QUALITY_VERY_HIGH);
}//extractAnswer()
```

Running the QA System

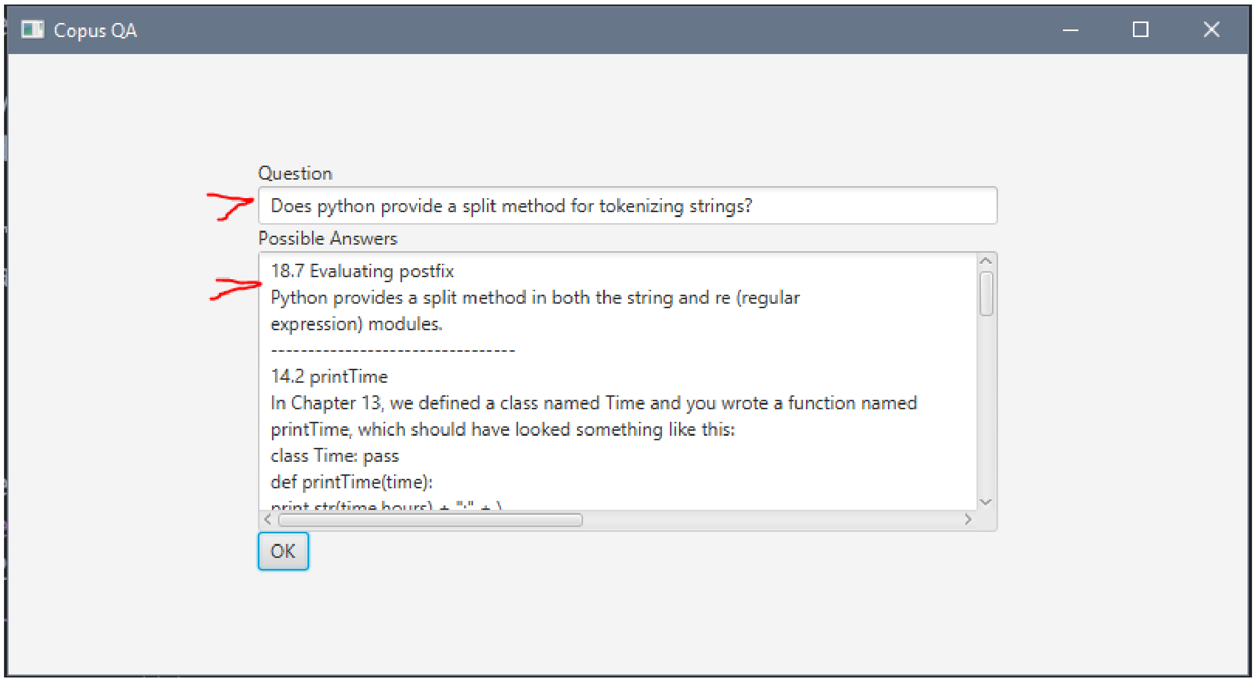
The system as delivered is designed to be run in the IntelliJ IDEA IDE with the Scala Build Tool (SBT) and Scala language features installed. There are five run configurations, though three are legacy, and only two are currently used: “`TF-IDF Document Processor`” and “`fxQA`”.

The `TF-IDF Document Processor` run configuration performs all document processing (not just TF/IDF as the name suggests). It has its entry point in `TfIdfDocumentProcessor.scala` as discussed above in the section called Document Processing. The documents should be prepared .txt files in the “corpus” directory. When complete, the “preparedCorpus” directory will contain Spark object files for the document objects, sentence object, Word2Vec, and LDA.

The `fxQA` run configuration is the simple Question-Answering interface implemented in JavaFX. Its entry point is in `fxQA.scala`. The user types a question, clicks the button, and receives the system’s five best answers within the interface.

Results and Conclusion

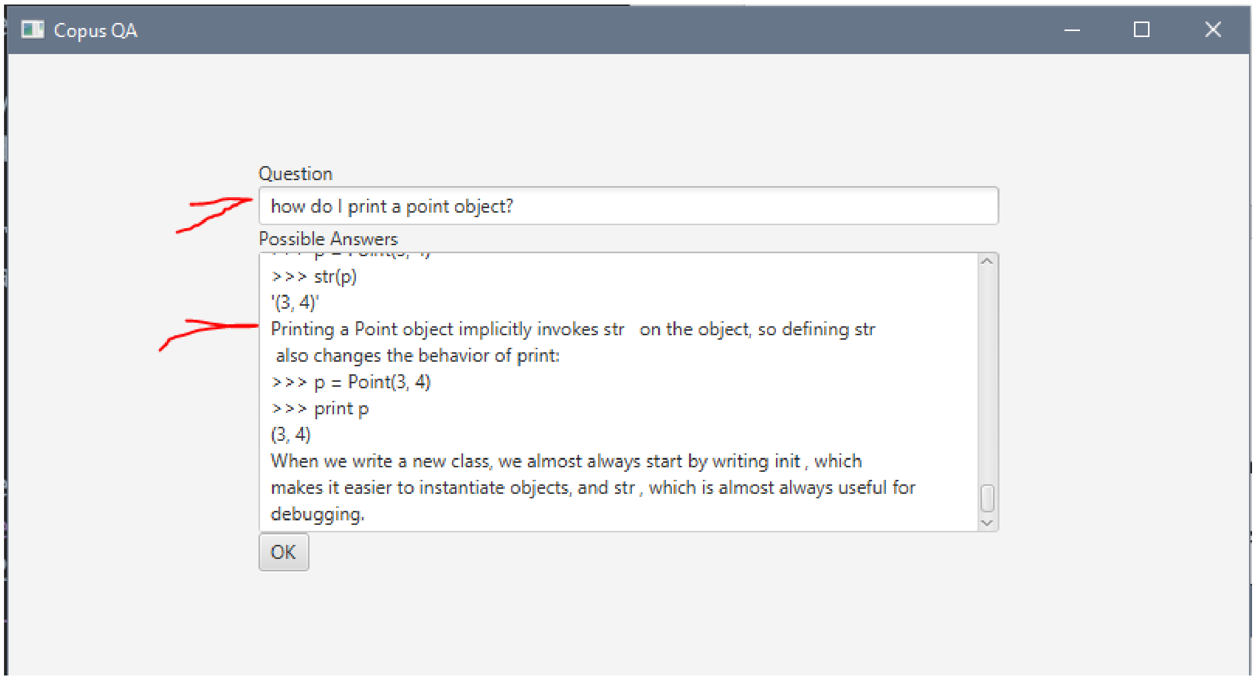
Answer extraction is somewhat more difficult for this type of corpus. In particular, many of the techniques that are applicable to broad knowledge are inadequate, e.g., knowledge graph or ontology-based mechanisms require extensive domain knowledge, which is best captured by a very manual and painstaking process; and other extraction cues such as named entity recognition require a corpus that makes some use of named entities—not a computer programming textbook. These factors weigh heavily on this system’s ability to extract quality answers in the given domain. The following examples demonstrate both the abilities and limitations of the system.



Does python provide a split method for tokenizing strings?

```
18.7 Evaluating postfix
Python provides a split method in both the string and re (regular
expression) modules.
--------------------------------
14.2 printTime
In Chapter 13, we defined a class named Time and you wrote a function named
printTime, which should have looked something like this:
class Time: pass
def printTime(time):
```

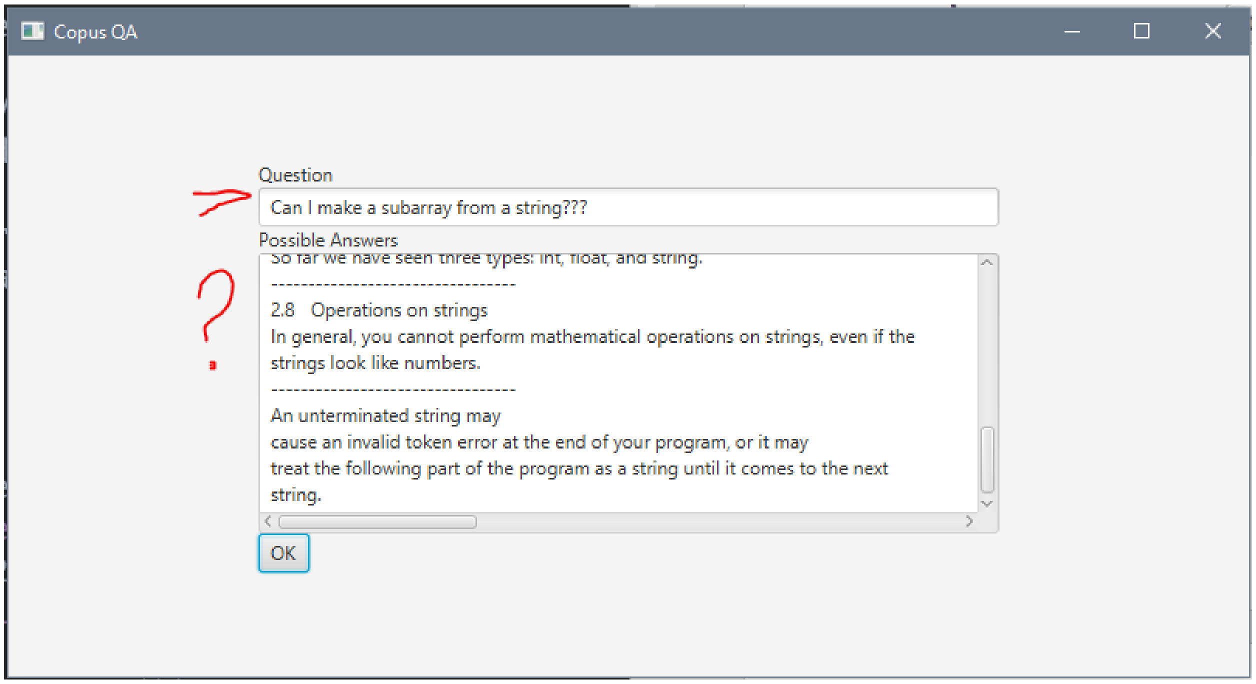
(Figure, above) When the QA system works, it provides very helpful responses. The answer to the question is actually, “Yes,” which the user must synthesize from the system’s best guess answer—but it’s decent.



how do I print a point object?

```
>>> str(p)
'(3, 4)'
Printing a Point object implicitly invokes str   on the object, so defining str
 also changes the behavior of print:
>>> p = Point(3, 4)
>>> print p
(3, 4)
When we write a new class, we almost always start by writing init , which
makes it easier to instantiate objects, and str , which is almost always useful for
debugging.
```

(Figure, above) Again, the answer to the question—"Just invoke print on the object"—is implicit in the answer, not explicit; but the results are useful in any case.



Can I make a subarray from a string???

```
So far we have seen three types: int, float, and string.
--------------------------------
2.8   Operations on strings
In general, you cannot perform mathematical operations on strings, even if the
strings look like numbers.
--------------------------------
An unterminated string may
cause an invalid token error at the end of your program, or it may
treat the following part of the program as a string until it comes to the next
string.
```

(Figure, above) In this case, we are looking for an answer that captures the concept of performing string "slice" operations, because that's what string subarrays are called in Python. However, the Python-only corpus does not provide sufficient evidence for Word2Vec to associate "subarray" with "slice," so our statistical methods fail due to insufficient corpus volume.

Anecdotally, the 2-Gram implementation did seem to improve results in cases of compound words and concepts, as expected. Word2Vec and LDA did not prove as effective as expected in this application. I believe this may be a function of the corpus size more than anything else. The corpus consists of a lengthy text divided into twenty distinct documents, but I believe it may require perhaps as much as two orders of magnitude more content to provide enough fodder for Word2Vec and LDA to have a reasonable training set. Corpus development at this scale would be a very large exercise.

FURTHER WORK

In my original paper for the course, I wrote,

> *Knowledge Discovery of unstructured text could be effectively applied to the computer program learning systems used by introductory computer programming students. The addition of this technology to many of the aforementioned Intelligent Tutoring Systems brings additional advantages such as allowing student assistance with questions outside of the context of writing code, the ability to provide a vast amount of content or knowledge, since many textbooks have been written on any chosen programming language, returning answers that seek to educate rather than provide a solution to a coding problem.*
>
> *A future direction for this project could be to use the QA system for a new augmentation to an Intelligent Tutoring System. This system would have knowledge of common programming errors in addition to textbook knowledge. The system would also track errors not previously unrecognized so as to add to the corpus.*

I still believe this is correct. The QA system shows promise and might provide significant value in the context of an Intelligent Tutoring System (ITS) for beginning computer science students. To be useful in this context, however, the system would require substantial, additional development: first, in corpus size and breadth, to allow the statistical methods to function well; and second, in ontology development, to provide adequate ontological models to allow the system to reason about student intent, and thus enhance answer extraction. These are large and fertile fields of future research.