A Trivial Pursuit Player

A Computer Program Using Natural Language Processing

Michael Chen, Christopher Lee, and Alex McLeese
Computer Science 182
December 12, 2011
Professor Radhika Nagpal

Abstract—**We implemented a simple but effective program that answers trivia questions. We discovered that a combination of Google search; careful selection of keywords; and sensitive scoring, weighting, and decision functions make correctly answering over two-thirds of Trivial Pursuit multiple-choice questions possible (77 percent correct out of 90 questions). These straightforward methods for search and direct processing of text, though they cannot answer some classes of questions, have proven more effective for our purposes than more complicated tools like SPARQL and Wordnet. We first use Google search to generate blocks of text relevant to a particular question. At the same time, we use the capabilities of the NLTK toolkit for Python, including measuring frequencies of words in million-word corpora and tagging words with parts of speech, to effectively generate keywords and score answers. It turns out that these natural language tools, combined with Google, can go a long way in successfully answering trivia questions.**

I. Introduction

Impressed by the spectacle of IBM's Watson winning Jeopardy, we decided to implement a program to answer trivia questions. We wanted to learn more about the technology stacks needed to implement this type of AI, namely natural language processing, semantic search, statistical artificial intelligence, decision theory, and engineering strategies. We were also interested in experimenting with various strategies to organizing information and answering questions. Previous work in question answering has made use of human-generated keywords (Microsoft AskMSR), the SPARQL query language for accessing the semantic web (Watson), and the Wordnet resource for synonyms (Watson). Watson demonstrated how powerful processing raw natural language could be. We set out to test the relative strengths of fully automated programs, of the semantic web, and of natural language processing. We found, with the help of Google and the NLTK toolkit, that the semantic web was too frustrating to be of use, and instead that fully automated natural language processing could be fast, simple, and accurate.

II. Model

One of the advantages of creating a program to play a game is that the problem is clearly defined. We set out to answer history trivia questions, and finally settled on answering questions from Trivial Pursuit for children since they have defined multiple choice answer options. These questions come in six categories, with three multiple choice answers each. We decided to move beyond history to include all topics, and have attempted to answer as many of the questions correctly as possible. We enter the questions into our program by filling in a "question" variable and three "answer" variables. We chose a set of 100 questions to test our program on, 90 of which returned answers without any strange parsing and exceptions errors, and then broken down into 3 subsets of 18, 18, and 54 (referred in this paper as subset_0, subset_1, subset_2) which we trained, tested, and validated upon.

III. Algorithms

A. Generating Keywords

Given a question and three possible answers, the first step in answering the question is to find out more about the words given, and to generate keywords. We used information about frequencies of words and parts of speech to select keywords. First, we used an NTLK tool to filter out common and irrelevant words. Then, to determine the frequency of each remaining word, we searched through two corpora built into NLTK, the million-word Brown and Reuters collections. We wanted to choose the distinctive words that captured the essence of the question and answers, so we favored uncommon words. As long as words tested were relatively infrequent, below several hundred words in the two NLTK corpora, they remained candidates for becoming keywords. In addition to determining frequencies of words, we used a part-of-speech tagger built into NLTK to filter out words unlikely to be of much use, and to focus attention on important adjectives, verbs, and nouns. Given this information by the NLTK tools, we chose keywords from the question and answers that were least frequent (through searching the corpora) and most important to the meanings of the phrases (through the POS tagger). For example, given the question, "Which king was overthrown during the French Revolution?", our approach returns 'king': (25, 'NN'), 'Revolution': (28, 'NNP'), 'French': (487, 'JJ'), and 'overthrown': (5, 'VBN'). In this case, "overthrown," as an uncommon verb, would be weighted heavily.

B. Scoring

Armed with keywords for the question and answers, we then moved on to the raw text returned by Google. To find relevant text, we entered the exact text of the question into Google, and focused on the first several dozen URLs the search returned (usually the first three pages of results). We used an NLTK tool to strip HTML and concentrate only on the words in these top-ranked web pages. We then used 4 scoring methods, which are referred throughout the paper:

Scoring_1). We simply counted the number of times the full answer phrases appeared in the combined text.

Scoring_2). We counted the occurrences of keywords generated from the answers, combining the scores of multi-word choices.

Scoring_3). We began to make use smart use of the keywords generated from the question. We assigned a score to each occurrence of an answer phrase or an answer keyword. The specific numbers in our scores have been drawn from several rounds of empirical testing. We discovered that we should heavily weight full answer phrases and groups of nearby keywords to better capture the precise meanings of questions and answers. For the full phrase, we assigned a base score of 10; for keywords, we assigned a base score of 1. Many occurrences of single keywords proved irrelevant to the question, so we rewarded full phrases heavily. For every question keyword within 50 words, we doubled the score. This exponential form for nearby keywords captured effectively the precise meaning of a question, as represented by multiple words together. Finally, for each candidate answer we added up the scores of our occurrences of its related phrases and keywords. This scoring strategy of rewarding full phrases and clusters of keywords has proven highly effective in practice.

Scoring_4). For our most sophisticated scoring method, we made use of more parameters, feeding them all into a general scoring function. This approach is still evolving. Our parameters were the following: whether the answer was a full phrase or a keyword fragment, the part of speech of the relevant answer keyword (if it was not a phrase), the frequency of the relevant answer keyword (if it was not a phrase), the number of nearby question keywords, the distances to those question keywords, the parts of speech of those question keywords, and the frequencies of those question keywords. For infrequent answer and question keywords tagged with important parts of speech, the score rose. In addition, the score increased with the number of nearby question keywords. Given these relevant parameters, we experimented with many functional forms and constant weights, seeking to answer more difficult questions. What follows is our current draft of these formulas. To calculate the weight of a keyword from the question or the answer, we use this formula: *200/(frequency + 1),* where the frequency is the number of times the word appears in the Brown and Reuters corpora. This calculation grants higher scores to uncommon and distinctive words. To calculate the weight of a question keyword near an answer keyword, we use this formula: *50/(distance + 1)*. This calculation grants higher scores to keywords clustered near one another. To emphasize the importance of verbs and of full phrases, we multiply by constants if the answer in question is a full phrase (10), and if any of the keywords involved are verbs. Although we have not perfected the numerical weights in this fourth approach, as we have the third, we know that its general form, taking into account many factors, represents the best path forward.

Remark The descriptions of `scoring_3` and `scoring_4` above are the culmination of all improvements made after running `TrivialPursuitQA` on training set `subset_0` and testing set `subset_1`.

An example may help to clarify our scoring methods. Consider this question, with the correct answer in bold:

Q1: "What was illegal for kids to do on a Sunday in a colonial New England village?"
A: "walk to church", **"kiss their parents"**, "eat lunch,"

When scoring by counting the number of times answer keywords appear, we get these results:

Raw: {'kiss their parents': 35, 'eat lunch': 34, 'walk to church': 106}

The raw scores are incorrect by a large margin. When we make use of question keywords, the situation improves. With base scores of 10 for full answer phrases and 1 for fragments, plus multiplication times two for every nearby question keyword within 50 words, we get these results:

Keyword: {'kiss their parents': 3541, 'eat lunch': 892, 'walk to church': 3866}

Unfortunately, this approach still gives the wrong answer. However, we can do better if we weight answer phrases near keywords even more heavily. With a base score of 100 for full answer phrases and 1 for fragments, with multiplication times 10 and 5 respectively for nearby question keywords, we get very large, but more accurate, numbers:

Weights: {'kiss their parents': 11516763, 'eat lunch': 2065038, 'walk to church': 8025882}.

Finally, this scoring mechanism, with especially heavy weights for answer phrases near question keywords, has returned the correct answer.

To improve our scoring even more, we can move beyond numerical weights to functions that relate parameters in more complex ways, as discussed above. For example, instead of cutting off a range of nearby question keywords at 50 words, we could weight according to a linear or polynomial function decreasing in the distance between answer keyword and question keyword. Instead of assigning a standard base score of 1 or 10, we could create a more nuanced weight based on frequencies and parts of speech. These changes have been implemented in scoring algorithm number 4.

We have implemented these more sensitive scoring algorithms, and they return similar results to the approaches with fixed weights, though they are sometimes more accurate. In analyzing this single question, we have seen how various scoring mechanisms work, from simple counting to weighting question keywords.

IV. IMPLEMENTATION

Remark: Code and Readme can be accessed from Github repository < http://goo.gl/J0lwx>.

C. NLTK

We implemented our program with several hundred lines of Python code, using a scraper for Google results and many functions from the NLTK toolkit. Our scraper pretended to be a Firefox browser to retrieve several pages of URLs from Google results. Next, we used our NLTK tools to search two million-word corpora and tag the question with parts of speech. We computed frequencies and parts of speech for all question and answer keywords. Then we used NLTK to filter out common words and HTML and return a pristine chunk of text for searching.

After this preparation, armed with our question and answer keywords, we created a dictionary of occurrences of question keywords near every occurrence of an answer keyword in our chunk of text. We passed relevant information to a scoring function, then added up the scores of all occurrences of answer phrases or keywords.

D. Decision Theory

After various scoring strategies are used, the next AI challenge is to add meaning to these scores when producing a single answer. Determining the final answer is analogous to orchestrating a dizzying array of numerical scores and weights. The implementation of `trivialpursuitQA` used multiple rounds of normalizations, edge case scenarios, and basic mathematical manipulations.

In `determineAnswer()`, each score is individually normalized by dividing all scores by the min score. To determine the confidence, we then calculate the factor by which the 1st most confident answer (highest score) over the 2nd most confident answer (middle score). For example, in `score_result1 = {'storm coming': 1.0, 'cloudy': 1.5, 'fair': 5.0}`, 'fair' is 5x more confident than 'cloudy'. Therefore it has a raw confidence level `conf=5` that can be multiplied later by an optimized delta or multiple variable. The same computation is done over all scoring strategies.

This straightforward approach quickly increases in complexity when we actually observe our results from `determineAnswer()` on multiple queries. Each `score_result` dictionary result can be of `length = 1, 2, 3`, with up to 2 or 3 matching when applicable (e.g. `'storm coming': 1.5 || 0.0, 'cloudy': 1.5|| 0.0`), and normalized value equals non-zero, positive integers or zero. If we calculated all total outcomes correctly, given 4 scoring strategies, we create a total of 43 edge case scenarios that have to be treated separately. Human-guided intuition and decisions (albeit imperfect and very prone to error) handled each case separately, categorized with a specific weight/damping factor and normalized for the final answer generation with aggregate confidence.

E. Bayes Rule and Probabilities

Though we did not cover Bayes Rule or Law of Total Probability in this course, Joe Blitzstein's STAT 110 provides ample exposure that covers more the material presented in AIMA as well as Sebastian Thrun and Peter Norvig's 2011 Stanford online AI course. In an attempt to create a reliable confidence score, we decided to utilize conditional probability. We calculate a total of 43 outcomes by:

```
P(success) =
  [[P(success|no zeros) ]+
  [ P(success | 1 zero) +
     P(success | 2 or 1 zeros, when applicable) +
     P(success | 3, 2 or 1 zeros, when applicable)]+
  [ P(success | 1 value) +
     P(success | 2 or 1 identical values, '') +
     P(success | 3, 2 or 1 identical values, '')]
]
```

The total outcomes are illustrated in the table below, which may help the reader better understand why the case of all different values is different than the case of multiple zeros is different than the case of multiple identical values, where score result can be `length = 1, 2, 3`. The underlined values are true by symmetry.

Length	3	2	1	0	Total
No identical	3!	2!	-	0!	9
3 zeros	1	-	-	-	1
2 zeros	**2*3**	1	-	-	7
1 zero	**3*2**	**2*1**	1	-	9
1 identical	**3*2**	**1*2**	1	-	9
2 identical	**2*3**	1	-	-	7
3 identical	1	-	-	-	1
Total	32	8	2	1	**43**

Table 1 - Interestingly, we see begin to see a mathematical pattern emerge of `2^fib`, where `fib={0,1,3,5...}`, the Fibonacci sequence as `n={0,1,2,3...}`. We suspect that this occurrence is true by some underlying by mathematical principle that we, frankly speaking, just don't understand. If we add a 5th set `n=4`, we expect to observe `299 = 256+43` individual edge cases to handle. Our conjecture may be entirely off the mark, and there may be a more elegant approach to this problem.

While we (hopefully) calculated the number of edge cases to handle, we face further theoretical ignorance of this uncertainty calculation. While we would like to map the discrete random variable confidence to the event probability of success, we found that the simpler approach seemed more reliable (mostly because conditional probabilities were a headache to implement). The said simpler approach is merely a crude mapping approximation that yielded the results presented below. The ideal situation is to understand the entire scoring mechanism as a probability density function that returns a confidence score with high correlation to the a probability score. Though this section concludes our efforts in Bayesian probabilities, it is an interesting problem worth revisiting.

We attempted to treat each 43-edge scenario, with a (crude) function relation. Each numerical value of the confidence score `nScore` is analogous to a payoff matrix in game theory. The are similar in the sense that 1) if you are confident and correct, the `tpScoreWeight` will return a hot score (strong), 2) if you are confident but incorrect, the `tpScoreWeight` will return a cold score (hurtful), 3) if you are `not_confident` and correct, the `tpScoreWeight` will return a lukewarm score, and 4) if you are `not_confident` and incorrect, the `tpScoreWeight` will return a chilly score.

F. Importance of Scoring and Weighting

Numerically, a neutral score `nScore = 50` means that our `trivialpursuitQA` completely failed and could not generate an answer. The raw confidence level `conf=0` and is added/subtracted to the random guess depending if it is correct or incorrect (50 +/- 0). The range of the scores are normalized and bounded below by 0 and above by 99. This means that if we added a stimulant weight that is meant to aggressively mark more correct answers (e.g. `conf=32572`) even after normalizing, we cap it at 99 per cent confident. The lower bound of 0 per cent is to protect the confidence score from becoming negative in the event that we have high confidence, but an incorrect answer. Conversely, if we added a suppressant weight that is meant to subtly mark more uncertainty in our methodology or results, (e.g. `conf=0.73101`), that would not need to be bound since the delta +/- to the neutral score `nScore = 50` is minimal.

Visually, a hot score is steel-blue, a cold score is tomato-red, and a neutral score is grey. In Figure 1, we show how smart scoring and weighting takes a trivial baseline strategy of Google search and dramatically improves its success rate. We opted for python-based matplotlib for quick manipulation and depth of control, despite having poor features in color manipulation and mathematical tooks as compared to other toolkits protovis.js, processing and matlab. We mapped RGB 0-255, to matplotlib-interpretable MxNx3 arrays. The values from steel-blue to tomato-red were calculated to selected for a one-to-one mapping to the 7 quantiles over the distribution of the adjusted confidence scores 0 to 99. Each box represents a single query score, each column represents the entire scoring values, and each row represents how each scoring function did.

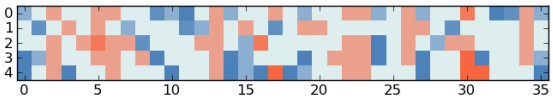

Figure 1 Example results of initial 36 randomly-selected trivial pursuit questions, against 4 scoring schemes, and index 0 being the culminating `finalConfidenceScore`. Types of questions were not uniformly distributed into 6 categories, but chosen as random as would happen in game play. Dark blue means an accurate answer with high confidence positive reinforcement. Dark red means an incorrect with high confidence negative reinforcement. Lighter shades hold the same principle but with lower confidence. The vast number of neutral blue-tint events demonstrates how smart scoring and weighting tipped the scale to favor success.

Remark: The improvements made for weighting, scoring, and decision functions were optimized over subset_0 and subset_1. However, testing on subset_2 yielded success percentages within the error margin (3 percentage points) as the subset_0 and subset_1, so here we present the total results of 90 questions.

From our preliminary results of our benchmark run in Figure 1, we can note that in our initial run Score4 strategy was behaved aggressively, marking high confidence scores. In queries 3, 16, and 19, we observe how the strategy led us toward the right answer. If the weight was increased, it could help in situations such as query 5, 26, and 24, where it was the only scoring strategy that yielded the right answer. However, by doing so, we now encounter an updated challenge: over-adjustment can exacerbate already incorrect queries 17, 31 and others. The same can be said of the remaining scoring strategies.

Thus, we arrive at the complexity of improving our scores: by adjusting one weight, the equilibrium of a system that we do not understand is shifted in a unknown. Tuning this `trivialpursuitQA` to perfection would necessitate a much more rigorous analysis of a much larger data set and proper statistical analysis.

In this methodology section, we did an overview of the normalization, delta, multiples, permutations, distributions and decay strategies used to produce a better `trivialpursuitQA` system. We favored surveying a wide array of approaches over focusing on rigor and in-depth investigation. The ability to experiment beyond AI textbook strategies known to us was most rewarding. We look forward to further development of this field of computer science.

G. What We Did Not Implement

What we have learned through failures has been nearly as substantial as what we have learned through successes. At the outset of our project, we intended to make great use of SPARQL to search the semantic web, and of Wordnet to generate synonyms. However, we discovered that both tools have limitations. Converting a natural language question into a near-logical form for SPARQL is nearly impossible to automate. In addition, the knowledge bases of semantic web resources are often incomplete in strange ways. They often lack information crucial for answering trivia questions. Wordnet has not been much help, either. This resource is built into NLTK, and reliably produces lists of synonyms. However, it produces large numbers of these synonyms, without grouping them into meaningful categories. When we tried using all of the synonyms as keywords, we found that the presence of so many words confused and diluted our scoring mechanisms. Finally, we tried to implement key phrases as well as keywords, but found that doing so is costly, with only occasional benefits. Identifying key phrases requires running all possible combinations of consecutive words in questions and answers through the two NLTK corpora. In addition, key phrases are only rarely more helpful than keywords. When a key phrase occurs repeatedly, its parts occur near one another,

racking up large scores even without the help of separate key phrases.

V. Results

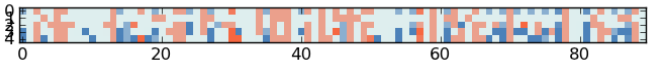

Figure 2 - Note that `subset_0, subset_1` and `subset_2` correspond to size 18, 18 and 54, where we trained, tested and validated. The color mapping is calculated over a larger question set, therefore it differs from Figure 1. Note that in question range 37-90, our scoring, weighting and decision improvements yielded stronger overall confidence scores.

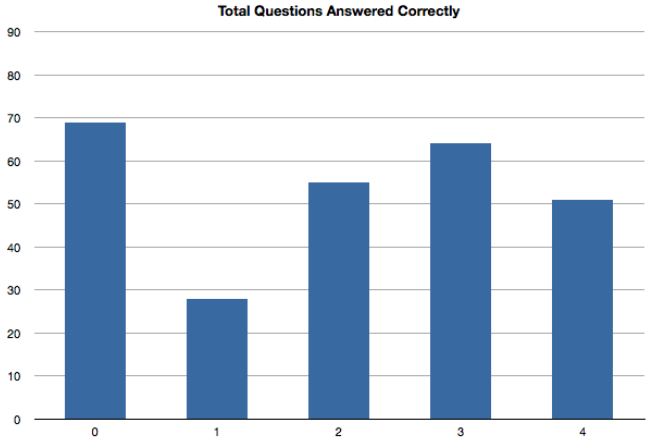

Figure 3 - Comparison of how `score_0` (cumulative score) compares to its individual `score_(1-4)` parts over the entire 90 question dataset. Score_1 performs worst, however, our weights and decisions made in the cumulative `score_0` is able to take account of this during the 43 edge case scenarios, and properly compute a valid overall score. Including `score_1, score_2`, and `score_4` are important in the scenarios where `score_3` fails.

We tested our program in three stages. First, we used a group of 18 questions as a training set to explore various strategies. We answered only 11 correctly, for a score of 61 percent, but we used this experience to make improvements to our code. Out of a second group of 36 randomly chosen Trivial Pursuit questions, our testing set, we answered 72 percent, or 26 questions, correctly. Out of a larger group of 90 randomly chosen Trivial Pursuit questions, our validation set, we answered 77 percent, or 69 questions, correctly, using an aggregate of our various algorithms. Our first brute force method of counting full answer phrases answered only 28 correctly. Our second brute force method of counting answer keywords answered 55 correctly. Our third method, using fixed weights to reward answer occurrences near question keywords, did the best, answering 64 questions correctly. Our fourth method, the most sophisticated in theory, answered 51 questions correctly. This last function is more complicated, and so needs more fine-tuning. However, the excellent performance of the aggregate (69), and in particular of the function that weights answer phrases near question keywords (64), shows that our approach is working very well. When we set out to make a question answering system, we aimed at a success rate over 60 percent, and we have surpassed that goal by a significant margin.

VI. Discussion

A. Performance of Brute Force Methods

First, we discovered the strengths and limitations of brute force approaches. Initially, we tried feeding Google the text of the question plus the text of each possible answer. We counted the number of hits generated, and chose the answer with the largest number. This approach works surprisingly well for many questions, such as,

Q2: "When was the Treaty of Versailles signed?"
A: "1919," whose meaning rests on distinctive words (like dates).

However, this approach runs into trouble when certain words or phrases are very popular. For example, where we paired the respective answers with the question in the query, we observed the following hits:

Q3: "Who was the king overthrown during the French Revolution?"
A3a: "Louis XVI" = 165,000 hits.
A3b: "Marie Antoinette" = 773,000 hits.
A3c: "Barack Obama" = 1,460,000 hits.

Clearly we need to do better.

The next brute force methods we tried involved not entering answer choices directly into Google, but instead counting the number of times answer phrases and keywords appeared in the text of the pages returned by Google when the search engine was fed the text of the question. These strategies performed better than the first method, and the strategy based on keywords answered 61 percent of questions correctly, 55 out of 90. In the case of the French Revolution question, counting answer occurrences gave the correct answer of "Louis XVI." However, these strategies, too, have weaknesses. For example, consider the previously mentioned question, "What was illegal for kids to do on a Sunday in a colonial New England village?" If one scores by counting occurrences of answer keywords, the program returns "walk to church." It makes sense that the word "church" would appear often in results about a Sunday, but this answer is wrong. A better approach is needed, one that pays more attention to the wording not only of possible answers, but of the question itself.

B. Performance of More Sophisticated Methods

To improve our results, we generated keywords from questions and implemented more complicated scoring systems. Though these approaches did not solve all the problems with prior systems, they did answer a larger number of questions correctly.

1. Easy Questions

Questions whose meaning can be captured by just a few keywords are easy for our program to answer correctly. For example,

Q4: "Where's your funny bone located?"
A: "near your elbow", "on your wrist", "just below your shoulder,"

depends completely on "funny," "bone," and, in the correct answer, "elbow." In addition,

Q5: "How long does it take an albatross egg to hatch?"
A: "1 week", "1 month", "80 days,"

depends completely on "albatross," "egg," "hatch," and, in the correct answer, "80" and "days." Finally,

Q6: "What is each member of a winning Super Bowl team given?"
A: "a bronzed jersey", "a green jacket", "a ring,"

depends completely on "Super," "Bowl," and, in the correct answer, "ring." Keywords represent the meanings of these questions and phrases very well, so our method succeeds in these cases.

2. Hard Questions

However, the English language is complicated, and keywords do not always tell the whole story. Some classes of questions pose particular problems, especially negations and comparisons. For example, the question,

Q7: "Which of these sea creatures is not a mammal"
A: "a sea cow", "a sea lion", "a sea horse,"

depends completely on the word "not," which is not captured by any keyword. In addition, the question,

Q8: "What does xylophone music sound the most like", with
A: "bells", "drums", "rattles,"

depends completely on "most" and "like," which are again not captured by keywords. Questions in these two categories occur repeatedly in Trivial Pursuit, and drag down our scores. To answer them correctly in future versions of our program, we would have to analyze questions more carefully.

Other kinds of hard questions include ones that depend not on keywords but on key phrases, and ones that contain wrong answers that are very common in textual samples. An example of the first category is

Q9: "Which face on Mount Rushmore was actually blown up and sculpted again in a better location?"
A: "Washington", "Jefferson", "Lincoln." Our program answers "Lincoln," but the correct answer is "Jefferson."

The problem is that the essence of the question is captured not by the individual words "blown" and "up," but by the combined phrase "blown up." When one runs the question with a key phrase, "blown up," the program answers "Jefferson" correctly. However, identifying helpful key phrases is very difficult, depending on careful analysis of word and phrase frequencies. Key phrases usually do not matter, and their computation slows the program down considerably. But in situations like this one they are helpful. An example of the second category is the

Q10: "What material makes up the most kind of trash in US landfills?"
A: "paper", "plastic", "metal."

In this case, the correct answer is "paper," but the word "plastic" appears so often in text relating to landfills that our program mistakenly chooses it. Here, the difficulty is that much of the text returned by Google is not precisely related to the questions, so that popular words or phrases can overwhelm the correct answer.

3. The Power of Keywords

Sometimes, our use of question keywords and sophisticated scoring methods improved the performance of our program significantly. In fact, our third algorithm using question keywords outperformed our brute force counting methods by a score of 64 to 28 (brute force full phrases) and 55 (brute force keywords). These tools helped us overcome popular phrases to focus on phrases directly related to the meaning of the question. For example, when answering the question

Q11: "How do you make bagels shiny?"
A: "boil them before baking", "buff them", "add vegetable oil to the dough,"

our less sophisticated methods choose "add vegetable oil to the dough" because "vegetable," "oil," and "dough" are all common words in text relating to making food. However, our use of keywords "bagels" and "shiny", combined with scoring rewarding answer occurrences near question keywords, enabled the program to correctly choose "boil them before baking." Evidently, "boil" and "baking" appeared especially often near the keywords "bagels" and "shiny," overcoming the other answers. Here is how the normalized scores changed between algorithm two (brute force keywords) and algorithm three (weights for keywords):

Algorithm 2: {'add vegetable oil to the dough': 1.9803921568627452, 'boil them before baking': 1.0}
Algorithm 3: {'add vegetable oil to the dough': 1.0, 'boil them before baking': 1.2460233297985153}

Similarly, when answering the question

Q12: "What goes clink, clink, clink in a song about the wheels of a bus going round and round?"
A: "the brakes", "the windshield wipers", "the money"

Using brute force methods the program chose "the windshield wipers," which are obviously related to the topic of buses. However, placing greater emphasis on question keywords like "song" and "clink" and "round" enabled the program to correctly choose "the money," which appear often in particular blocks of text related to the famous song. The normalized results improved as follows:

Algorithm 2: {'the money': 1.0, 'the windshield wipers': 1.2727272727272727}
Algorithm 3: {'the money': 111.25427323168879, 'the windshield wipers': 1.0}

A similar success story involves the question about colonial New England mentioned above. The word "church" occurs often in pages related to Sundays. But our program successfully chose "kiss their parents" as the answer when we adjusted the weights to place greater emphasis on question keywords like "illegal" and "kids," making it possible for the correct choice to overcome the more common phrase.

VII. Conclusion

We learned much from our project. First, we experienced the power of Google search. We had anticipated needing to use SPARQL to query the semantic web, and Wordnet to find synonyms, but we found that those tools could be distracting. Converting a natural language query into a SPARQL query can be extremely difficult, and semantic web resources do not always contain relevant information. Wordnet is easier to use, but returns a larger set of synonyms than is useful, and that leads to diffuse scoring of results. A simple Google search, by contrast, does a wonderful job of selecting relevant text for scoring and analysis.

Second, we experienced the power of natural language processing tools. We initially thought that these tools would be too complicated, and too slow in practice, to rely on much. But the NLTK for Python toolkit made determining word frequencies and parts of speech easy. In addition, these functions did not run as slowly as we feared. We found that it is possible to search two million words from corpora, plus several dozen web pages, in about a minute. This speed made accurate scoring possible.

Third, we learned the strengths and limitations of our keyword-based approach. When the meaning of a question or an answer hinged on a few distinctive words, like a proper noun or an unusual verb, our strategy worked well. But in other cases, when meaning depended on one or two common words (like "not" or "most"), or on human reasoning that transcended language, our approach failed. We did find that adding more complicated weights to our scoring function and giving higher scores to answer phrases near question keywords improved our performance. It turns out that nuanced and precise scoring really does pay off. Full phrases are far more informative than single keywords, and groups of keywords are far more informative than lone keywords. Our scoring methods correctly reward phrases and clusters of words. We achieved greater success than we had anticipated, answering 69 questions correctly out of a test set of 90. This 77 percent success rate is higher than the level around 60 percent that we had anticipated. To make our algorithm even better, we would need to do a better job of analyzing what a question is asking for (a person? a date? a place?), and also of capturing the meaning of not just individual words but complicated phrases, including negations and figures of speech.

Fourth, we learned the bounds of what we could achieve in a few weeks. We were not able to make an exhaustive catalog of general forms of questions to better capture their meaning, and we were not able to exhaustively search all possible forms of scoring algorithms to determine the optimal one. Given a few more weeks, we would improve our knowledge of question formats like comparisons, make greater use of smaller sets of synonyms from Wordnet, and run local search algorithms on our scoring weights.

More broadly, in the future we would like to explore question answering beyond multiple-choice trivia games. We want to be able to answer complex questions without the aid of provided choices. To make progress toward that goal, already achieved in part by the Watson team, we would need to make greater use of statistical natural language processing. By using more complicated mathematical methods, we could better capture the meanings of groups of words and filter out types of common phrases that are not possible answers. Going forward, we hope to continue to tinker with our program to enable it to answer more difficult questions.

Acknowledgments

We thank our teachers Professor Radhika Nagpal and Charles Herrmann.

References

Allemang, Dean and Jim Hendler. *Semantic Web for the Working Ontologist*. 2nd ed. New York: Elsevier, 2011.

Bird, Steven, Ewan Klein and Edward Loper. *Natural Language Processing with Python.* Sebastopol, Calif.: O'Reilly, 2009.

Brill, Eric, Susan Dumais and Michele Banko. "An Analysis of the AskMSR Question-Answering System," in *Proceedings of 2002 Conference on Empirical Methods in Natural Language Processing*.

DuCharme, Bob. *Learning SPARQL*. Sebastopol, Calif.: O'Reilly, 2011.

Ferrucci, David, et al. "Building Watson: An Overview of the DeepQA Project." *AI Magazine* (Fall 2010).

Gentile, Sal. "From Watson to Siri: As Machines Replace Humans, Are They Creating Inequality Too?" *PBS - The Daily Need*. Public Broadcasting Service, 25 Oct. 2011. Web. <http://goo.gl/j9b8Q>.

Kwok, C., O. Etzioni and D. Weld. "Scaling question answering to the Web," in *Proceedings of WWW'10* (2001).

Russell, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach.* New York: Prentice Hall, 2010.

Strzalkowski, Tomek and Sanda Harabagiu, eds. *Advances in Open Domain Question Answering*. New York: Springer, 2006.

APPENDIX

A. Engineering a TrivialPursuitQA System

The IBM Watson-Jeopardy! event demonstrated how smart AI algorithms are just as important as smart engineering techniques. The IBM DeepQA computation is similar to our own TrivialPursuitQA computation in that both are 1) embarrassingly parallel, 2) have the ability to cache web pages and documents, 3) and is able to compute within a second or less. However, the scale and complexity of our problem is about 2-4 orders of magnitude smaller. We utilized a 2011 Intel i5 Core quad-core processor with 8GB ram, multi-threading, 4 scoring techniques, open-source NLTK toolkit, pre-fetched and pre-processed URLs relevant to keywords and queries stored in cache. IBM utilized 2880 compute cores on a massively parallel POWER7 architecture with 16TB, multi-processing, more than 100 scoring techniques, proprietary natural processing, and pre-processed answer and evidence sources.

	TrivialPursuitQA	**DeepQA**
Knowledge Type	Web knowledge	Shallow and deep knowledge
Question Type	Pre-defined trivia questions	Amorphous jeopardy questions
AI Generation	Fact-based generation	Interpretation and hypothesis generation
Answer Output	Multiple choice type	Lexical Answer Types (LAT)
Ancestry	AskMsr, Google	TREQ, OpenEphrya, DBPedia, et. al.
Compute Power	2.4 GHz Intel Core i5 four-core, 8GB ram	3.5 GHz IBM POWER7 eight-core, 16TB ram
Testing Hours	8 CPU hrs, 120 independent experiments	2000 CPU hrs, 5500 independent experiments
Tuning Data	7.9MB error-analysis data	10GB error-analysis data

Table 2 System architecture comparison between TrivialPursuitQA and DeepQA. We are able to demonstrate that everyday computers are powerful enough for very basic, fact-based QA queries. Just as AI found in pre-installed chess packages found in Windows and Mac operating systems have converged to 1997 IBM Deep Blue chess-playing performance in recent years, we hope that all computing devices in the near future will have intelligent, knowledge and hypothesis generating QA systems.

Each initial TrivialPursuitQA query took ~72 seconds to complete. After the URL corpus was processed and saved, the scoring and weighting functions took less than ~1 second to complete. We benchmarked every step of the algorithm by recording time and saving output. The entire framework was refactored to be entirely automated from batched query search, log files, output files, answer files, re-run files so that we could produce and verify results.

Remark: Amazon was generous in giving us AWS Educational Credit. We tried running our script on a High-CPU Extra Large Instance with 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each, where 1 EC2 Compute Unit 1.2 GHz 2007 Xeon processor). The slowdown in workflow outweighed the minor speedup, mostly due to our relatively small computing problem, data set, limited search, scoring, and weighting strategies. Running the queries locally was more practical and did not require code adjustments.

Remark: IBM decided to use its Power7 architecture over its BlueGene architecture to promote the former enterprise line. Given our experience for this term project, we do not believe such massive architecture is necessary for enterprise and consumer use. A parallel speedup of 2,400x = 7200/3 seconds may have been necessary to give Watson the edge for Jeopardy!, but given our experience with Amazon it is our suspicion that a powerful $2500 desktop today with GPU implementations should perform just as well.

B. README

```
A Trivial Pursuit Player
A Computer Program Using Natural Language Processing

#Answering Trivial Pursuit questions
This project provides code that uses natural language processing to answer trivia questions.

##Summary of included files
  • determine.py: Given results, determines answer and confidence
  • googleResults.py: Returns the top pages returned by Google, given a query
  • importcache.py: Reads in previously cached results
  • output.py: Caches results
  • questions.py: Encoded triplets of sample questions and answers for testing
  • scoring.py: 4 ways to calculate scores, and a method to use them all
  • test.py: Runs provided questions through given scoring function
  • tp.py: Wrapper to handle import and update
  • trivialpursuitfunctions.py: Given a question and answer options, finds keywords and instances of them
  • weights.py: Functions to determine how heavily each kind of keyword should be weighted
```

Remark: Data analysis scripts, plots and all 90+ query results can be accessed at < http://goo.gl/J0lwx>.