Aptitude Question Solver: AptitudeQS

Mohammed Mustafa*, Ronald Anthony D'souza, Deepthi H., Jeevan T. B., and Dr. Pushpalatha K

Department of Computer Science and Engineering, Sahyadri College of Engineering & Management, Mangaluru-575007
*Email: musthafa.m1996@gmail.com

ABSTRACT

An aptitude test is a systematic means of testing a candidate's abilities to perform specific tasks and react to a range of different situations. Quantitative aptitude problems are difficult to analyze and formulate without having a clear understanding. Basic concepts must be known thoroughly to solve the problem efficiently. Many people lack the knowledge of solving the problem using the basic concepts. When solving aptitude tests for companies, candidates should have the ability to solve the problem within a limited time. Hence, we have introduced Aptitude Question Solver (AQS) which provides a step-by-step procedure for each quantitative problem.

***Keywords:** Mathematical word problem, natural language processing, aptitude questions.*

1. INTRODUCTION

With the advent of computers, all aspects of society have been influenced by it, including education. Computers are used at all levels of education. But with recent advancements in technology, even children are learning with computers. Wikipedia, Freebase, YAGO, Microsoft, Satori and Google Knowledge Graph are some of the well-known knowledge bases. Information present in them could be used to build specific decision making advisory systems. Question Answering systems, which are a part of advisory systems are viewed as futuristic replacement of call centers and are called as virtual assistants [9]. An aptitude test is a systematic means of testing a candidate's abilities to perform specific tasks and react to a range of different situations. Quantitative problems are a part of Aptitude tests. It involves a wide range of mathematical problems. A mathematical problem is a problem which can be controlled, analyzed and solved using methods that contain sequence of formulae, theorems, postulates, and axioms. The duration for solving each quantitative problem is limited. Various studies show that students often face problems while solving mathematical word problems like generating variables and forming equations without any basic knowledge and techniques to solve [1]. Sometimes, the correct calculations will result in incorrect answers due to incorrect problem representation [10].

Existing application such as Wolfram Alpha [2] requires input in terms of equations or simple math word problems. It fails to provide solution as the complexity of the mathematical word problems increase. So, if the users are not able the form the equations from the given word problem, he/she will get completely stuck. Other applications are MathWay [3] and WebMath [4] that helps to solve mathematical problems, but it cannot process these problems when given in English language. And it also requires the users to choose what operation to be performed.

Keeping in mind these issues in existing systems we have proposed AptitudeQS for solving these problems. AptitudeQS can be of great use to understand the solution and to know the techniques to solve these problems in a quick and efficient way. The main purpose of our system is to provide stepwise approach to solve any given aptitude question. Our system can be used by any person who intends to learn and solve and can also be used by students or candidates to improve their aptitude solving ability. Users, who intend to use this system, will need to enter the word problem question. The system will interpret the question entered by the user and generate an appropriate solution. The solution will be represented in a step by step manner, which will help the student to understand the solution and concept behind it.

This paper is structured as follows. Section III describes the system architecture and the various components involved in it. Section IV deals with the generation of training dataset and the development of neural network. The process of analyzing the query from user is explained in section V. Section VI illustrates the system performance and its evaluation. Finally, we conclude about AptitudeQS in Section VI.

2. RELATED WORKS

In recent years, technologies such as Wolfram Alpha [2], WebMath [3] and MathWay [4] were developed which can solve verbal mathematical problem only if the question is simple. Wolfram Alpha is able to solve simple verbal mathematical problems but fails to do so when the complexity

of it increases. WebMath places the overhead of extracting numerical data form the verbal statement upon the user. MathWay is an interactive chat bot application which tries to solve mathematical word problems but the problem here is that we must choose the operation to be perform. Therefore, to overcome all these drawbacks, we propose a new system called Aptitude Question Solver. AptitudeQS can solve complex verbal mathematical problems. The main goal of our system is to provide the detailed procedure to solve any given aptitude question.

3. SYSTEM OVERVIEW

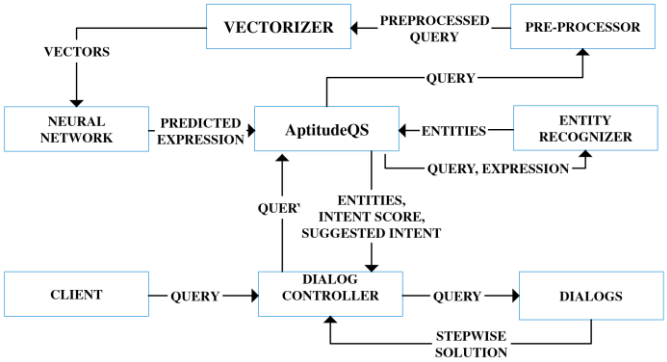

Figure 1: System architecture

The system architecture of the AptitudeQS is shown in figure 1. The components present in this system are dialogs, pre-processor, vectorizer, neural network, entity recognizer and AptitudeQS.

Initially, the system is trained before it can be used to solve the questions. The dialogs component defines a set of expressions and the corresponding methods to find the solution. An expression is a template or a pattern that describes a question, which can also include one or more entities. An entity is a parsed element found within the user's query. Each expression has an intent, which is a method, to solve the question associated with it. Using these expressions, the system generates a training dataset that consists of questions and the name of the expression to which it belongs to. The system uses a neural network to classify the questions to the corresponding expression and uses the generated training set for training. Before feeding questions to the neural network, they preprocessed using Natural Language Processing (NLP) to understand the meaning and to remove unwanted information from it. Finally, the output from NLP is converted into feature vectors. These feature vectors are given as input to the neural network and a similarity score associated to each expression is produced as output. The expression with the maximum score is considered as the classification result.

The AptitudeQS component is the central unit of our system. This component is responsible for starting the training process. It accepts the question from the user and consults the neural network to identify the expression to which it belongs to. Once the expression is known, the entity recognizer matches the question with the expression to extract the entities from it. The intent associated with the expression is then invoked to compute the solution.

4. TRAINING NEURAL NETWORK

An Artificial Neural Network (ANN) is a computing system inspired by the biological neural network present in animal brains [5]. The main idea here is to build an ANN model to classify the user's query to a matching expression. Once the classification of the query is done and a matching expression is identified, the corresponding intent is invoked to compute the solution.

A prerequisite to understand the input given by the user is to train the system with predefined datasets. The steps involved in training phase are generating training dataset, pre-processing, vectorizing and training of neural network. The system begins by generating a set of sample questions from the expressions in the Dialogs component to form a training dataset. Since the neural network accepts only fixed sized inputs containing numerical data, the sample questions, before feeding for training, need to be converted into a form that the neural network can understand. This task is done by Preprocessor and Vectorizer components. After this, the neural network is trained using the converted training dataset and can be used for prediction, to find out to which expression in the Dialogs a given query matches.

4.1 Generating training dataset

A dialog is a collection of expressions with corresponding intents. An expression is a pattern that defines the user input, while an intent is a void function with expression and entity attributes. The proposed system has a set of dialogs like "*profit and loss*", "*average*", "*clocks*" and "*ages*" which are used to train the neural network. An example of an expression is as follows:

"*Find the average of prime numbers between @number.integer:num1 and @number.integer:num2*"

where, *@number.integer* is an entity that is used for parsing integer values present in the query, and *num1* and *num2* are the alias names given to the two entities. *num1* refers to the first integer number in the query, while *num2* refers to the second integer value.

To train the neural network, a set of sample data is required. For this, we generate a set of sample queries for each of the expressions present in the system. The sample queries are generated by replacing any entities present in the expression with the values that they define. For example, using the above expression, a sample query can be generated by replacing the entities *@number.integer:num1* and *@number.integer:num2* by random integers such as 10 and 20. The so formed sample query after replacing will be

"*Find the average of prime numbers between 10 and 20"*

In this way, at least five sample questions are generated for each expression. Let *T* indicate the training dataset. For each of these sample questions, a dictionary is created with a key named *expression* to store the expression and another key named *query* to store the sample query. The *expression* part behaves as label while the *query* part behaves as data. The dictionary is then appended to the training dataset *T*.

Table 1: Variable definition.

Symbols	Description
T	Labeled training dataset
W	List of words in each data
E	List of expressions
IW	List of words or symbols that needs to be ignored
D	Document in our corpus
training	Training dataset after vectorization
output	List of output labels
output_empty	Initial output label filled with zeros
X	Input dataset to neural network
Y	Output dataset matrix of neural network
hno	Number of hidden neurons present in neural network
s0	Synaptic weights of links between input layer and hidden layer
s1	Synaptic weights of links between hidden layer and the output layer
psu0	Updates of previous synaptic weights between input layer and the hidden layer
psu1	Updates of previous synaptic weights between hidden layer and output layer
lmerror	Last mean error
sdcount0	Direction count of *s0*
sdcount1	Direction count of *s1*
E	Epoch – number of iterations of training
l1error	Hidden layer error
l2error	Output layer error
l1delta	Layer 1 error rate
l2delta	Layer 2 error rate
s1wu	Synaptic 1 weight update
s0wu	Synaptic 0 weight update
QW	List of words present in a query

Algorithm 1 describes the process of generating documents, expressions and words from the training dataset. All variables are shown in Table 1 along with their description.

Algorithm 1 Generating words, expressions and documents.

Input: Labeled dataset *T*
Output: Word List *W*, Expression list *E*, Document list *D*

```
1:  procedure GENERATE_WED(T)
2:    W ← null
3:    E ← null
4:    D ← null
5:    IW ← { '.', '?', ',', '-' }
6:    for p in T do
7:      w ← nltk.word_tokenize(p[expression])
8:      W.extend(w)
9:      D.append((w.p[expression]))
10:     if p[expression] not in E then
11:       E.append(p[expression])
12:     end if
13:   end for
14:   W ← [stemmer.stem.(w.lower( )) for w in W if w
      not in IW]
15:   W ← list(set(W))
16:   E ← list(set(E))
17:   return W, E, D
18: end procedure
```

The algorithm begins by looping through each query in the training data *T*. For each query, the algorithm tokenizes it, using the word_tokenize method of NLTK Python package, into a list of words. These words are appended to the word list *W*. A tuple consisting tokenized words and the expression to which the query belongs is created and then appended to the documents list *D*. The word list *W* is then refined by removing the unwanted words or symbols from it which are present in the ignored word list *IW* (from line 6 to line 13). In line 14, each word in *W* is transformed to lowercase letters. Finally, the algorithm converts the lists *W* and E, each consisting of unique words and expressions respectively. The algorithm ends by returning *W, E, D.*

4.2 Transforming training data into bag of words

The next step after generating training data and organizing data structures for words, expressions and documents is to transform the training data into bag of words. Text Analysis is a significant field for machine learning algorithm. The raw data, sequence of symbols cannot be directly fed to the neural network because most of them expect numerical feature vectors with fixed size. To address this problem, we convert text into fixed-length vectors of numbers using Bag-of-Words (BoW) model [6]. This model focuses on the occurrence of words in the document and does not keep track of their order. In this model, each word is assigned a unique number. The document is encoded as a fixed-length vector with the length of the vocabulary of known words. The value in each position in the vector is filled with a count or frequency of each word in the encoded document.

Algorithm 2 Training data vectorization algorithm

Input: Document *D*
Output: Preprocessed dataset

```
procedure TD_VECTORIZATION(D)
  training ← [ ]
  output ← [ ]
  Output_empty ← [0] * len(expression)
  for d in D do
    bag ← [ ]
    pattern_words ← d[0]
   pattern_words ← [stemmer.stem(word.lower()) for word
  in pattern_words]
    for w in W do
      if w in pattern_words then
        bag.append(1)
      else
        bag.append(0)
      end if
    end for
    training.append(bag)
    output_row = list(output_empty)
    output_row[classes.index(d[1])] = 1
    output.append(output_row)
  end for
  return training, output
end procedure
```

Algorithm 2 describes the process of preparing the training data in terms of vectors and the output label in terms of numbers. In this algorithm, each of the queries present in document is converted into a fixed-length vector using the BoW model. It first gets all the tokenized words of the query. Then the algorithm iterates through the word list *W*. If a word in *W* exists in the query, a 1 is appended for that particular word in the *bag* list, otherwise 0 is appended. Finally, an output label is created by adding 0 at all the indices except for the index belonging to the expression that the query matches. The algorithm returns the *training* and the *output* lists.

4.3 Training the neural network

The classification of a query entered by the user to identify expression to which the query matches the most can be attained using an Artificial Neural Network (ANN). Neural Network is an information processing model that process information. The main component is the novel structure of information processing system. Information processing system takes information in one form and processes it into another form. Neural network is organized in layers. Layers are made up of interconnected 'nodes' that includes activation function which defines the output of the node for the given set of inputs.

An activation function in neural networks is used to determine its output. It maps the resulting values in between 0 to 1. AptitudeQS uses the Sigmoid function as activation function.

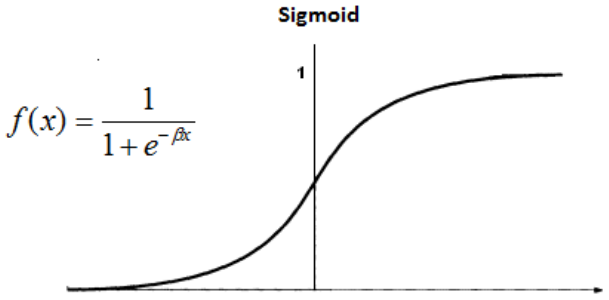

Figure 2: Sigmoid function

The Sigmoid function curve looks like an S-shape, as shown in figure 2. The main reason to use Sigmoid function is because the curve always exists between 0 and 1. Therefore, it is used by our neural network to predict the probability as an output. This system uses Sigmoid function to normalize values and its derivative to measure the error rate. The value Sigmoid function can be calculated using the following equation:

$$f(x) = \frac{1}{1+e^{-x}} \quad (1)$$

Algorithm 3 Training the neural network

Input: Input dataset *X*, Output dataset matrix *y*, Number of hidden neurons *hno*, Epoch *e*, Alpha *a*
Output: Synaptic weights *s0*, *s1*

```
procedure TRAIN( )
  np.random.seed(1)
  lmerror ← 1
  s0 ← 2*np.random.random((len(X[0]), hno)) - 1
  s1 ← 2*np.random.random((hno, expression)) - 1
  psu0 ← np.zeros_like(s0)
  psu1 ← np.zeros_like(s1)
  sdcount0 ← np.zeros_like(synapse_0)
  sdcount1 ← np.zeros_like(synapse_1)
  for j in iter(range(e + 1)) do
    layer_0 ← X
    layer_1 ← sigmoid(np.dot(layer_0, s0))
    layer_2 ← sigmoid(np.dot(layer_1, s1))
    l2error ← y – layer_2
   if (j % 10000) == 0 and j > 5000 then
      if np.mean(np.abs(l2error)) < lmerror then
        lmerror ← np.mean(np.abs(l2error))
      else
        break
      end if
    end if
    l2delta ← l2error *
sigmoid_to_derivative(layer_2)
    l1error ← l2delta.dot(s1.T)
    l1delta ← l1error *
sigmoid_to_derivative(layer_1)
    s1wu ← (layer_1.T.dot(l2delta))
    s0wu ← (layer_0.T.dot(l1delta))
    if j > 0 do
      sdcount0 ← sdcount0 + np.abs(((s0wu > 0) + 0) -
((psu0 > 0) + 0))
```

```
29:        sdcount1 ← sdcount1 + np.abs(((s1wu > 0) + 0) -
       ((psu1 > 0) + 0))
30:      end if
31:      s1 ← s1 + a * s1wu
32:      s0 ← s0 + a * s0wu
33:      psu0 ← s0wu
34:      psu1 ← s1wu
35:    end for
36:    return s0, s1
37:  end procedure
```

For error propagation, it is necessary to find the derivative of the output of the sigmoid function. This derivative can be calculated using the following equation:

$$f'(x) = f(x) * (1 - f(x)) \quad (2)$$

Now that the algorithms to calculate the sigmoid function and its derivative are defined, it is time to develop a training function to create synaptic weights of the neural network. Algorithm 3 describes this process of training the neural network.

Finally, the algorithm returns the synaptic weight between layer 0 and layer 1, *s0*, and the synaptic weight between layer 1 and layer 2, *s1*. These weights are converted into list and stored in the form of JSON in a file named "synapses.json".

5. PROCESSING QUERY

This section describes about how the user's query is processed to compute the solution. The query asked by the user is first accepted by the dialog controller. If this is the first time an instance of dialog controller is created, it calls Algorithm 3 for training.

5.1 Pre-Processing

Intention of the query needs to be understood to process it. [7]. This can be done by using Natural Language Processing. The next step is to pre-process the query entered by the user. The main task of pre-processing is tokenizing, stemming and transforming all the letters into lower case. The NLP process is done by using Natural Language Toolkit (NLTK) Python package [8].

- Tokenization – The process of breaking up the given text into units called tokens. The tokens may be in the form of words, numbers or punctuation symbols. The main aim of tokenization is to explore the words in the given query and to give an integer id to every token. The list of tokens becomes input for vectorization.
- Stemming – The process of reducing words to their root form. The main goal of stemming is to reduce inflectional forms and convert it to the base form. English Stemmer is used for reducing the words to their root form.

Algorithm 4 Pre-processing algorithm

```
Input: User's query Q
Output: Words in the query QW
1:   procedure PREPROCESS( )
2:     QW ← nltk.word_tokenize(Q)
3:     QW ← [stemmer.stem(word.lower()) for w in QW]
4:     return QW
5:   end procedure
```

Algorithm 4 is used to pre-process a query given by the user. It begins by breaking the query statement into a list of words called tokens (line 2). Each of these words are transformed into lower case and stemmed down into their root form (line 3). The algorithm then returns a list of preprocessed words.

5.2 Vectorizer

The vectorizer component takes in the user's query as input and converts it into a fixed size vector using BoW model. The following algorithm explains the process of vectorizing.

Algorithm 5 Vectorization Algorithm

```
Input: User's query Q
Output: Vector V
1:   procedure VECTORIZE( )
2:     QW ← PREPROCESS(Q)
3:     bag ← [0] * len(W)
4:     for s in QW do
5:       for i,w in enumerate(W) do
6:         if w == s then
7:           bag[i] ← 1
8:         end if
9:       end for
10:    end for
11:    return np.array(bag)
12:  end procedure
```

In algorithm 5, a list named *bag* of length similar to that of words list, *W*, is created and filled with 0s. It calls Algorithm 6 with the query to obtain a list of pre-processed words, *QW*. The algorithm then iterates through *W* and checks if each word is in *QW*. If that word is present in *QW*, a 1 is added to the *bag* at the position corresponding to that word. In this way, the algorithm returns a vector that represents the user's query.

5.3 Neural Network for prediction

Algorithm 6 creates an initial neural network model. The neural network used in this system has three layers: an input layer that accepts vector inputs, a hidden layer that processes the information and an output layer that produces the output.

Algorithm 6 Prediction algorithm

Input: User's query *Q*
Output: Output layer of the neural network *l2*

```
1:  procedure PREDICT( )
2:    x ← VECTORIZE(Q.lower, W)
3:    l0 ← x
4:    l1 ← SIGMOID(np.dot(l0, s0))
5:    l2 ← SIGMOID(np.dot(l1, s1))
6:    return l2
7:  end procedure
```

Now that the neural network model is developed and trained using the sample training data generated earlier, it is now possible to classify the query entered by the user to which expression it belongs to. Algorithm 7 describes this classification process. In this algorithm, we maintain a minimum error threshold of 0.7. After prediction, the output of the neural network with the highest score is considered as the prediction result only if the score is greater than the threshold. This algorithm finally returns the classified expression.

Algorithm 7 Classification algorithm

Input: User's query *Q,* Synapse weights *s1* and *s2*
Output: Expression to which the query belongs *e*

```
8:  procedure Classify( )
9:    Error_threshold ← 0.7
10:   results ← PREDICT(Q)
11:   results ← [[i,r] for i,r in enumerate(results) if r >
      Error_threshold]
12:   results.sort(key = lambda x: x[1], reverse=True)
13:   e ← [[expression[r[0]],r[1]] for r in results]
14:   return e
15: end procedure
```

5.4 Entity Recognizer

Before calling the Invoke method of AptitudeQS component, it is required to extract the entities, if any, from the query. The task of extracting entities is done by Entity recognizer. This component first identifies the entity types present in the expression. Based on these entity type, it compares the query with the expression to locate the values of each of these entity types. Entity recognizer then returns a list of tuples containing the entity types along with their values.

6. RESULTS AND PERFORMANCE EVALUATION

We have implemented our system using python 3.6.0 programming language on a system running Windows 10 Operating System. For evaluating the performance of the system, testing, was carried out on AptitudeQS. Table 2 shows the test cases for AptitudeQS and its results for the given input.

Table 2: Test Cases.

Question	Find the average of first 10 prime numbers.
Expression	Find the average of first @number.integer:num2 prime numbers
Expected Output	4.25
Output	4.25
Result	Success

Question	Find the average of all prime numbers between 30 and 50.
Expression	Find the average of all {prime} numbers between @number.integer:num1 and @number.integer:num2
Expected Output	39.8
Output	39.8
Result	Success

Question	Find the average of first 40 natural numbers.
Expression	Find the average of first @number.integer:count natural numbers.
Expected Output	20.5
Output	20.5
Result	Success

Question	Find the average of first 20 multiples of 7
Expression	Find the average of first @number.integer:count multiples of @number.integer:num
Expected Output	73.5
Output	73.5
Result	Success

Test ID	**5**
Question	The average of four consecutive even numbers is 27. Find the largest of these numbers.
Expression	The average of @number:count consecutive @evenOrOdd numbers is @number.integer:average. Find the @largestOrSmallest of these numbers.
Expected Output	30
Output	30
Result	Success

The goal of the evaluation is to find the accuracy of the system. Based on the test cases executed using few sample

questions, it is estimated that the system provides a classification accuracy of 98%.

7. CONCLUSION

In this paper, we have proposed Aptitude Question Solver that solves mathematical word problems. AptitudeQS accepts query from the user and provides detailed solution. This helps the users to learn how to solve aptitude questions and improves their skills in solving. Currently the AptitudeQS system solves four classes of questions, which are average, profit and loss, ages and clocks. In future, the system can be upgraded to solve other categories of questions. The system can also be enhanced for solving aptitude questions by accepting speech as an input.

REFERENCES

[1] D. Cummins et al., "The role of understanding in solving word problems", Cognitive Psychology, vol. 20, pp. 405-438, 1988.

[2] Wolfram Alpha (2009). [Online]. Available: http://www.wolframalpha.com/

[3] MathWay, online available at http://www.mathway.com as on 18-05-2018

[4] WebMath, online available at http://www.mathway.com as on 18-05-2018

[5] "Artificial Neural Networks as Models of Neural Information Processing | Frontiers Research Topic". Retrieved 2018-02-20.

[6] McTear, Michael (et al) (2016). The Conversational Interface. Springer International Publishing.

[7] Sarkar et al., "NLP Algorithm Based Question and Answering System",

[8] Natural Language Toolkit. (2001). [Online]. Available: http://www.nltk.org/

[9] Dong. X. L. Murphy.K.Gabrilovich .E. Heitz.G.Horn. W.Lao.N. & Zhang.W. (2014). Knowlegde Vault. A web-scale approach to probabilistic knowledgr fusion.

[10] R. Schumacher and L. Fuchs, ―Does understanding relational terminology mediate effects of intervention on compare word problems?,‖ Journal of Experimental Child Psychology, vol. 111, pp. 607-628, 2012.