ABSTRACT

Crowdsourcing has applications where machine learning falls short. We describe the design, implementation, and deployment of CrowdReply, a crowdsourced multiple choice question answering (MCQA) system. We deployed an Android app to enable the audience watching “Who wants to be millionaire?” quiz-show on TV to play along on their smartphones simultaneously. The app enabled us to collect data from thousands of users about MCQA dynamics. Our findings indicate that it is possible to aggregate the crowd’s answers to build a super-player.

1. INTRODUCTION

Crowdsourcing has applications where machine learning falls short. Crowdsourcing has been used for image tagging on the web, and finding applications to enable the blind to “take a picture, speak a question, and get an answer” in real-time [6]. It has also been employed for answering subjective, relative, or multidimensional location-based queries, for which the traditional search engines perform poorly [7, 9]. Other applications of crowdsourcing include citizen science (e.g., the Audubon society national bird counting survey), and surveillance (monitoring public building cameras for dangerous events, border monitoring, etc.) [10].

In this paper, we consider the problem of building a crowdsourced multiple-choice question-answering (MCQA) service. The insight behind the MCQA service is that, for crowdsourcing, asking multiple choice questions is more productive than asking open-ended questions. Open-ended questions lead to varying open-ended answers and makes it hard to aggregate the answers from the crowd to provide a final answer. Also, multiple-choice questions make the job of the crowd easier: punch in the choice (A, B, C, or D) instead of figuring out how to present the answer. Although it is beyond the scope of our current work, we suggest that providing multiple choice questions is often feasible. The original asker can provide the multiple options when the question is about deciding among some choices (hotels to stay, products to buy, music to listen). It may also be possible to automate the process of adding multiple choices to an open space question using ontologies or lightweight machine learning techniques [12].

Properly aggregating MCQA for identifying the best answer is an important problem for building practical crowdsourced MCQA systems. To study the real-world dynamics of MCQA systems, we built a crowdsourced system to play “Who wants to be a millionaire” (WWTBAM) [4] TV quiz-show live. Our work was inspired by IBM Watson’s success at Jeopardy and aims to utilize the crowd (not a supercomputer!) to answer WWTBAM questions accurately and fast. We developed an Android app and the backend software to enable the audience watching WWTBAM on TV to play along on their smartphones simultaneously. We targeted a Turkish audience due to the high popularity of the show there. (Our app has been installed more than 207K times [2].) When the show is on air in Turkey, this app makes a notification sound to alert users to pick up their phones and start playing. Our two project members type the questions and multiple choices as they appear on TV, and the app users enter their answers using their phones. The users are incentivized to participate as they enjoy the game-play and the participation; they can see their ranking among the other players, and share their scores on Facebook and Twitter.

The game enables us to collect large-scale data about MCQA dynamics. The ground-truth of a question is the correct answer announced on TV. The question difficulty is determined by the show’s format: There are up to 12 questions with increasing difficulty levels. For
the first 2 questions there is a 15 seconds time limit, and for the next 5 questions there is a 45 seconds time limit. The remaining 5 questions has no time limit and can be answered without time pressure. There are total of 2 safe zones, at questions 2 and 7, where the contestant guarantees to get the earned money even if she fails to answer the upcoming questions. The second safe zone presents a very clear breakpoint to differentiate easy and hard questions. In Section 3, we use this when analyzing our experimental results.

The data we collect suggests that it is feasible to build a superplayer by aggregating answers from many players. By just going with the most selected answer, we can answer 88% of the questions correctly, however these are mostly entry-level questions. After question 6, the most selected answers’ success rates plummet to 60%, 50%, and 30% quickly with each consecutive question. We investigate how to improve the success rates of these tougher questions. As one approach, our app asks the players to enter a confidence level (“certain”, “guessing”, and “no idea”) when they answer a question. Our results show that by taking the confidence levels into account, we can increase the success rates for the tougher questions. As another approach, we investigate whether the latency of answers can also be helpful in determining the correct answer. In this work, we focus on non-indivudual methods in order to investigate techniques that would fit systems with high worker churn or worker anonymity. We leave per-user/expertise-based approaches [5] as future work, such as using the answer history of the users or their social network profiles to learn about their expertise and weighing this in accordingly when aggregating answers.

2. CROWDREPLY DESIGN

The overall architecture of CrowdReply is shown in Figure 1. CrowdReply consists of three main parts, an admin part for entering questions & multiple choices while the game is live on TV, a mobile side for presenting questions to users and letting them answer the questions, and a server side for dispatching questions, collecting answers, and providing useful statistics. We have 7 KLOC for Admin and Server side, and 2 KLOC for mobile application.

2.1 Admin side

The questions are entered into the system manually as they appear live. Since, the time allowed for the first 7 questions are limited, we distributed question typing to two project members. One types in the question, and the other types in the multiple choices simultaneously. The server merges and sends these to the mobile clients.

2.2 Mobile side

Mobile side consists of three subcomponents that are MessageService, LiveGame, and Statistics. In order to enable the interaction between mobile and server side, we utilize Google Cloud Messaging (GCM) [1]. GCM is a push notification service provided by Google to send data to Android powered devices. GCM is provided as free and without any quota. By delegating our message pushing tasks to GCM, we avoid any scalability issues we might encounter during the game.

MessageService: MessageService is responsible for parsing and processing the GCM messages sent to our app. The GCM messages are initiated by our own server and pushed to our app clients through GCM. Once a message arrives to the app, MessageService parses and takes appropriate action such as sending questions to LiveGame component.

LiveGame: Figure 2 shows a screenshot of the LiveGame user interface. LiveGame presents the current question along with the answer and statistics of the previous question. In addition, LiveGame also sends the user’s answer back to our server. All the operations are done asynchronously and in an exponential back-off manner in order not to block game play and allow client app to retry if it gets no response from our server due to high traffic load.

Statistics: Statistics component provides the list of first 100 users of each game, ranked based on the correct answer count and response time.
2.3 Server side

Server side consists of five subcomponents, namely Asker, Collector, UserManager, DBHandler and Client APIs. Server side is responsible for managing the game play and providing the data persistence of the system. It is implemented using Java servlets, and runs as an Apache Tomcat web service.

Asker: Asker manages the questions and forwards them to the users in real time during the TV game show. As explained in the mobile side, we use Google Cloud Messaging (GCM) for question distribution. This way, we are able to push the questions to the apps rather than polling for the questions from the apps, and this gives real time playing experience and also reduces the server side load by using a connection just when it is needed.

Collector: Collector provides APIs to the mobile clients for submitting the user responses. We use Java HTTP Servlets to handle requests from mobile devices on server side, through the client API mentioned below.

Decider: Decider component implements our decision-making algorithms. It provides services for online feedback to the users as seen on Figure 2 as well as post-processing of responses for more detailed analysis.

UserManager: UserManager manages the user related contents. It defines user properties such as profile information and handles user specific functionality such as performance tracking and ranking.

DBHandler: DBHandler manages the data persistence and provides an abstraction for the other subcomponents and servlets for logging, storing, and accessing application data. It stores the data using a MySQL server.

Client APIs: This component provides APIs for various information and statistics about the game. It also provides a RESTful API (using JSON formatted messages) for answer collection and other interactions with client applications.

3. EXPERIMENT RESULTS

In this section, we present the results for CrowdReply. Over the period of 20 weeks, we collected 1.5GB of MCQA-related data using our Android app. The data consists of 619 questions and 130K live answers to these questions. Our data also includes timestamps of events, statistical and experimental extractions, user device details and GCM logs. After we clean and anonymize the data, we will share this dataset with the crowdsourcing research community in order to advance the understanding of MCQA dynamics. We consider this as another contribution of our work.

3.1 Timing of questions

Since we use GCM for question distribution, we measured if GCM provides sufficient speed to provide real-time game play. For a GCM message, we timestamp the server initiation and client arrival using NTP time [3]. We then analyze these timestamps to evaluate arrival latency of GCM messages. The results show that, among the users who play the game live, we are able to send the message to the vast majority of them within 6 seconds, and reach almost all the users by 12 seconds. Figure 3 shows the cumulative distribution of message sending times.

3.2 Accuracy of answers

For aggregating the answers of the users, we applied three decision algorithms: the majority-voting decision algorithm and two more confidence-based decision algorithms. Figure 4 shows the answer accuracy rates per question level using these three algorithms.

Majority voting: Using basic majority voting, crowd is able to answer 88%-92% of the lower level questions correctly. As seen in Figure 4(a), while majority voting is very successful on lower level questions, its success rate plummets on higher level questions. The results also reveal two interesting observations about the format of the TV show. First, there seems to be a hidden barrier on the 6th question, the one before safe zone
(i.e. 7th question) to guarantee a major award. Second, hardness of questions increase quickly after the safe zone. Especially between 7th and 8th question, there is a clear breakpoint which can also be observed on answering times in Section 3.3.

Confidence weighted: 88% seems as a promising rate to build on. We introduce answer-confidence levels to pull it up. Confidence is a label (“certain”, “guessing”, and “no idea”), which are generated by user along with her answer. The plain confidence level data is noisy as some users select the same confidence level for all the answers. This reduces the improvement of the confidence-level based decision algorithm over the majority voting. In order to eliminate the noise, the variation of confidence labels per user is calculated. Since we have enough data thanks to the popularity of our app, we are able to ignore answers from users with no confidence-level variation. Then we calculate total votes for every choice by weighing with answer’s confidence on it. However, confidence data still does not have much to offer for low level questions, i.e. below 7th question where majority-voting is actually successful enough. Both algorithms find the correct answer for more than 95% of easy questions and failure is a rare case. Confidence-weighted algorithm is able to answer some of the questions that majority fails on, but on the other hand it fails on some other questions where majority voting answers correctly. However, confidence weighted decision algorithm seems to be successful on harder questions, i.e. after 7th question where majority voting fails more.

Confident voting: This algorithm is similar to the majority voting, but this time it only considers the answers labeled as “certain”. Note that we have eliminated the noisy confidence level data for this method too. Our assumption here is, when a user labels an answer as “certain”, she is sure about it because she has labeled some other questions as “uncertain” (otherwise her all confidence labels are already eliminated as noise).

Figure 4 (c) shows the success rate of this algorithm. Confident voting outperforms the confidence-weighted decision algorithm slightly on higher level questions. On the other hand, noise elimination and picking only “certain” labeled answers reduces the data size rapidly. Therefore, using confidence-weighted decision algorithm and confident-voting together would be more dependable when the data size is not big enough.

A last note about the confidence-based algorithms is that, the 100% success rate for level 11 questions might be caused by situation rareness. In this 20-week period, we observed only 4 of level 11 questions, which is only 5% of level 7+ questions. And we observed no level 12 questions yet.

3.3 Timing and distribution of answers

In this part, we explore how the response-time and response-count change per question level. In our analysis, we examine correct and incorrect responses separately. As shown in Figure 5, both the number and the ratio of correct answers decreases as the level of question increases. This is an expected outcome due to the fact that contestants’ ability to answer the questions decreases as the question level increases. Also note that incorrect response ratio increases proportional to the level of question.

Another facet of Figure 5 is the response time of correct and incorrect answers. As shown in the Figure 6, both correct and incorrect response time increases as the difficulty of questions increase. Moreover, there is also a sharp increase in response time at the 7th question for both correct and incorrect answers. Since the 7th question is a barrier question, once the contestant reaches to this point, difficulty of questions increases substantially and therefore results in higher response time. We could

Figure 6 represents the data for last 8 weeks instead of all 20-week period, as in our early experiments, we were timestamping the events on the server side. We then observed server side timestamping is not accurate enough for timing analysis since several factors such as GCM or queuing delays affect it excessively. Therefore, for the last 8 weeks, we use mobile side NTP timestamping as described in Section 2.2.

1 In our future work, to keep this confidence level assessment accurate, we will award/deduce virtual coins to/from the user by factoring with respect to this confidence level.
Finally, correlation between the correctness and response time of answers demonstrates an interesting phenomenon. Response time for incorrect answers is higher than the correct ones until the second safe zone, i.e. 8th question. However, it is the opposite for the higher level questions. Furthermore, slower increase on the response time after the 8th question is prevalent. This is due to openness of noisy and risky behavior of faulty responders. We interpret this as an exciting finding as it represents a distinct metric, which neither needs comparative verification nor depends on prior information like user history or gold data, for quality control of crowdsourcing [11].

5. CONCLUSION

We describe the design, implementation and the deployment of CrowdReply, a crowdsourced multiple choice question answering system. We propose and compare three decision algorithms to obtain the correct answer. We first consider majority voting and are able to find the correct choice for the entry level questions with 88% success rate. However, success ratio is decreased considerably as the question difficulty increases. Therefore, we develop two other methods based on the confidence-level of the users’ responses. We find out that using confidence-level is especially useful for answering difficult questions in which majority voting falls short. Our findings indicate that using crowd to answer multiple choice questions is feasible and put creating a super-player from the crowd into the realm of possibility.

5. REFERENCES