Introduction:
(Highlight the social and privacy-awareness features of the application)

With the increasing use of the mobile phones in our daily and social lives has given birth to the dire need for the users to take care of the ringing modes in different environments. It’s a really sensitive aspect of using a mobile device and users must be very careful while letting her phone ring in a social setup. There are embarrassing situations that we face regularly when our phones make sounds in settings like a meeting or in class. On the other hand, we sometimes miss calls because our phone is in silent mode when we are in a noisy environment. Also, sometimes the urgency of the call and the importance of the caller plays a role in whether we must receive a call or not. Keeping all these things in mind, it would be really helpful if we have an automatic ringer manager which decides the ringer mode of the phone depending on the different variables just mentioned above. With that in mind, we present an architecture here which helps us design such an application. The central idea of the design here rests on multi-class decision trees and linear aggregation of different sub-attributes on which the final decision depends. Our models are very flexible in ways it can add new people and places to its database and also learns incrementally from the feedbacks that we receive from the past calling experiences. Also it uses a dynamic training data generation based on initial feedbacks and that makes this model much robust compared to naive models. The Latest model also supports Argumentation theory in a way where it synthesizes the Rationales behind the predictions that are made using the models. It also analyses the neighbors’ behaviors on different calls and makes some comments on those as well. Essentially, it sends two feedbacks to them, first one based on own beliefs and the updated one after analyzing the callee’s Rationales on that particular decision. Finally, it
it dynamically updates its beliefs on Rationales based on the incoming Rationales from its neighbors. We use a web service which simulates a social network of family, friends, colleagues and strangers and their different expectations in different social settings. We also present a java project which implements the whole architecture.

**Actors, Goals, Plans:**

*List the actors, goals, and plans*

**Actors:** The primary actors in the system are the device users, his neighbors in his immediate social environment and the caller who makes a call to this user.

**Goals:** The main goal of this application is to achieve a equilibrium of satisfaction for both the caller and the neighbors so that everyone is happy from the decision of the call receive or rejection which is a direct function of the ringer mode. It also sends the rationales behind the decision makings and also has features for analyzing the rationales of others in the system.

**Plans:** Find a way to perceive the data from the environment and the caller and take decisions which will predict the ringer mode that is most likely to make everyone happy. We also have ways to receive feedbacks on past calls so that we can update the models making the decisions. Keep a dynamic set of beliefs in terms of rationale arguments and makes feedback decisions based on that.

**How can we achieve this?**

*Produce system-as-is and system-to-be models*

The *as-is* model of a system describes the current situation which is prevailing in the user’s perspective. It is normally used to state the problem of the existing model and the aim of the experiment we are trying to perform here. The *to-be* model captures the advancements that we are proposing on top of the existing model using which we can eliminate shortcomings of the as-is model.

The *system-as-is* for the current problem can be defined as follows. The ringer managers that we have in our devices are manual usually, where we have to design different user profile suiting to different environments. Then whenever we enter one scene, then we set that profile manually. We miss that, everything is a mess!
If we see the existing setup we see that there are plenty of issues. The system-to-be that we are proposing here tries to set the ringing mode for a particular call automatically. It senses the environment using sensors and also uses a database of the caller information to decide whether to make the user aware of the call. We have considered three ringer modes here, Loud, Vibrate and Silent with decreasing order of urgency of the call. The updated system also shares the decision rationales behind every prediction. In addition to that, it analyses the prevailing calls in the arena and gives normal and updated feedbacks to the neighbors. It then updates its own set of belief concerning the decision rationales, if needed.

**Implementation:**
*(Details of the application)*

Predicting a ringer mode depending on different variables is inherently a multi-class classification problem which has mostly ordinal attributes and an ordinal class variable. The response variable under consideration here is the Ringer Mode for the call. The possible ringer modes we consider here are Loud, Vibrate and Silent. The independent variables under consideration here are,

- Location
- Noise level (currently it’s a static function of the Location and its type)
- Neighbor’s Information (Name, ID, Relationship Types, Relationship Strength, Current Ringer Mode, Expected Ringer Mode)
- Caller’s Information (Name, ID, Call Id, Relationship Type, Strength of the Relationship)
- Urgency of the call
- Brightness of the place

These abstract attributes are finally mapped into actual ordinal attributes which are actually capable of taking a decision in terms of the final predictable ringer mode. The final attributes which are used for learning purpose are (the actual attributes it depends on are given in the parenthesis),

- LOCATION_TYPE (Location Name)
- NOISE_LEVEL (Location Name)
- NEIGHBOR_JUDGEMENT (Relationship Type, Relationship Strength, Current Ringer Mode, Expected Ringer Mode)
- CALLER_EXPECTATION (Relationship Type, Relationship Strength)
- URGENCY (Urgency of the Call)
• BRIGHTNESS_LEVEL(Location Name)

The Location Type, Urgency, Brightness Level and the Noise Level are pretty much self explanatory in nature. The Neighbor Expectation and the Caller’s expectation are two new features which are synthesized from the information on the neighborhood and the caller respectively. The aggregation scheme that we have used here rests on the notion of weights which corresponds to different levels. All the features used here for the prediction here are ordinal attributes and hence we have assigned numerical values to them as in decreasing order of preference for the loudness of the ringing tone. For example, say we have an enumeration which talks about the URGENCY. It has three values NORMAL, CASUAL and URGENT. So the values will be 1 for NORMAL, 5 for CASUAL and 10 for URGENT. This signifies that the higher value means high probability of favoring a loud call. We have kept a general notion of assigning 1 one to the lowest value and then increment in steps of 5. Using this value assignment, we can now quantize the rankings from different variables with equal weightage. We use a linear combination of the values and then find out the range of the final rank. We divide the range in the no. of bins of the output prediction and then assign accordingly. For example, when we have to synthesize the CALLER_EXPECTATION value which can be any of the two values from {MUST_RECEIVE, SHOULD_RECEIVE}, we use the following formula,

$$final\_rank\_value = 0.5 * value \ of \ the \ Relationship \ Strength + 0.5 * value \ of \ the \ Relationship \ Type \ Strength$$

Where Relationship Strength can be any of {HIGH(10), MEDIUM(5), LOW(1)}
and Relationship Type can be {FAMILY(15), FRIEND(10), COLLEAGUE(5), STRANGER(1)}

The weights above are assigned according to the scheme just stated above. Similar to the caller’s expectation, we also aggregate the values of the neighbor’s expectations and map their collective opinion as one of the three possible expected ringer modes namely, Loud, Vibrate and Silent. Here the trick is we may have up to N different neighbors, so we must to a normalization so that the final value of the aggregated rank has a fixed range. We finally divide the range into no. of bins equal to the possible values to which the aggregations will be mapped. For example, caller’s expectation has 2 possible values and the range for the above equation is 12.5. Dividing the range in 2, we have 1 - 6.75 for SHOULD_RECEIVE and 6.75 - 12.5 for MUST_RECEIVE. This aggregation is naive but effective on capturing the collective opinion of the users so that we can feed them as inout attributes in the next modeling phase.
The modeling: Decision Trees:

The problem under consideration is a multi class classification problem where almost all attributes are nominal or ordinal. This setup signals a direct usage of decision trees, which are one of the most powerful models in Machine Learning. The basic idea of a decision tree learning is to start off with a basic combination of values for different attributes and learn a tree structure using which we can make predictions for upcoming data points which are represented in terms of those attributes. The central idea here is to start off with an initial train data which will enumerate sufficient combinations of values for the above 5 attributes so that we have a basic tree in our hand to make the initial predictions. We use something like this in the beginning,

```
LOCATION_TYPE|NOISE_LEVEL|BRIGHTNESS_LEVEL|NEIGHBOR_JUDGEMENT|CALLER_EXPECTATION|URGENCY|CLASS
PARTY|7|3|Loud|MUST_RECEIVE|URGENT|Loud
PARTY|8|7|Loud|SHOULD_RECEIVE|CASUAL|Loud
MEETING|5|7|Vibrate|MUST_RECEIVE|CASUAL|Vibrate
CLASSROOM|4|9|Loud|MUST_RECEIVE|URGENT|Loud
LIBRARY|2|9|Vibrate|SHOULD_RECEIVE|CASUAL|Silent
PARTY|10|3|Vibrate|MUST_RECEIVE|URGENT|Loud
PARTY|7|7|Loud|SHOULD_RECEIVE|URGENT|Loud
PARTY|10|3|Vibrate|SHOULD_RECEIVE|URGENT|Loud
LIBRARY|2|9|Vibrate|SHOULD_RECEIVE|URGENT|Silent
CLASSROOM|4|9|Vibrate|MUST_RECEIVE|URGENT|Vibrate
LAB|3|8|Vibrate|SHOULD_RECEIVE|CASUAL|Vibrate
LAB|3|8|Vibrate|MUST_RECEIVE|CASUAL|Vibrate
MEETING|5|7|Vibrate|MUST_RECEIVE|URGENT|Vibrate
```

We learn this basic tree in the construction time and keep in it the memory. When a new call comes in, we flatten the data in the call to the above 5 attributes and make a prediction on that data point using this tree. We use a dynamic update of the train data which is one of the most critical phases of the whole architecture, we will discuss that in detail shortly. We make the prediction and get back the feedbacks for that call. Now when the feedback for the last prediction comes in, we see the crowd’s judgement for the same call. If it was different than our judgement, then we go ahead and check if that combination of attributes are there in the initial train data, if it’s there then we go ahead and replace that point’s class variable to the new judgement of the crowd. This is the update phase of the system. Also, the system is flexible about adding new people and location data into it and behaves dynamically according to them. The possible values that we are using for the above mentioned final synthesized 5 variables are all ordinal attributes. They can take values as defined in the class EnumCollection.java in our implementation. A sample partial Decision Tree that is generated in the process looks like below,
Turns out that decision tree work very well for this setup under the assumptions made. Definitely it can give us a very reliable first point to start off.

**Dynamic Update of the Training Set:**

The principal idea behind the running of this algorithm is the dynamic update paradigm being used. There are two different kinds of updates happening here. First, we start with a minimal training data. Then obviously first the Decision Trees will fail to
predict. The we add this data point to the train data with an initial class variable value as suggested by the neighbor’s judgement aggregation. Then we recreate the decision trees in memory. After we are done with this first level of dynamic updates, we go for the second kind of updates. This happens when the feedbacks come in. We see if the combined sentiment from the feedbacks are favoring the last decision. We aggregate the feedbacks using a majority voting of the positive and neutral feedbacks compared to the negative feedbacks. If we have more combined positive and neutral feedbacks than combined negative feedbacks, we take it as a safe prediction, else we consider it as aren flag. Now we check to see if that particular data point exists in the train data, if yes then we update the class signal or else we add a new data point with updated class signal. The updated class label is estimated using a very simple logic. It’s evident that the negative feedbacks mostly come from the fact that the phone rang loudly when it wasn’t supposed to ring. So when we get a bad feedback, we lower the ringer level from Loud to Vibrate to Silent one step at a time. However, there is a catch. If the feedback is negative and it’s from the caller, then the whole scenario is completely different and behaves opposite. Currently we don’t have anything taken care for that. maybe going forward we can incorporate that.

**New changes for the Argumentation Implementation:**

In this section we discuss the new implementations related to the S_D part of the assignment in details. The basic idea here is to incorporate the sharing the “reason” behind making a certain prediction, which earlier we were not sharing. We also need to do some additional work of helping other users in the arena by providing them with initial and updated feedbacks. Then we can also go ahead and update the user’s models based on rationsales received from others.

In our implementation, we keep the initial decision tree models for the prediction purposes, just we also added the support for brightness levels which we left out last time. And also, while analyzing the feedbacks now, we give preference to the updated feedback over the first feedback, which are small changes. For the newer feature implementations, we have implemented 3 major classes which handles the Rationales. These classes are *RationalManager, DynamicArgumentManager* and *RationaleSerializerParser*. The Rationale Manager is the entry point which is owned by the RingerManagerCore class. The rational Manager makes the decisions based on the informations kept by it’s owned class Dynamic Argument manager. Dynamic Argument Manager maintains an initial set of complete mappings of the expected RINEGR MODEs with respect to different input values of the attributes. The Rational manager is asked by Ringer Manger Core to find out the best matching Rationale Behind a
particular prediction. The ringer manager then asks the dynamic argument manager for the expected beliefs and synthesizes the rationale based on the following order of preference of conditions.

URGENCY > CALLER_EXPECTATION > LOCATION_TYPE > NEIGHBOR_JUDGEMENT > NOISE_LEVEL > BRIGHTNESS_LEVEL

It also uses rules like,

**Rule 1**: If Urgency Matched and/or caller expectation matched, then don't even bother about others!

**Rule 2**: If one of the top 2 top conditions match, then look for a match from the next three, if there's one, then use that along with the first one and give a positive argument.

**Rule 3**: If none of the above true, then send the positive matches as Positive arguments and others as negative arguments.

The details of the implementations are properly documented in the respective code files, please look into for more details.

After deciding the best reasons for the prediction, it uses the helper class RationaleSerializerParser to construct the rationale string according to the given rules. Once it gets that back, sends to the Driver to send it back along with the prediction. After sending the predictions, it makes another call to get the feedbacks and then updates the models based on the feedbacks.

**Analyzing the calls in the arena:**

Having completed the flow of the RINGER MODE prediction, we now move to analyzing the neighbors calls and providing feedbacks to them. The Driver class collects the calls of the neighbors using the structure “CallInfo” and send back to call analyses module. It first sent the first feedback based on it’s own expectations and the LOCATION_TYPE. Then it collects the rationals of the neighbors’ precautions and send then to the Rational manager for analysis. It then forms the RationaleInfo struct out of the rationale strings using the RationaleSerializerParser class and sends those to the DynamicArgumentManager for processing. Those are then matched with the known beliefs which are maintained by the Argument manager. The class decides a second feedback based on the no of matches in positive arguments and negative arguments. It keeps track of the update requests, when it crosses a certain limit
MAX_UPDATE_REQUEST defined inside it, it sends for a update/replacement of beliefs. This completes the updating requirement of the assignment.

**Social Benefit Function: Pseudo Code:**

The main API that we have implemented is called “getRecommendedRingerMode()” and uses a decision tree and aggregated inputs for prediction purposes. A very simple high level pseudo code is given below for the function.

```java
RINGER_MODE getRecommendedRingerMode(LOCATION_TYPE, NOISE_LEVEL, NEIGHBOR_JUDGEMENT, CALLER_JUDGEMENT, URGENCY){
    1. Build the decision tree from the ground truth data.
    2. Map the location to location type.({OUTDOOR, PARTY, LAB, CLASSROOM, LIBRARY, MEETING, HOSPITAL};)
    3. Get the noise level for the particular call or map it from the static list corresponding to the locations(QUIET(1), NORMAL(5), NOISY(10);).
    4. Synthesize the neighbor’s judgement from the neighbors data collected from the environment with the API getAggregatedNeighborJudgement() and get the result as one expected RINGER_MODE({Loud, Vibrate, Silent})
    5. Get Caller’s Expectation mapped as values ({MUST_RECEIVE, SHOULD_RECEIVE}) using API getCallerExpectation()
    6. Map urgency of the call in URGENCY_TYPE {NONE, CASUAL, URGENT}
    7. Make a prediction using the above values of the attributes for a test data point.
    8. Synthesize a Rationale of the decision just made. This is achieved by keeping a dynamic model in memory which takes care of the Rationales with respect to the input conditions.
    9. Send the predictions and the corresponding rationales to the system.
   10. Get the feedbacks, analyze them and update the models based on those. Always give preference to the second feedback if it is present, otherwise just go with the first one.
    11. Optional : Check to find out the current users in the place.
    12. Optional : Send a feedback to the calls based on the current place and user’s expectation.
    13. Optional : Analyses the rationale behind the call and then send a second updated feedback.
    14. Optional : Use the newly received feedbacks to decide if we have to add to or replace to any previous beliefs.
}
```

**The Java Application for the simulation:**

There’s a web service running which simulates a social environment and feeds our application with all the data as needed. The application we have developed on our end has the following main classes.
1. **AttributeVectorInfo.java** (keeps data of the final attributes)
2. **CallerInfo.java** (Information of one particular call)
3. **Driver.java** (The Driver class which simulates one full flow, also the one interacting with the network to get the data)
4. **EnumCollection.java** (Lists all the different enumeration used for different ordinal variables used in the modeling)
5. **FeedbackInfo.java** (Information on the feedbacks received from a call)
6. **LocationManager.java** (Keep track of all the locations currently being used for the simulation, their types and noise levels)
7. **NeighborInfo.java** (information on the neighbors, basically it tells about the relation with that person)
8. **PeopleManager.java** (manages all the people in the database)
9. **RingerManagerCore.java** (*This is the main class which is responsible for owning and maintaining the decision tree structure, synthesizing the attributes and giving ten final recommendations*)

**Newly added Classes:**

1. **ArgumentInfo.java** -> Structure for storing one Argument which has a context keyword, predicates if any and the values.
2. **CallInfo.java** -> Information about one particular call. used when analyzing the calls of the neighbors.
3. **DynamicArgumentManager.java** -> This is the main class which keeps the ground truth beliefs for generating and analyzing the Rationales. It does three main tasks, creates the Rationales before sending out a prediction, analyses the incoming Rationales from the neighbors and keeps updating it’s models.
4. **RationaleInfo.java** -> Keeps information of the whole Rationale. It has two kinds of components. It stores the arguments, the predictions and the connectives for both ArgInFavor and the ArgInOpp counterparts.
5. **RationaleManager.java** -> Wrapper class for the DynamicArgumentManager class which own one DynamicArgumentManager in it’s memory and makes the decisions on how to handle the Rationales. It uses the Static Class and RationaleSerializerParser for generating and parsing the Rationales.
6. **RationaleSerializerParser.java** -> Utility class with static helper functions for generating the Rationale String to send out of the system and also to parse the incoming Rationale Strings from the neighbors.
The program Flow:

The basic flow of the system has roughly the following sequence from the point of view of the APIs that the Web Service ha exposed to us.
1. Enter place
2. List neighbors
3. Request call
4. Response call with prediction and rationale behind it
5. Get Feedbacks
6. Process Feedback and update models
7. Get Calls in current place
8. Give first and second feedback
9. Update Rationale Deciding beliefs.
10. Exit Place
11. Update the prediction models from the feedbacks received on the last call.

The inputs that are required from the user is the ID of the user, his current ringer mode, his expected ringer mode and the name of the place that he wishes to enter. The logic is implemented fully in the class Driver.java in our application. There are 2 simulation APIs in the code. “simulateOneFlow()” takes in the above values and simulates the above flow once and shows the results. “simluateRandomFlow()” doesn’t take any input values, but takes random values for the required fields and generates results. This API is particularly useful for the dynamic update of the Decision Tre Models in the very beginning. It asks for the no. of iterations which you want to run the randomization as an input in the beginning. The Driver class owns an object of the RingerManagerCore class which is responsible for most of the processing and the Decision Tree maintenance and updating.

Conclusion and Future Work:

This was a project simulated in ideal environments and ignores many practical scenarios and issues which will occur if we try to do it in an actual setting. Most of the assumptions that we are making may not be consistent when we will be needing information from the environment. So we need to take care of those details going forward. Also, the latest versions of the web service also has the brightness of the place as an input for the decision process but we have not implemented anything to take that into account while analyzing the Rationales. As of now we do not have any define and unambiguous notion of how ringer manager should behave according to the lighting
conditions of the environment. If we consider that more brightness gives more probability of Ringer Being loud, then this might give a very bad user experience in the settings like Classrooms or Hospitals. However, this assumption may be accurate for settings like Movie Theatre. Hopefully, going forward we will have more understanding of this. However, we do use it for the prediction of the ringer modes as our Decision tree models are capable to adapting itself to dependence of the brightness levels for the predictions purposes. In this version of the implementation we are using weighting of the feedbacks based on the relationships while analyzing the feedbacks. There are many things that are ambiguous in our scenarios, like for example, when updating rationale analyzer module based on neighbors rationales, we don’t have a way of perfectly knowing whether those norms are rational or random. We are using a weight based criteria right now which may not be accurate. Finally, the whole simulation environment behaves irrationally at times as its being simulated in real time environment where all the students are developing and experimenting. We are using those information to adapt our models so at times we hit the “Never Never get here!” parts of the codes! Hopefully, we can have more sophisticated algorithms and a stabler and rational testbed for these decisions in future and hopefully will have better results on this.

**NB:** Kindly look into the codebase properly as it has become pretty huge by now and it is not possible to explain every small detail in this report. The code is very well documented and should not have trouble going through. The read file explains the running instructions elaborately. Kindly contact me for any more information you might need on the implementations and/or the modeling.