Last updated: 15 May 2018

Problem Statement

We need to efficiently support the following query:

> Get **N** unresolved answers submitted to any version of the state **STATE_NAME** of exploration **EXP_ID**.

Definitions and constraints

- It can be assumed that STATE_NAME is a *valid* state name for the latest version of the exploration EXP_ID.
- A **resolved answer** satisfies one of the following conditions:
  - It is matched by a rule of some answer group in the latest version of the exploration.
  - It is part of the training data of some answer group in the latest version of the exploration.
- An **unresolved answer** is an answer that satisfies one of the following conditions:
  - It is not matched by any of the rules and has default outcome or statistical classification.
  - It is not part of training data.
- A **confirmed unclassified answer** is an answer satisfying following condition:
  - It is not matched by any of the rules and has default outcome. Thus, creator expects default outcome for such answers. For e.g. it is a nonsense answer -- such answers would be stored in the confirmed_unclassified_answers field of the state.
- The “any version” part of the problem statement is intended to account for renamings of the state.

What are our raw inputs?

We currently have a StateAnswersModel in the backend, keyed by (exp_id, exp_version, state_name). This model contains a list of all answers submitted to that state for the given version of the exploration.

Current approach

Following is the basic approach:

- Modify InteractionAnswerSummariesMRJobManager and introduce a new type of calculation to find out top N unresolved answers and store the output in StateAnswersCalcOutputModel with exp_version set to ‘all’. This way we can have unresolved answers across multiple exploration versions.

- Use StateAnswersCalcOutputModel to fetch top unresolved answer for a particular <exp_id, exp_version='all', state_name> and transfer them to frontend. The model should store unresolved answers across multiple exp versions for which state_name stays same across all versions.
- ~~In frontend filter out resolved answers from unresolved answer based on current state of the exploration.~~ No need to do this as it might be better to allow a few resolved answers in training data (as discussed in mail thread).

Interaction Answer Summaries MRJob calculation

Following are the list of tasks to be done by new calculation

- The new calculation should iterate through all answers submitted to a particular <exp_id, state_name> and hence calculating unresolved answers across multiple exp versions.
- Then uniquify all the answers so that each answer is present in the list exactly once. For each answer keep its latest resolution detail because when aggregating answers across multiple versions it may happen that an unresolved answer in old version is now resolved in new version and vice-versa.
- From this answers list remove all the answers which are resolved or part of training data or part of confirmed unclassified answers.
- Calculate frequency of each unresolved answer with the help of original list of answers used in step 1.
- Extract top N unresolved answers along with their frequencies from above list and store it in StateAnswersCalcOutputModel. This model should store unresolved answers for each <exp_id, exp_version='all', state_name> (with exp_version='all' it spans across multiple exploration versions). The value of N is fixed and predetermined for all explorations.
- **NOTE:** This job will store unresolved answers only for those states which contains a trainable interaction.
- **NOTE:** When aggregating unresloved answers across multiple states, make sure that answers are aggregated only for those versions which have same interaction id as the most recent exploration version which is included in aggregating answers.
- Output of calculation job:

```
[
  {
    'answer': actual answer,
    'frequency': frequency count of the answer.
  } for answer in answers
]
```

Generating training data

- Creator can fetch unresolved answers for a particular <exploration, state>.

- When fetched in frontend, filter out resolved answers from unresolved answers based on latest state of the exploration.
- Show remaining unresolved answers to the creator.
- To remove an answer from confirmed unclassified answers creator would submit the answer as a new answer in training interface and assign an answer group to it. Note that {training_data + confirmed_unclassified_answers} should be unique and they should not have an overlap in between them. No need for a separate tab of confirmed unclassified answers to remove answers from it.

Future reference

Things to be considered in future iterations.

- Use state_id instead of state_name in calculation models.
- What happens when a new answer comes in (and in so doing, hits a rule, or is an exact match to training data, or is classified statistically, or undergoes default classification)?
- What happens when an exploration is reverted to a previous version?
- What happens when an exploration version is incremented due to an edit?
    - How do we handle a change in an answer group's rules?
    - How do we handle an addition to an answer group's training data?
    - How do we handle a removal from an answer group's training data?
    - How do we handle an answer being moved from one answer group's training data to another's?
    - How do we handle the addition of a new state to an exploration?
    - How do we handle the deletion of a state in an exploration?
    - How do we handle the change of a state's interaction ID?
    - How do we handle changes to a state's non-interactive content, or the outcome of any answer group, or its interaction customization args, or its hints/solutions, or its parameters? (Answer: no-op.)
    - How do we handle an addition to confirmed_unclassified_answers?
    - How do we handle a removal from confirmed_unclassified_answers?
- How do we regenerate, from scratch, the backend models that support this query?

Training Interface

Current training interface

It allows us to do following things:

- Test a new answer. Check what will be its outcome. Allow creator to assign this new answer to some answer group's training data **if** answer has default outcome otherwise (answer is successfully classified using hard rules) it allows us to do nothing.

New training interface

It should be able to do following things **in addition to the abilities of current interface**

- **Show unresolved answers** to creator. Unresolved answers are student submitted answers that have either default outcome or statistical classification (uses ML).
- Ask creator whether she / he is happy with outcome. If she / he is then add answer to training data of that answer group (if answer is classified using statistical classification).
- If answer is classified using default outcome and creator is happy with it then add it to list of "confirmed_unclassified_answers".
- Allows creator to open list of "confirmed_unclassified_answers" and remove answers from there.
- The interface should also allow creator to remove or assign some different answer group to an answer from training of a particular answer group.

Components of system

New calculation

- A new calculation will be added to Interaction Answer Summaries MRJob as described above.

A function to retrieve unresolved answers stored in calculations model in exp service

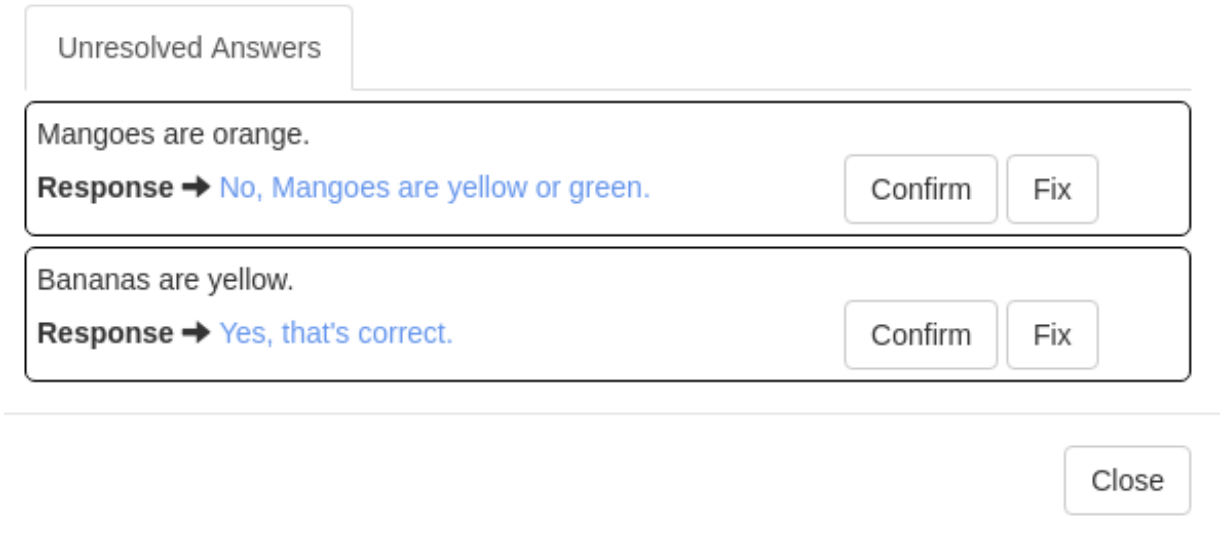
- A new function will be added to exp_services which will take (exp_id, exp_version, state_name) as its input and return the unresolved answers stored in calculations model generated by summarizers job.

A controller to send these unresolved answers to frontend

- GET
    - Get method will take (exp_id, exp_ver, state_name) from input parameters and return the unresolved answers for that state.

Teach Oppia modal

- This modal contains unresolved answers.

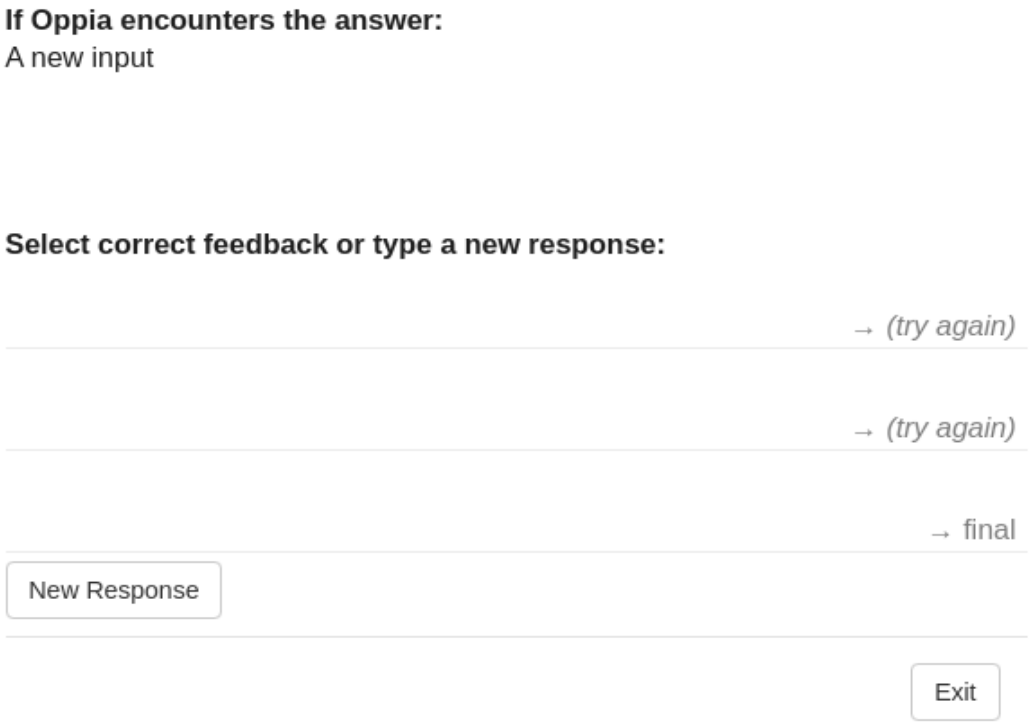


- As shown above unresolved answers tab contains a list of unresolved answers with each answer having the Oppia feedback, confirm button and fix button with it.
- The outcome for each answer will be calculated on the fly by passing each answer through answer classification service.

- The “Confirm” button will add answer to training data of answer group to which answer is classified.
- The “Fix” button will open Answer Group Assignment modal for that answer so that creator can fix answer group to which answer should be classified.

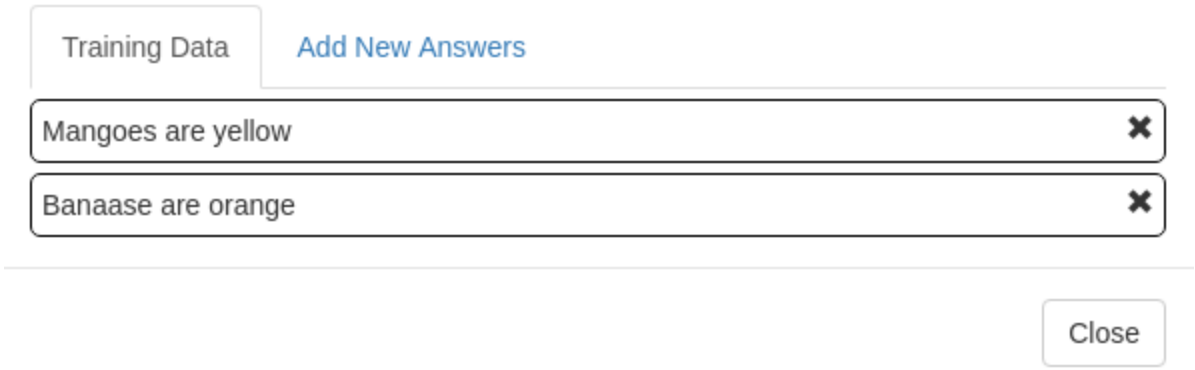
Answer Group Assignment modal

- Answer Group Assignment modal allows creator to assign an answer to answer group to which it will be added.

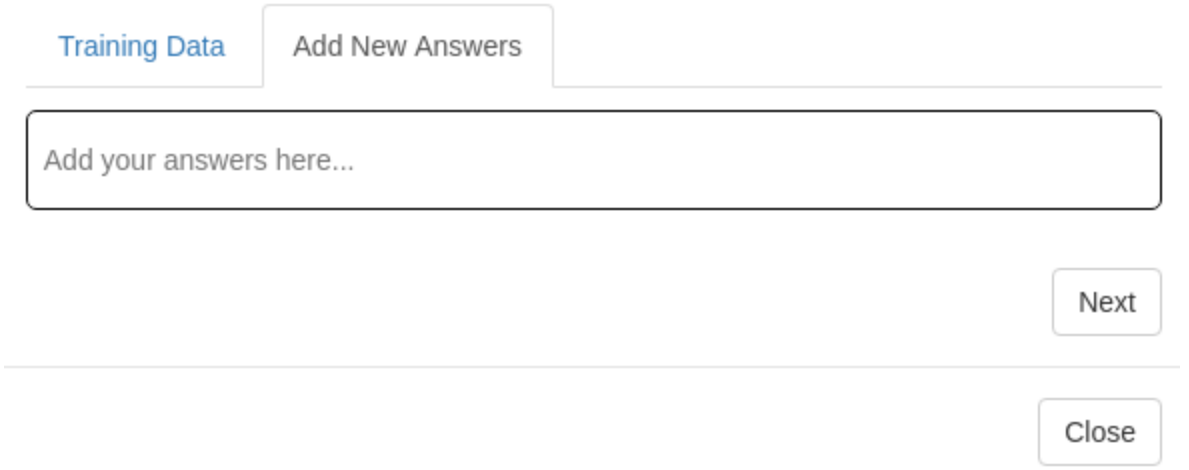


Training data editor modal

- This modal will be used to edit training data of any answer group. The modal will show a list of answers present in training data of answer group.



- Upon clicking on answer Answer Group Assignment modal will be opened for that particular answer.
- Upon clicking on remove icon an answer will be removed from training data. (It won't be added to unresolved answers as unresolved answers are calculated by summarizers job in backend everyday. Thus unresolved answers will be updated daily).
- The model also contains new answers tab.



- The new answers tab allows creator to add answers directly to particular answer group's training data.

Milestones

Milestone 1

- Code
    - Implement new calculations in interaction summarizers MR job.
    - Implement stats_service function to retrieve unresolved answers from backend storage.
    - Implement backend controller to send unresolved answers to frontend.
- Tests
    - Test new calculation output.
    - Test exp_services function and backend controller.

Milestone 2

- Code
    - Implement Teach Oppia modal.
    - Implement Answer Group Assignment modal.
    - Update Training Data Services factory service to properly update answers group's training data upon change.
    - Implement training data editor modal.
- Tests
    - Frontend tests for training data factory service.

Deprecated approach

Following is the approach we were going to use. Keeping it here for reference.

AnswerWithFrequencyModel

We will need a way to store an answer together with the number of times it appears, so that we can prioritize resolution of answers by their frequencies, and also so that we can provide information to creators in the Statistics tab. This motivates the definition of an AnswerWithFrequencyModel keyed by (exp_id, state_id, hashed_answer), with the following fields:

- exp_id (string)
- state_id (string)
- hashed_answer (a string hash of the answer, used for indexing since we can't index on the answer)
- answer (the full answer, of arbitrary type)
- most_recent_exp_version (the most recent version of the exploration for which this answer was submitted. This allows us to deprecate old answers in the future.)
- frequency (integer)
- is_part_of_training_data (boolean)
- latest_resolution_reason (string or None): it is a string representing which classification was used to give responses, i.e. hard rule, default outcome, statistical classification.
- latest_resolution_answer_group_index (Integer or None): index of answer group using which answer was classified.
- latest_resolution_rule_index (Integer or None): index of a rule within answer group using which answer was classified.
- expected_resolution_reason (string or None): this represents which classification is expected in response to answer. This field is only valid if answer is part of training data or confirmed unclassified answer.

See definitions given above. Following details gives information about how to distinguish between resolved, unresolved and confirmed unresolved answer with the help of above model.

Resolved answers

- Answers which are classified by hard rules (represented by "latest_resolution_reason" field) or are of training data (represented by "is_part_of_training_data" field).

Unresolved answers

- Answers which have default outcome or statistical classification as their latest resolution reason and are not part of training data.

Confirmed unresolved answers

- Answers which are part of training data and have default outcome as their **expected resolution reason.**

Generation of AnswerWithFrequencyModel for all existing exploration in Oppia
This model is populated by a one-off job that iterates along all existing answers in StateAnswersModel.

Regeneration: Simulate the "update_or_store_new_answer" flow below, iterating along the StateAnswersModel entities for each version of each exploration.

Handling collisions

- To handle the collisions, we will be using master with children model. In this scheme, there is one master model which stores one answer and along with this answer it also stores references to the child models which store other answers. All the answers (master as well as children) have same hash value and results in collision.
- For reference, see the implementation of "stats_models.StateAnswersModel".

Handling changes in the exploration

This is part of frontend training interface UX. Hence it should be taken care of with implementation of frontend interface.

There are four kinds of changes that could affect resolved / unresolved / confirmed unclassified answers:

1. Change in interaction id of the state.
2. Addition of answer group / rule.
3. Removal of answer group / rule.
4. Modification of answer group / rule.

Due to this, the following changes can occur:

1. All the previous stored answers for that particular state may become invalid. This can happen when creator modifies interaction id.
2. Previously resolved answers may become unresolved. This can happen if creator deletes / modifies answer group or its rules.
3. Previously unresolved answers and / or "confirmed_unclassified_answers" may become resolved due to addition / modification of answer group / rules.

- **Note**: Proper data structure (lets call this 'answers_data') will be required to handle all these changes atomically and consistently. Basically this data structure keeps track of all the changes that are happening to various answers across above three changes.

Handling change in interaction id

- When creator clicks on the button to fetch unresolved answer to create training data check the current interaction id. If it is different than the interaction id of the state at the beginning then all of the previous answers are invalid for this new state.
- Basically this should be done for all the conditions which result in some significant change and requires change in state id. As of now, we only consider change in interaction id to be significant enough for change in state id of that particular state.

Handling addition of an answer group / rule

- When a new answer group / rule is added, re-classify all the unresolved answers / confirmed_unclassified_answers again and see if some of them become resolved. Reclassification should be done only for answers which are cached in frontend. Answers will be cached in frontend when creator has previously opened training interface. **Note**: confirmed unclassified answers are stored with exploration state itself, thus they will always be cached in frontend, regardless whether creator modifies training data and / or confirmed unclassified answers.
- If they do then store such answers appropriately in ‘answers_data’ so that their backend AnswerWithFrequencyModels can be updated at the time of exploration publishing.

Handling removal of an answer group / rule

- Keep track of changes happening in answer group and its rules. Whenever an answer group or one of its rules is removed, fetch all answers that were classified by that answer group (or classified to that rule, in case when only a rule of answer group has been removed.)
- Once all answers have been fetched re-classify them through hard rules (AnswerClassificationService) so that we have new resolution for each of these answers. If an answer becomes unresolved then add it to the list of unresolved answers. If it stays resolved then add the answer to appropriate (answer group, rule to which **now** they classify) in ‘answers_data’.
- Keep these changes in cache (answers_data) so that their respective AnswerWithFrequencyModels can be updated at the time of exploration publishing.
- When creator publishes exploration, propagate these changes to the backend along with changes made in exploration and store them in AnswerWithFrequency models.

Handling changes in an answer group / rule

- If an answer group or one of its rule changes then perform handling of removal an answer group / rule (for updating resolved answers) and then perform handling of addition of an answer group / rule scenario (for updating unresolved answers / confirmed unclassified answers).

Fetching unresolved answers

- When creator clicks on the button to create new training data, fetch the answers page by page and before showing them to creator pass them through answer classification to re-classify them.
- If an answer becomes resolved then add this answer to resolved answers list (answer group, rule) to which it is classified in ‘answers_data’.
- Show all the unresolved answers to the creator (page by page).
- **Note**: this method performs lazy updation of unresolved answers because answers are only updated (when answer changes, unresolved → resolved) when creator fetched them in frontend.

Handling changes in training data

- Whenever an answer is added in training data it changes from unresolved answer to resolved answer.
- Changes in training data also impact AnswerWithFrequency models. To handle this changes we can compare previous training data and current training data.
- From this comparison we can derive 3 lists: new added answers, removed answers, modified answers (answer transferred from training data of one answer group to other)
- This, then, can be modified in AnswerWithFrequency model by doing necessary changes in “is_part_of_training_data”, “expected_resolution_reason” and “expected_resolution_details”.

Handling changes in confirmed unclassified answers

Addition of an answer

- Upon adding answer, set ‘is_part_of_training_data’ to True and change expected_resolution_reason to default outcome. The answer changes from unresolved answer to confirmed unclassified answer.

Removal of an answer

- Upon adding answer, set ‘is_part_of_training_data’ to False. Set expected_resolution_reason to None. The answer changes from confirmed unclassified answer to unresolved answer.

Required components to be implemented

Datastore models

- AnswerWithFrequencyModel

    - Implement this model as shown above.

MapReduce Jobs

- MR job to populate AnswerWithFrequenyModel for current explorations.
    - This job should iterate through all (exploration, version, submittedAnswers) and populate AnswerWithFrequencyModel for each submitted answer.

Methods

- get_top_n_unresolved_answers(exploration_id, state_name, cursor=None):
    - Find the state ID corresponding to (exp_id, latest exp_version, state_name)
    - Query AnswerWithFrequencyModel to retrieve N instances with the given exp_id, state_id, and a latest_resolution_reason of either None or ‘statistical’ and ‘is_part_of_training_data’ is False. (We could declare a composite field for these conditions, and update it on-put, in order to make the query a simple one with just one clause.)
    - Use Query cursor and retrieve answers page by page.

- update_or_store_new_answer(exploration_id, state_name, answer, exp_version):
    - Find the state ID corresponding to (exp_id, exp_version, state_name)
    - Find the AnswerWithFrequencyModel corresponding to (exp_id, state_id, hashed_answer) and update its count, most_recent_exp_version, frequency, latest_resolution_reason and latest_resolution_details.
    - If the model already exists then check
        - if (exp_version >= model.most_recent_exp_version). If it is then increment frequency count and store latest resolution reason and details.
        - If (exp_version < model.most_recent_exp_version) then increment frequency but do not change count.

Training interface UX components

These components should be implemented alongside training interface UI.

Backend methods

- update_state_answers_upon_exploration_publish(exploratio_id, answers_data)
    - ‘answers_data’ will be the updated data received from the frontend.
    - Use this data and update corresponding models in AnswerWithFrquencyModel.
    - Design appropriate data structure (answer_data) for efficient storing and retrieving of answers. Basically “answers_data” should store each answer with its new resolution details due to changes in exploration state answer groups.

- update_training_data_answers(exploration_id, exp_version, state_name, training_data):
    - Fetch previous version's training data and compare it with new training_data.
    - Generate 3 lists new_answers, removed_answers, changed_answers.
    - Find state_id corresponding to (exploration_id, state_name).
    - For each answer in new_answers, fetch their respective AnswerWithFrequencyModels. Set 'is_part_of_training_data' to True. Change expected resolution reason to statistical classification and other expected resolution details accordingly.
    - For each answer in removed_answers, fetch their respective AnswerWithFrequencyModels. Set 'is_part_of_training_data' to False. Change expected resolution reason and expected resolution details to None.
    - For each answer in changed_answers, fetch their respective AnswerWithFrequencyModels. Change expected resolution reason and expected resolution details accordingly.

- get_answers_classified_using_given_rule(exploration_id, state_name, answer_group_index, rule_index):
    - Retrieve state_id corresponding to (exploration_id, state_name).
    - Use Query to retrieve all answers that have given exploration_id, state_id, latest_resolution_reason=hard rule classification and has matching answer_group_index and rule_index.
    - If rule_index is None then fetch all answers classified using given answer_group_index.
    - Return list of all such answers.

- update_confirmed_unclassified_answers(exploration_id, state_name, confirmed_unclassified_answers)
    - Fetch previous confirmed_unclassified_answers and compare it with new list. Confirmed unclassified answers are stored as an attribute of exploration state.
    - Create two lists from this new_added_answers, removed_answers.
    - For each answer in new_added_answers perform addition task.
    - For each answer in removed_answers perform removal task.

Controllers

- /get_top_unresolved_answers
    - Returns unresolved answers for given (exploration_id, state_name).

- /get_answers_classified_using_given_rule
    - Returns resolved answers for (exploration_id, state_name) which were classified using given (answer_group_index, rule_index).

Frontend methods

- handle_changes_in_answer_group()
    - Fetch all answers from backend that were classified using given answer_group_index, rule_index.
    - Run all the fetched answers through AnswerClassificationService.
    - Store all the newly classified answers appropriately in ‘answers_data’.

- fetch_unresolved_answers()
    - Retrieve all unresolved answers from ‘answers_data’.
    - If answers are less than page_size then fetch more unresolved answers from backend page by page.
    - Run all the newly fetched unresolved answers through AnswerClassificationService and keep only those which remain unresolved.
    - All the answers which are resolved now, add them to ‘answers_data’ appropriately.
    - Return page_size number of answers.

- add_to_confirmed_unclassified_answers(answer):
    - Add the answer to confirmed_unclassified_answers.

- get_confirmed_unclassified_answers():
    - Confirmed unclassified answers will be stored as an attribute of exploration state. Thus it will be transmitted to frontend along with all exploration data.
    - Return the list of confirmed unclassified answers retrieved from state’s attribute.

- remove_from_confirmed_unclassified_answers():
    - Remove particular answer from confirmed_unclassified_answers list and add it as unresolved answers in ‘answer_data’.

When a new exploration version is committed (**TODO**):

- How do we handle a change in an answer group’s rules?(Answer: see above)
- How do we handle an addition to an answer group’s training data? (Answer: see above)
- How do we handle a removal from an answer group’s training data? (Answer: see above)
- How do we handle an answer being moved from one answer group’s training data to another’s? (Answer: see above)
- How do we handle the addition of a new state to an exploration? (Answer: assign state id to new state)
- How do we handle the deletion of a state in an exploration? (Answer: no-op)
- How do we handle the change of a state’s interaction ID? (Answer: assign new state id)

- How do we handle changes to a state's non-interactive content, or the outcome of any answer group, or its interaction customization args, or its hints/solutions, or its parameters? (Answer: no-op.)
- How do we handle an addition to confirmed_unclassified_answers? (Answer: see above)
- How do we handle a removal from confirmed_unclassified_answers? (Answer: see above)

Expected workflow

Earlier work (already done)

StateIdMappingModel (finished)

In order to handle state renames, we will need a way to keep track of state IDs across different versions of an exploration. This leads to a StateIdMappingModel keyed by (exp_id, exp_version). It is created each time a new exploration version is committed, and contains the following properties:

- exp_id (string)
- exp_version (integer)
- state_names_to_ids (mapping of state names to IDs for this article version of the exploration)
- latest_state_id_used (integer)

IDs are non-negative integers that are assigned starting from 0 and counting upwards.

Regeneration: This model can be generated for all existing explorations by starting at version 1 of the exploration and mapping each state name to a distinct ID, increasing the counter accordingly. When a new version of the exploration is committed, the changelist is analyzed and new IDs are assigned to newly-added states, and IDs are preserved for states that have been renamed.

- Note: If an interaction ID has changed, then we basically have a new state: old answers are no longer valid. We need to account for this. The simplest way is probably to just assign the state a new ID.
- **Risk:** If a data inconsistency happens, or we change the definition of “state identity”, this model can’t be regenerated because we’re referring to these IDs elsewhere. For example, if we later decide that an edit to the non-interactive content in a long-ago version is a major one and we want to stop answers from before that edit showing up, the state IDs will all need to be updated, but this has knock-on effects (to both the current exploration and also to any other data we store that uses the state IDs).
    - **Resolution:** We could, if such circumstances occur, define a new field (e.g. state_id_new) and repopulate that, and also repopulate derived model data based on the new state IDs. However, the above risk suggests that we should not use these state IDs in any canonical versions of data -- they should only be referenced in derived, computed objects. We should therefore add docstrings in the state ID mapping model and domain objects saying: “only use this for derived data, but not for data that’s regarded as the source of truth”.

This model supports the following methods:

- get_id(state_name): returns the state ID corresponding to the state name of this version of the exploration, for correlation with states in other versions of the same exploration.

Components for StateIdMappingModel

exp_domian.StateNameToIdMap class

Attributes:

- State_name_to_id_map: dict. A dictionary that maps each state_name to unique state_id.

Methods:

1. get_state_name_to_id_mapping(cls, old_exp, new_exp, cmd_list) (@classmethod)

- Construct state_id to state_name mapping for old_exp. Initialize new_state_names to [].
- Go through change list one by one:
    - If CMD is to add a new state then add state_name to new_state_names list.
    - If CMD is to remove a state then check the id_to_name mapping dict and find a key whose value is the removed state name. If such pair is found then remove it from id_to_name mapping. If it is not found then look for the removed state's name in new_state_names list and remove it from there.
    - If CMD is to rename a state then check the id_to_name mapping dict and find a key whose value is the old name of renamed state. If such pair is found then change the value to new name of state. If it is not found then find the renamed state's old name in new_state_names list, remove it from there, and add new name of renamed state in list.
    - If CMD is to revert exploration then get the version_number from CMD and obtain StateIdMappingModel for (exp_id, version_number). Call cls.get_state_id_mapping_from_model(StateIdMappingModel instance) and **return** the result of called method.
- Go through each key value (id, name) pair in id_to_name mapping. Create a list of modified_state_names as follows:
    - Find the state that has "id" as state ID in old exploration.
    - Find the state that has "name" as state name in new exploration.
    - Check for following differences and if any one of the following is True then treat the state as a new state and add it to list of modified_state_names:
        - Changes in interaction_id (we can add more checks here in future)
- Add all state_names in modified_state_names to new_state_names.
- Reverse id_to_name mapping to name_to_id mapping.
- Add a name in name_to_id mapping for each name in new_state_names with new state IDs as their values.
- Generate StateNameToIdMap domain object using name_to_id dict, validate it and return it.

2. validate()

- Check that state_id assigned to each state is unique and is instance of integer type and is less than or equal to the latest_state_id_used.

3. get_id_for_state_name(state_name)

- Return unique ID assigned to this state_name.

4. get_state_name_to_id_mapping_for_new_exploration(cls, Exploration object) (@classmethod)

- Go through each state in exploration and assign ids starting with 0.
- Create StateNameToIdMap object, validate it and return it.

5. get_state_name_to_id_mapping_from_model(cls, StateIdMappingModel object) (@classmthod)

- Create new StateNameToIdMap domain object and copy StateIdMappingModel.state_names_to_ids to state_name_to_id_map attribute of new object.
- Call validate() method on new object and return it.

exp_domain.Exploration class

Methods:

1. store_state_id_mapping_model(change_list)

- If (exp_version > 1) then
    - Retrieve previous version of exploration.
    - Call exp_domain.StateNameToIdMap.get_state_name_to_id_mapping(old_exp, new_exp (self), change_list) and obtain StateNameToIdMap object.
- If (exp_version == 1) then
    - exp_domain.StateNameToIdMap.get_state_name_to_id_mapping_for_new_exploration(cls, Exploration object) and obtain StateNameToIdMap object.
- Store StateIdMappingModel instance for this exploration version with the help of IDs assigned in StateNameToIdMap object.

Mapreduce job for generation of StateIdMappingModel instance for current exploration database:

- Iterate over all the entities of exploration model (ExplorationModel). This will iterate over latest versions of all explorations.
- Map function
    - Do a get_multi() and retrieve all versions of the current exploration. Also do a get_multi() and retrieve change list for all versions of exploration.

    - Start with version 1 and go upto latest version and create StateIdMappingModel instance for all versions. That is for each version call store_state_id_mapping_model with relevant change list.
- Reduce function
    - Do nothing.